António Joaquim André Esteves

Uma Metodologia de Partição para o Co-projecto de Sistemas Digitais Embebidos

Tese submetida à Escola de Engenharia da Universidade do Minho
para a obtenção do grau de Doutor em Informática
(Área de Especialização em Engenharia de Computadores)

Universidade do Minho

Escola de Engenharia

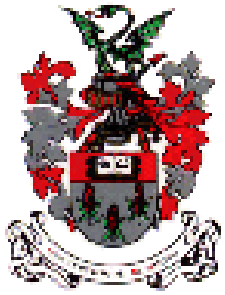
Departamento de Informática

Braga — Julho 2001

António Joaquim André Esteves

Uma Metodologia de Partição para o Co-projecto de Sistemas Digitais Embebidos

Tese submetida à Escola de Engenharia da Universidade do Minho
para a obtenção do grau de Doutor em Informática
(Área de Especialização em Engenharia de Computadores)



Universidade do Minho

Escola de Engenharia

Departamento de Informática

Braga — Julho 2001

António Joaquim André Esteves

Uma Metodologia de Partição para o Co-projecto de Sistemas Digitais Embebidos

Tese submetida à Escola de Engenharia da Universidade do Minho
para a obtenção do grau de Doutor em Informática,
Área de Especialização em Engenharia de Computadores

Dissertação realizada sob a orientação de
Prof. Doutor Alberto José Gonçalves de Carvalho Proença,
Professor Catedrático do Departamento de Informática da
Escola de Engenharia da Universidade do Minho
e de
Prof. Doutor Henrique Manuel Dinis dos Santos,
Professor Associado do Departamento de Sistemas de Informação da
Escola de Engenharia da Universidade do Minho

Universidade do Minho

Escola de Engenharia

Departamento de Informática

Braga — Julho 2001

Título: Uma Metodologia de Partição para o Co-projecto de Sistemas Digitais Embebidos

Autor: António Joaquim André Esteves

Tese de Doutoramento em Informática, Especialidade em Engenharia de Computadores, Departamento de Informática, Escola de Engenharia, Universidade do Minho

Candidatura a Doutoramento aceite pelo Conselho Científico da Escola de Engenharia da Universidade do Minho em 08 de Fevereiro de 1995

Orientadores: Alberto José Gonçalves de Carvalho Proença e
Henrique Manuel Dinis dos Santos

Conclusão: Julho de 2001

Aos meus pais

"*Saúdo todos os que me lerem,*
Tirando-lhes o chapéu largo
Quando me vêem à minha porta
Mal a diligência levanta no cimo do outeiro.
Saúdo-os e desejo-lhes sol,
E chuva, quando a chuva é precisa,
E que as suas casas tenham
Ao pé duma janela aberta
Uma cadeira predilecta
Onde se sentem, lendo os meus versos.
E ao lerem os meus versos pensem
Que sou qualquer cousa natural –
Por exemplo, a árvore antiga
À sombra da qual quando crianças
Se sentavam com um baque, cansados de brincar,
E limpavam o suor da testa quente
Com a manga do bibe riscado."

O Guardador de Rebanhos, em
Poemas Completos de Alberto Caeiro.

Prefácio

Embora a caracterização mais imediata que ocorre quando se pensa num sistema embebido seja a dum sistema que exerce funções de controlo, a sub-classe dos sistemas embebidos que executam maioritariamente tarefas de processamento de dados é cada vez mais frequente. Os sistemas embebidos predominantemente de fluxo de dados surgem em áreas como o multimédia, a electrónica de consumo ou as comunicações. Em muitas aplicações, o projecto destes sistemas constitui um desafio interessante: embora a complexidade e os requisitos tendam a aumentar, para responder às exigências do mercado, espera-se que o tempo (logo o custo) de projecto se mantenha o menor possível para que o produto final seja competitivo. O desafio é ainda maior se o tipo de sistema em causa tiver um tempo de vida reduzido.

Uma solução para este problema consiste em implementar os sistemas em plataformas híbridas, i.e. com *hardware* e *software*, projectando de forma unificada as diferentes partes que constituem a implementação – paradigma do co-projecto de *hardware* e de *software*. O *software* consegue implementar sistemas cada vez mais complexos, o tempo de projecto para uma implementação em *software* é normalmente inferior ao tempo de projecto para *hardware* e o *software* garante uma maior longevidade aos sistemas, ao permitir e facilitar a sua reformulação. O *hardware* permite atingir mais facilmente os requisitos de desempenho e de interacção com o sistema envolvente.

Para implementar um sistema em *hardware* e *software*, a partir duma descrição unificada, é necessário reformular o comportamento (preferencialmente) ou a estrutura do sistema, decompondo-o(a) em partes. A partição da descrição dum sistema em partes, uma tarefa essencial a uma metodologia que aplique o paradigma do co-projecto de *hardware* e de *software*, constitui o tema central desta tese.

Outro factor que releva o paradigma de co-projecto de *hardware* e de *software* é a constatação de que, quando se trata de projectar sistemas para uma implementação com um único componente, as actuais ferramentas de ECAD deixam pouco espaço para a investigação académica fora do seu "raio de acção". O mesmo não se passa quando a implementação se faz com múltiplos componentes e com uma arquitectura alvo heterogénea, uma área que recomenda a aplicação do paradigma de co-projecto de *hardware* e de *software* e é menos coberta pelas ferramentas de ECAD comerciais.

Agradecimentos

Apesar de uma tese de doutoramento ser um trabalho essencialmente individual, o contributo de outras pessoas além de inevitável é salutar. Esta tese não foge à regra e por isso é meu dever agradecer merecidamente às pessoas e entidades que mais directamente estiveram envolvidas nesta tarefa não pequena.

Começo por agradecer ao Professor Doutor Alberto José Proença, meu orientador científico, pelo seu contributo para as grandes decisões que foi necessário tomar, pelo sentido crítico que sempre me incutiu, o qual contribui decididamente para a qualidade do trabalho desenvolvido e pelo empenhamento colocado na melhoria deste documento. O meu maior agradecimento é-lhe devido, pois foi a pessoa que mais me apoiou.

Agradeço ao Professor Doutor Henrique Dinis Santos, meu co-orientador, pela ajuda prestada em áreas específicas da tese e pela visão pragmática que me transmitiu, bem necessária para levar este trabalho a "bom porto".

Ao Sr. Jaime Ferreira Gomes, técnico do Departamento de Informática da Universidade do Minho, pelo envolvimento imprescindível no desenvolvimento da plataforma EDgAR-2, utilizada na implementação dos casos de estudo. Estou-lhe grato ainda pelos constantes incentivos para concluir esta tese com sucesso e celeridade.

Ao colega Professor Doutor João Miguel Fernandes, pelo esforço de revisão de partes da tese e pelas inúmeras discussões de índole técnica e estratégica que tivemos, tão importantes num meio em que o número de pessoas com os mesmos interesses científicos é diminuto.

Aos restantes membros do Grupo de Engenharia de Computadores, Professor Doutor António Manuel Pina, Professor Doutor João Luís Sobral e Professor Doutor Luís Paulo Santos, por terem acompanhado a evolução da minha tese, participando em discussões informais e partilhando os problemas que surgiram no decorrer da preparação do doutoramento.

Aos Docentes, Investigadores, Técnicos e Funcionários do Departamento de Informática da Universidade do Minho, pela compreensão e incentivo.

À Universidade do Minho, à Escola de Engenharia e ao Departamento de Informática, por terem permitido e apoiado institucionalmente a concretização desta tese.

À Xilinx por ter doado a solução para um dos módulos da plataforma EDgAR-2, sem a qual o tempo de projecto teria sido muito superior.

Ao Max-Planck-Institut für Informatik de Saarbrücken, Alemanha, que desenvolve o projecto LEDA e permitiu que o utilizasse livremente. Foi sem sombra de dúvida uma preciosa ajuda para a implementação da ferramenta *ParTiTool*.

Ao aluno César Gonçalves e aos alunos da disciplina de Síntese de Sistemas Digitais II da LESI, pelo envolvimento em projectos relacionados com este trabalho.

Por último, agradeço com grande afecto aos meus pais e irmã o amor e solidariedade que sempre me dispensaram. Por me terem proporcionado aquilo que considero serem os alicerces da vida, educação e valores, a minha incomensurável gratidão.

Braga, 2 de Julho de 2001

António Joaquim André Esteves

Resumo

Esta tese propõe uma metodologia de partição automática, que é parte integrante duma metodologia mais abrangente orientada para o desenvolvimento de sistemas embebidos predominantemente de fluxo de dados. É formulado o problema de partição, tarefa que depende da arquitectura alvo e que se integra na fase de implementação dos sistemas com múltiplos componentes de *hardware* e/ou de *software*.

A partir do levantamento sobre os meta-modelos mais utilizados em sistemas embebidos, justifica-se a selecção de PSM para representação de interface entre a tarefa de partição e a restante metodologia de desenvolvimento. Para representar os sistemas internamente ao processo de partição desenvolveu-se o meta-modelo PSMfg.

Apresenta-se a arquitectura alvo actualmente suportada pela metodologia de partição proposta, a qual é composta por uma plataforma EDgAR-2 ligada a um sistema hospedeiro.

A organização da metodologia de partição desenvolvida inclui um algoritmo de partição construtivo e a respectiva função de proximidade, um algoritmo de partição iterativo e a correspondente função de custo e os estimadores de métricas. O algoritmo construtivo gera uma solução de partição inicial, que o algoritmo iterativo procura melhorar em sucessivas tentativas.

Mostra-se como é que, partindo dum conjunto de algoritmos de partição referenciados na bibliografia, se chegou à selecção do algoritmo construtivo de crescimento de grupos e do algoritmo iterativo de pesquisa tabu.

Descreve-se o processo de construção de soluções de partição com o algoritmo construtivo de crescimento de grupos, a função de proximidade por ele utilizada e a estimação das métricas necessárias a esta função.

A adaptação do algoritmo de pesquisa tabu ao presente problema de partição é explicada, sendo de destacar a implementação dos tipos de tabu, da lista tabu, do historial de soluções visitadas, da pesquisa na subvizinhança, dos critérios de aspiração e dos mecanismos de convergência para a solução óptima e de fuga a mínimos locais. A função de custo utilizada pelo algoritmo de pesquisa tabu é definida a partir dum conjunto de métricas e dos condicionalismos ou requisitos que lhe estão associados.

Para estimar as métricas da função de custo definiram-se modelos de *software* para o processador, de *hardware* para as FPGAs e os CPLDs e de comunicação para os recursos de interligação. Para reduzir o tempo de cálculo e manter uma precisão elevada nas estimativas, a estimação é incremental e funciona em dois níveis de abstracção: (i) no nível de abstracção mais elevado, obtêm-se estimativas para o espaço ocupado pelo caminho de dados e pela unidade de controlo das partições de *hardware* e para o desempenho do sistema e (ii) no nível de abstracção mais baixo, obtêm-se estimativas para métricas relativas aos objectos do modelo do sistema.

A validação da metodologia de partição traduziu-se no estudo de dois exemplos: a convolução duma imagem e um algoritmo de criptografia. Os aspectos avaliados foram a qualidade das soluções de partição geradas automaticamente, a precisão e a fidelidade das estimativas, o desempenho da ferramenta desenvolvida e o apoio que presta à implementação das soluções de partição.

Palavras-chave: metodologia de partição, estimação de métricas, co-projecto de *hardware* e de *software*, sistemas digitais embebidos, algoritmo de pesquisa tabu, algoritmo de crescimento de grupos, meta-modelo PSM, meta-modelo PSMfg, FPGA, CPLD, arquitectura alvo reconfigurável.

Abstract

This thesis presents an automatic partitioning methodology included in a broader methodology oriented to the development of data flow dominated embedded systems. It is described the partitioning problem, which depends on the target architecture and is considered a crucial task to implement a system with software and several hardware components.

The thesis starts with a survey of the more relevant meta-models used with embedded systems, to justify the option for PSM to describe the systems at the interface between the partitioning task and the remaining development methodology. A new meta-model, PSMfg, was developed as an internal way to describe the systems during the partitioning process.

A target architecture is presented, based on two types of reconfigurable devices (FPGAs and CPLDs): the EDgAR-2. The prototype system contains an EDgAR-2 platform connected to a PC host system. EDgAR-2 was built to support the proposed partitioning methodology.

The developed partitioning methodology includes a constructive partitioning algorithm and its closeness function, an iterative partitioning algorithm and its cost function and the metrics estimators. The constructive partitioning algorithm generates an initial solution that iterative algorithm tries to successively improve.

A literature review on partitioning algorithms suggested the adoption of the cluster growth constructive algorithm and the *tabu* search iterative algorithm, as the base foundation for the proposed methodology.

The process of creating partitioning solutions with cluster growth constructive algorithm, the applied closeness function and the estimation of the metrics required by this function are presented.

Tailoring the *tabu* search algorithm to the presented partitioning problem is detailed, with an emphasis on the implementation of the *tabus*, the *tabu* list, the visited solutions memory, the partial neighbourhood search, the aspiration criteria and the techniques that reinforce the convergence to the optimum solution and the escape from local minima of the cost function. The cost function applied by the *tabu* search algorithm depends on a set of metrics and on the constraints or requirements associated with these metrics.

To estimate the metrics whose cost function depends on, a software model for the processor, a hardware model for the FPGAs and CPLDs and a communication model for the interconnection resources were defined. To reduce the estimation time while providing a high degree of precision, the estimation process is incremental and it is performed at two abstraction levels: (i) at the higher abstraction level, one estimates the hardware partitions data path and control unit area and the system performance and (ii) at the lower abstraction level, one estimates values for the system model objects metrics.

The validation of the partitioning methodology was carried out through two examples: the convolution of an image and a cryptography algorithm. The quality of the automatically generated partitioning solutions, the precision and fidelity of the estimations, the performance of the developed tool and its support to implement the partitioning solutions were evaluated with promising results.

Keywords: partitioning methodology, metrics estimation, hardware/software co-design, digital embedded systems, *tabu* search algorithm, cluster growth algorithm, PSM meta-model, PSMfg meta-model, FPGA, CPLD, reconfigurable target architecture.

Conteúdo

Lista de Figuras

Lista de Tabelas

Parte I

Enquadramento do Trabalho

Capítulo 1

Introdução

Conteúdo

O desenvolvimento dum sistema decorre em quatro grandes etapas: a análise, a concepção, a implementação e o teste. Ao conjunto de tarefas utilizadas no projecto dum sistema, juntamente com a ordem pela qual se realizam as tarefas mais o conjunto de ferramentas CAD e notações que suportam cada tarefa, atribui-se o nome de metodologia de desenvolvimento. Nos últimos anos as metodologias do tipo *bottom-up*, ou *capturar-e-simular*, têm dado lugar a metodologias do tipo *top-down*, ou *descrever-e-sintetizar*, em que se descrevem os sistemas de forma comportamental e se obtém a sua estrutura através duma ferramenta de síntese automática. Nas metodologias do tipo *descrever-e-sintetizar* a descrição inicial dum sistema não está orientada nem para uma determinada tecnologia de implementação nem para um comportamento temporal definido com precisão *a priori*. Nestas metodologias o grande esforço do projectista reside na obtenção de uma descrição correcta do comportamento do sistema a implementar, deixando uma boa parte das tarefas necessárias à obtenção da melhor solução de implementação para as ferramentas de CAD. Deste modo, conseguem-se ganhos de produtividade significativos relativamente às metodologias do tipo *capturar-e-simular*. As metodologias do tipo *descrever-e-sintetizar* serão ainda mais vantajosas quando aplicáveis no desenvolvimento de sistemas compostos por *hardware* e *software*. Nestes casos, para diminuir o tempo de projecto convém que a tradicional análise *ad-hoc*, usada para obter a descrição dos sistemas, seja substituída pela obtenção duma descrição executável na fase inicial do desenvolvimento desses sistemas. A descrição executável dum sistema, além de representar a sua funcionalidade, permite validá-la antes de se obter uma implementação.

1.1 Partição em Componentes Diferenciados

O presente trabalho incidiu sobre a fase de implementação dos sistemas e, em especial, no desenvolvimento de uma metodologia de partição automática. Esta metodologia inclui-se numa outra mais abrangente que cobre todas as fases do desenvolvimento dos sistemas (figura 1.1), em que a fase de análise é tratada em [Fer00] e a fase de concepção em [Mac00].

A partição destina-se ao projecto de sistemas embebidos, predominantemente de fluxo de dados, de média complexidade e que podem apresentar requisitos de tempo real, em que um computador e uma plataforma EDgAR-2 constituem a arquitectura alvo reconfigurável a usar na implementação. Como a implementação dos sistemas é feita com componentes ditos de *hardware* e de *software*, o presente trabalho aplica o paradigma do co-projecto de *hardware* e de *software*.

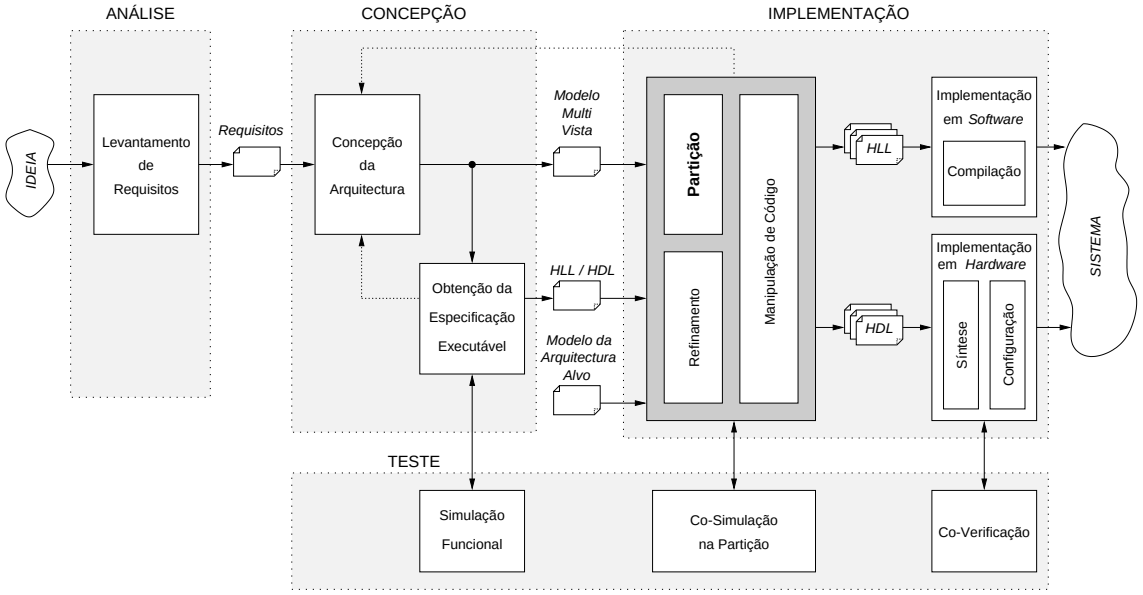


Figura 1.1: A metodologia de desenvolvimento em que se insere a partição.

Dado que a arquitectura alvo inclui componentes, como o processador e a memória, que implementam a parte de *software* dos sistemas e componentes, como FPGAs e CPLDs[1], que implementam a parte de *hardware*[2], o problema de partição em causa deixa de ser apenas uma partição entre *hardware* e *software* para passar a ser um problema mais rico de partição entre múltiplos componentes diferenciados.

A partição constitui um problema de optimização NP-completo que procura obter a distribuição óptima dos objectos dum sistema pelos diversos componentes da arquitectura disponível para a sua implementação. A definição de óptimo resulta duma função de custo definida pelo

[1] ***F**ield **P**rogrammable **G**ate **A**rrays* e ***C**omplex **P**rogrammable **L**ogic **D**evices*, na terminologia inglesa.

[2] Daqui para a frente, a expressão "componente de *software* (*hardware*)" deve ser entendida como "componente que implementa a parte de *software* (*hardware*) do sistema".

projectista. Deste modo, o processo de partição converte a representação unificada e não comprometida dum sistema, numa representação composta por várias partes comprometidas com os componentes da arquitectura alvo. Cada parte que compõe a melhor solução gerada pelo processo de partição iterativo é depois refinada manualmente com detalhes de implementação e convertida para uma linguagem HLL ou HDL[3] que pode ser sintetizada para o tipo de componente alvo a que essa parte foi atribuída. Em cada iteração do processo de partição estimam-se métricas como o espaço requerido pelos recursos e o tempo de execução que, de acordo com o conjunto de requisitos e condicionalismos impostos ao sistema, permitem medir o grau de optimização da solução actual.

1.2 Co-projecto de *Hardware* e de *Software*

O paradigma do co-projecto de *hardware* e de *software* designa uma forma unificada, ou cooperativa, de projectar as partes de *hardware* e de *software* dum sistema. A forma natural de implementação dum sistema assim projectado é uma arquitectura heterogénea que contenha componentes ditos de *hardware* e de *software*. Utilizando este paradigma, o projectista pretende (i) aumentar a fiabilidade do produto final, porque se projecta e testa em simultâneo todo o sistema, em vez de se conceber em separado cada componente da implementação, (ii) aumentar a flexibilidade dos sistemas, de modo a permitir a sua reformulação ou a experimentação de várias alternativas de implementação e (iii) reduzir o tempo e o custo de projecto, por forma a que o produto final se apresente ao consumidor com o melhor preço possível. Os componentes da arquitectura alvo responsáveis pela execução da parte de *software* dos sistemas, como os processadores ou a memória RAM, têm por objectivo permitir implementações com custo inferior, mais flexíveis e com um tempo de projecto menor; os componentes responsáveis pela implementação da parte de *hardware*, como os ASICs ou os PLDs, permitem atingir o desempenho exigido ao sistema, mas com um custo e tempo de projecto superior.

Idealmente, as abordagens que utilizam o paradigma do co-projecto de *hardware* e de *software* partem duma descrição única para todo o sistema. Esta descrição pode resultar de uma série de transformações com regras estabelecidas, que vão desde a análise até à concepção duma arquitectura, ou então ser uma especificação introduzida directamente pelo projectista, com base na sua experiência e no conhecimento do sistema a desenvolver.

Paralelamente às tarefas de concepção e implementação, entre as quais se inclui a partição e o refinamento de descrições, o co-projecto de *hardware* e de *software* inclui tarefas de validação para detectar erros em diferentes níveis de abstracção. Uma das formas de validação é a

[3] ***H**igh-**L**evel **L**anguage* e ***H**ardware **D**escription **L**anguage*, na terminologia inglesa.

co-verificação, ou seja, a verificação conjunta dum sistema a implementar com *hardware* e *software*. A co-verificação prova, de modo formal, que propriedades importantes (como a ausência de bloqueios) vão ocorrer na implementação. Quando se pretende validar todo um sistema, a co-verificação será mais difícil de concretizar no caso do sistema estar representado por um conjunto de descrições parcelares, do que no caso de estar descrito de forma unificada. Se as descrições parcelares aplicarem mais do que um meta-modelo, a co-verificação fica ainda mais dificultada.

A co-simulação é outra forma de validação, que pode ser aplicada em diferentes pontos do processo de desenvolvimento, e baseia-se na aplicação de estímulos e na observação dos resultados produzidos. Quando se trata de validar sistemas complexos, embora a co-simulação apresente limitações, porque não é exequível testar todas as combinações de estímulos, é muitas vezes a única alternativa disponível. Isto porque a verificação formal e exaustiva de sistemas complexos, ao ser difícil de concretizar e exigir um tempo de cálculo muito elevado, deixa de ser uma alternativa válida.

Para projectar sistemas complexos é conveniente utilizar uma metodologia de desenvolvimento baseada na abordagem operacional [Zav84]. Segundo esta abordagem, para conseguir a implementação dum sistema concebe-se uma especificação executável, à qual será aplicado um conjunto de refinamentos, acompanhados de transformações. Com sistemas complexos, a implementar com vários componentes de *hardware* e de *software*, é conveniente que a metodologia funcione num nível de abstracção tão alto quanto possível (do sistema ou de transferência de registos), por forma a reduzir a complexidade do problema e diminuir o tempo de projecto dos sistemas. Uma metodologia que funciona num nível de abstracção alto permite que o projectista domine mais facilmente o desenvolvimento dos sistemas, dado que os objectos que descrevem esses sistemas são em menor número e mais inteligíveis para o projectista. As abordagens que usam o paradigma da modelação orientada ao objecto correspondem aos requisitos enunciados, uma vez que permitem facilmente o refinamento de objectos do domínio de aplicação durante todo o processo. Estas abordagens empregam normalmente modelos multi-vista que englobam os objectos do sistema e as perspectivas dinâmica e funcional. Contudo, existem poucas metodologias para co-projecto de *hardware* e de *software* que explorem os benefícios das técnicas de modelação orientadas ao objecto.

A metodologia associada à abordagem MOOSE segue os princípios citados no parágrafo anterior [MEGT96]. Esta metodologia apoia o desenvolvimento de sistemas computacionais embebidos, suporta o paradigma de co-projecto de *hardware* e de *software* e emprega vários modelos abstractos, entre eles um modelo executável. O MOOSE usa modelação multi-vista para especificar os sistemas: (i) um diagrama de objectos para modelar a funcionalidade (estática) do sistema, (ii) um modelo do domínio de aplicação para modelar os objectos do

domínio de aplicação e (iii) um diagrama de transição de estados para modelar o comportamento dinâmico do sistema.

A metodologia de desenvolvimento adoptada (figura 1.1), na qual se inclui a metodologia de partição aqui proposta, foi inspirada no MOOSE. Entre as principais diferenças incluem-se:

- os diagramas de transição de estado (STD[4]) da metodologia do MOOSE, cujo metamodelo é uma extensão da FSM[5], foram substituídos por modelos PSM[6] [GVN94], o que permite uma gestão mais adequada da concorrência existente entre os objectos do sistema; por outro lado, enquanto os STDs servem apenas para documentar a funcionalidade do sistema, os modelos PSM são aplicados na tarefa de partição;

- a metodologia do MOOSE segue maioritariamente um modelo de processo[7] em cascata, enquanto a presente metodologia recorre a uma abordagem mais iterativa para obter a implementação dos sistemas, uma vez que as fases de análise [Fer00], de concepção [Mac00] e de implementação funcionam de forma iterativa; por exemplo, a solução de partição óptima é obtida através dum processo de optimização iterativo;

- incluiu-se uma fase de teste que acompanha a concepção e a implementação dos sistemas, e permite simular a funcionalidade presente na especificação do sistema, co-simular o resultado da partição desta funcionalidade e co-verificar a implementação; a única forma de validação disponível no MOOSE é o modelo executável para a arquitectura do sistema, obtido na fase de concepção; este modelo valida o comportamento estático e alguns aspectos do comportamento dinâmico do sistema; como o comportamento temporal é modelado de forma inexacta, dado que o modelo executável não contém detalhes relativos à implementação, o comportamento dinâmico do sistema não pode ser correctamente validado;

- a partição manual no MOOSE é baseada na experiência e na intuição do projectista, enquanto a metodologia aqui proposta utiliza uma partição automática com reduzida intervenção do projectista;

- a metodologia proposta assume que as soluções de partição são implementadas com uma arquitectura alvo composta por uma plataforma EDgAR-2 ligada por um barramento PCI a um sistema uni ou multi-processador; a plataforma EDgAR-2 dispõe de dois tipos de componente reconfigurável, um mais adequado para implementar a unidade de controlo dos sistemas e o outro para implementar o seu caminho de dados (*data path*); a

[4] ***S**tate **T**ransition **D**iagrams*, na terminologia inglesa.
[5] ***F**inite **S**tate **M**achine*, na terminologia inglesa.
[6] ***P**rogram-**S**tate **M**achine*, na terminologia inglesa.
[7] O **modelo de processo** também é conhecido por **fluxo de projecto**.

metodologia do MOOSE não considera uma arquitectura alvo pré-definida, antes deixa para o projectista a tarefa de desenvolver a arquitectura que mais se adequa ao sistema em projecto.

1.3 Sistemas Embebidos

Os sistemas digitais podem ser categorizados em duas grandes classes: a classe dos sistemas genéricos que executam uma gama alargada de tarefas e a classe dos sistemas vocacionados para executar apenas determinadas tarefas. Os sistemas embebidos incluem-se na classe dos sistemas vocacionados para fins específicos. Não existindo uma definição consensual para estes sistemas, é habitual apresentar as suas características e apontar exemplos paradigmáticos deste tipo de sistema. Pode assim dizer-se que um sistema embebido apresenta as seguintes características: faz parte dum sistema de maior dimensão com o qual interactua, deve estar preparado para responder a estímulos externos, está normalmente sujeito a requisitos temporais exigentes, a sua concepção é fortemente determinada por requisitos não funcionais (tais como o custo, o desempenho ou o consumo), exige-se-lhe uma elevada fiabilidade e correcção por ser aplicado em sistemas fortemente penalizados por falhas (como é o caso de automóveis, aviões ou máquinas industriais), não é programável pelo utilizador final, é orientado para tarefas específicas mas permite alguma configuração e habitualmente integra um microprocessador ou um microcontrolador. Exemplos de sistemas embebidos são os módulos electrónicos que controlam o funcionamento de diversas componentes dum automóvel (sistema de travagem com anti-bloqueio, sistema de tracção ou *airbags*), a parte electrónica de electrodomésticos e impressoras, o sistema de navegação e controlo dum avião, os sistemas de aquisição de dados usados em equipamento laboratorial ou os sistemas personalizados que monitorizam e controlam os processos industriais.

A relação entre o valor e o custo dum sistema embebido é determinada por factores como os custos de projecto e produção, o tempo de produção, a facilidade de depuração e de teste, o desempenho e o consumo do produto final, o aspecto exterior e as dimensões, a fiabilidade e a correcção, a facilidade de actualização e os custos de manutenção e assistência.

Os sistemas embebidos podem ainda ser subdivididos em controladores embebidos e sistemas de processamento de dados embebidos.

Os **controladores embebidos** são sistemas predominantemente de controlo, normalmente reactivos a estímulos exteriores, dos quais se espera um tempo de resposta inferior a um limite pré-definido, que normalmente é da ordem dos milisegundos. A ordem de grandeza do tempo de resposta está relacionada com a rapidez com que o estado do sistema envolvente pode variar. Como na maior parte das situações o sistema envolvente é do tipo electro-

mecânico, o seu estado não varia muito bruscamente e o controlador fica sujeito a requisitos de desempenho pouco exigentes. Porque se lhe exige um desempenho baixo ou moderado e porque possuem um dimensão tipicamente mediana, os controladores embebidos são normalmente implementados com um microcontrolador. Alguns exemplos de aplicação para estes sistemas são os automóveis, os electrodomésticos, as aplicações domésticas ou os robôs industriais.

Os **sistemas de processamento de dados embebidos** são sistemas predominantemente de fluxo de dados, normalmente de tempo real e aos quais se exige um desempenho elevado. Para atingir um desempenho elevado, a sua implementação utiliza DSPs[8], ASIPs[9] e microprocessadores genéricos de alto desempenho, bem mais poderosos que os microcontroladores utilizados nos controladores embebidos. A arquitectura usada na implementação é habitualmente heterogénea e inclui componentes de *software* e de *hardware*, tais como microprocessadores, RAM, ROM, ASICs, PLDs e/ou conversores analógico/digital e digital/analógico. Deste facto resulta que o paradigma do co-projecto de *hardware* e de *software* tem maior aplicabilidade no projecto de sistemas de processamento de dados embebidos do que no projecto de controladores embebidos. O domínio de aplicação destes sistemas é o multimédia (áudio e vídeo), a electrónica de consumo (consolas de vídeo, impressoras e faxes), as comunicações sem fios (telefones sem fios e celulares) ou outros tipos de comunicação (ATM[10] e RDIS[11]).

1.4 Objectivos do Trabalho

Como se afirmou anteriormente, o principal objectivo do presente trabalho consistia em desenvolver e concretizar uma metodologia de partição automática para apoiar o projecto de sistemas embebidos predominantemente de fluxo de dados, tendo em vista a sua implementação com uma arquitectura alvo composta por componentes de *hardware* e de *software*. Para atingir este desiderato definiram-se os seguintes objectivos parcelares:

- ◇ escolher os meta-modelos mais adequados ao processo de partição;
- ◇ seleccionar o tipo de granulosidade utilizada com os objectos manipulados durante o processo de partição;
- ◇ seleccionar e implementar os algoritmos de partição envolvidos no processo de partição;
- ◇ desenvolver as funções de proximidade e de custo que informam os algoritmos sobre a qualidade das soluções de partição;

[8] ***D**igital **S**ignal **P**rocessors*, na terminologia inglesa.
[9] ***A**pplication **S**pecific **I**nstruction-set **P**rocessors*, na terminologia inglesa.
[10] ***A**synchronous **T**ransfer **M**ode*, na terminologia inglesa.
[11] **R**ede **D**igital **I**ntegrada de **S**erviços.

- ◇ desenvolver e concretizar a estimação das métricas utilizadas pelas funções de proximidade e de custo;
- ◇ optimizar a metodologia de partição, reduzindo o número de alternativas de partição a analisar pelos algoritmos;
- ◇ validar a metodologia de partição através de casos de estudo.

Para escolher os meta-modelos a utilizar à entrada, à saída e internamente ao processo de partição, estudou-se a adequação dos meta-modelos gerados pelas fases de análise e concepção para o tipo de sistema (embebidos predominantemente de dados) e tarefas a executar (partição em componentes e estimação de métricas). O meta-modelo com que se representam as soluções de partição deve poder ser convertido num dos meta-modelos suportados pela arquitectura alvo usada na implementação.

Ao estudar a questão da granulosidade dos objectos manipulados no processo de partição pretendia-se determinar qual o tamanho de objecto mais recomendável para a tarefa de partição e resolver a questão de saber se os objectos manipulados pelo algoritmo de partição eram explicitamente definidos no modelo de entrada ou não. A granulosidade seleccionada resultou duma solução de compromisso entre poucos objectos mas de grande dimensão, que se constituem como um problema de partição mais simples e uma solução menos optimizada, versus muitos objectos de pequena dimensão, que originam um problema de partição mais complexo e uma solução mais optimizada. Relativamente à qualidade da solução, tem que se ter em atenção que um problema complexo, como é o caso em que se usam muitos objectos na partição, pode esconder a melhor solução. Por outro lado, lidar com objectos maiores que tendem a coincidir com os módulos incluídos na especificação do sistema, significa que o processo de partição preserva a estrutura deste sistema. Esta característica é favorável à interacção com o projectista [CLL+96], mas dificulta a rentabilização dos recursos da arquitectura alvo.

De acordo com a organização da metodologia de partição proposta, o processo de partição começa com a geração duma alternativa de partição inicial, por intermédio dum algoritmo construtivo, a qual é depois sucessivamente melhorada por um algoritmo iterativo. Deste modo, foi necessário seleccionar um algoritmo construtivo e outro iterativo.

Os requisitos não funcionais (tempos de resposta ou atrasos) que resultaram da fase de análise ou concepção, foram utilizados para definir o que é uma solução óptima. Analogamente, estabeleceu-se como exequível uma solução que respeita todos os condicionalismos impostos pela arquitectura alvo à implementação. Os principais condicionalismos são o número de componentes a usar na implementação, a quantidade de recursos em cada componente, a largura da ligação entre componentes e os meta-modelos de comunicação suportados. Para verificar se os requisitos e condicionalismos são respeitados, definiram-se funções de proximidade e de

custo. A cada requisito e condicionalismo que contribui para a função de proximidade ou custo foi atribuído um peso.

Para verificar a qualidade das soluções de partição, através do valor devolvido pela função de proximidade ou de custo, foi preciso estimar valores para as métricas associadas aos requisitos e condicionalismos presentes nesta função. A estimação de métricas exigiu a definição de modelos para os recursos presentes na implementação, mais ou menos pormenorizados consoante a precisão desejada para as estimativas e o tempo que se pretende despender nos cálculos. Idealmente a estimação deveria permitir variar a relação entre precisão e tempo de cálculo. No entanto, a metodologia proposta deu prioridade à qualidade das estimativas, mas não descurou a optimização dos estimadores por forma a minimizar o tempo de cálculo. As métricas a estimar são o espaço ocupado pelos recursos de *hardware* e o tempo de execução em *software* e em *hardware*, que por sua vez exige conhecer o espaço e o tempo associado com a computação e a comunicação entre os diversos componentes. Como a capacidade de armazenamento de informação nos sistemas computacionais baseados em processadores é elevada, não constitui um condicionalismo para a implementação da maioria dos sistemas embebidos. Foi este o motivo que levou à não inclusão da métrica "espaço ocupado pelos recursos de *software*" no processo de partição.

Entre as alternativas com que se procurou reduzir o espaço de projecto, definido como o universo de soluções manipuladas durante o processo de partição, incluem-se (i) o recurso a uma biblioteca de componentes (ou ficheiro tecnológico), (ii) a utilização do conhecimento que o projectista tem das características do sistema em desenvolvimento e (iii) a aplicação dos resultados da síntese na melhoria da convergência do processo de partição iterativo para a solução óptima.

Os resultados obtidos com os casos de estudo permitiram avaliar a qualidade das soluções de partição, a precisão das estimativas, o desempenho da ferramenta que concretiza a metodologia de partição proposta e o apoio que presta às restantes tarefas da implementação dos sistemas, nomeadamente ao refinamento das descrições com detalhes sobre a interface entre partições.

1.5 Estrutura da Tese

Esta tese encontra-se dividida em três partes. A primeira parte, composta pela introdução mais os capítulos 2 a 4, caracteriza o trabalho realizado e estabelece a sua ligação com o estado da arte na área científica em que se enquadra, ou seja, a partição de sistemas digitais para implementação com componentes de *software* e de *hardware*. Na segunda parte, que inclui os capítulos 5 a 9, descrevem-se as contribuições científicas e as concretizações que resultaram da metodologia de partição desenvolvida. Na terceira e última parte, constituída

pelos apêndices A a H, apresenta-se informação complementar sobre as duas partes anteriores.

O **capítulo 2** resume os meta-modelos mais utilizados em sistemas digitais embebidos, os requisitos que eles devem possuir, as linguagens mais utilizadas para os concretizar e qual o meta-modelo adoptado no presente trabalho para descrever os sistemas antes e após o processo de partição.

O **capítulo 3** define o processo de partição de sistemas em componentes diferenciados, apresenta exemplos de abordagens ao problema de partição e estabelece a organização da abordagem típica a este problema. Exemplos relevantes de algoritmos de partição construtivos, iterativos e específicos de determinada abordagem são aqui analisados. Desta análise resultaram os algoritmos seleccionados para o presente trabalho. Fez-se ainda o estudo dum conjunto de funções de proximidade e de custo, que auxiliaram a definição das funções aplicadas neste trabalho.

A forma como a estimação das métricas envolvidas nas funções de proximidade e de custo é tratada na bibliografia está resumida no **capítulo 4**. Além de definições, este capítulo discute os modelos de estimação. Partindo duma abordagem genérica, para estimar um vasto conjunto de métricas de desempenho e de custo relativas a *hardware* e *software*, termina-se com um conjunto significativo de abordagens que se especializaram na estimação duma ou mais métricas de *hardware*, de *software* ou associadas à comunicação entre componentes.

O **capítulo 5** discute uma parte fundamental da arquitectura alvo considerada na metodologia de partição proposta: a plataforma EDgAR-2. Apresentam-se os meta-modelos de computação e de comunicação suportados, o tempo de reconfiguração dos componentes e os condicionalismos impostos pela arquitectura às tarefas de partição e síntese.

O algoritmo construtivo e a função de proximidade por ele utilizada para estabelecer a solução inicial do processo de partição iterativo, assim como o algoritmo iterativo de pesquisa tabu e a respectiva função de custo, são apresentados no **capítulo 6**. Os fundamentos do método de pesquisa tabu (os tipos de tabu, a lista tabu, a pesquisa na vizinhança, os critérios de aspiração e os mecanismos de diversificação/intensificação) e a sua adaptação à presente metodologia são explicados em pormenor. O PSMfg, o meta-modelo aplicado na representação dos sistemas durante o processo de partição, é aqui introduzido.

A abordagem seguida pela presente metodologia de partição para estimar as métricas exigidas pela função de custo do processo de partição iterativo é discutida no **capítulo 7**. Os modelos de estimação para os recursos de *software*, de *hardware* e de interligação disponíveis para a implementação das soluções de partição são apresentados. O processo de estimação

decorre em dois níveis de abstracção. No nível mais alto estima-se o espaço ocupado pelas partições de *hardware* e o desempenho do sistema, enquanto no nível mais baixo estimam-se métricas relativas aos objectos do grafo que representa o sistema. O método de estimação do desempenho baseia-se no cálculo do tempo de execução nos vários fluxos do grafo do sistema.

O **capítulo 8** descreve a validação da metodologia de partição proposta, através de dois casos de estudo. Para cada exemplo estudado documenta-se a melhor solução de partição encontrada pelo algoritmo de pesquisa tabu e uma implementação *hardware/software* optimizada com a qual se avalia a qualidade desta solução. Com os resultados obtidos quantifica-se a precisão/fidelidade das estimativas e o desempenho da ferramenta de partição desenvolvida.

Finalmente, o **capítulo 9** apresenta as conclusões sobre o trabalho realizado, com realce para os seus contributos e o trabalho futuro, sob a forma de tarefas não concretizadas ou de melhoramentos à metodologia de partição proposta.

Nos **apêndices A** a **H** encontra-se:

- ◇ informação adicional sobre os algoritmos de Kernighan/Lin, *simulated annealing*, pesquisa binária condicionada, PACE e GCLP, discutidos no capítulo 3;
- ◇ a notação $\mathcal{O}$, Ω e Θ que se aplica na representação da complexidade de algoritmos;
- ◇ informação adicional sobre algumas das funções de custo apresentadas no capítulo 3;
- ◇ um resumo das potencialidades da ferramenta *parTiTool* desenvolvida para automatizar o processo de partição;
- ◇ o código que implementa o algoritmo de crescimento de grupos, concretamente a função de proximidade por ele aplicada;
- ◇ informação complementar sobre o processo de estimação do tempo de comunicação entre estados programa e variáveis, discutido no capítulo 7;
- ◇ o código que implementa a estimação do tempo de comunicação entre estados programa e variáveis e do tempo de execução do sistema.

Capítulo 2

Modelação de Sistemas

Sumário

Com o objectivo de mostrar a importância da modelação no desenvolvimento de sistemas, apresenta-se a metodologia do MOOSE, uma metodologia que usa vários modelos ao longo do processo de desenvolvimento. Depois definem-se as características que os meta-modelos a aplicar na modelação de sistemas digitais embebidos devem conseguir representar. Faz-se um resumo dos meta-modelos mais utilizados com sistemas embebidos, organizado de acordo com a classificação de orientados ao estado, orientados à actividade, orientados aos dados, orientados à estrutura e heterogéneos. Identificam-se as linguagens que habitualmente concretizam os meta-modelos apresentados e para concluir, justifica-se a selecção do meta-modelo PSM para modelar os sistemas no presente trabalho.

Conteúdo

2.1 Introdução

A modelação surge da necessidade de definir a funcionalidade dos sistemas a desenvolver de uma forma não ambígua e completa, tarefa não possível com descrições em linguagem natural. Um **meta-modelo**, ou o modelo dum modelo, pode ser definido como sendo um conjunto de elementos de composição (funcionais ou estruturais) e as regras de composição que permitem construir um modelo conceptual (virtual ou abstracto) para o sistema. Um meta-modelo deve (i) ser formal, o mesmo é dizer preciso e rigoroso, para evitar ambiguidades no modelo do sistema, (ii) ser completo, para permitir a descrição total do modelo pretendido para a vista do sistema e (iii) gerar modelos compreensíveis ao projectista, por forma a que este

os possa manusear e alterar. Por **vista do sistema** entende-se uma perspectiva, parcial ou total, do sistema a descrever. Por exemplo, a parte de controlo e a parte de dados são duas vistas complementares dum sistema, enquanto a estrutura e o comportamento são duas vistas alternativas, ou complementares, para o mesmo sistema.

Por seu lado, um **modelo** é a representação conceptual de uma vista do sistema segundo um determinado meta-modelo. Assim, com diferentes meta-modelos podem representar-se vistas diferentes do mesmo sistema, tais como a relação entre entradas e saídas ou a evolução do estado do sistema ao longo do tempo.

A notação (sintaxe e semântica) que permite concretizar (especificar) o modelo do sistema numa dada representação constitui uma **linguagem**. Um modelo pode ser concretizado em várias linguagens, e uma linguagem pode concretizar modelos provenientes de meta-modelos distintos.

A escolha dum meta-modelo é condicionada pelos domínios de representação que ele suporta e pelos níveis de abstracção em que permite trabalhar. Um meta-modelo deve ainda permitir representar os condicionalismos impostos ao sistema, bem como os requisitos que lhe são exigidos. Deste modo, tem que se seleccionar um meta-modelo adequado para o tipo de sistema a projectar.

O meta-modelo utilizado também pode mudar com o decorrer do processo de desenvolvimento, escolhendo-se sempre que se justifique, o meta-modelo mais adequado para as tarefas em execução e para representar a informação a manipular. Neste caso, deve garantir-se que há continuidade entre as representações sucessivas, o que quer dizer que é preciso manter a informação contida na descrição do sistema, ao evoluir-se dum modelo para outro.

Em sistemas digitais, é comum o **domínio** de representação ser classificado numa de três alternativas: comportamental, estrutural ou físico; enquanto os **níveis de abstracção**, do mais alto para o mais baixo, habitualmente considerados são: sistema, algorítmico, arquitectura ou transferência de registos (RTL), porta lógica, transístor e físico. O nível de abstracção está intimamente ligado à granulosidade dos objectos manipulados nesse nível, verificando-se que quanto mais elevado é o nível de abstracção mais "grossos" são os objectos.

Uma descrição comportamental define apenas a funcionalidade do sistema, não contém informação relativa à sua implementação. As descrições estruturais representam o sistema através de um conjunto de componentes interligados, aos quais o comportamento foi atribuído. Por fim, as descrições físicas adicionam à estrutura composta pelo tipo de componentes e sua interligação, informação relativa a características físicas desses componentes e interligações, tais como o tamanho, a capacidade de dissipação de calor, a localização dos pinos e o consumo. Convém referir que no processo de desenvolvimento dum sistema se evolui normalmente

duma descrição comportamental para uma estrutural (**síntese**) e de um nível de abstracção mais alto para outro mais baixo (**refinamento**). Uma descrição pode ainda ser optimizada, sem que para isso se altere o domínio de representação ou o nível de abstracção (**optimização**). Conforme se vão adicionando pormenores relativos à implementação, a descrição obtida vai perdendo clareza na forma como representa o sistema. Dai que quando se desenvolvem sistemas cada vez mais complexos, o domínio comportamental é preferível porque facilita a obtenção de soluções com funcionalidade correcta.

Um projectista deve optar pelo nível em que a representação do sistema é mais perceptível e onde o número de objectos presentes na representação é tratável. No domínio comportamental e ao nível de transferência de registos, o tipo de objectos que os meta-modelos incluem são algoritmos, diagramas de fluxo, conjuntos de instruções ou máquinas de estados finitos (FSMs) generalizadas; enquanto no domínio comportamental e ao nível do sistema, o tipo de objectos presentes nos meta-modelos são especificações executáveis numa HDL, algoritmos ou programas.

Quando se pretende efectuar a concepção maioritariamente num determinado nível, há que seleccionar um meta-modelo adequado às tarefas comuns nesse nível de abstracção e que ao mesmo tempo represente adequadamente o ponto de vista que o projectista tem do sistema. Assim, ao projectar sistemas no nível de transferência de registos e usando descrições comportamentais dos sistemas, algumas das tarefas que o meta-modelo deve permitir são a partição de descrições, a estimação de métricas (espaço, desempenho, largura da interface) e as tarefas de síntese comportamental ou de alto nível (escalonamento de operações, selecção de componentes e atribuição das operações e variáveis aos componentes) que transformam o comportamento numa estrutura com detalhes da implementação.

O modelo do sistema pode definir a arquitectura desse sistema, sem contudo incluir informação relativa à forma de implementação. O tipo de arquitectura obtida será diferente consoante o nível de abstracção seleccionado para a definir.

Como o presente trabalho se concentra na fase de implementação de sistemas num ambiente de co-projecto, este capítulo aborda o problema da escolha dos meta-modelos mais adequados a esta fase do desenvolvimento. Interessa pois considerar os meta-modelos devolvidos pela concepção e os possíveis meta-modelos internos à implementação. Nas secções seguintes apresenta-se um conjunto de meta-modelos, realçando os seus pontos fortes/fracos e o tipo de sistemas para os quais apresentam maiores potencialidades de modelação.

2.2 Uma Metodologia de Desenvolvimento Orientada aos Modelos

Com o objectivo de mostrar a importância da modelação no desenvolvimento de sistemas, apresenta-se a metodologia do MOOSE [MEGT96], uma metodologia que aplica vários modelos ao longo do processo de desenvolvimento, por forma a representar os detalhes de implementação que vão sendo adicionados à descrição inicial do sistema. Para representar a funcionalidade do sistema empregam-se diferentes meta-modelos na representação das diferentes vistas do sistema, o que equivale a dizer que se utiliza **modelação heterogénea**. Ao refinar e detalhar a funcionalidade dum sistema, as regras de composição do meta-modelo vão-se mantendo e os elementos de composição são estendidos quando as tarefas a executar e a nova informação a representar o exigem. A metodologia suporta o paradigma do co-projecto de *hardware* e de *software*, uma vez que modela e concebe sistemas a implementar com componentes de *hardware* e de *software*, tratando de forma indiferenciada e unificada as partes de *hardware* e de *software*. A metodologia está vocacionada para desenvolver sistemas complexos embebidos e/ou reactivos.

Nesta metodologia as fases do processo de desenvolvimento estão associadas aos modelos que se pretende obter e não ao tipo de tarefas a realizar. A utilização de diferentes modelos deve-se ao facto de um único modelo não conseguir representar o sistema da forma mais adequada na totalidade das etapas do processo de desenvolvimento. A figura 2.1 ilustra os modelos que intervêm no desenvolvimento dum sistema na abordagem MOOSE, sendo este processo organizado em três grandes fases: a análise, a concepção e a implementação.

2.2.1 Análise

A primeira fase do processo de desenvolvimento é habitualmente designada de análise. Com base numa proposta inicial de produto obtém-se uma lista classificada de **requisitos**, que podem ser requisitos funcionais ou condicionalismos do projecto. Os condicionalismos do projecto podem ainda ser classificados em requisitos não funcionais, decisões de projecto ou objectivos do projecto. Os requisitos funcionais são o ponto de partida do modelo funcional e os condicionalismos do projecto limitam o espaço de projecto analisado durante o processo de desenvolvimento. Os objectivos do projecto definem requisitos que não possuem quantificação precisa, as decisões de projecto são opções sobre o tipo de implementação assumidas numa fase inicial do desenvolvimento, enquanto os requisitos não funcionais definem objectivos para o preço, o tamanho, o consumo, a fiabilidade ou o desempenho da implementação. O resultado da análise é um documento que descreve os requisitos em linguagem natural e é compreendida pela maioria das pessoas envolvidas no projecto. A concepção do sistema

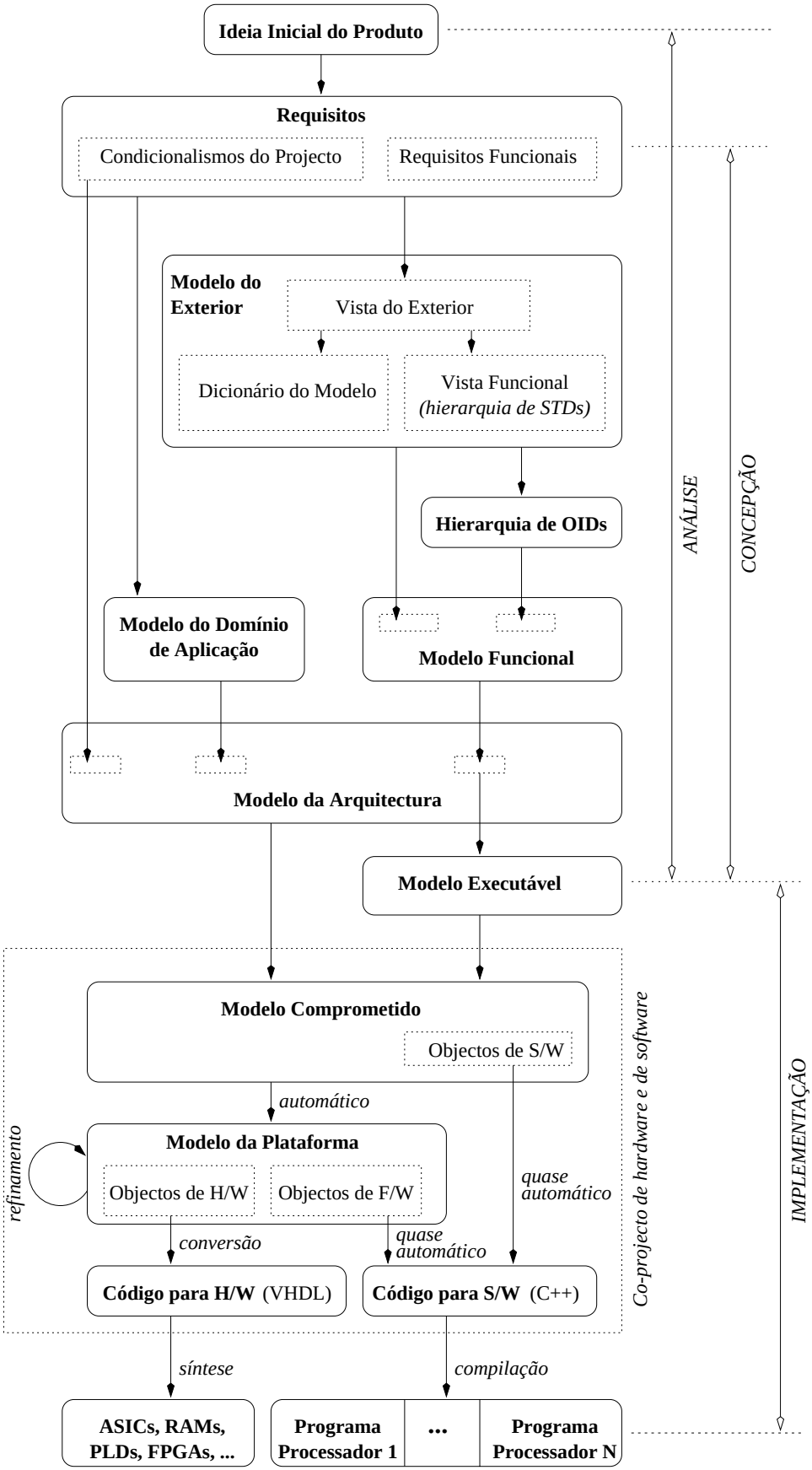


Figura 2.1: Modelos utilizados na abordagem MOOSE para desenvolver um sistema.

deverá obedecer aos requisitos obtidos nesta fase, por forma a obter uma implementação optimizada segundo determinados critérios, garantindo deste modo a satisfação do utilizador e uma maior longevidade ao produto final.

2.2.2 Concepção

A segunda fase designa-se por concepção e permite obter a **arquitectura** de todo o sistema[1], a qual define a sua organização em componentes com funcionalidade e requisitos próprios. Na arquitectura não entram detalhes relativos à posterior implementação dos componentes desta arquitectura. A modelação da arquitectura começa com a obtenção do **modelo funcional** do sistema, que é condicionado pelos requisitos obtidos na fase anterior e é composto pelo modelo do exterior e por uma hierarquia de diagramas OID[2].

A hierarquia de diagramas OID define estruturalmente a funcionalidade do sistema através de objectos e ligações, explora a concorrência entre actividades e pode conter o conceito de tempo.

Por sua vez, o modelo do exterior é formado por: (i) uma vista do exterior, que define a interface do sistema com o seu ambiente de funcionamento; esta vista é modelada com um OID onde que se inclui o objecto sistema, os objectos externos que interactuam com o sistema e as ligações entre objectos; (ii) uma vista funcional com a forma de hierarquia de STDs, que modela a funcionalidade do sistema através da sua organização em etapas temporais (estados); esta vista funciona como documentação da funcionalidade do sistema; (iii) um dicionário do modelo, que define formalmente e de forma descritiva o significado de cada ligação presente no modelo.

O **modelo da arquitectura** é depois obtido pela conjugação dos requisitos, com o modelo funcional e com o modelo do domínio de aplicação. O modelo do domínio de aplicação é um diagrama que captura informação sobre as entidades do domínio de aplicação do sistema. Esta informação é útil à transformação da arquitectura numa implementação. O meta-modelo deste diagrama coincide com o meta-modelo do *initial object model* do OMT [RBP+91].

Diagramas OID

O meta-modelo dos diagramas OID é idêntico ao *DFD* ou *DFG*[3]. Os tipos de objecto presentes neste meta-modelo são o objecto externo, o objecto não comprometido que representa o sistema na vista do exterior, o objecto não comprometido presente nos OIDs e definido à custa de outros objectos, o objecto primitivo presente nos OIDs e que representa a instanciação

[1] Distinta da arquitectura da plataforma usada na implementação do sistema.
[2] ***O**bject **I**nteraction **D**iagrams*, na terminologia inglesa.
[3] ***D**ata **F**low **D**iagram* ou ***D**ata **F**low **G**raph*, na terminologia inglesa.

duma classe e o objecto da biblioteca e que representa a instanciação duma classe. Os objectos podem ainda ser coloridos, por forma a indicar um tipo de implementação, aplicando-se cores diferentes na representação dos objectos comprometidos de *hardware*, de *software*, de *firmware* e compostos.

Como o MOOSE permite a troca de informação entre objectos de *software* e de *hardware*, os quais podem apresentar características bastante distintas, a modelação em MOOSE permite que os objectos comuniquem por vários mecanismos e não apenas pela forma de passagem de mensagens intrínseca aos meta-modelos OO e que é vocacionada para implementações puramente de *software*. Os mecanismos de comunicação, ou seja, os tipos de ligação entre objectos dos OIDs permitidos pelo MOOSE são: (i) a **interacção**, que representa a comunicação por passagem de mensagens e permite a um objecto requisitar um método doutro objecto; (ii) o **fluxo de informação**, que modela a troca de informação contínua ou discreta entre objectos, sem que haja necessidade de o emissor da informação conhecer o instante em que o receptor faz uso dessa informação; (iii) o **evento**, que permite a um objecto comunicar a outros a detecção de mudanças de estados, as quais podem ser relevantes a esses objectos; (iv) o **grupo**, que representa uma comunicação composta por uma hierarquia de outras formas de comunicação mais elementares (interacções, eventos ou fluxos de informação).

O meta-modelo dos OIDs permite outros elementos de composição:

- o *nodo de definição*, vocacionado para atribuir um valor a uma constante e permitir assim a distinção de objectos da mesma classe que diferem apenas duma constante;
- o nodo que cria e elimina dinamicamente um objecto, ficando estes objectos comprometidos com uma implementação de *software*;
- anotações em objectos ou ligações, de modo a instanciar de forma implícita uma série de objectos ou ligações do mesmo tipo.

Diagramas STD

O meta-modelo dos diagramas STD é a FSM do tipo Mealy, com uma sintaxe particular e com algumas extensões ao meta-modelo FSM base. Primeiro, possui o conceito de estado do sistema, que mais não é do que um macro estado que introduz hierarquia no meta-modelo e permite modelar sistemas complexos duma forma mais tratável do que se consegue com FSMs. Segundo, permite a definição de estados concorrentes por forma a modelar sistemas com actividades concorrentes. Para isso, a sintaxe do meta-modelo permite transitar simultaneamente dum estado para vários estados.

Modelo Executável

Na fase de concepção obtém-se também um modelo executável numa HLL, que funciona como protótipo de *software* que valida o aspecto funcional da arquitectura desenvolvida e verifica a correcção da sua estrutura e operação internas. Este modelo permite ainda verificar aspectos relacionados com o comportamento dinâmico do sistema, embora com a limitação que resulta do facto de o comportamento temporal ser modelado com valores inexactos. Esta limitação acontece porque o modelo não contém detalhes relativos à implementação.

Para obter o modelo executável adiciona-se ao modelo funcional o código que descreve a funcionalidade dos seus objectos primitivos. A instanciação dos objectos e a comunicação entre eles é obtida automaticamente do modelo funcional. O código C++ obtido é reutilizável na síntese da implementação do sistema. A geração de código para as classes correspondentes aos objectos primitivos é suportada por quatro formas de representação:

- a especificação da interface duma classe, ou CIS[4], que é a representação textual que define as entradas e saídas dos objectos pertencentes a essa classe; para cada entrada ou saída define-se o tipo de ligação (interacção, evento), a direcção (entrada, saída), o nome e o tipo da informação que flui pela ligação (inteiro, booleano, *void*) e o tipo da informação de retorno no caso de interacções;
- o diagrama de implementação duma classe, ou CID[5], define os objectos e as interligações que compõem essa classe e possui uma representação gráfica idêntica à dos diagramas OID; os objectos dum diagrama CID representam métodos, variáveis que guardam o estado duma classe, máquinas de estados, classes herdadas, funções para enviar ou receber eventos, funções *construtor*, funções *destruidor* ou funções do tipo *activo*; as funções do tipo *activo* permitem manter a concorrência no modelo executável, ao permanecerem activas mesmo quando não recebem estímulos;
- a representação textual em C++ das funções ou métodos (FSPECs) e das variáveis (DSPECs);
- a representação textual que converte o nome das ligações presentes no modelo funcional para o nome usado nas classes (OSPECs).

O comportamento dinâmico do modelo executável reside essencialmente na concorrência entre objectos primitivos. O grau de concorrência é medido pelo número de funções activas presentes no modelo. Para simplificar a dinâmica do modelo executável, considera-se que todas as

[4] ***C**lass **I**nterface **S**pecification*, na terminologia inglesa.
[5] ***C**lass **I**mplementation **D**iagram*, na terminologia inglesa.

funções activas que enviam ou recebem eventos são concorrentes, e mesmo que uma função seja activada por múltiplas interacções ela só possui um fio de execução (*thread*).

2.2.3 Implementação

Na terceira fase do processo de desenvolvimento obtém-se uma implementação para o sistema.

O primeiro passo para obter uma implementação é a geração do modelo comprometido. O comprometimento de objectos com a plataforma alvo, tarefa que se pode designar por partição, e a definição dos próprios recursos da plataforma, são decisões que o utilizador assume com base no conhecimento que possui do sistema e nos requisitos do sistema. Deste modo, a metodologia do MOOSE efectua uma partição manual.

A partir do modelo comprometido evolui-se para o modelo da plataforma. O modelo da plataforma inclui além dos objectos de *hardware* e dos objectos de *firmware* do modelo comprometido, novos objectos de *firmware* que representam o adicionar de detalhes relativos à interface entre *hardware* e *software*. Adicionam-se detalhes quando uma interacção (ou fluxo) entre *hardware* e *software* implica a introdução no modelo de registos e das operações a eles associadas, ou então quando um evento entre *hardware* e *software* é substituído por uma implementação através do mecanismo de interrupção.

O meta-modelo do modelo da plataforma é o diagrama hierárquico OID, com a vista do exterior na raiz da hierarquia. A vista do exterior contém um objecto por cada processador utilizado na implementação, que representa todos os objectos atribuídos a esse processador. Além da vista do exterior, o modelo da plataforma é constituído por outras vistas, que representam diferentes aspectos do processo de conversão da funcionalidade numa implementação: a vista das comunicações do sistema, a vista da interface com o *software* e as vistas de *hardware*.

Posteriormente, refina-se o modelo da plataforma adicionando detalhes de *hardware* e detalhes de baixo nível de *software*, de modo a gerar o código que permitirá sintetizar uma implementação. Duma forma sintética, este processo envolve as seguintes tarefas:

- projectar e implementar os objectos de *software*, de *hardware* e de *firmware* que implementam a interface entre *software* e *hardware* e a interface entre objectos de *software* atribuídos a processadores distintos;
- projectar e implementar o gestor de fios de execução para cada processador, que pode ser visto como o escalonador que gere o tempo de processamento;
- converter a descrição dos objectos de *hardware*, um diagrama CID contendo FSPECs e DSPECs em C++, para uma descrição numa linguagem sintetizável (por exemplo

VHDL).

Com base nos modelos comprometido e da plataforma, gera-se código HDL e HLL que ferramentas comerciais de síntese de *hardware* e de *software* converterão nas componentes de *hardware* e de *software* da implementação.

Na última etapa da implementação, a descrição VHDL gerada na etapa anterior é sintetizada em *netlists*, que descrevem os componentes de *hardware* duma forma estrutural. Enquanto as classes dos objectos de *software*, definidas no modelo comprometido, mais as classes dos objectos de *firmware* do modelo detalhado da arquitectura são convertidas quase automaticamente para código C++, que após compilação gera o programa que corre em cada processador seleccionado.

Os procedimentos que geram o código sintetizável a partir dos modelos executável e da arquitectura são aqueles que mais directamente abordam o paradigma de co-projecto de *hardware* e de *software* (figura 2.1).

2.3 Meta-Modelos para a Representação de Sistemas

Um meta-modelo ideal deveria conseguir representar totalmente um sistema e exigir um esforço mínimo. Como os meta-modelos são normalmente instanciados através duma linguagem de especificação, as características exigidas a um meta-modelo transitam para a linguagem de especificação a usar. Um meta-modelo, ou linguagem de especificação, a aplicar na modelação de sistemas embebidos deve conseguir representar as características que a seguir se descrevem [Nie98].

1. A **hierarquia** permite lidar de forma elegante com a complexidade dos sistemas. A hierarquia pode ser estrutural ou funcional (comportamental). Quando se representa um sistema através dum conjunto de componentes interligados, que por sua vez também são compostos por componentes interligados, está a usar-se hierarquia estrutural. Quando se descreve o comportamento dum sistema através dum conjunto de comportamentos mais elementares (tais como funções, procedimentos ou processos) está a utilizar-se hierarquia funcional.

2. A capacidade de explicitar **concorrência** facilita e clarifica a modelação de sistemas compostos por partes que funcionam em simultâneo. O paralelismo de funcionamento pode ser expresso a vários níveis, entre eles pode citar-se o da tarefa, do processo ou até da operação. Quando o meta-modelo permite concorrência, também deve suportar a definição de **comunicação** entre actividades concorrentes e de **sincronização** dessa comunicação.

3. Com os conceitos de **estado** e de transição de estado modela-se de forma explícita e fácil o estado dos sistemas. O conceito de estado é fundamental em sistemas predominantemente de controlo e/ou reactivos.

4. A representação de **informação dependente do tempo**, como o tempo necessário para realizar uma tarefa (latência) ou o tempo entre a geração de dois valores consecutivos na mesma saída (débito), é importante para garantir que se obtém uma implementação com o comportamento desejado.

5. Os **construtores** e **tipos de dados** comuns em linguagens de programação de alto nível, como por exemplo `while`, `for`, `if`, `struct` ou `record`, são adequados à modelação de sistemas com elevada complexidade algorítmica e que manipulem dados complexos e/ou de forma extensiva. Os sistemas predominantemente de fluxo de dados possuem as características apresentadas, sendo por isso modelados mais confortavelmente com um meta-modelo que permite ciclos, construtores condicionais e consegue representar *arrays*, listas ou outros tipos de dados estruturados.

6. O tratamento de situações de **excepção** é vital em sistemas embebidos do tipo reactivo em que ocorrem eventos assíncronos (como interrupções ou ordens de iniciação) que têm que ser servidos de imediato. Deste modo, o meta-modelo tem que permitir a interrupção da tarefa associada ao estado actual do sistema, a qual será retomada quando terminar o tratamento da excepção.

7. A sinalização da **conclusão de actividades** é importante para se poder especificar tarefas a executar em sequência.

8. A especificação dum sistema de forma **não-determinística**, em que o comportamento é descrito de forma incompleta, permite refinamentos na fase de implementação.

9. Quando um meta-modelo é **formal** pode verificar-se matematicamente se determinadas propriedades dos modelos são garantidas.

10. Um meta-modelo que gera **modelos executáveis** valida as especificações dos sistemas através do processo de simulação, ao gerar um protótipo para todo o sistema numa HLL ou HDL.

O que se verifica é que nenhuma linguagem de especificação, ou meta-modelo, possui todas as características exigidas para a modelação da globalidade dos sistemas embebidos. O que existem são linguagens de especificação, ou meta-modelos, vocacionadas para um sub-conjunto dos sistemas embebidos: predominantemente de controlo, predominantemente de fluxo de dados, reactivos ou tempo real.

Os meta-modelos mais utilizados com sistemas digitais embebidos podem ser classificados em cinco categorias: orientados ao estado (FSM, extensões da FSM, redes de Petri), orientados à actividade (DFG, CFG), orientados ao dado (diagrama de relacionamento entre entidades, diagrama de Jackson), orientados à estrutura e heterogéneos [GVNG94].

Os meta-modelos orientados ao estado representam os sistemas através dum conjunto de estados e de transições entre estados, sendo adequados para sistemas de tempo real reactivos; enquanto os meta-modelos orientados à actividade descrevem os sistemas usando um conjunto de actividades relacionadas por dependências de dados e por dependências de execução das actividades. Estes meta-modelos adequam-se a sistemas em que ao fluxo de informação se aplica uma série de transformações, como acontece, por exemplo, nos processadores digitais de sinal (DSPs). Nos meta-modelos orientados à estrutura usam-se módulos e interligações com correspondência física em vez de componentes da descrição funcional do sistema. Por seu lado, os meta-modelos orientados ao dado representam os sistemas por um conjunto de dados organizados segundo os seus atributos, classes ou segundo outro tipo de classificação. Estes meta-modelos aplicam-se sobretudo em sistemas de informação. Os meta-modelos heterogéneos integram características provenientes de meta-modelos pertencentes às outras quatro categorias, sendo apropriados para modelar sistemas complexos.

2.3.1 Meta-Modelos Orientados ao Estado

Máquina de Estados Finitos

O meta-modelo FSM clássico é composto por um conjunto de estados, um conjunto de transições e um conjunto de acções associadas com os estados e com as transições. Esta definição pode ser representada formalmente por

$$< S, I, O, f : S \times I \rightarrow S, h : S \times I \rightarrow O > \tag{2.1}$$

em que $S = \{s_1, s_2, ..., s_m\}$ representa o conjunto de estados, $I = \{i_1, i_2, ..., i_n\}$ representa o conjunto de entradas, $O = \{o_1, o_2, ..., o_p\}$ representa o conjunto de saídas, f é a função que gera o próximo estado do sistema a partir das entradas e do estado presente e h é a função que gera as saídas a partir das entradas e do estado presente. A equação 2.1 define uma máquina de estados do tipo *Mealy*, na qual as saídas dependem do estado e das entradas. Numa máquina de estados do tipo *Moore* as saídas dependem apenas do estado presente, ou seja, $h : S \rightarrow O$. Para o mesmo sistema, uma máquina de estados do tipo *Moore* pode apresentar um número de estados significativamente maior que uma máquina tipo *Mealy*.

Máquina de Estados Finitos com Caminho de Dados

Um dos problemas das máquinas de estados clássicas é a possibilidade de ocorrer o fenómeno designado por "explosão do número de estados". Uma das situações em que o número de estados explode é quando o sistema lida com variáveis codificadas com muitos bits, como é o caso de variáveis do tipo inteiro ou vírgula flutuante. Para reduzir a explosão do número de estados pode codificar-se um conjunto elevado de estados através duma variável. Por exemplo, uma variável de 16 bits ao codificar 2^{16} estados reduz o número de estados do mesmo factor 2^{16}. O meta-modelo que resulta de estender o meta-modelo FSM com variáveis designa-se por *máquina de estados finitos com caminho de dados* [GDWL92], ou simplesmente *FSMD*[6].

Uma FSMD pode ser definida formalmente por

$$< S, I \cup E, O \cup A, f : S \times (I \cup E) \rightarrow S, h : S \times (I \cup E) \rightarrow (O \cup A) > \tag{2.2}$$

As alterações na equação 2.2 em relação à equação 2.1, resumem-se a que as entradas foram alargadas para incluir algumas variáveis indicadoras de estado E ($E \subseteq ESTADO$) e as saídas foram alargadas com algumas atribuições a variáveis de armazenamento A ($A \subseteq ATRIB$).

Para a equação 2.2 são ainda necessárias as seguintes definições:

- VAR é o conjunto de todas as variáveis de armazenamento;
- $EXP = \{f(x, y, z, ...) \mid x, y, z, ... \in VAR\}$ é o conjunto de todas as expressões;
- $ATRIB = \{X \Leftarrow e \mid X \in VAR, e \in EXP\}$ é o conjunto de atribuições a variáveis de armazenamento;
- $ESTADO = \{rel(a, b) \mid a, b \in EXP\}$ é o conjunto de variáveis indicadoras de estado, definidas através duma relação entre duas expressões do conjunto EXP.

O meta-modelo FSMD adequa-se tanto a sistemas predominantemente de controlo como a sistemas predominantemente de fluxo de dados, mas não tem potencialidade para modelar sistemas complexos e/ou concorrentes. Como não se consegue especificar concorrência, a modelação dum sistema com partes concorrentes faz-se com uma única FSMD, o que se traduz numa probabilidade alta de ocorrer explosão do número de estados. Por outro lado, a modelação dum sistema com partes concorrentes através deste meta-modelo, o qual não permite especificar explicitamente hierarquia, introduz frequentemente uma complexidade elevada na função que gera o próximo estado. O meta-modelo FSMD permite trabalhar no nível de abstracção de transferência de registos.

[6] ***F**inite **S**tate **M**achine with **D**ata path*, na terminologia inglesa.

A figura 2.2 ilustra os blocos que compõem uma FSMD genérica.

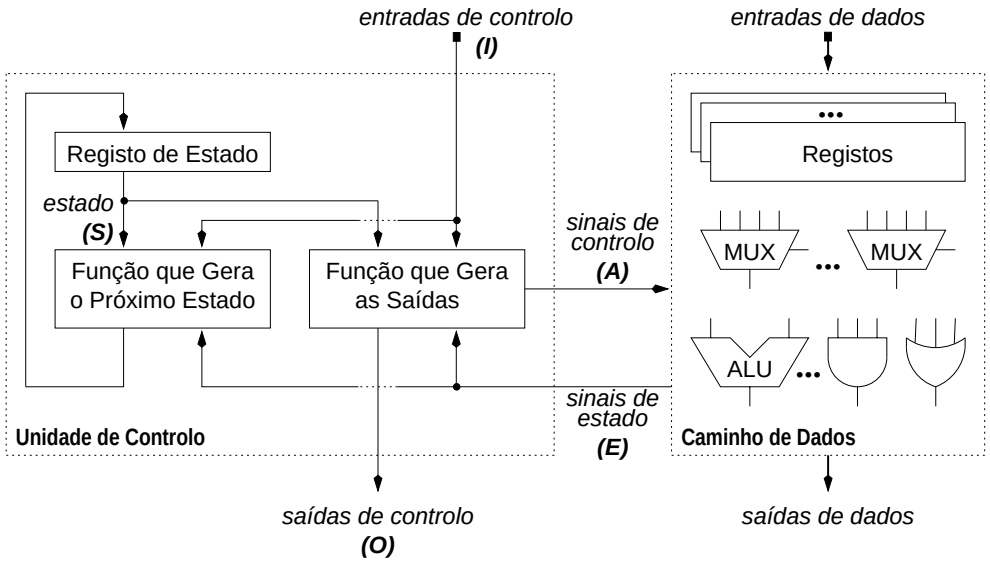


Figura 2.2: Diagrama de blocos duma FSMD genérica.

Máquina de Estados Finitos Concorrente e Hierárquica

A máquina de estados finitos concorrente e hierárquica, ou abreviadamente *HCFSM*[7], resulta da extensão do meta-modelo FSM para suportar hierarquia e concorrência [GVNG94]. Para conseguir esse objectivo permite-se que um estado seja decomposto em sub-estados e que os sub-estados sejam concorrentes. Os sub-estados concorrentes funcionam em paralelo e comunicam por variáveis globais. Como existe hierarquia no meta-modelo, vão existir transições entre estados no mesmo nível hierárquico e transições entre estados em níveis hierárquicos diferentes.

A linguagem gráfica *StateCharts* suporta o meta-modelo HCFSM. Esta linguagem possui os seguintes elementos de composição: (i) um rectângulo, para representar um estado, (ii) um rectângulo encapsulado dentro de outro, para representar hierarquia, (iii) uma linha a dividir um estado, para representar concorrência, (iv) um arco anotado com eventos/condições/acções, para representar transições e (v) um ponto no início dum arco, para representar o ponto de entrada num sub-estado.

Como o meta-modelo HCFSM suporta hierarquia e concorrência, permite representar sistemas de controlo complexos. Contudo, por ser orientado ao estado, não é adequado para representar sistemas que manipulem estruturas de dados complexas ou façam operações com elevada complexidade algorítmica. Este meta-modelo carece também duma formalização, por exemplo para verificar a correcção dos modelos obtidos.

[7] ***H****ierarchical* ***C****oncurrent* ***F****inite* ***S****tate* ***M****achine*, na terminologia inglesa.

Máquina de Estados Finitos para Co-projecto

O meta-modelo *máquina de estados finitos para co-projecto* [CGH+93], ou simplesmente *CFSM*[8], pode ser aplicado na especificação de sistemas, na sua partição em componentes e na obtenção de implementações mistas de *hardware* e *software*. Isto porque as CFSMs tanto podem ser implementadas em *hardware* como em *software*. O meta-modelo está vocacionado para sistemas de controlo embebidos e de tempo real, com baixa complexidade algorítmica.

O formalismo das CFSMs possui uma semântica muito próxima da que se usa nas FSMs, e utiliza como primitiva de comunicação a difusão de eventos. Um modelo com CFSMs será composto por uma rede interactiva de CFSMs, comunicando através de eventos. Os eventos são emitidos por uma CFSM e serão detectados por uma ou mais CFSMs, havendo portanto comunicação por difusão. Os eventos implementam um protocolo síncrono entre interlocutores, logo um protocolo que não necessita de sinal de aceitação. O meta-modelo temporal usado é discreto, uma vez que os elementos computacionais demoram um tempo não nulo a executar as tarefas, e não é puramente reactivo porque o emissor dum evento não o desactiva a não ser quando emite outro.

Os modelos CFSM são descritos numa linguagem de mais alto nível, como por exemplo VHDL ou Esterel.

Uma CFSM, tal como uma FSM, transforma um conjunto de entradas num conjunto de saídas, usando uma quantidade finita de estado interno. O comportamento dum sistema discreto e de estado finito pode ser descrito com sendo uma interligação de objectos CFSM. A diferença entre CFSMs e FSMs reside na não imposição de sincronismo nas CFSMs. Uma CFSM é constituída basicamente por um conjunto de pares {evento de entrada, conjunto de valores}, um conjunto de trios {evento de saída, conjunto de valores, valor inicial} e uma relação de transição. A relação de transição descreve como é que os eventos de entrada geram os eventos de saída.

A transição das CFSMs é não determinística porque o tempo de reacção não é conhecido e os eventos de saída podem não ter uma definição única. O segundo motivo traz abstracção ao modelo em causa.

Uma CFSM possui dois tipos básicos de eventos de entrada: gatilhadores e de valor puro. Os primeiros são utilizados para gerar as transições duma CFSM, funcionando como mecanismo de sincronismo. Os segundos permitem escolher entre um conjunto de possibilidades para o mesmo conjunto de eventos gatilhadores.

A definição formal duma CFSM é dada pelo seguinte quinteto

[8] ***C**odesign **F**inite **S**tate **M**achine*, na terminologia inglesa.

$$C = < I, E, O, R, F > \tag{2.3}$$

em que

- ***I*** é um conjunto finito de pares *{nome de evento entrada, conjunto de valores}*;
- ***E*** é um sub-conjunto de I ($E \subseteq I$) e um conjunto de nomes de eventos gatilhadores;
- ***O*** é equivalente a I para eventos de saída ($E \cap O = \oslash$);
- ***R*** é um conjunto de valores iniciais dos eventos de saída;
- ***F*** é a relação de transformação, que pode ser definida como o produto cartesiano entre os conjuntos de valores de entrada e de saída.

Cada CFSM descreve um componente do sistema a modelar, enquanto o sistema completo é descrito por uma rede de CFSMs interactivas. Uma rede de CFSMs é um conjunto de CFSMs, em que os conjuntos de eventos de saída são disjuntos. Esta característica evita a necessidade do mecanismo de exclusão mútua. Por outro lado, os conjuntos de eventos de entrada das diferentes CFSMs da rede não precisam de ser disjuntos, exigindo um mecanismo de comunicação por difusão.

Uma CFSM não deve ignorar (deve reagir a) um evento quando surge um segundo evento sem ela ter reagido ao primeiro. Um evento de entrada pode causar infinitas transições numa CFSM, mas as transições também podem nunca ocorrer, e a CFSM nunca reagir a esse evento. O que é imposto é que se um dos eventos de saída associado com uma transição for emitido, então todos os eventos de saída têm que ser emitidos.

Prova-se que é possível representar o comportamento assíncrono duma rede de CFSMs através do modelo síncrono duma rede de FSMs porque existe uma equivalência comportamental entre os dois tipos de rede. Isto é importante na síntese porque garante a consistência entre as especificações e um conjunto de opções de implementação. O interesse da conversão duma rede de CFSMs noutra de FSMs reside na possibilidade de fazer verificação formal das propriedades da especificação, usando ferramentas já existentes e baseadas em FSMs. A outra razão é a possibilidade de se provar a correcção de implementações que misturam *hardware* e *software*, modelando directamente o *hardware* segundo um conjunto de FSMs cooperantes e alguns componentes de *software*.

Em relação ao tipo de sistemas a que se destina (sistemas embebidos predominantemente de controlo e de tempo real, com baixa complexidade algorítmica), o meta-modelo CFSM apresenta algumas vantagens sobre outros meta-modelos também passíveis de ser usados com

estes sistemas: (i) a CFSM resolve uma das limitações apresentadas pela FSM na modelação de sistemas com *software* e *hardware*, e que resulta do facto de todas as FSMs dum módulo do sistema mudarem de estado e produzirem valores de saída simultaneamente; (ii) a CFSM utiliza uma noção de temporização determinística, o que não acontece no meta-modelo da linguagem Esterel ou nas redes de Petri; (iii) as redes de Petri podem resultar num número infinito de estados e não impõem grandes limitações sintácticas às redes a implementar; além disso, só modelam directamente a causalidade AND (acção desencadeada pela conjunção de condições) e não a causalidade OR; (iv) as linguagens síncronas, como Esterel, requerem um tempo de reacção nulo, o que não é realístico para o tipo de sistema em causa; mais, os modelos descritos nestas linguagens são implementados ineficientemente em *hardware*.

A CFSM resultou assim duma tentativa de agregar as melhores características da FSM (estados em número finito, possibilidade de se fazer verificação), das redes de Petri e das linguagens de programação síncronas (atrasos não limitados e semântica reactiva).

Redes de Petri

Os elementos de composição disponíveis no meta-modelo redes de Petri (RdP) são o lugar, a transição e a marca. As marcas estão associadas a lugares, podem circular pela rede e são produzidas e consumidas quando as transições disparam [Pet81]. A definição formal de RdP é dada por

$$< L, T, I, O, m > \tag{2.4}$$

em que

- $L = \{l_1, l_2, ..., l_m\}$ é o conjunto de lugares;
- $T = \{t_1, t_2, ..., t_m\}$ é o conjunto de transições, em que L e T são disjuntos;
- $I : T \rightarrow L^+$ é a função que define o conjunto dos lugares que funciona como entrada para cada transição; L^+ designa um conjunto com um ou mais lugares l_i;
- $O : T \rightarrow L^+$ é a função que define o conjunto dos lugares que funciona como saída de cada transição;
- $m : L \rightarrow \mathcal{N}$ é a função que define o número de marcas em cada lugar, sendo $\mathcal{N}$ o conjunto dos números inteiros não negativos.

O conjunto das funções I mais o conjunto das funções O define todas as interligações existentes na rede de Petri. Tanto a função I como a função O contêm pelo menos um elemento por cada transição.

O estado da rede evolui quando uma ou mais transições disparam. Para que uma transição t_n dispare é necessário que todos os lugares pertencente à sua função $I(t_n)$ possuam pelo menos uma marca. Uma transição t_n termina o seu disparo quando retirar dos lugares pertencente à sua função $I(t_n)$ todas as marcas e colocar uma marca em cada um dos lugares pertencentes à sua função $O(t_n)$.

Uma das vantagens do meta-modelo RdP é que consegue representar um leque alargado de propriedades dos sistemas a modelar, tais como sequenciação, ramificação não determinística, sincronização, contenção e concorrência. Outra vantagem do meta-modelo é que permite verificar e validar formalmente um conjunto de propriedades, entre as quais se incluem a segurança e a ausência de bloqueios. Como principais desvantagens apontam-se o facto de gerar modelos incompreensíveis, quando aplicado a sistemas complexos e não conseguir modelar o caminho de dados.

2.3.2 Meta-modelos Orientados à Actividade

Grafo de Fluxo de Dados e Grafo de Fluxo de Controlo

O meta-modelo *grafo de fluxo de dados*, ou simplesmente *DFG*, é do tipo orientado à actividade. O DFG adequa-se a sistemas em que as saídas são obtidas pela execução de um conjunto de transformações sobre as entradas, não contendo o conceito de estado ou de evento.

Um DFG é composto por um conjunto de nodos e um conjunto de arcos que interligam estes nodos. Os tipos de nodo disponíveis num DFG diferem duma variante do meta-modelo para a outra, mas em relação à versão minimalista pode dizer-se que um DFG define os seguintes tipos de nodo: nodos que representam a entrada ou saída de informação do sistema, nodos operacionais que representam manipulação da informação e nodos de armazenamento da informação. O meta-modelo é flexível na questão da granulosidade dos nodos operacionais, permitindo a definição de nodos que representam uma simples operação lógica/aritmética e nodos que representam uma função ou procedimento. Os nodos de armazenamento também se adaptam a situações muito diferentes, podendo representar um ficheiro, uma variável em memória ou um registo. Os arcos são direccionados de modo a indicar o fluxo da informação, e podem incluir uma etiqueta que define o tipo de dados que representam. Os arcos também podem ser vistos como dependências de dados impostas às operações a realizar pelos nodos. Estas dependências traduzem-se na circulação de *tokens* pelos arcos que habilitam ou inibem as operações.

O DFG permite hierarquia uma vez que um nodo pode ser descrito por um sub-grafo do mesmo tipo, o que faz com possa ser aplicado em sistemas de elevada complexidade algorítmica. O meta-modelo DFG minimalista apenas permite expressar as dependências que resultam da

própria estrutura do grafo. Como não consegue representar outro tipo de dependências, tais como as impostas por requisitos de ordem temporal, o meta-modelo não é o mais adequado para especificar sistema embebidos.

Como o meta-modelo permite jogar com a granulosidade dos nodos, pode optar-se por (i) um grafo de grão fino, que permite uma boa optimização na implementação mas que não é recomendável para modelar sistemas complexos ou (ii) um grafo de grão grosso, que embora não facilite a obtenção de implementações optimizadas, permite modelar sistemas complexos.

O meta-modelo *grafo de fluxo de controlo* (*CFG*[9]) é idêntico ao DFG, residindo a maior diferença na semântica dos arcos. Neste meta-modelo, os arcos impõem aos nodos uma determinada ordem de execução, ou seja, representam o fluxo de controlo. O CFG é equivalente ao meta-modelo subjacente aos fluxogramas comuns nas linguagens de programação para *software*. Por isso, os tipos de nodo permitidos no CFG são os nodos de arranque e de paragem do fluxo de controlo, o nodo que efectua operações, essencialmente sob a forma de atribuições, e o nodo de decisão, que possibilita a ramificação do fluxo de controlo em várias alternativas.

2.3.3 Meta-modelos Orientados à Estrutura

Os meta-modelos *orientados à estrutura* descrevem a estrutura física dum sistema, razão pela qual se destinam a fases avançadas do processo de desenvolvimento dos sistemas. Os elementos que compõem um modelo são (i) nodos, representando componentes do sistema com correspondência física e (ii) arcos, traduzidos em interligações entre os componentes. Este tipo de meta-modelo permite trabalhar em diferentes níveis de abstracção, consoante o tipo de objectos que corresponde aos nodos e aos arcos do meta-modelo. Deste modo, se os nodos e os arcos forem elementos definidos ao nível do sistema (tais como processador, memória, ASIC, programa ou barramento do sistema) obtém-se um diagrama de blocos ao nível do sistema, mas se forem elementos definidos ao nível de transferência de registos (tais como registo, ALU, barramento ou multiplexador) o resultado é um esquemático no nível de transferência de registos. Quanto mais baixo for o nível de abstracção do modelo, mais pormenores de implementação ele inclui.

2.3.4 Meta-modelos Orientados ao Dado

Os meta-modelos *orientados ao dado* são significativamente distintos dos modelos orientados ao estado ou à actividade, porque enquanto os últimos modelam as operações que actuam sobre a informação, os primeiros concentram-se na representação da própria informação. Um exemplo de meta-modelo orientado aos dados é o *diagrama de relacionamento entre entida-*

[9] ***C**ontrol **F**low **G**raph*, na terminologia inglesa.

des, que usa como elementos de composição um conjunto de entidades e um conjunto de relações entre entidades. Este meta-modelo permite obter uma boa representação da informação de todo um sistema, sendo por isso adequado para modelar sistemas em que se exige uma organização das complexas relações entre vários tipos de informação.

Quando se pretende modelar sistemas embebidos, estes meta-modelos são de difícil aplicação, porque não modelam directamente nem a funcionalidade lógica nem a temporal dos sistemas. A área prioritária de aplicação destes meta-modelos é o desenvolvimento de sistemas de informação, como aqueles que operam sobre bases de dados.

2.3.5 Meta-modelos Heterogéneos

Grafo de Acesso *SLIF*

O formato *SLIF* [10], com o meta-modelo associado, define um grafo de acesso com nodos e arcos interligados [VG95c]. Os nodos representam variáveis, processos, procedimentos ou blocos de declarações, e os arcos direccionados representam canais de comunicação entre nodos. A granulosidade dos nodos não é explicita no modelo. Os canais correspondem à chamada duma função, à leitura/escrita duma variável/porto ou à passagem duma mensagem. Neste formato, as diferentes utilizações do mesmo procedimento são, por exemplo, representadas pelo mesmo nodo, o que não impede que em certos casos se opte por replicar a implementação desse nodo. O formato SLIF suporta a representação de informação estrutural relativa ao comprometimento da descrição com uma dada implementação. Esta facilidade consiste em associar os nodos aos componentes da arquitectura alvo, como por exemplos os processadores, os ASICs ou os dispositivos de lógica programável, e associar os canais a barramentos ou a outros recursos de interligação da arquitectura. O facto de o formato representar além da funcionalidade, informação estrutural, permite que a concepção decorra segundo um processo de refinamento iterativo. Um modelo SLIF pode ser anotado com informação obtida por estimação: (i) nos canais anota-se o número de vezes que em média esse canal é utilizado e o tamanho dos dados trocados; (ii) nos nodos anota-se o tempo de execução das tarefas que lhe estão atribuídas ou o tempo de leitura (escrita) desse (nesse) nodo; (iii) os barramentos seleccionados são anotados com o tempo de comunicação entre os componentes por eles interligados.

Meta-modelos do tipo SLIF têm a vantagem de serem adequadamente representados através de HLLs ou HDLs. Outras vantagens do meta-modelo SLIF, em relação a meta-modelos de grão fino, são: (i) consegue descrever os sistemas num nível de abstracção mais elevado, (ii) utiliza muito menos objectos, reduzindo assim o tempo de cálculo necessário para obter uma implementação e melhorando a interactividade com o projectista, (iii) permite obter

[10] ***S**pecification-**L**evel **I**ntermediate **F**ormat*, na terminologia inglesa.

soluções através dum refinamento iterativo e (iv) possui um suporte mais adequado para manusear as estimativas de métricas e explorar o espaço de projecto. A última vantagem ocorre porque quando se usam nodos de grão fino, por exemplo ao nível da operação, a estimativa para um conjunto de nodos a atribuir a um componente da arquitectura, não pode ser obtida por uma simples soma da estimativa para os diferentes nodos incluídos no conjunto. Isto porque na implementação ocorrem partilhas de recursos, por parte de nodos distintos, que dificilmente podem ser previstas nas etapas iniciais da concepção, dado que o modelo do sistema contém muitos nodos.

Entre as limitações do meta-modelo SLIF destacam-se: (i) como não representa operações com uma granulosidade fina, não é adequado para um escalonamento fino das operações, (ii) não é possível simular um modelo SLIF, (iii) a estimativa para um conjunto de nodos, obtida por uma simples soma da estimativa para os diferentes nodos, não é totalmente correcta e (iv) não suporta procedimentos do tipo *forked*.

Grafo de Fluxo de Dados e de Controlo

Um *grafo de fluxo de dados e de controlo*, ou *CDFG*, combina no mesmo modelo um grafo de fluxo de controlo e vários grafos de fluxo de dados. O CFG descreve a sequência das actividades de todo o sistema e o fluxo de informação para cada nodo que executa operações no CFG é representado por um DFG. Cada nodo de operação do CFG está ligado ao DFG correspondente. A figura 2.3 ilustra como é que um pequeno exemplo, descrito em linguagem C, é representado por um CDFG. Existem outras variantes do meta-modelo CDFG, incluindo CDFGs que ao invés de estarem centrados num CFG (como acontece no CDFG apresentado), centram-se num DFG.

O meta-modelo CDFG é adequado para modelar sistemas de tempo real, sendo uma das representações internas preferenciais do processo de partição. Como inconveniente pode apontar-se o facto de não representar de forma estruturalmente unificada os fluxos de controlo e de dados.

Meta-modelos das Linguagens de Programação

Os meta-modelos que suportam as linguagens de programação mais divulgadas permitem modelar a informação, o controlo e as actividades dos sistemas. A informação é modelada por estruturas de dados, sendo os tipos normalmente permitidos o inteiro, o real, o *array* e as estruturas compostas por tipos de dados mais simples. A modelação do fluxo de controlo define a ordem de execução das actividades, através de construtores de sequenciação, de ramificação, ciclos ou invocação de funções/procedimentos. Por seu lado, as actividades são modeladas por declarações, funções ou procedimentos.

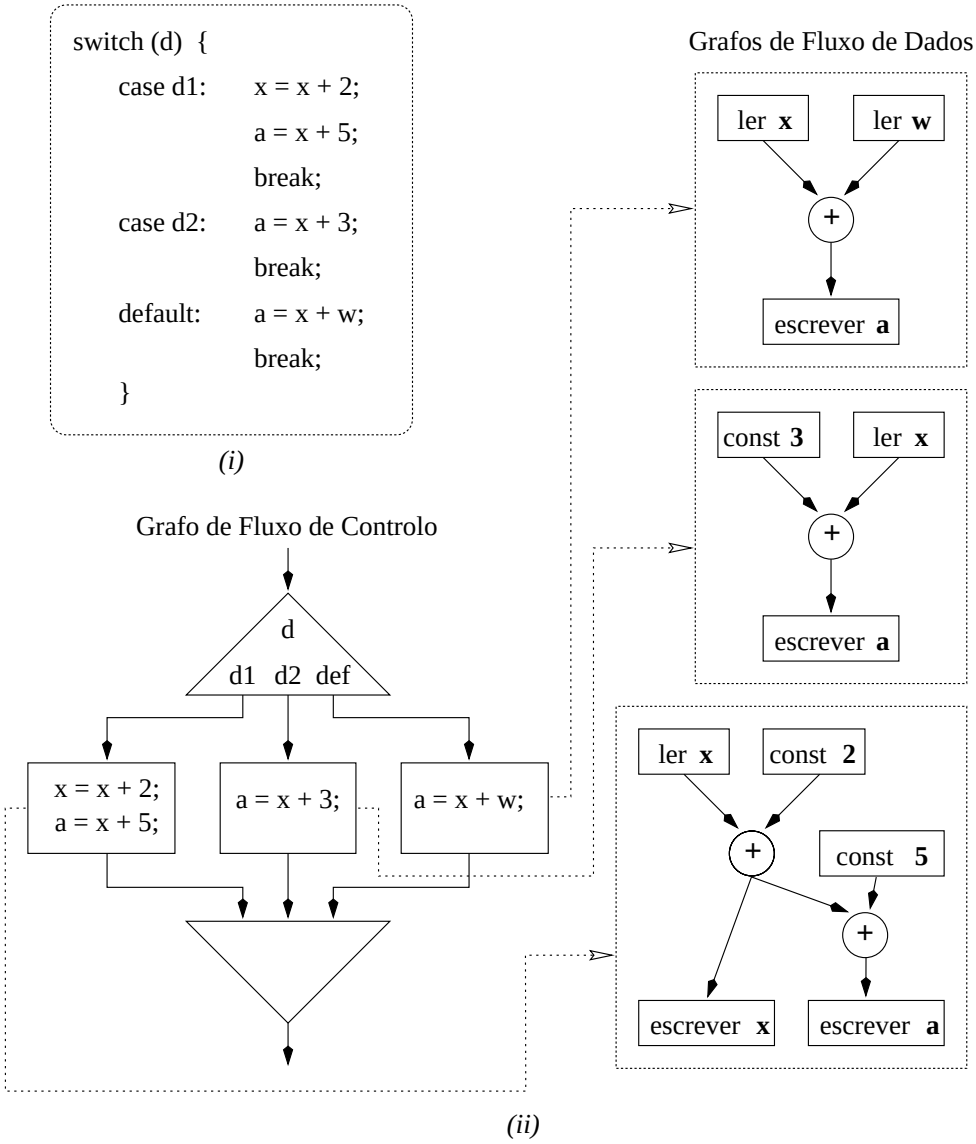


Figura 2.3: Exemplo dum modelo CDFG: da descrição em C (i) à representação em CDFG (ii).

Os meta-modelos das linguagens de programação não estão vocacionados para serem representados de forma gráfica, mas sim de forma textual.

A maioria das HLLs pertence a uma de duas classes: a classe das linguagens imperativas (C, Pascal, Fortran, C++, Java) ou a classe das linguagens declarativas. As linguagens declarativas podem ainda ser sub-classificadas como funcionais (LISP, Haskell, ML) ou lógicas (PROLOG). As linguagens imperativas definem de forma explícita a ordem de execução das actividades, enquanto as declarativas centram a atenção do utilizador na definição dos alvos das actividades. As primeiras apresentam a vantagem de poderem modelar sistemas essencialmente de fluxo de dados, em que as actividades são executadas por algoritmos de elevada complexidade. Como a maioria das linguagens de programação orientadas para implementações em *software* não permitia expressar actividades concorrentes, surgiram linguagens baseadas em meta-modelos que suportam a execução de actividades de forma concorrente. Entre

estas linguagens incluem-se HLLs e HDLs, tais como CSP, ADA, VHDL ou Verilog.

As principais limitações que as HLLs colocam à modelação de sistemas embebidos derivam do facto de não conseguirem representar de forma explicita o estado e o comportamento temporal dos sistemas.

Máquina com Estados Programa

O meta-modelo *máquina com estados programa*, ou simplesmente *PSM*, combina o meta-modelo HCFSM com o meta-modelo das linguagens de programação. Segundo este meta-modelo, um sistema é representado por um conjunto hierarquizado de *estados programa* e pelas variáveis por eles acedidas. Os estados programa representam unidades de computação, que podem estar activas ou inactivas em determinado momento [GVN94].

Um sistema pode incluir estados programa do tipo composto (E_1 e $E_{1,1}$ na figura 2.4) ou do tipo terminal/folha ($E_{1,1,1}$, $E_{1,1,2}$ e $E_{1,2}$). Os estados programa compostos são definidos por um conjunto de estados programa concorrentes ($E_{1,1}$ e $E_{1,2}$) ou sequenciais ($E_{1,1,1}$ e $E_{1,1,2}$), e os estados programa terminais são definidos por um bloco de código numa linguagem HLL ou HDL. Se os estados programa são concorrentes, estão todos activos ao mesmo tempo, mas se forem sequenciais apenas um está activo em cada instante.

Para um estado programa composto, a sequência pela qual os seus estados sequenciais entram em actividade é definida por arcos direccionados que ligam esses estados. Os arcos direccionados podem ser de dois tipos: (i) arcos que representam a ocorrência duma transição quando se verificam simultaneamente as seguintes condições: a actividade do estado terminou e a condição associada ao arco ficou verdadeira (arcos A_{tf}) e (ii) arcos que representam a ocorrência duma transição imediatamente após a condição que lhes está associada ficar verdadeira (arcos A_{ti}). Uma transição num arco direccionado significa que o estado alvo desse arco ficará activo.

Para representar graficamente os sistemas, o meta-modelo PSM possui os seguintes elementos de composição (figura 2.4): caixa rectangular para representar a entidade sistema, caixa rectangular de cantos arredondados para representar um estado programa, linha ponteada para identificar dois estados programa que são concorrentes entre si, arco direccionado para indicar uma transição, triângulo para identificar o estado programa que entra primeiro em actividade, ponto rectangular no início dum arco direccionado para definir um arco do tipo A_{tf}, ponto rectangular (simultaneamente no fim dum arco direccionado e dentro dum estado) para identificar o fim da actividade do estado em que está incluído o ponto, bloco de código HLL/HDL[11] para definir a funcionalidade dos estados terminais e as propriedades das variá-

[11] No exemplo da figura 2.4, os estados terminais são especificados com a linguagem C.

veis. As variáveis também podem ser representadas de forma gráfica através de um hexágono. Os arcos A_{ti} partem da periferia dum estado (arco $a2$ na figura 2.4), enquanto os arcos A_{tf} partem dum ponto rectangular dentro dum estado (arcos $a1$, $a3$, $a4$ e $a5$).

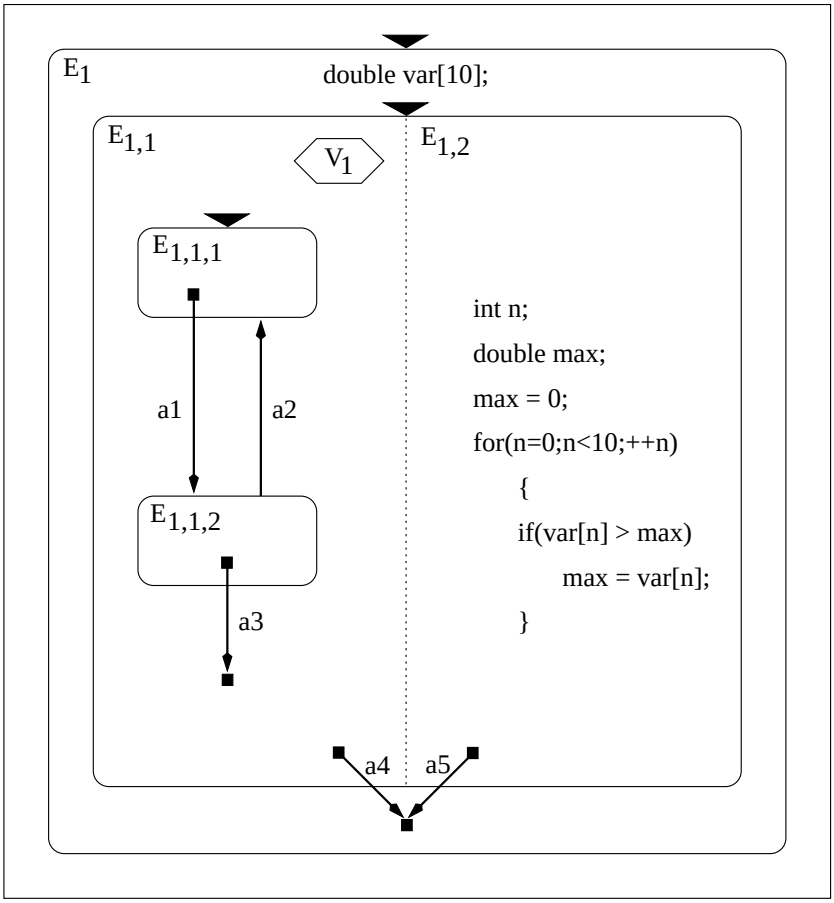


Figura 2.4: Exemplo dum modelo PSM.

No limite, um modelo PSM será uma HCFSM se os estados terminais não contiverem código HLL/HDL e será um programa numa linguagem de programação se o modelo incluir apenas um estado programa.

Como o meta-modelo PSM consegue representar num único modelo o estado, a informação e a actividade, é mais adequado que o meta-modelo HCFSM para representar sistemas em que os estados possuem dados e actividade complexos. Em relação ao meta-modelo das linguagens de programação, o PSM apresenta a vantagem de permitir representar explicitamente o estado do sistema.

Meta-modelo Orientado ao Objecto

O meta-modelo orientado ao objecto (OO) pode ser encarado como uma evolução do meta-modelo orientado aos dados, onde um sistema é representado por um conjunto de objectos e por um conjunto de transformações que ajudam a definir a evolução temporal do sistema.

Cada objecto é composto por um conjunto de dados e por um conjunto de métodos que operam sobre os dados deste objecto. Deste modo, o meta-modelo OO dum sistema S é dado por

$$S = < O, T > \tag{2.5}$$

em que

- $O = \{o_1, o_2, ..., o_m\}$ é o conjunto de objectos;
- $T = \{t_1, t_2, ..., t_n\}$ é o conjunto de transformações que ajudam a definir a evolução temporal do sistema;
- cada objecto $o_i =< D_i, M_i >$ é composto pelo conjunto de dados $D_i = \{d_{i,1}, d_{i,2}, ..., d_{i,p}\}$ e pelo conjunto de métodos $M_i = \{m_{i,1}, m_{i,2}, ..., m_{i,q}\}$.

Entre as principais características do meta-modelo OO incluem-se: (i) representa os sistemas de forma idêntica à sua maneira de funcionar, (ii) permite uma abstracção de dados elevada, (iii) possui a capacidade de encapsular nos objectos a funcionalidade que não precisa ser vista do seu exterior, (iv) os objectos suportam de forma natural a concorrência entre actividades e (v) os objectos comunicam por mensagens que representam um pedido de execução de métodos do objecto alvo da comunicação; o objecto que inicia a comunicação espera que o objecto alvo termine a execução dos métodos requisitados [Boo94].

A principal limitação do meta-modelo para representar sistemas embebidos, reside no facto de ao modelar sistemas que exibam um comportamento temporal complexo (descrito por T na equação 2.5), ser necessário recorrer a outro meta-modelo para o definir. Por outro lado, o elevado nível de abstracção das descrições faz com que o meta-modelo seja mais adequado para as fases de análise e concepção, do que para a fase de implementação.

2.3.6 Resumo das Alternativas de Modelação

A tabela 2.1 sintetiza as características relevantes para a modelação de sistemas embebidos, disponíveis em cada um dos meta-modelos introduzidos. Com as notas da tabela 2.1, em conjunto com os próximos parágrafos, procura explicar-se as opções de classificação menos consensuais.

As linguagens de programação de alto nível como C, Pascal ou Fortran não possuem concorrência explícita. A linguagem C++ define actividades concorrentes à custa de bibliotecas adicionais e no caso da linguagem Java pode dizer-se que apresenta concorrência ao considerar-

se a classe `thread` como parte integrante da linguagem. Já a linguagem Occam possui um construtor (`PAR`) para definir explicitamente actividades concorrentes.

META-MODELO	Nível de Abstracção	Hierarquia Estrutural	Hierarquia Funcional	Concorrência	Definição de Estados	Informação Temporal	Complexidade de Dados	Informação de Controlo	Representação Gráfica	Granulosidade Variável	Implementação Preferencial
FSM	baixo	×	×	×	✓	×	baixa	✓	✓	×	h/w
FSMD	baixo	×	×	×	✓	×	média	✓	✓	×	h/w
HCFSM	baixo	×	✓	✓	✓	×	baixa média	✓	✓	×	h/w
CFSM	baixo	×	×	✓ (1)	✓	×	baixa	✓	✓	×	h/w
RdP	baixo	✓	✓ (2)	✓	✓	× (2)	baixa média (2)	✓	✓	×	h/w,s/w
DFG	baixo	×	✓ (2)	✓ (2)	×	× (2)	média	×	✓	✓	h/w,s/w
CFG	baixo	×	✓ (2)	✓ (2)	×	× (2)	baixa	✓	✓	✓	h/w,s/w
grafo SLIF	alto	× (3)	× (3)	✓	×	✓	média alta	× (4)	✓	✓	h/w,s/w
CDFG	baixo	×	✓ (2)	✓ (2)	×	× (2)	média	✓	✓	✓	h/w,s/w
HLLs	alto	×	✓	(5)	× (6)	×	alta	✓	×	×	s/w
PSM	alto	×	✓	✓	✓	×	média, alta (7)	✓	✓	✓	h/w,s/w
OO	alto	✓	✓	✓	×	✓ × (8)	alta	✓	✓ (9)	✓ (10)	s/w

(1) O meta-modelo CFSM possui concorrência dado que um modelo CFSM é composto por uma rede de CFSMs interligadas, podendo em cada instante haver várias CFSMs activas;
(2) Embora o meta-modelo base não apresente esta característica, ela está presente em versões estendidas do meta-modelo base;
(3) O facto do grafo SLIF permitir utilizar nodos com granulosidade grossa compensa a falta de hierarquia no meta-modelo;
(4) O grafo de acesso SLIF não está vocacionado para representar sistemas predominantemente de controlo porque não dispõe do conceito de estado nem de elementos de composição para definir explicitamente o fluxo de controlo;
(5) O suporte para especificar concorrência varia de linguagem para linguagem;
(6) Embora permitam especificar a funcionalidade duma máquina de estados, a maior parte das HLLs não possui suporte explícito para definir estados;
(7) Os tipos de dados permitidos pelo meta-modelo PSM dependem da HLL ou HDL seleccionada para especificar a funcionalidade dos estados programa terminais;
(8) O suporte dos meta-modelos OO para especificar informação temporal varia com a instanciação do meta-modelo;
(9) A notação UML permite a representação gráfica de (vistas dos) sistemas;
(10) Um meta-modelo OO apresenta um suporte parcial para a variação da granulosidade dos objectos.

Tabela 2.1: Características relevantes para a modelação de sistemas embebidos.

A maioria dos meta-modelos discutidos não foi originalmente concebida para representar a informação temporal, mas pode ser estendida de modo a incluir esta potencialidade. Por exemplo, a informação temporal pode ser incluída nos meta-modelos do tipo grafo de fluxo (DFG, CFG ou CDFG) sob a forma de arcos que em vez de representarem o fluxo de dados ou de controlo, estabelecem um requisito temporal entre os dois nodos por eles interligados.

Algumas metodologias de desenvolvimento OO usam meta-modelos ou notações (por exemplo o UML) que suportam a especificação de informação temporal. Outras metodologias, ou não representam informação temporal ou só o permitem sob a forma de requisitos não funcionais, como acontece no MOOSE.

Os meta-modelos OO apresentam um suporte parcial para a variação da granulosidade a utilizar na implementação dos sistemas, uma vez que nesta fase apenas se pode usar como grão o objecto ou o agrupamento de objectos, e nunca um grão mais fino que o objecto. Dado que os objectos definidos durante as fases anteriores do desenvolvimento apresentam normalmente um grão grosso, o espaço de projecto passível de ser analisado é menor. Como consequência, as possibilidades de obter uma solução optimizada são menores.

2.3.7 Linguagens para a Concretização de Meta-modelos

Entre as linguagens mais utilizadas para especificar sistemas embebidos, à luz de determinado meta-modelo, incluem-se:

- linguagens formais vocacionadas para descrever sistemas de telecomunicações, como por exemplo LOTOS [EVD89], SDL [DMVJ97] ou Estele [ISO87];
- linguagens orientadas para a modelação de sistemas tempo real, como por exemplo Esterel [BG92], StateCharts [Har87] ou linguagens baseadas em redes de Petri (estendidas com suporte adequado para representar o caminho de dados) [Ess96][MFP98];
- linguagens orientadas para a descrição de *hardware* (HDLs), como por exemplo VHDL [IEE93], Verilog [IEE96], HardwareC [KdM88] ou SpecCharts [NVG92];
- linguagens de programação estendidas, como por exemplo C^x (uma extensão da linguagem C) [EHB93], SpecC (linguagem tipo C que possui características das HDLs) [ZDG97][GZD$^+$00] ou SystemC (linguagem que combina as características do C++ com as das HDLs) [SVDH01];
- linguagens adequadas para modelar sistemas a implementar com arquitecturas multiprocessador e que exploram o paralelismo da implementação, como por exemplo o CSP [Hoa78] e o Occam [Inm84], que são linguagens baseadas na álgebra de processos;
- linguagens adequadas para modelar sistemas predominantemente de fluxo de dados, tais como os processadores digitais de sinal (DSPs); Silage é um exemplo deste tipo de linguagem [Hil85].

2.4 Selecção do Tipo de Modelação

Considerando que os sistemas a implementar no presente trabalho são sistemas embebidos, com requisitos de desempenho e complexidade média, em especial ao nível dos dados manipulados, apresenta-se a seguir a justificação por que se escolheu o meta-modelo PSM para descrever os sistemas à entrada para o processo de partição.

O meta-modelo FSM só é adequado para modelar sistemas (ou a vista de sistemas) de baixa complexidade e predominantemente de controlo; logo nunca poderia representar a globalidade dum sistema do tipo citado. Embora o meta-modelo FSMD permita modelar sistemas predominantemente de controlo e de fluxo de dados, também não é uma boa solução para modelar sistemas complexos porque não suporta concorrência nem hierarquia. Ao suportar concorrência e hierarquia, o meta-modelo HCFSM consegue representar sistemas complexos, mas tal como outros meta-modelos baseados em FSM não está vocacionado para manusear estruturas de dados nem operações sobre dados complexos. O meta-modelo CFSM está orientado para modelar sistemas de controlo reactivos de baixa complexidade computacional, apresentando assim as mesmas limitações que o meta-modelo FSM.

A boa formalização do meta-modelo RdP permite que se validem diversas propriedades dos modelos, contudo os modelos de sistemas complexos são pouco compreensíveis e com a RdP original apenas se consegue representar a parte de controlo dos sistemas. Deste modo, uma alternativa para a modelação seria uma RdP estendida com a capacidade de representar a parte de dados dos sistemas e a informação temporal necessária à obtenção duma implementação de acordo com os requisitos desses sistemas. A rede de Petri de alto nível [Jen85] e a rede de Petri colorida [KCJ98] estendem o meta-modelo base com suporte para modelar o caminho de dados.

O DFG está vocacionado para sistemas predominantemente de fluxo de dados, mas carece duma formalização tão sólida quanto a das RdP. Sem extensões ao meta-modelo não se consegue descrever o comportamento temporal dos sistemas, e sem macro-nodos não existe hierarquia, o que implica que a complexidade não é tratada convenientemente. A possibilidade de se poder variar a granulosidade dos nodos é uma característica útil ao processo de partição. O meta-modelo CFG é idêntico ao DFG, mas está orientado para sistemas predominantemente de controlo.

O meta-modelo SLIF representa convenientemente a arquitectura dos sistemas mas não é especialmente adequado para descrever a funcionalidade desses sistemas. O grafo SLIF foi pensado para facilitar a execução de tarefas da síntese de alto nível, com destaque para a partição em componentes de *hardware* e/ou de *software*. No entanto, ao representar a funcionalidade dos sistemas com uma granulosidade grossa, o escalonamento fino das operações

não pode ser obtido a partir duma representação em SLIF.

Como descreve a funcionalidade dos sistemas usando uma granulosidade fina e num nível de abstracção baixo, o meta-modelo CDFG facilita as tarefas da síntese de alto nível (partição, escalonamento e selecção de recursos), sendo por isso atractivo para a fase de implementação dos sistemas. Das características anteriores resulta também uma limitação do CDFG, traduzida na inadequação do meta-modelo para a fase de concepção. Por outro lado, se o CDFG não for estendido de modo a suportar hierarquia ou granulosidade variável, mostra-se inadequado para representar sistemas complexos. Ao meta-modelo falta ainda uma boa formalização que permita verificar a correcção das representações e das transformações sobre elas efectuadas.

As linguagens de programação permitem mais do que um meta-modelo, mas para representarem convenientemente os sistemas embebidos à entrada para o processo de partição deverão incluir concorrência, suportar o conceito de estado, se possível possuir uma representação gráfica e funcionar num nível de abstracção que possibilite a obtenção de implementações optimizadas.

O meta-modelo orientado ao objecto tem uma forte implantação no desenvolvimento de sistemas puramente de *software*, tendo contudo ocorrido migrações para ambientes de co-projecto de *hardware* e de *software*. Isto porque o meta-modelo apresenta características que o tornam atraente também neste caso, ao permitir modelar sistemas embebidos complexos que serão implementados com *hardware* e *software*. Entre os trunfos do meta-modelo incluem-se a hierarquia, a concorrência, a capacidade para manusear a complexidade e a definição de modelos bastante abstractos. O facto de se obterem modelos bastante abstractos resulta em vantagens e desvantagens. Por um lado, facilita a concepção e não se prende um modelo com uma implementação particular (permitindo assim a reutilização do modelo), mas em contrapartida o mais provável é que a implementação seja menos optimizada do que seria se o modelo fosse menos abstracto. O meta-modelo OO clássico tem limitações na definição do comportamento temporal dos sistemas e na definição do seu estado.

Combinando o meta-modelo duma HLL/HDL com HCFSM obteve-se o meta-modelo PSM, que suporta os sistemas embebidos complexos de uma forma superior. Isto deve-se ao facto do PSM incluir as melhores características de ambos os meta-modelos que o originaram, entre as quais se incluem hierarquia funcional, concorrência, conceito de estado, potencialidades para manusear a complexidade (algorítmica e dos dados), possibilidade de indicar o fim dos blocos funcionais (*behaviour completion*), possibilidade de incluir tratamento de excepções e uma representação gráfica. Além disto a modelação com PSM é bastante intuitiva. Entre as maiores limitações do PSM incluem-se a ausência de hierarquia estrutural e de suporte automático para validar formalmente os modelos. No presente trabalho seleccionou-se a linguagem

VHDL para descrever as variáveis e os estados programa terminais. Ao optar-se por VHDL, é possível modelar explícita e elegantemente a comunicação e sincronização entre actividades concorrentes.

2.5 Conclusões

Neste capítulo apresentaram-se diversos meta-modelos para sistemas digitais. Os meta-modelos orientados ao estado estão vocacionados para modelar sistemas predominantemente de controlo, enquanto os meta-modelos orientados à actividade apresentam vantagens na modelação de sistemas predominantemente de fluxo de dados. Os meta-modelos orientados à estrutura destinam-se essencialmente à fase de implementação, uma vez que modelam a estrutura dos sistemas e não a sua funcionalidade. Por seu lado, os meta-modelos orientados aos dados aplicam-se sobretudo em sistema de informação, não permitindo modelar elegantemente determinadas características dos sistemas digitais embebidos, como por exemplo o comportamento temporal. Como os meta-modelos heterogéneos integram características de vários meta-modelos, são uma boa solução para modelar sistemas complexos. Outra alternativa é a modelação dum sistema através de várias vistas, utilizando-se com cada vista o meta-modelo mais apropriado.

A metodologia do MOOSE é um exemplo de modelação multi-vista. Da fase de análise resulta um documento que descreve os requisitos do sistema em linguagem natural. A concepção gera a arquitectura do sistema, através dum processo incremental em que se combinam vários modelos. Estes modelos seguem o meta-modelo mais adequado ao tipo de informação que representam: diagramas hierarquizados que representam a estrutura do sistema e seguem um meta-modelo do tipo DFG, um diagrama que descreve a funcionalidade do sistema segundo um meta-modelo do tipo FSM, um diagrama que descreve as entidades do domínio de aplicação segundo o meta-modelo *initial object model* do OMT e um modelo executável que valida a arquitectura do sistema e utiliza o meta-modelo duma HLL. Na obtenção do modelo executável recorre-se ainda a diversas formas de representação para definir (i) a interface de cada classe, (ii) os objectos e as interligações que implementam cada classe, (iii) os métodos e variáveis que entram na implementação das classes e (iv) a correspondência entre as ligações usadas no modelo funcional e as ligações utilizadas nas classes. A abordagem MOOSE apresenta aspectos interessantes, como sejam a modelação de diversas vistas do sistema com o meta-modelo mais adequado, a adição de forma incremental dos detalhes relativos à implementação, a diversidade de mecanismos de comunicação permitidos e a possibilidade de obter modelos executáveis. Como aspectos negativos podem apontar-se o problema de não se garantir a continuidade entre os diversos modelos e o fraco suporte que dispõe para a fase de implementação, nomeadamente para a partição em componentes.

O meta-modelo escolhido para descrever os sistemas, à entrada para a fase de partição, foi o PSM. A escolha deveu-se à sua adequação para descrever sistemas complexos embebidos, com uma forte componente de dados, uma vez que consegue modelar os aspectos mais importantes deste tipo de sistema: hierarquia funcional, concorrência, transições de estado explicitas, complexidade ao nível das estruturas de dados e do controlo, indicação de fim dos blocos funcionais e a possibilidade de incluir tratamento de excepções. A modelação com PSM é bastante intuitiva, o que facilita a tarefa do projectista.

Capítulo 3

Partição de Sistemas em Componentes Diferenciados

Sumário

Este capítulo começa por definir o processo de partição e por classificar as diferentes abordagens. Descrevem-se os módulos que compõem a abordagem típica ao problema de partição: um algoritmo de partição construtivo, um algoritmo de partição iterativo, as funções de proximidade e de custo e os estimadores de métricas. Introduzem-se alguns algoritmos de partição construtivos, como sejam os algoritmos exaustivo, crescimento de grupos, agrupamento hierárquico e baseados em programação linear com inteiros. Assim como os algoritmos de partição iterativos de Kernighan/Lin, simulated annealing, pesquisa tabu e evolução genética. São ainda referenciados algoritmos específicos de determinada abordagem ao problema de partição. Com base nas qualidades dos algoritmos e na sua complexidade, indica-se quais os algoritmos seleccionados no presente trabalho. Sumaria-se também um conjunto significativo de funções de proximidade e de custo referenciadas na bibliografia e, partindo delas, quais as funções a aplicar neste trabalho. Para concluir, apresenta-se um conjunto representativo de ambientes de desenvolvimento que possuem suporte para a partição de sistemas em componentes e enquadra-se a presente abordagem ao problema de partição no conjunto dessas abordagens.

Conteúdo

3.1 Definição do Processo de Partição

A partição de sistemas em componentes diferenciados é um processo utilizado no desenvolvimento de sistemas, que estabelece a **atribuição** dos objectos que compõem essa descrição aos múltiplos componentes da arquitectura alvo e o **instante** em que cada objecto entra em funcionamento (escalonamento), de modo a obedecer aos objectivos definidos pela função de custo [KL94]. Como a função de custo impõe a busca de uma solução óptima para a atribuição e o escalonamento, o processo de partição é um problema NP-completo [VG95a]. Por **componentes diferenciados** deve entender-se os componentes da arquitectura alvo que apresentem funcionalidade significativamente diferente. Exemplos de componentes diferenciados, que é comum encontrar nas arquitecturas alvo utilizadas no co-projecto de *hardware* e de *software*, são os microprocessadores, os microcontroladores, os DSPs, a memória RAM, os ASICs, as FPGAs e os CPLDs.

O processo de partição transforma a representação dum sistema, unificada e sem compromisso com qualquer implementação, numa representação comprometida com uma arquitectura alvo. A representação resultante do processo de partição deverá especificar os componentes a usar na implementação, a sua funcionalidade e a forma como se interligam. Ou seja, duma representação unificada gera-se uma especificação para cada componente a utilizar na implementação.

As abordagens ao problema de partição são classificadas de acordo com vários critérios, entre os quais os mais importantes são:

- ◇ o domínio em que se processa a partição - segundo este critério, as abordagens são do tipo estrutural ou do tipo funcional (comportamental);
- ◇ os objectivos a que se destina a partição - de acordo com este critério, as abordagens são designadas por intra-componentes ou inter-componentes;
- ◇ o grau de automação - com base neste critério, as abordagens do problema da partição são automáticas, manuais ou parcialmente automáticas.

Numa abordagem **funcional** ao problema de partição, divide-se o comportamento do sistema em partes que depois são sintetizadas em separado; enquanto numa abordagem **estrutural**, sintetiza-se uma estrutura global que depois é separada em partes. Esta diferença traduz-se nas seguintes vantagens da partição funcional relativamente à estrutural:

- ◇ a partição funcional permite tomar decisões arquitecturais mais correctas, a que correspondem soluções mais compactas e com melhor desempenho (tanto em *hardware* como em *software*); do ponto de vista do *hardware*, uma distribuição mais correcta da funcionalidade pelos componentes da arquitectura alvo, tem por consequência uma redução do

número de pinos necessário e a ocorrência de menos caminhos críticos; o último aspecto traduz-se noutra vantagem, que é a possibilidade de utilizar um relógio com frequência mais baixa e por consequência o consumo será menor;

- ◇ como se manipulam menos objectos da descrição do sistema, o processo de partição é mais simples, exigindo por isso menos tempo de cálculo;
- ◇ cada parte em que se decompõe a funcionalidade do sistema pode ser simulada e refinada (corrigida) em separado;
- ◇ a metodologia aplica-se de igual modo a *hardware* e a *software*;
- ◇ a partição funcional é mais intuitiva.

A abordagem estrutural também apresenta vantagens sobre a partição funcional, embora sejam em menor número e menos relevantes:

- ◇ as estimativas dos recursos necessários à implementação dum sistema são mais precisas e fáceis de obter, uma vez que a partição decorre após a estrutura do sistema ter sido sintetizada;
- ◇ as implementações geradas utilizam normalmente menos recursos, porque contêm mais partilhas de recursos.

Uma partição do tipo **intra-componentes** tem por objectivo simplificar as tarefas de síntese, como por exemplo o escalonamento de operações e a selecção de recursos, pelo facto de se dividir um problema complexo e de grande dimensão em vários problemas de menor dimensão. Nestes casos o processo de partição é essencialmente uma tarefa de agrupamento de objectos presentes no modelo do sistema. Portanto, este tipo de abordagem tem como alvo um único componente da arquitectura de implementação.

Um abordagem de partição do tipo **inter-componentes** aplica-se quando o processo de síntese utiliza uma arquitectura alvo heterogénea e multi-componente. Deste modo, a tarefa de partição é essencialmente um processo em que se decompõe a funcionalidade do sistema em partes a atribuir aos componentes da arquitectura alvo. Este processo deve gerar uma solução de partição que satisfaça os requisitos do projecto e respeite os condicionalismos impostos à implementação. Os requisitos são definidos pelo projectista (comportamento temporal, consumo ou dimensões físicas) e os condicionalismos são impostos principalmente pela arquitectura que implementa o sistema. Estes condicionalismos estão relacionados com aspectos físicos, tais como o número de componentes ou o número de pinos por componente,

o espaço de recursos por componente, as particularidades da arquitectura ou os modelos de comunicação permitidos [GDWL92].

Na maior parte das abordagens do tipo inter-componentes, a implementação dos sistemas inclui componentes de *hardware* e de *software*. Assim, o processo de partição designa-se por **partição em *hardware* e *software***. A explicação para que alguns autores considerem o co-projecto de *hardware* e de *software* como sendo apenas um processo de partição, deve-se ao facto de ser esta a tarefa que lida de forma mais explícita com o relacionamento entre *hardware* e *software*.

Entre as razões para se desejar uma metodologia de partição **automática** contam-se:

- como é comum que quem descreve e implementa um sistema em desenvolvimento são pessoas distintas, convém que a partição seja automática para evitar que a pessoa que executa a partição tenha que perceber os detalhes da descrição do sistema;
- ao reduzir o tempo de desenvolvimento, uma partição automática permite que sejam analisadas mais alternativas de implementação;
- podem aplicar-se técnicas de optimização, bastante evoluídas, desenvolvidas para os compiladores.

No presente trabalho pretende-se uma partição funcional, inter-componentes e automática. Dado que a partição duma estrutura sintetizada é pouco intuitiva, complexa e conduz a soluções com desempenho limitado, a partição sobre o comportamento do sistema é a opção natural. Como a arquitectura alvo, descrita no capítulo 5, inclui múltiplos componentes e componentes diferenciados (FPGAs, CPLDs, processador e memória), a partição a efectuar é do tipo inter-componentes. Dispondo de recursos de *hardware* bastante limitados, é de todo conveniente ter a possibilidade de analisar uma quantidade razoável de alternativas de partição, o que só é possível se a partição for executada automaticamente.

3.2 Metodologia Típica de Partição

Uma abordagem típica ao problema de partição inclui quatro grandes módulos: um algoritmo de partição construtivo, um algoritmo de partição iterativo, as funções de proximidade e de custo e os estimadores de métricas [VGG94]. O algoritmo de partição construtivo permite ter um ponto de partida que facilite o sucesso do processo de partição iterativo (figura 3.1).

O processo de partição recebe como entradas o modelo que descreve o sistema, resultante das fases de análise e concepção, e a caracterização da arquitectura alvo a usar na implementação. Como foi mencionado, os modelos do sistema que entram no processo de partição não estão

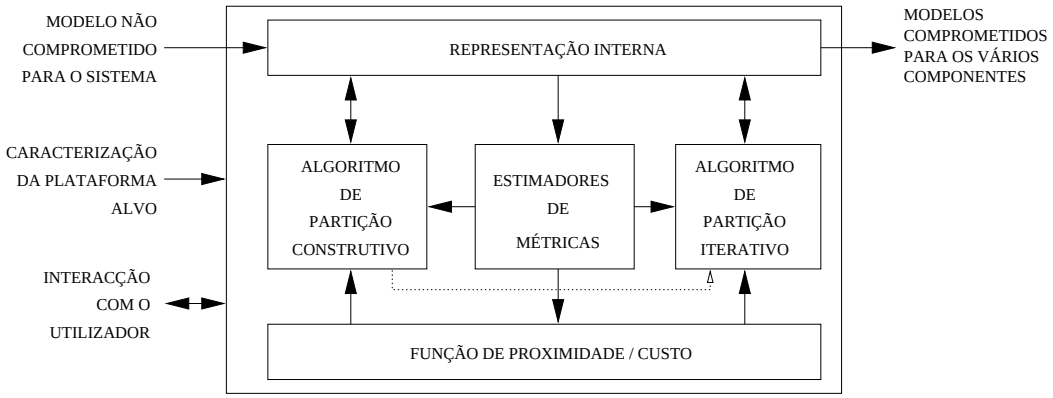


Figura 3.1: A metodologia típica de partição.

comprometidos com uma implementação e representam completamente o sistema, enquanto os modelos a gerar incluem detalhes que comprometem a implementação com uma determinada arquitectura. A partição gera um modelo separado para cada componente da arquitectura alvo seleccionado.

Nas abordagens descritas na bibliografia, entre os meta-modelos mais utilizados para descrever os sistemas à entrada para o processo de partição encontram-se o CDFG [CA97, MMMK95], o DFG [GM94], a FSM [CGH+94], a RdP [EPD94], o CSP [BS94], uma versão alterada de um dos meta-modelos anteriores [EHB93], ou uma combinação destes meta-modelos [MEGT96]. Apesar da diversidade de meta-modelos, a maioria das abordagens transforma o modelo de entrada numa representação interna do tipo diagrama de fluxo (nomeadamente CDFG). O meta-modelo que se selecciona condiciona o tipo de objectos manipulados no processo de partição, os quais terão uma granulosidade muito diferente entre as várias abordagens, dado que estas usam meta-modelos bastante distintos. Deste modo, a granulosidade poderá ser do tipo grão fino (operação), do tipo grão intermédio (bloco de operações) ou do tipo grão grosso (FSM, tarefa, função ou procedimento, processo).

Quando à estratégia usada para deslocar objectos nas sucessivas iterações, os algoritmos de partição podem ser classificados em algoritmos do tipo *greedy* ou do tipo *hill-climbing*. Os algoritmos do tipo ***greedy*** empregam uma estratégia em que um objecto é deslocado para outra partição desde que o custo da solução de partição resultante seja melhor que o custo da solução actual. Para resolver o problema de paragem em soluções que só são óptimas a nível local, os algoritmos do tipo ***hill-climbing*** permitem um número limitado de deslocamentos que pioram a qualidade da solução de partição.

Relativamente à forma como convergem para a alternativa de partição que é devolvida, os algoritmo de partição podem ser classificados em construtivos ou iterativos. Um **algoritmo de partição construtivo** permite obter uma alternativa de partição partindo do ponto zero, ou seja, com base apenas na descrição do sistema e no número de partições a gerar, os

objectos da descrição são distribuídos por esse número de partições. A alternativa pode ser uma solução óptima (como acontece nos métodos de programação linear com inteiros) ou então nem sequer ser uma solução exequível. A decisão de como atribuir os objectos às partições é um problema com solução local e guiado por uma **função de proximidade**. O princípio base da função de proximidade é medir, num dado momento do processo de construção das partições, a proximidade que existe entre um objecto e as partições parciais estabelecidas até esse momento. É comum os algoritmos de partição construtivos serem utilizados na fase inicial do processo de partição, com o objectivo de estabelecer um ponto de partida para a busca iterativa da melhor solução de partição. Uma boa solução inicial de partição, reduz o tempo de cálculo e aumenta a probabilidade de se obter uma solução de maior qualidade com a partição iterativa que se lhe segue. Estas vantagens serão potenciadas se na solução inicial se efectuar o agrupamento dos objectos mais próximos num único objecto. A proximidade entre objectos tem a ver com a existência de uma troca de informação intensa, ou com o facto de possuírem uma funcionalidade fortemente relacionada. A selecção aleatória, o algoritmo de crescimento de grupos[1] ou o agrupamento hierárquico [BS94] são exemplos de algoritmos de partição construtivos [GVNG94].

As soluções de partição geradas por um **algoritmo de partição iterativo** resultam dum processo em que sucessivamente se tenta melhorar uma solução anteriormente gerada, usando para isso uma função de custo que mede a qualidade de cada solução. Deste modo, consegue explorar-se o espaço de projecto. A um algoritmo iterativo está associada uma **heurística de optimização** da solução de partição que lhe é fornecida. Para melhorar uma solução de partição deslocam-se objectos entre partições, de modo a encontrar uma solução com custo inferior. Os componentes fundamentais duma heurística de optimização são:

- a estratégia de controlo da evolução do algoritmo, que é utilizada para evitar mínimos locais da função de custo e reduzir ao mínimo o número de deslocamentos de objectos envolvidos na procura da solução de partição óptima;
- o manuseamento da informação relativa ao custo da solução de partição, que exige (i) uma estrutura de dados contendo a informação de cada objecto necessária ao cálculo do custo da solução, (ii) um conjunto de regras para actualizar a informação guardada na estrutura de dados, após o deslocamento de um objecto e (iii) uma função de custo que define a qualidade da solução de partição.

Um dos algoritmos de partição iterativos mais utilizados é o de *simulated annealing* [EHB93] [EPD94] [FSV97] [EPKD97], mas também se usam algoritmos genéticos [Nie98], variantes do algoritmo de Kernighan/Lin [VG92] [Vah97] [VL97], o algoritmo de pesquisa tabu [FSV97]

[1] *Cluster growth*, na terminologia inglesa.

[EPKD97], um algoritmo que usa vários critérios que variam dinamicamente ao longo do processo de partição (GCLP) [KL94] ou um algoritmo baseado em programação dinâmica e que dá ênfase à comunicação (PACE) [KM96b]. O algoritmo *simulated annealing* consegue gerar soluções bastante optimizadas, gastando para isso um tempo de cálculo elevado. Para atenuar este aspecto negativo do algoritmo, existe a possibilidade de jogar com a relação entre a qualidade da solução e o tempo de cálculo.

Para uma determinada solução de partição, a **função de custo** combina várias métricas num valor designado por custo, que define quão óptima é a solução de partição. A função de custo quantifica a proximidade de uma solução, obtida na iteração corrente do processo de partição, em relação à solução óptima. Para conseguir isso, o algoritmo procura minimizar a violação dos condicionalismos impostos e maximizar o sucesso em atingir os requisitos exigidos. Se uma solução de partição não violar nenhum dos condicionalismos impostos à implementação, diz-se que é uma solução exequível. Se a solução não violar nenhum dos condicionalismos e atingir todos os requisitos exigidos, é uma solução óptima. Deste modo, quando se estima que determinado requisito não é atingido, a função de custo devolve um valor pior do que no caso em que o requisito é conseguido. Analogamente, ao estimar-se que um condicionalismo não será respeitado, a função de custo devolve um valor que será tanto pior quanto mais grave for o desrespeito por esse condicionalismo.

Uma função de custo necessita de **estimativas das métricas** que parametrizam as soluções de partição. As métricas mais utilizadas pela função de custo são o espaço ocupado pelos recursos (como por exemplo, o código em *software* ou a lógica em *hardware*), os tempos de comunicação e de computação e os aspectos físicos (como por exemplo o número de pinos no componente ou o número de registos no processador utilizados). Métricas menos relevantes para o processo de partição são o consumo e o custo de produção. Para estimar o espaço ocupado pelos recursos em *hardware* é necessário efectuar uma selecção simplificada e para estimar tempos faz-se o escalonamento simplificado das operações. O número de pinos pode ser estimado usando o conhecimento relativo ao número de sinais que atravessam os limites das partições. A execução destas tarefas apoia-se em modelos para os recursos de *hardware*, de *software* e de comunicação.

3.3 Algoritmos de Partição Construtivos

Os algoritmos de partição construtivos mais comuns são o exaustivo, crescimento de grupos, agrupamento hierárquico e os algoritmos baseados em programação linear com inteiros.

Exaustivo

O algoritmo de partição **exaustivo** analisa todas as possibilidades de partição, obtendo sempre a solução de partição óptima para os objectivos propostos. Se os sistemas forem descritos por um número elevado de objectos, este algoritmo é inviável porque exige um tempo de cálculo muito alto. No caso de n objectos e de se desejarem p partições, o algoritmo tem que analisar p^n alternativas de partição. Contudo, o algoritmo exaustivo é por vezes utilizado na validação de outros algoritmos de partição.

Crescimento de Grupos

O algoritmo de crescimento de grupos começa por seleccionar o objecto semente de cada grupo, ou partição, a criar e depois, de uma forma sucessiva, incorpora em cada grupo o objecto mais próximo dos objectos que já se encontram nesse grupo [GDWL92]. O algoritmo é composto por três tarefas, que no caso particular em que se geram p partições com o mesmo número n de objectos, são repetidas p vezes: (T_1) seleccionar o objecto semente que originará o grupo; o objecto semente pode ser seleccionado aleatoriamente, por intervenção do projectista ou de acordo com o maior número de ligações entre o objecto e todos os outros ainda não agrupados; (T_2) até perfazer um total de n objectos no grupo, seleccionar o objecto mais próximo do conjunto de objectos já agrupados no grupo; (T_3) colocar o objecto seleccionado no grupo a que se destina. O algoritmo de crescimento de grupos é de fácil implementação, mas gera soluções pouco optimizadas se a regra de selecção dos objectos não for "inteligente". Por exemplo, se se utilizar a seguinte regra na selecção dos objectos que farão parte dum grupo:

> *O objecto não agrupado que apresente a maior proximidade com os objectos já agrupados no grupo em causa;*

os objectos ainda não agrupados são excluídos da definição de máxima proximidade, ou seja, a selecção nem sempre escolherá os objectos mais adequados. Para melhorar a selecção dos objectos a atribuir às partições pode usar-se, em vez dum critério baseado numa única métrica, uma função de proximidade que envolva várias métricas.

A **selecção aleatória** é um caso particular do algoritmo de crescimento de grupos, em que a regra de selecção dos objectos é uma função aleatória. A implementação da selecção aleatória é simples e a sua aplicação exige pouco tempo de cálculo, mas os resultados produzidos são fracos. A funcionalidade do algoritmo resume-se a seleccionar aleatoriamente objectos de entre os não agrupados e a colocá-los num grupo, ou partição, enquanto não se atingir o limite imposto ao tamanho de cada partição. O processo repete-se até estarem agrupados

todos os objectos.

Agrupamento Hierárquico

O algoritmo de agrupamento hierárquico efectua sucessivos agrupamentos de pares de objectos que originam novos objectos. O processo de agrupamento de objectos repete-se até se verificar uma condição de paragem. As partições a criar são definidas com base nos objectos que resultaram dos agrupamentos anteriores. Em cada iteração o algoritmo agrupa os dois objectos que estiverem mais próximos, de acordo com uma, ou várias, métricas de proximidade [GVNG94] [EKP98a]. Os dois objectos a agrupar podem ser objectos da descrição do sistema ou objectos que resultaram de um agrupamento efectuado numa iteração anterior do algoritmo. Após o agrupamento de dois objectos, as métricas de proximidade entre cada par de objectos são recalculadas e o processo de agrupamento é repetido até se verificar a condição de paragem. No exemplo da figura 3.2 (i) a (iv), a anotação nos arcos dos grafos representa a métrica de proximidade entre dois objectos. Para recalcular a métrica de proximidade aplica-se a função *máximo*. Um exemplo de condição de paragem é aquela que indica que o processo de agrupamento deve parar quando a métrica de proximidade entre qualquer par de objectos for inferior a um valor pré-definido. O algoritmo pode ser ligeiramente alterado de modo a que o processo de agrupamento se repita até restar apenas um objecto, que agrupa todos os objectos iniciais e em que, com o historial dos agrupamentos, se forma uma árvore de grupos. Depois, é fácil gerar diferentes alternativas de partição, bastando para tal que se aplique à árvore de grupos uma linha de corte num nível distinto (figura 3.2 (vi) e (vii)).

Uma variante do algoritmo de agrupamento hierárquico é o **agrupamento por fases**, em que cada fase utiliza uma métrica de proximidade diferente [LT91]. Cada fase funciona com os grupos gerados pela fase anterior. A ordem de aplicação das métricas é determinada pela sua importância, ou seja, o agrupamento começa com a métrica mais prioritária e termina na menos prioritária. A similaridade entre operadores, a quantidade de ligações partilhadas e a existência de concorrência são exemplos de métricas a empregar nas diferentes fases do agrupamento. A principal vantagem do agrupamento por fases relativamente ao agrupamento hierárquico simples advém do facto de no primeiro algoritmo, para se aplicarem várias métricas, só é preciso determinar a maior ou menor importância de cada uma em relação às outras, no segundo algoritmo é necessário quantificar a importância de cada métrica para definir uma função de proximidade. Em contrapartida, o agrupamento por fases apresenta uma desvantagem, relativamente ao agrupamento hierárquico simples, que se traduz em ser necessário determinar o número de iterações em que se aplica cada uma das métricas.

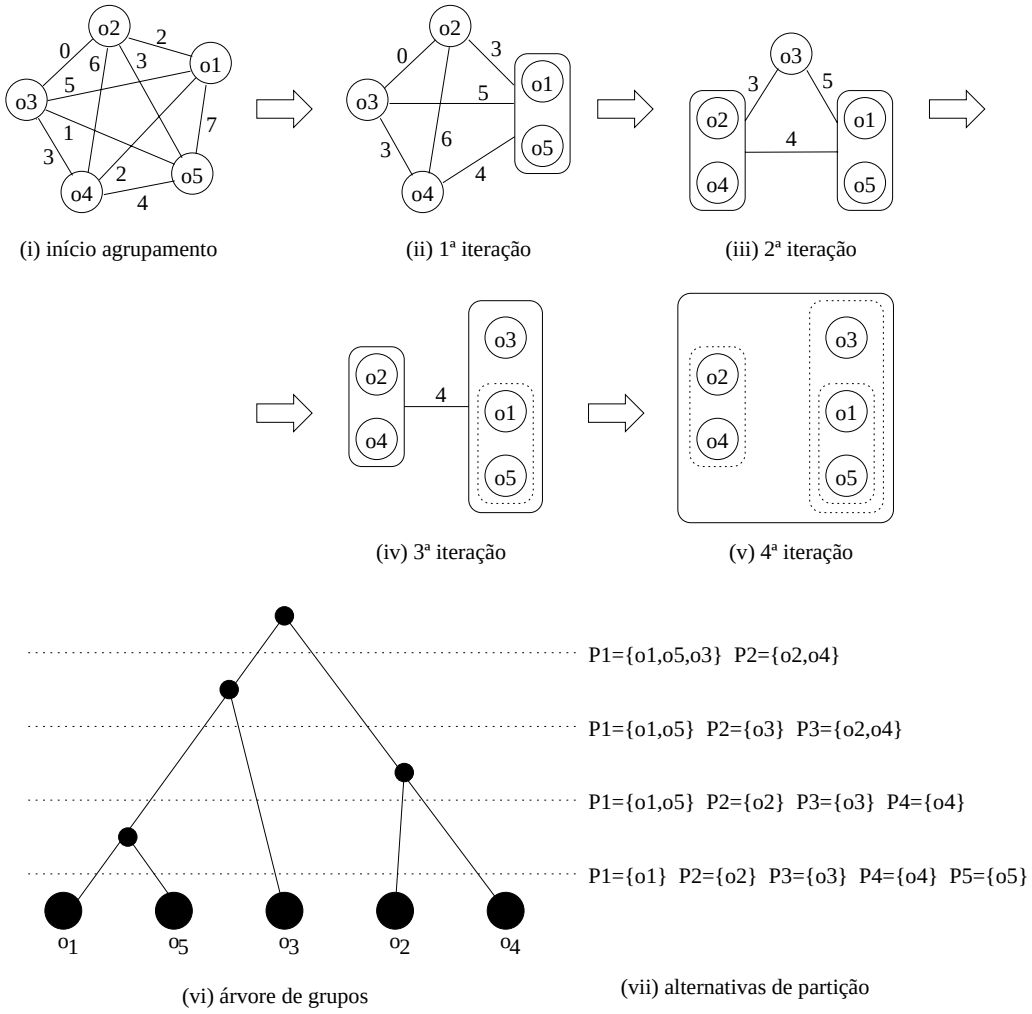


Figura 3.2: Ilustração do funcionamento do algoritmo de agrupamento hierárquico.

Programação Linear com Inteiros

O problema de partição em componentes pode ser resolvido por um método baseado em programação matemática, mais concretamente por um método de programação linear com inteiros[2] (PLI) [GVNG94] [Nie98]. A formulação dum programa linear (com inteiros) inclui um conjunto de variáveis (inteiras), um conjunto de inequações lineares que condicionam os valores das variáveis e uma função de custo, também ela uma função linear das variáveis. Uma definição formal de programação linear é:

> *Programação linear é um processo em que é necessário calcular os valores positivos para o conjunto de variáveis* $V = v_1, v_2, ..., v_n$ *que minimizam a função de custo* $Fcusto(V) = \sum_{j=1}^{n} k_j * v_j$, *em que* k_j *são constantes. As variáveis* V *devem verificar as* m *inequações do tipo* $\sum_{j=1}^{n} a_{ij} * v_j \leq b_i$, *com* a_{ij} *e* b_i *a serem constantes e* $1 \leq i \leq m$.

[2] ***I**nteger **L**inear **P**rogramming (**ILP**)*, na terminologia inglesa.

Quando se aplica a programação linear à resolução do problema de partição, as variáveis representam decisões assumidas durante o processo de partição e estimativas de métricas. Por exemplo, a informação de que um objecto o_i foi atribuído a uma partição P_j é guardada numa variável inteira $map_i = j$, ou então a estimativa do espaço ocupado pela partição P_j é guardado na variável $area_j$. Uma possível inequação, utilizada para garantir que o condicionalismo imposto ao espaço (A) não é violado, é dada por $\sum_{j=1}^{n} area_j \leq A$.

Os condicionalismos e requisitos, utilizados para garantir a exequibilidade e o correcto funcionamento das soluções de partição, são traduzidos em inequações da formulação PLI. Por exemplo, se existir um condicionalismo relacionado com a ordem de execução de duas tarefas T_i e T_j, indicando que T_j está dependente de T_i, usa-se uma inequação 3.1 por cada par de processadores P_k e P_m que possam implementar as tarefas T_i e T_j [EKP98a].

$$Ts_i + Texec_{ki} + Tcom_{ij} \leq Ts_j + (2 - M\mu P_{ki} - M\mu P_{mj}) * C \tag{3.1}$$

em que

- Ts_i (Ts_j) é a variável que define o instante de arranque da execução da tarefa T_i (T_j);
- $Texec_{ki}$ é a constante que define o tempo de execução da tarefa T_i no processador P_k;
- $Tcom_{ij}$ é a constante que representa o tempo de comunicação da tarefa T_i para T_j, no caso de as tarefas estarem a correr em processadores diferentes;
- $M\mu P_{ki}$ ($M\mu P_{mj}$) é a variável inteira/booleana que assume o valor 1 se a tarefa T_i (T_j) for atribuída ao processador P_k (P_m) ou o valor 0 nos outros casos;
- C é um parâmetro, ao qual se atribui um valor muito elevado, que garante que a inequação só se verifica se houver uma atribuição da tarefa T_i ao processador P_k ($M\mu P_{ki} = 1$) e da tarefa T_j ao processador P_m ($M\mu P_{mj} = 1$).

Poderia escrever-se inequações semelhantes à inequação 3.1 para (i) garantir uma correcta ordem de execução entre duas tarefas, que apresentam uma dependência na ordem de execução e estão atribuídas ao mesmo processador, (ii) evitar conflitos no acesso aos recursos dum processador, por parte de duas tarefas atribuídas a esse processador e que não apresentam dependências na ordem de execução, (iii) expressar a dependência existente entre uma tarefa que envia dados e a tarefa que os recebe e (iv) evitar a utilização simultânea dos barramentos por parte de mais do que uma tarefa.

Após a obtenção da formulação PLI para um problema de partição concreto, é necessário resolvê-la através dum método como o de *branch and bound* ou de relaxação *Lagrangeana*. A ferramenta *lp_ solve* [Ber98], que implementa o método *branch and bound*, está disponível para

resolver formulações PLI. No caso duma formulação com inequações como a que se definiu em 3.1, os resultados seriam (i) os valores para as variáveis Ts, que definem um escalonamento óptimo e (ii) os valores binários para as variáveis $M\mu P$, que definem a partição óptima das tarefas pelos componentes da arquitectura.

Para resolver uma formulação PLI, um problema NP-completo, em vez dum método exacto aplica-se uma heurística. Mesmo assim, o tempo de cálculo será sempre superior ao exigido pelos outros algoritmos construtivos apresentados, mas é possível obter soluções de partição quase-óptimas.

3.4 Algoritmos de Partição Iterativos

Kernighan/Lin

O princípio que está subjacente ao algoritmo de Kernighan/Lin[3] é a comutação de um objecto incluído num grupo, ou partição, com um objecto doutro grupo, por forma a optimizar o custo associado ao conjunto dos grupos. No algoritmo base, a solução inicial que se pretende optimizar deve ser constituída por duas partições com igual número de objectos [EKP98a]. O algoritmo base utiliza o seguinte critério de evolução:

> *Os dois objectos que comutam de grupo são aqueles que provocam a maior descida no valor devolvido pela função de custo*[4] *ou então a menor subida nesse valor.*

Ao critério de evolução definido corresponde uma estratégia do tipo *hill-climbing*, uma vez que o algoritmo pode evoluir numa direcção em que a função de custo piora, evitando-se assim alguns mínimos locais.

Cada iteração do algoritmo analisa todos os objectos ainda não deslocados, atribuídos a ambos os grupos, comutando o par de objectos que validar o critério de evolução. Após a execução da comutação de um par de objectos, obtém-se uma nova solução de partição que é guardada temporariamente. Quando todos os objectos tiverem sido deslocados, ou não for possível validar o critério de evolução, a iteração termina. Para evitar bloqueios no algoritmo, cada objecto só é deslocado uma vez por iteração. No final da iteração guarda-se a solução de partição que, de entre todas as obtidas nesta iteração, apresenta o melhor valor devolvido pela função de custo.

As tarefas definidas como uma iteração são repetidas, utilizando como entrada a melhor solução de partição obtida na iteração anterior, enquanto o custo da solução de partição obtida na iteração corrente for inferior ao da solução de partição obtida na iteração anterior.

[3] Este algoritmo também é designado por ***min-cut*** ou **migração de grupos**.

[4] No caso duma função de custo que se pretende mínima.

O algoritmo KL foi estendido por forma a aceitar e gerar duas partições de tamanho diferente, designando-se esta extensão por **KL/não-balanceado** [GVNG94]. Fiduccia e Mattheyeses propuseram outra alteração ao algoritmo de Kernighan/Lin (extensão **KL/FM**), traduzida na definição duma estrutura de dados mais eficiente, que permite que o deslocamento dum objecto seja executado num tempo constante [FM82]. O algoritmo KL foi ainda alterado[5] de modo a efectuar uma partição funcional, em vez da partição estrutural para a qual a versão inicial do algoritmo está orientada, originando a extensão **KL/funcional** [VL97].

Simulated Annealing

O funcionamento do algoritmo de *simulated annealing* (SA) é inspirado no processo de temperar materiais, no qual o material é sujeito a uma temperatura superior ao seu ponto de fusão e depois a temperatura é reduzida lentamente até ao estado de energia mínima. O estado de energia mínima é atingido, desde que para cada temperatura se atinja o ponto de equilíbrio [KGV83]. No caso da partição, a variável energia usada no processo de temperar os materiais é substituída pela variável custo da solução de partição e o estado de energia mínima equivale à solução de partição óptima.

O funcionamento do algoritmo apresentado na figura 3.3 começa com uma solução de partição P, gerada por um algoritmo construtivo e com uma temperatura inicial $TemperaturaInicial$ [GVNG94]. A temperatura é depois diminuída lentamente, correspondendo a cada temperatura de simulação T uma nova iteração. Para diminuir a temperatura utiliza-se a função $ReducaoTemperatura$ que possui uma funcionalidade definida pela equação 3.2.

```
algSA (P, Fcusto, TemperaturaInicial) ≡

    // P é a solução de partição inicial
    T = TemperaturaInicial
    custo = Fcusto(P)
    enquanto CondicaoParagem() = FALSO fazer
        enquanto Equilibrio() = FALSO fazer
            P_deslocamento = MovimentoAleatorio(P)
            custo_deslocamento = Fcusto(P_deslocamento)
            Δcusto = custo_deslocamento − custo
            se Aceitacao(Δcusto, T) > Aleatorio(0, 1) então
                P = P_deslocamento
                custo = custo_deslocamento
            fse
        fenquanto
        T = ReducaoTemperatura(T)
    fenquanto
```

Figura 3.3: Algoritmo de *simulated annealing* (SA).

[5] As alterações efectuadas são discutidas no apêndice A.

$$T_{nova} \;=\; \alpha * T_{antiga} \qquad [\text{ com } \; (0 < \alpha < 1) \;] \tag{3.2}$$

Uma iteração do algoritmo, à qual corresponde uma temperatura fixa, envolve as seguintes tarefas:

- ◇ gerar aleatoriamente uma série de deslocamentos de objectos (com a função $MovimentoAleatorio$), até se atingir o ponto de equilíbrio; o ponto de equilíbrio pode ser definido como uma situação em que após uma série de deslocamentos, a melhoria do custo da solução de partição é inferior a um nível de decisão (função $Equilibrio$);
- ◇ para cada deslocamento gerado, calcular o custo da solução que daí resulta ($custo_{deslocamento}$);
- ◇ para cada deslocamento gerado, verificar se ele é aceite; um deslocamento é aceite se a variação do custo satisfizer o critério de aceitação de deslocamentos ($Aceitacao >$ número aleatório entre 0 e 1), em que $Aceitacao$ é definido na equação 3.3.

$$Aceitacao(\Delta custo, T) \;=\; minimo(1, e^{-\Delta custo/T}) \tag{3.3}$$

Quando o deslocamento dum objecto conduz a uma solução de partição com custo inferior ao da solução anterior a esse deslocamento ($\Delta custo < 0$), a deslocação do objecto é sempre aceite. Mesmo que a solução de partição apresente um custo superior ao da solução anterior ao deslocamento ($\Delta custo > 0$), a probabilidade de o deslocamento ser aceite varia directamente com a temperatura T e inversamente com a diferença de custo $\Delta custo$. Ou seja, quanto pior for a solução obtida com o deslocamento do objecto, o mesmo é dizer quanto mais elevado for o $\Delta custo$, maior será a probabilidade de a solução ser rejeitada. Por outro lado, quando menor for a temperatura T, menor é a probabilidade de se aceitarem maus deslocamentos de objectos.

O algoritmo de *simulated annealing* possui dois tipos de controlo: (i) um ciclo externo, relativo à variação de temperatura, que faz terminar o algoritmo quando a temperatura T for aproximadamente nula ou quando o sistema estiver num estado que não evolui mais (função $CondicaoParagem$) e (ii) um ciclo interno, que possibilita a execução de várias deslocações de objectos para a mesma temperatura (função $Equilibrio$). Para cada deslocação é gerado um número aleatório entre 0 e 1 que condiciona a aceitação dessa deslocação (função $Aleatorio$).

Através do número de iterações a executar por cada valor da temperatura de simulação ($N_{iteracoes}$) e da taxa de redução da temperatura (parâmetro α), é possível controlar a relação entre a qualidade das soluções de partição e o tempo de cálculo. Teoricamente, para

se garantir uma solução óptima a partir de qualquer ponto inicial, o número de iterações a efectuar varia exponencialmente com o número de objectos do sistema, razão pela qual é necessário usar uma implementação eficiente que reduza o tempo de cálculo e mantenha elevada a probabilidade de obter soluções quase-óptimas [GL95]. No apêndice A são apontadas recomendações para a temperatura inicial, a taxa de redução da temperatura, o número de iterações e a condição de paragem, que permitem melhorar a eficiência do algoritmo SA.

Pesquisa Tabu

A pesquisa tabu (PT) é um algoritmo que para evoluir da solução de partição corrente pesquisa apenas numa vizinhança dessa solução. A pesquisa na vizinhança é ainda limitada pela existência de um conjunto de deslocamentos que estão proibidos (são **tabu**). Para evitar parar em mínimos locais da função de custo, o algoritmo PT emprega uma pesquisa determinística, melhor que a selecção aleatória do algoritmo SA e um esquema flexível de controlo da pesquisa baseado na memorização do historial desta pesquisa [EKP98a]. A principal informação memorizada é a indicação de quais os deslocamentos, na vizinhança da solução corrente, que estão proibidos de ser efectuados. Por **vizinhança duma solução** P_k, na forma mais simples, entende-se o conjunto de soluções que se atingem a partir de P_k com uma transformação simples, ou seja, com o deslocamento de um objecto.

Os tabus são definidos com base no historial da pesquisa e mantêm-se válidos durante um determinado número de iterações, designado por **validade do tabu**[6]. Um deslocamento tabu pode ser efectuado desde que origine a solução com o menor custo de entre todas as soluções da vizinhança, significando isso o desrespeito pela validade do tabu. Para contemplar a possibilidade de ultrapassar a validade dum tabu, definem-se **critérios de aspiração**. Alguns exemplos de critério de aspiração são:

- **aspiração por objectivo**: o deslocamento a efectuar é aquele que gera a solução com menor custo, mesmo que o deslocamento esteja proibido por alguma restrição;
- **aspiração por direcção de pesquisa**: um deslocamento d restringido por um tabu, a que corresponde a característica c, será efectuado se (i) o último deslocamento possuindo a característica c resultou numa solução com custo inferior à solução que a precedeu e (ii) o deslocamento d origina uma solução com custo inferior à solução actual;
- **aspiração por defeito**: é definido para permitir uma alternativa quando todos os deslocamentos são tabu e os outros critérios de aspiração também não conseguem seleccionar um deslocamento para ser efectuado.

[6] ***Tabu tenure***, na terminologia inglesa.

A informação sobre os tabus válidos em cada momento é guardada numa memória de curta duração, a qual funciona como historial dos deslocamentos mais recentes. O historial dos deslocamentos mais frequentes é guardado numa memória de longa duração, que complementa a memória de curta duração e permite diversificar a pesquisa para zonas do espaço de projecto não visitadas nas iterações mais recentes. Um terceiro tipo de historial que se pode usar, diz respeito aos deslocamentos de maior qualidade efectuados. Este historial constitui uma memória de duração intermédia e pode ser utilizado com o objectivo de reforçar a convergência para a solução de partição óptima.

A validade dos tabus deve ser escolhida com cuidado, uma vez que a pesquisa efectuada pelo algoritmo PT é bastante sensível ao seu valor. A regra principal é: usar uma validade mais curta com um tabu mais restritivo do que com um tabu menos restritivo. No que respeita ao valor da validade dos tabus, a regra é: a validade não deve ser nem muito curta (para evitar entrar em ciclo), nem muito longa (para impedir que a pesquisa se afaste do mínimo absoluto da função de custo).

Para seleccionar o melhor deslocamento a efectuar em cada iteração, é necessário analisar todas as soluções na vizinhança da solução actual. Como esta tarefa ocupa um tempo de cálculo muito elevado, é conveniente reduzir o número de alternativas analisadas, para assim acelerar a selecção do deslocamento. Dois métodos que podem ser aplicados na redução do número de alternativas a analisar são: (i) seleccionar o deslocamento a efectuar de forma aleatória e (ii) decompor o conjunto de deslocamentos que define a vizinhança da solução actual em sub-conjuntos, utilizando-se em cada iteração sucessiva um sub-conjunto diferente.

A figura 3.4 apresenta uma implementação do algoritmo PT [EPKD97] [EKP98a]. Cada iteração analisa todas as alternativa de partição P_k pertences à vizinhança da solução actual $V(P_{actual})$, ou seja, todas as soluções obtidas a partir da solução actual P_{actual} através do deslocamento de um objecto duma partição para outra. A primeira alternativa de deslocamento considerada em cada iteração, é um deslocamento que melhora a função de custo e obedece a uma das condições seguintes: não é um deslocamento tabu ($tabu(k) = INVALIDO$) ou então é um deslocamento tabu que pode ser efectuado devido a um critério de aspiração ($c\_aspiracao\_tabu(P_k) = VERDADE$). Se não houver nenhum deslocamento elegível pela primeira alternativa, o deslocamento seleccionado é aquele não sendo tabu conduz ao menor aumento do custo da solução de partição. Neste caso, aplica-se ao custo das soluções uma bonificação $bonificacao(P_k)$ que favorece o deslocamento dos objectos que estão à mais tempo na mesma partição. Como terceira e última alternativa, o deslocamento a efectuar será aquele cuja validade do tabu está mais perto do fim. O algoritmo termina quando se atinge o limite imposto ao número de repetições da pesquisa ($LIMITE\_Nrep$). Uma repetição da pesquisa é um conjunto sucessivo de iterações, que começa com uma solução de partição fornecida e

algPT $(P, Fcusto, LIMITE\_Nrep, LIMITE\_Nidms) \equiv$

$N_{repeticoes} = 0$, $N_{iteracoes\_desde\_melhor\_solucao} = 0$
$P_{actual} = P$ // P é a solução inicial
$P_{melhor} = P_{actual}$, $custo\_melhor = Fcusto(P_{melhor})$

INICIO:
para *cada alternativa de solução* $P_k \in V(P_{actual})$ fazer
Calcular a variação de custo $\rightarrow \Delta C_k = Fcusto(P_k) - Fcusto(P_{actual})$
fpara

// Primeira alternativa de deslocamento
para $(\Delta C_k < 0)$ *E* (ΔC_k *por ordem crescente de valor)* fazer
se $(tabu(P_k) = INVALIDO)$ *OU* $(c\_aspiracao\_tabu(P_k) = VERDADE)$ então
$P_{actual} = P_k$
saltar para FIM
fse
fpara

// Segunda alternativa de deslocamento
para *cada alternativa de solução* $P_k \in V(P_{actual})$ fazer
$\Delta C'_k = \Delta C_k + bonificacao(P_k)$
fpara
para $(\Delta C'_k < 0)$ *E* ($\Delta C'_k$ *por ordem crescente de valor)* fazer
se $(tabu(P_k) = INVALIDO)$ então
$P_{actual} = P_k$
saltar para FIM
fse
fpara

// Terceira alternativa de deslocamento
Definir P_{actual} usando o deslocamento (tabu) cuja validade do tabu está mais perto do fim

FIM:
Actualizar Historial $\rightarrow$ $tabu()$ e $c\_aspiracao\_tabu()$
Incrementar $N_{iteracoes\_desde\_melhor\_solucao}$
se $(Fcusto(P_{actual}) < Fcusto(P_{melhor}))$ então
$P_{melhor} = P_{actual}$, $custo\_melhor = Fcusto(P_{melhor})$
$N_{iteracoes\_desde\_melhor\_solucao} = 0$
fse

se $N_{iteracoes\_desde\_melhor\_solucao} < LIMITE\_Nidms)$ então
saltar para $INICIO$
fse

Incrementar $N_{repeticoes}$
se $N_{repeticoes} < LIMITE\_Nrep)$ então
Estabelecer uma nova solução inicial P_{actual} usando o historial
saltar para $INICIO$
fse
Devolver a melhor solução obtida $\rightarrow$ P_{melhor}

Figura 3.4: Algoritmo de pesquisa tabu (PT).

termina quando se atinge o limite imposto ao número de iterações sem que haja melhoria da função de custo ($LIMITE\_Nidms$).

Evolução Genética

Os algoritmos de evolução genética (EG) surgem como uma imitação do processo natural de evolução genética [GVNG94][Nie98]. Uma característica que distingue os algoritmos EG dos algoritmos iterativos de Kernighan/Lin, *simulated annealing* e pesquisa tabu, resulta de os algoritmos EG guardarem um conjunto de soluções de partição por iteração, enquanto os outros algoritmos guardam apenas uma solução. Ao conjunto de soluções de partição guardados numa iteração dá-se o nome de **geração**. Para obter uma geração, é comum os algoritmos EG imitarem três métodos que ocorrem na evolução natural:

- **selecção** é o método pelo qual os melhores elementos (soluções de partição) da geração actual são transmitidos à geração seguinte, sendo seleccionados entre os que apresentam os menores valores da função de custo;
- **cruzamento** é o método em que a partir de dois elementos fortes da geração actual, se obtém um ou mais elementos para a geração seguinte, pela mistura de características (partições) de ambos os elementos; o algoritmo pode optar por uma política em que os elementos gerados por cruzamento só são aceites se um valor, gerado aleatoriamente em cada iteração, for superior à taxa de cruzamento;
- **mutação** é o método que altera ligeiramente um elemento da geração actual e o transmite à geração seguinte; a alteração duma solução de partição é conseguida pelo deslocamento entre partições de alguns objectos escolhidos aleatoriamente; o método de mutação, embora não seja tão importante como o método de cruzamento, potencia a diversidade dentro de cada geração; esta característica é útil para alterar situações como aquela em que todos os elementos da geração estão perto do mesmo mínimo local; tal como acontece no método de cruzamento, pode rejeitar-se os elementos gerados por mutação quando um valor, gerado aleatoriamente em cada iteração, for inferior à taxa de mutação.

As definições anteriores mostram que apenas os métodos cruzamento e mutação conseguem gerar elementos novos para a geração seguinte. Como os elementos gerados são novos, é preciso garantir que são soluções de partição exequíveis. Com a evolução do algoritmo, as soluções de partição da geração actual vão se aproximando da solução óptima.

A figura 3.5 ilustra de forma simplificada o algoritmo EG [GVNG94]. O algoritmo começa com a criação da primeira geração, composta por *tamGeracao* soluções de partição, usando

a selecção aleatória ou outro algoritmo construtivo. A geração actual (G) é iterativamente alterada, pela aplicação dos métodos de selecção, cruzamento e mutação, dando origem a uma nova geração. O processo repete-se até se verificar a condição de paragem ($Terminar() = verdade$). A função $Seleccao(G, N_{sel})$ obtém N_{sel} soluções de partição aplicando o método de selecção à geração actual G, enquanto a função $Cruzamento(G, N_{cruz})$ gera N_{cruz} soluções de partição aplicando o método de cruzamento a G e $Mutacao(G, N_{mut})$ obtém N_{mut} soluções de partição aplicando o método de mutação a G.

```
algEG (O, Fcusto, tamGeracao, N_sel, N_cruz, N_mut) ≡

    // Criar a primeira geração com tamGeracao soluções de partição

    G = ∅
    para k = 1 até tamGeracao fazer
        P_k = CriarParticao(O) // O ≡ objectos a incluir nas partições
        G = G ∪ P_k
    fpara
    P_melhor = MelhorParticao(G)
    custo_melhor = Fcusto(P_melhor)

    // Efectuar a evolução da geração

    enquanto Terminar() = falso fazer
        G = Seleccao(G, N_sel) ∪ Cruzamento(G, N_cruz)
        Mutacao(G, N_mut)
    fenquanto

    se (Fcusto(MelhorParticao(G)) < custo_melhor) então
        P_melhor = MelhorParticao(G)
        custo_melhor = Fcusto(P_melhor)
    fse

    // Devolver a melhor solução da última geração

    devolver (P_melhor)
```

Figura 3.5: Algoritmo de evolução genética (EG).

Os valores seleccionados para os parâmetros envolvidos num algoritmo EG influenciam fortemente o seu sucesso. Os principais parâmetros são o número de elementos duma geração, a taxa de cruzamento e a taxa de mutação. Utilizar um número de elementos por geração elevado significa uma maior diversidade de elementos, uma menor probabilidade do algoritmo convergir prematuramente para um mínimo local da função de custo, mas o tempo de execução e os requisitos de memória também serão maiores. Usar uma taxa de cruzamento maior significa que a possibilidade de combinação de elementos é maior e que a ruptura de elementos fortes também é maior. Por fim, uma taxa de mutação elevada implica uma diversidade maior na geração, mas aproxima o processo de evolução duma selecção aleatória.

3.5 Algoritmos Específicos de Abordagens

Contrariamente aos algoritmos de partição iterativos apresentados na secção 3.4, que se baseiam em algoritmos de optimização genéricos e remetem o grosso das especificidades do problema concreto a resolver para a função de custo, os algoritmos descritos nesta secção são específicos de determinada abordagem ao problema de partição. Ou seja, parte das especificidades do problema de partição é transposta para a estratégia de controlo da evolução do algoritmo. Embora sejam algoritmos específicos duma abordagem, isso não significa que não possam ser adaptados a outras abordagens.

As especificidades de determinada abordagem ao problema de partição em componentes de *hardware* e de *software* devem-se em grande medida ao facto de a partição depender de aspectos físicos da arquitectura alvo considerada. Os limites impostos ao tamanho da ligação entre componentes e à quantidade de recursos disponíveis em cada componente, o mecanismo de comunicação entre componentes e a heterogeneidade dos componentes são aspectos a considerar na partição *hardware/software*.

Tipo *Greedy*

Um algoritmo iterativo do tipo *greedy* desloca um objecto duma partição para outra, desde que o valor devolvido pela função de custo para a solução de partição resultante seja inferior ao custo da solução actual. Como o algoritmo básico de *greedy* pára em mínimos locais da função de custo, a sua aplicação é pouco atractiva e fez com que fosse estendido. A abordagem Vulcan aplica uma extensão do algoritmo básico de *greedy*, com o objectivo de garantir que a solução final verifica todos os requisitos de desempenho exigidos ao sistema [GM94][GM96]. No caso de se pretender uma implementação com um componente de *hardware* e outro de *software*, o algoritmo de partição é apresentado de forma sintética na figura 3.6.

O algoritmo arranca com uma solução de partição totalmente em *hardware*, que depois é melhorada pelo deslocamento de objectos para *software*, desde que o custo da solução melhore e os requisitos de desempenho continuem a ser verificados. A função $Desloca(P, o_i)$ gera uma nova partição P', pelo deslocamento do objecto o_i para a outra partição. Quando se desloca um objecto, o algoritmo procura deslocar outros objectos próximos dele. Para isso recorre à função $Sucessores(o_i)$ que determina o conjunto de objectos que sucedem ao objecto o_i no grafo do sistema. A função $VerificaDesempenho(P)$ indica se a solução de partição P verifica ou não todos os requisitos de desempenho exigidos ao sistema.

```
algGreedy (O, Fcusto) ≡

    H = O, S = ∅, P = H ∪ S // H=partição de hw, S=partição de sw
                            // Todos os objectos de O em hw (O = {o_i})
    repetir
        P_antigo = P
        para o_i ∈ H fazer
            TentarDeslocamento(P, o_i)
        fpara
    até P_antigo = P

// Tentar o deslocamento dum objecto para a outra partição

TentarDeslocamento(P, o_i) ≡
    P' = Desloca(P, o_i)
    se (VerificaDesempenho(P') = VERDADE) E (Fcusto(P') < Fcusto(P)) então
        P = P'
        para o_j ∈ Sucessores(o_i) fazer
            TentarDeslocamento(P, o_j)
        fpara
    fse
```

Figura 3.6: Algoritmo de partição do tipo *greedy* usado na abordagem Vulcan.

Pesquisa Binária Condicionada

O algoritmo iterativo de pesquisa binária condicionada[7] (PBC) foi pensado para resolver o problema da partição em dois componentes, um de *hardware* e outro de *software*, com o objectivo de obter a solução que ocupando o espaço mínimo em *hardware* atinge o desempenho exigido [VGG94].

Encarando o problema de partição como uma tarefa em que se procura a solução de partição com custo mínimo e que emprega a menor quantidade de recursos de *hardware* possível, a estratégia do algoritmo de partição é definida por

> *Com o condicionalismo imposto ao espaço ocupado em hardware a variar desde zero (solução totalmente em software) até ao valor máximo $TamMaxHw$ (solução totalmente em hardware), procurar a primeira solução com custo mínimo.*

De acordo com esta definição, o objectivo do algoritmo é determinar o menor valor do espaço ocupado em *hardware*, designado por condicionalismo c_{tamHw}, que atinge o desempenho exigido. Se um dado valor de c_{tamHw} minimizar a função de custo ($Fcusto$), estabelecendo a solução para o problema de partição, qualquer valor maior que c_{tamHw} e menor que a quantidade de recursos de *hardware* disponível ($TamMaxHw$), origina uma solução válida. Desta observação resulta que o problema de minimização do espaço ocupado em *hardware*, respeitando um condicionalismo, se transforma na tentativa de encontrar o primeiro zero na

[7] *Binary Constraint-Search*, na terminologia inglesa.

sequência de custo. Por sequência de custo entende-se uma sequência de valores da função de custo, em que a cada zero corresponde uma solução exequível. O resultado é um problema idêntico ao da pesquisa num *array* ordenado, resolúvel por uma **pesquisa binária**.

Em alternativa a uma estratégia que em cada iteração procura melhorar ligeiramente um dos objectivos[8] sem piorar muito o outro, optou-se por uma estratégia em que se "relaxa" um dos objectivos. Assim, em vez de se tentar minimizar o espaço ocupado em *hardware*, apenas se pretende mantê-lo abaixo dum certo valor (c_{tamHw}). Deste modo, quando o algoritmo iterativo procura a solução óptima dispõe duma maior flexibilidade para lidar com os dois aspectos envolvidos na partição e o problema de minimização dos recursos de *hardware* assume a seguinte definição:

> *Dada uma solução de partição P, um algoritmo de partição iterativo PartAlg, os requisitos Req e a função de custo Fcusto, o problema da minimização dos recursos de hardware e simultâneo respeito pelos condicionalismos consiste no cálculo do menor* c_{tamHw} *para o qual*
>
> $$Fcusto(PartAlg(P, Req, c_{tamHw}, Fcusto()), Req, c_{tamHw}) \;=\; 0 \qquad (3.4)$$

A figura 3.7 ilustra o algoritmo de pesquisa binária condicionada, aplicado a uma sequência de custo definida a partir da gama de valores do condicionalismo imposto ao espaço ocupado em *hardware*. As variáveis inf e sup são os dois valores do condicionalismo que delimitam a janela onde pode existir a solução de partição com custo mínimo. A variável *meio* guarda o condicionalismo usado em cada iteração e coincide com o ponto intermédio da janela anterior. Em cada instante, a melhor solução encontrada pelo algoritmo $PartAlg$ está guardada em P_{melhor}, significando isso que de entre todas as soluções analisadas é aquela que usa o menor condicionalismo. A função $PartAlg$ é um algoritmo de partição iterativo, como por exemplo *simulated annealing*, a que o algoritmo PBC recorre.

O algoritmo PBC é aplicável noutras abordagens da partição *hardware*/*software*, onde a métrica a minimizar não seja o espaço ocupado em *hardware* e/ou o requisito não seja o desempenho.

Partição com Ênfase na Comunicação

O algoritmo construtivo **PACE**[9], desenvolvido no âmbito da abordagem Lycos [MGK+97], baseia-se em programação dinâmica e apresenta como principal característica a ênfase dada à

[8] Minimizar o espaço ocupado em *hardware* **ou** atingir o desempenho exigido.
[9] ***P**artitioning **A**lgorithm with **C**ommunication **E**mphasis*, na terminologia inglesa.

```
algPBC (P, Req, TamMaxHw, Fcusto) ≡

    inf = 0, sup = TamMaxHw
    enquanto (inf < sup) fazer
        meio = (inf + sup + 1)/2      // meio é usado como c_tamHw
        P = PartAlg(P, Req, meio, Fcusto(P, Req, meio))
        se Fcusto(P, Req, meio) = 0 então
            sup = meio − 1
            P_melhor = P
        senão
            inf = meio
        fse
    fenquanto
    devolver (P_melhor)
```

Figura 3.7: Algoritmo de pesquisa binária condicionada (PBC).

comunicação entre os objectos do sistema. Os objectos manipulados pelo algoritmo são blocos de código do modelo do sistema, designados por BBEs[10] [Knu95][KM96b]. Para efectuar a partição, cada objecto é anotado com o espaço que ocupa em *hardware*, a aceleração obtida ao deslocá-lo para *hardware* e a aceleração adicional resultante da redução da comunicação entre *hardware* e *software* (figura 3.8). Obtém-se uma redução na comunicação entre *hardware* e *software*, quando os objectos adjacentes ao objecto deslocado para *hardware* também se encontram em *hardware*. O modelo de comunicação aplicado no algoritmo PACE limita a comunicação entre objectos de *hardware*, sem que haja intervenção do *software*, aos objectos adjacentes.

Com uma arquitectura alvo composta por um componente de *hardware* e um microprocessador, o problema de partição é formulado da seguinte maneira:

> *O processo de partição é a pesquisa de sequências de objectos não sobrepostas, a atribuir à partição de hardware, que resultem na maior aceleração para o sistema e não ocupem mais do que o espaço disponível no componente de hardware.*

Para determinar o melhor conjunto de sequências a atribuir a *hardware*, utiliza-se o algoritmo descrito na figura A.3 do apêndice A. O algoritmo não selecciona duas ou mais sequências com o mesmo objecto porque, ao pesquisar as sequências, elas estão organizadas em grupos ordenados. Um grupo é composto por todas as sequências que terminam no mesmo objecto. Para o sistema introduzido na figura 3.8, os grupos e a sua ordenação são definidos na tabela 3.1. Nesta tabela, o espaço ocupado em *hardware* por uma sequência $S(o_i : o_j)$ obtém-se somando o espaço ocupado por todos os BBEs o_i até o_j e a aceleração, resultante de se deslocar para *hardware* a mesma sequência, calcula-se somando a aceleração associada a todos os BBEs com a aceleração associada a todas as ligações da sequência.

[10] **B**locos **B**ásicos de **E**scalonamento.

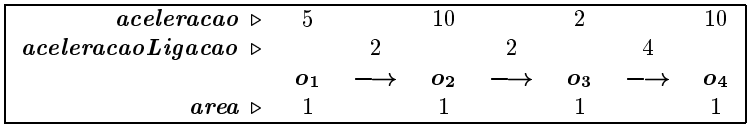


Figura 3.8: Um sistema definido por quatro objectos, com as anotações utilizadas pelo algoritmo PACE.

Grupo	*Sequência*	*BBEs*	*Espaço*	*Aceleração*
1	$S(o_1 : o_1)$	o_1	1	5
2	$S(o_1 : o_2)$	o_1, o_2	2	17
	$S(o_2 : o_2)$	o_2	1	10
	$S(o_1 : o_3)$	o_1, o_2, o_3	3	21
3	$S(o_2 : o_3)$	o_2, o_3	2	14
	$S(o_3 : o_3)$	o_3	1	2
	$S(o_1 : o_4)$	o_1, o_2, o_3, o_4	4	35
4	$S(o_2 : o_4)$	o_2, o_3, o_4	3	28
	$S(o_3 : o_4)$	o_3, o_4	2	16
	$S(o_4 : o_4)$	o_4	1	10

Tabela 3.1: Definição dos grupos de sequências e da ordenação usados no algoritmo PACE, para o sistema da figura 3.8.

Após efectuar a partição dos objectos, o algoritmo tem que reconstruir o conjunto das sequências que constituem a partição de *hardware*. Com os grupos de sequências definidos e ordenados como se mostra na tabela 3.1, procura-se a melhor sequência[11] do grupo com o maior índice (4 no exemplo em discussão) que ocupe um espaço a não superior ao disponível ($1 \leq a \leq maxArea$). Supondo que a sequência do grupo que introduz a maior aceleração é $S(o_i : o_4)$, duas situações podem ocorrer: (i) o espaço ocupado pela sequência é igual ao espaço disponível ou (ii) o espaço ocupado pela sequência (a) é inferior ao espaço disponível. No primeiro caso, a sequência $S(o_i : o_4)$ será a solução de partição. No segundo caso, procura-se no grupo $i-1$ a melhor sequência, que ocupe um espaço a_r não superior ao restante ($a_r = maxArea - a$). Deste modo garante-se a escolha dum conjunto de sequências sem objectos repetidos. O processo repete-se até se esgotar o espaço disponível. Quando isto ocorrer, a solução de partição é composta pelo conjunto das melhores sequências encontradas. Para o sistema da figura 3.8, se o espaço disponível em *hardware* fosse 3, a partição de *hardware* obtida pelo algoritmo coincidiria com a sequência $S(o_2 : o_4) = \{o_2, o_3, o_4\}$ e a aceleração resultante seria 28.

Partição *Hardware/Software* Direccionada pela Urgência Global e Fase Local

O algoritmo construtivo **GCLP**[12] faz parte duma metodologia de desenvolvimento de sistemas embebidos, para os quais se desejam implementações de baixo custo que obedeçam aos requisitos de desempenho exigidos, usando para isso uma arquitectura alvo heterogénea com-

[11] A melhor sequência é a que induz a maior aceleração.
[12] ***G**lobal **C**riticality/**L**ocal **P**hase*, na terminologia inglesa.

posta por um processador e *hardware* dedicado (ASIC) [KL94]. Em cada etapa do processo de partição, o algoritmo GCLP aplica a função de proximidade mais apropriada a esse momento. A função de proximidade a aplicar depende da maior ou menor necessidade de optimizar o desempenho e da heterogeneidade do sistema.

O problema de partição que se pretende resolver com o algoritmo GCLP é definido do seguinte modo:

> *O processo de partição procura atribuir cada nodo do grafo do sistema a hardware ou a software e estabelecer o instante de arranque desses nodos (escalonamento), considerando as limitações impostas à implementação e o custo de comunicação, com o objectivo de minimizar o espaço ocupado em hardware.*

As limitações do projecto são o requisito de desempenho (latência) e os recursos disponíveis em *hardware* e em *software* (memória).

A função de proximidade aplicada em cada etapa do processo de partição é seleccionada de acordo com duas medidas: uma medida que é uma estimativa da urgência temporal global[13] (***CG***) e uma medida da heterogeneidade dos nodos (***FL***). O algoritmo aplica a medida CG para manter a exequibilidade da implementação e usa a fase local FL para obter optimização local, tendo em atenção as características dos nodos.

O algoritmo GCLP, esquematizado na figura A.5 do apêndice A, possui um ciclo exterior relativo ao número de nodos. Em cada iteração deste ciclo, ou seja, em cada etapa do processo de partição, o algoritmo selecciona um nodo, determina a partição a que ele é atribuído e calcula o instante em que entra em funcionamento. Para efectuar a atribuição consideram-se as medidas CG e FL.

A medida CG, baseada na percentagem de nodos escalonados e não escalonados e nos requisitos de desempenho, orienta a escolha da função de proximidade que controla a atribuição dos nodos a uma das partições. Se o tempo for crítico, a medida CG favorece a selecção duma função que minimiza o tempo de conclusão da execução, senão favorece a selecção duma função que minimiza o espaço ocupado em *hardware*. Para escolher uma função de entre as duas disponíveis, o valor de CG é comparado com um nível de decisão, como se mostra na figura 3.9.

Uma atribuição de nodos que considerasse apenas a medida CG, dificilmente seria globalmente óptima porque os nodos são heterogéneos e essa medida não dependeria das características do nodo envolvido em cada etapa.

[13] Urgência temporal representa a maior ou menor necessidade de optimizar o desempenho num determinado momento.

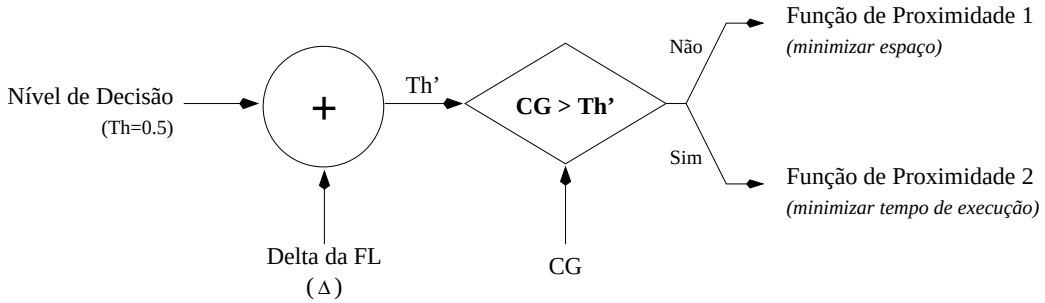


Figura 3.9: Selecção da função de proximidade a aplicar em cada etapa do algoritmo GCLP.

A medida $CG(k)$, o mesmo é dizer o valor de CG quando já foram atribuídos k nodos, é definida como a probabilidade de um nodo não escalonado ser atribuído a *hardware*, por forma a atingir a exequibilidade global da solução de partição. Quanto maior for CG, mais nodos deverão ser deslocados para *hardware*, para tornar a solução de partição exequível.

A preferência local por um tipo de nodo, em determinada etapa, é traduzida no *delta da fase local FL* (Δ). O valor de Δ afecta o nível de decisão (Th') usado na selecção da função de proximidade (figura 3.9). Para obter o valor de FL definiram-se três tipos de nodos: extremidade, repelente e normal. Um nodo é uma **extremidade** duma implementação I, ou seja, é um mau candidato a essa implementação, se consumir muitos recursos preciosos de I e poucos recursos preciosos da implementação complementar $\overline{I}$. A intensidade com que um nodo é uma extremidade, é quantificada na medida de extremidade. Um nodo n_i é um **repelente** a uma implementação se não for uma extremidade e possuir um valor efectivo de repelente (VR_i) não nulo, sendo VR uma medida dos recursos que se poupam quando se trocam dois nodos idênticos do tipo repelente entre implementações complementares. Um nodo é considerado **normal**, se não for nem uma extremidade nem um repelente.

As características de cada nodo afectam o nível de decisão usado na escolha da função de proximidade ($Th' = 0.5 + \Delta$) através do *delta da fase local* (Δ). Se o nodo é do tipo extremidade, o *delta da fase local* assume um comportamento linear relativamente à medida de extremidade. Nos nodos do tipo repelente o *delta da fase local* assume o valor de VR ($Th' = 0.5 + VR$) e nos nodos normais o *delta da fase local* é nulo ($Th' = 0.5$).

3.6 Complexidade dos Algoritmos de Partição

Uma característica relevante nos algoritmos de partição, especialmente nos algoritmos iterativos, é a sua complexidade temporal. Isto porque uma complexidade elevada pode inviabilizar a aplicação dum algoritmo num problema que exija a análise de muitas alternativas de partição.

A **complexidade temporal (assimptótica)** dum algoritmo pode ser representada por uma função dos parâmetros que quantificam a dimensão do problema a resolver e constitui uma boa

aproximação ao tempo de cálculo exigido pela sua resolução, quando a dimensão do problema assume valores muito elevados [Knu97][AG94]. Do mesmo modo, a **complexidade espacial (assimptótica)** dum algoritmo pode ser representada por uma função dos parâmetros que quantificam a dimensão do problema e constitui uma boa aproximação ao espaço exigido pela execução deste algoritmo, quando a dimensão do problema assume valores muito elevados. O apêndice B apresenta a notação-$\mathcal{O}$ utilizada para expressar a complexidade dos algoritmos de partição.

Por exemplo, num problema cuja dimensão é medida pelo valor de n, se o tempo de cálculo utilizado por um algoritmo que resolve este problema for $k1 * n^2 + k2$ [14], a complexidade temporal do algoritmo é expressa na notação-$\mathcal{O}$ por $\mathcal{O}(n^2)$. De acordo com a definição da notação-$\mathcal{O}$, o significado de $\mathcal{O}(n^2)$ é o de que existe uma constante H e um valor de n (n_0) a partir do qual o tempo de cálculo é menor ou igual a $H * n^2$:

$$k1 * n^2 + k2 \ \leq \ H * n^2 \qquad (n \geq n_0) \tag{3.5}$$

A complexidade temporal dos algoritmos introduzidos neste capítulo, aplicando a notação-$\mathcal{O}$, é sintetizada na tabela 3.2.

Quando aplicado na partição dum sistema composto por n objectos, a complexidade temporal do algoritmo de Kernighan/Lin é $\mathcal{O}(n^3)$, mas após algumas alterações do algoritmo assume o valor de $\mathcal{O}(n^2 * log(n))$. A complexidade passa a ser $\mathcal{O}(n^2)$ na extensão KL/não-balanceado, $\mathcal{O}(n)$ na extensão KL/FM e $\mathcal{O}(n * log(n))$ na extensão KL/funcional.

Para o algoritmo de *simulated annealing* apresentado na figura 3.3, a complexidade temporal depende das funções *CondicaoParagem*, *Equilibrio*, *Aceitacao* e *ReducaoTemperatura*. Deste modo, a dependência da complexidade temporal em relação ao número de objectos do sistema é expressa por um polinómio de grau desconhecido *a priori*.

Analisando o algoritmo PT descrito na figura 3.4 conclui-se que, a tarefa responsável pela maior parcela do tempo de cálculo exigido por cada iteração é a pesquisa da melhor alternativa de partição em toda a vizinhança da solução actual. O elevado tempo de cálculo é uma consequência de ter que se estimar valores para as métricas envolvidas na função de custo, de modo a qualificar todas as alternativas de partição pertencentes à vizinhança. Como o tamanho da vizinhança, num problema com n objectos e p partições, é $n * (p - 1)$, a complexidade temporal do algoritmo de pesquisa tabu apresentado é $\mathcal{O}(n * p)$. Se o problema de partição considerar apenas 2 partições, a complexidade é $\mathcal{O}(n)$.

A complexidade temporal do algoritmo EG é influenciada pela condição de paragem, repre-

[14] Considerando que $k1$ e $k2$ são constantes, ou seja, são parâmetros que não dependem de n.

Tipo	***Algoritmo***	***Complexidade Temporal***
Construtivo	Exaustivo	$\mathcal{O}(2^n)$
	Crescimento de grupos (Selecção aleatória)	$\mathcal{O}(n)$
	Agrupamento hierárquico	$\mathcal{O}(n^2)$
	Programação linear com inteiros	muito elevada
	PACE	$\mathcal{O}(n^2 * e)$
	GCLP	$\mathcal{O}(n * a)$
Iterativo	Kernighan/Lin	$\mathcal{O}(n^2 * log(n))$
	KL/não-balanceado	$\mathcal{O}(n^2)$
	KL/FM	$\mathcal{O}(n)$
	KL/funcional	$\mathcal{O}(n * log(n))$
	Simulated annealing	$\mathcal{O}$*(polinómio(n))*
	Pesquisa tabu	$\mathcal{O}(n)$
	Evolução genética	elevada
	Greedy	$\mathcal{O}(n)$
	Pesquisa binária condicionada (‡)	$\mathcal{O}(PartAlg) * \mathcal{O}(log_2(1/d))$

Notas:

n	→ número de objectos
e	→ espaço num componente
a	→ número de arcos do grafo
‡	→ o algoritmo recorre a outro algoritmo iterativo, designado por $PartAlg$
d	→ factor de precisão

Tabela 3.2: Complexidade dos algoritmos de partição, considerando soluções com duas partições.

sentada pela função $Terminar$ no algoritmo da figura 3.5. Contudo, para obter boas soluções de partição o tempo de cálculo médio é seguramente muito elevado. A complexidade espacial, além de elevada, é bastante superior à dos outros algoritmos iterativos, uma vez que o algoritmo EG guarda um conjunto de soluções de partição por iteração.

No pior dos casos, a complexidade temporal do algoritmo de pesquisa binária condicionada é igual ao produto da complexidade do algoritmo iterativo a que ele recorre ($\mathcal{O}(PartAlg)$) pela complexidade inerente ao próprio algoritmo PBC ($\mathcal{O}(log_2(1/d))$).

A complexidade temporal do algoritmo PACE é $\mathcal{O}(n^2 * e)$ e a complexidade espacial é $\mathcal{O}(n * e)$, com n a ser o número de objectos e e o espaço disponível em *hardware*. Com esta complexidade temporal, a partição de sistemas de grande dimensão é morosa.

No pior dos casos, a complexidade temporal do algoritmo GCLP é $\mathcal{O}(n * a)$, em que n é o número de nodos e a o número de arcos. Para sistemas predominantemente de dados, como os sistemas de processamento de sinal ou de imagem, em que os número de nodos e de arcos se aproximam, a complexidade do algoritmo é quadrática ($\mathcal{O}(n^2)$).

3.7 Algoritmos Seleccionados

Embora gerem soluções de partição de elevada qualidade, o algoritmo exaustivo e os métodos de programação linear com inteiros não foram opção para algoritmo construtivo do presente trabalho porque exigem um tempo de cálculo muito elevado. Mais, a formulação PLI do problema de partição é difícil de obter e de automatizar. Como os algoritmos PACE e GCLP, que tal como os métodos PLI dispensam a utilização dum algoritmo iterativo, são demasiado específicos duma abordagem e consideram apenas soluções com duas partições, só por si não se mostraram atractivos para adaptação ao presente trabalho.

Apesar da estratégia de construção das soluções ser diferente, pode dizer-se que os algoritmos de agrupamento hierárquico e de crescimento de grupos permitem obter os mesmos resultados. Como o algoritmo de crescimento de grupos é mais fácil de implementar, foi o algoritmo construtivo seleccionado neste trabalho. Embora seja um algoritmo simples, a qualidade das soluções de partição e o tempo de cálculo necessário para as obter estão dependentes da função de proximidade utilizada. Com uma função de proximidade mais elaborada e que use estimativas de métricas mais precisas, aumenta-se o tempo de cálculo e a qualidade das soluções.

Embora o algoritmo de Kernighan/Lin aplique uma estratégia de evolução do tipo *hill-climbing*, a sua capacidade para evitar mínimos locais da função de custo é limitada. O algoritmo de *simulated annealing* apresenta maiores potencialidades para atingir soluções óptimas do que os algoritmos do tipo *greedy* e de Kernighan/Lin, mas exige um tempo de cálculo muito elevado. Ao usar pesquisa local numa vizinhança, com dimensão reduzida pelos deslocamentos proibidos, o tempo de cálculo necessário ao algoritmo de pesquisa tabu é reduzido significativamente. Contrariamente ao que acontece no algoritmo de *simulated annealing*, em que os deslocamentos que pioram o custo da solução de partição são seleccionados aleatoriamente, na pesquisa tabu esses deslocamentos são escolhidos de forma determinística, tendo em consideração o historial dos deslocamentos efectuados. Deste modo, sem aumentar muito o tempo de cálculo, a pesquisa tabu possui capacidades acrescidas para convergir para a solução óptima.

Os algoritmos de evolução genética são eficientes na redução do espaço de projecto, mas são menos bons a convergir para a solução de partição óptima. Uma forma de rentabilizar as potencialidades dum algoritmo EG consiste em aplicá-lo em conjunto com outro algoritmo, como o de Kernighan/Lin, *simulated annealing* ou pesquisa tabu. O algoritmo de pesquisa binária condicionada também exige a cooperação com outro algoritmo iterativo e permite apenas uma partição em dois componentes.

Considerando a qualidade esperada para as soluções de partição a obter com os algoritmos

iterativos discutidos, a opção sobre que algoritmo iterativo seleccionar recai sobre os algoritmos de *simulated annealing* e de pesquisa tabu. Na questão do tempo de cálculo, o algoritmo de pesquisa tabu sai claramente vencedor [EKP98a]. Em termos de implementação e utilização, as vantagens dividem-se: o algoritmo de *simulated annealing* é mais fácil de adaptar a qualquer problema de partição, mas o algoritmo de pesquisa tabu é mais fácil de parametrizar. Em conclusão, como existe equilíbrio na qualidade das soluções obtidas e nos aspectos de implementação/utilização, mas o algoritmo de pesquisa tabu exige um tempo de cálculo significativamente inferior, a escolha recaiu sobre o algoritmo de pesquisa tabu.

3.8 Função de Proximidade

Em cada instante do processo de partição construtivo e de acordo com as partições parciais estabelecidas até esse momento, a função de proximidade tem por objectivo indicar a partição mais adequada para atribuir os objectos por agrupar.

Função de Proximidade usada no Agrupamento por Fases

O processo de partição descrito em [BRX93] [BS94] considera uma arquitectura alvo contendo um componente de *software* e vários componentes de *hardware*. Esta abordagem de partição aplica um algoritmo construtivo de agrupamento por fases. A primeira fase do processo de agrupamento decorre sob controlo duma função de proximidade que tem por objectivo agrupar os objectos com implementação mais similar (equação 3.6).

$$Fprox(o_i, o_j) = M_{com}(o_i, o_j) + M_{similar}(o_i, o_j) + M_{partilhaUF}(o_i, o_j) \tag{3.6}$$

em que

- $\boldsymbol{M_{com}(o_i, o_j)}$ mede o custo da comunicação entre os objectos o_i e o_j;
- $\boldsymbol{M_{similar}(o_i, o_j)}$ mede a similaridade entre os dois objectos, com base no tipo de atribuições neles incluídas;
- $\boldsymbol{M_{partilhaUF}(o_i, o_j)}$ quantifica a possibilidade de partilhar unidades funcionais por parte das operações dos dois objectos.

Para estabelecer a similaridade entre objectos seguem-se as seguintes orientações: objectos que exibem o mesmo grau de paralelismo devem ser agrupados, objectos com dependência de dados devem ser agrupados e objectos mutuamente exclusivos e exibindo graus de paralelismo diferentes devem ficar separados.

A melhor linha de corte a aplicar à árvore de grupos c, obtida após uma sequência de s agrupamentos, é dada pela equação 3.7.

$$Fcorte(s,c) = M_{paralel}(s,c) + M_{atribSw}(s,c) + M_{areaAtraso}(s,c) \tag{3.7}$$

onde

- $\boldsymbol{M_{paralel}(s,c)}$ mede o potencial de paralelismo existente entre os objectos atribuídos ao mesmo grupo e o seu valor é definido pela relação entre o custo duma implementação em paralelo (ou *pipeline*) e a aceleração que dai resulta;
- $\boldsymbol{M_{atribSw}(s,c)}$ mede o nível de separação[15] entre os objectos que são bons candidatos para *software* e os objectos que são bons candidatos para *hardware*;
- $\boldsymbol{M_{areaAtraso}(s,c)}$ mede a relação entre o espaço ocupado pelos recursos e o tempo de execução, tendo em consideração a quantidade de recursos partilhados.

O processo de agrupamento decorre enquanto $Fcorte(s+1,c)$ for menor do que $Fcorte(s,c)$. Na segunda fase do processo de agrupamento, emprega-se uma função de proximidade cujo objectivo é melhorar o paralelismo da solução de partição, implementando em *pipeline* os objectos sem dependências de dados.

Função de Proximidade Seleccionada Dinamicamente

Na abordagem [KL94] aplicam-se duas funções de proximidade em conjunto com o algoritmo construtivo GCLP (equação 3.8). Das duas funções de proximidade disponíveis, uma delas procura minimizar o espaço ocupado pelos recursos seleccionados ($Fprox_1$ na equação 3.9) e a outra procura maximizar o desempenho ($Fprox_2$ na equação 3.10).

$$Fprox = \begin{cases} Fprox_1 & \text{, se } (CG \leq Th') \\ Fprox_2 & \text{, nos outros casos} \end{cases} \tag{3.8}$$

$$Fprox_1 = \begin{cases} I_i \equiv S & \text{, se } \frac{Area^i_{sw}}{maxArea_{sw}} \leq \frac{Area^i_{hw}}{AreaResta_{hw}} \\ I_i \equiv H & \text{, nos outros casos} \end{cases} \tag{3.9}$$

em que

- $\boldsymbol{H}$ ($\boldsymbol{S}$) designa o componente de *hardware* (*software*) da implementação;
- $\boldsymbol{I_i}$ é o tipo de implementação do objecto o_i;

[15] Dois objectos dizem-se separados se estiverem colocados em grupos distintos.

- $\boldsymbol{Area_{hw}^{i}}$ ($\boldsymbol{Area_{sw}^{i}}$) é o espaço ocupado por o_i em *hardware* (*software*);
- $\boldsymbol{maxArea_{sw}}$ quantifica o espaço disponível em *software*;
- $\boldsymbol{AreaResta_{hw}}$ é o espaço que resta em *hardware* no momento da atribuição de o_i a uma dada implementação.

$$Fprox_2 = \begin{cases} I_i \equiv H & , \text{ se } T_{finish}(o_i, H) < T_{finish}(o_i, S) \\ I_i \equiv S & , \text{ nos outros casos} \end{cases} \tag{3.10}$$

em que $T_{finish}(o_i, I_i)$ designa a estimativa do instante em que termina a execução de o_i quando atribuído à implementação I_i. Esta estimativa é calculada a partir do instante de conclusão da execução dos objectos antecessores de o_i, do instante de conclusão da execução do último objecto atribuído a I_i, do tempo de comunicação entre o_i e os seus antecessores e do tempo de computação de o_i.

Conforme o algoritmo avança, a função de proximidade $Fprox_1$ indica com mais frequência que os objectos devem ser atribuídos a *software*, uma vez que o espaço que resta em *hardware* vai diminuindo. Ao aplicar duas funções de proximidade, resolve-se o dilema colocado pelos objectivos contraditórios definidos para o processo de partição: obter uma solução que ocupa um espaço mínimo e atingir um desempenho máximo.

Selecção da Função de Proximidade

A função de proximidade descrita em [BRX93, BS94] procura agrupar os objectos que apresentem a maior troca de informação e a implementação mais semelhante. Agrupar objectos com implementação semelhante não é relevante no presente trabalho, uma vez que se assume não haver partilha de unidades funcionais por parte de objectos atribuídos a *hardware*. A abordagem [KL94] optou por duas funções de proximidade, estando cada uma associada a um objectivo distinto: tempo de execução mínimo ou quantidade de recursos mínima. Nesta abordagem, as características do objecto a atribuir influenciam a função seleccionada em cada momento, o que, embora de forma diferente, também ocorre no presente trabalho: se o objecto a atribuir for uma variável usa-se a função F_{var}, senão utilizam-se as funções F_{epHwSw} e F_{epHw}.

A função de proximidade $F_{var} = f(M_{comm}, M_{area})$ atribui uma variável à partição que apresente a maior intensidade de comunicação (M_{comm}) entre a variável e os estados programa já atribuídos a essa partição, desde que o espaço ocupado pela variável (M_{area}) seja compatível com uma implementação de *hardware*. Caso contrário, a função indica que a variável deve ser atribuída a *software*.

A função de proximidade $F_{epHwSw} = f(M_{compSw}, M_{compHw}, M_{comm})$ indica qual é o tipo da partição mais adequada para atribuir um estado programa: *software* ou *hardware*. No primeiro caso o estado programa é imediatamente atribuído à partição de *software*, enquanto no segundo caso recorre-se à função $F_{epHw} = f(M_{area}, M_{comm})$ para indicar qual é a melhor partição de *hardware*. A função F_{epHwSw} garante que o objecto é atribuído à partição que conduz ao melhor desempenho, através dum tempo de computação mínimo (M_{compSw} ou M_{compHw}) e/ou duma intensidade de comunicação máxima dentro da partição (M_{comm}). Por seu lado, a função F_{epHw} procura atribuir os objectos que são bons candidatos para *hardware* à partição (de *hardware*) com a qual apresentam a maior intensidade de comunicação e/ou à partição que esteja menos preenchida.

As três funções mencionadas, F_{var}, F_{epHwSw} e F_{epHw}, são implementadas através da função de proximidade $selectBestObjAssign$, descrita na secção 6.4.

3.9 Função de Custo

Quando a solução de partição resulta dum processo de partição iterativo, para chegar à solução óptima o algoritmo de partição apoia-se numa medida da qualidade das alternativas de partição que analisa. Esta medida, a que se chama custo da solução, é obtida pela função de custo[16] e resulta da combinação de várias métricas num único valor. Além de funcionar como medida da qualidade duma alternativa de partição, o custo pode ser utilizado para indicar se a alternativa de partição é ou não uma solução exequível. Por exemplo, uma alternativa de partição que exige mais recursos de *hardware* do que os disponíveis na arquitectura alvo não é exequível.

Da análise de diversas abordagens, resulta uma função de custo tipo que é definida por um somatório de termos que dependem de métricas. Os termos podem ser as métricas afectadas por um peso ou expressões envolvendo as métricas. Uma situação em que se usam expressões envolvendo as métricas ocorre quando se pretende representar o sucesso com que se respeitam as limitações impostas à implementação do sistema. É comum atribuírem-se pesos aos termos que contribuem para a função de custo. Deste modo, para se obter uma solução que ocupe um espaço mínimo, atribui-se ao termo que depende da métrica *espaço ocupado pelos recursos* um peso superior ao peso de qualquer outro termo. Para obter uma solução com o maior desempenho possível, o termo que representa o incumprimento do desempenho tem associado um peso que é superior a qualquer outro.

Seguindo os princípios anteriores, uma função de custo elementar é expressa por uma soma pesada das métricas M_i mais relevantes envolvidas no processo de partição (equação 3.11),

[16] Alguns autores usam o termo **função objectivo** para designar função de custo.

mas poderá assumir uma forma mais complexa como a da equação 3.12.

$$Fcusto = \sum_i K_i * M_i \tag{3.11}$$

$$Fcusto = \sum_i K_i * f_i(M_i) \tag{3.12}$$

Nas próximas secções faz-se uma síntese dum conjunto significativo de funções de custo referenciadas na bibliografia, algumas das quais são apresentadas com maior detalhe no apêndice C.

Função de Custo da Abordagem de Peng e Eles

A função de custo utilizada em [EPD94] [EKP98a] aplica-se a um processo de partição que visa gerar uma partição de *hardware* (H) e outra de *software* (S). A função favorece o agrupamento dos objectos que possuem mais características em comum e apresentam uma troca de informação elevada. O processo de partição é orientado por um função de custo que inclui três termos, como consta na equação 3.13.

$$Fcusto = K_1 * M_{comHwSw} + K_2 * M_{compCom} + K_3 * M_{adeqImp} \tag{3.13}$$

Os pesos associados às métricas, constantes K_1 a K_3, são escolhidos de modo a estabelecer uma relação equilibrada entre a contribuição das três métricas. As métricas utilizadas na função de custo possuem o seguinte significado:

- $\boldsymbol{M_{comHwSw}}$ mede a comunicação entre *hardware* e *software*, sendo definida pelo somatório da intensidade de comunicação para as ligações entre a partição de *hardware* e de *software*; para a intensidade de comunicação contribuem o número de transacções e a largura da informação envolvida em cada transacção;
- $\boldsymbol{M_{compCom}}$ quantifica o valor médio, no conjunto de todos os objectos atribuídos a *hardware*, da relação entre a comunicação e a carga computacional dum objecto; a carga computacional dum objecto, é definida à custa do número de execuções do objecto e do peso computacional de cada operação contida no objecto;
- $\boldsymbol{M_{adeqImp}}$ mede o grau de inadequação dos objectos para o tipo de implementação a que foram atribuídos; quantos mais forem os objectos que, sendo adequados a uma dada implementação (*hardware* ou *software*), estiverem atribuídos à implementação complementar (*software* ou *hardware*), maior será o valor desta métrica; para calcular $M_{adeqImp}$

recorre-se à métrica M_{n2}^{i}, que faz o balanço entre as características que tornam o objecto o_i adequado para uma implementação de *hardware* (carga computacional relativa, uniformidade de operações e potencial de paralelismo elevados) e as características que o tornam adequado para uma implementação de *software* (quantidade de operações adequadas para serem implementadas em *software* elevada).

Durante a pesquisa da solução óptima, o algoritmo iterativo de partição procura minimizar o valor devolvido pela função de custo e simultaneamente respeitar os limites impostos ao espaço ocupado em *hardware* e em *software*.

Função de Custo para Sistemas Tempo Real

Nos sistemas embebidos tempo real o cumprimento dos requisitos temporais influencia de forma determinante o processo de partição. Nestes sistemas, em vez de se usar uma função de custo que combina várias métricas, é pertinente aplicar separadamente um conjunto de métricas (equação 3.14), que representam os aspectos temporais ou não funcionais do sistema e controlam a exploração do espaço de projecto. A abordagem [DH94] descreve como é que através duma métrica M_p, designada por factor de exequibilidade, se verifica se um sistema é ou não exequível.

$$Fcusto = \overrightarrow{M} = \begin{bmatrix} M_1 \\ \dots \\ M_p \\ \dots \\ M_n \end{bmatrix} \tag{3.14}$$

O processo de partição *hardware/software* recebe como entradas a descrição do sistema, segundo um conjunto de funções do tipo *time-critical*, e os requisitos/limitações associados a cada função. Nos sistemas de tempo real as características das funções tipo *time-critical* são geralmente especificadas por um trio (a, d, t), em que a é o instante de activação, d é o instante em que a função deve terminar a actividade e t é o intervalo entre repetições da função. Como o cumprimento dos requisitos por parte das funções implementadas em *hardware* é mais fácil de verificar, o esforço concentra-se nas funções implementadas em *software*.

O factor de exequibilidade M_p, para um microprocessador p, é definido por

$$M_p = \begin{cases} \frac{TR_p - TR_L}{TR_U - TR_L} & , \; se \; (TR_p - TR_L) < (TR_U - TR_L) \\ 1 & , \text{ nos outros casos} \end{cases} \tag{3.15}$$

$\boldsymbol{TR_p}$ define o débito do microprocessador p e $\boldsymbol{TR_T}$ representa o valor mínimo de TR_p para que o microprocessador p consiga escalonar todas as tarefas que lhe foram atribuídas. Os

limites superior (TR_U) e inferior (TR_L) para TR_T são definidos pelas equações 3.16 e 3.17, respectivamente.

$$TR_U \;=\; k(N) * \sum_{i=1}^{N} v_i \tag{3.16}$$

$$TR_L \;=\; max_{n=1}^{N} \left[\; \sum_{i=1}^{n} g_i * v1_i \quad , \quad \sum_{i=1}^{n} h_i * v2_i \; \right] \tag{3.17}$$

em que

- a tarefa $\boldsymbol{T_i}$, atribuída a *software* e exigindo para a sua execução $\boldsymbol{c_i}$ instruções, tem associada a função $\boldsymbol{F_i = (a_i, d_i, t_i)}$, uma função crítica em termos temporais;
- $\boldsymbol{N}$ é o número de tarefas atribuídas ao microprocessador p e $\boldsymbol{k(N)}$ é um factor dependente de N;
- a equação 3.17 pressupõe que as tarefas T_i, com $i \in [1:n]$, estão ordenadas por ordem crescente do valor d_i, logo d_n corresponde à tarefa com conclusão mais posterior;
- $\boldsymbol{v_i}$, $\boldsymbol{v1_i}$, $\boldsymbol{v2_i}$ é o número de instruções da tarefa T_i a executar por segundo durante o intervalo $[a_i : d_i]$, $[a_i : d_n]$, $[a_n : d_n]$;
- $\boldsymbol{g_i}$, $\boldsymbol{h_i}$ é o número mínimo de vezes que a tarefa T_i tem que ser executada no intervalo $[a_i : d_n]$, $[a_n : d_n]$.

Se $M_p = 1$ o conjunto de tarefas atribuídas a p é exequível, se $M_p \leq 0$ o conjunto de tarefas não é exequível e se $0 < M_p < 1$ a probabilidade do conjunto de tarefas ser exequível é M_p.

Função de Custo da Abordagem Cosyma

A função de custo proposta em [HE98] para a abordagem Cosyma, tem por objectivo reduzir ao mínimo o tempo de execução do sistema e o espaço ocupado em *hardware* (equação 3.18). Nesta equação, B designa o modelo do sistema e $f_2(M_1)$ é um factor aplicado à métrica espaço (M_2), dependente da métrica desempenho (M_1). Adicionando algum detalhe obtém-se a função de custo da equação 3.19.

$$Fcusto(B) = K_1 * f_1(M_1) + K_2 * f_2(M_1) * M2 \tag{3.18}$$

$$Fcusto(B) = K_T * T_{execNorm}(B) + K_{area} * \omega_{area} * \frac{Area_{HW}(B)}{\overline{A}} \tag{3.19}$$

O termo $Area_{HW}(B)$ é a estimativa do espaço ocupado em *hardware*, calculada mais à frente com a equação 4.24, e $\overline{A}$ é um factor de normalização do espaço ocupado em *hardware*. O termo $T_{execNorm}(B)$, que representa o sucesso com que se atinge o desempenho exigido ao sistema, é definido na equação 3.20. Nesta equação, $T_{exec}(B)$ é a estimativa para o tempo de execução do sistema e $reqT$ é o requisito de desempenho.

$$T_{execNorm}(B) = \frac{\mid T_{exec}(B) - reqT \mid}{reqT} + 1 \tag{3.20}$$

Os factores K_T e K_{area} são constantes, enquanto ω_{area} é um factor que varia dinamicamente, ao longo do processo de partição, com o grau de aproximação ao desempenho exigido ao sistema. O comportamento de ω_{area}, tal como é definido na equação 3.21, garante que o peso do espaço no custo da solução de partição é nulo quando se está longe do requisito de desempenho ($\omega_{area} = 0$) e que à medida que o tempo de execução se aproxima do requisito de desempenho, o peso do espaço aumenta cada vez mais. Quando se atinge o desempenho exigido ao sistema, o valor de ω_{area} assume o valor máximo e o peso do espaço torna-se dominante na função de custo. O valor da constante T_{th} é seleccionado de acordo com os resultados experimentais, mas deve ter-se em atenção que a influência de T_{th} sobre ω_{area} é como se ilustra na figura 3.10 e que não faz sentido escolher $T_{th} \geq reqT$.

$$\omega_{area} = \begin{cases} \frac{T_{exec}(B) - T_{th}}{reqT - T_{th}} & , \text{ se } T_{th} \leq T_{exec}(B) \leq reqT \\ \frac{2 * reqT - T_{th} - T_{exec}(B)}{reqT - T_{th}} & , \text{ se } reqT \leq T_{exec}(B) \leq (2 * reqT - T_{th}) \\ 0 & , \text{ nos outros casos} \end{cases} \tag{3.21}$$

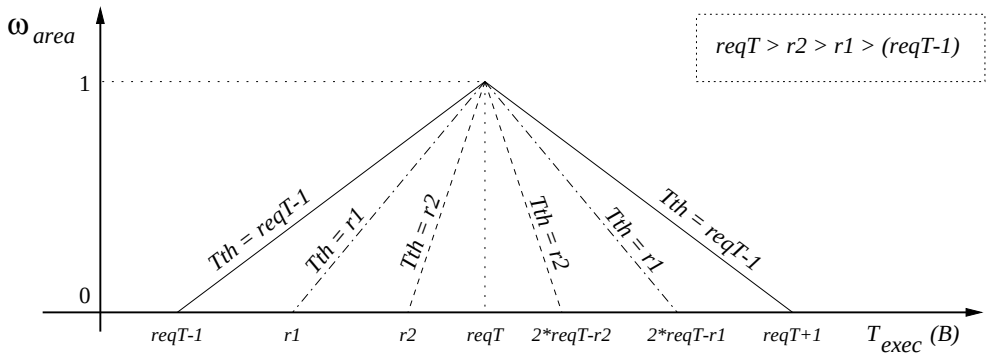


Figura 3.10: Comportamento do factor variável ω_{area} aplicado ao espaço em *hardware*, para alguns valores de T_{th}.

Com a função de custo apresentada, em que se aplica à métrica de espaço um peso que varia dinamicamente com a métrica de desempenho, obtêm-se soluções de partição que exigem menos recursos de *hardware* e atingem o mesmo desempenho que as soluções obtidas com uma função de custo equivalente, que não use o factor variável ω_{area}.

Função de Custo da Abordagem SpecSyn

Na abordagem SpecSyn [VG92][VGG94] a partição é do tipo *hardware/hardware* e tem por objectivo respeitar as limitações impostas pela arquitectura alvo e atingir o requisitos de desempenho. Para atingir este objectivo, desenvolveram uma função de custo que soma o grau de desrespeito pelos condicionalismos impostos pela arquitectura alvo com o grau de incumprimento do requisito de desempenho. O requisito é o tempo de execução e os condicionalismos são o número de componentes, o espaço ocupado por componente e o número de pinos por componente. Os sistemas são representados por um grafo de fluxo, em que os nodos correspondem aos objectos o_j e os arcos correspondem às ligações entre objectos $a_{j,k}$.

A função que quantifica a qualidade duma alternativa de partição P, constituída pelas partições $\{P_i\}$, é definida de forma sintética na equação 3.22 e de forma mais detalhada na equação 3.23.

$$Fcusto = K_1 * f_1(M_{area}) + K_2 * f_2(M_{pinos}) + K_3 * f_3(M_{chips}) + K_4 * f_4(M_{tempo}) \tag{3.22}$$

$$\begin{aligned} Fcusto = \; & k_a * \sum_i [100 * \frac{excessoArea(P_i)}{maxArea(P_i)}]^2 + \; k_p * \sum_i [100 * \frac{excessoPinos(P_i)}{maxPinos(P_i)}]^2 + \\ & + k_c * [100 * \frac{excessoChips}{maxChips}]^2 + \; k_t * \sum_j [100 * \frac{excessoTempo(o_j)}{maxTempo(o_j)}]^2 \end{aligned} \tag{3.23}$$

em que

- $excessoArea(P_i)$ mede o desrespeito do espaço ocupado em *hardware* pela partição P_i ($area(P_i)$) relativamente ao condicionalismo imposto a esse espaço ($maxArea(P_i)$);

$$excessoArea(P_i) = MAX\ (area(P_i) - maxArea(P_i)\ ,\ 0) \tag{3.24}$$

- $excessoPinos(P_i)$ mede o desrespeito do número de pinos necessário à partição P_i em relação ao condicionalismo que lhe é imposto pela arquitectura alvo ($maxPinos(P_i)$);
- $excessoChips$ mede o desrespeito do número de componentes seleccionado pela alternativa de partição relativamente ao número disponível na arquitectura alvo ($maxChips$); a expressão de $excessoPinos(P_i)$ e de $excessoChips$ é idêntica à de $excessoArea(P_i)$;
- k_a, k_p, k_c e k_t são constantes que representam o peso atribuído ao espaço, aos pinos, ao número de componentes e ao desempenho;

- $excessoTempo(o_j)$ mede o grau de incumprimento do tempo de execução de o_j relativamente ao valor que lhe é exigido pelo requisito do sistema;

$$excessoTempo(o_j) = MAX\ [\ execTempo(o_j) - maxTempo(o_j)\ ,\ 0\ ] \tag{3.25}$$

- $execTempo(o_j)$ é a estimativa do tempo de execução do objecto o_j, na qual se contabilizam o tempo de comunicação (dentro e fora dos componentes) e o tempo de computação necessário à execução das operações do objecto o_j;
- $maxTempo(o_j)$ é requisito de desempenho ajustado para a posição (do escalonamento) em que o_j conclui a sua execução.

De acordo com a função de custo da equação 3.23, uma alternativa de partição é uma solução óptima se lhe corresponder um custo nulo.

Função de Custo da Abordagem Vulcan

Na abordagem Vulcan, o processo iterativo de partição entre *hardware* e *software* é guiado pela função de custo apresentada na equação 3.26. A função de custo procura minimizar o espaço ocupado em *hardware* e a comunicação entre componentes, e usar o máximo possível de recursos de *software* [GM94] [GM96].

$$Fcusto = K_1 * M_{areaHw} - K_2 * M_{areaSw} + K_3 * M_{comHwSw} - K_4 * M_{proc} + K_5 * M_{vars} \tag{3.26}$$

O termo M_{areaHw} representa o espaço ocupado no componente de *hardware* (em número de células lógicas), M_{areaSw} mede o espaço que o código e os dados ocupam no componente de *software* e o termo $M_{comHwSw}$ contabiliza a comunicação entre *hardware* e *software*, efectuada através dum barramento. A taxa de utilização do processador é quantificada em M_{proc} e o tamanho das variáveis trocadas entre *hardware* e *software* é representado por M_{vars}. Cada termo da função de custo é afectado por uma constante (K_1 a K_5). A função anterior não considera qualquer tipo de requisito de desempenho, deixando para o algoritmo de partição a tarefa de rejeitar as alternativas de partição que não atinjam os requisitos de desempenho.

Selecção da Função de Custo

A função de custo utilizada na abordagem de Peng e Eles [EPD94, EKP98a] aplica-se apenas à partição em dois componentes, o que não é o caso do presente trabalho, e assemelha-se às funções de proximidade que se aplicam no processo construtivo de partição. Outra limitação

desta função, considerando os objectivos do presente trabalho, está relacionada com a não quantificação da exequibilidade das alternativas de partição analisadas. Por seu lado, a função de custo da abordagem [DH94] está demasiado orientada para sistemas tempo real, tendo como principal objectivo garantir que as tarefas são executadas dentro dos limites definidos pelos requisitos de desempenho impostos ao sistema. Embora não tenham sido mencionadas métricas representativas do espaço ou do número de ligações, isso não impede que elas estejam incluídas no vector de métricas $\overrightarrow{M}$.

A abordagem Cosyma [HE98] aplica uma função de custo que procura maximizar o desempenho e minimizar o espaço ocupado em *hardware*, ou seja, os objectivos traduzidos nesta função são diferentes dos do presente trabalho, onde se pretende atingir um determinado desempenho com os recursos de *hardware* disponíveis. A abordagem Vulcan [GM94][GM96] também recorre a uma função de custo que visa minimizar o espaço ocupado em *hardware*, através da utilização máxima dos recursos de *software*, em vez de tentar rentabilizar os recursos de *hardware* disponíveis. Embora a função utilize métricas que afectam o desempenho, intensidade de comunicação e variáveis transferidas, a sua contabilização é pouco rigorosa, uma vez que o desempenho do sistema é muito mais do que uma simples soma de duas métricas relacionadas com a comunicação entre partições. A função possui a mesma limitação das anteriores, ao não verificar a exequibilidade das alternativas de partição, quer seja em termos de requisitos quer seja em termos de condicionalismos.

A função de custo utilizada na abordagem SpecSyn [VG92][VGG94] é aquela que se aproxima mais da pretendida para a presente abordagem ao problema de partição: uma função que considera como óptima uma alternativa de partição que respeita os condicionalismos impostos pela arquitectura alvo (espaço, ligações e número de componentes) e atinge o desempenho exigido. A função de custo da abordagem SpecSyn, mediante um conjunto de alterações, esteve na origem da função desenvolvida no presente trabalho para orientar o processo iterativo de partição. Esta função é abordada em pormenor na secção 6.7.

3.10 Ambientes de Desenvolvimento com Suporte à Partição

Complementa-se a revisão do estado da arte sobre partição, apresentando um conjunto representativo de ambientes de desenvolvimento que apoiam o projectista na partição dos sistemas em componentes diferenciados. Cada ambiente tem associada uma abordagem ao problema de co-projecto de *hardware* e de *software*. Para cada abordagem, procurou enunciar-se o estilo de modelação empregue, o tipo de arquitectura alvo usada na implementação e o domínio de aplicação (tabela 3.3), o tema central em investigação e o suporte para partição (tabela 3.4).

Ambiente	***Referências***	***Arquitectura alvo***	***Domínio de aplicação***	***Linguagem de especificação***	***Modelação***
Ptolemy	[KL93] [KL94]	Multi-DSP	Dados	C++	OO heterogénea
Castle	[CW96] [WC97]	1 μP	Dados, Controlo	C++, VHDL ou Verilog	CDFG
Cosyma	[EHB93] [HE98]	1 μP 1 ASIC	Dados, Controlo	C^x	CDFG estendido
Lycos	[KM96b] [MGK+97]	1 μP 1 ASIC	Dados, Controlo	C ou VHDL	CDFG
Mickey	[MQB95] [MRB96b]	1 μP Multi-ASIC	Controlo	StateCharts	CDFG
Tosca	[BFS95] [FSV97] [BFS98]	1 μP, RAM, Múltiplos componentes de hw	Controlo	Occam2	Álgebra de processos
Vulcan	[GM94] [GM96]	1 μP Múltiplos componentes de hw	Dados, Controlo	HardwareC	DFG
Chinook	[BCO95] [COB95]	Multi-μP Múltiplos componentes de hw	Controlo	Verilog	Comportamento modelado com o meta-modelo de StateCharts
Cosmos	[JO94] [IJ95]	Multi-μP Múltiplos componentes de hw	Dados, Controlo	SDL	Formato *Solar*
CoWare	[RVBM96]	Multi-μP Múltiplos componentes de hw	Dados, Controlo	C, VHDL ou DFL	Processos comunicantes
COOL	[Nie98]	Multi-μP Múltiplos componentes de hw	Dados	VHDL	CDFG
Polis	[CEG+96]	Multi-DSP ou Multi-μP Múltiplos componentes de hw	Controlo	Esterel	CFSM
SpecSyn	[GV95] [GVNG96]	Multi-μP Múltiplos componentes de hw	Dados, Controlo	SpecCharts	Grafo de acesso SLIF

Tabela 3.3: Caracterização dos ambientes de desenvolvimento.

Ptolemy

Ptolemy é uma abordagem orientada ao objecto para especificar, simular, criar protótipos e efectuar a síntese de *software* para implementações com uma arquitectura baseada em DSPs [KL93] [KL94]. A modelação de sistemas complexos é heterogénea, dado que se utilizam vários meta-modelos computacionais, designados por domínios, para representar o fluxo de dados síncrono, o fluxo de dados dinâmico e os eventos discretos. Os modelos são concretizados em C++.

Castle

Na abordagem Castle, a implementação dum sistema é conseguida à custa da síntese dum processador e do *software* que nele será executado [CW96] [WC97]. O desenvolvimento começa com uma descrição única em C++, Verilog ou VHDL, que depois é convertida para

uma representação com o meta-modelo CDFG. Os resultado obtidos por análises estática e dinâmica do comportamento do sistema, permitem ao projectista definir a estrutura do processador. Recorrendo a uma biblioteca de componentes descritos em VHDL[17], sintetiza-se o processador e o respectivo compilador.

Ambiente	***Partição***	***Função de custo e métricas***	***Ênfase da investigação***
Cosyma	◇ Automática ◇ Algoritmo SA	$F = K_1 * f_1(M_1) + K_2 * f_2(M_1) * M_2$ M_1 - tempo de execução (desempenho) M_2 - espaço em *hardware* $f_1(M_1)$ - sucesso a atingir o desempenho	Co-síntese
Lycos	◇ Automática ◇ Algoritmo PACE	$F = f(M_1, M_2, M_3, M_4)$ M_1 - tempo execução em hw M_2 - tempo execução em sw M_3 - espaço em *hardware* M_4 - tempo de comunicação hw/sw	Partição e comunicação entre hw/sw
Mickey	◇ Automática ◇ Técnica CLP (‡) e ferramenta Minnie	–	Co-projecto de sistemas reactivos
Tosca	◇ Automática ◇ Algoritmos P.Tabu, SA, E.Genética e Agrup. Hierárquico	$F = \sum_{i=1}^{6} K_i * M_i + \overrightarrow{K_7} \times \overrightarrow{M_7}$ M_1 (M_2) - espaço em hw (sw) M_3 (M_4) - tempo de execução hw (sw) M_5 (M_6) - consumo do hw (sw) $\overrightarrow{M_7}$ - métricas obtidas estaticamente	Co-projecto de sistemas reactivos
Vulcan	◇ Automática ◇ Algoritmo do tipo *greedy* estendido	$F = \sum_{i=1}^{3} K_i * M_i - \sum_{i=4}^{5} K_i * M_i$ M_1 - espaço em *hardware* M_2 - comunicação hw/sw M_3 - tamanho das variáveis trocadas M_4 - espaço em *software* M_5 - taxa de utilização do μP	Co-síntese
COOL	◇ Automática ◇ Algoritmos E.Genética e PLI	$F = \sum_{i=1}^{5} K_i * M_i \;\; [K_1 \gg K_2..K_5]$ M_1 - espaço em hw M_2 - espaço em sw para dados M_3 - espaço em sw para código M_4 - tempo execução em hw M_5 - tempo execução em sw	Partição
SpecSyn	◇ Automática ◇ Algoritmos SA, KL, *greedy*, ...	$F = \sum_{i=1}^{3} K_i * f_i(M_i) + K_4 * f_4(M_4)$ $f_i(M_i) = \sum_n [MAX(M_i^n - c_i^n, 0)]^2$ $f_4(M_4) = [MAX(M_4 - c_4, 0)]^2$ M_1 - espaço por componente M_2 - pinos por componente M_3 - tempo de execução (sw , hw) M_4 - número de componentes	Partição

(‡) *Consistent Labelling Problem.*

Tabela 3.4: Caracterização adicional dos ambientes de desenvolvimento que suportam a partição.

[17] Somadores, registos ou comparadores são exemplos de componentes definidos na biblioteca.

Cosyma

Utilizando uma arquitectura alvo com um processador e um coprocessador (ASIC), a abordagem Cosyma procura obter o máximo desempenho à custa do componente de *hardware* [EHB93]. Os sistemas são descritos na linguagem C^x, uma extensão da linguagem C com suporte para especificar a concorrência e o comportamento temporal. As descrições em C^x são convertidas para um grafo de fluxo com sintaxe estendida, que facilita o processo de partição *hardware/software*. A partição é efectuada de forma automática com o algoritmo de *simulated-annealing* e as implementações obtidas não exploram a concorrência entre *hardware* e *software*.

Lycos

A abordagem Lycos utiliza uma arquitectura alvo com um processador e um componente de *hardware* [KM96b] [MGK+97]. O principal objectivo da abordagem é explorar o espaço de projecto através da partição *hardware/software*. Partindo da descrição do comportamento do sistema em C ou VHDL, gera-se uma representação com o meta-modelo CDFG. Antes da partição do sistema, determina-se o número de execuções de cada objecto da descrição e o número de acessos a cada variável, de modo a detectar as zonas críticas em termos de tempo de execução e seleccionar os componentes a incluir na arquitectura alvo (de entre um conjunto de ASICs, processadores e barramentos).

No processo de partição aplica-se o algoritmo PACE, um algoritmo baseado em programação dinâmica, que realça a comunicação entre os objectos do sistema e recorre a estimativas do espaço ocupado pelos recursos e do desempenho. Os objectos manipulados pela partição (BBEs) possuem uma granulosidade intermédia, embora o seu tamanho seja flexível [MH95] [KM96a]. A estimação do tempo de execução em *software* utiliza um modelo genérico, como o que se descreve na secção 4.3.1. Para estimar o tempo de execução em *hardware*, escalonam-se as operações de cada BBE através dum algoritmo baseado em listas ligadas. A estimação do espaço ocupado em *hardware* contabiliza o espaço ocupado pela unidade de controlo e o espaço ocupado pelo caminho de dados. Por fim, a estimação de métricas associadas à comunicação recorre a uma biblioteca para comunicação e um modelo válido em ligações ponto a ponto, composto por *software*, *driver* de *software*, canal de comunicação, *driver* de *hardware* e *hardware*.

Mickey

Mickey é um ambiente para desenvolver sistemas reactivos, que serão implementados com uma arquitectura alvo contendo um processador e vários componentes de *hardware* [MRB96b]. Ini-

cialmente os sistemas são descritos numa linguagem gráfica, cujo meta-modelo é uma extensão ao meta-modelo da linguagem StateCharts. Esta descrição é posteriormente refinada, resultando num grafo CDFG. Os nodos do grafo representam operações básicas para as quais existe uma implementação de *hardware* e de *software* numa biblioteca. Aplicando a técnica *Consistent Labelling Problem* [MQB95] decompõe-se o grafo em componentes de *hardware* e de *software*. Se esta técnica de alto nível não gerar uma solução de partição exequível, recorre-se a técnicas de partição de mais baixo nível incluídas na ferramenta Minnie [MRB96a].

Tosca

Na abordagem Tosca os sistemas são implementados numa arquitectura alvo com um *core* de processador, memória e vários componentes de *hardware* interligados por linhas dedicadas [BFS95] [BFS96a]. A abordagem está orientada para sistemas reactivos tempo real e predominantemente de controlo. O comportamento dum sistema é definido através duma descrição em Occam2, que utiliza a álgebra de processos como meta-modelo. Esta descrição é convertida num grafo em que os nodos representam os processos da descrição Occam2. Sobre o grafo aplica-se um algoritmo de partição que efectua um conjunto de transformações formalizadas com a álgebra de processos. O processo de partição guia-se por estimativas de métricas obtidas estaticamente (número de declarações, número de dependências de dados entre declarações, número de utilizações dos processos, caminho médio/máximo do grafo, número de caminhos completos do grafo), métricas obtidas dinamicamente (tempo de execução em *hardware* e *software*, espaço ocupado em *hardware* e *software*) e métricas obtidas com a síntese da implementação [BFS96b]. O tempo de execução é obtido a partir de métricas mais elementares, como a latência da execução dos processos, a frequência de execução dos processos, a taxa de utilização do processador ou a taxa de utilização dos canais de comunicação. Para o espaço ocupado em *hardware* contribuem o espaço ocupado pela interface, pelo caminho de dados e pela unidade de controlo. O espaço ocupado em *software* engloba a memória ocupada pelo código e pelos dados do sistema, mais a memória ocupada pelo sistema operativo [BFS98]. Pode ainda usar-se métricas que medem o consumo dos componentes de *hardware* e do processador. A abordagem TOSCA disponibiliza o algoritmo construtivo de agrupamento hierárquico e os algoritmos iterativos de pesquisa tabu, *simulated annealing* e evolução genética [FSV97].

Vulcan

A abordagem Vulcan também se enquadra no grupo das abordagens em que a arquitectura alvo contém um processador e vários componentes de *hardware* [GM92] [GM94] [GM96]. A principal tarefa da metodologia de desenvolvimento é a co-síntese. Para especificar um sistema,

descreve-se o seu comportamento na linguagem HardwareC e definem-se os seus requisitos não funcionais e os condicionalismos impostos pela arquitectura alvo. A descrição inicial do comportamento do sistema é convertida para uma descrição cujo meta-modelo é um grafo de fluxo, que se adequa à estimação do desempenho. Partindo duma solução totalmente em *software*, o processo iterativo de partição procura minimizar o espaço ocupado em *hardware*, mas garantindo que o desempenho exigido ao sistema é atingido. No caso concreto duma implementação com um componente de *hardware* e outro de *software*, o algoritmo iterativo de partição e a função de custo que controla a sua evolução foram apresentados neste capítulo.

Chinook

Chinook é uma abordagem centrada na co-síntese de sistemas embebidos tempo real com predominância para controlo, e em especial na síntese da interface entre componentes e nos métodos de comunicação [BCO95] [COB95]. Os sistemas são implementados numa arquitectura com vários processadores e vários componentes de *hardware*. Os sistemas são especificados numa descrição única com a linguagem Verilog. A descrição dum sistema define o seu comportamento, os requisitos não funcionais (desempenho) e a definição dos módulos que farão parte da sua implementação (estrutura). Na modelação do comportamento dos sistemas aplica-se um meta-modelo análogo ao da linguagem StateCharts. Os módulos que na descrição do sistema definem a sua estrutura são seleccionados a partir duma biblioteca de componentes de *hardware* e de processadores. A partição é manual, mas as tarefas de escalonamento e de síntese da comunicação são efectuadas de forma automática. A abordagem permite simular a descrição inicial e a descrição sintetizável.

Cosmos

Cosmos é um ambiente de desenvolvimento que apoia o projectista no refinamento duma descrição de muito alto nível do sistema, de modo a gerar uma descrição em C ou VHDL para cada um dos componentes de *hardware* ou de *software* a usar na sua implementação [IJ95]. A implementação do sistema pode envolver vários processadores e vários componentes de *hardware*. Partindo duma descrição inicial na linguagem SDL, gera-se uma representação intermédia no formato *Solar* [JO94]. A síntese opera sobre esta representação, decompondo-a em componentes que comunicam por canais definidos de forma abstracta. Para concretizar o protocolo de cada canal utiliza-se uma biblioteca de protocolos. A partição automática em componentes, a síntese da comunicação e a síntese da descrição dos componentes a usar na implementação não são suportadas pelo ambiente Cosmos. Como a partição é manual, é necessário que o projectista repita todo o processo, começando na decomposição em componentes, passando pela geração dos mecanismos de comunicação e terminando na geração de

código C ou VHDL (a partir de descrições no formato Solar), até que a solução de partição satisfaça os requisitos e condicionalismos impostos ao sistema.

CoWare

CoWare é um ambiente de desenvolvimento, com suporte para a modelação, refinamento e implementação de sistemas [RVBM96]. A modelação é heterogénea, dado que um sistema é modelado por um conjunto de processos comunicantes, que podem ser descritos em linguagens de alto nível como C, VHDL ou DFL. A arquitectura de implementação também é heterogénea, na medida em que é composta por vários *cores* de processador e por vários componentes de *hardware*. A semântica da comunicação entre processos inspira-se no modelo *remote procedure call*. Após simular todo o sistema, (i) efectua-se a sua partição manual, agrupando os processos a atribuir a cada processador da arquitectura alvo, (ii) selecciona-se o esquema de comunicação entre componentes da implementação, (iii) refina-se a comunicação com detalhes de implementação e (iv) gera-se o código que descreve a funcionalidade atribuída a cada componente da implementação (processador ou ASIC).

COOL

COOL é um ambiente integrado orientado para a modelação de sistemas embebidos, predominantemente de fluxo de dados e sujeitos a fortes requisitos de tempo real [Nie98]. A funcionalidade dos sistemas é modelada de forma heterogénea, através dum sub-conjunto da linguagem VHDL. A modelação dum sistema inclui também os requisitos e condicionalismos de projecto e a arquitectura do sistema, representados graficamente. O ambiente COOL permite que os sistemas em projecto possuam uma arquitectura heterogénea, composta por diversos processadores, componentes de *hardware* e canais de comunicação. A abordagem subjacente ao ambiente COOL está focalizada (i) na partição *hardware/software* automática, empregando vários algoritmos de partição e explorando o espaço de projecto e (ii) na co-síntese automática das implementações, especialmente das interfaces. O processo de partição recorre a estimativas de métricas de *hardware*, tais como o tempo de execução e o espaço ocupado pelos recursos, e a estimativas de métricas de *software*, como por exemplo o tempo de execução. As métricas de *hardware* são estimadas com a ferramenta que se usa na síntese dos componentes de *hardware* e o tempo de execução em *software* é estimado pelo compilador utilizado para gerar os componentes de *software* da implementação. O ambiente permite ainda simular o sistema.

Polis

Polis é um conjunto de ferramentas para apoiar o co-projecto de sistemas reactivos tempo real, predominantemente de fluxo de controlo [CEG+96]. São permitidas implementações em arquitecturas multi-processador e com vários componentes de *hardware*, tais como ASICS ou DSPs. Os sistemas são modelados por uma rede de CFSMs, proveniente duma descrição inicial na linguagem Esterel. Embora a partição em componentes seja manual, é assistida por estimativas calculadas automaticamente. Polis permite executar de forma automática a síntese das CFSMs para *hardware* ou *software*, a síntese da interface entre componentes e a geração de um núcleo de sistema operativo tempo real, que coordena o funcionamento dos diversos componentes da implementação.

SpecSyn

SpecSyn é um ambiente que apoia o projectista no desenvolvimento de sistemas embebidos [GVNG94] [GV95] [GVNG96]. Os sistemas são descritos na linguagem SpecCharts, uma linguagem que combina os StateCharts com VHDL e tem subjacente o meta-modelo PSM. Estas descrições são depois convertidas para uma representação interna que emprega o grafo SLIF como meta-modelo. As principais tarefas executadas automaticamente são a selecção de recursos, a partição do sistema em componentes de *hardware* e de *software* e o refinamento das descrições resultantes do processo de partição. A partição testa diferentes algoritmos, entre os quais se incluem o algoritmo *greedy*, *simulated annealing* ou Kernighan/Lin [VL96]. Exemplos de detalhes, adicionados à descrição dos sistemas durante a fase de refinamento, têm a ver com a ocorrência de partilha de recursos e com o protocolo de comunicação seleccionado para a comunicação entre componentes. A arquitectura alvo pode incluir múltiplos processadores e componentes de *hardware*, memória e barramentos.

A Abordagem Proposta

As figuras 3.11 e 3.12 enquadram a presente abordagem ao problema de partição no conjunto das outras abordagens apresentadas. A figura 3.11 resume as potencialidades das diferentes abordagens para modelar sistemas, identificando o estilo de modelação, o tipo de validação e o domínio de aplicação. Por seu lado, a figura 3.12 sumaria o apoio que as diferentes abordagens prestam à implementação dos sistemas com componentes diferenciados. Este apoio é traduzido em três aspectos: o grau de automação em que se processa a partição, o suporte dado à síntese da interface entre componentes e o tipo de arquitectura alvo permitido. Dado que as abordagens Ptolemy e Castle não implementam os sistemas com componentes diferenciados, não necessitam da tarefa de partição em componentes diferenciados. Por este

motivo, as duas abordagens foram excluídas da figura 3.12.

A abordagem proposta, concretizada na ferramenta *parTiTool*[18], inclui-se numa metodologia de desenvolvimento que utiliza modelação heterogénea, é aplicável a sistemas mistos de dados e controlo mas está orientada para sistemas predominantemente de dados, nesta fase de evolução ainda não suporta co-verificação dos sistemas, a arquitectura alvo inclui um processador e múltiplos componentes de *hardware*, a partição é automática e como a estimação de métricas modela de forma precisa a comunicação, pode obter-se daqui informação sobre a interface entre componentes.

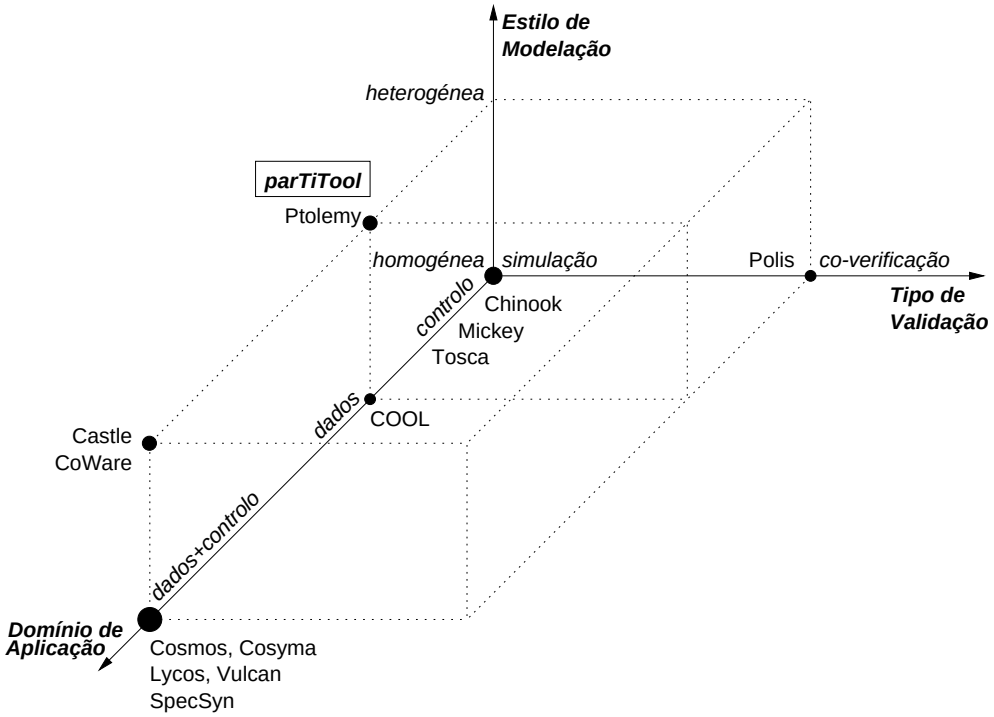


Figura 3.11: Caracterização das abordagens em função da modelação de sistemas.

3.11 Resumo e Conclusões

A presente abordagem ao problema de partição é do tipo funcional, inter-componentes e automática, porque é esta a estratégia mais adequada para obter soluções de partição com maior qualidade, aplicar a sistemas complexos e obter soluções com qualquer número de componentes.

A análise de vários algoritmos de partição construtivos revelou que a aplicação dum algoritmo exaustivo não é viável porque isso exigiria um tempo de cálculo inaceitável, os algoritmos de crescimento de grupos e de agrupamento hierárquico constroem as soluções de partição de forma distinta mas produzem praticamente os mesmos resultados, os métodos baseados em

[18] O nome ***parTiTool*** derivou da expressão *"**parTi**tioning **Tool**"*.

PLI geram soluções quase-óptimas e dispensam a aplicação dum algoritmo de optimização iterativo mas exigem um tempo de cálculo elevado e não são fáceis de formular, e os algoritmos PACE e GCLP, por serem demasiado específicos, não se revelaram atractivos para ser adaptados ao presente trabalho. Ao aplicar no processo de partição um algoritmo construtivo em conjunto com outro iterativo, não se exige que o algoritmo construtivo gere soluções com qualidade tão elevada. Deste modo, optou-se por um **algoritmo construtivo** de fácil implementação, o algoritmo de crescimento de grupos, que embora disponha duma heurística de optimização simples, através da função de proximidade consegue melhorar-se a sua capacidade para gerar soluções de qualidade.

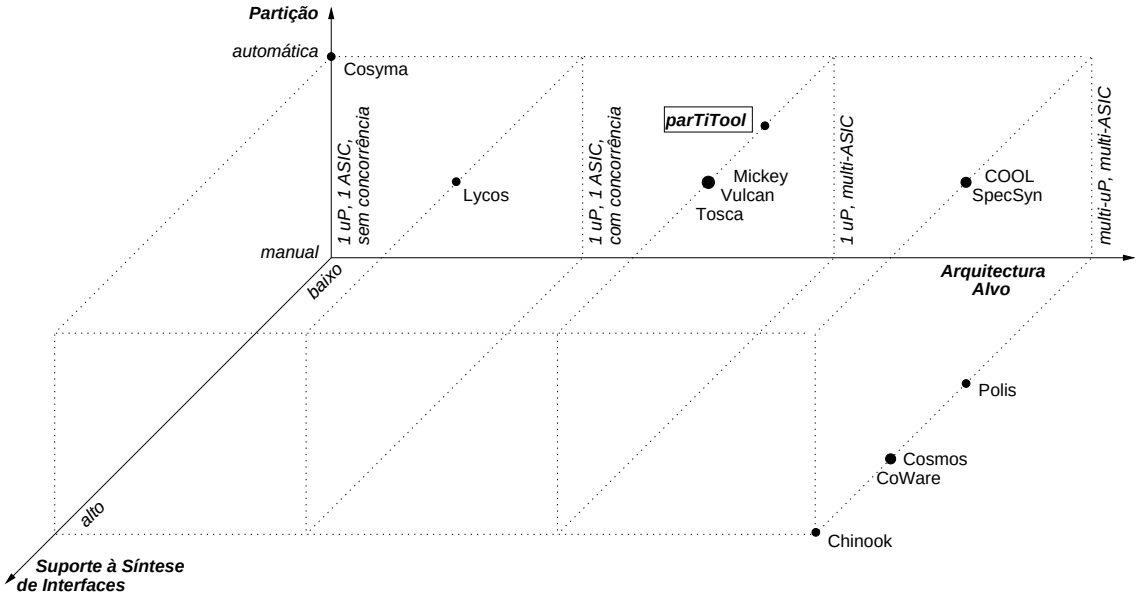


Figura 3.12: Caracterização das abordagens considerando o suporte para a implementação de sistemas.

A avaliação dum conjunto significativo de algoritmos de partição iterativos mostrou que o algoritmo de Kernighan/Lin possui uma capacidade limitada para evitar mínimos locais da função de custo, o algoritmo de *simulated annealing* apresenta maiores potencialidades para atingir soluções óptimas do que os algoritmos do tipo *greedy* e de Kernighan/Lin, mas o tempo de cálculo é muito elevado, e o algoritmo de pesquisa tabu consegue reduzir o tempo de cálculo restringindo a pesquisa das soluções de partição apenas a uma vizinhança dessas soluções. A pesquisa na vizinhança é ainda limitada pela existência de deslocamentos que estão proibidos. Os algoritmos de evolução genética distinguem-se dos outros algoritmos iterativos apresentados porque guardam um conjunto de alternativas de partição por iteração, enquanto os restantes guardam apenas uma. Esta estratégia de optimização introduz nos algoritmos de evolução genética uma maior eficácia na redução do espaço de projecto, mas também uma menor capacidade de convergência para a solução de partição óptima.

Estabelecendo como critério prioritário de selecção dum **algoritmo iterativo** a sua capacidade para encontrar soluções com mais qualidade, as opções de escolha são os algoritmos de *simulated annealing* e de pesquisa tabu. De acordo com o segundo critério de selecção, o tempo de cálculo, o algoritmo de pesquisa tabu é a opção acertada. Em terceiro lugar, a facilidade de implementação e de utilização, as vantagens dividem-se: o algoritmo de *simulated annealing* é mais fácil de adaptar a qualquer problema de partição, mas o algoritmo de pesquisa tabu é mais fácil de parametrizar. Como existe equilíbrio na qualidade das soluções obtidas e nos aspectos de implementação/utilização, mas o algoritmo de pesquisa tabu exige um tempo de cálculo significativamente inferior, o algoritmo iterativo de pesquisa tabu foi a escolha da presente abordagem.

A presente abordagem de partição implementou uma **função de proximidade** que se ajusta ao tipo de objecto a atribuir: variável ou estado programa. Para escolher a partição à qual se atribuem as variáveis, a função utiliza as métricas intensidade de comunicação e adequação para uma implementação em *hardware*. No caso dos estados programa, a função recorre às métricas intensidade de comunicação, tempo de computação em *software* e em *hardware* e espaço ocupado em *hardware*. Quando se trata de atribuir estados programa, a função selecciona de forma prioritária a partição que conduz ao melhor desempenho e em segundo lugar a partição que esteja menos preenchida.

As abordagens ao problema de partição analisadas aplicam uma função de custo com o objectivo de (i) agrupar os objectos mais próximos, porque possuem mais características em comum e/ou apresentam uma troca de informação superior, (ii) reduzir os recursos de *hardware* utilizados ou (iii) reduzir a comunicação entre componentes. A **função de custo** seleccionada neste trabalho tem na sua génese a função de custo da abordagem SpecSyn, que considera como óptima uma alternativa de partição que respeita os condicionalismos impostos pela arquitectura alvo e atinge o desempenho exigido.

Relativamente às potencialidades para a modelação de sistemas, constatou-se que nenhuma das abordagens que utiliza modelação heterogénea permite co-verificação, o que é compreensível dado que a verificação formal usa como estratégia principal a detecção de violações das propriedades dum determinado meta-modelo. Esta tarefa torna-se complicada quando a modelação dum sistema se faz com vários meta-modelos em vez de apenas um. Concluiu-se ainda que a maioria das abordagens adopta o estilo de modelação homogéneo e dispõe apenas da simulação como método para validar os sistemas. A descrição dos sistemas faz-se com uma linguagem orientada para *software* (C, uma variante do C, Occam ou C++) ou uma linguagem orientada para *hardware* (VHDL, Verilog ou HardwareC). Uma boa parte das abordagens não efectua a partição dos sistemas automaticamente, como acontece nas abordagens Chinook, Cosmos, CoWare e Polis. Nenhuma das abordagens suporta totalmente a partição e a sín-

tese da interface entre os componentes a usar na implementação. No caso das abordagens que dispõem de partição automática, as métricas mais recorrentes na função de custo são o desempenho, a quantidade de recursos utilizada e o custo da comunicação entre *hardware* e *software*. Quanto aos algoritmos de partição, estes vão desde algoritmos específicos até algoritmos de optimização genéricos como *simulated annealing*, Kernighan/Lin, evolução genética ou pesquisa tabu.

Capítulo 4

Estimação de Métricas

Sumário

Antes de detalhar o problema de estimação de métricas introduzem-se os conceitos de precisão, fidelidade, classificação de métricas e modelos para estimação. O problema de estimação é depois aflorado através duma abordagem suficientemente genérica, onde se descreve a estimação dum conjunto alargado de métricas de hardware e de software, sem considerar em profundidade alguns aspectos de optimização. A apresentação foi organizada em estimação de métricas de hardware e estimação de métricas de software. Ambos os tipos de métricas, hardware ou software, foram ainda divididos em métricas de desempenho e de custo. No caso das métricas de hardware, mostra-se como se estimam as métricas de desempenho associadas à computação (como a frequência de relógio, o número de etapas de controlo ou o tempo de execução), as métricas de desempenho associadas à comunicação (como o tempo de comunicação) e as métricas de custo (como o espaço ocupado pelo caminho de dados e pela unidade de controlo dos componentes ou o número de pinos necessário à comunicação entre componentes). Relativamente às métricas de software, indica-se como se estimam as métricas de desempenho (como o tempo de execução) e as métricas de custo (como o espaço ocupado em memória pelo código e pelos dados). Para aprofundar os problemas relacionados com a estimação das métricas mais importantes de hardware, software e associadas à comunicação, faz-se a síntese dum conjunto significativo de abordagens referenciadas na bibliografia.

Conteúdo

4.1 Introdução

O capítulo 3 mostrou como as funções de proximidade e de custo, utilizadas durante a partição, aplicam métricas relacionadas com os requisitos e os condicionalismos impostos ao sistema

em desenvolvimento. Para verificar se estas limitações são ou não respeitadas, é preciso determinar os valores das métricas que lhe correspondem. Para se poder analisar um conjunto alargado de alternativas de partição, é desejável que essas métricas sejam rapidamente calculadas. Deste modo, embora a implementação seja um método que fornece valores exactos para as métricas, é preterida pelo método de estimação, que disponibiliza valores aproximados mas que são calculados em muito menos tempo.

Para obter estimativas das métricas recorre-se a **modelos de projecto**, concretamente modelos para estimação de métricas de *hardware*, de *software* e de comunicação. Em principio, quanto mais detalhado for um modelo mais precisas e demoradas serão as estimativas por ele obtidas. A **precisão** duma estimativa representa a proximidade entre o valor estimado para uma métrica e o valor medido após a implementação do sistema. Como no processo de partição as estimativas das métricas servem para comparar as alternativas de partição, podem usar-se estimativas menos precisas e mais rápidas, desde que apresentem um grau de fidelidade elevado. A **fidelidade** dum método de estimação é a percentagem de situações, definidas para cada par de alternativas de partição P_i e P_j, em que a diferença entre o par de estimativas M_i^E e M_j^E apresenta a mesma tendência da diferença entre o par de valores obtidos com as respectivas implementações (M_i^I e M_j^I). Dadas duas alternativas de partição P_i e P_j, diz-se que a diferença entre a estimativa duma métrica para as duas alternativas de partição ($\delta M_{ij}^E = M_i^E - M_j^E$) segue a mesma tendência da diferença entre o valor obtido com a implementação das duas alternativas de partição ($\delta M_{ij}^I = M_i^I - M_j^I$), se $\delta M_{ij}^E > 0$ quando $\delta M_{ij}^I > 0$, ou se $\delta M_{ij}^E < 0$ quando $\delta M_{ij}^I < 0$ ou então se $\delta M_{ij}^E = 0$ quando $\delta M_{ij}^I = 0$.

O processo de partição emprega normalmente métricas de custo[1] e de desempenho, relativas a implementações em *hardware* e em *software*. Embora em menor escala, também se aplicam métricas como a capacidade de dissipação de calor, o consumo, a facilidade de teste, o tempo de desenvolvimento, o tempo que o produto demora a chegar ao mercado[2] e o custo de produção.

As **métricas de custo** do *hardware* mais relevantes para o processo de partição são o espaço ocupado pelos recursos de *hardware* e os pinos necessários à implementação. No caso de implementações em FPGAs, o espaço ocupado pode ser expresso através duma métrica como o número de blocos de lógica configurável (CLBs) seleccionados. Por seu lado, as métricas de custo do *software* mais recorrentes são o espaço ocupado em memória pelo código executável e o espaço ocupado em memória pelos dados manipulados pelo sistema em funcionamento.

As **métricas de desempenho** podem ser classificadas em métricas associadas à computação e métricas associadas à comunicação. As métricas de computação estão relacionadas com o tempo despendido nos cálculos impostos pela funcionalidade do sistema, enquanto as métricas

[1] Com as *métricas de custo* não pretende quantificar-se o preço das soluções, mas sim quantificar características que têm influência sobre a dimensão ou a dificuldade de implementação das soluções.

[2] Também designado por *time to market*.

de comunicação estão relacionadas com o tempo despendido na troca de informação entre módulos do sistema. Entre as métricas de computação mais utilizadas encontram-se a frequência de relógio, as etapas de controlo[3] e o tempo de execução. A selecção da frequência de relógio é importante porque influencia o desempenho e o custo da implementação dos sistemas. Quando a parte de *hardware* dum sistema é implementada com uma arquitectura constituída por uma unidade de controlo e um caminho de dados, uma etapa de controlo representa um estado da máquina de estados correspondente à unidade de controlo.

A estimação do **tempo de execução em *hardware*** é importante porque se o tempo de execução coincidir com um requisito de desempenho imposto ao sistema, o seu valor influencia a selecção de componentes a usar na implementação. Por exemplo, se o tempo de execução for superior ao limite permitido será necessário seleccionar mais componentes ou componentes mais rápidos, aumentando assim o custo da implementação. Para atenuar o aumento do custo duma implementação, deve melhorar-se a taxa de utilização dos componentes seleccionados, através da partilha de componentes por parte de várias operações e/ou da selecção de componentes que possuam uma arquitectura com vários níveis de *pipeline*.

Para estimar o **tempo de execução** dum programa **em *software*** é preciso conhecer o número de execuções de instruções, o número médio de ciclos de relógio por instrução (CPI) e a frequência de relógio do processador (equação 4.1). A frequência de relógio é normalmente conhecida, mas para obter o número de execuções de instruções é preciso recorrer a um dos seguintes métodos: quando o código é perfeitamente determinístico, usa-se uma análise estática (*profiling*) para obter o valor exacto, caso contrário recorre-se a uma análise dinâmica (execução) para determinar um valor médio. O número médio de ciclos de relógio por instrução pode ser substituído por um valor médio para uma dada implementação de processador e para um tipo de sistema. O valor do CPI depende dos níveis de *pipeline*, do número de instruções que podem ser executadas em paralelo, da arquitectura da hierarquia de memória e dos acessos à memória.

$$TexecSW = N_{execucoes} * CPI * \frac{1}{F_{relogioCPU}} \tag{4.1}$$

Uma das métricas de comunicação mais utilizadas é a **taxa de transferência**, definida como a quantidade de informação por unidade de tempo, que é trocada entre dois módulos (da funcionalidade do sistema) através dum canal de comunicação. A estimação da taxa de transferência justifica-se porque (i) permite saber se os barramentos disponíveis na arquitectura alvo suportam a comunicação entre módulos atribuídos a diferentes componentes ou se são necessários barramentos mais largos, (ii) permite calcular o tempo de execução dos módu-

[3]Métrica válida apenas para o *hardware*.

los envolvidos na comunicação[4] e (iii) influencia a partição dos módulos pelos componentes da arquitectura alvo, favorecendo-se a atribuição ao mesmo componente de módulos que comunicam entre si com uma taxa de transferência elevada e favorecendo-se a atribuição a barramentos diferentes de canais que apresentam uma taxa de utilização elevada.

O consumo e a quantidade de calor a dissipar por um componente variam directamente com a frequência do relógio, com o número de portas lógicas activas em cada ciclo de relógio e com o tempo de funcionamento. As questões do consumo e da dissipação de calor são relevantes em sistemas de dimensões reduzidas, em sistemas a que se exige um baixo consumo e em sistemas com baixa tolerância a falhas.

A facilidade de teste, o tempo de desenvolvimento, o tempo que o produto demora a chegar ao mercado e o custo de produção são métricas a considerar na definição da metodologia de desenvolvimento dos sistemas, na qual a metodologia de partição se insere, mas como afectam pouco as tarefas de partição não é habitual incluí-las na função de custo que controla o processo de partição.

4.2 Estimação de Métricas de *Hardware*

4.2.1 Modelo para Estimação de Métricas de *Hardware*

Os três grandes componentes dum modelo de estimação das métricas de *hardware* são o modelo da arquitectura alvo, a política de selecção de recursos e a política de escalonamento das operações. A arquitectura alvo da implementação pode estar pré-definida ou ser seleccionada pelo projectista. Para ilustrar a aplicação dos modelos de *hardware* na estimação de métricas, considera-se o modelo genérico em que a arquitectura de implementação inclui uma unidade de controlo (UC) e um caminho de dados (CD), apresentado na figura 4.1 [GVNG94]. O caminho de dados inclui registos, pilhas de registos, memória, unidades funcionais e multiplexadores. A unidade de controlo é composta pelo registo de estado, a lógica que gera o próximo estado e a lógica de controlo. Este modelo intervém na estimação de métricas de *hardware* como a frequência de relógio, as etapas de controlo, o tempo de execução, a taxa de transferência, o espaço e os pinos.

4.2.2 Estimação de Métricas de Desempenho Associadas à Computação

Nesta secção introduz-se o problema de estimação do tempo de execução em *hardware*, nas situações em que (i) o código não contém nem ciclos nem ramificações e (ii) quando ambos os tipos de construtores fazem parte do código do sistema. Apresenta-se também a estimação

[4] Uma vez que o tempo de execução é a soma do tempo de computação com o tempo de comunicação.

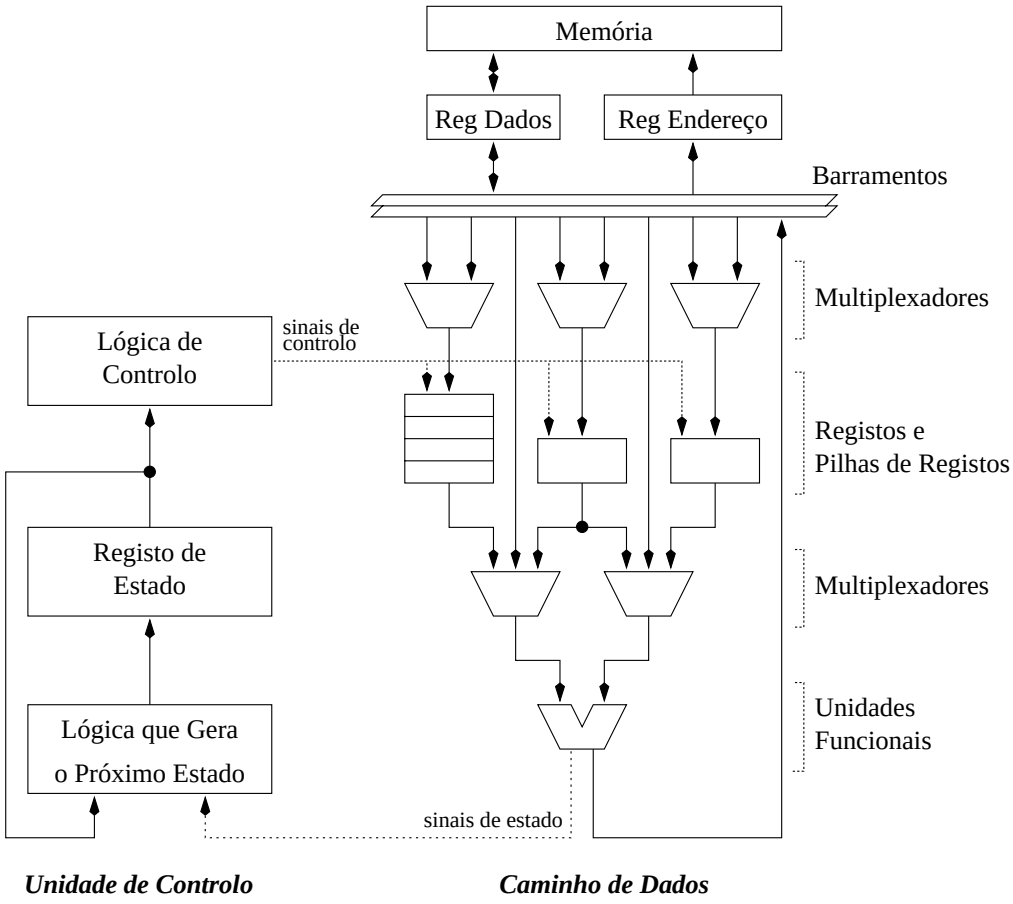


Figura 4.1: Modelo UC/CD para estimação de métricas de *hardware*.

das métricas frequência de relógio e número de etapas de controlo, envolvidos no cálculo do tempo de execução.

Em algumas abordagens a **frequência de relógio** é conhecida, porque é fixa pela arquitectura alvo ou porque é definida noutros módulos em que o sistema está embebido. Quando isto não acontece convém estimar o seu valor, uma vez que ele afecta o desempenho e o espaço ocupado pelo sistema. Apresentam-se a seguir métodos para estimar a frequência de relógio.

A frequência de relógio pode ser definida como o inverso do tempo de execução da unidade funcional mais lenta presente no sistema. Este método é rápido e fácil de implementar, mas origina uma sub-utilização das unidades funcionais mais rápidas e em consequência disto o tempo de execução global aumenta. Para diminuir o tempo de execução global é preciso diminuir o tempo de inactividade das unidades funcionais. Surge assim outro método de estimação da frequência de relógio, segundo o qual se selecciona para valor da frequência aquele que minimizar a inactividade média das diversas unidades funcionais do sistema [NG92b]. Quando se emprega o modelo da figura 4.1 e se deseja que a execução duma operação, no caminho de dados, demore apenas um ciclo da unidade de controlo, então o período do relógio deve ser superior à soma dos atrasos no caminho de dados com os atrasos na unidade de

controlo [NG92a].

Nos casos em que o código que descreve o sistema não inclui construtores condicionais nem ciclos, o número de **etapas de controlo** pode ser calculado pelo método de rentabilização dos recursos ou pelo método de escalonamento. Se o sistema incluir construtores condicionais ou ciclos, faz-se a sua decomposição em blocos básicos de escalonamento (BBEs) e aplica-se um dos métodos mencionados a cada um dos blocos em que se decompôs o sistema.

Um **bloco básico de escalonamento** é uma sequência de instruções, ou declarações numa HLL/HDL, que não inclui ramificações do fluxo de controlo, ou então é uma sequência de uma ou mais instruções em que apenas a última instrução altera o fluxo de controlo. Supondo que o comportamento do sistema está descrito em VHDL, pode aplicar-se as seguintes regras para definir os BBEs: (i) um BBE começa numa declaração, que se designa de ponto de entrada, e termina na declaração anterior ao próximo ponto de entrada da descrição e (ii) os pontos de entrada que derivam da definição de BBE são a primeira declaração da descrição, uma declaração `wait`, a invocação de um procedimento, as declarações para onde é transferido o controlo na ramificação imposta por uma declaração condicional, a declaração que se segue a uma declaração `wait`, a declaração que se segue à invocação dum procedimento ou a declaração que se segue a uma declaração condicional.

Quando existem ramificações do fluxo de controlo há que considerar duas situações para calcular o número total de etapas de controlo: numa das alternativas permite-se que os blocos básicos de escalonamento partilhem etapas de controlo, enquanto na outra alternativa cada bloco é escalonado numa etapa de controlo distinta.

Com base nos recursos disponíveis para implementar o sistema, estando os recursos fixos à partida, o **método de rentabilização dos recursos** procura maximizar a taxa de utilização desses recursos. Para conseguir a melhor utilização de recursos, em cada etapa de controlo usa-se o maior número possível de operadores de cada tipo, dentro do número disponível. Suponha-se que um sistema é constituído pelos objectos O, que as operações distintas formam o conjunto Ω, que os recursos disponíveis incluem $numUF(\omega_i)$ operadores com atraso $ciclos(\omega_i)$ capazes de implementar a operação ω_i e que cada operação ω_i é utilizada $numOP(\omega_i, o_j)$ vezes num objecto o_j. Então, o número de etapas de controlo necessárias ao objecto o_j é aproximado por

$$etapas(o_j) = max_{\omega_i \in \Omega} \left[\left\lceil \frac{numOP(\omega_i, o_j)}{numUF(\omega_i)} \right\rceil * ciclos(\omega_i) \right] \tag{4.2}$$

e o número de etapas de controlo necessárias à totalidade dos objectos O é aproximado por

$$etapasCtl(O) = \sum_{o_j \in O} etapas(o_j) \tag{4.3}$$

O número de etapas de controlo também pode ser estimado através de um dos **métodos de escalonamento**, como por exemplo o escalonamento baseado em listas, em que, dado um conjunto limitado de recursos, se procura minimizar o número de etapas de controlo [GDWL92]. No algoritmo baseado em listas apresentado na figura 4.2, o escalonamento das operações processa-se de uma forma iterativa, com uma iteração a corresponder a uma nova etapa de controlo. A cada tipo de operação ω_i existente no sistema, está associada uma lista $listaOP(\omega_i)$ com as operações que estão prontas a ser escalonadas na etapa de controlo corrente ($etapaCtl$). O facto de uma operação estar numa lista, significa que todas as operações de que depende já foram escalonadas em etapas de controlo anteriores. As operações duma lista são ordenadas segundo a prioridade com que devem ser escalonadas. Assim, em cada etapa de controlo e para cada tipo de operação ω_i, a função $ActualizarEscalonamento$ escalona as operações mais prioritárias de entre as operações que estão prontas, respeitando o máximo de operadores disponíveis para esse tipo de operação ($numUF(\omega_i)$). No fim de cada iteração, as operações escalonadas são retiradas pela função $RemoverPrioritaria$ das respectivas listas e são-lhe adicionadas as operações que ficaram prontas para ser escalonadas na etapa de controlo seguinte (função $ActualizarLPrioridade$).

```
// O é o conjunto dos objectos do sistema, Φ é o escalonamento do sistema,
// numUF(ω_i) é o número de operadores capazes de executar a operação ω_i,
// Ω é o conjunto de operações distintas no sistema

escalonamentoBListas (O, Φ, numUF, Ω) ≡

    CriarLPrioridade(O, listaOP)
    etapaCtl = 0
    enquanto (listaOP(ω_1) ≠ ∅) OU (listaOP(ω_2) ≠ ∅) OU (...) fazer
        etapaCtl = etapaCtl + 1
        para ω_i ∈ Ω fazer
            para j = 1 até numUF(ω_i) fazer
                se (listaOP(ω_i) ≠ ∅) então
                    ActualizarEscalonamento(Φ, Prioritaria(listaOP(ω_i)), etapaCtl)
                    listaOP(ω_i) = RemoverPrioritaria(listaOP(ω_i))
                fse
            fpara
        fpara
        ActualizarLPrioridade(O, listaOP)
    fenquanto
    devolver (etapaCtl)
```

Figura 4.2: Algoritmo de escalonamento baseado em listas para calcular o número de etapas de controlo.

No algoritmo da figura 4.2 a função $CriarLPrioridade$ estabelece uma lista para cada tipo de operação, inserindo nela as operações que não dependem da execução de outras operações, a função $Prioritaria$ devolve a operação mais prioritária duma lista e a função $RemoverPrioritaria$ actualiza uma lista de operações, removendo dessa lista a operação mais prioritária. A estimativa para o número de etapas de controlo, necessário ao escalo-

namento do sistema, coincide com o valor de $etapaCtl$ no final da execução do algoritmo $escalonamentoBListas$.

Quando o código que descreve o comportamento dum sistema não inclui construtores condicionais nem ciclos, a estimativa do **tempo de computação** é proporcional ao número de etapas de controlo em que o comportamento foi escalonado. Se B for o conjunto de BBEs em que se decompôs o comportamento, $etapasCtl(B)$ o número de etapas de controlo em que se escalonou o comportamento e $T_{relogio}$ o período do relógio utilizado na implementação, então o tempo de computação do sistema é simplesmente

$$T_{comp}(B) = etapasCtl(B) * T_{relogio} \tag{4.4}$$

Na situação genérica, em que o código que descreve o comportamento do sistema inclui construtores condicionais e/ou ciclos, para estimar o tempo de computação é preciso efectuar uma análise estatística do fluxo de controlo do sistema, de modo a obter a probabilidade de execução de cada BBE ($Freq(bbe_i)$). A probabilidade de execução dum BBE é definido como o número médio de vezes em que o BBE é executado em cada execução do sistema [GVNG94]. Aplicando o método de rentabilização dos recursos, ou o método de escalonamento, estima-se o número de etapas de controlo para cada BBE, e com a equação 4.4 obtém-se o respectivo tempo de computação ($T_{comp}(bbe_i)$). A estimativa para o tempo de computação do sistema é assim uma soma pesada do tempo de computação dos diversos BBEs em que se decompôs o sistema, ou seja

$$T_{comp}(B) = \sum_{bbe_i \in B} T_{comp}(bbe_i) * Freq(bbe_i) \tag{4.5}$$

Para efectuar a análise estatística do fluxo de controlo do sistema descreve-se o sistema através dum grafo de fluxo de controlo $G = \{N, A\}$, em que o nodo $n_i \in N$ representa o BBE bbe_i e o arco $a_{ji} \in A$ indica a existência de uma dependência de controlo do bloco bbe_i relativamente ao bloco bbe_j. Para estimar a probabilidade de execução duma ramificação, imposta por um ciclo ou construtor condicional, recorre-se a uma de três alternativas:

- a análise estática parte do princípio que o número de iterações nos ciclos é conhecida; se o número de iterações for nc, atribui-se uma probabilidade $\frac{nc}{nc+1}$ ao arco pelo qual o fluxo de controlo regressa ao início do ciclo e uma probabilidade $\frac{1}{nc+1}$ ao arco pelo qual o fluxo de controlo sai do ciclo; às diferentes ramificações dum construtor condicional atribui-se uma probabilidade igual, que no caso de haver nr ramificações é $\frac{1}{nr}$;
- o projectista anota as diversas ramificações do grafo de fluxo G com probabilidades por ele definidas;

- ◇ as ramificações do grafo de fluxo G são anotadas com probabilidades obtidas com uma análise dinâmica, ou seja, a probabilidade média para cada ramificação do grafo resulta de um conjunto de simulações executadas com diferentes dados.

Com base na probabilidade das ramificações do grafo G, deriva-se a probabilidade de execução dos BBEs da seguinte forma:

- ◇ adicionar ao início do grafo G um nodo n_{inic} com frequência de execução igual a 1;
- ◇ formular a frequência de execução de cada nodo n_i de G através dum somatório, em que cada termo do somatório é *o produto da frequência de execução dum nodo n_j (que passa o fluxo de controlo ao nodo n_i[5]) pela probabilidade do arco a_{ji} (que vai de n_j a n_i)*; a formulação da frequência de execução do nodo n_i traduz-se na equação 4.6;

$$Freq(n_i) = \sum_{n_j \in Antecessores(n_i)} Freq(n_j) * Prob(a_{ji}) \tag{4.6}$$

- ◇ resolver o sistema de equações lineares obtidas no passo anterior, por exemplo através do método de eliminação de Gauss, de modo a obter a frequência de execução de cada nodo n_i (ou bloco bbe_i).

4.2.3 Estimação de Métricas de Desempenho Associadas à Comunicação

Para estimar a taxa de transferência envolvida na comunicação entre dois módulos de comportamento, começa por definir-se os conceitos de número de bits trocados através dum canal e de tempo de comunicação, que por sua vez recorrem ao conceito de número de acessos a um canal. O número médio de acessos a um canal, efectuados por um módulo de comportamento, é determinado pelo método da análise estatística do fluxo de controlo. O número de bits recebidos ou enviados por um módulo de comportamento B através dum canal C é definido como o produto do número de acessos a esse canal ($Acessos(B, C)$) pelo número de bits que compõem a mensagem trocada em cada acesso ($Bits(C)$):

$$TotalBits(B, C) = Acessos(B, C) * Bits(C) \tag{4.7}$$

O **tempo de comunicação** é definido como o tempo despendido por um módulo de comportamento a aceder à informação localizada no seu exterior. Se o atraso envolvido no envio ou recepção duma mensagem através dum canal C for $AtrasoCanal(C)$ e supondo que só se utiliza um canal, o tempo de comunicação é

[5]Neste caso, o nodo n_j diz-se antecessor do nodo n_i.

$$T_{com}(B,C) = Acessos(B,C) * AtrasoCanal(C) \tag{4.8}$$

O **tempo de execução** dum módulo de comportamento B é a soma do tempo de computação $T_{comp}(B)$ com o tempo de comunicação $T_{com}(B,C)$, em que o tempo de computação é definido pela equação 4.5 e o tempo de comunicação é definido pela equação 4.8. Daqui resultam as estimativas para a **taxa de transferência** média ($TaxaCom_{media}(C)$) e de pico ($TaxaCom_{pico}(C)$):

$$TaxaCom_{media}(C) = \frac{TotalBits(B,C)}{T_{comp}(B) + T_{com}(B,C)} \tag{4.9}$$

$$TaxaCom_{pico}(C) = \frac{Bits(B)}{AtrasoCanal(C)} \tag{4.10}$$

4.2.4 Estimação de Métricas de Custo

Nesta secção descreve-se o processo de estimação do espaço ocupado em cada componente de *hardware* e do número de pinos necessários à comunicação entre componentes. Para se poder estimar o espaço ocupado pelos recursos numa implementação de *hardware* é preciso determinar o número e o tipo de componentes envolvidos nessa implementação. Numa implementação com FPGAs o espaço pode ser medido em portas lógicas equivalentes e numa implementação com ASICs o espaço é medido em transístores. Seguindo o modelo de arquitectura ilustrado na figura 4.1, o espaço ocupado em *hardware* é composto por um caminho de dados e por uma unidade de controlo.

Estimação do espaço ocupado pelo caminho de dados

O caminho de dados inclui elementos de armazenamento (tais como registos, *latches* ou memória), unidades funcionais (tais como somadores, multiplicadores ou comparadores) e elementos de interligação (tais como multiplexadores ou barramentos).

Estimação dos elementos de armazenamento

Os elementos de armazenamento servem para guardar os valores das variáveis ou das constantes do comportamento do sistema. A estratégia mais simples a aplicar na estimação da quantidade de elementos de armazenamento necessária à implementação, considera a selecção dum elemento distinto por cada variável. Embora seja uma estratégia simples de implementar, a quantidade de recursos seleccionados é elevada. Para diminuir a quantidade de recursos

seleccionados, opta-se por partilhar cada elemento de armazenamento por mais do que uma variável. Para se atribuir um conjunto de variáveis ao mesmo elemento, tem que se garantir que o tempo de vida dessas variáveis não se sobrepõe. Um dos métodos que resolve o problema da atribuição das variáveis aos elementos de armazenamento é a **partição clique** [CLR90]. Este método determina quais as variáveis a atribuir a cada elemento de armazenamento, de modo a minimizar o número de elementos seleccionados. De entre as possíveis implementações do método de partição clique, apresenta-se apenas uma delas. O método começa pela construção dum grafo $G = \{N, A\}$, em que um nodo $n_i \in N$ representa uma variável v_i e um arco a_{ij}, entre os nodos n_i e n_j, significa que as variáveis v_i e v_j podem ser atribuídas ao mesmo elemento de armazenamento. Depois, em cada iteração do algoritmo agrupam-se os dois nodos que possuírem o maior número de nodos vizinhos em comum, substituem-se os dois nodos por apenas um e ligam-se ao nodo criado os arcos que antes ligavam aos dois nodos agora agrupados. O processo iterativo termina quando não for possível efectuar mais agrupamentos. Os nodos que resultarem do processo de agrupamento, conjuntamente com a informação relativa aos nodos que foram agrupados, fornecem uma estimativa para o número de elementos de armazenamento a seleccionar e uma atribuição de variáveis a esses elementos.

Estimação de unidades funcionais

As unidades funcionais implementam as operações presentes no comportamento do sistema. O método utilizado para estimar a quantidade de unidades funcionais depende do contexto em que decorre a estimação.

Quando o projectista define explicitamente a selecção de unidades funcionais, a estimativa para a quantidade de recursos é exacta e imediata.

Se o escalonamento do comportamento do sistema já foi efectuado, o método de partição clique permite estimar a quantidade de unidades funcionais a seleccionar. O objectivo do método é determinar que operações são atribuídas a cada unidade funcional a seleccionar, de modo a minimizar o número dessas unidades. A forma como se aplica a partição clique é idêntica à que se descreveu para a estimação dos elementos de armazenamento.

Quando o sistema está sujeito a um requisito de desempenho, traduzido por um número limite de etapas de controlo, a quantidade mínima de unidades funcionais a seleccionar pode ser estimada pelo método de escalonamento *force-directed* [GDWL92]. Este método procura distribuir uniformemente as operações do mesmo tipo pela totalidade das etapas de controlo, resultando numa utilização eficiente das unidades funcionais. Considerando apenas as operações que podem ser escalonadas em mais do que uma etapa de controlo, em cada iteração do algoritmo escalona-se uma das operações, começando com as operações que resultam no maior decréscimo do custo do escalonamento. Para baixar o custo do escalonamento, deve

aumentar-se a uniformidade com que as unidades funcionais são utilizadas nas diversas etapas de controlo.

Estimação de elementos de interligação

Depois de estimar os elementos de armazenamento e as unidades funcionais, estima-se a quantidade de elementos de interligação necessária para ligar as saídas dos elementos de armazenamento às entradas das unidades funcionais e a quantidade de elementos de interligação utilizada na ligação das saídas das unidades funcionais às entradas dos elementos de armazenamento. Como exemplo, apresenta-se o processo de estimação dos elementos que ligam as saídas dos elementos de armazenamento às entradas das unidades funcionais.

Conhecendo a atribuição das variáveis e das operações aos recursos seleccionados, uma análise da descrição do sistema identifica que ligações, entre elementos de armazenamento e unidades funcionais, é preciso implementar com multiplexadores, barramentos ou linhas dedicadas (figura 4.3 (i)). O problema que constitui a estimação dos elementos de interligação efectivamente necessários à implementação destas ligações também pode ser resolvido pelo método de partição clique. Tal como acontece com a estimação dos elementos de armazenamento ou das unidades funcionais, o método começa por construir um grafo. A cada ligação entre um elemento de armazenamento e uma entrada duma unidade funcional corresponde um nodo n_i no grafo. Quando duas ligações net_i e net_j forem utilizadas em etapas de controlo totalmente diferenciadas, inclui-se no grafo um arco a_{ij} a ligar os nodos n_i a n_j. Após efectuar todos os agrupamentos possíveis, cada nodo nf_i do grafo final representa um barramento bus_i que vai ligar a uma entrada duma determinada unidade funcional. A indicação de que os nodos iniciais $n_p, ..., n_q$ foram agrupados no nodo final nf_i, significa que os elementos de armazenamento $Reg_p, ..., Reg_q$ ligam ao barramento bus_i. Se mais de um barramento ligar à mesma entrada duma unidade funcional, será seleccionado um multiplexador para ligar os barramentos ao destino comum. A figura 4.3 (ii) ilustra o resultado da aplicação da partição clique na estimação dos elementos de interligação.

Com as estimativas dos elementos de armazenamento, das unidades funcionais e dos elementos de interligação é possível calcular o espaço ocupado pelo caminho de dados. Considerando que a estimativa para o número de elementos de armazenamento é $numReg$, para o número de unidades funcionais é $numUF$ e para o número multiplexadores incluídos nos elementos de interligação é $numMux$, o espaço ocupado pelo caminho de dados (CD) é aproximado por

$$area(CD) = \sum_{i=1}^{numReg} area(Reg_i) + \sum_{i=1}^{numUF} area(UF_i) + \sum_{i=1}^{numMux} area(Mux_i) \qquad (4.11)$$

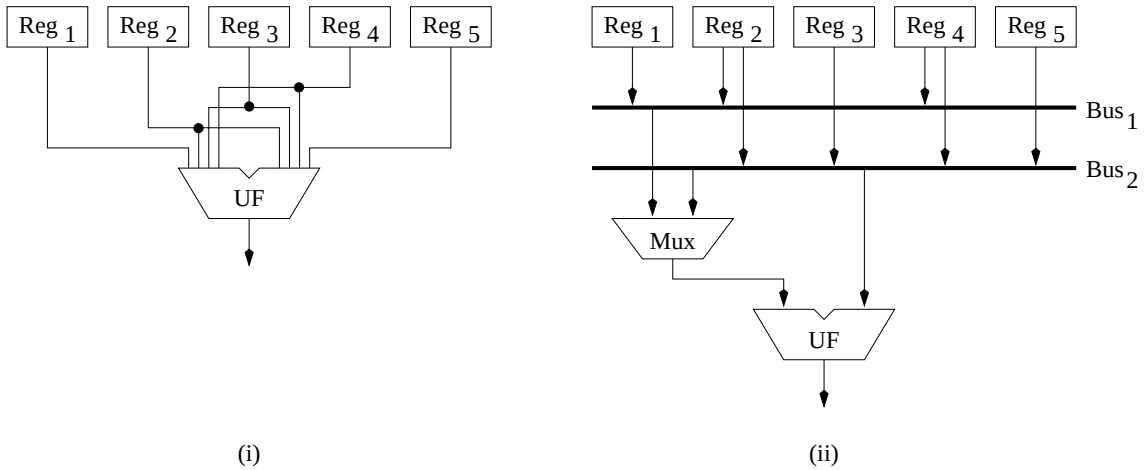


Figura 4.3: Exemplo de estimação dos elementos de interligação através do método de partição clique: (i) ligações que é preciso implementar e (ii) estimativa dos elementos de interligação necessários à implementação das ligações.

Estimação do espaço ocupado pela unidade de controlo

A unidade de controlo é composta pelo registo de estado, a lógica que gera o próximo estado e a lógica de controlo.

Para estimar o espaço ocupado pelo **registo de estado** é preciso calcular o número de bits do registo de estado ($Bits(RE)$). Se o número de etapas de controlo do escalonamento do sistema for $etapasCtl$, o valor de $Bits(RE)$ é dado por $log_2 etapasCtl$ ou $etapasCtl$, no caso de se optar por uma codificação de estados do tipo *one-hot*[6].

Para ilustrar a estimação do espaço ocupado pela **lógica que gera o próximo estado** e pela **lógica de controlo**, considera-se a implementação duma máquina de estados do tipo Moore através duma estrutura $AND - OR$. A partir das equações *booleanas* para os sinais de controlo (sinais gerados pela lógica de controlo) e para os sinais do próximo estado (sinais gerados pela lógica que gera o próximo estado), estima-se o número de portas lógicas OR, AND e NOT.

O número de **portas *OR*** coincide com o número de sinais de controlo mais o número de sinais do próximo estado. O número de entradas dum OR, envolvido na geração dum sinal de controlo, é dado pelo número de etapas de controlo em que esse sinal é utilizado no caminho de dados[7]. No caso em que o número de bits do registo de estado é $log_2 etapasCtl$ e assumindo que em média um sinal do próximo estado está activo em metade das etapas de controlo, o número de entradas dum OR envolvido na geração desse sinal é $\frac{etapasCtl}{2}$.

O número de **portas *AND*** envolvidas na geração dos sinais de controlo pode ser aproximado pelo seu limite superior, ou seja, pelo número de etapas de controlo ($etapasCtl$). Se o número

[6] O código associado a cada estado só possui um bit a 1.

[7] Um sinal de controlo serve para escrever (ler) num (dum) registo ou para operar uma unidade funcional.

de bits do registo de estado for $log_2etapasCtl$, o limite superior para o número de entradas destas portas AND coincide com $log_2etapasCtl$. Quando o número de bits do registo de estado for $log_2etapasCtl$ e a codificação dos estados obedecer ao código de Gray, o número de portas AND envolvidas na geração de sinais do próximo estado é aproximado pelo número de etapas de controlo. Nestas condições e considerando que na geração dum sinal do próximo estado intervém um sinal de estado (entrada da UC), o número de entradas dum AND é $log_2etapasCtl + 1$.

O número adicional de portas $NAND$ com duas entradas, que implementam as **portas *NOT***, coincide com o número de bits do registo de estado.

A partir do momento em que se dispõe duma estimativa para o número de bits do registo de estado, para o número de portas lógicas OR, AND e $NAND$ e para o número de entradas das portas lógicas, a estimativa do espaço ocupado pela unidade de controlo resulta da contabilização dos diferentes tipos de porta:

$$area(UC) = Bits(RE) * area(FlipFlop) + \sum_{n=2}^{max} num(OR_n) * area(OR_n) +$$

$$+ \sum_{n=2}^{max} num(AND_n) * area(AND_n) + Bits(RE) * area(NAND_2) \quad (4.12)$$

em que $num(OR_n)$ e $num(AND_n)$ designam o número de portas lógicas OR e AND com n entradas.

Estimação do número de pinos

A comunicação entre módulos de comportamento implementados em componentes distintos, processa-se obrigatoriamente através dos pinos dos componentes. A estimação do número de pinos necessário a esta comunicação depende do modelo de comunicação da arquitectura alvo e da maneira como a comunicação é especificada no comportamento do sistema. Considerando o modo como a comunicação é especificada, convém enumerar as situações mais comuns:

- o tamanho dos dados, envolvidos na comunicação entre componentes, pode ser definido explicitamente na declaração dos dados; por exemplo, quando numa HLL se declara que uma variável é do tipo inteiro, o seu tamanho coincide com o número de bits dum inteiro no sistema em que o código é compilado; numa HDL, como é o caso do VHDL, o tamanho dos sinais envolvidos na comunicação pode ser definido ainda mais explicitamente, bastando para isso que na declaração dos portos das entidades se indique o número de bits dos portos;

- ◇ algumas HDLs (como HardwareC ou SpecCharts) e HLLs (como o CSP) possuem o conceito abstracto de canal de comunicação; a tradução dum canal de comunicação em pinos é obtida pela síntese do canal, podendo resultar na concretização do canal através dum barramento com um certo número de linha de dados e de controlo;

- ◇ os módulos de comportamento também podem comunicar por variáveis (em HLLs) ou sinais (em HDLs) globais; a comunicação através de dados globais é menos explicita porque por exemplo, quando os módulos de comportamento que comunicam entre si são duas funções numa HLL, as variáveis globais não fazem parte da lista de parâmetros nem dos dados devolvidos;

- ◇ quando um módulo de comportamento invoca uma função implementada num módulo distinto, a largura do canal que suporta este tipo de comunicação depende do tamanho dos parâmetros da função, do tamanho dos dados devolvidos e do número de sinais usados no controlo do arranque e da conclusão da função.

Se num módulo de comportamento B, definido sob a forma de uma entidade em VHDL, a interface da entidade com o exterior incluir o conjunto de portos P, o conjunto de canais de comunicação utilizados for C, o conjunto de variáveis globais acedidas for V e o conjunto de funções invocadas for F, então o número de pinos que suporta a comunicação de B com o seu exterior é dado por

$$pinos(B) = \sum_{p_i \in P} Bits(p_i) + \sum_{c_i \in C} Bits(c_i) + \sum_{v_i \in V} Bits(v_i) + \sum_{f_i \in F} Bits(f_i) \tag{4.13}$$

em que $Bits(obj)$ representa o número de bits envolvidos na comunicação com o objecto obj.

4.3 Estimação de Métricas de *Software*

Apresentam-se agora métodos para estimar as métricas de *software* comuns em partição *hardware*/*software*, seja o tempo de execução, o espaço ocupado pelo código em memória ou o espaço necessário para armazenar os dados em memória.

4.3.1 Modelos para Estimação de Métricas de *Software*

Para estimar as métricas de *software* pode optar-se por um modelo genérico ou por um modelo que é específico dum processador. Os modelos genéricos aplicam-se nas situações em que não se exigem estimativas de grande qualidade. Os modelos específicos podem ser simples ou complexos.

Quando se opta por um modelo de estimação genérico, o código que define o comportamento do sistema é compilado para instruções genéricas. Partindo destas instruções, para obter estimativas de métricas relativas a um determinado processador exige-se apenas o ficheiro tecnológico desse processador. O ficheiro tecnológico define a temporização e o tamanho das instruções genéricas no processador em causa.

O modelo genérico que define o comportamento dum processador, concretizado num ficheiro tecnológico, deve suportar o seguinte leque de instruções ao nível da linguagem máquina: (i) instruções aritméticas/lógicas, (ii) instruções para transferência de informação entre memória, registos e I/O e (iii) instruções para alterar o fluxo de execução (salto condicional/incondicional e invocação/retorno de rotinas). Para obter o ficheiro tecnológico relativo a um determinado processador, é preciso determinar que conjunto de instruções (nativas desse processador) implementa a funcionalidade de cada instrução do modelo genérico. Somando o tempo de execução, ou o tamanho, das várias instruções que compõem esse conjunto obtém-se a temporização, ou o tamanho, da instrução genérica. Embora um modelo genérico gere estimativas menos precisas do que um modelo específico, apresenta vantagens significativas relativamente aos modelos específicos: (i) exige apenas um compilador e um mecanismo de estimação, (ii) suporta facilmente a inclusão de mais um processador na lista de processadores alvo da implementação, porque para isso só é necessário gerar um ficheiro com a temporização e o tamanho das instruções genéricas desse processador, (iii) a compilação é mais simples, uma vez que as instruções genéricas são em menor número e não incluem as instruções que exploram as especificidades do processador. As especificidades dum processador são modeladas, de forma simplificada, através dos atributos das instruções incluídas no ficheiro tecnológico desse processador.

Segundo o modelo específico simples, para obter estimativas das métricas basta compilar o código que define o comportamento do sistema para instruções dum processador específico. Com base na temporização e no tamanho das instruções nativas desse processador, determinam-se as métricas tamanho e tempo de execução associadas ao código gerado pelo compilador. O modelo específico gera estimativas mais precisas do que um modelo genérico equivalente (em termos de complexidade), mas obriga a ter um compilador para cada processador que seja opção de implementação dos sistemas e o elevado tempo de estimação limita a exploração do espaço de projecto.

Um modelo específico complexo é exigido apenas para a estimação de tempos de execução, uma vez que o cálculo do espaço ocupado pelo código e pelos dados não requer um modelo sofisticado para o processador. Um modelo de estimação complexo considera, além da temporização e do espaço ocupado pelas instruções, as optimizações relacionadas com a utilização de registos, a implementação de ciclos (mecanismo em que se desenrolam os ciclos), a utilização

de estágios de *pipeline* ou da hierarquia de memória.

4.3.2 Modelo Genérico para o *Software*

Esta secção descreve a aplicação dum modelo genérico na estimação de métricas de desempenho e de custo do *software*.

Estimação de métricas de desempenho

O **tempo de execução em *software*** pode ser estimado pelo método de simulação dinâmica ou pelo método de estimação estática. Para estimar o tempo de execução em *software* dum módulo comportamental do sistema com o método de simulação dinâmica, executa-se esse módulo diversas vezes com diferentes conjuntos de dados. Como é possível que a cada conjunto de dados corresponda um tempo de execução diferente, para estimativa do tempo de execução pode seleccionar-se o tempo médio das diversas execuções. Se o tamanho dos ciclos e a probabilidade de execução das ramificações do módulo forem conhecidos, consegue-se de forma estática uma estimativa para o tempo de execução desse módulo comportamental.

O método de estimação estática do tempo de execução em *software* é idêntico ao utilizado na secção 4.2.2 para estimar o tempo de execução em *hardware*. O método começa pela decomposição da descrição do módulo comportamental em blocos básicos de escalonamento (BBEs), que posteriormente são compilados para código com instruções genéricas. Com os BBEs a funcionar como nodos e os arcos a indicar as dependências de controlo entre dois BBEs, constrói-se um grafo de fluxo de controlo. O tempo de execução $T_{exec}(bbe_i)$ de cada bloco bbe_i é calculado a partir do tempo e da frequência de execução das diversas instruções genéricas que fazem parte de bbe_i. O tempo de execução das instruções genéricas provém do ficheiro tecnológico para o processador a usar na implementação e a frequência de execução de cada bloco bbe_i é obtida pela equação 4.6. Depois de conhecer o tempo e a frequência de execução de cada bloco, o tempo de execução do módulo B é calculado através da equação 4.5.

O tempo de execução obtido com a equação 4.5 não contabilizada as optimizações efectuadas pelo compilador. Uma alternativa simples para contabilizar no tempo de execução as optimizações de código, consiste em aplicar um **factor de optimização do compilador** (λ_t) sobre o tempo de execução relativo a código não optimizado:

$$T_{execOptim}(B) = \lambda_t * T_{exec}(B) \tag{4.14}$$

Resultados experimentais produziram um factor de optimização médio de 0.74, 0.68, 0.54 e 0.49 na compilação para o processador 8086, 80286, 68000 e 68020, respectivamente [GGN94].

Para processadores mais sofisticados, como o Pentium, o PowerPC ou o Sparc, este método é limitado para contabilizar a aceleração resultante dos vários estágios de *pipeline* nas unidades de inteiros e de vírgula flutuante, do grau de superescalaridade, dos esquemas de previsão de saltos ou dos esquemas de gestão da hierarquia de memória.

Estimação de métricas de custo

Para estimar o **espaço ocupado em memória pelo código** dum módulo comportamental, a descrição deste módulo é compilada para código com instruções genéricas. O espaço ocupado pelo módulo B obtém-se somando o espaço ocupado por cada uma das instruções genéricas $ig \in B_g$, em que B_g é o resultado da compilação de B:

$$TAM_{codigo}(B) = TAM_{codigo}(B_g) = \sum_{ig \in B_g} TAM_{instrucao}(ig) \tag{4.15}$$

O espaço ocupado pelas instruções genéricas, $TAM_{instrucao}(ig)$, faz parte do ficheiro tecnológico do processador alvo da implementação.

Para considerar as optimizações de código efectuadas pelo compilador, o espaço ocupado pelo código optimizado resulta da aplicação dum factor de optimização λ_c sobre o espaço ocupado pelo código não optimizado:

$$TAM_{codigoOptim}(B) = \lambda_c * TAM_{codigo}(B) \tag{4.16}$$

Convém lembrar que, a aplicação de factores representando a optimização do espaço ocupado pelo código (λ_c) e a optimização do tempo de execução (λ_t) não pode ser simultânea, uma vez que o compilador optimiza o código em relação ao espaço ou em relação ao tempo de execução.

A estimativa do **espaço ocupado em memória pelos dados** dum módulo comportamental, obtém-se analisando todas as declarações de dados contidas na descrição do comportamento. Deste modo, o espaço ocupado em memória pelos dados dum módulo B, que contém na sua descrição D declarações de dados, obtém-se somando o espaço ocupado pelos dados associados a cada uma das declarações $d \in D$:

$$TAM_{dados}(B) = \sum_{d \in D} TAM_{dados}(d) \tag{4.17}$$

O espaço ocupado pelos dados associados a uma declaração d depende do tamanho do tipo de dados declarado e do número de elementos na declaração:

$$TAM_{dados}(d) = nElementos(d) * TAM(tipo_{Lf}(d)) \tag{4.18}$$

em que

- ***nElementos(d)*** designa o número de elementos da declaração d;
- $\boldsymbol{TAM(tipo_{Lf}(d))}$ é o tamanho que o tipo de dados $tipo_{Lf}(d)$, associado à declaração d, ocupa na linguagem Lf utilizada para sintetizar a parte de *software* do sistema; se esta linguagem for distinta da linguagem Li em que o sistema está descrito, é preciso converter os tipos de dados da linguagem Li para os tipos da linguagem Lf (operador $conversao_{Li \rightarrow Lf}$ na equação 4.19):

$$tipo_{Lf}(d) = conversao_{Li \rightarrow Lf}(tipo_{Li}(d)) \tag{4.19}$$

Por exemplo, a tabela 4.1 apresenta uma possível conversão entre tipos de dados pré-definidos na linguagem VHDL (Li) e tipos de dados da linguagem C/C++ (Lf).

$\boldsymbol{conversao_{Li \rightarrow Lf}}$		$\boldsymbol{TAM(tipo_{Lf}(d))}$
$tipo_{Li}(d)$	$tipo_{Lf}(d)$	*(bytes)*
`bit`	`char`	1
`boolean`	`char`	1
`std_logic`	`char`	1
`character`	`char`	1
`integer`	`int`	4
`real`	`float/double`	4/8

Tabela 4.1: Conversão entre tipos de dados da linguagem VHDL (Li) e tipos de dados da linguagem C/C++ (Lf).

4.4 Abordagens para o *Hardware*

Nas secções 4.2 e 4.3 apresentou-se uma abordagem suficientemente genérica do problema de estimação dum conjunto alargado de métricas de *hardware* e de *software*, sem considerar em profundidade determinados aspectos de optimização. Nas próximas três secções descrevem-se algumas abordagens que se concentram na estimação de uma ou duas métricas de *hardware*, de *software* ou associadas à comunicação.

Estimação baseada numa biblioteca de componentes parametrizados

Na abordagem descrita em [DJ92] estimam-se as métricas **espaço e atraso em *hardware***, através dum processo de síntese. O método de estimação recorre a uma biblioteca de componentes parametrizados (no nível de transferência de registos), de modo a reduzir o número de alternativas a analisar durante a exploração do espaço de projecto.

O problema de estimação é formalizado por um trio $< G, PG, Fe >$, em que $G = \{G_i\}$ é o conjunto dos geradores de componentes, $PG = \{PG_i\}$ é o super-conjunto dos conjuntos de parâmetros $PG_i = \{P_{i1}, P_{i2}, ..., P_{in}\}$ para cada um dos geradores G_i e $Fe = \{FE, FA\}$ é o conjunto das funções para estimação do espaço ($FE = \{FE_i\}$) e do atraso ($FA = \{FA_i\}$). O conjunto de componentes C_i que se pode obter com o gerador G_i é definido por $C_i = G_i * (D_{i1} \times D_{i2} \times ... \times D_{in})$[8], com D_{ik} a representar o conjunto de alternativas para o k-ésimo parâmetro de PG_i. O conjunto de alternativas de implementações para $C_{ij} \in C_i$ é $S_{ij} = \{S_{ijk}\}$, onde S_{ijk} representa a implementação ordem k do componente C_{ij}. Cada implementação S_{ijk} faz corresponder um valor E_{ijk} à métrica espaço e um valor A_{ijk} à métrica atraso. Com base no conjunto de valores E_{ij} do espaço e no conjunto de valores A_{ij} do atraso, para um sub-conjunto de componentes C_{ij} passível de ser gerado por G_i, derivam-se modelos para o espaço (FE_{ij}) e para o atraso (FA_{ij}). Estes modelos permitem estimar o espaço e o atraso relativos a componentes (a gerar por G_i) com parâmetros diferenciados dos apresentados pelos componentes usados na geração destes modelos. Como os valores das métricas dependem dos parâmetros PG_{ij} e do tipo de implementação S_{ij} do componente C_{ij}, os modelos do espaço (FE_{ij}) e do atraso (FA_{ij}) são expressos pelas equações 4.20 e 4.21, respectivamente.

$$FE_{ij} = e_{ij0} + \sum_k e_{ijk} * fE_{ijk}(PG_{ijk}) \tag{4.20}$$

$$FA_{ij} = a_{ij0} + \sum_k a_{ijk} * fA_{ijk}(PG_{ijk}) \tag{4.21}$$

em que fE_{ij} e fA_{ij} são funções e os vários e_{ij} e a_{ij} são constantes ou coeficientes.

Por exemplo, os modelos de estimação do espaço e do atraso dum multiplexador com ne entradas de te bits são traduzidos pelas equações 4.22 e 4.23. As estimativas das métricas de *hardware* obtidas pelos modelos descritos nesta abordagem, apresentam um erro inferior a 10% em relação aos valores obtidos nas implementações.

$$Area = te * (e_1 + e_2 * log_2(ne) + e_3 * ne) \tag{4.22}$$

$$Atraso = a_1 + a_2 * log_2(ne) + a_3 * ne \tag{4.23}$$

Estimação na abordagem SpecSyn

Com o objectivo de reduzir o tempo de cálculo necessário à obtenção de estimativas para o **espaço ocupado em *hardware***, durante a aplicação de algoritmos que analisam milhares

[8] O operador "×" significa o produto cartesiano.

de alternativas de partição, a abordagem SpecSyn aplica um método de estimação incremental [VG95b]. Este método baseia-se no princípio de que para obter a estimativa relativa a uma alternativa de partição, se pode usar a estimativa da alternativa de partição anterior e efectuar apenas os ajustes devidos às alterações que ocorreram da alternativa anterior para a actual. Para atingir este objectivo, desenvolveu-se uma estrutura de dados e um algoritmo de estimação. A estrutura de dados representa o sistema de acordo com o modelo de estimação UC/CD da figura 4.1.

Na primeira fase do processo de estimação, inclui-se na estrutura de dados a informação relativa à implementação em *hardware* de cada um dos objectos do sistema. Esta informação, obtida *a priori* do processo de partição, é definida pelo quarteto $infPP =< O, CD_{in}, CD_{out}, U >$, em que O é o conjunto de objectos de comportamento do sistema, CD_{in} é o conjunto de entradas do caminho de dados, CD_{out} é o conjunto de saídas do caminho de dados e U é o conjunto de unidades funcionais e de armazenamento disponíveis para a implementação. Uma unidade u_i é representada por um par $< tam_i, ctl_i >$, onde tam_i define o espaço ocupado pela unidade u_i e ctl_i o número de sinais que controla u_i. Cada objecto o_i é definido pelo par $<estados_i, dest_i>$, onde $estados_i$ representa o número de estados necessários ao escalonamento de o_i e $dest_i = \{dest_{i1}, dest_{i2}, ...\}$ é o conjunto de alvos das operações de escrita de o_i. Um alvo $dest_{ij}$ é definido por $< id_{ij}, orig_{ij}, activ_{ij} >$, em que id_{ij} identifica o alvo da escrita, $orig_{ij} = \{orig_{ij1}, orig_{ij2}, ...\}$ é a lista dos pontos de origem que vão ligar ao alvo $dest_{ij}$ e $activ_{ij}$ define o número de estados em que o alvo $dest_{ij}$ guarda informação válida para o objecto o_i.

Na segunda fase constrói-se o modelo UC/CD para a implementação, combinando a informação[9] relativa aos vários objectos atribuídos a *hardware*. A estrutura de dados que permite a actualização incremental do modelo de estimação é definida por $infD =$ < *tam*, *unid*, *ctl*, *tamDP*, *dest* >, em que *tam* é o espaço ocupado por todas as unidades[10], *unid* é o número dessas unidades, *ctl* é número de sinais de controlo entre a UC e o CD, *tamDP* representa a largura do CD e *dest* é o conjunto de todos os alvos das operações de escrita. Cada alvo $dest_i$ é definido por $< id_i, conn_i, activ_i >$, em que id_i identifica o alvo da escrita, $conn_i = \{conn_{i1}, conn_{i2}, ...\}$ é a lista de ligações $conn_{ij} =< orig_{ij}, obj_{ij} >$ que começam no ponto $orig_{ij}$, terminam no alvo $dest_i$ e são utilizadas pelo objecto obj_{ij} e $activ_i$ define o número de estados em que o alvo $dest_i$ está activo.

Na terceira fase, ou seja, em cada iteração do algoritmo de partição, actualiza-se a informação $infD$ do modelo UC/CD com a informação de $infPP$ relativa ao objecto que se atribui ou retira de *hardware*.

[9] Nesta informação incluem-se as unidades funcionais, o número de entradas dos multiplexadores e o número de estados da UC.

[10] Unidades funcionais, elementos de armazenamento e multiplexadores.

No caso dum sistema com N objectos, a complexidade temporal do algoritmo que constrói as estruturas de dados $infPP$ e $infD$ é $\mathcal{O}(N)$. Este método de estimação reduz o tempo de cálculo cerca de duas ordens de grandeza relativamente aos métodos que em cada iteração do processo de partição refazem todos os cálculos envolvidos na estimação das métricas. O tempo necessário ao deslocamento dum objecto é praticamente constante, ou seja, é independente da dimensão dos objectos e da dimensão do sistema a implementar. Contudo, as estimativas obtidas por este método só são precisas quando se usam objectos de comportamento com granulosidade grossa.

Estimação na abordagem Cosyma

O processo de partição seguido na abordagem Cosyma decorre em dois ciclos, um ciclo interior com milhares de iterações geradas pelo algoritmo de *simulated annealing* e um ciclo exterior controlado pelo processo de estimação. A estimativa E_k obtida numa determinada iteração do ciclo exterior, é corrigida pelo valor real V_k obtido com a síntese da alternativa de partição corrente. Com o valor real V_k estima-se o valor E_{k+1} para a iteração seguinte. A convergência do ciclo exterior para uma solução que atinge os requisitos de desempenho faz-se em poucas iterações [HE95] [HE98].

No método apresentado em [HE95], as estimativas do **tempo de execução em *hardware*** são obtidas escalonando os caminhos que compõem o grafo que modela o sistema. O escalonamento dos caminhos do grafo, efectuado com o algoritmo de partição clique[11] [CLR90], envolve as seguintes tarefas: (i) converter o grafo CDFG que modela o sistema para um grafo acíclico direccionado (DAG), (ii) determinar todos os caminhos do grafo, (iii) escalonar cada caminho do grafo o mais cedo possível (estratégia ASAP[12]) e (iv) verificar se a junção de todos os caminhos atinge os requisitos de desempenho exigidos. Para diminuir o tempo necessário à estimação do tempo de execução, deve reduzir-se o número de caminhos e/ou de operações a analisar. A redução do número de caminhos consegue-se com uma divisão do grafo em partes e com o escalonamento em separado de cada parte em que se dividiu o grafo. A divisão dum grafo em sub-grafos pode conduzir a um escalonamento mais longo, que normalmente se traduz num aumento do tempo de execução. O nível de degradação do escalonamento é tanto maior quanto maiores forem as dependências entre as operações localizadas em ambos os lados de cada ponto em que se cortou o grafo e quantos mais recursos houver para o escalonamento.

Verifica-se que o tempo de cálculo, necessário para estimar o tempo de execução, baixa com o aumento do número de pontos de corte do grafo e que são os primeiros cortes do grafo que têm mais impacto na redução do tempo de cálculo. A partir dum certo número de pontos de

[11] A secção 4.2 contém exemplos de aplicação deste algoritmo na estimação de métricas.
[12] ***A**s **S**oon **A**s **P**ossible*, na terminologia inglesa.

corte, designado de ponto de saturação, a redução do tempo de cálculo não é significativa. As estimativas obtidas nas condições do ponto de saturação apresentam um erro inferior a 20% em relação à melhor estimativa do tempo de execução.

A estimação do **espaço ocupado em *hardware*** aplica uma técnica de alto nível [HE98], que contabiliza o espaço do caminho de dados (CD) e da unidade de controlo (UC) e considera que o grafo CDFG que descreve o sistema já foi escalonado. O modelo de estimação para o CD é idêntico ao apresentado na figura 4.1, ou seja, o caminho de dados é composto por registos, multiplexadores, unidades funcionais e multiplexadores.

Para estimar o número de unidades funcionais percorrem-se as várias etapas de controlo, atribuindo a cada operação escalonada nessas etapas uma unidade funcional previamente seleccionada, mas que está livre, ou então uma unidade funcional seleccionada no momento. A unidade funcional atribuída tem que ser capaz de executar o tipo de operação em causa e quando houver alternativa de selecção, a escolha recai sobre aquela que ocupar menos recursos.

Para estimar o número de registos percorre-se cada um dos caminhos do grafo do sistema, de modo a determinar o tempo de vida de todas as variáveis. O **tempo de vida duma variável** é o conjunto de etapas de controlo que vai desde a primeira atribuição a essa variável até à última utilização da variável. Aplicando o algoritmo de partição clique ao conjunto que contém o tempo de vida de todas as variáveis, obtém-se o número mínimo de registos que implementa todas as variáveis do sistema. A estimativa do espaço ocupado pelos registos é imprecisa por considerar que todos os registos possuem o mesmo tamanho.

A estimativa do espaço ocupado pelos multiplexadores é composta por duas parcelas: (i) o espaço ocupado pelos multiplexadores que encaminham os operandos dos registos para as unidades funcionais (multiplexadores de entrada) e (ii) o espaço ocupado pelos multiplexadores que encaminham os resultados das unidades funcionais para os registos (multiplexadores de saída). Por exemplo, a estimação do número de multiplexadores de entrada processa-se do seguinte modo:

Para cada tipo t de unidade funcional seleccionado e para cada instância k de uma unidade funcional desse tipo t, guarda-se numa posição distinta da lista $MuxList$ o número de operações $numOP(UF_t, k)$ que partilha a instância k da unidade funcional UF_t. Se a posição *pos* da lista $MuxList$ contiver o valor 3, o CD inclui dois[13] multiplexadores 3 para 1. A estimativa não é muito precisa porque despreza a informação relativa à atribuição de variáveis aos registos e considera que todas as variáveis são do mesmo tamanho. Assim, a estimativa do número de entradas dos multiplexadores assume o limite superior.

Combinando o espaço ocupado por unidades funcionais ($Area_{UF}$), registos ($Area_{Reg}$), multi-

[13] No caso duma unidade funcional UF_t com dois operandos.

plexadores ($Area_{MuxIn}$ e $Area_{MuxOut}$) e unidade de controlo ($Area_{UC}$), obtém-se o valor do espaço ocupado em *hardware* (equação 4.24).

$$Area_{HW} = Area_{UF} + Area_{Reg} + Area_{MuxIn} + Area_{MuxOut} + Area_{UC} \tag{4.24}$$

Embora o tempo de cálculo das estimativas seja muito menor que o tempo exigido pela síntese efectuada no nível de transferência de registos, um grau de fidelidade médio de 80% e uma precisão média de 50% são valores apenas razoáveis.

Estimação numa abordagem baseada na representação formal LOTOS

Na abordagem descrita em [CLL+96] os sistemas são especificados através dum conjunto de processos concorrentes que comunicam e interagem entre si, concretizado com a linguagem **LOTOS**. Os sistemas, predominantemente de controlo, são implementados numa arquitectura alvo que inclui um processador, um ASIC e memória, interligados por um barramento comum. Antes de efectuar a partição, converte-se a descrição LOTOS do sistema para um grafo que representa a comunicação entre processos (GCP), em que os nodos estão associados aos processos e os arcos à comunicação entre processos.

Cada nodo do grafo GCP é anotado com estimativas para o espaço ocupado em *hardware*, o tempo de execução em *hardware* desde cada uma das entradas até cada uma das saídas, o espaço ocupado em *software*, o tempo de execução em *software* e o número de vezes que o nodo é executado. Analogamente, cada arco é anotado com estimativas do tempo gasto em sincronização e do número de vezes que existe sincronização. Os condicionalismos a respeitar pelo processo de partição são o espaço disponível em *hardware* e a quantidade de memória disponível para guardar o *software*, enquanto o requisito de desempenho a atingir é o tempo de resposta exigido ao sistema.

A estimação de métricas de *hardware* aplica um modelo UC/CD, como o que foi apresentado na figura 4.1. As entradas no processo de estimação são (i) os CDFGs que descrevem os blocos que resultaram da decomposição dos nodos do GCP e (ii) uma biblioteca de componentes parametrizados em relação à arquitectura do componente e à tecnologia de implementação. A estimativa do **tempo de execução** dum CDFG obtém-se somando o tempo de execução dos nodos incluídos em cada caminho do grafo. Em função do tempo de execução dum DFG puro, definiu-se o tempo de execução para os seguintes tipos de nodo dum CDFG: nodo correspondente à utilização duma função, nodo correspondente a uma operação condicional e nodo correspondente a um ciclo.

Para estimar a estrutura do caminho de dados dum DFG modelam-se as tarefas da síntese de alto nível, tais como a selecção de componentes da arquitectura alvo, a atribuição de operações

a esses componentes, a selecção da frequência de relógio e o escalonamento das operações nas etapas de controlo. Na tarefa de atribuição considera-se que cada operação só é executável por um tipo de unidade funcional. A frequência de relógio é seleccionada de modo a minimizar a inactividade média de todas as unidades funcionais [NG92a]. As tarefas de escalonamento e de selecção geram a informação sobre o tempo e os recursos a utilizar na execução do CDFG. Aplicando a estratégia ASAP, efectua-se o escalonamento com duas selecções de componentes distintas: ($S1$) uma selecção com componentes em número ilimitado e ($S2$) uma selecção que inclui apenas um componente por cada tipo de operação.

O escalonamento com a selecção $S1$ determina o número máximo de componentes para cada tipo i de operação (NC_i), o número de operações atribuídas a cada instância j dum componente do tipo i ($N_{i,j}$) e o tempo mínimo necessário para executar o CDFG (T_{min}). O valor de T_{min} resulta da soma de duas parcelas: o tempo de execução das operações, calculado na situação em que se executam em paralelo todas as operações do mesmo tipo que podem ser escalonadas na mesma etapa de controlo e o tempo que se espera para aceder à memória comum. Este tempo adicional resulta de acessos à memória, efectuados em simultâneo pelo *hardware* e pelo *software*.

Efectuando um escalonamento com a selecção $S2$, é possível determinar experimentalmente o tempo máximo necessário à execução do CDFG (T_{max}) e depois calcular uma constante β necessária ao escalonamento com uma selecção genérica. Os valores de T_{max} e β são obtidos na situação em que se executam de forma sequencial todas as operações do mesmo tipo.

Considerando a situação realista em que se seleccionam NS_i componentes do tipo i, a estimativa para o tempo de execução do CDFG resulta da execução em paralelo de NS_i operações do tipo i e da execução sequencial das restantes ($NC_i - NS_i$) operações desse tipo:

$$T_{CDFG} = T_{min} + \beta * \sum_{i} [\ \frac{Tuf_i}{NS_i} * \sum_{j=NS_i+1}^{NC_i} N_{i,j}\ ] \tag{4.25}$$

em que os valores de T_{min}, NC_i e $N_{i,j}$ resultaram do escalonamento com a selecção $S1$, β resultou do escalonamento com a selecção $S2$ e Tuf_i é o tempo de resposta dum componente do tipo i. A estratégia apresentada para estimar o tempo de execução dum CDFG possui duas limitações: concentra a influência das dependências de dados numa simples constante (β) e aplica parâmetros que são determinados experimentalmente antes do processo de partição. A primeira limitação introduz erros significativos nas estimativas e a segunda condiciona o grau de automatização do processo de partição.

Na estimação do número de registos necessário para guardar as variáveis considera-se o tempo de vida dessas variáveis, de modo a que um registo possa ser partilhado por mais de uma

variável.

A unidade de controlo é sintetizada como uma máquina de estados, que é implementada com um registo de estado e uma ROM. O tamanho do registo de estado é inferido a partir do número de etapas de controlo presentes no escalonamento do CDFG do sistema. Para estimar o espaço ocupado pela ROM considera-se que ela é responsável por gerar um sinal de controlo por cada registo e por cada unidade funcional e $log_2 TAM^k_{mux}$ sinais de controlo por cada multiplexador k com TAM^k_{mux} entradas.

Estimação na abordagem COOL

A abordagem COOL [Nie98] efectua a partição com o objectivo de obter soluções que ocupam o menor espaço de *hardware* possível e respeitam os requisitos de desempenho. As métricas de *hardware* relativas aos módulos da estrutura do sistema são calculadas com as ferramentas de síntese de *hardware*. Como a síntese é morosa, esta estratégia de estimação não é a mais adequada para um processo que normalmente analisa muitas alternativas de partição. Para atenuar o efeito negativo que resulta de a síntese ser morosa, as métricas de *hardware* relativas aos módulos do sistema só são calculadas uma vez.

A estimação do **espaço ocupado** e do **tempo de execução** em *hardware* decorre nas seguintes etapas:

- construir um grafo CDFG a partir da descrição em VHDL do comportamento da parte de *hardware* do sistema; este grafo tem a estrutura dum CFG, em que cada nodo é um DFG que representa as operações presentes num bloco da descrição VHDL;
- aplicar um conjunto de transformações algébricas às expressões que fazem parte de cada bloco VHDL, de modo a reduzir o custo da lógica a sintetizar para estas expressões;
- equacionar o escalonamento de operações, a selecção de componentes e a atribuição de operações aos componentes através duma formulação PLI; a resolução desta formulação, por intermédio duma ferramenta como *lp_ solve* [Ber98], gera uma lista com informação sobre cada etapa de controlo necessária ao escalonamento do grafo CDFG; por cada etapa de controlo e unidade funcional seleccionada, a resolução da formulação PLI especifica que operação está atribuída a essa unidade funcional durante a etapa de controlo em causa;
- determinar o número de registos, de unidades funcionais e de multiplexadores do caminho de dados, que satisfazem todas as etapas de controlo resultantes do escalonamento do sistema; desprezando o espaço ocupado pela unidade de controlo, pelo facto de nos sistemas predominantemente de fluxo de dados ser muito menor que o espaço do CD,

a soma do espaço ocupado por registos, unidades funcionais e multiplexadores fornece uma estimativa para o espaço ocupado pela parte de *hardware* do sistema;

- ◇ anotar os nodos do CFG obtido na primeira etapa com o número de etapas de controlo necessárias à execução das operações incluídas nesses nodos; atravessando o CFG, o tempo de execução dum caminho obtém-se somando o contributo dos vários nodos incluídos nesse caminho; o maior tempo de execução, de entre todos os caminhos, funciona como estimativa para o tempo de execução da parte de *hardware* do sistema.

4.5 Abordagens para o *Software*

Esta secção faz uma síntese de abordagens ao problema de estimação das métricas de *software*. As métricas estimadas são o desempenho, sob a forma de tempo de execução, e o espaço ocupado pela implementação em *software*.

Estimação na abordagem Vulcan

Na abordagem Vulcan estimam-se métricas de *software* com o objectivo de verificar se a funcionalidade atribuída ao *software* respeita os requisitos de desempenho exigidos ao sistema [Gup93][GM94]. O comportamento do sistema é especificado em HardwareC e depois convertido para um grafo de fluxo (G) com grão fino. Os requisitos do sistema são introduzidos no grafo sob a forma de arcos pesados.

Para estimar o tamanho da implementação do grafo G em *software*, contabiliza-se o tamanho do programa e dos dados. O modelo G é complementado com a representação $M(G)$ dos elementos de armazenamento utilizados pelas operações de G e com a lista $P(G)$ que contém o tamanho de todas as entradas e saídas do sistema. Deste modo, o modelo do sistema é definido por $\Upsilon =< G, M(G), P(G) >$. A estimação do tamanho do *software* exige que se conheça, além do modelo do sistema, o modelo do processador a utilizar na implementação e o ambiente de suporte à execução[14].

Do ponto de vista da compilação, o **modelo do processador** é caracterizado pela arquitectura do seu conjunto de instruções[15], que é composta pelo conjunto de instruções e pelo modelo de memória. A abordagem Vulcan considera um processador do tipo máquina de registos genérica, em que as instruções só suportam operandos explícitos e o modelo de memória emprega um esquema de endereçamento ao nível do byte. O sub-conjunto de instruções seleccionado permite manipular memória, executar operações aritméticas/lógicas e alterar o fluxo de controlo.

[14] *Runtime System*, na terminologia inglesa.
[15] ***I****nstruction* ***S****et* ***A****rchitecture* (***ISA***), na terminologia inglesa.

Em *software* exige-se um **ambiente de suporte à execução** quando os recursos e as tarefas são determinados durante a execução, e não durante a compilação. Na situação comum em que não se conhece a ordem de execução de todas as rotinas, não é possível determinar de forma estática todas as atribuições de recursos a tarefas. Deste modo, é impossível obter uma sequência com todas as operações, tendo que se recorrer a fios de execução (*threads*) concorrentes. Um **fio de execução** dum programa é um conjunto de operações com execução puramente sequencial, em que o único não-determinismo existente é o início da execução.

Implementar um sistema em *software* equivale a gerar uma sequência de instruções do processador, a partir do modelo Υ. Para compensar a diferença de nível de abstracção entre os pontos de partida e de chegada, este processo decorre nas seguintes etapas: (i) gerar um conjunto linear de operações reunidas em fios de execução, (ii) obter o código para as rotinas que controlam os fios de execução e (iii) obter o código máquina correspondente às rotinas (compilação). Um conjunto linear de operações é uma sequência de operações, escalonadas como se em cada instante só existisse um operador disponível para executar cada tipo de operação.

Na abordagem Vulcan considera-se que o tempo de execução em *software* é composto por duas parcelas, uma representa o tempo de execução das operações presentes no grafo G e a outra provém do ambiente de suporte à execução. Para simplificar, considera-se que a segunda parcela constitui apenas um intervalo constante. Para uma operação *op* do grafo G, a que correspondem $num(op)$ instruções em *assembly*, com um total de $m_r(op)$ acessos à memória para leitura e de $m_w(op)$ acessos à memória para escrita, o **tempo de execução** dessa operação em *software* $T_{exec}(op)$ é estimado pela equação 4.26.

$$T_{exec}(op) = \sum_{i=1}^{num(op)} t_{exec}(I_i(op)) + [m_r(op) + m_w(op)] * t_{mem} \tag{4.26}$$

em que $t_{exec}(I_i(op))$ designa o tempo de execução da instrução I_i e t_{mem} é o tempo necessário para ler (escrever) um operando (resultado) de (em) memória. O tempo t_{mem} inclui o cálculo do endereço efectivo e o acesso à memória.

O **espaço ocupado em *software*** por um sistema modelado por $\Upsilon =< G, M(G), P(G) >$ obtém-se somando o espaço ocupado pelo código com o espaço ocupado pelos dados declarados estaticamente. Para o espaço ocupado pelos dados contribuem as variáveis que guardam os valores necessários à execução das operações do grafo G, assim como as variáveis que guardam os valores intermédios utilizados nas instruções que codificam essas operações. Este espaço pode ser aproximado por um sub-conjunto das variáveis necessárias à execução das operações do grafo G, uma vez que nem todas as variáveis têm um tempo de vida igual ao tempo de execução de todo o programa. Constata-se que o número de instruções geradas pelos

compiladores está directamente relacionado com o número de termos incluídos no lado direito das atribuições. Deste modo, o espaço ocupado pelo código pode ser aproximado pelo número de termos incluídos no lado direito de todas as atribuições.

A aproximação utilizada no cálculo do espaço ocupado em *software*, em que se considera o espaço ocupado pelo código e pelos dados declarados estaticamente e se despreza o espaço necessário à utilização de rotinas (parâmetros e variáveis locais), é acertada se se tratar dum processador RISC, mas incorrecta quando se trata dum processador CISC e/ou duma situação em que as rotinas possuem muitos parâmetros ou variáveis locais. Para uma boa parte das situações, um processador RISC possui registos suficientes para guardar toda a informação associada com a invocação de rotinas.

Estimação na abordagem Polis

Em [SSV96] descreve-se a estimação do **desempenho em *software***, aplicada no desenvolvimento de sistemas reactivos tempo real com a abordagem Polis. Inicialmente e no nível de abstracção mais alto, os sistemas são modelados por uma rede de CFSMs. Depois, a parte de *hardware* gerada pela partição do modelo CFSM é convertida numa representação orientada para *hardware* e a parte de *software* é convertida num grafo orientado para *software* (grafo-S). O grafo-S é um grafo acíclico direccionado que representa o fluxo de controlo. A figura 4.4 mostra o modelo CFSM dum pequeno sistema e a correspondente representação com o grafo-S. As métricas calculadas ao nível da CFSM são aplicadas no processo de partição, enquanto as métricas mais precisas, estimadas ao nível do grafo-S, permitem optimizar e escalonar a parte de *software* do sistema.

A estimação do desempenho em *software* baseia-se no modelo da arquitectura alvo e na estrutura do código que descreve o sistema. A estrutura do código é fixa, assumindo a forma de uma função composta por três partes: (i) um bloco B_{inic} em que se iniciam variáveis, (ii) um bloco B_{constr} com construtores condicionais (`if, switch`) e atribuições e (iii) a instrução `return`. Deste modo, para o tempo de execução $Texec_{SW}$ e para o espaço TAM_{SW} ocupado pelo código contribuem as três partes que compõem a função, como se indica nas equações 4.27 e 4.28. Os ciclos que possam existir num sistema são tratados pelo sistema operativo.

$$Texec_{SW} = T_{pp} + k * T_{inic} + T_{constr} \tag{4.27}$$

$$TAM_{SW} = TAM_{pp} + k * TAM_{inic} + TAM_{constr} \tag{4.28}$$

onde T_{pp} representa o tempo para chamar e sair da função, $k * T_{inic}$ é o tempo para iniciar k

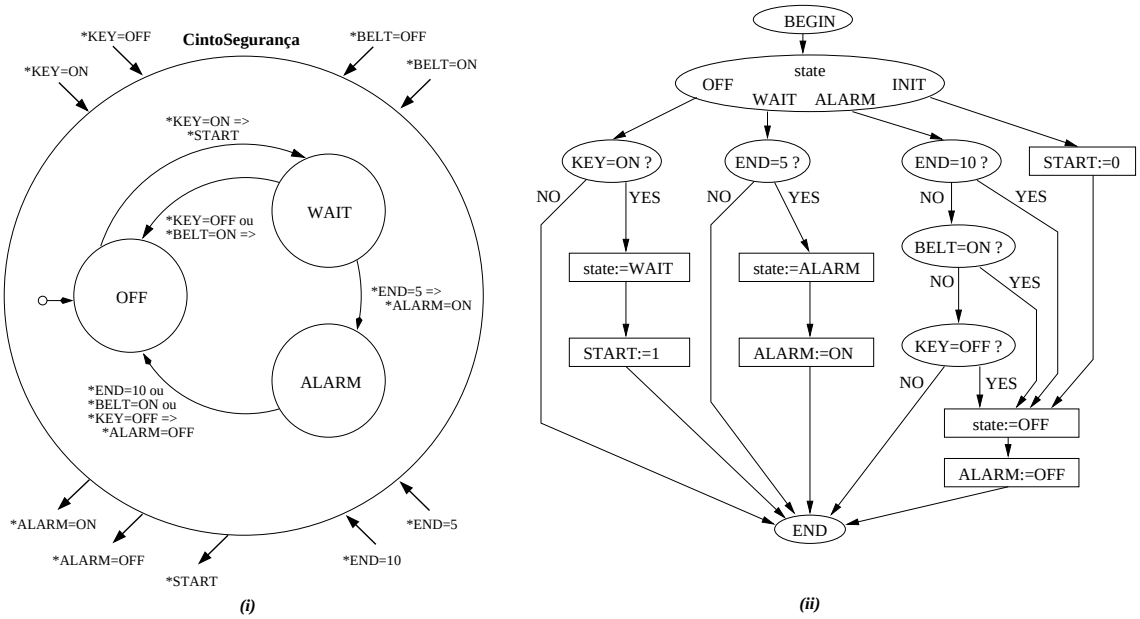


Figura 4.4: Sistema *CintoSegurança* (i) modelado com CFSMs e (ii) representado com o grafo-S.

variáveis e T_{constr} é tempo envolvido na execução dos construtores condicionais e atribuições.

Para o valor de T_{constr} contribuem os nodos de decisão e de atribuição nos caminhos do grafo-S, ou seja, os nodos elípticos com uma interrogação "?" e os nodos rectangulares na figura 4.4 (ii). O tempo de execução e o espaço ocupado por cada declaração do bloco B_{constr} depende do tipo de declaração, do código gerado pelo compilador e do desempenho do processador a usar na implementação. A definição do espaço TAM_{SW} ocupado pelo código é análoga à definição do tempo de execução $Texec_{SW}$.

Como o código C é gerado a partir dum grafo acíclico direccionado não inclui ciclos, e por isso a influência da memória *cache* só ocorre se o mesmo código tiver sido executado recentemente. A probabilidade de determinado código estar na *cache* depende do escalonamento das CFSMs, do número e tamanho das CFSMs implementadas em *software* e do tamanho da *cache*. A percentagem de *cache hits* e de falhas no *pipeline*, originadas por saltos no fluxo de execução, podem ser obtidas por simulação das CFSMs.

A arquitectura alvo é caracterizada por (i) atributos como a identificação do processador, a unidade de tempo, a unidade de espaço e o tamanho duma variável inteira e (ii) parâmetros como o tempo e o espaço associados à invocação e conclusão duma função, o tempo e o espaço associados a uma operação de ramificação resultante dum *goto*, o tempo e o espaço necessários para iniciar uma variável local, o tempo e o espaço médios associados a funções pré-definidas[16], o tamanho dum apontador e dum inteiro.

A estimação ao nível do grafo-S aplica um algoritmo que percorre todos os caminhos do grafo,

[16] Funções aritméticas, lógicas e relacionais são exemplos de funções pré-definidas.

determinando o caminho com o maior e o menor tempo de execução e o espaço ocupado pelo código de todo o grafo. O tempo de execução dum caminho obtém-se somando o tempo de execução dos nodos com o tempo associado aos arcos incluídos nesse caminho. A precisão das estimativas ao nível do grafo-S é aproximadamente 90% no caso do tempo de execução máximo, 80% no caso do tempo de execução mínimo e 80% no caso do espaço ocupado pelo código.

O comportamento duma CFSM é representado por uma árvore de decisão multi-valor (ADMv) cuja estrutura é idêntica à dum grafo-S. Uma ADMv pode ser convertida num grafo-S, o que torna possível a estimação do tempo de execução, ao nível da CFSM, com um algoritmo idêntico ao usado ao nível do grafo-S. A precisão das estimativas obtidas ao nível da CFSM é menor que a das estimativas calculadas ao nível do grafo-S, baixando para 75% no caso do tempo de execução mínimo.

Estimação na abordagem COOL

Na abordagem COOL [Nie98] para obter estimativas das métricas de *software* compila-se o código C, que descreve a parte de *software* do sistema, para código *assembly* do processador alvo. O espaço ocupado pelo código é calculado pela soma do tamanho de todas as instruções *assembly* geradas pela compilação, enquanto o espaço ocupado pelos dados se obtém somando o tamanho das diversas variáveis globais existentes no código C. Para estimar o tempo de execução na pior situação, utiliza-se um grafo CFG que é analisado de forma estática para encontrar o caminho com o maior tempo de execução. A análise do grafo decorre em quatro fases: (i) construir o CFG, em que os nodos são instruções do código *assembly* e os arcos representam o fluxo de controlo, (ii) anotar os nodos do CFG com o tempo de execução das instruções que lhe estão associadas e anotar os arcos que representam ciclos com o número de iterações, (iii) multiplicar o tempo de execução dos nodos incluídos dentro de ciclos pelo número de iterações e (iv) atravessar todo o grafo, somando o tempo dos diversos nodos de cada caminho, de modo a gerar o tempo de execução desse caminho. O maior tempo de execução entre todos os caminhos, funciona como estimativa do tempo de execução da parte de *software* do sistema.

Tal como acontece na abordagem Vulcan, a estimação do espaço não contabiliza o espaço necessário para guardar os parâmetros e as variáveis locais das funções. Por outro lado, a estimação do tempo de execução não considera as dependências de dados entre as partes de *hardware* e de *software* da implementação, nem as ramificações impostas por construtores condicionais.

Estimação incorporando a memória *cache*

A abordagem [LMW95] apresenta um método de estimação do tempo de execução em *software* que incorpora um modelo da memória *cache* para instruções do tipo atribuição directa. A influência da memória *cache* é um problema cuja solução exige uma análise de todo o código do sistema. Para que o problema de estimação do tempo de execução seja resolúvel não podem existir funções recursivas, ciclos de tamanho indefinido ou estruturas de dados definidas dinamicamente.

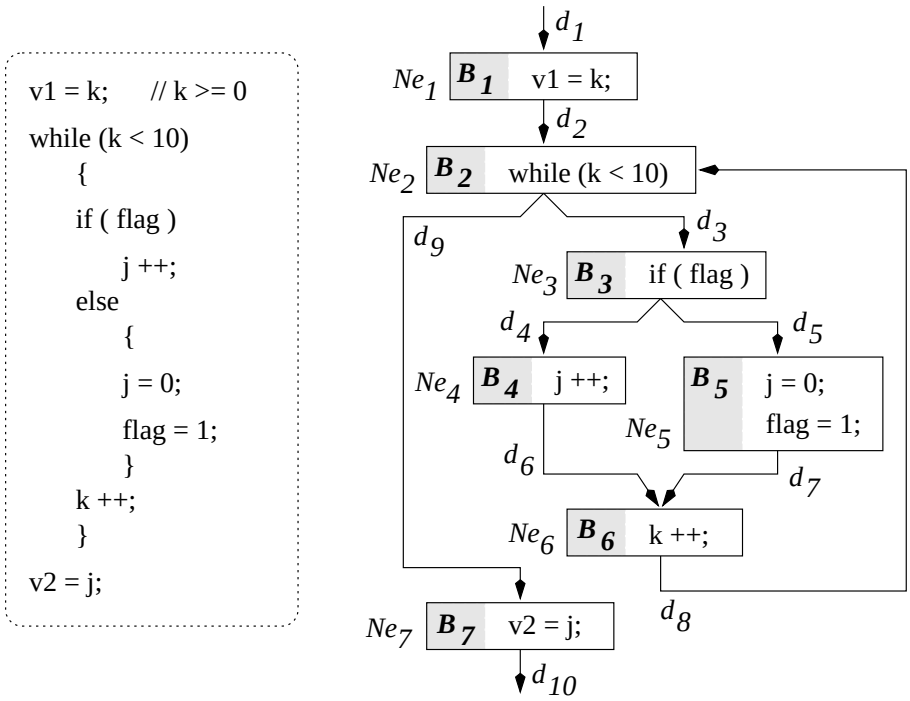
Considerando um modelo de processador em que o tempo de execução associado a cada tipo de instrução é constante, a estimativa do tempo de execução transforma-se numa tarefa de calcular o número de execuções de cada instrução. Se um programa estiver organizado em N blocos B_i, cada um com um tempo de execução $Texec_i$ e um número de execuções Ne_i, o tempo de execução do programa é calculado pela equação 4.29.

$$T_{exec} = \sum_{i=1}^{N} Ne_i * Texec_i \tag{4.29}$$

Os valores Ne_i são calculados a partir de condicionalismos estruturais (resultantes da estrutura do código) e de condicionalismos funcionais (definidos pelo utilizador). Traduzindo os condicionalismos em inequações, o problema de estimação do tempo de execução é resolúvel pelo método de programação linear com inteiros.

A partir da figura 4.5 ilustra-se a obtenção dos condicionalismos envolvidos no cálculo do tempo de execução. Cada nodo do CFG representa um bloco de código B_i e tem associado o número de vezes que o nodo é executado (Ne_i), e cada arco a_j está anotado com o número de vezes que o fluxo de controlo passa por ele (d_j). Para estabelecer as inequações que definem os condicionalismos estruturais empregam-se as seguintes regras: (i) o código é executado uma vez e (ii) o número de execuções dum bloco B_i coincide com a soma da anotação d_j de todos os arcos que chegam (partem) a esse nodo. No caso da figura 4.5, o resultado da aplicação destas regras é traduzido no conjunto de equações 4.30 (1-8). As equações anteriores são complementadas com duas inequações que traduzem os seguintes condicionalismos funcionais: (i) a variável k[17] é não negativa, logo o ciclo *while* é executado entre 0 e 10 vezes (equação 4.30 (9)) e (ii) a ramificação *else* é executada no máximo uma vez (equação 4.30 (10)). Resolvendo este sistema de inequações pelo método PLI, obtêm-se os valores Ne_i necessários para calcular o tempo de execução na pior situação (equação 4.29).

[17] Variável presente no fragmento de código da figura 4.5 (i).



(i) Fragmento de código *(ii) CFG*

Figura 4.5: Exemplo dum fragmento de código C e respectivo CFG, com o qual se ilustra a obtenção dos condicionalismos utilizados no cálculo do tempo de execução.

$$\begin{cases} d_1 = 1 & (1) \\ Ne_1 = d_1 = d_2 & (2) \\ Ne_2 = d_2 + d_8 = d_3 + d_9 & (3) \\ Ne_3 = d_3 = d_4 + d_5 & (4) \\ Ne_4 = d_4 = d_6 & (5) \\ Ne_5 = d_5 = d_7 & (6) \\ Ne_6 = d_6 + d_7 = d_8 & (7) \\ Ne_7 = d_9 = d_{10} & (8) \\ 0 * Ne_1 \leq Ne_3 \leq 10 * Ne_1 & (9) \\ Ne_5 \leq 1 * Ne_1 & (10) \end{cases} \tag{4.30}$$

Para considerar a influência da memória *cache* sobre o cálculo do tempo de execução, tem que se alterar a equação 4.29 de modo a incluir os condicionalismos relacionados com a memória *cache*. O tempo de execução de cada instrução deixa de ser constante e passa a ser definido à custa do número de execuções com *cache hit* e do número de execuções com *cache miss*. Para estimar o tempo de execução passa a usar-se como bloco básico de código, um ***bloco-linha***. Um bloco-linha coincide com uma sequência contígua de instruções que são atribuídas à mesma linha da memória *cache*. Ao considerar-se apenas a memória *cache* para instruções, existe um conflito entre dois blocos-linha quando eles são atribuídos à mesma linha da memória *cache* e a execução de um deles substitui o outro na memória *cache*. Com o objectivo de simplificar a estimação, assumiu-se que todas as instruções dum bloco-linha apresentam o mesmo número de execuções com *cache hit*, o mesmo número de execuções com *cache miss* e consequentemente

o mesmo número de execuções. Para cada bloco-linha $B_{i.j}$, de entre os n_i que pertencem ao mesmo bloco original B_i, o número de execuções com *cache hit* ($Ne_{i.j}^{hit}$) mais o número de execuções com *cache miss* ($Ne_{i.j}^{miss}$) é igual ao número de execuções de B_i (equação 4.31). O tempo de execução do programa passa a ser calculado pela equação 4.32, onde $Texec_{i.j}^{hit}$ ($Texec_{i.j}^{miss}$) representa o tempo de execução do bloco-linha $B_{i.j}$ com *cache hit* (*cache miss*).

$$Ne_i = Ne_{i.j}^{hit} + Ne_{i.j}^{miss} \quad , \quad 1 \leq j \leq n_i \tag{4.31}$$

$$T_{exec} = \sum_{i=1}^{N} \sum_{j=1}^{n_i} Ne_{i.j}^{hit} * Texec_{i.j}^{hit} + Ne_{i.j}^{miss} * Texec_{i.j}^{miss} \tag{4.32}$$

Os condicionalismos relacionados com a memória *cache* são aplicados no cálculo do número de execuções com *cache hit* e com *cache miss*. Quando a cada linha da memória *cache* só está atribuído um bloco-linha, apenas a primeira execução de cada bloco $B_{r.s}$ resulta em *cache miss* ($Ne_{r.s}^{miss} \leq 1$). Se os Nb blocos-linha atribuídos à mesma linha não apresentam conflitos entre si, a partir do momento em que um deles é executado, o código de todos os outros estará na memória *cache* quando eles forem executados ($Ne_{r.s}^{miss} + ... + Ne_{u.v}^{miss} \leq 1$)[18]. Por fim, quando existem conflitos entre os blocos-linha atribuídos à mesma linha, o número de execuções de cada bloco nas situações de *cache hit* e de *cache miss* depende da sequência pela qual os blocos são executados. No pior dos casos, o tempo de execução dum bloco-linha $B_{r.s}$ é afectado pelos blocos-linha atribuídos à mesma linha que ele, o que faz com que a influência entre blocos-linha possa ser resolvida linha-a-linha com um **grafo de conflitos na *cache*** (GCC).

Aplicando o método de programação linear com inteiros às equações que resultaram da resolução de conflitos na memória *cache* com o grafo GCC, obtém-se o valor do número de execuções com *cache hit* ($Ne_{i.j}^{hit}$) e com *cache miss* ($Ne_{i.j}^{miss}$) para os vários blocos-linha $B_{i,j}$. Dispondo do número de execuções dos blocos-linha, para estimar o tempo de execução, T_{exec} na equação 4.32, é preciso calcular o tempo de execução de cada bloco. O valor de $Texec_{i.j}^{hit}$ para o bloco-linha $B_{i.j}$ é aproximado pela soma do tempo de execução das diversas instruções que constituem $B_{i.j}$. O valor de $Texec_{i.j}^{miss}$ é aproximado por $Texec_{i.j}^{hit}$ mais a penalidade que se aplica à ocorrência duma falha na *cache*.

O resultados obtidos por [LMW95] mostram que as estimativas do tempo de execução, quando se considera a influência da memória *cache* para instruções, são mais próximas dos valores reais do que nos casos em que se exclui a influência da memória *cache*. O método de estimação proposto não considera a influência da memória *cache* para dados, nem memória *cache* para

[18] Esta soma contempla um termo por cada um dos Nb blocos-linha.

instruções do tipo *set-associative*. Por outro lado, em arquitecturas com memória *cache* de dimensão razoável e com sistemas complexos, o tempo de computação dispara dado o número de linhas para o qual é necessário resolver conflitos entre blocos de código.

4.6 Abordagens para a Comunicação

Nesta secção introduz-se a estratégia de estimação do tempo de comunicação entre *hardware* e *software* seguida pelas abordagens Cosyma e Lycos.

Estimação na abordagem Cosyma

Na abordagem Cosyma a comunicação entre *hardware* e *software* utiliza o mecanismo de memória partilhada [HE98]. A comunicação faz-se sob controlo do *software*, que em determinados momentos invoca procedimentos de *hardware*. Nestas alturas, o controlo e os dados são passados ao *hardware*, que após terminar a execução da tarefa requisitada devolve o controlo e os dados ao *software*. O modelo de comunicação é ilustrado na figura 4.6.

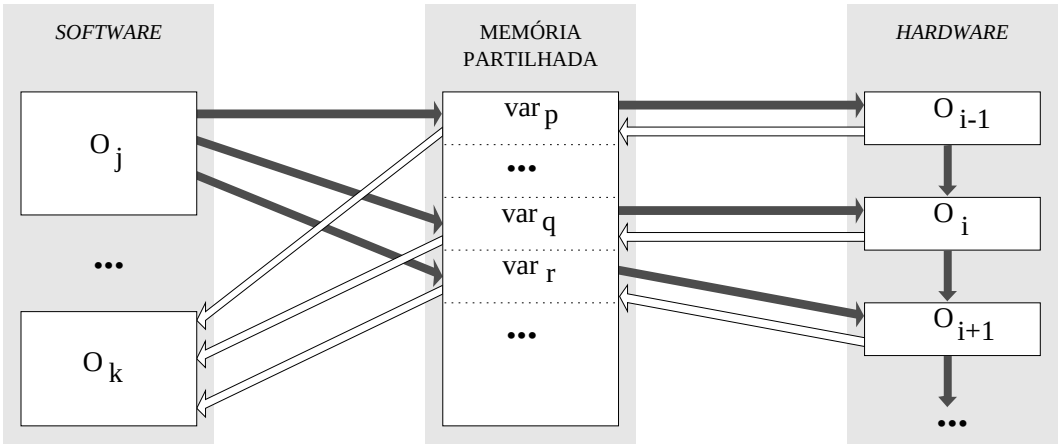


Figura 4.6: Modelo de comunicação entre *hardware* e *software* usado na abordagem Cosyma.

Na estimação considera-se apenas a comunicação de dados, desprezando-se a transferência do controlo. Deste modo, para a estimativa do tempo de comunicação de *software* para *hardware* contribuem a transferência de dados do *software* para a memória partilhada e a transferência de dados da memória partilhada para o *hardware*. Analogamente, para a estimativa do tempo de comunicação de *hardware* para *software* contribuem a transferência de dados do *hardware* para a memória partilhada e a transferência de dados da memória partilhada para o *software*. Para reduzir o excesso de comunicação é permitido que a informação produzida por um objecto de *hardware* seja utilizada por outro objecto de *hardware*, sem intervenção do *software*. Por questões de simplicidade da estimação, restringe-se a comunicação directa entre um objecto o_i de *hardware* e os objectos de *hardware* executados antes (o_{i-1}) e depois (o_{i+1}) desse objecto. A

figura 4.7 resume o algoritmo que estima a comunicação entre *hardware* e *software*, aplicando os princípios enunciados. O algoritmo apenas calcula o número de transferências num e noutro sentido da comunicação *hardware-software* ($Transf_{HWSW}$). Para obter o tempo de comunicação multiplica-se este número pela duração de uma transferência, cometendo-se a imprecisão de considerar que as transferências num e noutro sentido demoram o mesmo tempo. No algoritmo da figura 4.7, $B_{ant}(o_i)$ designa o conjunto de objectos de *hardware* e de *software* executados antes do objecto o_i, $B_{suc}(o_i)$ designa o conjunto de objectos de *hardware* e de *software* executados depois de o_i, $prod(o_i)$ é o conjunto de variáveis definidas em o_i e $usado(o_i)$ é o conjunto de variáveis utilizadas em o_i.

O método de estimação definido apresenta um bom compromisso entre a qualidade das estimativas e o tempo de cálculo.

calcComunicacao $(O) \equiv$

// O é o conjunto de objectos do sistema

$Transf_{HWSW} = 0$
para (*cada invocação dum objecto de hardware* $o_i \in O$) fazer

// Calcular o número de transferências de *software* para a memória partilhada
$Transf_{SW \rightarrow MEM}(o_i) = \mid prod(B_{ant}(o_i)) \cap usado(o_i) \mid$

// Reduzir a comunicação devido ao objecto de *hardware* anterior (o_{i-1})
se (o_{i-1} *existe em hardware*) então
$Transf_{SW \rightarrow MEM}(o_i) = Transf_{SW \rightarrow MEM}(o_i) - \mid prod(o_{i-1}) \cap usado(o_i) \mid$
fse

// Calcular o número de transferências de *hardware* para a memória partilhada
$Transf_{HW \rightarrow MEM}(o_i) = \mid prod(o_i) \cap usado(B_{suc}(o_i)) \mid$

// Reduzir a comunicação devido ao objecto de *hardware* posterior (o_{i+1})
se (o_{i+1} *existe em hardware*) então
$Transf_{HW \rightarrow MEM}(o_i) = Transf_{HW \rightarrow MEM}(o_i) - \mid prod(o_i) \cap usado(o_{i+1}) \mid$
fse

$Transf_{HWSW} = Transf_{HWSW} + Transf_{SW \rightarrow MEM}(o_i) + Transf_{HW \rightarrow MEM}(o_i)$
fpara

devolver ($Transf_{HWSW}$)

Figura 4.7: Algoritmo utilizado pela abordagem Cosyma para estimar a comunicação entre *hardware* e *software*.

Estimação na abordagem Lycos

A abordagem Lycos aplica um modelo genérico para estimar a comunicação entre *hardware* e *software*, concretizado numa biblioteca de comunicação [KM98]. Para um conjunto alargado

de unidades de processamento[19] e de protocolos de comunicação, a biblioteca possui valores para as métricas de desempenho, de espaço e de custo associadas à comunicação. O modelo de comunicação definido é válido para uma ligação ponto a ponto e inclui como elementos: o *software*, o *driver* de *software*, o canal de comunicação, o *driver* de *hardware* e o *hardware*.

Para estimar o atraso despendido por um *driver* numa transmissão (equação 4.33), é preciso conhecer o número de palavras a enviar (n_t), a frequência de relógio do processador responsável pela transmissão (f_t), o número de ciclos necessários à invocação do *driver* para transmissão (C_{tc}) e o número de ciclos necessários para enviar uma palavra (C_{tp}).

$$t_{td} = (C_{tc} + n_t * C_{tp})/f_t \tag{4.33}$$

O atraso envolvido na propagação através dum canal (equação 4.34) é estimado com base no número de palavras enviadas pelo canal (n_c), no número de ciclos despendidos em sincronização (C_{cs}), na frequência de relógio do canal de comunicação (f_c) e no número de ciclos necessários para transmitir uma palavra (C_{ct}).

$$t_{cd} = (C_{cs} + n_c * C_{ct})/f_c \tag{4.34}$$

Muitas vezes o *driver* responsável pela transmissão tem que juntar ou dividir as palavras a enviar, de modo a rentabilizar a largura do canal de comunicação. Ou seja, quando se pretende enviar n_t palavras com largura w_t elas são enviadas em n_c palavras com largura w_c, mas antes de juntar/dividir as palavras elas são fragmentadas em palavras com largura w_g. Conhecendo os parâmetros w_t e w_g usados na transmissão, a frequência de relógio do processador responsável pela recepção (f_r), o número de ciclos necessários à invocação do *driver* de recepção (C_{rc}) e o número de ciclos despendidos pelo *driver* de recepção por cada palavra que o *driver* de transmissão recebeu para enviar[20] (C_{rp}), o atraso despendido por um *driver* numa recepção é estimado pela equação 4.35.

$$t_{rd} = (C_{rc} + n_t * C_{rp})/f_r \tag{4.35}$$

Considerando que o envio, propagação e recepção funcionam em *pipeline*, e desprezando a diferença entre o número de palavras a enviar e o número de palavras realmente transmitidas pelo canal, o tempo total envolvido na transmissão de n_t palavras é aproximado pelo maior valor entre t_{td}, t_{cd} e t_{rd}.

[19] A unidade de processamento pode ser um microprocessador, um coprocessador, um ASIC ou um componente de lógica programável. Daqui para a frente, uma unidade de processamento será designada apenas por processador.

[20] Estas palavras passam a ser designadas por palavras originais a enviar.

A biblioteca de comunicação também suporta o modo *burst*, em que as palavras a enviar são transmitidas em blocos com determinado número de palavras. Neste caso, para além do tempo associado à transferência de palavras, é necessário considerar o tempo envolvido na sincronização dos blocos e o tempo envolvido na sincronização duma sucessão de blocos.

Se a largura w_t das n_t palavras originais a enviar for menor que a largura w_c do canal de comunicação, é possível juntá-las em n_{cd} ($n_{cd} \leq n_t$) palavras a transmitir pelo canal. A tarefa de **junção de palavras** originais em palavras a transmitir, é executada em duas fases: na primeira fase dividem-se as palavras originais de modo a gerar fragmentos de tamanho w_g[21] e na segunda fase junta-se o maior número possível de fragmentos em palavras do tamanho do canal de comunicação (w_c) (figura 4.8). A junção de palavras pode ter como consequência um desperdício de largura do canal de comunicação. Para utilizar de forma óptima a largura do canal deve seleccionar-se $w_g = 1$ durante a junção de palavras (figura 4.8 (i)).

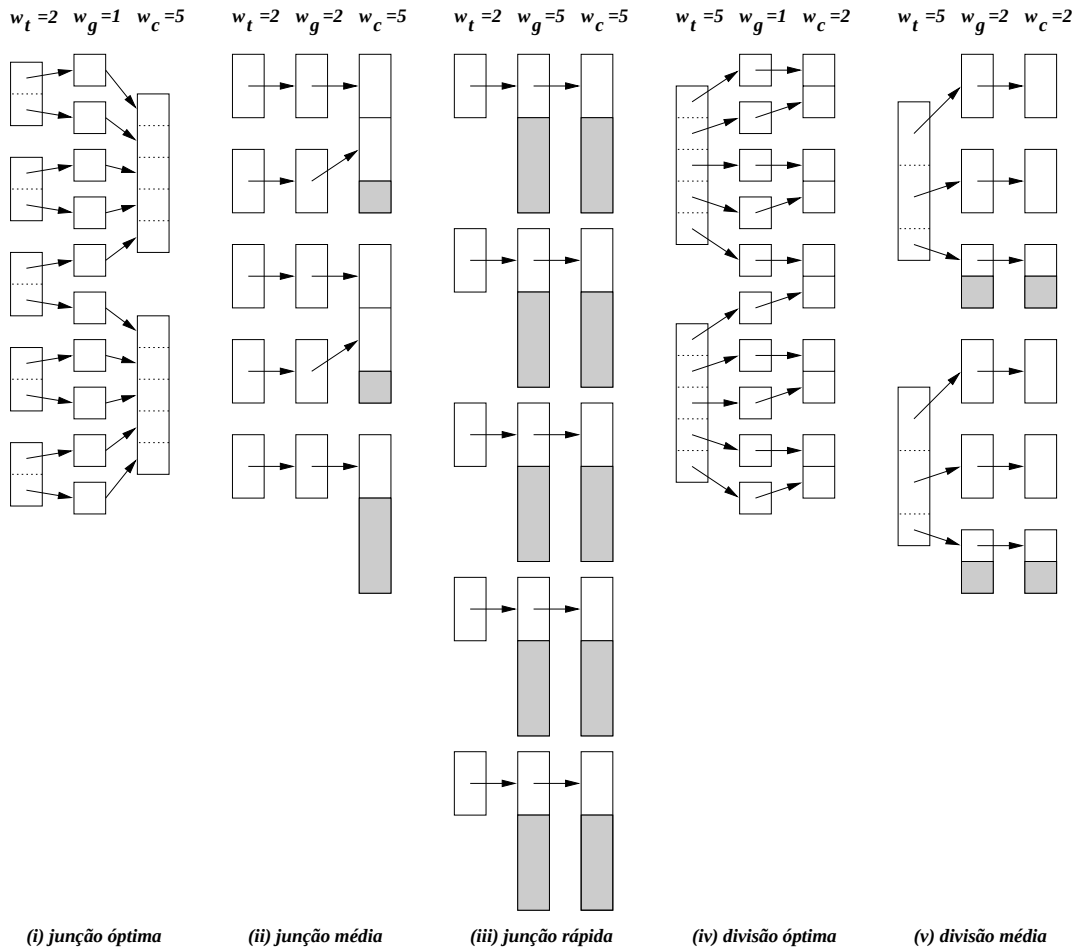


Figura 4.8: Junção e divisão de palavras para serem enviadas por um canal de comunicação.

Contrariamente à situação anterior, se a largura w_t das palavras originais a enviar for maior que a largura w_c do canal de comunicação, é preciso dividi-las em palavras do tamanho do

[21] Se $w_g > w_t$ cada fragmento fica apenas com uma palavra.

canal. A **divisão de palavras** é idêntica ao processo de junção, razão pela qual a divisão óptima também ocorre com $w_g = 1$ (figura 4.8 (iv)).

O modelo apresentado prova, por exemplo, que um processador rápido mas que demore muito tempo na junção das palavras a enviar para o canal de comunicação pode ser substituído, com vantagem, por um processador mais lento mas que demore menos tempo na junção das palavras a transmitir.

4.7 Resumo e Conclusões

Verificou-se que as métricas mais frequentes no processo de partição são o espaço ocupado em *hardware*, o espaço que o código e os dados ocupam em memória e o tempo de execução em *hardware* e em *software*, no qual se inclui o tempo de comunicação entre componentes. Para analisar milhares de alternativas de partição, é conveniente dispor de estimadores rápidos. Como as estimativas das métricas servem para comparar as alternativas de partição, pode usar-se estimativas menos precisas e mais rápidas, desde que apresentem um grau de fidelidade elevado.

De acordo com o modelo de estimação UC/CD comummente utilizado, o espaço do caminho de dados dum componente inclui elementos de armazenamento, unidades funcionais e elementos de interligação. Por seu lado, o espaço da unidade de controlo considera o registo de estado, a lógica que gera o próximo estado e a lógica de controlo.

A estimação dos elementos de armazenamento e de interligação necessários a uma implementação de *hardware* pode ser resolvida com o método de partição clique. A selecção de unidades funcionais, efectuada com o objectivo de obter uma estimativa para este tipo de recurso, decorre num de três cenários: (i) é efectuada manualmente com uma estratégia *ad-hoc*, (ii) é condicionada por um escalonamento que visa minimizar os recursos seleccionados ou (iii) condiciona o escalonamento de modo a atingir um desempenho pré-definido.

O espaço ocupado pelo registo de estado é proporcional ao número de etapas de controlo. Considerando a implementação da unidade de controlo com uma estrutura do tipo $AND{-}OR$, para estimar o espaço ocupado pela lógica que gera o próximo estado e pela lógica de controlo determina-se o número de portas lógicas OR, AND e NOT envolvidas nessa implementação.

Para estimar as métricas de *software* aplicam-se modelos genéricos ou modelos específicos dum processador. Aplica-se um modelo genérico quando não se exigem estimativas de grande qualidade. Um modelo específico complexo, exigido apenas para a estimação de tempos de execução, considera além da temporização e do espaço ocupado pelas instruções, as optimizações relacionadas com a utilização de registos, a implementação de ciclos, a utilização de estágios de *pipeline* ou a hierarquia de memória. Uma forma simples de contabilizar no

tempo de execução as optimizações de código, consiste em aplicar um factor de optimização do compilador. Contudo, este método é nitidamente limitado quando aplicado a processadores com uma arquitectura evoluída. Seguindo um modelo genérico, para estimar o espaço que o código ocupa em memória compila-se o código para instruções genéricas e depois recorre-se a um ficheiro tecnológico que traduz as instruções genéricas para instruções do processador pretendido para a implementação.

Com o objectivo de reduzir o tempo de cálculo necessário à estimação do espaço ocupado em *hardware*, a abordagem SpecSyn aplica um método de estimação incremental. Na abordagem Cosyma, a diminuição do tempo necessário para calcular o tempo de execução, é conseguida através duma divisão do grafo do sistema em sub-grafos a escalonar em separado.

Na abordagem [CLL+96], o cálculo do tempo de execução do sistema apoia-se numa biblioteca de componentes parametrizados e no escalonamento dos DFGs que modelam o sistema com duas selecções de componentes distintas, determinando assim um tempo mínimo e um tempo máximo. A estimação do tempo de execução dos DFGs apresenta algumas limitações: (i) gera estimativas pouco precisas, ao representar a influência das dependências de dados através duma simples constante e (ii) compromete o grau de automatização do processo de partição, ao exigir que o projectista calcule parâmetros do modelo de estimação.

Na abordagem Vulcan, a estimativa do tempo de execução em *software* considera o tempo de execução de instruções (operando sobre registos), o tempo gasto em acessos à memória e um atraso constante introduzido pelo ambiente de suporte à execução. A estimação do espaço ocupado em *software* considera apenas o espaço ocupado pelo código e pelos dados declarados estaticamente; desprezar o espaço necessário à utilização de rotinas foi uma decisão correcta, uma vez que a implementação dos sistema é feita com um processador que dispõe de registos capazes de guardar toda a informação envolvida na invocação de rotinas.

A abordagem Polis estima o tempo de execução e o espaço ocupado pelo código em *software* em dois níveis de abstracção. Para estimar as métricas relativas aos módulos da descrição do sistema, a abordagem COOL emprega a informação resultante da síntese de *hardware* e de *software*. Embora a síntese seja um método de estimação que gera valores precisos, é um método lento e difícil de automatizar, logo não é recomendável para um processo de partição iterativo em que se analisam milhares de alternativas de partição.

O método de estimação do tempo de execução em *software* descrito em [LMW95] considera a influência duma memória *cache* para instruções. Deste modo, o tempo de execução dos blocos que descrevem o sistema deixa de ser constante para passar a depender do número de execuções com *cache hit* e com *cache miss*. Tendo o sistema organizado em blocos-linha, para estimar o número de execuções de cada bloco, com *cache hit* e com *cache miss*, o método

procura resolver, linha-a-linha, os conflitos que possam existir entre os blocos atribuídos à mesma linha da *cache*. Para esse efeito, os conflitos de cada linha, representados num grafo, são resolvidos por um método PLI. A precisão das estimativas obtidas para o tempo de execução é melhor do que nos casos em que se exclui a influência da memória *cache*. Contudo, o método proposto não considera a influência da memória *cache* para dados, nem memória *cache* para instruções do tipo *set-associative*.

A estimação do tempo de comunicação entre *hardware* e *software* utilizada na abordagem Cosyma emprega o mecanismo de memória partilhada, considera apenas a transferência de dados e assume que as transacções em ambos os sentidos (enviar e receber) demoram o mesmo tempo. O método de estimação apresenta um bom compromisso entre a qualidade das estimativas e o tempo de cálculo. De acordo com o modelo de comunicação usado na abordagem Lycos, válido para uma ligação ponto a ponto, o tempo gasto em comunicação inclui o tempo de transmissão, o tempo de propagação através do canal de comunicação e o tempo de recepção da informação. Para rentabilizar a largura do canal de comunicação, o modelo de estimação contempla a hipótese de juntar ou dividir as palavras a enviar. Este modelo de estimação está orientado para protocolos de comunicação complexos.

Em conclusão, pode dizer-se que a generalidade das abordagens opera sobre uma representação do sistema em forma de grafo, a maioria assume um modelo de *hardware* com um caminho de dados e uma unidade de controlo e um modelo de *software* que exclui as optimizações de código resultantes dos estágios de *pipeline*, da superescalaridade do processador e da hierarquia de memória. Este procedimento é aceitável para a maioria dos problemas de partição que operam sobre sistemas embebidos. Constata-se que uma boa parte das abordagens não estima, ou não realça, métricas associadas à comunicação. Para estimar o tempo de execução é recorrente percorrer-se o grafo do sistema à procura do caminho com o maior tempo de execução acumulado. A precisão das estimativas para o tempo de execução e para o espaço situa-se no intervalo 75-90%.

Parte II

Trabalho Desenvolvido

Capítulo 5

A Arquitectura Alvo Reconfigurável

Sumário

Este capítulo descreve a plataforma EDgAR-2, uma parte fundamental da arquitectura alvo considerada na metodologia de partição proposta. A arquitectura da plataforma é apresentada, duma forma mais conceptual do que tecnológica, através dos meta-modelos de computação e de comunicação suportados. Quantifica-se o tempo de reconfiguração dos componentes e identificam-se os condicionalismos impostos pela arquitectura às tarefas de partição e síntese.

Conteúdo

5.1 Arquitectura

EDgAR-2 é um sistema, ou plataforma, baseado em FPGAs/CPLDs vocacionado para o co-projecto de *hardware* e de *software* e para a prototipagem rápida de sistemas digitais [Est98] [MFES00]. EDgAR-2 é o sucessor da plataforma EDgAR. Ambos os sistemas foram desenvolvidos no Departamento de Informática da Universidade do Minho. A plataforma EDgAR surgiu como uma ferramenta autónoma para emulação de sistemas digitais [EFP97]. O EDgAR-2 apresenta uma arquitectura melhorada, que inclui componentes mais actualizados e poderosos, é totalmente programável no circuito[1] e dispõe duma interface para ligar a um barramento PCI, o que permite que seja usado como um módulo de *hardware* reconfigurável

[1] ***In System Programmable (ISP)***, na terminologia inglesa.

dum sistema hospedeiro (PC), implementar uma máquina de co-projecto ou funcionar como uma ferramenta de prototipagem de sistemas digitais versátil.

Os componentes de lógica programável[2] podem ser divididos em duas grandes categorias: uma possui células de lógica adequadas para lógica a dois níveis e que constituem uma estrutura com um grão grosso (CPLDs), garante um atraso máximo em qualquer tipo de lógica configurada e é utilizada normalmente em módulos de controlo ou em circuitos com requisitos temporais exigentes; a outra baseia-se em células de lógica adequadas para lógica multi-nível e que constituem uma estrutura com um grão fino (FPGAs), aplica-se maioritariamente em módulos que constituem caminhos de dados ou em circuitos que exijam um espaço para armazenamento de informação elevado, mas permite implementar lógica com atrasos menos previsíveis [San96]. Dado que cada categoria de PLD é adequada para implementar partes complementares dum sistema digital típico, o EDgAR-2 foi projectado por forma a incluir componentes de ambas as categorias. As FPGAs facilitam a inclusão de estágios de *pipeline* num caminho de dados. Em relação à parte de controlo, a questão da adição de estágios de *pipeline* normalmente não se coloca, uma vez que se pretende que a lógica a dois níveis dos CPLDs opere a uma velocidade elevada.

O núcleo da arquitectura (figura 5.1) é um módulo composto por um conjunto de quatro **módulos de processamento** (MPs), onde cada um inclui uma unidade de controlo e uma unidade para caminhos de dados. Os MPs estão interligados duma forma linear através de "barramentos" dedicados, formando uma cadeia de MPs. Ambos os extremos da cadeia são disponibilizados para permitir interligar várias plataformas EDgAR-2, formando um conjunto ou cadeia de maior dimensão. Cada MP é implementado com uma FPGA Xilinx 4013XL - caminho de dados - e um CPLD Xilinx 95108 - caminho de controlo [Xil98]. Relativamente ao sistema hospedeiro (PC), cada MP está ligado a um byte distinto dos quatro presentes no barramento de dados PCI constituído por 32 bits. Deste modo, a componente de *software* numa implementação obtida por co-projecto pode aceder aos quatro MPs durante o mesmo ciclo do barramento. O barramento PCI também é utilizado para (re)programar as FPGAs.

As principais características do EDgAR-2 são: (1) é totalmente programável no circuito, (2) possui uma geração de sinais de relógio flexível (relativamente à sua frequência e origem), resultando num suporte limitado para problemas assíncronos, (3) suporta os mecanismos de sondagem e de interrupção para comunicar com o sistema hospedeiro, (4) permite elevadas taxas de transferência de informação através do modo *burst* do PCI [SA95], (5) possui uma estrutura do tipo cadeia (ou *pipeline*) e (6) uma arquitectura escalável.

[2] ***P**rogrammable **L**ogic **D**evices (PLDs)*, na terminologia inglesa.

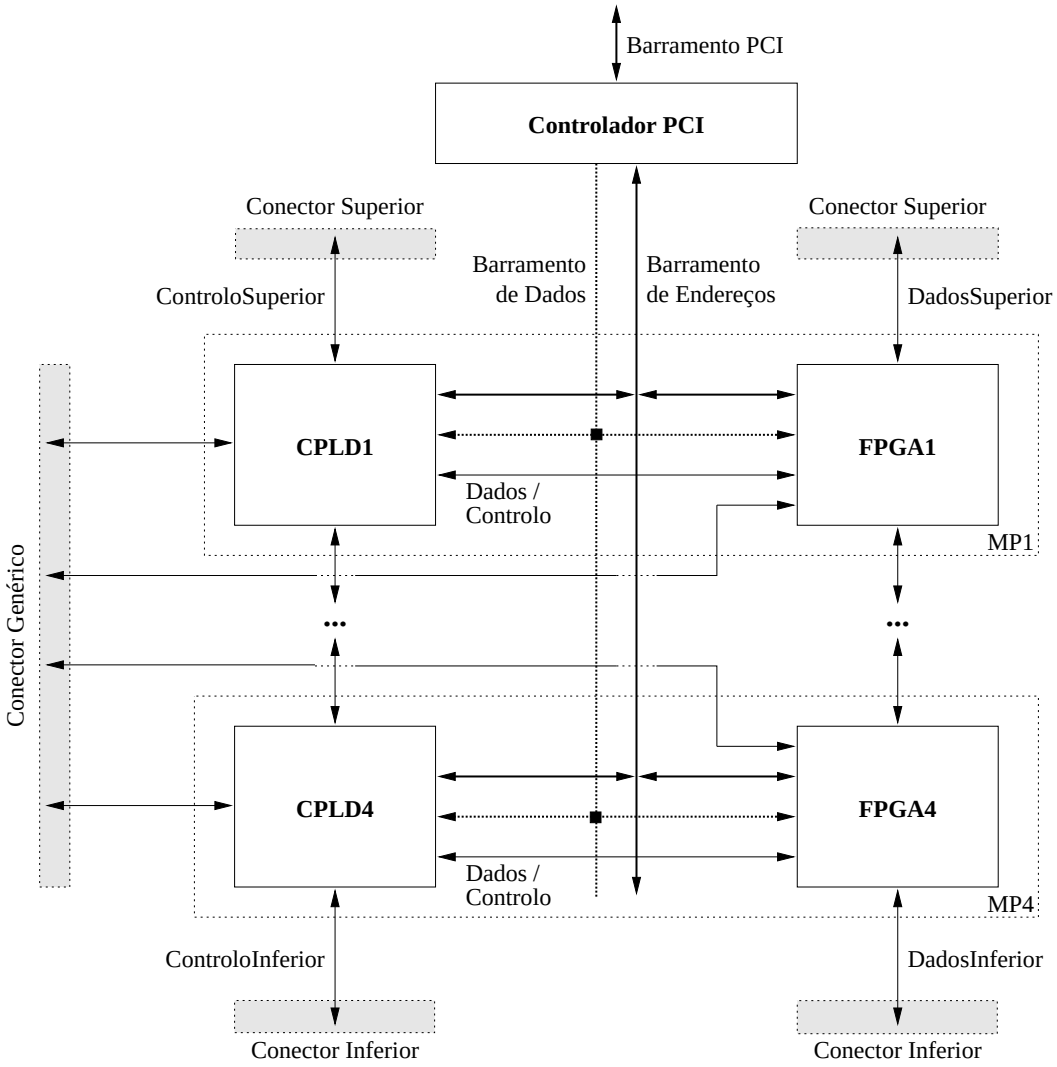


Figura 5.1: A arquitectura da plataforma EDgAR-2.

5.2 Meta-modelo de Computação

A arquitectura do EDgAR-2 foi projectada para implementar de forma directa o meta-modelo das *máquinas de estados finitos com caminho de dados* (FSMD) [GDWL92]. Este meta-modelo é essencialmente uma extensão do bem conhecido meta-modelo das *máquinas de estados finitos* (FSM) com um caminho de dados para permitir uma abstracção de dados de nível superior, incluindo primitivas para representar objectos do tipo variável bem como os operadores lógicos e aritméticos associados. Do ponto de vista estrutural, e tendo em atenção as respectivas propriedades, o meta-modelo FSMD é composto por 2 componentes: uma FSM de controlo e um caminho de dados. O par (FSM de controlo, caminho de dados) pode ser designado por **elemento de processamento** (EP) e é atribuído de forma directa a um MP do EDgAR-2. Uma vez que a arquitectura pode albergar várias FSMDs, quer ao nível do MP - conjunto de EPs sem restrições de interligação - quer ao nível da plataforma - conjunto unidi-

mensional de EPs, o meta-modelo representa a concorrência duma forma natural, passando a designar-se por *máquinas de estados finitos concorrentes e com caminho de dados* (CFSMD). Mais ainda, se o ambiente de desenvolvimento suportar hierarquia, é possível funcionar no nível arquitectural [GK83] usando para isso meta-modelos como *máquinas de estados finitos, concorrentes, hierárquicas e com caminho de dados* (HCFSMD) ou até máquinas com estados programa (PSM) [GMZ97]. Estes dois últimos meta-modelos podem também ser utilizados em modelação no nível do sistema. Como os dispositivos do tipo PLD possuem recursos limitados e as ferramentas de síntese de alto nível não sintetizam de forma eficiente determinados construtores permitidos em meta-modelos usados em níveis de abstracção elevados, a utilização dos meta-modelos HCFSMD ou PSM exige alguns cuidados nas fases de partição e síntese de sistemas com eles descritos. A questão relacionada com o nível de abstracção e os recursos consumidos é essencialmente a mesma que se verifica no domínio do *software* entre a utilização de linguagens de alto nível vs. a utilização de linguagem *assembly*. Deste ponto de vista, o EDgAR-2 ao possuir recursos lógicos limitados pode ser comparado com um computador com poucos Kbytes de memória.

Como foi afirmado anteriormente, o meta-modelo computacional da arquitectura também suporta *pipeline*. No nível exterior, o EDgAR-2 pode ser definido como uma cadeia formada por grupos de EPs, em que cada grupo é constituído por EPs funcionando de forma concorrente. Esta arquitectura é adequada, por exemplo, para uma linha de produção com várias máquinas interdependentes, cada uma com um conjunto de processos concorrentes. No nível dos EPs, o *pipeline* não surge explicitamente na arquitectura, mas as FPGAs facilitam a introdução de estágios de *pipeline* no caminho de dados. Relativamente ao caminho de controlo, normalmente não advêm vantagens da introdução de estágios de *pipeline*, porque com a lógica a dois níveis dos CPLDs consegue operar-se com frequências bastante elevadas.

Em resumo, a arquitectura dum EP da plataforma EDgAR-2 é composta por (1) um bloco de controlo baseado em FSMs (CPLDs), adequado para registos e lógica a dois níveis; este bloco não inclui memória suficiente para ser microprogramado; a opção de microprogramação permitiria o meta-modelo de microprocessador, o qual não é necessário porque o sistema hospedeiro contém um processador; (2) um bloco para caminho de dados (FPGAs), capaz de implementar operadores lógicos e aritméticos com operandos com um número de bits variável (podendo ir perfeitamente até 32 bits), estruturas de dados de baixa complexidade (até 4 Kbytes) e as estruturas lógicas mais comuns (multiplexadores, descodificadores, contadores, registos, ALUs até 32 bits). Um sistema é composto por um número arbitrário de EPs concorrentes, formando grupos, eventualmente ligados segundo uma estrutura do tipo cadeia.

5.3 Meta-modelo de Comunicação

A comunicação é um aspecto chave em arquitecturas heterogéneas, onde diversos tipos de *hardware* e de *software* interagem. Deste modo, é conveniente definir quais os mecanismos de comunicação suportados pela plataforma EDgAR-2.

Num nível de abstracção alto podem identificar-se duas classes de mecanismo de comunicação: passagem de mensagens e memória partilhada. Memória partilhada significa um acesso coordenado por parte de todos os intervenientes na comunicação a uma memória comum, exigindo-se por isso a definição de regras de arbitragem para resolver o problema resultante de acessos simultâneos à memória. O facto do EDgAR-2 não possuir uma zona de memória comum com dimensão elevada, impõe restrições à implementação do mecanismo de memória partilhada. Enquanto no mecanismo de memória partilhada os intervenientes partilham a informação trocada, no mecanismo de passagem de mensagens apenas partilham o canal de comunicação. O mecanismo de passagem de mensagens caracteriza-se pelo nível de abstracção das mensagens, o tipo de protocolo que o suporta (síncrono ou assíncrono) e as operações permitidas. Algumas das operações comuns no mecanismo de passagem de mensagens são: comunicação ponto-a-ponto (relação um para um), difusão (relação um para todos), dispersão (um interveniente envia uma mensagem distinta para todos os outros) e junção (um interveniente recebe uma mensagem distinta de todos os outros) [XH96].

Num nível de abstracção mais baixo, a topologia física de comunicação também impõe algumas restrições à implementação dos mecanismos de comunicação anteriores. Algumas das topologias comuns em sistemas idênticos ao EDgAR-2 são: (1) em **cadeia**, onde todos os nodos, excepto os nodos terminais, comunicam com dois nodos vizinhos e constroem uma cadeia, (2) em **anel**, uma topologia tipo cadeia em que os dois nodos terminais estão interligados, (3) **malha n-dimensional**, em que cada nodo comunica com os nodos adjacentes, formando um hipercubo n-dimensional, (4) **centralizada** ou em **estrela**, uma topologia onde um único nodo (concentrador) se liga com todos os outros, (5) **hierarquizada** ou em **árvore**, em que as ligações entre nodos formam uma estrutura com hierarquia ou em árvore, (6) **completa** ou totalmente ligada, onde cada nodo comunica com todos os outros e (7) uma topologia irregular.

Estas topologias podem ser definidas explicitamente na arquitectura da plataforma, implementadas em *software* ou implementadas com dispositivos programáveis, incluídos na arquitectura especificamente para esse fim (encaminhamento de ligações) e que permitem alterações na topologia física de comunicação.

No que diz respeito ao EDgAR-2, convém distinguir a comunicação entre processos em execução no *hardware*, no *software* e processos em que uma parte é executada no *hardware* e a

outra é executada no *software*. Os processos em execução no *hardware* devem obedecer ao meta-modelo de computação CFSMD proposto. Se este meta-modelo não for condicionado, exige uma topologia do tipo totalmente ligada no nível dos EPs. Quer os CPLDs quer as FPGAs possuem estruturas programáveis bastante ricas em termos de interligações dentro do componente, permitindo na maior parte das aplicações, uma topologia totalmente ligada no nível dos conjuntos de EPs. Para implementar a mesma topologia no nível dos MPs tem que se utilizar componentes maiores, que normalmente possuem mais pinos e por isso resultam em sistemas com produção mais cara e mais complexa. A complexidade adicional deriva essencialmente do facto de o encaminhamento a efectuar na(s) placa(s) de circuito impresso ser mais complexo.

A análise do meta-modelo CFSMD para vários sistemas embebidos, mostrou que apenas um número reduzido desses sistemas necessita de uma topologia totalmente ligada. A partir desta conclusão decidiu dotar-se a arquitectura do EDgAR-2 com uma topologia resultante da combinação de duas das topologias já mencionas: (i) uma topologia totalmente ligada ou centralizada, ao nível do conjunto de EPs[3] e (ii) uma topologia em cadeia, ou opcionalmente em anel, ao nível do MP ou do sistema.

Os processos em execução no *software* utilizam os mecanismos de comunicação permitidos pelo sistema operativo do hospedeiro. Estes mecanismos não influenciam o meta-modelo de comunicação do EDgAR-2, razão pela qual não serão detalhados aqui. Finalmente, num ambiente de co-projecto, existem processos em *hardware* a comunicar com processos em *software*. Como foi mencionado anteriormente, o hospedeiro pode aceder a qualquer dos MPs através do barramento PCI. Para limitar a complexidade da plataforma e porque ela funcionará maioritariamente como coprocessador, a interface com o barramento PCI não implementa a funcionalidade de iniciador de transferências sobre esse barramento. Esta decisão tem como consequência que os processos em *software* controlam os processos em *hardware*, o que corresponde a uma topologia de comunicação centralizada. A figura 5.2 ilustra o meta-modelo de comunicação agora descrito, válido para uma arquitectura alvo composta por uma ou mais plataformas EDgAR-2 e um sistema hospedeiro baseado em processadores.

Qualquer comunicação que envolva o EDgAR-2 pode utilizar um protocolo síncrono ou assíncrono. A aplicação dum protocolo assíncrono requer a selecção dum conjunto de sinais de protocolo e a definição da sua temporização. Para transferir o conteúdo dum registo para outro, uma situação de comunicação frequente, um protocolo assíncrono exige apenas um sinal de pedido e outro de aceitação da comunicação.

Usando a topologia de comunicação descrita, podem implementar-se meta-modelos de comunicação de nível superior. A decisão sobre que meta-modelo utilizar será relegada para a

[3]Esta topologia resulta inteiramente da arquitectura dos CPLDs e FPGAs.

metodologia de desenvolvimento. Pode no entanto, idealizar-se um protocolo de comunicação que gaste a quase totalidade dos recursos do EDgAR-2. Algumas directrizes relativas a esta decisão são apresentadas em seguida.

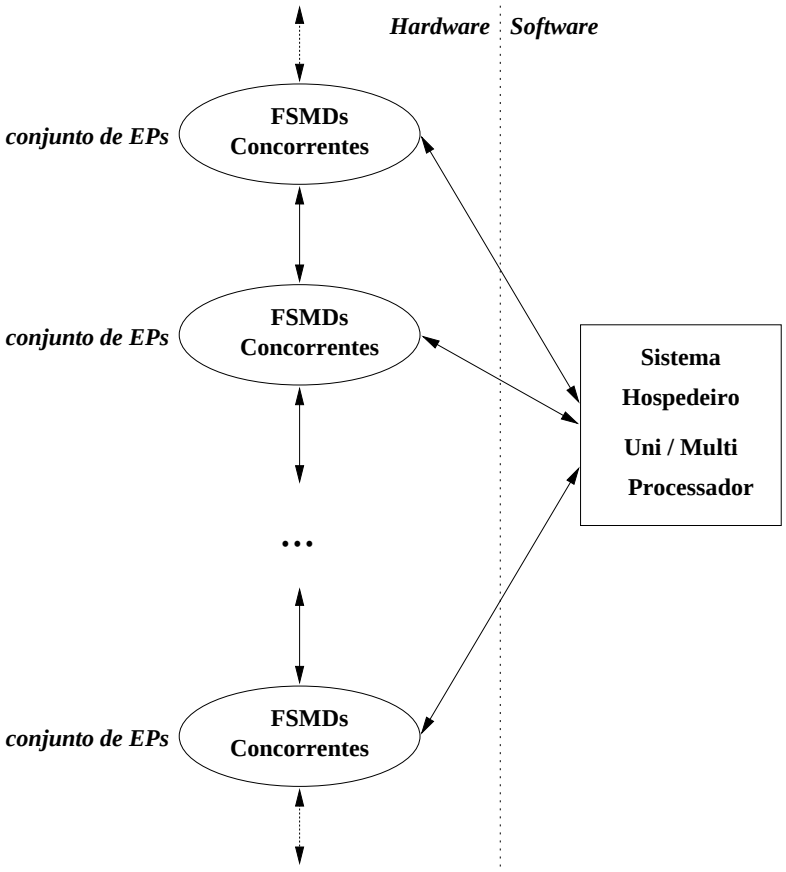


Figura 5.2: O meta-modelo de comunicação do EDgAR-2.

Primeiro, meta-modelos de comunicação baseados na utilização intensa de memória não são suportados devido à disponibilidade de pouco espaço de armazenamento na plataforma.

Segundo, a operação de difusão usada no meta-modelo de passagem de mensagens não é totalmente suportada em níveis diferentes do nível do conjunto de EPs. Aparentemente, a topologia não suporta a operação de difusão além do nível do conjunto de EPs, ou seja, a difusão só é possível dum processo para os processos a correr no mesmo dispositivo. Para que a operação de difusão fosse completamente suportada, a arquitectura teria que permitir a escrita simultânea da mesma informação de um para todos os outros dispositivos. Embora a escrita simultânea não seja possível, indicam-se dois casos em que consegue um efeito idêntico ao da operação de difusão: (i) ao escrever o mesmo valor em todos os 4 bytes do barramento PCI, quando o processo que envia a informação se encontra em *software*, obtendo-se uma difusão precisa; (ii) ao efectuar uma sequência de comunicações ponto-a-ponto, introduzindo-se assim um atraso sucessivamente crescente duma comunicação ponto-a-ponto para a seguinte.

Terceiro, é conveniente seleccionar protocolos de comunicação não muito complexos, por forma

a obter uma boa relação entre a quantidade de recursos usados na comunicação e a quantidade de recursos usados em computação.

A figura 5.3 contém um fotografia da plataforma EDgAR-2.

Figura 5.3: A plataforma EDgAR-2.

5.4 Reconfiguração dos Componentes

Quando se opera com sistemas dinamicamente reconfiguráveis é comum desejar-se que o tempo necessário para (re)configurar os componentes seja o mais baixo possível, mas em muitas outras situações não se impõe um requisito tão apertado ao tempo de configuração dos componentes.

Na plataforma apresentada, todos os componentes podem ser configurados no modo *JTAG boundary scan* e as FPGAs também podem ser configuradas através do barramento PCI no modo byte-a-byte assíncrono.

O tempo de reconfiguração da totalidade da plataforma, considerando que as FPGAs são configuradas através do barramento PCI, a funcionar a 33 MHz, é aproximadamente:

$$t_{reconfiguracao} = 4 * t_{configuracao-1-cpld} + 1 * t_{configuracao-1-fpga} \tag{5.1}$$

$$t_{reconfiguracao} = 10\ s + 18\ ms \tag{5.2}$$

$$t_{reconfiguracao} \approx 10\ s \tag{5.3}$$

O elevado tempo de reconfiguração dos CPLDs por JTAG advém do facto de se utilizar uma ferramenta comercial, que configura os componentes através duma ligação RS-232. Este tempo pode ser reduzido se for implementado um utilitário proprietário para configuração no modo *JTAG boundary scan*.

Na equação 5.1, o tempo de configuração dum CPLD é afectado pelo factor "4" porque os 4 CPLDs são configurados de forma sequencial no modo *JTAG boundary scan*, enquanto ao tempo de configuração duma FPGA é aplicado o factor "1" porque as FPGAs são configuradas em simultâneo através do barramento PCI. A configuração simultânea das 4 FPGAs é possível porque cada uma está ligada a um byte distinto dos 32 bits de dados do barramento PCI.

Se apenas for necessário reconfigurar as FPGAs, o tempo de reconfiguração baixa para um valor compatível com o paradigma da reconfiguração dinâmica:

$$t_{reconfiguracao} \approx 18\ ms \tag{5.4}$$

5.5 Condicionalismos da Arquitectura

Algumas das características da plataforma EDgAR-2 condicionam a fase de implementação da metodologia de desenvolvimento. Concretamente, as tarefas de partição e de síntese são as que mais dependem destes condicionalismos. Os condicionalismos podem ser classificados numa de duas categorias: os que resultam de limitações físicas dos componentes da plataforma e os que derivam de decisões arquitecturais tomadas durante o projecto da plataforma. O número de pinos disponíveis em cada componente, a localização física desses pinos, os recursos disponíveis em cada componente (comummente designados por área do componente e medidos em portas lógicas equivalentes) e o número de componentes pertencem à primeira categoria (tabela 5.1). A não disponibilidade de memória RAM na plataforma (a não ser a que se implementa com as FPGAs), a impossibilidade de se iniciar, sob controlo da plataforma, uma transferência de informação entre ela e o sistema hospedeiro (o mesmo é dizer, ausência de funcionalidade PCI *master*) e a topologia usada na interligação dos componentes, são os condicionalismos mais relevantes pertencentes à segunda categoria. Alguns destes condicionalismos derivam de decisões estratégicas que tinham como objectivo manter o custo e o tempo de desenvolvimento da plataforma em valores aceitáveis.

Tipo de componente	***Número de componentes***	***Recursos em portas lógicas equivalentes*** *(por componente)*	***Pinos definidos pelo utilizador*** *(por comp.)*	***Pinos com funcionalidade fixa*** *(por componente)*
CPLD	4	2.4 K	20	44
FPGA	4	10.9 K (lógica) 49.1 K (memória)	96	39

Tabela 5.1: Condicionalismos físicos da plataforma EDgAR-2.

5.6 Conclusões

A arquitectura da plataforma EDgAR-2 implementa de forma directa o meta-modelo CFSMD, uma vez que dispõe dum conjunto de pares FPGA-CPLD adequado para implementar os pares CD-UC obtidos a partir da descrição dos sistemas. Tal como a arquitectura pode ser estendida, também a quantidade de pares CD-UC disponível o pode ser.

O conjunto formado pelo sistema hospedeiro e a(s) plataforma(s) EDgAR-2 possui uma topologia de comunicação que resulta da combinação entre uma topologia centralizada (no hospedeiro) e uma topologia em cadeia/anel ao nível dos módulos de processamento (pares FPGA-CPLD). Com esta topologia e o mecanismo de interrupção é possível implementar completamente, embora com degradação de desempenho, as operações de comunicação ponto-a-ponto, difusão e dispersão. O mecanismo de comunicação por memória partilhada não é recomendável para a plataforma EDgAR-2, dado não existir memória comum acessível directamente por todos os componentes.

Considerando apenas as FPGAs, o tempo de reconfiguração da arquitectura é compatível com a reconfiguração dinâmica, uma temática não abordada no presente trabalho.

Os condicionalismos impostos à implementação de sistemas por uma arquitectura heterogénea e com vários componentes de *hardware* são mais importantes do que numa arquitectura com um único componente de *hardware*. Quando a arquitectura não é totalmente ligada, como acontece no EDgAR-2, a importância dos condicionalismos é ainda maior. No caso do EDgAR-2, os condicionalismos mais importantes são o número de componentes, o espaço disponível por componente, as ligações entre componentes, a ausência de memória (extra FPGAs) na plataforma e a impossibilidade de estabelecer certas ligações, entre componentes de *hardware*, sem intervenção do *software*.

À parte as limitações resultantes da evolução da tecnologia dos componentes reconfiguráveis, que se traduzem essencialmente na disponibilidade de componentes com muito mais lógica do que a contida nos componentes do EDgAR-2, os pontos fracos desta plataforma são: a inexistência de memória em quantidade, a ausência da funcionalidade PCI *master* e uma ligação com largura reduzida nos pares FPGA-CPLD. Os aspectos fortes da plataforma são: a boa correspondência entre o meta-modelo usado na descrição dos sistemas e o meta-modelo de computação da plataforma, a possibilidade de estender a arquitectura e as potencialidades que fornece para a exploração do problema de partição em múltiplos componentes diferenciados. Como se afirmou, uma boa parte das limitações da plataforma resultou de decisões que tinham como objectivo manter o seu custo e tempo de desenvolvimento em valores aceitáveis.

Capítulo 6

Metodologia de Partição Proposta

Sumário

Quatro dos módulos que compõem a metodologia de partição desenvolvida são apresentados neste capítulo: o algoritmo de partição construtivo, a função de proximidade, o algoritmo de partição iterativo e a função de custo. Antes que tudo introduz-se o meta-modelo PSMfg, o meta-modelo utilizado na representação dos sistemas durante o processo de partição. Depois descreve-se o processo de construção de soluções de partição com o algoritmo construtivo de crescimento de grupos, assim como a função de proximidade por ele utilizada e a estimação das métricas necessárias à função de proximidade. A procura de soluções de partição óptimas com o algoritmo iterativo de pesquisa tabu é apresentado em pormenor. Os fundamentos do método de optimização e a sua aplicação na presente metodologia são explicados, destacando-se os tipos de tabu e a lista tabu, o historial de soluções visitadas, a pesquisa na vizinhança ou subvizinhança, os critérios de aspiração, os mecanismos de convergência para soluções óptimas e de fuga a mínimos locais da função de custo. A função de custo utilizada pelo algoritmo de partição iterativo é definida a partir do conjunto de métricas, condicionalismos e requisitos que lhe estão associados.

Conteúdo

6.1 Introdução

As abordagens analisadas no capítulo 3 apresentam limitações que as tornam pouco atractivas para o tipo de sistemas que se pretende desenvolver: sistemas embebidos, predominantemente

de fluxo de dados, razoavelmente complexos e com requisitos de tempo real. Como principais limitações pode mencionar-se: (i) uma boa parte das abordagens está orientada para sistemas embebidos predominantemente de fluxo de controlo e para uma implementação com um único componente de *hardware* e outro de *software*; (ii) muitas abordagens não utilizam uma arquitectura alvo bem definida mas antes uma arquitectura conceptual, o que se traduz em estimativas menos precisas que penalizam as decisões nelas baseadas; (iii) exploram de forma incorrecta e/ou incompleta o espaço de projecto; (iv) atribuem pouca importância à comunicação entre partições. Existem, contudo, honrosas excepções a cada uma das limitações apontadas. Outra característica da presente abordagem, que a distingue das descritas na bibliografia, deriva do facto de ter que lidar com a heterogeneidade na arquitectura alvo: além da diversidade entre *hardware* e *software*, existe diversidade no próprio *hardware*.

Relativamente à classificação apresentada na secção 3.1, a abordagem ao problema de partição aqui apresentada é do tipo funcional, inter-componentes e automática.

Contrariamente a uma boa parte das abordagens, que visam obter uma solução óptima que ocupe o mínimo de recursos possíveis, no presente trabalho o objectivo do processo de partição é obter um desempenho pré-definido com os recursos disponíveis. Este facto reflecte-se na função de custo que controla a evolução do processo de partição. Para atingir o objectivo definido deve:

- identificar-se as partes do sistema responsáveis pela parcela mais significativa do tempo de execução e atribuí-las à partição em que apresentem o menor tempo de execução;
- minorar-se a comunicação entre partições, nomeadamente entre partições de *hardware* e de *software*;
- utilizar-se o máximo de paralelismo possível nas partições de *hardware*.

A organização da metodologia de partição desenvolvida baseia-se na estrutura descrita na secção 3.2, mais um módulo que efectua a conversão entre o modelo PSM e o modelo da representação interna ao processo de partição (figura 6.1).

Além do módulo de conversão de modelos, a metodologia inclui os algoritmos de partição construtivo e iterativo, as funções de proximidade e de custo e os estimadores de métricas. Os dois algoritmos de partição e as funções de proximidade e de custo são desenvolvidos em pormenor neste capítulo, enquanto a estimação de métricas será desenvolvida no capítulo 7.

O **módulo de conversão de modelos** tem por missão converter o modelo PSM que alimenta a partição para a representação interna ao processo de partição e converter a representação interna para o modelo PSM de cada componente da solução de partição. O meta-modelo

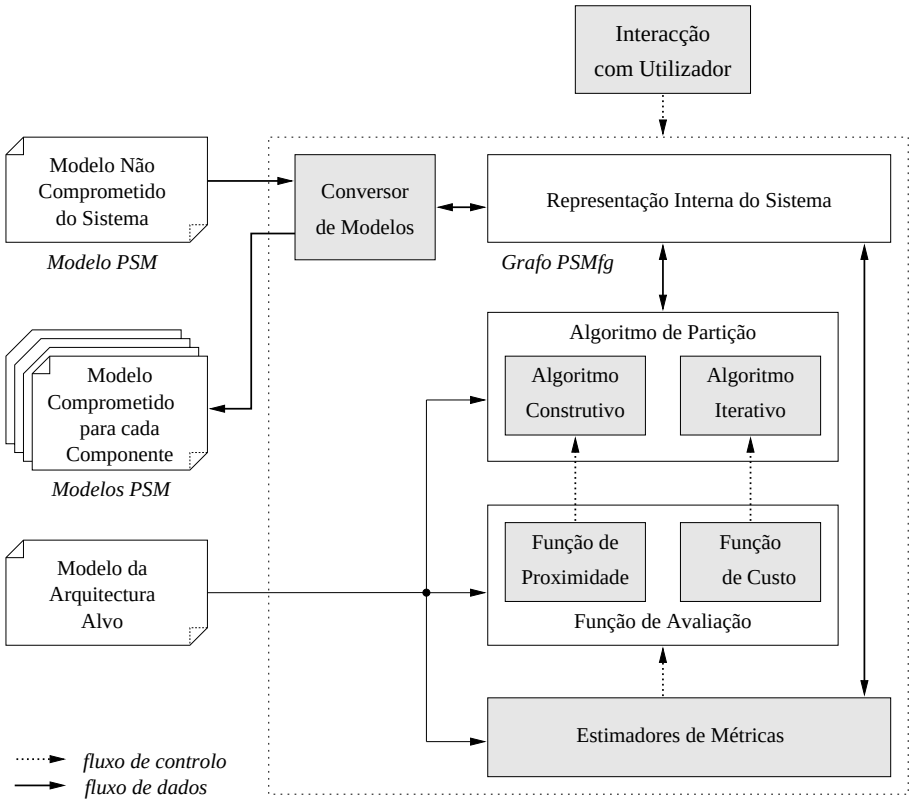


Figura 6.1: Os componentes da metodologia de partição proposta.

da representação interna é o **grafo PSMfg**, descrito na secção 6.2. A tarefa mais complexa é a conversão do modelo PSM para PSMfg, uma vez que a conversão de PSMfg para PSM reaproveita a maior parte da informação do modelo PSM que alimenta o processo de partição. A conversão de PSM para PSMfg exige (i) um *parser* de VHDL, uma vez que a funcionalidade dos estados programa do modelo PSM é definida em VHDL e (ii) uma ferramenta de *profiling*, que através da simulação, da análise estática ou da acção do projectista determina informação como a probabilidade de execução ou o número de execuções das operações presentes no modelo do sistema. O módulo de conversão de modelos não foi automatizado, encontrando-se em fase de desenvolvimento, razão pela qual não será aprofundado nesta tese.

A metodologia de partição recebe como entradas o modelo que descreve a funcionalidade do sistema e o modelo da arquitectura alvo e gera um modelo separado para cada componente da arquitectura alvo seleccionado. O modelo do sistema que entra no processo de partição não está comprometido com uma implementação, enquanto os modelos a gerar incluem detalhes que comprometem a implementação com uma arquitectura específica. Do ponto de vista da modelação de sistemas, a selecção de um meta-modelo é determinada essencialmente pela sua capacidade para representar as características desse tipo de sistema, mas no contexto da partição a selecção de um meta-modelo deve ter em atenção a possibilidade de manipular a granulosidade dos objectos durante o processo de partição. O PSM é uma boa opção para modelar sistemas embebidos e também para efectuar a sua partição em componentes,

conforme se justificou em 2.4. O modelo do sistema que entra no presente ambiente de partição descreve a funcionalidade do sistema em PSM e assume-se que foi gerado nas fases de análise e concepção. O modelo da arquitectura alvo descreve, de forma simplificada, os aspectos da arquitectura que são relevantes para o processo de partição. Entre os aspectos relevantes encontram-se o meta-modelo de computação, o meta-modelo de comunicação e os condicionalismos que resultam da implementação destes meta-modelos, os quais foram apresentados no capítulo 5.

O algoritmo de partição construtivo gera uma solução inicial que será melhorada por um algoritmo de partição iterativo. A construção da solução inicial é efectuada pelo algoritmo de crescimento de grupos.

O algoritmo de partição iterativo é orientado pelas estimativas de métricas de *hardware*, de *software* e de comunicação, que por sua vez recorrem à informação de *profiling* obtida pelo módulo de conversão de modelos. No processo de partição iterativo aplicou-se o algoritmo de pesquisa tabu.

6.2 Representação Interna ao Processo de Partição

Antes de apresentar os módulos funcionais da metodologia de partição, introduz-se o meta-modelo que representa internamente os sistemas durante o processo de partição. O meta-modelo escolhido é um grafo do tipo CFG, designado por grafo de fluxo para o meta-modelo PSM, ou simplesmente **PSMfg**.

O desenvolvimento dum meta-modelo novo resultou mais da necessidade de automatizar o processo de partição do que da necessidade dum meta-modelo com características novas. O requisito mais relevante que se exige à representação interna, e que não consta da lista de requisitos do meta-modelo PSM, é a capacidade de associar a informação gerada durante o processo de partição com os objectos do modelo do sistema. O incentivo à criação do meta-modelo PSMfg resultou da disponibilidade duma aplicação com interface gráfica que permite editar grafos. A aplicação faz parte da biblioteca LEDA[1] [MN99] [MNSU00], uma biblioteca que reúne uma colecção de algoritmos e estruturas de dados complexas, como é o caso dos grafos. Deste modo, mediante um conjunto de adaptações aplicadas ao editor de grafos direccionados genéricos e à estrutura de dados que lhe está associada, foi possível obter o suporte computacional para operar sobre os grafos PSMfg. As adaptações efectuadas tiveram por objectivo (i) personalizar o aspecto gráfico dos nodos, criando o conjunto de tipos de nodo que à frente se descreve, (ii) aumentar a funcionalidade dos nodos e arcos, de acordo com o tipo de nodo ou arco e (iii) introduzir limitações na interligação dos diferentes tipos de

[1] ***L**ibrary of **E**fficient **D**ata types and **A**lgorithms.*

nodo.

Utilizando a biblioteca LEDA, desenvolveu-se a ferramenta ***parTiTool*** que visualiza, edita e efectua a partição de grafos PSMfg. O apêndice D ilustra algumas das facilidades da ferramenta *parTiTool*.

O meta-modelo do PSMfg representa a semântica do meta-modelo PSM, o meta-modelo com que se descrevem os sistemas à entrada e à saída do processo de partição e toda a informação necessária ao processo de partição, como sejam as estimativas das métricas e a atribuição dos objectos às partições. Para se poder controlar a granulosidade dos objectos manipulados durante o processo de partição, o grafo PSMfg deve conseguir representar a estrutura dos estados programa. Como a funcionalidade dos estados programa é descrita em VHDL [Nav93] [IEE93], o grafo PSMfg suporta os seguintes construtores da linguagem: o paralelismo imposto pelos processos, os construtores condicionais (`if ... elsif` e `case`), os ciclos (`while` e `for`) e os construtores que suspendem os processos (`wait`).

Um modelo PSMfg é um grafo acíclico, direccionado e polar, representado por uma estrutura $G = \{V, E\}$ composta por uma lista de nodos V e uma lista de arcos E. O grafo é acíclico porque nem todos os caminhos dos grafo constituem um percurso fechado, é direccionado porque cada arco possui uma e só uma direcção e é polar porque existem dois nodos, um para entrada e outro para saída do grafo, dos quais todos os outros nodos são sucessores e antecessores, respectivamente [EKP+98b]. Os nodos do grafo representam os estados programa, com a granulosidade do modelo PSM ou uma granulosidade mais fina, e as variáveis do modelo PSM. Os arcos representam o fluxo de controlo entre nodos.

Um grafo PSMfg tem associada a informação relevante para o processo de partição, nomeadamente:

- a partição a que os nodos do grafo foram atribuídos;
- a indicação sobre que partições o projectista estabeleceu como proibidas para os nodos;
- a informação necessária para estimar o espaço ocupado em *hardware*, como por exemplo o valor das métricas associadas ao espaço ocupado pelas unidades funcionais, pelos elementos de armazenamento e pelos elementos de interligação;
- a informação necessária para estimar o desempenho do sistema, disponível apenas nos estados programa e quantifica métricas como as variáveis lidas e/ou escritas pelo estado programa, o tempo de computação, o tempo de comunicação com outros estados programa ou a frequência de execução.

As características gráficas dos diferentes tipos de nodo permitidos num grafo PSMfg são

esquematizadas na figura 6.2. Além das características cor e forma, que identificam o tipo de nodo, a figura indica o número e o tipo de ligações (arcos) que cada tipo de nodo pode estabelecer com outros nodos. O nome de cada instância dum nodo é colocado no seu interior. Os nomes da figura 6.2 não são mais do que o código atribuído a cada tipo de nodo. Por exemplo, *CC* é o código atribuído aos nodos do tipo *controlo dum ciclo*.

Símbolo	Descrição do Nodo	Símbolo	Descrição do Nodo
IS	Ponto de entrada no sistema	E	Ciclo de espera explícito
FS	Ponto de saída do sistema	A	Activação da condição do ciclo de espera explícito
IP	Início dum construtor paralelo	S	Ciclo de espera numa mudança de partição
FP	Fim dum construtor paralelo	C	Activação duma mudança de partição
II	Início dum construtor condicional	N	Normal
FI	Fim dum construtor condicional	V	Variável
in CC out	Controlo dum ciclo		

Figura 6.2: Tipos de nodo do grafo PSMfg.

Apresentam-se agora as características funcionais dos diferentes tipos de nodo do meta-modelo PSMfg:

- **Nodo IS** ***(inicioSistema)*** nodo que define o ponto de entrada no grafo do sistema, do qual todos os outros nodos são sucessores; não tem funcionalidade atribuída; os nodos IS e FS facilitam o manuseamento automático do grafo por parte dos algoritmos;
- **Nodo FS** ***(fimSistema)*** nodo que define o ponto de saída do grafo do sistema, do qual todos os outros nodos são antecessores; não tem funcionalidade associada;
- **Nodo IP** ***(inicioParalelo)*** define o ponto de arranque de processos paralelos; na descrição do sistema com o meta-modelo PSM (figura 6.3 (i)), ou numa descrição com o grafo PSMfg (figura 6.3 (ii)), o início dum construtor paralelo não tem funcionalidade

associada; deste modo, um nodo IP do grafo PSMfg serve apenas para dividir o fluxo de controlo e facilitar o manuseamento automático do grafo; já a modelação dum construtor paralelo com o meta-modelo FSM exige alguma atenção; como o meta-modelo das máquinas de estado não permite especificação de paralelismo, um construtor paralelo tem que ser descrito com várias FSMs, uma FSM principal e uma FSM por cada ramo do construtor paralelo (figura 6.3 (iii)); como é visível pela figura 6.3 (iii), a funcionalidade dum nodo IP é concretizada pelos estados assinalados com "*" nas FSMs;

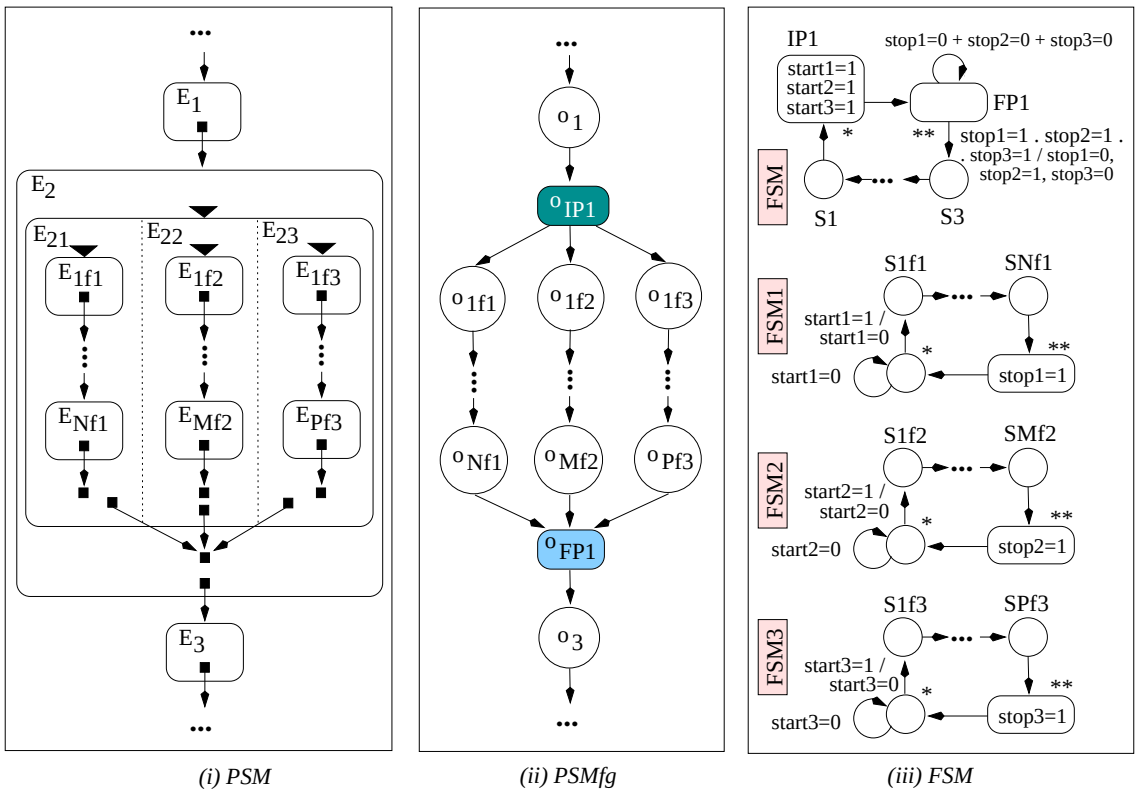


Figura 6.3: Modelação dum construtor paralelo: (i) com o meta-modelo PSM, (ii) com o grafo PSMfg e (iii) com máquinas de estados.

- **Nodo FP** ***(fimParalelo)*** define o ponto de conclusão de processos paralelos; um nodo FP do grafo PSMfg serve apenas para reunir o fluxo de controlo e facilitar o manuseamento automático do grafo; para o exemplo da figura 6.3 (iii), em que se modela um construtor paralelo com FSMs, o nodo FP é traduzido nos estados assinalados com "**";

- **Nodo II** ***(inicioIf)*** nodo que define o início dum construtor condicional; a sua funcionalidade é definida pela expressão que determina qual a ramificação por onde avança o fluxo de controlo; dado que um construtor condicional, qualquer que seja o número de opções de ramificação nele contemplado, pode ser decomposto em vários construtores com duas ramificações, decidiu implementar-se apenas o construtor condicional com

duas alternativas de ramificação;

- **Nodo FI** ***(fimIf)*** nodo que define o fim dum construtor condicional; não tem funcionalidade associada e serve apenas para reunir o fluxo de controlo;

- **Nodo CC** ***(controloCiclo)*** é um nodo que representa o controlo dum ciclo; a sua funcionalidade é definida pela expressão que determina se o fluxo de controlo continua no corpo do ciclo ou no exterior desse ciclo;

- **Nodo E** ***(espera)*** é um nodo que representa um ciclo de espera do tipo *wait until f(s1,s2,...,sn)* ou *wait on s1,s2,...,sn*, que obriga o processo em que o ciclo se insere a esperar até ocorrerem os eventos desejados nos sinais $s1$ a sn; a funcionalidade deste tipo de nodo é implementada pela unidade de controlo da partição a que o nodo foi atribuído, excepto a leitura de sinais que sejam externos à partição (figura 7.3 (v) e secção 7.3);

- **Nodo A** ***(activacao)*** nodo em que se activa um ou mais sinais necessários para que um nodo do tipo *espera*, operando em paralelo com este nodo, possa concluir a sua execução; do ponto de vista funcional equivale a um nodo do tipo *normal* que termina com a activação dos sinais citados; a distinção existe para facilitar a identificação automática dos nodos que interagem com os ciclos de espera; a identificação automática é útil à estimação do tempo de execução do sistema;

- **Nodo S** ***(sincronizacao)*** nodo que representa um objecto de *firmware* adicionado após o processo de partição, sempre que existe uma mudança de partição no fluxo de controlo (figura 7.4 e secção 7.3); a funcionalidade deste tipo de nodo traduz-se num ciclo que espera por um evento num sinal (*wait until* s_k); ou seja, um nodo tipo *sincronizacao* termina a execução após o nodo de *comutacao* a ele associado ter sido executado; como a funcionalidade dos nodos do tipo S é equivalente à de um nodo do tipo E, também é implementada pela unidade de controlo da partição a que o nodo foi atribuído, com a excepção da leitura do sinal s_k quando ele for externo à partição;

- **Nodo C** ***(comutacao)*** nodo associado dum nodo do tipo *sincronizacao*, representando um objecto de *firmware* adicionado após o processo de partição, sempre que existe uma mudança de partição no fluxo de controlo (figura 7.4); este nodo é responsável por gerar no sinal s_k o evento esperado pelo nodo do tipo S que lhe está associado;

- **Nodo N** ***(normal)*** nodo que não pertence a nenhum dos tipos anteriores, ou seja, é um nodo que representa a funcionalidade presente no modelo PSM do sistema, não contém ciclos de espera e não controla o fluxo de controlo de/para o seu exterior;

- **Nodo V** ***(variavel)*** nodo que representa uma variável do modelo PSM do sistema; os nodos do tipo *variavel* não se interligam com outros nodos.

Os arcos dum grafo PSMfg têm associada uma probabilidade de ramificação (relativa ao nodo de onde partem) e uma etiqueta. Excepto nos arcos de saída dos nodos do tipo $inicioIf$ e *controloCiclo*, a probabilidade de ramificação associada a um arco é 1, significando que o fluxo de controlo passa por esse arco o mesmo número de vezes que o nodo de origem é executado. A soma da probabilidade de ramificação associada aos dois arcos de saída dum nodo do tipo $inicioIf$ ou *controloCiclo* é 1. Para facilitar o processamento dos grafos, os dois arcos de saída dum nodo do tipo *controloCiclo* possuem uma etiqueta, "in" ou "out", que identifica o arco que entra ou sai do corpo do ciclo.

A figura 6.4 mostra um exemplo de grafo PSMfg contendo todos os tipos de nodos do meta-modelo.

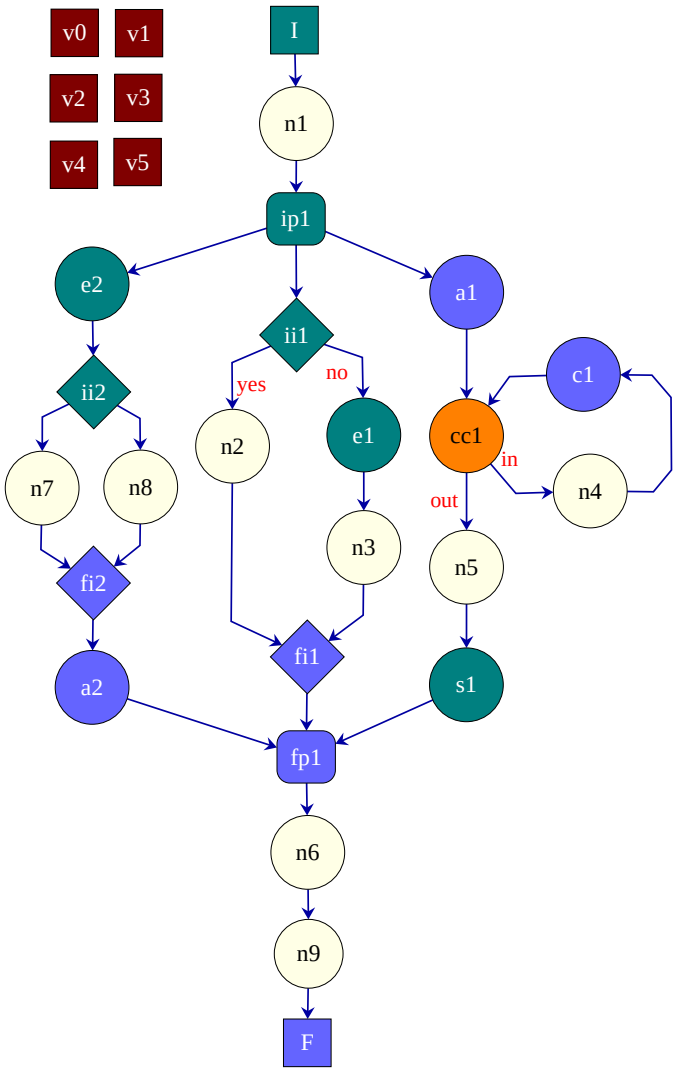


Figura 6.4: Exemplo dum grafo PSMfg.

6.3 Formulação do Processo de Partição

A partir da representação única dum sistema, o processo de partição gera uma descrição para cada componente da arquitectura alvo a usar na implementação desse sistema. Para conseguir este objectivo, é necessário dividir o conjunto de objectos que descreve o sistema, numa série de sub-conjuntos disjuntos a atribuir aos diversos componentes da arquitectura alvo. A tarefa de divisão dos objectos em sub-conjuntos é orientada pelos condicionalismos impostos pela arquitectura alvo e pelos requisitos do projecto. No presente trabalho, os **objectos** representam estados programa ou variáveis do modelo PSM do sistema. Apresenta-se agora uma definição formal de processo de partição.

Dado um conjunto de objectos que definem a funcionalidade do sistema $O = \{o_1, o_2, ..., o_n\}$, um conjunto de condicionalismos $Cons = \{c_1, c_2, ..., c_m\}$ e um conjunto de requisitos $Req = \{r_1, r_2, ..., r_p\}$ que ajudam a definir a exequibilidade e a qualidade das alternativas de partição a gerar, o processo de partição gera vários sub-conjuntos (ou partições) $H_1, ..., H_{nh}, S_1, ..., S_{ns}$, em que $H_i \subseteq O$, $S_i \subseteq O$, $\{H_i\}_{i=1..nh} \cup \{S_j\}_{j=1..ns} = O$, $H_i \cap S_j = \oslash$, $H_i \cap H_k = \oslash$ (com $k = 1..nh$ e $k \neq i$) e $S_j \cap S_l = \oslash$ (com $l = 1..ns$ e $l \neq j$).

Para seleccionar uma solução de partição, de entre as várias analisadas por um algoritmo de partição, recorre-se a uma função de custo $Fcusto$. Esta função utiliza os sub-conjuntos de objectos de *hardware* $H = \{H_1, ..., H_{nh}\}$, os sub-conjuntos de objectos de *software* $S = \{S_1, ..., S_{ns}\}$, o conjunto de condicionalismos $Cons$ e o conjunto de requisitos Req para devolver um valor que define a qualidade da solução. O algoritmo de partição iterativo é definido pela função

$$PartAlg(H, S, Cons, Req, Fcusto()) \tag{6.1}$$

que devolve H' e S' que garantem que

$$Fcusto(H', S', Cons, Req) \leq Fcusto(H, S, Cons, Req) \tag{6.2}$$

no caso de se empregar uma função de custo para a qual as condições que produzem o seu valor mínimo são aquelas que permitem obter a melhor solução de partição. O valor devolvido pela função de custo é obtido a partir de métricas, relacionadas com os condicionalismos e requisitos do sistema, que é preciso estimar. A estimação das métricas, abordada no capítulo 7, apoia-se em modelos dos recursos a usar na implementação do sistema: um modelo de *software* para o processador, um modelo de *hardware* para as FPGAs e os CPLDs e um modelo de comunicação para os recursos de interligação.

6.4 Construção duma Solução de Partição

A construção duma solução de partição, a utilizar como ponto de partida do processo de partição iterativo, é efectuada pelo algoritmo construtivo de crescimento de grupos (figura 6.5). O processo de construção duma solução começa com a selecção do objecto semente para cada partição. Para seleccionar o objecto semente das partições foram implementados quatro métodos:

- ◇ selecção aleatória dum objecto para cada partição, concretizada através da função *generateRandomSeeds* do algoritmo da figura 6.5;
- ◇ atribuição manual, por parte do utilizador, de pelo menos um objecto a cada partição; a confirmação de que todas as partições dispõem de pelo menos um objecto atribuído é implementada pela função *manualAssignedSeeds* do algoritmo da figura 6.5;
- ◇ selecção do objecto semente de cada partição através duma combinação da selecção aleatória com a atribuição manual; a prioridade de escolha recai na atribuição manual e apenas para as partições que não dispõem de nenhum objecto atribuído manualmente se escolhe o objecto semente de forma aleatória; este método é implementado pela função *manualRandomSeeds* do algoritmo da figura 6.5;
- ◇ para as partições que não possuem pelo menos um objecto atribuído manualmente, selecciona-se para objecto semente aquele que apresentar o maior número de ligações com todos os outros ainda não atribuídos; em termos de implementação, o número de ligações é representado pela intensidade de comunicação; no algoritmo da figura 6.5, este método é implementado pela função *manualMaxCommSeeds*.

A simplicidade dos métodos enunciados deve-se ao facto de a selecção de apenas um objecto por partição, mesmo que aplicando um método que não é o mais adequado, ter em princípio pouco impacto na qualidade da solução final de partição. Como os objectos visivelmente maus candidatos para uma determinada implementação (*software* ou *hardware*) podem ser atribuídos manualmente a uma partição que corresponda à implementação complementar (*hardware* ou *software*), também reforça a opção por métodos simples.

Após a selecção do objecto semente de cada partição, o algoritmo de crescimento de grupos atribui os objectos restantes à melhor partição para esses objectos. A indicação de qual a melhor partição para incluir um objecto é dada pela função *selectBestObjAssign* da figura 6.6. Em qualquer ponto do processo de construção, a função procura medir o grau de proximidade entre o objecto a atribuir e os objectos já atribuídos a cada partição.

```
// Construir uma solução de partição com o algoritmo construtivo de crescimento de
// grupos, atribuindo todos os objectos não atribuídos à melhor partição. A melhor
// partição é escolhida pela função de proximidade selectBestObjAssign.

algCrescimentoGrupos (O, seedMethod) ≡

    // O é o conjunto de objectos do sistema

    // Seleccionar o objecto semente de cada partição usando uma selecção aleatória,
    // uma atribuição manual, uma combinação de ambas ou o critério do número máximo
    // de ligações

    caso (seedMethod) seja
        randomSeed ⇒
            generateRandomSeeds(O)
        manualSeed ⇒
            manualAssignedSeeds(O)
        manRandSeed ⇒
            manualRandomSeeds(O)
        manCommSeed ⇒
            manualMaxCommSeeds(O)
    fcaso

    // Atribuir os objectos restantes à melhor partição

    para (cada objecto obj ∈ O) fazer
        se (obj não está atribuído a qualquer partição) então
            bestPart = selectBestObjAssign (O,obj)
            Atribuir o objecto à partição de que está mais próximo → bestPart
        fse
    fpara
```

Figura 6.5: Algoritmo de crescimento de grupos.

Para decidir qual a melhor partição para atribuir um objecto, a função $selectBestObjAssign$ dispõe de três funções de proximidade:

- $F_{var}(epsCG)$, em que $epsCG$ representa a intensidade de comunicação entre uma variável e os estados programa atribuídos a uma partição;
- $F_{epHwSw}(TexecSW, TexecHW, varsCG)$, onde $TexecSW$ ($TexecHW$) designa o tempo de computação dum estado programa em *software* (*hardware*) e $varsCG$ quantifica a intensidade de comunicação entre um estado programa e o conjunto das variáveis atribuídas a uma partição;
- $F_{epHw}(areaCG, varsCG)$, em que $areaCG$ designa o espaço ocupado por todas as variáveis e estados programa atribuídos a uma partição.

A função F_{var} é utilizada na atribuição de variáveis e as funções F_{epHwSw} e F_{epHw} são usadas na atribuição de estados programa. Na descrição da função $selectBestObjAssign$ (figura 6.6) o valor devolvido pelas funções de proximidade F_{var}, F_{epHwSw} e F_{epHw} é designado por $HWSWvarsCost$, $HWSWcost$ e $HWcost$, respectivamente.

selectBestObjAssign $(O, obj) \equiv$

se (obj *é uma variável*) então

se ($areaHWvar(obj) > LIMITE\_AREA$) então
pSel = SW
senão
Calcular a comunicação entre obj e cada uma das partições → $epsCG$
Usando $epsCG$ calcular a proximidade para cada partição → $HWSWvarsCost$

pSel = partição com o menor valor de $HWSWvarsCost[\,]$
fse

senão // *O objecto obj é um estado programa*

Calcular o espaço ocupado por cada partição de *hardware* → $areaCG$

Calcular a comunicação entre obj e cada uma das partições → $varsCG$

Usando $areaCG$ e $varsCG$ calcular a proximidade para cada partição de *hardware* → $HWcost$
// ($HWcost$ é utilizado na selecção da melhor partição de *hardware*)

Calcular o tempo de computação mínimo e máximo de obj → $minTcomp$ e $maxTcomp$

Usando $varsCG$ e tempos de computação calcular a proximidade de cada partição → $HWSWcost$
// ($HWSWcost$ é utilizado para optar entre uma partição de *software* ou de *hardware*)

se (*menor valor em* $HWSWcost[\,]$ *é* $HWSWcost[SW]$) então
pSel = SW
senão
pSel = partição (de *hardware*) com o menor valor de $HWcost[\,]$
fse

fse

devolver (pSel)

Figura 6.6: Selecção da melhor atribuição para um objecto, durante o processo de construção duma solução de partição com o algoritmo de crescimento de grupos.

Se o objecto a atribuir for uma variável que é uma má candidata para *hardware*, ou seja, é uma variável cujo espaço ocupado em *hardware* ultrapassa o limite fixado, a função $selectBestObjAssign$ atribui-o a *software*. Se a variável não for uma má candidata para *hardware*, a função atribui-a à partição que apresente a maior intensidade de comunicação com esta variável, ou seja, à partição p que apresente o menor valor $HWSWvarsCost[p]$.

Se o objecto a atribuir for um estado programa, a função $selectBestObjAssign$ calcula o valor $HWSWcost[p]$ com base no tempo de computação em *software* e em *hardware* e na quantidade de comunicação entre o estado programa e cada uma das partições p ($varsCG[p]$). Se a partição que gerou o menor valor $HWSWcost$ for de *software*, o estado programa é imediatamente atribuído a *software*, senão utiliza-se como critério de selecção da melhor partição de *hardware* o valor $HWcost$, que é mais adequado para *hardware* do que $HWSWcost$. O estado programa é atribuído à partição que apresente o menor valor de $HWcost$. Para o valor $HWcost[p]$ duma

partição de *hardware* p contribuem o espaço ocupado pela partição ($areaCG[p]$) e a quantidade de comunicação entre o estado programa e a partição ($varsCG[p]$).

Uma versão mais detalhada da função $selectBestObjAssign$, apresentada na figura 6.6, pode ser consultada nas figuras E.1 e E.2 do apêndice E.

6.4.1 Estimação das Métricas usadas na Construção duma Solução de Partição

Embora haja um capítulo dedicado à estimação de métricas, como a estimação das métricas envolvidas na função de proximidade do algoritmo de crescimento de grupos é relativamente simples, optou-se por apresentá-la no seguimento da descrição da função ($selectBestObjAssign$).

A proximidade utilizada na selecção da melhor partição p para atribuir uma variável v, $HWSWvarsCost$ na função $selectBestObjAssign$, é calculada com a equação 6.4. O valor de $HWSWvarsCost$ depende da estimativa para a intensidade de comunicação entre a variável v e cada uma das partições p, ***epsCG[p]*** definida na equação 6.3. O valor $epsCG[p]$ considera apenas a quantidade de leituras e escritas, sobre a variável v, efectuadas pelos estados programa atribuídos à partição p.

$$\begin{aligned} epsCG[p] = \\ = \sum_{o \in (OBJread(v) \cap p)} readVAR(o).numLeituras(v) * readVAR(o).probLeitura(v) * FN(o) + \\ + \sum_{o \in (OBJwrite(v) \cap p)} writeVAR(o).numEscritas(v) * writeVAR(o).probEscrita(v) * FN(o) \end{aligned} \quad (6.3)$$

$$HWSWvarsCost[p] = \frac{maxEps - epsCG[p]}{maxEps - minEps} \quad (6.4)$$

em que

- $\boldsymbol{OBJread(v)}$ ($\boldsymbol{OBJwrite(v)}$) é o conjunto de estados programa que lêem (escrevem) a variável v;
- $\boldsymbol{readVAR(o)}$ ($\boldsymbol{writeVAR(o)}$) representa o conjunto de variáveis lidas (escritas) pelo estado programa o;
- $\boldsymbol{readVAR(o).numLeituras(v)}$ ($\boldsymbol{writeVAR(o).numEscritas(v)}$) é o número de vezes que a variável v é lida (escrita) por o em cada execução;
- $\boldsymbol{readVAR(o).probLeitura(v)}$ ($\boldsymbol{writeVAR(o).probEscrita(v)}$) designa a probabilidade de leitura (escrita) da variável v por parte de o;

- ***FN(o)*** é a frequência de execução de o;
- ***minEps*** (***maxEps***) é o valor mínimo (máximo) da intensidade de comunicação entre a variável e uma partição, no conjunto de todas as partições.

A estimativa do espaço ocupado por cada partição de *hardware* p, ***areaCG[p]*** na função $selectBestObjAssign$, é obtida pela equação 6.5. O valor $areaCG[p]$ considera apenas o espaço ocupado pelas variáveis e estados programa da partição p, o que significa que não inclui o espaço ocupado pelos recursos de interligação ou de interface.

$$areaCG[p] = areaEProgramaCG[p] + areaVariaveisCG[p] =$$

$$= \sum_{(o \in p) \ \wedge \ (tipo(o) \neq variavel)} areaHW(o) \ + \sum_{(v \in p) \ \wedge \ (tipo(v) = variavel)} areaHWvar(v) \qquad (6.5)$$

onde

- ***areaHW(o)*** representa o espaço ocupado pelo estado programa o;
- ***areaHWvar(v)*** designa o espaço ocupado pela variável v.

Para estimar a intensidade de comunicação entre o estado programa o e as variáveis atribuídas a uma partição p, ***varsCG[p]*** na função $selectBestObjAssign$, aplica-se a equação 6.6. O valor $varsCG[p]$ considera apenas a quantidade de leituras e escritas efectuadas por o sobre as variáveis atribuídas à partição p.

$$varsCG[p] = FN(o) *$$

$$[\sum_{v \ \in \ (readVAR(o) \ \cap \ p)} readVAR(o).numLeituras(v) * readVAR(o).probLeitura(v) +$$

$$+ \sum_{v \ \in \ (writeVAR(o) \ \cap \ p)} writeVAR(o).numEscritas(v) * writeVAR(o).probEscrita(v) \ ] \qquad (6.6)$$

A estimativa da proximidade utilizada na selecção da melhor partição de *hardware* para um determinado estado programa, $HWcost$ na função $selectBestObjAssign$, é calculada pela equação 6.7. A expressão que define $HWcost$ pretende seleccionar a partição de *hardware* que seja acedida mais vezes pelo estado programa o, para ler ou escrever variáveis atribuídas a essa partição e/ou a partição de *hardware* que esteja menos ocupada. Para atribuir uma importância maior ao primeiro objectivo, aplicou-se um peso maior ao termo que depende da intensidade de comunicação do que ao termo que depende do espaço ocupado na partição ($CKvars > CKarea$ na equação 6.7).

$$HWcost[p] = CKvars * \frac{maxVars - varsCG[p]}{maxVars - minVars} + CKarea * \frac{areaCG[p] - minArea}{maxArea - minArea} \quad (6.7)$$

onde

- ***minVars*** (***maxVars***) é o valor mínimo (máximo) da intensidade de comunicação entre o estado programa e uma partição, no conjunto de todas as partições;
- ***minArea*** (***maxArea***) é o valor mínimo (máximo) do espaço ocupado por uma partição, no conjunto de todas as partições de *hardware*;
- ***CKvars***=3 e ***CKarea***=1.

Por último, a estimativa da proximidade que permite optar entre a atribuição dum estado programa a uma partição de *software* ou de *hardware*, designada por $HWSWcost[\,]$ na função $selectBestObjAssign$, é calculada pela equação 6.8. O objectivo da expressão que define $HWSWcost$ é seleccionar a partição que seja acedida mais vezes por o para ler ou escrever variáveis atribuídas a essa partição e/ou a partição em que o estado programa o apresente o tempo de computação inferior. Para atribuir maior importância ao primeiro objectivo, aplicou-se um peso maior ao termo que depende da intensidade de comunicação do que ao termo que depende do tempo de computação ($CKvars > CKtcomp$ na equação 6.8).

$$HWSWcost[p] = \begin{cases} CKtcomp * \frac{TexecSW(o) - minTcomp}{maxTcomp - minTcomp} + CKvars * \frac{maxVars - varsCG[p]}{maxVars - minVars} & , p = SW \\ CKtcomp * \frac{TexecHW(o) - minTcomp}{maxTcomp - minTcomp} + CKvars * \frac{maxVars - varsCG[p]}{maxVars - minVars} & , p \neq SW \end{cases} \quad (6.8)$$

em que

- ***TexecSW(o)*** (***TexecHW(o)***) é o tempo de computação em *software* (*hardware*) do estado programa o;
- ***minTcomp*** (***maxTcomp***) é o menor (maior) valor entre $TexecSW(o)$ e $TexecHW(o)$;
- ***CKtcomp***=1.

6.5 Processo de Partição Iterativo

O método de **pesquisa tabu** pode ser visto como uma extensão das estratégias de **pesquisa local**, onde uma nova solução se obtém da vizinhança da solução actual, através de regras bem definidas [GL95] [EKP98a] [Pir96]. Quando na iteração n do processo de pesquisa se

procura minimizar a função $Fcusto(P_n)$, a nova solução P_{n+1} é seleccionada a partir da vizinhança da solução actual $V(P_n)$, aplicando um critério de optimização. Geralmente, o critério traduz um objectivo que visa seleccionar a melhor solução presente na vizinhança. Se a estratégia de pesquisa for do tipo *descent*[2] ou *steepest descent*, a melhor solução da vizinhança $V(P_n)$ é seleccionada desde que seja melhor ou equivalente à solução actual, ou seja, se $Fcusto(P_{n+1}) \leq Fcusto(P_n)$. Quando esta condição não for verificada a pesquisa termina. Aplicando uma estratégia do tipo *descent*, os algoritmos não são capazes de escapar aos mínimos locais de $Fcusto$, o que constitui uma forte limitação para a resolução de problemas em que se deseja uma solução próxima do mínimo absoluto de $Fcusto$. A pesquisa tabu é uma estratégia de pesquisa local que procura não parar em mínimos locais de $Fcusto$, mas a sua filosofia não se esgota nesta característica. De facto, a estratégia foi designada de pesquisa tabu porque em cada iteração são definidas zonas proibidas no espaço de soluções, ou por outras palavras, certas soluções são consideradas tabu. Para atingir esse objectivo, a pesquisa tabu recorre a uma estrutura flexível de memória que permite implementar diversas estratégias de pesquisa, entre as quais, a fuga a mínimos locais. A **memória flexível** é constituída por componentes designados de **curta duração** e de **longa duração**. Os componentes de curta duração baseiam-se no historial das soluções visitadas mais recentemente, enquanto os componentes de longa duração se baseiam nas soluções mais frequentes. Existem ainda componentes de **duração intermédia** orientados para soluções com qualidade e soluções influentes. Deste modo, a estrutura de memória da pesquisa tabu guarda informação sobre as soluções visitadas mais recentemente, as soluções mais frequentes, com melhor qualidade e as mais influentes. Com a estrutura de memória apresentada, a pesquisa tabu pretende atingir os seguintes objectivos:

- ◇ diversificar a pesquisa, de modo a escapar a mínimos locais;
- ◇ intensificar a pesquisa, para reforçar a convergência para o mínimo absoluto;
- ◇ evitar ciclos durante a pesquisa.

Diz-se que a pesquisa entra em **ciclo**, quando repetidamente se visita a solução visitada na iteração anterior da pesquisa. Nos métodos baseados em pesquisa local, a ocorrência de ciclos deve-se essencialmente à estrutura de vizinhança adoptada. A pesquisa tabu selecciona de uma forma muito acentuada a melhor solução da vizinhança, ou subvizinhança, da solução actual, mesmo que essa solução seja pior que a solução actual. Este comportamento, que não está presente nos métodos do tipo *descent*, só por si já contribui para que durante a pesquisa se consiga escapar a mínimos locais da função de custo. Contudo, a estrutura de vizinhança

[2] As estratégias do tipo *descent* também são conhecidas por estratégias do tipo *greedy*.

resultante desta estratégia de pesquisa não consegue evitar os ciclos, porque ao seleccionar-se a melhor solução P_{n+1} da vizinhança $V(P_n)$, em que P_n é a solução de partição actual, pode acontecer que P_n seja a melhor solução da vizinhança $V(P_{n+1})$ na iteração seguinte, o que levaria a pesquisa a voltar ao estado imediatamente anterior. Para evitar esta situação são guardadas as últimas L soluções visitadas, no que se designou de **lista tabu**. Se uma solução P_n está na lista tabu então a evolução para esta solução é proibida. Evita-se assim que se regresse, pelo menos durante L iterações, a uma solução visitada. A lista tabu funciona como memória de curta duração, dado guardar apenas o historial recente da pesquisa. Em sistemas de elevada dimensão, guardar as últimas L soluções visitadas pode exigir um tempo de cálculo elevado. Numa implementação mais viável da lista tabu guarda-se apenas uma característica ou atributo da transição da solução actual para a nova solução, o que naturalmente depende da estrutura de vizinhança implementada, ou seja, depende do modo como se evolui duma solução para a seguinte. Este tipo de implementação da lista tabu não consegue evitar que se entre em ciclo, mas ao introduzir diversificação na pesquisa, permite escapar aos mínimos locais. A lista tabu introduz diversificação na pesquisa porque, ao definir zonas do espaço de soluções como sendo proibidas, permite que se visitem soluções em zonas menos exploradas. Deste modo, a pesquisa tabu consegue escapar a mínimos locais de uma forma mais efectiva. A razão para a eficácia deste tipo de lista tabu reside no facto de que quando se considera que uma característica ou atributo duma transição é tabu, excluem-se do espaço de soluções todas as soluções que partilhem dessa característica ou atributo, resultando daqui uma lista mais restritiva que favorece a diversificação.

O tamanho da lista tabu é determinante para a evolução da pesquisa, uma vez que ele influencia o grau de restrição aplicado ao espaço de soluções que se pode pesquisar. O tamanho da lista tabu também é designado de **validade do tabu**. Para definir o valor da validade do tabu pode aplicar-se as seguintes regras:

- ◇ regra estática: a validade do tabu deve ser uma constante ou uma expressão que depende da dimensão do problema;
- ◇ regra dinâmica: a validade do tabu varia dinamicamente dentro dum intervalo; essa variação pode ser aleatória ou sistemática; a dimensão do problema pode ser incluída nos limites de variação, o que faz com que o número de soluções proibidas varie ao longo duma pesquisa;
- ◇ a validade do tabu não deve ser nem muito curta (para evitar que a pesquisa entre em ciclo), nem muito longa (para impedir que a pesquisa se afaste do mínimo absoluto da função de custo);
- ◇ quanto mais restritivo for o tipo de tabu, menor deve ser a validade do tabu que lhe

corresponde.

Com base nas experiências efectuadas estabeleceu-se uma recomendação para a validade dos tabus: a validade do tabu aplicado aos objectos e aos deslocamentos deve ser 5 a 10% do número de objectos do sistema.

Da exclusão temporária de soluções, não resultam apenas vantagens para o método de pesquisa tabu. As desvantagens surgem quando se excluem duma pesquisa soluções de elevada qualidade, que no fundo são o objectivo da pesquisa. Para ultrapassar este inconveniente recorre-se a um mecanismo que permite desautorizar a classificação duma solução como sendo tabu, desde que seja uma solução de qualidade. A este mecanismo atribui-se o nome de **critério de aspiração**, dado que uma solução considerada tabu e por isso impedida de ser seleccionada, pode aspirar a ser visitada na presente iteração da pesquisa. Objectivamente, o critério de aspiração identifica as soluções de qualidade e, de acordo com o critério de aspiração mais consensual, uma solução tabu pode ser seleccionada desde que o seu custo seja menor que o custo da melhor solução encontrada até ao momento. Deste modo, obtém-se um problema de minimização da função de custo e o critério de aspiração é designado por **aspiração por objectivo**.

Pode acontecer que durante uma pesquisa todas as soluções candidatas sejam tabu e não possam ser seleccionadas por qualquer critério de aspiração definido, como por exemplo o critério de aspiração por objectivo. Nestes casos, é comum seleccionar-se a solução menos tabu, que pode ser definida como a solução que provoca o menor aumento no valor da função de custo. Existem outros critérios de aspiração, como por exemplo a aspiração por direcção de pesquisa e a aspiração por influência [GL95].

6.5.1 Estratégias de Diversificação e Intensificação na Pesquisa

Além dos componentes de curta duração, também os componentes de longa duração podem contribuir para a **diversificação** das pesquisas. Os componentes de longa duração guardam a frequência com que as soluções[3] ocorrem durante a pesquisa, uma informação utilizada na condução da pesquisa para zonas não exploradas, através da aplicação duma bonificação (sob a forma de redução do custo) que favorece a selecção das soluções menos visitadas, em detrimento das soluções mais frequentes. No entanto, esta estratégia de diversificação só deve ser aplicada quando a vizinhança não contém soluções melhores do que a solução actual, uma vez que o objectivo principal da pesquisa é encontrar uma solução óptima para o problema de partição. A frequência com que uma solução ocorre durante a pesquisa pode ser avaliada de diversas formas. Ao número de vezes que uma determinada solução ocorre na

[3]Como foi explicado anteriormente, em vez das soluções pode guardar-se apenas algumas características ou atributos da transição entre soluções.

pesquisa podemos associar outros indicadores, como o número total de iterações, formando assim quocientes que podem exprimir diferentes pontos de vista de frequência. Deste modo, pode distinguir-se dois tipos de frequência relativos a características ou atributos da transição entre soluções:

- ◇ a frequência de permanência, que como o nome indica, é uma medida da permanência dum determinado atributo num conjunto de soluções;
- ◇ a frequência de transição, que é um indicador da mudança que um determinado atributo sofre num conjunto de soluções.

O recurso a determinados tipos de frequência estabelece a base para o cálculo de indicadores de qualidade e de influência das soluções. Estes indicadores são componentes de duração intermédia e são utilizados num estratégia de **intensificação**, onde o objectivo principal é encontrar soluções de qualidade[4]. A intensificação consiste em incorporar numa solução atributos com qualidade. Se a qualidade desses atributos for estimada através dum indicador que envolve componentes de longa duração, então estamos perante uma intensificação de curta duração. Se, por outro lado, esses atributos forem seleccionados de soluções de elite (mínimos locais) que se distinguem pela sua qualidade ao longo da pesquisa, então estamos perante uma intensificação de longa duração. Normalmente, as estratégias de diversificação e de intensificação reforçam-se uma à outra, resultando daqui uma capacidade acrescida para a pesquisa evitar mínimos locais e consequentemente para encontrar soluções óptimas.

A pesquisa tabu pode ser dotada de mais elementos de diversificação e intensificação. Como foi referido, a pesquisa tabu selecciona a melhor solução da vizinhança, o que obriga a avaliar toda a vizinhança para garantir que se escolhe a melhor solução. Se o problema de partição possuir uma dimensão considerável, a avaliação de toda a vizinhança exige um tempo de cálculo elevado, tanto mais que a avaliação é repetida em cada iteração da pesquisa. Em alternativa à avaliação de toda a vizinhança, pode optar-se por avaliar apenas um subconjunto de soluções, ou seja, utilizar apenas uma subvizinhança. Este subconjunto de soluções designa-se por **lista de soluções candidatas**, uma vez que é desta lista que sai a solução a seleccionar na iteração presente.

Para construir a lista de soluções candidatas pode usar-se uma amostra aleatória de soluções da vizinhança, uma técnica também utilizada nos métodos Monte Carlo [GL95]. Para melhorar a eficácia da pesquisa tabu, deve evitar-se a selecção aleatória de soluções, efectuando em alternativa uma selecção mais inteligente. Surge então uma lista de soluções candidatas constituída pelo subconjunto das melhores soluções presentes na vizinhança. Esta forma

[4]Uma solução de qualidade corresponde a um mínimo local ou absoluto da função de custo.

de definição da subvizinhança, constitui mais um elemento de intensificação na pesquisa. Para que a lista de soluções candidatas permita uma acção de diversificação na pesquisa, aplica-se uma regra que faz com que as soluções não examinadas[5] na iteração presente, sejam examinadas nas próximas iterações da pesquisa. Desta forma, diferentes zonas da vizinhança são examinadas em diferentes iterações, o que naturalmente diversifica a pesquisa.

Outra técnica que introduz diversificação na pesquisa consiste em alterar a estrutura de vizinhança, tornando-a mais complexa. Esta técnica permite a execução duma sequência de transições simples, a partir da solução presente para uma nova solução, resultando daqui uma única transição mais complexa. Em termos práticos, a transição complexa é formada por uma sucessão de $maxTrans$ transições simples, em que a transição t provoca a transição $t+1$, terminando a sucessão quando for executada a transição $maxTrans$ ou quando se verificar uma condição de paragem. Com a estrutura de vizinhança assim definida consegue-se que, em cada iteração, as características da solução actual sejam alteradas com alguma profundidade, o que favorece a diversificação na pesquisa.

6.5.2 Estrutura de Vizinhança

A vizinhança duma solução P_n pode ser definida como o conjunto de todas as alternativas de solução que se obtêm através duma regra que modifica as características ou atributos de P_n. No problema de partição em *hardware* e *software*, a transição da solução presente para uma solução na sua vizinhança ocorre quando se desloca pelo menos um objecto da sua partição de origem para uma partição de destino, resultando daqui uma nova solução. Utilizando a notação da programação lógica, um deslocamento é representado como um facto $d(obj, Porig, Pdest)$. Em termos de tempo de cálculo é relevante estimar o número de alternativas de solução em cada iteração da pesquisa, ou seja, determinar o tamanho da vizinhança, porque a pesquisa tabu escolhe sempre a melhor alternativa na (sub)vizinhança da solução presente, o que implica a avaliação de toda a (sub)vizinhança. Para um problema de partição em que o número de objectos do sistema é $nObj$, o número de partições é $nPart$ e uma nova solução de partição é obtida com o deslocamento de apenas um objecto, o tamanho da vizinhança é expresso por $nObj * (nPart - 1)$.

O exemplo introduzido na tabela 6.1 mostra a relação entre a dimensão dum problema de partição (distribuir 5 objectos por 3 partições) e o tamanho da vizinhança (conjunto de 10 soluções de partição). As tabelas 6.1, 6.2, 6.3 e 6.4 apresentam os resultados da aplicação do algoritmo de pesquisa tabu na resolução deste problema de partição. Para cada uma das iterações 1 a 4, as tabelas mostram a solução de partição antes e após a iteração, a lista tabu e os deslocamentos por ela proibidos, os deslocamentos possíveis e as soluções da vizinhança

[5] As soluções não examinadas são as soluções não incluídas na lista de candidatas.

que resultam da sua concretização.

Qualquer deslocamento $d(o,p,p)$, em que a partição de origem e de destino são a mesma, não é válido porque não origina uma nova solução. Se se modificasse a estrutura de vizinhança, de modo a permitir o deslocamento de múltiplos objectos para gerar uma nova solução, o tamanho da vizinhança aumentaria consideravelmente.

Problema de Partição a Resolver	
Dimensão do problema	$nObj = 5$ (número de objectos) $nPart = 3$ (número de partições)
Tamanho da vizinhança	$nObj * (nPart - 1) = 5$*(3-1) $= 10$
Tamanho da lista de candidatos (ou subvizinhança a analisar)	$nObj = 5$
Estrutura de vizinhança	Deslocar um objecto da partição de origem para a partição de destino
Tamanho da lista tabu (ou validade dos tabus)	$L = 2$

Primeira Iteração	
Solução inicial	$s(1,3,2,2,3)$ †
Nova solução	$s(1,3,2,1,3)$
Lista tabu	{ }
Deslocamentos proibidos pela lista tabu	
Deslocamento → valor da proximidade entre o objecto a deslocar e a partição de destino	$d(1,1,2) \rightarrow Fprox = 3.3$ $d(1,1,3) \rightarrow Fprox = 0.3$ $d(2,3,1) \rightarrow Fprox = 2.1$ $d(2,3,2) \rightarrow Fprox = 0.8$ $d(3,2,1) \rightarrow Fprox = 1.9$ $d(3,2,3) \rightarrow Fprox = 0.5$ $d(4,2,1) \rightarrow Fprox = 1.8$ $d(4,2,3) \rightarrow Fprox = 4.4$ $d(5,3,1) \rightarrow Fprox = 2.4$ $d(5,3,2) \rightarrow Fprox = 3.1$
Lista de candidatos ‡ → solução vizinha resultante → custo da solução vizinha	$d(1,1,3) \rightarrow s(3,3,2,2,3) \rightarrow Fcusto = 5.3$ $d(2,3,2) \rightarrow s(1,2,2,2,3) \rightarrow Fcusto = 4.1$ $d(3,2,3) \rightarrow s(1,3,3,2,3) \rightarrow Fcusto = 5.7$ $d(4,2,1) \rightarrow s(1,3,2,1,3) \rightarrow Fcusto = 0.8$ $d(5,3,1) \rightarrow s(1,3,2,2,1) \rightarrow Fcusto = 1.9$
Deslocamento efectuado	$d(4,2,1)$

† $s(1,3,2,2,3)$ é a solução de partição em que os objectos 1,2,3,4,5 estão atribuídos às partições 1,3,2,2,3.

‡ Obtida do melhor deslocamento para cada objecto.

Tabela 6.1: Exemplo de aplicação da pesquisa tabu na resolução do problema de partição (primeira iteração duma pesquisa).

Como é comum o problema de partição em *hardware* e *software* ter que avaliar um número elevado de alternativas de partição, a obtenção duma solução de qualidade pode exigir um tempo de cálculo também elevado. Para evitar que se avaliem todas as alternativas presentes na vizinhança da solução actual implementa-se a lista de soluções candidatas que, como se afirmou anteriormente, equivale a ter que avaliar apenas uma subvizinhança da solução actual. Deste modo, em cada iteração da pesquisa gera-se uma lista de candidatos com um tamanho fixo ao longo da pesquisa e igual ao número de objectos do sistema. Para seleccionar as soluções

candidatas recorre-se a uma função $Fprox$ que exprime a proximidade dum objecto em relação aos objectos atribuídos a uma determinada partição, distinta daquela a que está atribuído na solução actual (secção 6.6.6). A partir do deslocamento dum objecto para a partição de que o objecto está mais próximo, ou seja, efectuando o deslocamento a que corresponde o menor valor de $Fprox$, obtém-se uma solução candidata. Repetindo esta operação com todos os objectos gera-se a lista de candidatos. A tabela 6.1 ilustra o processo de geração da lista de candidatos na primeira iteração da pesquisa. Como o factor de redução da vizinhança para a subvizinhança é $nPart - 1 = 2$, o tamanho da lista de candidatos é metade do total de alternativas de solução. Deste modo, quanto maior for o número de partições envolvidas, maior é a redução no tempo de cálculo resultante da utilização da lista de candidatos.

O deslocamento a efectuar numa iteração é aquele que conduz à solução, presente na lista de candidatos, com o menor custo. Na tabela 6.1, e de acordo com o menor valor de $Fcusto$, o deslocamento efectuado é $d(4, 2, 1)$.

6.5.3 Selecção da Lista Tabu

Na sua forma mais comum, a lista tabu contém as L soluções que são proibidas em cada iteração da pesquisa. No problema de partição em *hardware* e *software*, uma solução é definida pelo conjunto dos objectos do sistema e pela partição a que cada objecto está atribuído. Como os problemas de grande dimensão exigem pesquisas com milhares de iterações, guardar as L soluções e verificar se cada solução da lista de candidatos pertence à lista tabu[6] não é exequível. A solução para este problema conduz a uma implementação da lista tabu em que apenas se guarda uma ou várias características/atributos das últimas L soluções visitadas.

No âmbito do problema de partição, uma nova solução de partição é obtida a partir da solução actual deslocando um objecto para uma partição distinta daquela em que se encontra. Deste modo a classificação tabu pode transitar das soluções para os deslocamentos. Neste contexto definem-se 6 tipos de **classificação tabu**:

1. Classifica-se como tabu o deslocamento dum objecto *obj* de uma partição de origem *po* para uma partição de destino *pd*; este é o tipo de classificação tabu que impõe menos restrições à pesquisa;

2. Classifica-se como tabu todos os deslocamentos em que intervém o objecto *obj*; este tipo de tabu coloca maiores restrições ao espaço de procura do que o tipo anterior;

3. Classifica-se como tabu todos os deslocamentos que tenham por partição de origem *po*, o que constitui uma restrição na pesquisa ainda maior;

[6] Estas duas tarefas são executadas em cada iteração da pesquisa.

4. Classifica-se como tabu todos os deslocamentos que tenham por partição de destino pd; teoricamente, este tipo de tabu impõe à pesquisa as mesmas restrições que o tipo de tabu anterior;

5. Classifica-se como tabu todos os deslocamentos dum objecto obj para uma partição de destino pd, sem importar qual a partição de origem de obj; este tipo de tabu é ligeiramente mais restritivo que o tabu tipo 1;

6. Classifica-se como tabu todos os deslocamentos dum objecto obj de uma partição de origem po, sem importar qual a partição de destino de obj; teoricamente, este tipo de tabu impõe à pesquisa as mesmas restrições que o tabu tipo 5.

A escolha dos tipos de tabu a implementar deve ser bem ponderada, dado que os vários tipos introduzem restrições diferentes ao espaço de soluções pesquisáveis e, em boa parte, o efeito dos vários tipos de tabu é aditivo. É fácil constatar que o tabu menos restritivo é o tabu aplicado ao deslocamento dum objecto obj a partir duma partição de origem po para uma partição de destino pd (tabu tipo 1). De acordo com o tabu tipo 1, de todas as soluções que se podem atingir a partir da solução presente, através dum deslocamento $d(obj, po, pd)$, apenas uma das soluções poderá ser proibida. Isto porque a solução só será proibida se o objecto obj estiver atribuído à partição po. Caso não esteja, o deslocamento $d(obj, po, pd)$ não representa uma solução da vizinhança da solução actual. Por seu lado, ao classificar como tabu uma partição, de origem ou de destino, restringe-se fortemente o espaço de soluções a pesquisar.

A segunda iteração do exemplo em apresentação, tabela 6.2, ilustra o efeito dos diferentes tipos de tabu, indicando para cada classificação tabu o número de soluções que são proibidas. A análise efectuada neste exemplo, considera que se aplicam os seis tipos de classificação tabu definidos. Tal como previsto, o exemplo permite concluir que os tabus tipo 3 e tipo 4 são os mais restritivos. Esta conclusão é reforçada na terceira e quarta iterações. Contudo, a restrição teórica prevista para cada tipo de classificação tabu é afectada pela configuração da solução actual. A excepção é o tabu tipo 2, que depende exclusivamente da dimensão do problema e origina[7] um número de soluções proibidas que no máximo é $nPart - 1$.

Da simulação da terceira iteração da pesquisa extraem-se conclusões mais definitivas sobre o grau de restrição imposto pelos diferentes tipos de classificação tabu, uma vez que a lista tabu está totalmente preenchida nesta iteração (tabela 6.3). A terceira iteração mostra que, ao dispor-se de mais um deslocamento inserido na lista tabu[8], a restrição imposta pelo segundo, terceiro e quarto tipos de classificação tabu aumentou, o que não acontece com os restantes

[7] Em cada iteração da pesquisa e para cada objecto proibido.

[8] Os deslocamentos inseridos na lista tabu são os deslocamentos que originaram as duas últimas soluções visitadas.

Segunda Iteração			
Solução anterior	$s(1,3,2,1,3)$		
Nova solução	$s(3,3,2,1,3)$		
Lista tabu	$\{\ d(4,2,1)\ \}$		
Deslocamentos proibidos pela lista tabu	Se $d(4,2,1)$	é tabu tipo 1 $\Rightarrow$	nº soluções proibidas = 0
	Se $d(4,-,-)$	é tabu tipo 2 $\Rightarrow$	nº soluções proibidas = 2
	Se $d(-,2,-)$	é tabu tipo 3 $\Rightarrow$	nº soluções proibidas = 2
	Se $d(-,-,1)$	é tabu tipo 4 $\Rightarrow$	nº soluções proibidas = 3
	Se $d(4,-,1)$	é tabu tipo 5 $\Rightarrow$	nº soluções proibidas = 0
	Se $d(4,2,-)$	é tabu tipo 6 $\Rightarrow$	nº soluções proibidas = 0
Deslocamento $\rightarrow$ solução vizinha resultante	$d(1,1,2) \rightarrow$	$s(2,3,2,1,3)$	$\langle$ *permitido* $\rangle$
	$d(1,1,3) \rightarrow$	$s(3,3,2,1,3)$	$\langle$ *permitido* $\rangle$
	$d(2,3,1) \rightarrow$	$s(1,1,2,1,3)$	
	$d(2,3,2) \rightarrow$	$s(1,2,2,1,3)$	$\langle$ *permitido* $\rangle$
	$d(3,2,1) \rightarrow$	$s(1,3,1,1,3)$	
	$d(3,2,3) \rightarrow$	$s(1,3,3,1,3)$	
	$d(4,1,2) \rightarrow$	$s(1,3,2,2,3)$	
	$d(4,1,3) \rightarrow$	$s(1,3,2,3,3)$	
	$d(5,3,1) \rightarrow$	$s(1,3,2,1,1)$	
	$d(5,3,2) \rightarrow$	$s(1,3,2,1,2)$	$\langle$ *permitido* $\rangle$
Deslocamento efectuado	$d(1,1,3)$		

Tabela 6.2: Exemplo de aplicação da pesquisa tabu na resolução do problema de partição (segunda iteração duma pesquisa).

tipos de tabu. O facto de a restrição imposta pelos restantes tipos de tabu não aumentar deve-se à configuração das soluções visitadas.

A quarta iteração da pesquisa, mostra o deslocamento a permanecer tabu durante L iterações, sendo L o número de deslocamentos que compõe a lista tabu (tabela 6.4). A simulação de quatro iterações reforça a conclusão de que o terceiro e o quarto tipos de classificação tabu são os que impõem as mais fortes restrições ao espaço de procura. O primeiro, quinto e sexto tipos de classificação tabu, aproximadamente do mesmo grau de restrição, não impuseram restrições à pesquisa. De notar que a lista tabu pretende tornar a pesquisa diversificada, proibindo certas soluções do espaço de procura durante L iterações e permitindo que outras zonas possam ser exploradas. Como o exemplo de partição apresentado é de pequena dimensão, alguns tipos de classificação tabu são demasiado restritivos, tornando tabu mais de 50% das alternativas de solução presentes na vizinhança. Ou seja, o tipo de classificação tabu associado ao tamanho da lista tabu L define o grau de diversidade aplicado à pesquisa do espaço de soluções. Ao aplicar todos os tipos de tabu corre-se o risco de reduzir em demasia o tamanho da lista de soluções candidatas. É o que acontece no exemplo em discussão, onde a partir da quarta iteração muitas das iterações apenas dispõem dum deslocamento que não está classificado como tabu[9]. Se todos os deslocamentos ficassem tabu, para seleccionar um deslocamento teria que se recorrer a um critério de aspiração que anulasse a classificação tabu desse deslocamento.

O deslocamento $d(4,2,1)$ permanece tabu durante a segunda e a terceira iterações e torna-se

[9]Deslocamento assinalado com "$\langle$ *permitido* $\rangle$" nas iterações 2 a 4.

Terceira Iteração	
Solução anterior	$s(3,3,2,1,3)$
Nova solução	$s(3,3,2,1,2)$
Lista tabu	$\{\ d(1,1,3)\ ,\ d(4,2,1)\ \}$
Deslocamentos proibidos pela lista tabu	Se $d(4,2,1)$ e $d(1,1,3)$ são tabu tipo 1 $\Rightarrow$ ▶ nº soluções proibidas = 0 Se $d(4,-,-)$ e $d(1,-,-)$ são tabu tipo 2 $\Rightarrow$ ▶ nº soluções proibidas = 4 Se $d(-,2,-)$ e $d(-,1,-)$ são tabu tipo 3 $\Rightarrow$ ▶ nº soluções proibidas = 4 Se $d(-,-,1)$ e $d(-,-,3)$ são tabu tipo 4 $\Rightarrow$ ▶ nº soluções proibidas = 6 Se $d(4,-,1)$ e $d(1,-,3)$ são tabu tipo 5 $\Rightarrow$ ▶ nº soluções proibidas = 0 Se $d(4,2,-)$ e $d(1,1,-)$ são tabu tipo 6 $\Rightarrow$ ▶ nº soluções proibidas = 0
Deslocamento $\rightarrow$ solução vizinha resultante	$d(1,3,1) \rightarrow s(1,3,2,1,3)$ $d(1,3,2) \rightarrow s(2,3,2,1,3)$ $d(2,3,1) \rightarrow s(3,1,2,1,3)$ $d(2,3,2) \rightarrow s(3,2,2,1,3)$ $\langle$ *permitido* $\rangle$ $d(3,2,1) \rightarrow s(3,3,1,1,3)$ $d(3,2,3) \rightarrow s(3,3,3,1,3)$ $d(4,1,2) \rightarrow s(3,3,2,2,3)$ $d(4,1,3) \rightarrow s(3,3,2,3,3)$ $d(5,3,1) \rightarrow s(3,3,2,1,1)$ $d(5,3,2) \rightarrow s(3,3,2,1,2)$ $\langle$ *permitido* $\rangle$
Deslocamento efectuado	$d(5,3,2)$

Tabela 6.3: Exemplo de aplicação da pesquisa tabu na resolução do problema de partição (terceira iteração duma pesquisa).

um deslocamento elegível na quarta iteração. Deste modo, durante duas iterações[10] a característica associada ao deslocamento anterior, *atribuir objecto 4 à partição 1*, não se encontra nos dois deslocamentos seguintes: $d(1,1,3)$ e $d(5,3,2)$. Com a lista tabu não se pretende apenas diversificar a pesquisa, mas também evitar ciclos. Por exemplo, na segunda iteração, a solução actual foi gerada efectuando o deslocamento $d(4,2,1)$ na primeira iteração. Se nesta iteração for seleccionado o deslocamento $d(4,1,2)$, exactamente o deslocamento inverso do que originou a solução actual, é gerada exactamente a mesma solução que a solução inicial da pesquisa, originando um ciclo, já que a pesquisa voltou ao estado imediatamente anterior. De todos os tipos de classificação tabu, apenas o tipo 2 em que são classificados como tabu todos os deslocamentos em que intervém o objecto obj, não importando a partição de origem nem a partição de destino, classificaria como tabu o deslocamento inverso $d(4,1,2)$ evitando assim o funcionamento em ciclo. Assim, a não ser que se utilize uma lista tabu que implemente o tabu tipo 2, é conveniente definir o seguinte tipo de classificação tabu:

7. Classifica-se como tabu o deslocamento inverso daquele que originou a solução actual.

Desta forma, podem combinar-se tipos de classificação tabu que favorecem a diversificação da pesquisa e simultaneamente evitam ciclos.

[10] O tamanho da lista tabu é 2.

Quarta Iteração	
Solução anterior	$s(3,3,2,1,2)$
Nova solução	$s(3,3,1,1,2)$
Lista tabu	$\{\ d(5,3,2)\ ,\ d(1,1,3)\ \}$
Deslocamentos proibidos pela lista tabu	Se $d(1,1,3)$ e $d(5,3,2)$ são tabu tipo 1 $\Rightarrow$ ▶ nº soluções proibidas = 0 Se $d(1,-,-)$ e $d(5,-,-)$ são tabu tipo 2 $\Rightarrow$ ▶ nº soluções proibidas = 4 Se $d(-,1,-)$ e $d(-,3,-)$ são tabu tipo 3 $\Rightarrow$ ▶ nº soluções proibidas = 6 Se $d(-,-,3)$ e $d(-,-,2)$ são tabu tipo 4 $\Rightarrow$ ▶ nº soluções proibidas = 6 Se $d(1,-,3)$ e $d(5,-,2)$ são tabu tipo 5 $\Rightarrow$ ▶ nº soluções proibidas = 0 Se $d(1,1,-)$ e $d(5,3,-)$ são tabu tipo 6 $\Rightarrow$ ▶ nº soluções proibidas = 0
Deslocamento $\rightarrow$ solução vizinha resultante	$d(1,3,1) \rightarrow s(1,3,2,1,2)$ $d(1,3,2) \rightarrow s(2,3,2,1,2)$ $d(2,3,1) \rightarrow s(3,1,2,1,2)$ $d(2,3,2) \rightarrow s(3,2,2,1,2)$ $d(3,2,1) \rightarrow s(3,3,1,1,2)$ $\langle\ permitido\ \rangle$ $d(3,2,3) \rightarrow s(3,3,3,1,2)$ $d(4,1,2) \rightarrow s(3,3,2,2,2)$ $d(4,1,3) \rightarrow s(3,3,2,3,2)$ $d(5,2,1) \rightarrow s(3,3,2,1,1)$ $d(5,2,3) \rightarrow s(3,3,2,1,3)$
Deslocamento efectuado	$d(3,2,1)$

Tabela 6.4: Exemplo de aplicação da pesquisa tabu na resolução do problema de partição (quarta iteração duma pesquisa).

6.6 Implementação do Método de Pesquisa Tabu

Nesta secção resumem-se os principais aspectos relacionados com a implementação do método de pesquisa tabu, realçando-se os princípios em que assenta este método de optimização: listas tabu, historial de soluções visitadas, pesquisa na vizinhança, critérios de aspiração, mecanismos de convergência para soluções óptimas (intensificação) e de fuga a mínimos locais da função de custo (diversificação).

6.6.1 Algoritmo de Pesquisa Tabu

No cerne da pesquisa tabu encontra-se um algoritmo que, de forma iterativa, procura melhorar a solução de partição que lhe é fornecida. É o algoritmo de pesquisa tabu que reúne todos os componentes da pesquisa e controla a sua evolução. O algoritmo implementado inspirou-se no algoritmo de pesquisa tabu que se introduziu na figura 3.4 do capítulo 3 [EKP98a].

As alterações relevantes que o presente trabalho introduziu no algoritmo implementado e que se descreve na figura 6.7, relativamente ao algoritmo apresentado no capítulo 3, são as seguintes:

- em cada iteração, o algoritmo do capítulo 3 procura a nova solução de partição na

algoritmoPesquisaTabu() ≡

Estabelecer a solução de partição inicial → P_{actual}
$P_{melhor} = P_{actual}$, $custo_{melhor} = Fcusto(P_{melhor})$
Iniciar o historial da informação relativa aos deslocamentos e aos objectos deslocados
$N_{iteracoes} = 0$

enquanto ($N_{iteracoes} < LIMITEni$) fazer
  $N_{iteracoesDesdeMelhorSolucao} = 0$
  enquanto (($N_{iteracoesDesdeMelhorSolucao} < LIMITEnidms$) E ($N_{iteracoes} < LIMITEni$)) fazer
    para *cada alternativa de solução* $P_k \in subV(P_{actual})$ fazer
      Calcular a variação de custo → $\Delta C_k = Fcusto(P_k) - Fcusto(P_{actual})$
    fpara

    // Primeira alternativa de deslocamento
    para ($\Delta C_k < 0$) *E* (ΔC_k *por ordem crescente de valor)* fazer
      se ($tabu(P_k) = INVALIDO$) *OU* ($cAspiracaoTabu(P_k) = VERDADE$) então
        $P_{actual} = P_k$
        Saltar para FIM
      fse
    fpara

    // Segunda alternativa de deslocamento
    para *cada alternativa de solução* $P_k \in subV(P_{actual})$ fazer
      $\Delta C'_k = \Delta C_k + bonificacao(P_k)$
    fpara
    para ($\Delta C'_k < 0$) *E* ($\Delta C'_k$ *por ordem crescente de valor)* fazer
      se ($tabu(P_k) = INVALIDO$) então
        $P_{actual} = P_k$
        Saltar para FIM
      fse
    fpara

    // Terceira alternativa de deslocamento
    Definir P_{actual} usando o deslocamento menos tabu, ou o deslocamento menos tabu da subvizinhança $subV(P_{actual})$, ou o deslocamento menos frequente, ou o objecto que permanece na mesma partição há mais iterações ou o deslocamento que no passado apresentou a maior qualidade

FIM: Tornar tabu o deslocamento, o deslocamento inverso e o objecto deslocado, envolvidos na geração de P_{actual}
    Actualizar o historial com a informação relativa ao deslocamento e ao objecto envolvidos na geração de P_{actual}
    Actualizar a validade de todos os deslocamentos tabu e de todos os objectos tabu
    Incrementar o número de iterações na mesma partição ($NIMP$) de todos os objectos
    Incrementar $N_{iteracoesDesdeMelhorSolucao}$
    Incrementar $N_{iteracoes}$
    se ($Fcusto(P_{actual}) < Fcusto(P_{melhor})$) então
      $P_{melhor} = P_{actual}$, $custo_{melhor} = Fcusto(P_{melhor})$
      $N_{iteracoesDesdeMelhorSolucao} = 0$
    fse
  fenquanto
  Estabelecer uma nova solução inicial P_{actual}
  Iniciar o historial da informação relativa aos deslocamentos e aos objectos deslocados
fenquanto

Devolver a melhor solução de partição obtida → P_{melhor}

Figura 6.7: Algoritmo de pesquisa tabu implementado.

vizinhança da solução actual, enquanto o algoritmo implementado pesquisa apenas uma subvizinhança definida pelo conjunto das melhores soluções da vizinhança; deste modo, reduz-se significativamente o tempo de cálculo, mas acrescenta-se dificuldade à pesquisa da solução óptima, especialmente se a selecção das melhores soluções não for a mais correcta;

- ◇ no algoritmo implementado, a segunda alternativa de deslocamento aplica uma bonificação que se mostrou mais eficaz que a bonificação original [EKP98a];
- ◇ para terceira alternativa de deslocamento, o algoritmo do capítulo 3 selecciona o deslocamento cuja validade do tabu está mais perto do fim; no presente algoritmo, pode optar-se por um de vários métodos alternativos para escolha do deslocamento, consoante se deseje reforçar a diversificação ou a intensificação na pesquisa;
- ◇ o algoritmo apresentado no capítulo 3 termina quando se atinge o limite imposto ao número de pesquisas, enquanto o algoritmo implementado termina quando se atinge o limite de iterações executadas na totalidade das pesquisas; ambos os critérios de paragem permitem controlar a relação entre a qualidade da solução final e o tempo de cálculo do algoritmo.

Como a filosofia do algoritmo é evoluir para a melhor solução da vizinhança, em cada iteração o algoritmo dispõe de duas tentativas para atingir esse objectivo.

Na primeira alternativa de evolução, selecciona-se o deslocamento que provoca a maior melhoria no custo da solução de partição actual (P_{actual}) e obedece a uma das condições seguintes: o deslocamento não é tabu ou então é um deslocamento tabu elegível devido a um critério de aspiração. No algoritmo da figura 6.7, a condição $tabu(P_k) = INVALIDO$ significa que o deslocamento que gerou a solução de partição P_k não é tabu (tipo 1) e $cAspiracaoTabu(P_k) = VERDADE$ indica que, embora o deslocamento que gera a solução de partição P_k seja tabu, ele é elegível para ser executado devido a um critério de aspiração. No caso dum critério de aspiração por objectivo, $cAspiracaoTabu(P_k) = VERDADE$ significa que o custo da solução P_k é menor que o custo da melhor solução encontrada até ao momento (P_{melhor}).

Se não houver deslocamentos elegíveis pela primeira alternativa, a selecção recai sobre a solução que se obtém com o deslocamento não tabu que produz o menor aumento do custo da solução de partição. Uma vez que os deslocamentos analisados na segunda alternativa não geram soluções com custo inferior ao da melhor solução visitada na pesquisa actual (P_{actual}), para que esta alternativa seja viável é necessário diminuir o custo das soluções. A redução do custo consegue-se aplicando uma bonificação que favorece o deslocamento dos objectos que estão há mais tempo na mesma partição. A secção 6.6.8 define a bonificação aplicada.

Quando ambas as alternativas falham, a solução para a qual o algoritmo evolui é seleccionada aplicando uma estratégia distinta da que se utiliza nas duas primeiras alternativas. Como existem imensas estratégias passíveis de ser aplicadas nesta fase, optou-se pela implementação dum conjunto de cinco estratégias representativas da globalidade. São as seguintes as estratégias implementadas com o objectivo de definir o deslocamento que gera a terceira alternativa de evolução do algoritmo de pesquisa tabu:

- ◇ o deslocamento seleccionado é o deslocamento tabu que está mais próximo de deixar de ser tabu; no universo teórico de deslocamentos, um deslocamento $d(obj, Porig, Pdest)$ é **elegível** se o objecto obj estiver atribuído à partição $Porig$ e as partições de origem ($Porig$) e de destino ($Pdest$) forem distintas;
- ◇ o deslocamento seleccionado é o deslocamento tabu que está mais próximo de deixar de ser tabu, de entre os deslocamentos que definem a subvizinhança utilizada na presente iteração da pesquisa e em que o objecto a deslocar não é tabu; a estratégia pressupõe que a validade do tabu aplicado aos deslocamentos (tabu tipo 1) é superior à validade do tabu aplicado aos objectos (tabu tipo 2); este pressuposto está de acordo com as recomendações sobre a definição da validade dos tabus: com um tabu mais restritivo (tabu tipo 2) deve usar-se uma validade do tabu inferior à que se usa com um tabu menos restritivo (tabu tipo 1);
- ◇ o deslocamento seleccionado é o deslocamento elegível menos frequente;
- ◇ o deslocamento seleccionado é o deslocamento elegível que envolve o objecto que permanece na mesma partição há mais iterações; a partição de destino do deslocamento é a melhor partição para atribuir o objecto, calculada por uma função em tudo idêntica à função $selectBestObjAssign$ utilizada pelo algoritmo de crescimento de grupos na secção 6.4;
- ◇ o deslocamento seleccionado é o deslocamento elegível que no passado provocou a melhor variação de custo.

No fim de cada iteração do algoritmo actualizam-se as listas tabu, o historial dos deslocamentos efectuados e o historial dos objectos deslocados. Se a solução agora visitada apresentar um custo inferior ao da melhor solução encontrada até ao momento, actualiza-se a melhor solução.

Quando se atinge o limite imposto ao número de iterações sem que a melhor solução seja alterada ($LIMITEnidms$), inicia-se uma nova pesquisa gerando para o efeito uma nova solução inicial a partir da melhor solução de partição. Nesta situação pode optar-se por limpar a informação relativa ao historial de deslocamentos e de objectos deslocados.

O algoritmo termina quando se atinge o limite imposto ao número de iterações, designado por $LIMITEni$. É desejável que o valor de $LIMITEni$ esteja relacionado com a dimensão do problema de partição a resolver. A dimensão do espaço de soluções[11] ($nSolucoes$) dum problema de partição com $nObj$ objectos e $nPart$ partições é dado por

$$nSolucoes = nPart^{nObj} \tag{6.9}$$

Considerando um exemplo de partição com 80 objectos e 5 partições, a dimensão do espaço de soluções é $nSolucoes = 5^{80} \approx 8.27 * 10^{55}$. Daqui se conclui que a utilização dum algoritmo exaustivo, mesmo que somente para validar o algoritmo de pesquisa tabu, não é exequível.

O limite de iterações pode ser definido por uma fracção da dimensão do espaço de soluções, como se indica na equação 6.10.

$$LIMITEni = \lceil \frac{nPart^{nObj}}{K1ni} \rceil \tag{6.10}$$

Contudo, a opção escolhida para definir o limite de iterações não foi a equação 6.10 mas antes a equação 6.11. A segunda definição apresenta uma pequena vantagem relativamente à primeira: o valor definido pela equação 6.11 é mais intuitivo, porque significa que em média cada deslocamento é efectuado $K2ni$ vezes. Esta vantagem é reforçada quando é o projectista a definir a constante $K2ni$. O limite imposto ao número de iterações sem que a melhor solução seja alterada, $LIMITEnidms$, deve ser uma percentagem de $LIMITEni$.

$$LIMITEni = K2ni * nPart * (nPart - 1) * nObj \tag{6.11}$$

6.6.2 Tipos de Tabu

O exemplo descrito nas tabelas 6.1 a 6.4 permitiu concluir que a aplicação de todos os tipos de classificação tabu é muita restritiva para a pesquisa tabu. Deste modo, há que optar por um subconjunto de classificações tabu que reduza a dimensão da vizinhança a pesquisar e não coloque restrições excessivas à pesquisa. Neste subconjunto incluem-se os tipos de tabu cujo efeito restritivo mais se justifica. Concretamente, quando se efectua um determinado deslocamento, é lógico que o deslocamento e o objecto deslocado devem ficar tabu. Por outro lado, para se poderem evitar ciclos é necessário que o deslocamento inverso também fique tabu. O deslocamento e o objecto deslocado são as entidades que mais representam as características/atributos associados à transição da solução de partição actual para a nova solução. Quando se atribui a classificação tabu à partição de origem (tabu tipo 3) ou de destino (tabu tipo 4) do deslocamento, por serem classificações bastante restritivas, impedem

[11] O espaço de soluções também é conhecido por espaço de projecto.

que nas iterações seguintes se efectue o deslocamento de objectos muito pouco relacionados com as características/atributos definidas como sendo tabu na presente iteração. As restrições associadas aos tipos de tabu 5 e 6 são representadas pela classificação tabu aplicada aos objectos deslocados. O deslocamento inverso também é condicionado pela classificação tabu aplicada ao objecto deslocado, mas para que a validade do tabu aplicada aos objectos possa ser diferente daquela que se aplica ao deslocamento inverso, implementou-se a classificação tabu a aplicar aos deslocamentos inversos. Em resumo, os tipos de classificação implementados são os seguintes:

- ◇ **Tabu tipo 1** classifica como tabu o deslocamento dum objecto obj duma partição de origem $Porig$ para uma partição de destino $Pdest$: $d(obj, Porig, Pdest)$;
- ◇ **Tabu tipo 2** classifica como tabu todos os deslocamentos em que intervém o objecto obj: $d(obj, -, -)$;
- ◇ **Tabu tipo 7** classifica como tabu o deslocamento inverso daquele que originou a solução actual: $d(obj, Pdest, Porig)$, quando o último deslocamento efectuado foi $d(obj, Porig, Pdest)$.

Para um problema de partição com uma dimensão próxima de 5 partições e 100 objectos, os valores recomendados para a validade dos tabus implementados são

$$validadeTabu = \begin{cases} 8 - 10 & , \quad \textit{Tabu tipo 1} \\ 5 - 7 & , \quad \textit{Tabu tipo 2} \\ 6 - 8 & , \quad \textit{Tabu tipo 7} \end{cases} \tag{6.12}$$

Estes valores obedecem à regra que recomenda a aplicação duma validade do tabu tanto menor quanto mais restritivo for o tipo de tabu. No caso presente, o tabu mais restritivo é o tabu tipo 2, razão pela qual lhe corresponde a menor validade tabu de entre os três tipos de tabu usados. A selecção do valor da validade do tabu tipo 7 assenta em dois aspectos: (i) como já foi afirmado, a utilização do tabu tipo 7 em conjunto com o tabu tipo 2 só se justifica se os respectivos valores da validade do tabu forem diferentes, (ii) é razoável que a validade do tabu aplicada a um deslocamento inverso seja inferior, pelo menos em uma iteração, à validade do tabu aplicada ao próprio deslocamento.

Nesta fase de desenvolvimento do presente trabalho, não se utilizaram tabus em que a validade variasse dinamicamente ao longo do processo de partição.

6.6.3 Critérios de Aspiração

Na primeira alternativa que o algoritmo de pesquisa tabu dispõe para evoluir da solução actual para a seguinte, o algoritmo pode recorrer ao **critério de aspiração por objectivo** para seleccionar um deslocamento de qualidade que está classificado como tabu. Ou seja, o critério de aspiração por objectivo torna elegível o deslocamento tabu que provoca a maior melhoria no valor devolvido pela função de custo, em relação ao custo da melhor solução visitada até ao momento.

Da explicação anterior resulta que o valor do critério de aspiração por objectivo, associado a um deslocamento, é traduzido pela

> *Diferença entre o custo da solução gerada pelo deslocamento e o custo da melhor solução visitada até ao momento.*

O critério de aspiração por objectivo só selecciona os deslocamentos que geram uma solução com um custo inferior ao custo da melhor solução encontrada até esse momento.

Quando a primeira e a segunda alternativas de evolução do algoritmo de pesquisa tabu não conseguem seleccionar um deslocamento, pode recorrer-se a um **critério de aspiração por defeito** para seleccionar um deslocamento de entre os que são tabu. É o que acontece na primeira e segunda estratégias disponíveis na terceira alternativa de evolução do algoritmo, em que se selecciona (i) o deslocamento tabu que está mais próximo de deixar de ser tabu ou (ii) o deslocamento tabu que está mais próximo de deixar de ser tabu, de entre os deslocamentos que definem a subvizinhança da iteração presente.

6.6.4 Estrutura de Memória

Na estrutura de memória implementada regista-se o historial de deslocamentos efectuados e o historial de objectos deslocados. A informação guardada inclui componentes que se constituem como uma memória de curta duração, uma memória de longa duração e uma memória de duração intermédia. A informação guardada no historial de deslocamentos e o tipo de memória que lhe está associado é apresentada na tabela 6.5. A informação equivalente para o historial de objectos deslocados encontra-se na tabela 6.6.

Informação	***Tipo de Memória***
Partição de origem	$--$
Partição de destino	$--$
Validade do tabu (tipo 1,7)	Memória de curta duração
Frequência de execução	Memória de longa duração
Variação de custo (qualidade)	Memória de duração intermédia

Tabela 6.5: Informação relativa a cada deslocamento guardada no historial de deslocamentos.

Informação	***Tipo de Memória***
Validade do tabu (tipo 2)	Memória de curta duração
Frequência de deslocamento	Memória de longa duração
Número de iterações na mesma partição	Memória de longa duração
Variação de custo (qualidade)	Memória de duração intermédia

Tabela 6.6: Informação relativa a cada objecto guardada no historial de objectos deslocados.

6.6.5 Estrutura de Vizinhança

A estrutura de vizinhança implementada é do tipo **simples**. De acordo com esta estrutura, a vizinhança duma solução é definida pelo conjunto de todas as alternativas de solução que se obtêm deslocando um e um só objecto para uma partição distinta daquela que ocupa na solução actual. Para um problema de partição com $nObj$ objectos e $nPart$ partições, o tamanho da vizinhança é $nObj * (nPart - 1)$. Exceptuando a situação em que se inicia uma nova pesquisa, a evolução do algoritmo de pesquisa tabu faz-se sempre para uma solução da vizinhança da solução actual.

Uma estrutura de vizinhança **complexa**, em que se permite que uma iteração efectue uma série de deslocamentos e só depois é que o custo da nova solução é calculado, não foi implementada. A razão prende-se essencialmente com o aumento drástico que este tipo de vizinhança traria ao tempo de cálculo do algoritmo. Para que o efeito da vizinhança complexa seja relevante, a série de deslocamentos deve constar de pelo menos 3 deslocamentos. Num problema de partição com 4 objectos e 3 partições, quando se passa duma vizinhança simples para uma vizinhança complexa o tamanho da vizinhança aumenta de 8 para 32 alternativas, em que uma nova solução é obtida da solução actual após efectuar 3 deslocamentos com objectos distintos (tabela 6.7). O tamanho da vizinhança complexa não é maior porque a ordem dos deslocamentos que conduzem a uma nova solução é irrelevante. Deste modo, todas as combinação de deslocamentos que envolvem os mesmos objectos geram a mesma solução na vizinhança complexa.

Numa vizinhança complexa genérica, em que cada iteração efectua uma série de $nDesloc$ deslocamentos, o número de alternativas que compõem a vizinhança é definido pela equação 6.13, o que constitui um valor muito superior ao da vizinhança simples ($nObj * (nPart - 1)$).

$$
\begin{aligned}
tamVizinhanca &= (nPart-1)^{nDesloc} * \mathcal{C}_{nDesloc}^{nObj} = \\
&= (nPart-1)^{nDesloc} * \frac{nObj\ !}{nDesloc\ !\ *\ (nObj - nDesloc)\ !} = \\
&= (nPart-1)^{nDesloc} * \frac{nObj * (nObj-1) * ... * (nObj - nDesloc + 1)}{nDesloc * (nDesloc-1) * ... * 3 * 2 * 1}
\end{aligned}
\tag{6.13}
$$

em que $nDesloc$ é o número de deslocamentos efectuados em cada iteração da pesquisa, $nObj$

é o número de objectos do sistema e $nPart$ é o número de partições.

Dimensão do problema		$nObj = 4$ (número de objectos) $nPart = 3$ (número de partições)
Estrutura de vizinhança		Efectuar 3 ($nDesloc$) deslocamentos com 3 objectos distintos
Tamanho da vizinhança		32 alternativas
Solução actual		$s(2,3,1,3)$
Série de deslocamentos	1	$\{ d(1,2,1) , d(2,3,1) , d(3,1,2) \}$
	2	$\{ , , d(3,1,3) \}$
	3	$\{ , , d(4,3,1) \}$
	4	$\{ , , d(4,3,2) \}$
	5	$\{ , d(2,3,2) , d(3,1,2) \}$
	6	$\{ , , d(3,1,3) \}$
	7	$\{ , , d(4,3,1) \}$
	8	$\{ , , d(4,3,2) \}$
	9	$\{ , d(3,1,2) , d(4,3,1) \}$
	10	$\{ , , d(4,3,2) \}$
	11	$\{ , d(3,1,3) , d(4,3,1) \}$
	12	$\{ , , d(4,3,2) \}$
	13	$\{ d(1,2,3) , d(2,3,1) , d(3,1,2) \}$
	14	$\{ , , d(3,1,3) \}$
	15	$\{ , , d(4,3,1) \}$
	16	$\{ , , d(4,3,2) \}$
	17	$\{ , d(2,3,2) , d(3,1,2) \}$
	18	$\{ , , d(3,1,3) \}$
	19	$\{ , , d(4,3,1) \}$
	20	$\{ , , d(4,3,2) \}$
	21	$\{ , d(3,1,2) , d(4,3,1) \}$
	22	$\{ , , d(4,3,2) \}$
	23	$\{ , d(3,1,3) , d(4,3,1) \}$
	24	$\{ , , d(4,3,2) \}$
	25	$\{ d(2,3,1) , d(3,1,2) , d(4,3,1) \}$
	26	$\{ , , d(4,3,2) \}$
	27	$\{ , d(3,1,3) , d(4,3,1) \}$
	28	$\{ , , d(4,3,2) \}$
	29	$\{ d(2,3,2) , d(3,1,2) , d(4,3,1) \}$
	30	$\{ , , d(4,3,2) \}$
	31	$\{ , d(3,1,3) , d(4,3,1) \}$
	32	$\{ , , d(4,3,2) \}$

Tabela 6.7: Exemplo que mostra as séries de deslocamentos que se podem efectuar a partir duma solução de partição, quando se usa uma vizinhança complexa.

Por exemplo, se o número de objectos do sistema for 80, o número de partições 5 e cada nova solução de partição for obtida com uma série de 3 deslocamentos, o número de alternativas na vizinhança complexa é $\frac{4^3*80*79*78}{3*2} = 5258240$. Este valor é muito superior às 320 $(80 * (5-1))$ alternativas presentes na vizinhança simples e às 960 alternativas duma série de 3 deslocamentos efectuados com uma vizinhança simples.

6.6.6 Lista de Soluções Candidatas

Relativamente à definição da subvizinhança, ou lista de soluções candidatas, a aplicar em cada iteração duma pesquisa, resta apenas sintetizar o que foi sendo apresentado ao longo do capítulo.

A subvizinhança utilizada é um subconjunto da vizinhança da solução actual, com um ta-

manho fixo ao longo de toda a pesquisa e igual ao número de objectos do sistema ($nObj$). Ou seja, os $(nPart - 1)$ deslocamentos por objecto que definem a vizinhança são reduzidos para apenas um por objecto na subvizinhança. O subconjunto de deslocamentos que define a subvizinhança é constituído pelo melhor deslocamento de cada objecto do sistema, sendo este deslocamento calculado pela função $selectBestObjMove$. Com base na proximidade entre o objecto e o conjunto de objectos atribuído a cada partição, distinta daquela a que o objecto está atribuído, a função calcula a melhor partição para atribuir esse objecto. Esta função é em tudo idêntica à função $selectBestObjAssign$ do algoritmo de crescimento de grupos. A única diferença significativa entre as duas funções reside no tipo de estimativa que usam para o espaço ocupado pelas partições de *hardware*. A estimativa do espaço serve para impedir que se ultrapassem os recursos disponíveis num componente quando se lhe atribui um objecto. Enquanto a função do algoritmo construtivo aplica uma estimativa simples, que considera apenas o espaço ocupado por variáveis e estados programa, no processo de partição iterativo está disponível uma estimativa precisa para o espaço ocupado pelas partições de *hardware*.

No algoritmo da figura 6.7, a subvizinhança da solução de partição actual é calculada no início de cada iteração e é designada por $subV(P_{actual})$.

6.6.7 Construção duma Nova Solução Inicial

Um dos trunfos do algoritmo de pesquisa tabu resulta de ele efectuar várias pesquisas, cada uma com uma solução inicial distinta. Esta estratégia de diversificação aumenta a capacidade para pesquisar zonas não exploradas do espaço de soluções, aumentando assim a probabilidade de encontrar a solução de partição óptima. Por outro lado, em muitas situações interessa que a nova solução inicial preserve as melhores características da melhor solução encontrada, de modo a aumentar as possibilidades de convergência para a solução óptima. Ao construir uma nova solução inicial, modificando significativamente a vizinhança, consegue visitar-se soluções que seria impossível atingir com uma série de deslocamentos simples efectuados no decorrer duma pesquisa.

O método aplicado na construção da solução inicial de cada pesquisa resulta da conciliação das duas estratégias apresentadas: a nova solução inicial duma pesquisa deriva da melhor solução de partição encontrada (intensificação), alterando a atribuição duma percentagem significativa de objectos, seleccionados de acordo com a memória de longa duração (diversificação). Normalmente efectuam-se os deslocamentos menos frequentes, mas ao fim de um conjunto de pesquisas sem melhoria da solução actual, pode deslocar-se os objectos menos vezes deslocados para uma partição seleccionada de forma aleatória. A selecção aleatória reforça a diversificação na pesquisa. Como a percentagem de objectos deslocados de partição é um parâmetro do algoritmo, é possível controlar a relação entre a intensificação e a diversificação

aplicadas na construção da nova solução inicial. Para valor de referência da percentagem de objectos a deslocar considera-se 20%.

A construção duma solução inicial acontece no início duma nova pesquisa, ou seja, após o algoritmo da figura 6.7 ter atingido o seguinte critério de paragem:

$$N_{iteracoesDesdeMelhorSolucao} = LIMITEnidms.$$

6.6.8 Estratégias de Diversificação e Intensificação

As estratégias de diversificação apoiam-se na seguinte informação:

- o número de vezes que um deslocamento foi efectuado;
- o número de iterações que um objecto permaneceu na última partição a que foi atribuído.

Por seu lado, as estratégias de intensificação utilizam a seguinte informação:

- a melhor variação de custo gerada por um deslocamento.

Como foi referido, o objectivo das estratégias de diversificação é evitar mínimos locais durante as pesquisas, enquanto as estratégias de intensificação visam atingir soluções de qualidade. Considerando o algoritmo de pesquisa tabu apresentado na figura 6.7 pode enumerar-se os pontos em que estas estratégias se aplicam.

Construir a lista de soluções candidatas com as soluções de mais qualidade presentes na vizinhança da solução actual, obedece a uma estratégia de intensificação. No entanto, a qualidade das soluções não se baseia em qualquer informação do historial da pesquisa, já que a solução vizinha escolhida para deslocar cada objecto é definida pelo deslocamento em que ocorre a máxima proximidade entre o objecto a deslocar e os objectos atribuídos à partição de destino do deslocamento. Contudo, a maioria das estratégias de diversificação e de intensificação utiliza de algum modo a informação do historial da pesquisa, a qual está guardada na memória de duração intermédia, quando se trata de estratégias de intensificação, e de longa duração, quando se trata de estratégias de diversificação.

A diversificação, possível de ser incorporada na definição de subvizinhança, através da regra que indica que as soluções não examinadas numa iteração sejam examinadas nas iterações seguintes, não foi implementada.

A informação relativa à frequência com que um deslocamento foi efectuado, é utilizada na segunda alternativa de evolução do algoritmo da figura 6.7, onde se aplica uma bonificação baseada no número de iterações que um objecto permaneceu na última partição a que foi

atribuído ($NIMP$). Deste modo, beneficia-se pouco o deslocamento de objectos que estejam constantemente a ser efectuados, porque apresentam um valor baixo no parâmetro $NIMP$. Pelo contrário, beneficia-se fortemente os objectos que não são deslocados regularmente, uma vez que possuem um $NIMP$ elevado. A utilização desta bonificação faz com que a pesquisa seja orientada para zonas menos exploradas, diversificando a pesquisa. A bonificação aplicada é definida pela equação 6.14.

$$bonificacao(P_k) = -\frac{NIMP_k}{nObj} \tag{6.14}$$

em que

- $\boldsymbol{NIMP_k}$ designa o número de iterações que um objecto obj, que ao ser deslocado gerou a solução de partição P_k, permaneceu na última partição a que foi atribuído;
- $\boldsymbol{nObj}$ é o número de objectos da descrição do sistema.

A informação relativa ao número de vezes que um deslocamento foi efectuado é utilizada na construção da solução inicial das pesquisas. Como foi afirmado anteriormente, para construir uma nova solução inicial parte-se da melhor solução encontrada na pesquisa (intensificação), que é um mínimo local ou absoluto da função de custo, e desloca-se uma percentagem dos objectos menos vezes deslocados (diversificação). Se a melhor solução é um mínimo local, com a estratégia de diversificação pretende-se que a nova pesquisa consiga escapar a esse mínimo e ao mesmo tempo, com a estratégia de intensificação pretende-se preservar a qualidade que a melhor solução possui. Ao deslocar os objectos menos vezes deslocados pretende-se pesquisar zonas do espaço de soluções ainda não exploradas.

Os métodos de selecção do deslocamento que gera a terceira alternativa de evolução do algoritmo também produzem efeitos de diversificação e de intensificação:

- ◇ teoricamente, seleccionar o deslocamento tabu que está mais próximo de deixar de ser tabu não produz nenhum efeito de diversificação ou de intensificação;
- ◇ ao seleccionar o deslocamento tabu que está mais próximo de deixar de ser tabu, de entre os deslocamentos que definem a subvizinhança, gera-se um efeito de intensificação porque a subvizinhança é constituída por deslocamentos de qualidade;
- ◇ seleccionar o deslocamento menos frequente ou o deslocamento do objecto que permanece na mesma partição há mais iterações tem um efeito de diversificação;
- ◇ quando se escolhe o deslocamento que no passado provocou a melhor variação de custo, produz-se um efeito de intensificação porque o deslocamento efectuado é de qualidade.

A informação sobre a variação de custo resultante dum deslocamento pode ser utilizada na construção da nova solução inicial. Neste caso, o critério de selecção dos objectos a deslocar não é a frequência de deslocamento, mas sim a qualidade dos deslocamentos. Para construir uma solução inicial pode ainda combinar-se a informação relativa à frequência dos deslocamentos com a informação sobre a qualidade desses deslocamentos.

6.7 Função de Custo para o Método de Pesquisa Tabu

A função de custo a aplicar no processo de partição iterativo foi adaptada da função de custo da abordagem SpecSyn porque, tal como se afirmou na secção 3.9, é uma função que considera como óptima uma alternativa de partição que respeita os condicionalismos impostos pela arquitectura alvo e atinge os requisitos exigidos, e não uma função que considere como óptima uma solução que ocupe o menor espaço em *hardware* possível e/ou atinge o melhor desempenho possível. Para isso a função dispõe dum termo, por cada condicionalismo ou requisito, cujo valor é proporcional ao grau de desrespeito desse condicionalismo ou requisito verificado na alternativa de partição.

As alterações significativas introduzidas na função de custo da abordagem SpecSyn são agora discutidas.

Em primeiro lugar, incluiu-se um termo que depende do espaço ocupado pela unidade de controlo das partições de *hardware*, além do termo relativo ao espaço ocupado pelo caminho de dados. A razão de ser deste termo está em que a unidade de controlo duma partição de *hardware* é implementada com um componente (CPLD) distinto do caminho de dados da mesma partição. O termo introduzido quantifica o grau de desrespeito do espaço ocupado pela unidade de controlo duma partição de *hardware*, em relação ao condicionalismo imposto pela arquitectura alvo (espaço disponível num CPLD).

Em segundo lugar, os condicionalismos impostos ao número de componentes e à largura da ligação entre componentes são considerados durante a construção de soluções de partição exequíveis, através da modelação da comunicação entre partições, mas não aparecem explicitamente na função de custo. Incluir estes termos na função de custo, só se justificava se fosse desejável que o algoritmo de partição ponderasse alternativas com um número de partições além do permitido ou com comunicação por canais de largura superior ao permitido. Testar soluções com um número de partições além do permitido não traz vantagens, apenas alarga o espaço de projecto para zonas afastadas da solução de partição óptima e aumenta o tempo de cálculo. Testar soluções em que a comunicação se faz por canais de largura superior ao permitido, só faria sentido se a largura dos canais de comunicação não fosse restringida. Tendo-se constado que a comunicação por canais sem restrições de largura complicava drasticamente o

modelo de estimação da comunicação, optou-se por uma situação em que a largura dos canais só pode assumir um valor seleccionado dum conjunto de valores compatíveis com as ligações físicas da arquitectura alvo. Deste modo, a largura dos canais de comunicação nunca ultrapassa os condicionalismos impostos pela arquitectura alvo e a inclusão dum termo relativo ao respeito pela largura dos canais de comunicação (ou número de pinos) não se justifica.

Em terceiro lugar, o termo adoptado para medir o grau de incumprimento do desempenho funciona com o desempenho ao nível do sistema, enquanto o termo da abordagem SpecSyn mede o grau de incumprimento do desempenho dos objectos do sistema em relação ao valor que lhe é exigido pelo requisito do sistema. Esta opção apresenta inconvenientes relativamente à opção adoptada neste trabalho: (i) o requisito associado a cada objecto não é habitualmente conhecido, (ii) o que é relevante para a qualidade duma solução é o desempenho global do sistema, não o desempenho parcelar ao nível do objecto e (iii) o tempo de cálculo exigido pela função de custo é superior.

A função de custo desenvolvida é definida pela equação 6.15.

$$Fcusto(H, S, Cons, Req) \; = \sum_{M_i \in M} K_i * f_i(M_i) \tag{6.15}$$

onde

- $\boldsymbol{H}$ é o conjunto das partições de *hardware*;
- $\boldsymbol{S}$ é o conjunto das partições de *software*;
- $\boldsymbol{Cons}$ é o conjunto dos condicionalismos do projecto;
- $\boldsymbol{Req}$ é o conjunto dos requisitos exigidos ao sistema;
- $\boldsymbol{M}$ é o conjunto das métricas $\{M_i\}$;
- $\boldsymbol{K_i}$ é o coeficiente aplicado à métrica M_i;
- $\boldsymbol{f_i(M_i)}$ é a função que define o contributo da métrica M_i para a função de custo.

Concretamente $M = \{M_{areaCD}, M_{areaUC}, M_{texec}\}$, onde M_{areaCD} e M_{areaUC} designam o espaço ocupado nos componentes de *hardware* e M_{texec} representa o desempenho do sistema. Enquanto os condicionalismos do projecto $Cons = \{C_{areaCD}, C_{areaUC}\}$, aplicados ao espaço ocupado em *hardware*, definem o limite imposto pela arquitectura alvo ao espaço utilizável, R_{texec} define o desempenho exigido ao sistema. O desempenho pode surgir sob a forma de tempo de execução, latência ou débito. Desenvolvendo a função de custo obtém-se a seguinte expressão

$$Fcusto(H,S,Cons,Req) = K_{areaCD} * f_{areaCD}(M_{areaCD},C_{areaCD}) +$$
$$+ K_{areaUC} * f_{areaUC}(M_{areaUC},C_{areaUC}) + K_{texec} * f_{texec}(M_{texec},R_{texec}) \quad (6.16)$$

O contributo da métrica M_{areaCD} (M_{areaUC}) para a função de custo define a penalidade aplicada às soluções que excedem o condicionalismo C_{areaCD} (C_{areaUC}) imposto pela arquitectura alvo. O condicionalismo C_{areaCD} ou C_{areaUC} representa a quantidade de recursos disponível num componente de *hardware* do tipo FPGA ou CPLD[12], respectivamente. Estes contributos são definidos por

$$f_{areaCD}(M_{areaCD},C_{areaCD}) = \sum_{p_i \in H} f_{areaCD}(M^i_{areaCD},C^i_{areaCD}) =$$
$$= \sum_{p_i \in H} MAX\ [\ excess(M^i_{areaCD},C^i_{areaCD}),0\ ] \quad (6.17)$$

$$f_{areaUC}(M_{areaUC},C_{areaUC}) = \sum_{p_i \in H} f_{areaUC}(M^i_{areaUC},C^i_{areaUC}) =$$
$$= \sum_{p_i \in H} MAX\ [\ excess(M^i_{areaUC},C^i_{areaUC}),0\ ] \quad (6.18)$$

em que

- $\boldsymbol{excess(M^i_{areaCD},C^i_{areaCD})}$ é definido por

$$excess(M^i_{areaCD},C^i_{areaCD}) = \frac{M^i_{areaCD} - C^i_{areaCD}}{C^i_{areaCD}} \quad (6.19)$$

- $\boldsymbol{excess(M^i_{areaUC},C^i_{areaUC})}$ é definido por

$$excess(M^i_{areaUC},C^i_{areaUC}) = \frac{M^i_{areaUC} - C^i_{areaUC}}{C^i_{areaUC}} \quad (6.20)$$

- as estimativas do espaço são obtidas ao nível da partição;
- na plataforma alvo EDgAR-2 os pares de componentes (FPGA,CPLD) são todos iguais, logo os valores de C^i_{areaCD} e C^i_{areaUC} são iguais para todas as partições de *hardware* p_i.

[12] C_{areaCD} e C_{areaUC} estão definidos na tabela 5.1.

O contributo da métrica M_{texec} para a função de custo define a penalidade aplicada às soluções que não atingem o requisito de desempenho R_{texec} exigido ao sistema (equação 6.21). As estimativas do desempenho são obtidas ao nível do sistema e não da partição.

$$f_{texec}(M_{texec}, R_{texec}) = MAX[excess(M_{texec}, R_{texec}), 0] = MAX(\frac{M_{texec} - R_{texec}}{R_{texec}}, 0) \quad (6.21)$$

A estimação das métricas envolvidas na função de custo é desenvolvida no capítulo 7.

6.8 Resumo e Conclusões

Com o objectivo de conceber soluções de partição que atingem um desempenho pré-definido com os recursos disponíveis na arquitectura alvo, foi proposta uma metodologia de partição do tipo funcional, inter-componentes e automática. A organização da metodologia inclui um módulo de conversão de modelos, os algoritmos de partição construtivo e iterativo, as funções de proximidade e de custo e os estimadores de métricas.

O desenvolvimento dum meta-modelo novo, a aplicar na representação interna dos sistemas durante o processo de partição, deveu-se à necessidade de automatizar o processo de partição e à disponibilidade duma aplicação com interface gráfica que permite editar grafos com características próximas das desejadas para esta representação. Uma das limitações da implementação actual do PSMfg é a ausência de hierarquia estrutural, uma facilidade inexistente na aplicação que suporta o manuseamento automático dos grafos.

No algoritmo de crescimento de grupos, o objecto semente das partições pode ser seleccionado aleatoriamente, atribuído pelo utilizador ou definido com base no número de ligações com os objectos por atribuir. Os restantes objectos são atribuídos à partição que apresente a maior proximidade com estes objectos, ou seja, à partição que apresente a intensidade de comunicação mais elevada, em que o tempo de computação do estado programa seja menor, em que o espaço disponível seja maior. Ao contributo da intensidade de comunicação para a definição de proximidade é aplicado um peso superior ao das outras métricas.

No método de pesquisa tabu a validade dos tabu condiciona de forma determinante a evolução da pesquisa. Por isso, quanto mais restritivo for um tipo de tabu, menor deve ser a sua validade. Tal como era previsto, o exemplo apresentado mostrou que o grau de restrição dos tipos de tabu aumenta desde o tabu tipo 1, passando pelos tabus tipo 5 e 6, pelo tabu tipo 2 até aos tabus tipo 3 e 4. O tabu tipo 7 coloca o mesmo grau de restrição que o tabu tipo 1. Dado que a aplicação de todos os tipos de classificação tabu impõe demasiadas restrições à pesquisa, implementou-se apenas um subconjunto que classifica como sendo tabu o deslocamento (tipo 1), o objecto deslocado (tipo 2) e o deslocamento inverso (tipo 7).

Relativamente ao algoritmo de pesquisa tabu utilizado como referência, o algoritmo implementado pesquisa apenas numa subvizinhança da solução actual, dispõe de mais estratégias de evolução quando não há soluções de qualidade elegíveis e aplica uma bonificação mais eficaz quando nenhum deslocamento melhora o custo da solução actual. Ao pesquisar apenas numa subvizinhança, reduz-se o tempo de cálculo por iteração dum factor próximo do número de partições, mas a exploração do espaço de projecto é mais incompleta. Como a subvizinhança é constituída pelas melhores soluções, introduz-se um elemento de intensificação na pesquisa.

Os resultados experimentais mostraram que ao iniciar uma pesquisa convém limpar a informação relativa ao historial de deslocamentos e de objectos deslocados. Não limpar o historial induziria um efeito de dispersão prejudicial à convergência para a solução óptima, porque a bonificação usada na segunda alternativa de deslocamento, proporcional ao número de iterações em que um objecto não é deslocado, iria seleccionar com regularidade qualquer objecto.

O método PT implementado dispõe de dois tipos de critério de aspiração, por objectivo e por defeito, e guarda o historial de deslocamentos efectuados e de objectos deslocados. Implementou-se uma estrutura de vizinhança simples porque uma estrutura de vizinhança complexa iria aumentar drasticamente o tempo de cálculo. Uma vizinhança complexa favorece a diversificação na pesquisa, que tanto significa uma capacidade acrescida para fugir a mínimos locais como uma dificuldade acrescida para convergir para a solução de partição óptima.

Quando se inicia uma pesquisa com uma nova solução inicial modifica-se significativamente a vizinhança, o que permite visitar soluções impossíveis de atingir com uma série de deslocamentos simples efectuados durante uma pesquisa. Obter a nova solução inicial a partir da melhor solução visitada e deslocando 20% dos objectos, correspondentes aos deslocamentos menos frequentes, constitui uma boa relação de compromisso entre intensificação e diversificação. A possibilidade de atribuir os objectos a deslocar a uma partição seleccionada aleatoriamente, constitui um reforço da capacidade de fuga a mínimos locais.

O número de iterações efectuadas sem haver melhoria da solução de partição, o critério de paragem duma pesquisa, não deve ser demasiado alto para evitar que se desperdicem iterações em torno dum mínimo local, nem demasiado baixo para aumentar a possibilidade de convergência para um mínimo local ou absoluto da função de custo.

Relativamente à função de custo utilizada como referência, efectuaram-se as seguintes alterações: (i) introduziu-se um termo relacionado com o espaço ocupado pela unidade de controlo duma partição de *hardware*, (ii) os condicionalismos impostos ao número de componentes e à largura da ligação entre componentes não aparecem explicitamente na função de custo, por se considerar que estes condicionalismos nunca são desrespeitados e (iii) o desempenho é tratado ao nível do sistema e não do objecto.

Em conclusão, implementou-se um método de pesquisa tabu enriquecido com um conjunto de mecanismos que favorecem a convergência para a solução de partição óptima e que ajudam a evitar mínimos locais da função de custo. A contrapartida é que o método exige muita experimentação para afinar os parâmetros que controlam a influência destes mecanismos.

Capítulo 7

Estimação Aplicada na Metodologia de Partição Proposta

Sumário

Para estimar as métricas da função de custo a que recorre o algoritmo de partição iterativo utiliza-se um modelo de software para o processador, um modelo de hardware para as FPGAs e os CPLDs e um modelo de comunicação para os recursos de interligação. Apresenta-se um modelo de software simples, específico duma família de processadores e que quantifica as optimizações de código num factor obtido por simulação e um modelo melhorado que considera as optimizações sob a forma dum factor obtido estaticamente. A estimação de métricas é incremental e funciona em dois níveis de abstracção: no nível do sistema e no nível do estado programa. No nível de abstracção do sistema obtêm-se estimativas para o espaço ocupado pelas partições de hardware e para o desempenho do sistema. No nível de abstracção do estado programa, obtêm-se estimativas para métricas relativas aos objectos do modelo PSMfg da solução de partição. O espaço ocupado por uma partição de hardware inclui o caminho de dados e a unidade de controlo da partição. Para o espaço do caminho de dados contribuem as unidades funcionais, os elementos de armazenamento, os elementos de interligação e os recursos de interface; enquanto o espaço da unidade de controlo recebe o contributo do registo de estado, da lógica de controlo e da lógica do próximo estado. O desempenho do sistema é função do tempo de computação dos estados programa, da partição a que os objectos foram atribuídos, do tempo de comunicação entre estados programa e variáveis, da frequência de execução dos estados programa, do sincronismo explícito e implícito entre estados programa e do escalonamento do sistema. Para estimar o desempenho dum sistema, definiram-se métodos de estimação do tempo de execução dos construtores permitidos pela modelação dos sistemas, em especial dos ciclos de espera. Para estimar o tempo de execução dum sistema utiliza-se uma função que calcula o tempo de execução nos fluxos dum grafo e o conhecimento sobre os pontos de paragem, os pontos de sincronismo e os fluxos do grafo. No final, quantificam-se as métricas de hardware de baixo nível a que a estimação ao nível do estado programa e do sistema recorrem.

Conteúdo

7.1 Introdução

Para completar a metodologia de partição proposta descreve-se a estimação das métricas envolvidas na função de custo do processo de partição iterativo. A abordagem seguida na estimação de métricas é condicionada pelos seguintes aspectos:

- ◇ procurou conciliar-se a obtenção de estimativas precisas com a redução do tempo de cálculo;
- ◇ deve ter-se em consideração o enquadramento definido pelo processo de partição iterativo, uma vez que este condiciona a relação entre a qualidade das estimativas e o tempo de cálculo; embora a minimização do tempo de cálculo não seja o objectivo maior, é necessário optimizar os estimadores para que estes possam ser aplicados em problemas onde se analisam milhares de alternativas de partição;
- ◇ sempre que se justifique, a estimação acompanha a estratégia seguida pelas ferramentas de síntese a utilizar na implementação da solução de partição; por exemplo, na estimação não se considera a possibilidade de partilhar uma unidade funcional por parte de diferentes operações dado ser esta a opção seguida pela ferramenta de síntese;
- ◇ a metodologia deve adaptar-se ao tipo de modelação utilizado na descrição dos sistemas durante o processo de partição; a representação interna ao processo de partição foi introduzida na secção 6.2;
- ◇ a metodologia deve operar no nível de abstracção mais adequado para a estimação das métricas; deste modo, optou-se por dividir o processo em dois níveis: estimação ao nível do estado programa (mais baixo) e ao nível do sistema (mais alto); uma estimação a dois níveis de abstracção facilita o controlo da relação entre a precisão, ou fidelidade, das estimativas e o tempo de cálculo.

No nível de abstracção do sistema obtêm-se estimativas para o espaço ocupado pelas partições de *hardware* e para o desempenho do sistema, sendo as estimativas recalculadas em cada nova iteração do processo de partição. No nível de abstracção do estado programa, obtêm-se estimativas precisas para métricas relativas aos objectos do modelo PSMfg do sistema. Como estas métricas quantificam características dos objectos que permanecem inalteradas ao longo do processo de partição, não necessitam de ser recalculadas. O facto de as estimativas só serem calculadas uma vez permite que se funcione com um modelo de estimação mais detalhado. Juntando a explicação anterior com a constatação de que os cálculos envolvidos nestas estimativas representam uma parcela significativa dos cálculos necessários à obtenção do custo duma solução de partição, permite concluir que a estimação a dois níveis pode

aplicar um modelo de estimação mais detalhado ao nível do sistema, sem que o tempo de cálculo global seja superior ao exigido pela estimação efectuada num único nível.

Consegue-se uma diminuição adicional no tempo de cálculo aplicando na estimação ao nível do sistema um mecanismo de **estimação incremental**, que consiste em não refazer todos os cálculos de iteração para iteração, mas apenas os que estão relacionados com as alterações efectuadas na última iteração. Na maioria das situações, duma iteração para a seguinte as alterações na solução de partição são diminutas, dado que apenas um objecto é deslocado duma partição para outra. A redução do tempo de cálculo não é proporcional ao quociente entre o número total de objectos e o número de objectos deslocados por iteração, porque o método incremental é mais complexo que os métodos que refazem totalmente as estimativas.

7.2 Modelação de Recursos

Os recursos disponíveis para a implementação de sistemas, e para os quais se exigem modelos, são classificados em recursos de *software* (processador), de *hardware* (FPGAs e CPLDs) e de comunicação (interligação).

Para modelar o comportamento do *software* é preciso definir métricas, como o tempo de execução ou o espaço, e indicar como as calcular. No caso presente, o **modelo de *software*** caracteriza o desempenho da parte do sistema que é baseada em processador. Optou-se por um modelo simples, específico da família de processadores Intel Pentium. O modelo é concretizado num ficheiro tecnológico com tempos de execução para operações elementares, parametrizado através da frequência de relógio do processador e dum factor de optimização relativamente ao tempo de execução. Considera-se que além das tarefas que implementam o sistema não existem outras tarefas a competir pelos recursos do único processador.

O tempo de execução em *software*, para as operações elementares mais comuns em HLLs, encontra-se na tabela 7.1. Os valores da tabela foram determinados pela combinação de duas estratégias distintas: (i) executando 10^9 vezes cada operação, sem optimização de código e com a *cache* activa, se for subtraído ao tempo de execução total o tempo de execução das instruções de controlo do ciclo obtém-se o tempo total de execução da operação e (ii) contabilizando o tempo de execução das várias instruções *assembly* com que cada operação é codificada [Bre95]. Os tempos de execução são válidos para um sistema com um processador Intel Pentium, memória SDRAM a 66 MHz, sistema operativo *Windows* 2000 e considerando que os inteiros ocupam 32 bits e os reais 64 bits.

Como se afirmou no capítulo 4, recorrer a um simples factor λ_T para contabilizar as optimizações resultantes dos níveis de *pipeline*, da superescalaridade, dos esquemas de previsão de saltos ou dos esquemas de gestão da hierarquia de memória do processador, é uma opção

nitidamente limitada. Contudo, quando se dispõe dum modelo executável para o sistema, pode calcular-se um factor de optimização específico para o sistema em causa, o qual permite obter estimativas de qualidade aceitável. Este método incorre no erro de calcular um valor de λ_T a partir duma solução totalmente em *software* e de o aplicar em soluções mistas de *software* e *hardware*. Digamos que para uma aplicação em que o desempenho do sistema serve para comparar as soluções de partição e não para medir o seu desempenho absoluto, o factor λ_T constitui uma boa solução de compromisso entre a qualidade das estimativas e o tempo de cálculo (e a simplicidade de implementação). Nos casos em que não se dispõe dum modelo executável, utiliza-se como factor de optimização o valor médio obtido no conjunto de sistemas estudados.

Tipo de Operação	$\boldsymbol{T_{execSW}}$
Operações aritméticas e lógicas †	
operação lógica (E, OU) entre duas variáveis inteiras	$3 * T_{relogioCPU}$
deslocar uma variável inteira à esquerda/direita	$4 * T_{relogioCPU}$
adicionar/subtrair/comparar duas variáveis inteiras	$3 * T_{relogioCPU}$
adicionar/subtrair/comparar duas variáveis reais	$15 * T_{relogioCPU}$
multiplicar duas variáveis inteiras	$12 * T_{relogioCPU}$
multiplicar duas variáveis reais	$15 * T_{relogioCPU}$
dividir duas variáveis inteiras	$40 * T_{relogioCPU}$
dividir duas variáveis reais	$40 * T_{relogioCPU}$
converter uma variável real para inteira	$45 * T_{relogioCPU}$
Operações para transferência de informação	
copiar uma variável inteira para outra ‡	0
Operações que alteram o fluxo de execução	
ciclo, construtor condicional (apenas instrução de salto)	$4 * T_{relogioCPU}$
utilização duma função (2 argumentos e valor devolvido inteiros)	$27 * T_{relogioCPU}$

† Com as variáveis em registo.
‡ O tempo necessário à execução desta operação é considerado no tempo de comunicação.

Tabela 7.1: Valor médio do tempo de execução em *software* de operações elementares.

O **modelo de *hardware*** é utilizado na estimação do desempenho do sistema e do espaço ocupado pela parte do sistema que é atribuída a *hardware*. O modelo base assumido coincide com o que foi apresentado na secção 4.2. Inclui uma unidade de controlo e um caminho de dados [GVNG94], com a novidade de neste trabalho as duas unidades serem atribuídas a componentes distintos da arquitectura alvo: um CPLD no caso da unidade de controlo e uma FPGA no caso do caminho de dados. Ou seja, a arquitectura é uma concretização fiel ao modelo de estimação, garantido-se deste modo uma maior probabilidade de obter estimativas precisas e fiéis. Relativamente à modelação das tarefas da síntese de alto nível, o modelo é bastante detalhado, o que se traduz num cálculo moroso. Para diminuir o tempo de cálculo aplicou-se o mecanismo de estimação incremental.

O **modelo de comunicação** define o protocolo e a largura do canal de comunicação envolvi-

dos em cada tipo de transferência de informação, sendo utilizado na estimação do desempenho do sistema e do espaço ocupado em *hardware*. Enquanto a comunicação entre uma partição de *software* e uma partição de *hardware* obedece ao protocolo do barramento PCI, com a largura do canal para dados a ser de 8 bits, a comunicação entre duas partições de *hardware* adjacentes faz-se através dum protocolo síncrono simples, que envolve um sinal de controlo para leitura/escrita, os bits de endereço e 16 bits para dados. Em termos de espaço, o modelo de comunicação intervém apenas na estimação dos recursos de interligação necessários à interface entre uma partição de *hardware* e uma de *software*, ou entre duas partições de *hardware* adjacentes.

O modelo de estimação segue o modelo de passagem de mensagens com algumas limitações: (i) as operações ponto a ponto iniciadas em *hardware* e em que o *software* intervém, transformam-se em operações de sentido contrário iniciadas por *software*, (ii) as operações ponto a ponto iniciadas em *hardware* e em que participa outra partição de *hardware* não adjacente, situações sinalizadas a tracejado na figura 7.1, são decompostas em duas operações iniciadas por *software* e (iii) as operações difusão, dispersão e junção[1] são possíveis mas com um desempenho penalizado. As limitações resultam da topologia de comunicação da arquitectura alvo, que embora seja centralizada e em anel não é completa, e da inexistência de funcionalidade *master* no controlador PCI da plataforma EDgAR-2.

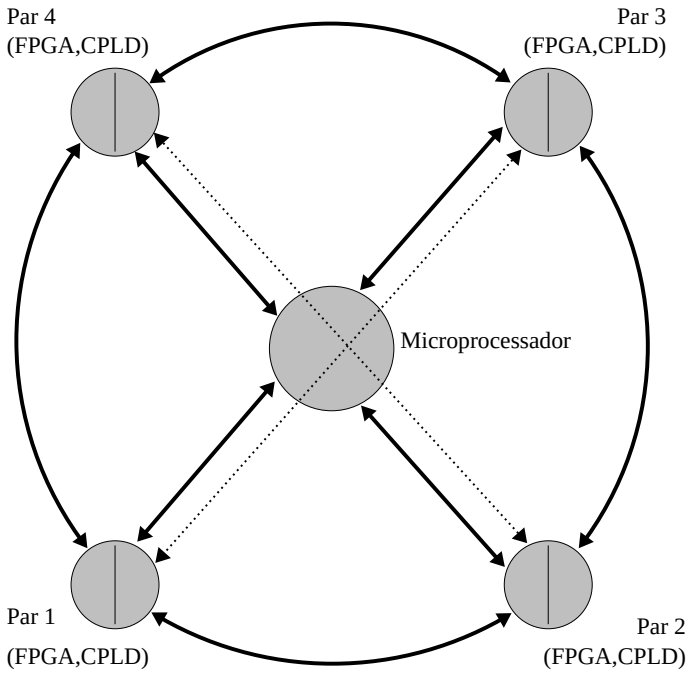


Figura 7.1: Topologia de comunicação da arquitectura alvo.

Para cada tipo de transacção que é possível efectuar na arquitectura alvo, a tabela 7.2 estabelece o valor do tempo de comunicação. Os valores apresentados aplicam-se a uma operação

[1] Estas operações foram definidas na secção 5.3.

de leitura ou escrita efectuada com uma única transacção.

Tipo de Operação		***Tempo***
transacção dentro duma partição de s/w	(registo → registo)	2 * $T_{relogioCPU}$
	(registo → memória)	12 * $T_{relogioCPU}$
	(memória → registo)	20 * $T_{relogioCPU}$
	(memória → memória)	22 * $T_{relogioCPU}$
	(constante → registo)	10 * $T_{relogioCPU}$
	(constante → memória)	12 * $T_{relogioCPU}$
leitura de h/w a partir duma partição de s/w †		$7 * TclkPCI + 40 * T_{relogioCPU}$
escrita em h/w a partir duma partição de s/w		$5 * TclkPCI + 15 * T_{relogioCPU}$
leitura dentro duma partição de h/w	(variável simples)	0
	(variável composta)	$TclkHW$
escrita dentro duma partição de h/w	(variável simples)	$TclkHW$
	(variável composta)	2 * $TclkHW$
leitura duma partição de h/w adjacente a partir de h/w §		$\lceil 110ns \ / \ TclkHW \rceil * TclkHW$
escrita numa partição de h/w adjacente a partir de h/w §		$\lceil 90ns \ / \ TclkHW \rceil * TclkHW$
leitura de s/w a partir duma partição de h/w		‡
escrita em s/w a partir duma partição de h/w		‡
leitura duma partição de h/w não adjacente a partir de h/w		‡
escrita numa partição de h/w não adjacente a partir de h/w		‡

† $TclkPCI$ é o período do relógio utilizado no barramento PCI.
§ A temporização desta transacção será explicada na secção 7.4.3.
‡ A temporização desta transacção, implementada através duma operação de leitura e/ou de escrita iniciada por *software*, será descrita na secção 7.4.3.

Tabela 7.2: Valor médio do tempo de comunicação para cada tipo de transacção elementar.

7.2.1 Modelo de *Software* Melhorado

Como complemento ao modelo simples aplicado na estimação do tempo de computação dos estados programa, apresenta-se um modelo melhorado que considera as optimizações de código resultantes dos níveis de *pipeline*, da superescalaridade e da hierarquia de memória do processador, através dum factor obtido estaticamente [PH97]. De acordo com este modelo, a estimativa do tempo de computação em *software* dum estado programa o_i pode ser aproximada pela equação 7.1, considerando que a funcionalidade de o_i é traduzida por um bloco de código em que o número de execuções de instruções é $N_{execucoes}(o_i)$ e o processador alvo funciona com uma frequência de relógio igual a $F_{relogioCPU}$.

$$TexecSW(o_i) = N_{execucoes}(o_i) * CPI * \frac{1}{F_{relogioCPU}} \tag{7.1}$$

O número médio de ciclos por instrução (CPI), na execução dum programa e para uma determinada implementação dum processador, depende dos níveis de *pipeline*, do grau de superescalaridade (GSe) e da arquitectura da hierarquia de memória, desde a memória *cache* nível 1 até ao dispositivo de armazenamento mais lento.

A duração média de execução duma instrução inclui duas contribuições: uma que depende

da organização interna do CPU e outra que contabiliza as penalidades resultantes do acesso a níveis mais lentos da hierarquia de memória (equações 7.2 e 7.3).

$$CPI = CPI_{CPU} + CPI_{MEM} \tag{7.2}$$

em que CPI_{CPU} é o valor do CPI quando a totalidade do código e dos dados do programa se encontra na *cache* nível 1 e

$$CPI_{MEM} = (1 + Taxa_{MediaAcessosMemoriaPI}) * Penalidade_{Acesso(Mem(1))} \tag{7.3}$$

No cálculo do valor médio do número de ciclos gastos em acessos à memória por instrução, CPI_{MEM} na equação 7.3, o termo "1" representa um acesso à memória para ler a instrução a executar e $Taxa_{MediaAcessosMemoriaPI}$ designa a percentagem de ciclos de relógio que o programa (conjunto de instruções) gasta com acessos à memória para ler ou escrever dados. O valor de $Taxa_{MediaAcessosMemoriaPI}$ depende essencialmente do código da aplicação.

A penalidade a aplicar a cada acesso a um nível mais lento da hierarquia de memória, $Penalidade_{Acesso(Mem(1))}$ na equação 7.3, é dada pela equação 7.4.

$$\begin{aligned} Penalidade_{Acesso(Mem(1))} = {} & HitRate(Mem(1)) * Nciclos_{Acesso(Mem(1))} \; + \\ & + MissRate(Mem(1)) * Penalidade_{Acesso(Mem(2))} \end{aligned} \tag{7.4}$$

em que $Nciclos_{Acesso(Mem(1))}$ representa o número médio de ciclos que é necessário a um acesso à memória de nível 1. Para calcular a penalidade a aplicar a cada acesso à memória de nível K, $Penalidade_{Acesso(Mem(K))}$ com $K \geq 2$, utiliza-se uma expressão equivalente à equação 7.4:

$$\begin{aligned} Penalidade_{Acesso(Mem(K))} = {} & HitRate(Mem(K)) * Nciclos_{Acesso(Mem(K))} \; + \\ & + MissRate(Mem(K)) * Penalidade_{Acesso(Mem(K+1))} \end{aligned} \tag{7.5}$$

Para uma memória de nível K, se o tempo de acesso for $T_{acesso(Mem(K))}$, a quantidade mínima de informação acessível na memória (bloco) for $bitsBloco(Mem(K))$, a largura do canal de comunicação entre os níveis $K-1$ e K da hierarquia de memória for $bitsCanal(K-1, K)$ e a frequência de relógio for $F_{relogioCPU}$, o número médio de ciclos que são necessários a um acesso a essa memória é definido pela equação 7.6. No caso da memória *cache* (nível 1, nível 2 ou outro inferior), a quantidade mínima de informação acessível é uma linha e no caso do disco rígido é um sector.

$$Nciclos_{Acesso(Mem(K))} = \lceil T_{acesso(Mem(K))} * F_{relogioCPU} * \lceil \frac{bitsBloco(Mem(K))}{bitsCanal(K-1,K)} \rceil \rceil \quad (7.6)$$

Por exemplo, se uma memória *cache* possuir um tempo de acesso de 20 ns e linhas de 128 bits, se o processador funcionar a 200 MHz e se a largura do canal de comunicação com esta memória *cache* for 32 bits, o número médio de ciclos que são necessários a um acesso à memória é

$$N_{MedioCiclosAcesso} = \lceil 20 * 10^{-9} * 200 * 10^{6} * \lceil \frac{128}{32} \rceil \rceil = 16$$

A penalidade pelo facto de ter que se ler um bloco duma memória de nível inferior, devido a uma falha[2], quando apenas se precisa de uma palavra, é atenuada se as várias palavras lidas forem utilizadas num futuro próximo.

Para que as estimativas obtidas com este modelo atinjam um grau de precisão elevado é necessário conhecer o estado do processador no momento da execução do código. Nesta situação, o modelo é incompatível com a filosofia de estimação a dois níveis (estado programa e sistema), uma vez que o tempo de computação dos estados programa deixa de poder ser calculado apenas no início do processo de partição para ter que ser recalculado em cada alternativa de partição analisada. Contudo, pode optar-se por uma solução de compromisso em que se consideram valores médios para todos os parâmetros do modelo, nomeadamente as taxas $HitRate$ e $MissRate$ para os vários níveis da hierarquia de memória e a taxa de ocupação do *pipeline*. Deste modo, embora a precisão das estimativas diminua, porque os parâmetros dependem muito da aplicação, o tempo de computação dos estados programa só precisa de ser calculado uma vez em todo o processo de partição.

7.3 Estimação do Espaço Ocupado em *Hardware* ao Nível do Sistema

De acordo com o modelo de estimação de *hardware*, o espaço ocupado por uma partição p de *hardware* é composto por duas parcelas: o espaço ocupado pelo caminho de dados da partição p ($CD(p)$) e o espaço ocupado pela unidade de controlo dessa partição ($UC(p)$).

7.3.1 Espaço Ocupado em *Hardware* pelo Caminho de Dados

Para o espaço ocupado pelo caminho de dados contribuem as unidades funcionais que implementam as operações dos estados programa do modelo do sistema, os elementos de armazenamento (ou seja, as variáveis do modelo do sistema), os elementos de interligação entre a saída das unidades funcionais e a entrada dos elementos de armazenamento e os recursos da

[2] A falha designa-se de *cache miss* no caso da memória *cache* e *page fault* no caso da memória RAM.

interface entre o caminho de dados e o seu exterior (equação 7.7). Como a ferramenta de síntese a utilizar não atribui múltiplas operações à mesma unidade funcional, a não ser que este facto seja explicitado na descrição do sistema, apenas se exige a inclusão de multiplexadores na entrada dos elementos de armazenamento.

$$\begin{aligned} areaHW(CD(p)) \;=\; & areaEPrograma(CD(p)) \;+\; areaVars(CD(p)) \;+ \\ & +\; areaMuxesExtra(CD(p)) \;+\; areaInterface(CD(p)) \end{aligned} \tag{7.7}$$

O termo $\boldsymbol{areaEPrograma(CD(p))}$ representa o espaço ocupado em *hardware* pelos estados programa o_i atribuídos à partição p. Neste espaço inclui-se a contribuição das unidades funcionais que implementam as operações dos estados programa e dos elementos de interligação (multiplexadores) e exclui-se o espaço ocupado pelos elementos de armazenamento. A estimativa para o espaço $areaEPrograma(CD(p))$ é dada pela equação 7.8, sendo o espaço ocupado por o_i ($areaHW(o_i)$) estimado ao nível do estado programa.

$$areaEPrograma(CD(p)) \;=\; \sum_{(o_i \in p) \;\wedge\; (tipo(o_i) \neq variavel)} areaHW(o_i) \tag{7.8}$$

A parcela $\boldsymbol{areaVars(CD(p))}$ constitui uma estimativa para o espaço ocupado pelas variáveis atribuídas à partição p e pelas variáveis locais de p. O valor da estimativa para $areaVars(CD(p))$ é dado pela equação 7.9.

$$\begin{aligned} areaVars(CD(p)) \;=\; & areaVarAtribuidas(CD(p)) \;+\; areaVarLocais(CD(p)) \;= \\ = & \sum_{va_j \;\in\; aVAR(CD(p))} areaHWvar(va_j) \;+\; \sum_{vl_k \;\in\; lVAR(CD(p))} areaHWvar(vl_k) \end{aligned} \tag{7.9}$$

sendo $aVAR(CD(p))$ o conjunto de variáveis atribuídas à partição p, $lVAR(CD(p))$ o conjunto de variáveis locais de p e $areaHWvar(v)$ a estimativa do espaço ocupado pela variável v, obtida ao nível do estado programa.

Com o objectivo de reduzir o excesso de comunicação entre partições, resultante de repetidas operações de leitura e/ou escrita sobre a mesma variável efectuadas por um estado programa, introduziu-se a possibilidade de criar uma cópia de variáveis externas nas partições em que elas são acedidas. A copia duma variável, numa partição distinta daquela a que a variável foi atribuída, é designada por **variável local**. Para as partições de *hardware*, as variáveis locais apresentam uma segunda vantagem: garantem maior fiabilidade na interacção entre partições. Ao reduzirem drasticamente o atraso entre a origem dos dados (externos à partição) e as unidades funcionais da partição, as variáveis locais aumentam a fiabilidade do processamento desses dados. O espaço que ocupam em *hardware* é um inconveniente da utilização de variáveis

locais, atenuado pelo facto de que quando se trata de variáveis compostas apenas se reserva o espaço necessário para guardar um elemento dessas variáveis.

Para identificar a necessidade duma variável local numa partição utiliza-se a regra VL.

Regra VL *Inserção de variáveis locais*

> Utiliza-se uma variável local na partição p_j, como cópia duma variável vL atribuída a uma partição distinta de p_j, quando existe pelo menos um estado atribuído a p_j que lê múltiplas vezes a variável vL (do exterior da partição). Numa partição, só pode existir uma variável local para substituir a mesma variável externa, mesmo que a variável seja lida por vários estados.

O termo $\boldsymbol{areaMuxesExtra(CD(p))}$ define uma estimativa para o espaço ocupado pelos elementos de interligação necessários à partição p, depois de excluídos os elementos de interligação contabilizados em $areaEPrograma(CD(p))$. O valor de $areaMuxesExtra(CD(p))$ é calculado com a equação 7.10.

$$areaMuxesExtra(CD(p)) = \\ = \sum_{v_j \in awVAR(CD(p))} area(MUX[inMux_{v_j,CD(p)}, 1, larguraVar(v_j)]) \tag{7.10}$$

em que

- $awVAR(CD(p))$ é o conjunto que reúne as variáveis atribuídas à partição p com as variáveis escritas pelos estados programa de p;
- $area(MUX[inMux_{v_j,CD(p)}, 1, larguraVar(v_j)])$ representa o espaço ocupado por um multiplexador com $inMux_{v_j,CD(p)}$ entradas, uma saída e as entradas possuem uma largura de $larguraVar(v_j)$ bits; o valor deste espaço é obtido através da equação 7.88 ou 7.89, apresentada mais à frente;
- $inMux_{v_j,CD(p)}$ é o número de entradas necessário ao multiplexador a colocar na entrada da variável v_j da partição p, depois de excluídas as entradas já contabilizadas nos estados programa.

O número de **sinais de selecção**, necessário ao multiplexador colocado na entrada duma variável v_j da partição p, é definido pela equação 7.11. Como este número é calculado em duas fases, ao nível do estado programa e ao nível da partição, poderá resultar num valor ligeiramente superior ao exigido.

$$ncontrolMux(v_j, CD(p)) \; = \; \sum_{o_i \; \in \; aOBJ(CD(p))} muxes(o_i).ncontrol(v_j) \; + \; \lceil log_2 \; inMux_{v_j, CD(p)} \rceil \tag{7.11}$$

em que

- $aOBJ(CD(p))$ é o conjunto de estados programa atribuídos à partição p;
- $muxes(o_i).ncontrol(v_j)$ é a parcela de sinais de selecção do multiplexador colocado na entrada da variável v_j, exigida pelas operações de escrita sobre v_j efectuadas pelo estado programa o_i; o valor de $muxes(o_i).ncontrol(v_j)$ é calculado ao nível do estado programa;
- a segunda parcela da expressão representa um número de sinais de selecção que depende (i) do número de estados programa atribuídos a p que escrevem v_j e (ii) do número de partições, distintas de p, que também escrevem v_j.

O termo $\boldsymbol{areaInterface(CD(p))}$ representa o espaço ocupado pelos recursos da interface entre o caminho de dados da partição p e o seu exterior. A interface é responsável pela comunicação entre a partição p e as outras partições.

Excluindo os *buffers tri-state*, a lógica que contribui para $areaInterface(CD(p))$ é responsável por implementar os sinais de controlo da leitura e escrita de variáveis do (no) exterior das partições. Designando por $LOAD$ o sinal que permite escrever numa variável (registo em *hardware*) e $notOE$ o sinal que permite ler uma variável, através do *buffer tri-state* que liga o registo a um barramento externo à FPGA, a estimativa para $areaInterface(CD(p))$ corresponde ao espaço ocupado pela lógica que gera os sinais $LOAD$ e $notOE$ para (i) as variáveis lidas e/ou escritas do (no) exterior da partição e (ii) as variáveis atribuídas à partição que são lidas e/ou escritas a partir do seu exterior (figura 7.2).

De acordo com a topologia de comunicação da arquitectura alvo, ilustrada na figura 7.1, existem 4 tipos de comunicação ponto a ponto envolvendo duas partições: (i) de *hardware* para *software*, (ii) de *software* para *hardware*, (iii) entre *hardware* e *hardware* adjacente e (iv) entre *hardware* e *hardware* não adjacente. Em cada um dos tipos há ainda que distinguir as operações de leitura e de escrita.

Nos tipos de comunicação (i), (ii) e (iv) o *software* é um dos intervenientes, razão pela qual o canal de comunicação utilizado é o barramento PCI. Deste modo, a temporização das operações de escrita e leitura é imposta pelo controlador do barramento PCI incluído na plataforma EDgAR-2, descrito em [Est98]. Como a ligação entre *hardware* e *hardware* adjacente é dedicada, o protocolo síncrono empregue na comunicação é muito simples: usa-se um endereço

para identificar a variável alvo da operação, um sinal para definir se a operação é de escrita ou de leitura e os dados envolvidos na comunicação. Efectuando algumas simplificações, justificadas pelo facto de a sua não efectivação exigir um tempo de cálculo elevado e porque a contribuição de $areaInterface(CD(p))$ para o espaço do caminho de dados é limitada, obtiveram-se as seguintes expressões para a estimativa do espaço necessário para gerar os sinais $notOE$ e $LOAD$ esquematizados na figura 7.2:

$$\begin{cases} area\_notOEsw = & 2 * nBitsEnderecoSW + 18 \\ area\_LOADsw = & 2 * nBitsEnderecoSW + 23 \\ area\_notOEdir = area\_notOEesq = & 2 * nBitsEnderecoHW2HWa \\ area\_LOADdir = area\_LOADesq = & 2 * nBitsEnderecoHW2HWa \end{cases} \tag{7.12}$$

sendo $nBitsEnderecoSW$ a largura dos endereços envolvidos na comunicação entre *hardware* e *software* e $nBitsEnderecoHW2HWa$ a largura dos endereços usados na comunicação entre *hardware* e *hardware* adjacente. O valor do espaço está quantificado em portas lógicas equivalentes.

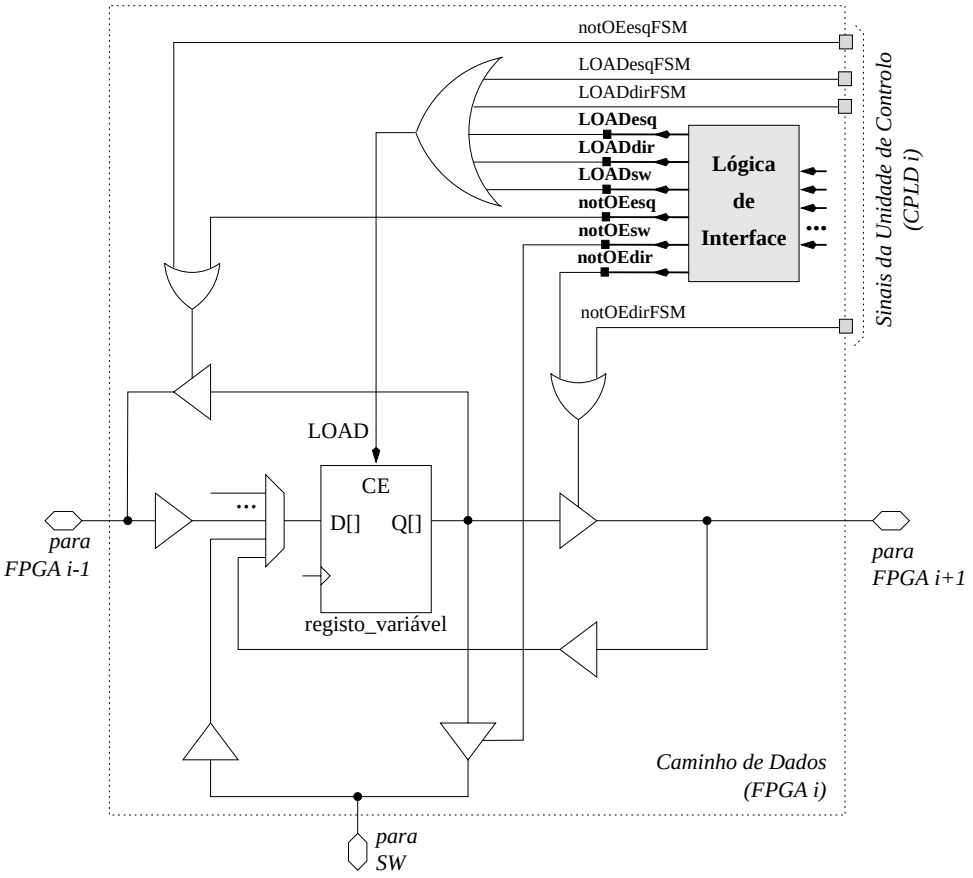


Figura 7.2: Interface entre uma variável numa partição de *hardware* e o exterior dessa partição, com destaque para os sinais que controlam a leitura ($notOE$) e a escrita ($LOAD$) da variável.

Para simplificar a estimação do valor de $nBitsEnderecoSW$ e especialmente para o manter constante ao longo do processo de partição, recorre-se ao limite superior para o número de endereços. Este limite resulta de considerar que todas as variáveis intervêm na comunicação

hardware ↔ *software* e *hardware* ↔ *hardware* não adjacente.

Do mesmo modo, para simplificar a estimação do valor de $nBitsEnderecoHW2HWa$ e para o manter constante durante o processo de partição, utilizou-se o pior cenário possível na comunicação entre *hardware* e *hardware* adjacente: cada partição de *hardware* lê/escreve todas as variáveis do sistema da outra partição, excepto as variáveis que possuem um número de elementos elevado e/ou uma largura elevada. As variáveis mencionadas foram excluídas porque as variáveis com número de elementos elevado e/ou largura elevada, ao ocuparem um espaço inaceitável em *hardware* serão atribuídas a *software*, não podendo assim intervir na comunicação entre partições de *hardware*.

Um limite razoável para o produto do número de elementos duma variável pela largura dos seus elementos, factor que decide se a variável é ou não considerada na estimação de $nBitsEnderecoHW2HWa$, é 2048 [(3)].

A estimativa do espaço ocupado pela interface duma partição de *hardware* p considera dois contributos (equação 7.13): (i) a lógica que suporta as operações iniciadas em p para ler ou escrever variáveis externas a esta partição ($areaOpInternas$) e (ii) a lógica que suporta as operações iniciadas no exterior de p para ler ou escrever variáveis atribuídas a esta partição ($areaOpExternas$).

$$areaInterface(CD(p)) = areaOpInternas(CD(p)) + areaOpExternas(CD(p)) \tag{7.13}$$

Para estimar $areaOpInternas(CD(p))$ analisa-se cada uma das variáveis lidas ou escritas pela partição de *hardware* p e, de acordo com a localização da outra partição interveniente na comunicação e o número de transacções exigido pela largura da variável, contabiliza-se o espaço ocupado pela lógica que gera o sinal $notOE$ ou $LOAD$ necessário a cada transacção da variável (equação 7.14). A leitura e escrita de variáveis localizadas numa partição de *hardware* adjacente, não requer lógica de interface na partição em que se iniciam as operações, dado que os sinais $notOE$ e $LOAD$[4] são gerados pela unidade de controlo dessa partição.

$$\begin{aligned} areaOpInternas(CD(p)) &= \\ = \sum_{v_j \in wVAR(CD(p)) \ \wedge \ (p_j=SW \ \vee \ p_j=pNA(p))} & nTsw(v_j) * area\_notOEsw \ + \\ + \sum_{v_j \in rVAR(CD(p)) \ \wedge \ (p_j=SW \ \vee \ p_j=pNA(p))} & nTsw(v_j) * area\_LOADsw \end{aligned} \tag{7.14}$$

[3] O valor **2048**, ou 256*8 bits, representa $\frac{1}{4}$ do tamanho da maior memória RAM que se pode implementar numa FPGA da arquitectura alvo (aproximadamente 1024*8 bits).

[4] Estes sinais surgem na figura 7.2 com os nomes $notOEesqFSM$, $notOEdirFSM$, $LOADesqFSM$ e $LOADdirFSM$.

onde

- $rVAR(CD(p))$ e $wVAR(CD(p))$ designam o conjunto das variáveis lidas e escritas pelos estados programa atribuídos à partição p, respectivamente;
- p_j identifica a partição à qual a variável v_j está atribuída;
- $nTsw(v_j)$ é o número de transacções envolvido numa leitura ou escrita da variável v_j, quando a partição de *software* intervém na comunicação;
- $pNA(p)$ identifica a partição não adjacente de p.

Para obter a estimativa de $areaOpExternas(CD(p))$ analisa-se cada partição distinta de p e, por cada variável atribuída a p que seja lida ou escrita por essa partição, contabiliza-se o espaço ocupado pela lógica que gera os sinais $notOE$ ou $LOAD$ envolvidos na transferência da variável (equação 7.15).

$$
\begin{aligned}
areaOpExternas(CD(p)) \;=\; & \\
= \sum_{v_k \in\, [\, rVAR(CD(SW)) \,\cup\, rVAR(CD(pNA(p)))\,] \,\wedge\, p_k = p} & nTsw(v_k) * area\_notOEsw \;+ \\
+ \sum_{v_k \in\, rVAR(CD(pDir(p))) \,\wedge\, p_k = p} & nTa(v_k) * area\_notOEdir \;+ \\
+ \sum_{v_k \in\, rVAR(CD(pEsq(p))) \,\wedge\, p_k = p} & nTa(v_k) * area\_notOEesq \;+ \\
+ \sum_{v_k \in\, [\, wVAR(CD(SW)) \,\cup\, wVAR(CD(pNA(p)))\,] \,\wedge\, p_k = p} & nTsw(v_k) * area\_LOADsw \;+ \\
+ \sum_{v_k \in\, wVAR(CD(pDir(p))) \,\wedge\, p_k = p} & nTa(v_k) * area\_LOADdir \;+ \\
+ \sum_{v_k \in\, wVAR(CD(pEsq(p))) \,\wedge\, p_k = p} & nTa(v_k) * area\_LOADesq
\end{aligned}
\tag{7.15}
$$

em que

- $nTa(v_k)$ é o número de transacções envolvido numa leitura ou escrita da variável v_k, quando a comunicação se faz entre partições de *hardware* adjacentes;
- $pDir(p)$ e $pEsq(p)$ identificam a partição adjacente à direita e à esquerda de p.

7.3.2 Espaço Ocupado em *Hardware* pela Unidade de Controlo

A unidade de controlo duma partição de *hardware* implementa a máquina de estados que temporiza as actividades do caminho de dados que lhe está associado. No caso concreto da plataforma EDgAR-2, a unidade de controlo também é responsável pela geração de sinais que coordenam a comunicação entre a partição de *hardware* e o *software*, através do barramento PCI. Deste modo, para o espaço da unidade de controlo contribuem a máquina de estados, os recursos da interface entre a unidade de controlo (CPLD) e o seu exterior e a lógica que gera os sinais de controlo da comunicação entre *hardware* e *software*. O valor do espaço é quantificado na equação 7.16.

$$areaHW(UC(p)) = \\ = areaHWfsm(UC(p)) + areaInterface(UC(p)) + areaSinaisCtlPCI(UC(p)) \quad (7.16)$$

em que

- $\boldsymbol{areaHWfsm(UC(p))}$ é o espaço ocupado pela máquina de estados que controla o caminho de dados $CD(p)$;
- $\boldsymbol{areaInterface(UC(p))}$ representa o espaço ocupado pelos recursos da interface entre a unidade de controlo e o caminho de dados que lhe está associado, uma vez que a $UC(p)$ praticamente só interactua com o $CD(p)$; a interface exige poucos recursos porque se limita a encaminhar os sinais de estado e de controlo entre a $UC(p)$ e o $CD(p)$, razão pela qual em termos de estimação se optou por desprezar este termo;

$$areaInterface(UC(p)) = 0 \quad (7.17)$$

- $\boldsymbol{areaSinaisCtlPCI(UC(p))}$ define o espaço ocupado pela lógica que gera os sinais que controlam a comunicação entre uma partição de *hardware*[5] e outra de *software*[6] através do barramento PCI; os sinais a gerar são S_READY, S_TERM, T_ABORT, $INTR\_N$, e $KEEPOUT$ [Est98]; assume-se que o valor desta estimativa é constante e igual a 150 portas lógicas equivalentes.

$$areaSinaisCtlPCI(UC(p)) = 150 \quad (7.18)$$

O modelo de estimação para o espaço ocupado pela máquina de estados da unidade de controlo $UC(p)$ considera três contribuições: o registo de estado, a lógica de controlo e a lógica

[5] Implementada com um par (FPGA,CPLD) da plataforma EDgAR-2.
[6] Implementada no sistema hospedeiro.

do próximo estado. Segundo este modelo, para obter estimativas para o espaço das três contribuições utilizaram-se as seguintes métricas: número de estados (***nE***), número de sinais de controlo gerados (***nSC***), número de sinais de estado lidos (***nSE***), número de activações de sinais de controlo (***nASC***) e número de leituras de sinais de estado (***nLSE***).

A estimação do espaço ocupado pela máquina de estados decorre nas seguintes etapas: (i) efectuar o escalonamento dos estados programa do sistema em etapas de controlo, (ii) para cada estado programa, determinar a máquina de estados que controla a sua operação, substituindo as tarefas[7] escalonadas em cada uma das etapas de controlo por uma secção de máquina de estados e estimar as métricas para esse estado programa, (iii) incluir nas métricas a informação relativa aos multiplexadores do caminho de dados e (iv) a informação sobre as mudanças de partição que ocorrem no fluxo de controlo.

Aquando da estimação de $areaHWfsm(UC(p))$ o **escalonamento** dos estados programa já é conhecido, uma vez que é necessário à obtenção das métricas no nível de abstracção do estado programa.

Na **segunda fase**, substituem-se as tarefas escalonadas na mesma etapa de controlo por uma secção de máquina de estados que resulta da combinação da secção de máquina de estados das várias tarefas. Como as tarefas resultam de construtores presentes na descrição VHDL dos estados programa, descreve-se nos próximos parágrafos a secção de máquina de estados para os principais construtores da linguagem VHDL:

- um **ciclo** é implementado com dois estados para a inicialização, um estado de teste e um estado que incrementa (ou decrementa) a variável de teste (figura 7.3 (i)); no total, utiliza quatro estados, gera dois sinais de controlo, lê um sinal de estado, ocorrem quatro activações de sinais de controlo e uma leitura de um sinal de estado; ou seja, $nE = 4$, $nSC = 2$, $nASC = 4$, $nSE = 1$ e $nLSE = 1$;

- se a execução de uma **operação aritmética ou lógica** *op* demorar $ne(op)$ ciclos e implicar o registo do resultado produzido, a secção de máquina de estados que lhe corresponde é ilustrada na figura 7.3 (ii); se a operação não registar o resultado, a secção de máquina de estados não gera o sinal de controlo indicado no último estado da figura; o valor de $ne(op)$, para um conjunto significativo de operações elementares, encontra-se na tabela 7.7 e o valor das métricas do modelo de estimação é $nE = ne(op)$, $nSC = 0|1$, $nASC = 0|1$, $nSE = 0$ e $nLSE = 0$;

- a **atribuição** dum valor a uma variável é implementada com um número de estados dependente da localização da variável em relação ao objecto que efectua a atribuição; o

[7] As tarefas resultam dos construtores presentes na descrição do estado programa e podem assumir a forma de operação, parte de controlo dum construtor ou atribuição.

sinal de controlo que habilita a escrita no elemento de armazenamento (ce) só é gerado pela máquina de estados quando a variável estiver localizada na partição em que se inicia a operação de escrita ou numa partição adjacente (figura 7.3 (iii)); o valor das métricas do modelo de estimação, para a atribuição dum valor a uma variável, está definido na tabela 7.3; uma atribuição implica sempre uma operação de escrita e poderá envolver também uma operação de leitura (por exemplo $var2 = var1$);

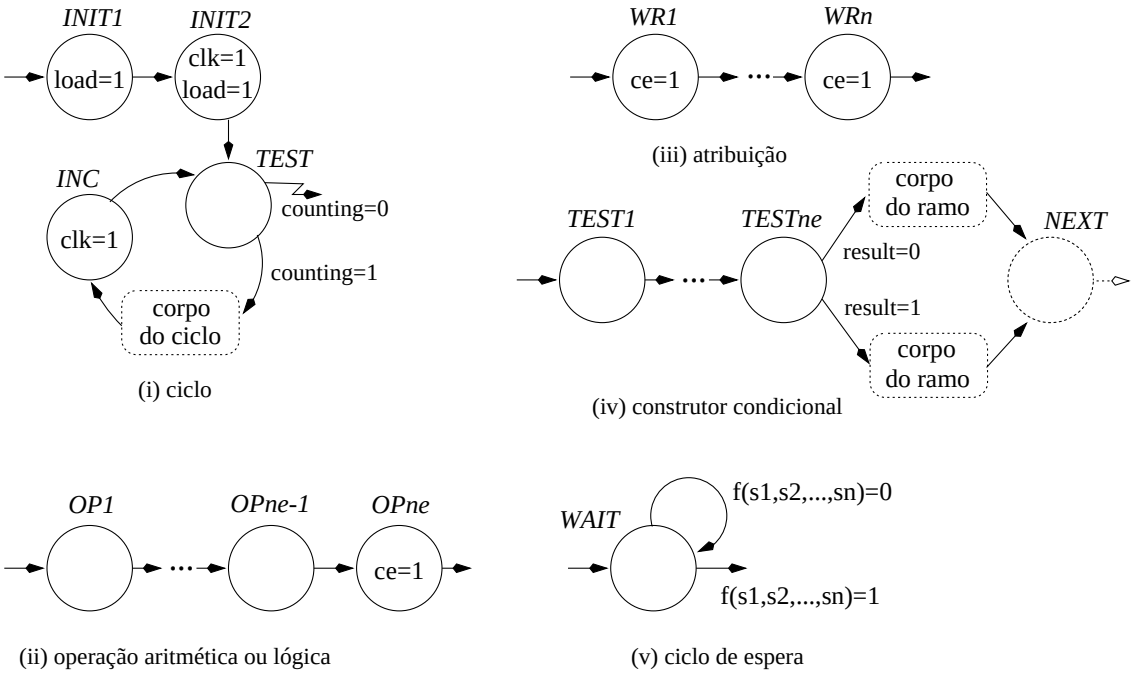


Figura 7.3: Secção de máquina de estados para os principais construtores da linguagem VHDL.

Tipo de Operação	***Métrica***				
	nE	***nSC***	***nASC***	***nSE***	***nLSE***
hardware lê de *software*	3	1	2	1	1
hardware escreve em *software*	3	1	2	1	1
hardware lê de *hardware* não adjacente	3	1	2	1	1
hardware escreve em *hardware* não adjacente	3	1	2	1	1
hardware lê de *hardware* adjacente	$TrdHa2H$ (†)	2	7	0	0
hardware escreve em *hardware* adjacente	$TwrH2Ha$ (†)	2	8	0	0
ler variável simples dentro da partição de *hardware*	0	0	0	0	0
ler variável composta dentro da partição de *hardware*	1	0	0	0	0
escrever variável simples dentro da partição de h/w	1	1	1	0	0
escrever variável composta dentro da partição de h/w	2	1	2	0	0

(†) $TrdHa2H$ e $TwrH2Ha$ representam o número de ciclos que demora uma leitura e escrita entre duas partições de *hardware* adjacentes, respectivamente.

Tabela 7.3: Métricas do modelo de estimação de $areaHWfsm(UC(p))$ relativas à leitura e escrita de variáveis por parte dos estados programa.

- um **construtor condicional**, que selecciona uma alternativa de evolução do sistema entre as várias possíveis, é implementado em *hardware* por um ou mais comparadores, consoante o número de alternativas de evolução discriminadas no construtor condicional;

na situação em que existem duas alternativas e supondo que a comparação demora *ne* ciclos, a secção de máquina de estados responsável pelo controlo da execução do construtor condicional é composta por *ne* estados, dispondo o último estado de duas transições (figura 7.3 (iv)); as condições de transição empregam o sinal de estado resultante da comparação (*result* na figura) e conduzem a duas secções de máquina de estado que se reúnem no primeiro estado associado ao código que se segue ao construtor condicional ($NEXT$ na figura); o valor das métricas do modelo de estimação é $nE = ne$, $nSC = 0$, $nASC = 0$, $nSE = 1$ e $nLSE = 1$;

- a um **ciclo de espera** do tipo *wait until f(s_1,s_2,...,s_n)*, ou de forma similar *wait on* s_1,s_2,...,s_n, que obriga o processo em que se insere o ciclo a esperar até ocorrerem eventos nos sinais s_1 a s_n que tornem a expressão $f(s_1, s_2, ..., s_n)$ verdadeira, corresponde uma secção de máquina de estados como a que se inclui na figura 7.3 (v); o valor das métricas do modelo de estimação é $nE = 1$, $nSC = 0$, $nASC = 0$, $nSE = \sum_{s_i \in rdVAR(o)} larguraVar(s_i)$ e $nLSE = nSE$; sendo que $rdVAR(o)$ representa o conjunto dos sinais s_1 a s_n lidos pelo estado programa o que impõe o ciclo de espera e $larguraVar(s_i)$ é o número de bits do sinal s_i;

- um **construtor paralelo**, que não é descrito de forma explícita em VHDL mas pode ser modelado em PSM ou PSMfg, é traduzido pelas secções de FSM ilustradas na figura 6.3 (iii); o valor das métricas do modelo de estimação é $nE = 2 * (nRamos + 1)$, $nSC = 2 * nRamos$, $nASC = 4 * nRamos$, $nSE = nSC$ e $nLSE = nSC$, em que $nRamos$ representa o número de ramos do construtor paralelo.

A combinação da secção de máquina de estados, para as várias tarefas dum estado programa escalonadas na mesma etapa de controlo, baseia-se nas regras a seguir apresentadas, das quais resultam as estimativas indicadas para as métricas do estado programa:

- o número de estados necessários a uma etapa de controlo é imposto pela tarefa mais lenta atribuída a essa etapa de controlo; como consequência, a estimativa para a métrica nE dum estado programa o é

$$nE(o) = \sum_{etapa_j \in schedule(o)} MAX_{tarefa_k \in etapa_j} [nE(tarefa_k)] \tag{7.19}$$

- para manter o processo de estimação simples e evitar incorrecções no funcionamento da máquina de estados, as tarefas não partilham sinais de controlo; deste modo, a estimativa para as métricas nSC e $nASC$ dum estado programa o é dada por

$$nSC(o) = \sum_{etapa_j \in schedule(o)} \sum_{tarefa_k \in etapa_j} nSC(tarefa_k) \tag{7.20}$$

$$nASC(o) = \sum_{etapa_j \in schedule(o)} \sum_{tarefa_k \in etapa_j} nASC(tarefa_k) \tag{7.21}$$

◇ a partilha de sinais de estado é obviamente permitida, significando isto que se excluem das métricas nSE e $nLSE$ as repetições relativas à leitura do mesmo sinal de estado e na mesma etapa de controlo, efectuadas por tarefas distintas; a estimativa para as métricas nSE e $nLSE$ dum estado programa o é definida pelas equações 7.22 e 7.23, respectivamente.

$$nSE(o) = \bigcup_{etapa_j \in schedule(o)} \bigcup_{tarefa_k \in etapa_j} nSE(tarefa_k) \tag{7.22}$$

$$nLSE(o) = \bigcup_{etapa_j \in schedule(o)} \bigcup_{tarefa_k \in etapa_j} nLSE(tarefa_k) \tag{7.23}$$

O contributo das tarefas de leitura e escrita de variáveis para as métricas dum estado programa merece uma atenção especial, apresentando-se para o efeito a estimação da métrica $nASC$. Num estado programa, cada variável lida só contribui uma vez para a métrica $nASC$ porque (i) as leituras múltiplas a partir do exterior da partição são convertidas numa única leitura e (ii) as leituras internas à partição não contribuem para o valor de $nASC$. Por seu lado, cada variável escrita por um estado programa o contribui para $nASC(o)$ um número de vezes igual ao "número de vezes que a variável é escrita em estados distintos do escalonamento de o". O contributo das operações de leitura e escrita de variáveis para o número de activações de sinais de controlo ($nASC(o)$), por parte dum estado programa o, é dado por

$$nASCvar(o) = \sum_{v \in rdVAR(o)} nASCread(p_v, p_o) +$$

$$+ \sum_{v \in wrVAR(o)} nASCwrite(p_v, p_o) * wrVAR(o).numEscritasDif(v) \tag{7.24}$$

em que

- $\boldsymbol{rdVAR(o)}$ é o conjunto de variáveis lidas por o;
- $\boldsymbol{wrVAR(o)}$ é o conjunto de variáveis escritas por o;
- $\boldsymbol{p_o}$ **(**$\boldsymbol{p_v}$**)** designa a partição a que o (v) foi atribuído(a);
- $\boldsymbol{wrVAR(o).numEscritasDif(v)}$ é o número de vezes que a variável v é escrita em estados distintos do escalonamento de o;

- utilizando a informação da tabela 7.3 obtém-se os valores de $\boldsymbol{nASCread(p_v, p_o)}$ e $\boldsymbol{nASCwrite(p_v, p_o)}$:

$$nASCread(p_v, p_o) = \begin{cases} 0 & , \text{ para } p_v = p_o \\ 7 & , \text{ para } p_v = pDir(p_o) \text{ ou } p_v = pEsq(p_o) \\ 2 & , \text{ outros casos} \end{cases} \quad (7.25)$$

$$nASCwrite(p_v, p_o) = \begin{cases} 8 & , \text{ para } p_v = pDir(p_o) \text{ ou } p_v = pEsq(p_o) \\ 1 & , \text{ para } p_v = p_o \text{ e } numElementosVar(v) = 1 \\ 2 & , \text{ outros casos} \end{cases} \quad (7.26)$$

  onde $pDir(p)$ e $pEsq(p)$ identificam a partição adjacente à direita e à esquerda de p e $numElementosVar(v)$ designa o número de elementos da variável v.

Na **terceira fase**, inclui-se no modelo de estimação de $areaHWfsm(UC(p))$ a informação sobre os sinais de selecção dos multiplexadores utilizados no caminho de dados. Para uma dada partição p, os multiplexadores a considerar são aqueles que se colocam na entrada das variáveis escritas e atribuídas a p. O contributo dos sinais de selecção para a métrica nSC é

$$nSCmux(p) = \sum_{v \in awVAR(p)} nControlMux(v, CD(p)) \quad (7.27)$$

em que

- $\boldsymbol{awVAR(p)}$ é o conjunto de variáveis escritas e/ou atribuídas a p;
- $\boldsymbol{nControlMux(v, CD(p))}$ designa o número de sinais de selecção do multiplexador a colocar na entrada da variável v atribuída e/ou escrita pela partição p; o valor de $nControlMux(v, CD(p))$ é obtido com a equação 7.11.

Assumindo que cada sinal de selecção dos multiplexadores é activado durante um estado, o contributo para o número de activações de sinais de controlo é igual ao contributo para o número de sinais de controlo:

$$nASCmux(p) = nSCmux(p) \quad (7.28)$$

Na **quarta fase**, inclui-se no modelo de estimação de $areaHWfsm(UC(p))$ a informação sobre as mudanças de partição que ocorrem no fluxo de controlo. Uma mudança de partição no fluxo de controlo implica a introdução dum estado programa do tipo *comutacao*, responsável por atribuir um valor a um sinal, e de um estado do tipo *sincronizacao* que espera por um evento num sinal (figura 7.4). O valor das métricas do modelo de estimação, em consequência duma mudança de partição, é $nE = 3$, $nSC = 1$, $nASC = 2$, $nSE = 1$ e $nLSE = 1$. Se o número de mudanças de partição, a partir de estados programa atribuídos à partição p

para estados não atribuídos a p, for $numMudancasParticao(p)$, as métricas do modelo de estimação para esta partição são definidas pelo conjunto de equações 7.29. Nestas equações, considerou-se que o sinal de controlo e o sinal de estado, usados nas mudança de partição, são partilhados por todos os estados programa do tipo *comutacao* e *sincronizacao*.

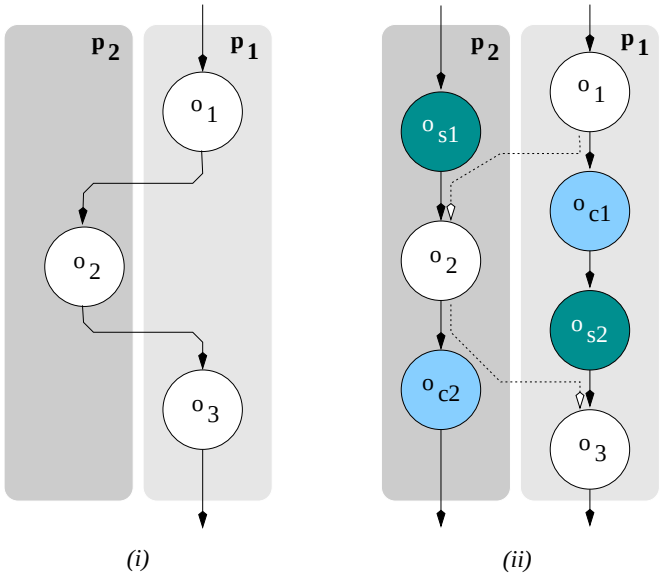


Figura 7.4: Alterações na descrição do sistema quando ocorre uma mudança de partição no fluxo de controlo: (i) descrição original e (ii) descrição alterada.

$$\begin{cases} nEpart(p) & = & 3 * numMudancasParticao(p) \\ nSCpart(p) & = & 1 \\ nASCpart(p) & = & 2 * numMudancasParticao(p) \\ nSEpart(p) & = & 1 \\ nLSEpart(p) & = & numMudancasParticao(p) \end{cases} \tag{7.29}$$

Combinando a informação obtida nas quatro fases anteriores resultam valores para as métricas nE, nSC, $nASC$, nSE e $nLSE$ ao nível da partição, como mostram as equações 7.30 a 7.34.

$$nE(p) = \sum_{o \in p} nE(o) + nEpart(p) \tag{7.30}$$

$$nSC(p) = \sum_{o \in p} nSC(o) + nSCpart(p) + nSCmux(p) \tag{7.31}$$

$$nASC(p) = \sum_{o \in p} nASC(o) + nASCpart(p) + nASCmux(p) \tag{7.32}$$

$$nSE(p) = \sum_{o \in p} nSE(o) + nSEpart(p) \tag{7.33}$$

$$nLSE(p) = \sum_{o \in p} nLSE(o) + nLSEpart(p) \tag{7.34}$$

Utilizando as métricas apresentadas obtém-se uma estimativa para o espaço ocupado pelo registo de estado, pela lógica de controlo e pela lógica do próximo estado.

Considerando uma codificação de estados do tipo *one-hot*, o espaço ocupado pelo **registo de estado** é proporcional ao número de estados da máquina de estados da $UC(p)$.

$$areaRegEstado(p) = nE(p) * area(FF\ D) \tag{7.35}$$

Se na estimação de métricas se considerar que a máquina de estados é do tipo Moore e a codificação de estados é do tipo *one-hot*, a **lógica de controlo** é um conjunto de portas OR para as saídas que estão activas em mais de um estado (figura 7.5 (i)). Ou seja, conhecidos o número de activações de sinais de controlo ($nASC(p)$) e o número de sinais de controlo ($nSC(p)$), a estimativa para o número de portas OR com duas entradas coincide com a diferença entre a estimativa da primeira e da segunda métrica (equação 7.36).

$$areaLogicaCtl(p) = [nASC(p) - nSC(p)] * area(OR2) \tag{7.36}$$

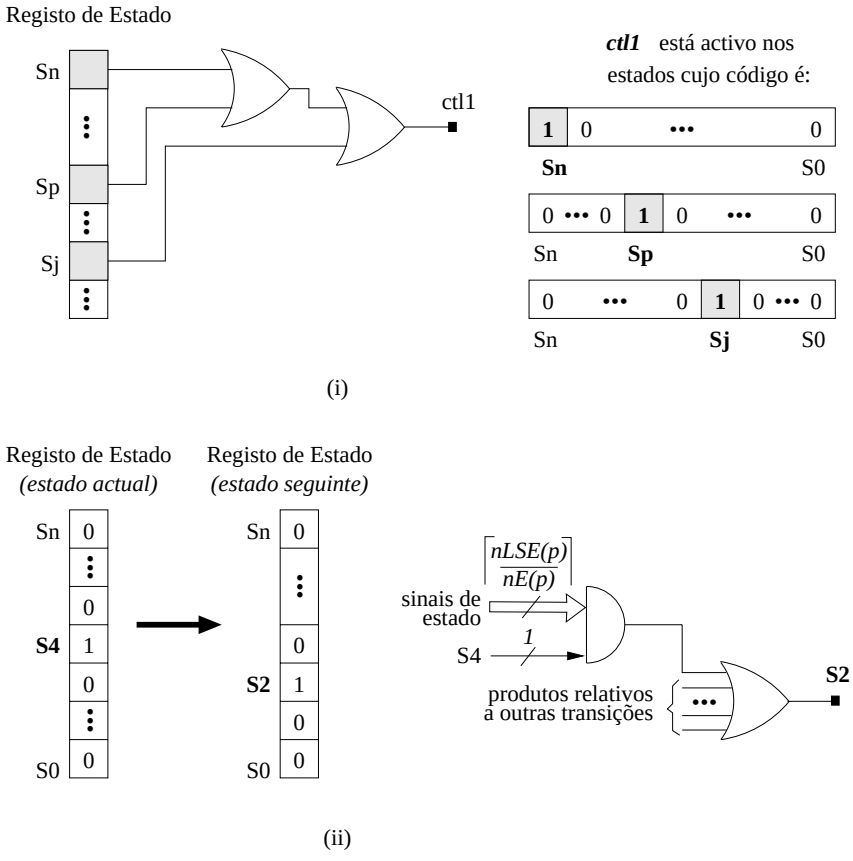


Figura 7.5: Exemplos que ilustram o processo de estimação do espaço ocupado (i) pela lógica de controlo e (ii) pela lógica do próximo estado duma máquina de estados.

Considerando uma codificação de estados do tipo *one-hot* e uma implementação com uma estrutura *AND-OR*, a estimativa para o espaço ocupado pela **lógica do próximo estado**

inclui uma parcela para portas AND e outra para portas OR.

O número de **portas *AND***, ou equivalentemente o número de produtos, coincide com o número de transições de estado $nTransicoesFsm(p)$. O número de entradas por porta AND, ou por produto, é aproximado por um valor médio calculado entre todos os estados. O valor médio incluído na equação 7.38 e na figura 7.5 (ii) é $\lceil \frac{nLSE(p)}{nE(p)} \rceil$ e resulta da aplicação dos seguintes princípios: (i) o número médio de sinais de estado lidos por estado é $\lceil \frac{nLSE(p)}{nE(p)} \rceil$, (ii) assume-se que todas as transições que partem dum estado utilizam a totalidade dos sinais de estado lidos por esse estado, (iii) segundo a codificação de estados do tipo *one-hot* cada produto utiliza apenas um bit do registo de estado e (iv) uma porta AND com num entradas equivale a $num - 1$ portas AND de duas entradas (figura 7.5 (ii)). O número de transições na máquina de estados duma partição p depende dos estados, das mudanças de partição, dos ciclos, dos construtores condicionais, dos ciclos de espera e dos construtores paralelos atribuídos à partição p (equação 7.37).

$$\begin{aligned} nTransicoesFsm(p) = nE(p) + numMudancasParticao(p) + nCiclos(p) + \\ + nCCondicionais(p) + nCEspera(p) + \sum_{cp_k \in CParalelo(p)} [1 + nRamos(cp_k)] \end{aligned} \tag{7.37}$$

onde $nCiclos(p)$, $nCCondicionais(p)$ e $nCEspera(p)$ designam o número de ciclos, de construtores condicionais e de ciclos de espera atribuídos à partição p e $CParalelo(p)$ é o conjunto dos construtores paralelos cp_k atribuídos a p, cada um com um grau de paralelismo $nRamos(cp_k)$.

O número de **portas *OR*** com duas entradas é definido pela diferença entre o número de transições de estado e o número de estados. Juntando as parcelas relativas a portas AND e OR obtém-se para estimativa do espaço ocupado pela lógica do próximo estado, o valor apresentado na equação 7.38.

$$\begin{aligned} areaLProxEstado(p) = \lceil nTransicoesFsm(p) * \frac{nLSE(p)}{nE(p)} * area(AND2) \rceil \ + \\ + (nTransicoesFsm(p) - nE(p)) * area(OR2) \end{aligned} \tag{7.38}$$

Por fim, a estimativa para o espaço ocupado pela máquina de estados que controla o caminho de dados da partição p, $areaHWfsm(UC(p))$, é calculada pela equação 7.39.

$$\begin{aligned} areaHWfsm(UC(p)) \ = \\ = \ areaRegEstado(p) \ + \ areaLogicaCtl(p) \ + \ areaLProxEstado(p) \end{aligned} \tag{7.39}$$

7.4 Estimação do Desempenho ao Nível do Sistema

O desempenho do sistema, sob a forma de tempo de execução[8], é função (i) do tempo de computação em *software* e em *hardware* dos estados programa do modelo do sistema, (ii) da partição a que os objectos foram atribuídos, (iii) do tempo de comunicação entre estados programa e variáveis, (iv) da frequência de execução dos estados programa ($FN(o_i)$), (v) do sincronismo explícito e implícito entre estados programa e (vi) do escalonamento do sistema.

7.4.1 Tempo de Computação dos Estados Programa

As estimativas para o tempo de computação em *software* ($TexecSW(o_i)$) e em *hardware* ($TexecHW(o_i)$) dos estados programa são obtidas com as tarefas T2 e T6 da secção 7.5, respectivamente. Os valores estimados para $TexecSW(o_i)$ e $TexecHW(o_i)$ permanecem fixos durante todo o processo de partição.

7.4.2 Atribuição dos Objectos às Partições

Naturalmente, a atribuição dos objectos do modelo do sistema às partições afecta o desempenho, uma vez vez que determina que tempo de computação se utiliza com cada estado programa ($TexecSW(o_i)$ ou $TexecHW(o_i)$) e impõe uma maior ou menor necessidade de comunicação entre o estado programa e o exterior da partição a que foi atribuído.

7.4.3 Tempo de Comunicação dos Estados Programa

O tempo de comunicação entre estados programa e variáveis depende das variáveis lidas e escritas por esses estados, das partições intervenientes nas operações de leitura e de escrita, do número de vezes que cada variável é lida e escrita, do número de transacções necessário a cada operação de leitura ou de escrita e do protocolo de comunicação. Alguns dos aspectos que afectam o tempo de comunicação e estão relacionados com o tipo de partições envolvidas na comunicação, são a topologia da arquitectura alvo da implementação, o tipo de comunicação permitido e as potencialidades do *device driver* da parte de *hardware* da arquitectura alvo, que se reflectem no desempenho dos tipos de comunicação citados. A comunicação pode efectuar-se através dum mecanismo de acesso directo a registo ou de acesso directo à memória (DMA), e por auscultação ou por interrupção.

Tendo como alvo das implementações a plataforma EDgAR-2 e o seu sistema hospedeiro, as transacções envolvidas na leitura ou escrita de variáveis recaem num dos seguintes tipos:

- leitura/escrita dentro duma partição de *hardware* ou de *software*;

[8] A transformação do tempo de execução numa medida de débito ou de latência não é problemática.

- ◇ leitura/escrita de *hardware* para/por *software*;
- ◇ leitura/escrita de *software* para/por *hardware*;
- ◇ leitura/escrita de *hardware* não adjacente para/por *hardware*;
- ◇ leitura/escrita de *hardware* adjacente para/por *hardware*.

Cada variável v_k que seja lida ou escrita por um estado programa o_i, contribui para o tempo de comunicação de o_i com um tempo do tipo

$$Tcom_i(v_k) = Tread_i(v_k) * numLeituras_i(v_k) * probLeitura_i(v_k) \tag{7.40}$$

ou

$$Tcom_i(v_k) = Twrite_i(v_k) * numEscritas_i(v_k) * probEscrita_i(v_k) \tag{7.41}$$

em que $Tread_i(v_k)$ ($Twrite_i(v_k)$) é tempo necessário para efectuar uma operação de leitura (escrita) da variável v_k, $numLeituras_i(v_k)$ ($numEscritas_i(v_k)$) é número de vezes que a variável v_k é lida (escrita) por o_i e $probLeitura_i(v_k)$ ($probEscrita_i(v_k)$) é a probabilidade de o_i ler (escrever) a variável v_k.

O cálculo do tempo de comunicação é ilustrado nas próximas secções através da leitura de *hardware* (*software*) por *software* (*hardware*) e da leitura (escrita) entre partições de *hardware* adjacentes. A estimação do tempo de comunicação para os restantes tipos de transacção pode ser consultada no apêndice F.

Leitura de *hardware* por *software*

A leitura duma variável v_k localizada em *hardware* por parte dum estado o_i atribuído a *software* exige um número de transacções $nTransf(v_k)$ que depende do número de bits da variável ($larguraVar(v_k)$) e da largura do canal de comunicação entre *hardware* e *software* ($pFPGA2SW$).

$$nTransf(v_k) = \lceil \frac{larguraVar(v_k)}{pFPGA2SW} \rceil \tag{7.42}$$

O tempo despendido numa leitura é definido pela equação 7.43. Aplicando este tempo na equação 7.40 obtém-se o tempo de comunicação envolvido na leitura de v_k por parte do estado o_i. Para o tempo de leitura contribuem três parcelas: a leitura do registo de *hardware*, o deslocamento dos bits lidos em cada transacção para a posição que vão ocupar na variável

alvo da leitura e a junção (operação *E-lógico*) dos bits lidos na transacção com a variável alvo da leitura.

$$Tread_i(v_k) = nTransf(v_k) * TreadHW2SW + (nTransf(v_k) - 1) * TandSW \; + \\ + nshifts(ordemHW, nTransf(v_k)) * TshiftSW \qquad (7.43)$$

em que

- $\boldsymbol{TreadHW2SW}$ é o tempo que o *software* demora a ler um registo de *hardware*, definido na tabela 7.2;
- $\boldsymbol{TandSW}$ e $\boldsymbol{TshiftSW}$ são o tempo de execução das operações *E-lógico* e *deslocação* em *software*, definidos na tabela 7.1;
- $\boldsymbol{nshifts(ordemHW, nTransf(v_k))}$ representa o tamanho dos deslocamentos envolvidos na leitura duma variável, localizada na partição de *hardware* cuja ordem é $ordemHW$, e em que $\boldsymbol{nTransf(v_k)}$ é o número de transacções necessárias para ler essa variável; os deslocamentos são necessários para guardar na posição correcta da variável alvo da leitura, a parcela de bits lida em cada transacção da leitura efectuada em múltiplas transacções; a função que calcula o valor de $nshifts$ é apresentada na figura G.4 do apêndice G.

Leitura de *software* por *hardware*

Nas operações de leitura ou escrita iniciadas numa partição de *hardware* e que exigem a participação do *software* podem ocorrer as seguintes situações: (i) a operação é iniciada por um estado programa (atribuído a *hardware*) que funciona ou não em paralelo com outros estados atribuídos a *software* e (ii) a transacção é iniciada por um estado programa inserido ou não no corpo de ciclos e/ou construtores condicionais. Quando o estado programa que inicia a operação funciona em paralelo com o *software*, a comunicação processa-se através do mecanismo de interrupção; caso contrário, utiliza-se o mecanismo de auscultação.

As operações que são iniciadas numa partição de *hardware* e exigem a participação do *software* são (i) a leitura de *software* para *hardware*, (ii) a escrita de *software* a partir do *hardware*, (iii) a leitura de *hardware* não adjacente para *hardware* e (iv) a escrita de *hardware* não adjacente a partir de *hardware*. Para introduzir na descrição do sistema a informação relativa à intervenção (não explicitada) do *software* nestas operações, é preciso efectuar modificações no código dos estados. As modificações relevantes são a alteração do código do estado de *hardware* que inicia a operação (o_i) e a criação ou alteração dum estado de *software* responsável pela execução

da operação (o_{com} ou o_{RSI}). Utiliza-se a designação o_{com} (o_{RSI}) quando a comunicação se processa com recurso ao mecanismo de auscultação (interrupção).

Nas leituras de *software* para *hardware* e nos casos em que o estado o_i que inicia a leitura não funciona em paralelo com outros estados atribuídos a *software*, ou seja, existe em *software* um estado o_s à espera que o fluxo de controlo regresse de *hardware* para *software*, a execução da leitura exige que se altere o código do estado o_i e que se introduza um estado o_{com} antes de o_s (figura 7.6). As alterações visam substituir a instrução de leitura de *software* para *hardware*, não possível na arquitectura alvo utilizada, por instruções que produzam o efeito desejado para essa instrução. Neste caso, a comunicação processa-se através do mecanismo de auscultação.

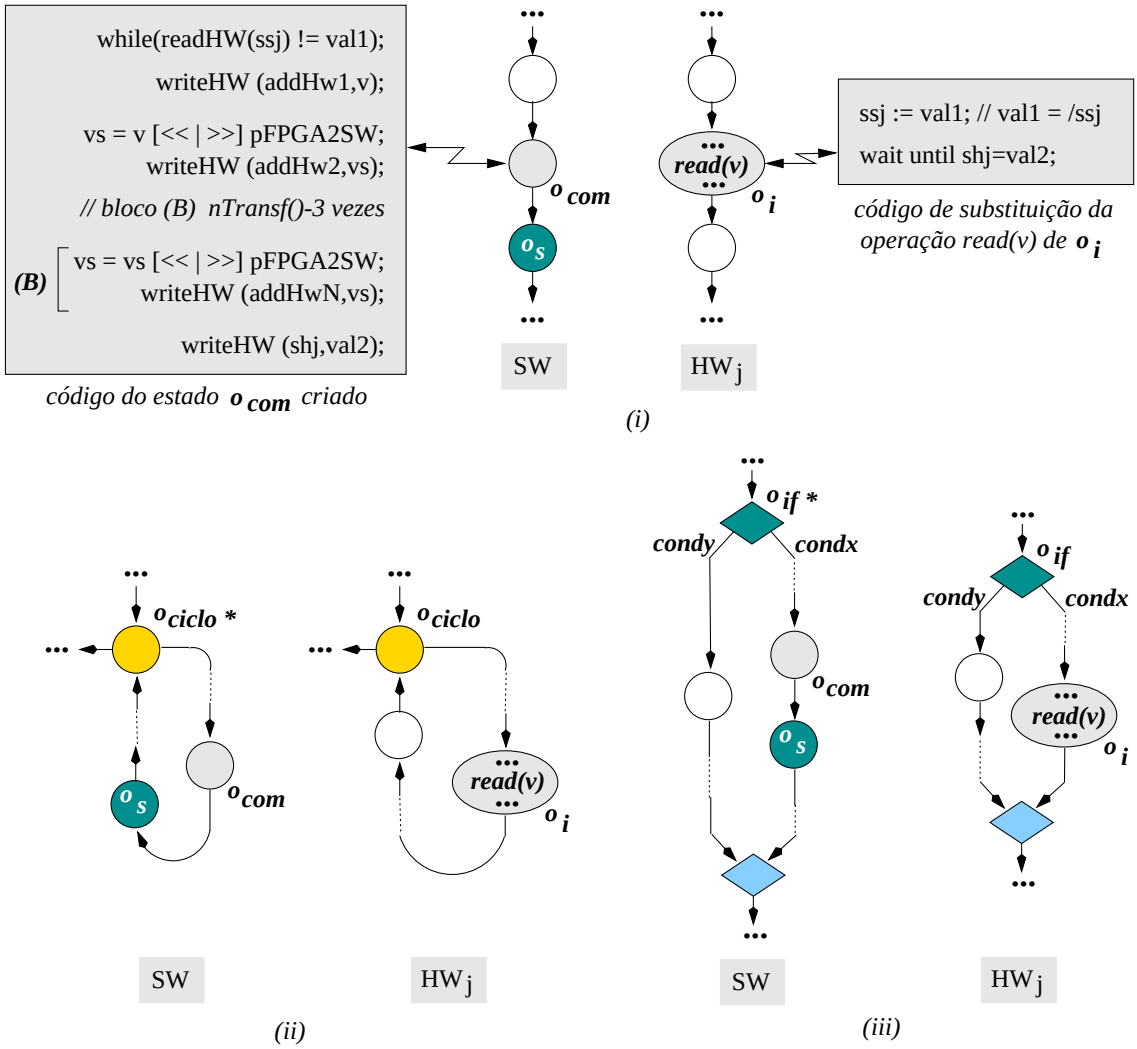


Figura 7.6: Ilustração da leitura duma variável de *software* para *hardware* utilizando o mecanismo de auscultação: (i) leitura simples, (ii) leitura inserida num ciclo e (iii) leitura inserida num construtor condicional.

Para o tempo duma leitura de *software* para *hardware* contribuem o tempo de execução das operações que substituíram a operação $read(v)$ no estado o_i e o tempo de execução das ope-

rações do estado o_{com} criado (figura 7.6 (i)). Somando o tempo de execução das instruções que implementam as operações descritas obtém-se a expressão incluída na equação 7.44. Aplicando esta expressão na equação 7.40 resulta o tempo de comunicação envolvido na leitura duma variável de *software* v_k por parte dum estado de *hardware* o_i.

$$Tread_i(v_k) = TwriteHW2HW + TreadHW2SW + TcomparacaoSW \; + \\ + (1 + nTransf(v_k)) * TwriteSW2HW + nshifts(ordemHW, nTransf(v_k)) * TshiftSW \quad (7.44)$$

onde

- $\boldsymbol{TwriteHW2HW}$[9] ($\boldsymbol{TwriteSW2HW}$) designa o tempo que o *hardware* (*software*) demora a escrever num registo de *hardware*, definido na tabela 7.2;
- $\boldsymbol{TcomparacaoSW}$ é o tempo que demora a comparação de duas variáveis inteiras em *software*, definido na tabela 7.1.

Verifica-se que não é necessário calcular o tempo de execução em *software* do estado o_{com} ($Texec(o_{com})$) por duas razões: (i) este tempo é considerado no tempo de execução em *hardware* de o_i ($Texec(o_i)$), sob a forma do tempo de comunicação agora definido e (ii) como o_{com} e o_i funcionam em paralelo[10] e $Texec(o_i)$ é maior do que $Texec(o_{com})$, não é preciso considerar o tempo de execução de o_{com}. Deste modo, os estados criados com a mesma finalidade que o_{com} não participam directamente na estimação do tempo de comunicação do sistema.

Nas situações (ii) e (iii) da figura 7.6, aplicou-se o princípio que indica que quando os estados pertencentes ao corpo dum ciclo, ou dum construtor condicional, estão repartidos por *software* e por *hardware*, se deve replicar os nodos de controlo destes construtores: $o_{ciclo}*$ é uma réplica de o_{ciclo} e $o_{if}*$ é uma réplica de o_{if}. O estado o_{com} é sempre inserido antes de o_s e no caso de o_i estar incluído num construtor condicional, o_{com} é inserido no ramo correspondente à condição de ramificação do estado o_i que inicia a leitura (*condx* na figura 7.6).

Nas leituras de *software* para *hardware* e nos casos em que o estado o_i que inicia a leitura funciona em paralelo com outros estados atribuídos a *software*, ou seja o fluxo em que se insere o_i está totalmente atribuído a *hardware*, a leitura tem que se processar através do mecanismo de interrupção. Deste modo, é preciso substituir a operação de leitura de *software* para *hardware*, pelas operações que geram uma interrupção e pelo código de serviço à interrupção. Na figura 7.7, as operações que substituíram a operação $read(v)$ no estado o_i têm por objectivo: (i) indicar qual a variável que o *software* deve enviar para *hardware* e que a operação

[9] $TwriteHW2HW$ e $THW\_register\_write$ são designações equivalentes.

[10] O paralelismo não advém do modelo do sistema, resulta duma mudança de partição introduzida no fluxo de controlo durante o processo de partição.

é de escrita; esta operação só é necessária se ocorrerem múltiplas operações via interrupção, (ii) gerar uma interrupção e (iii) esperar pela conclusão da operação.

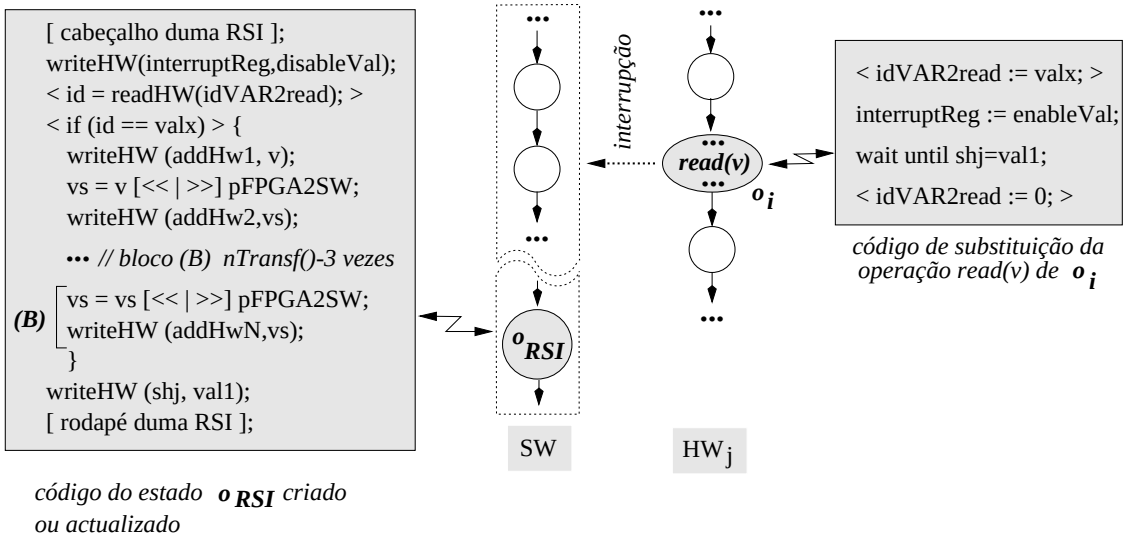


Figura 7.7: Ilustração da leitura duma variável de *software* para *hardware* através do mecanismo de interrupção.

O código de serviço à interrupção é introduzido no modelo do sistema sob a forma do estado o_{RSI} atribuído a *software*. Em termos de implementação, os estado criados com a mesma finalidade que o_{RSI} resultam numa rotina de serviço à interrupção. Com o código do estado o_{RSI} ilustrado na figura 7.7 pretende: (i) desactivar-se o sinal de interrupção, uma vez que no barramento PCI este sinal é sensível ao nível, (ii) verificar-se qual a variável a enviar para *hardware* e que a operação é de escrita; esta instrução só é necessária se ocorrerem múltiplas operações via interrupção, (iii) escrever-se a variável através de $nTransf(v_k)$ transacções; cada transacção pode exigir que se desloque a variável de modo a que os bits a enviar estejam alinhados com o componente alvo da escrita e (iv) sinalizar-se o fim da operação de escrita.

Aquando da leitura da variável v_k por parte do estado o_i, o tempo de execução do estado o_{RSI} ($TexecRSI(v_k)$) é definido pela equação 7.46 e o tempo de comunicação associado a o_i é obtido com a expressão da equação 7.48. Na implementação da estimação de métricas, o tempo de execução de o_{RSI} é atribuído a um dos estados de *software* que opere em paralelo com o_i.

$$TreadRSI(v_k) = TgestaoRSI + f1 * TreadHW2SW + TcomparacaoSW \; + \\ + (2 + nTransf(v_k)) * TwriteSW2HW + nshifts(ordemHW, nTransf(v_k)) * TshiftSW \quad (7.45)$$

$$TexecRSI(v_k) = TreadRSI(v_k) * numLeituras_i(v_k) * probLeitura_i(v_k) * FN_i \quad (7.46)$$

$$Tread_i(v_k) = f2 * TwriteHW2HW + TgestaoRSI + f1 * TreadHW2SW + \\ + TcomparacaoSW + (2 + nTransf(v_k)) * TwriteSW2HW + \\ + nshifts(ordemHW, nTransf(v_k)) * TshiftSW \quad (7.47)$$

$$Tcom_i(v_k) = Tread_i(v_k) * numLeituras_i(v_k) * probLeitura_i(v_k) \quad (7.48)$$

em que

- ***TgestaoRSI*** representa a soma do tempo envolvido na invocação duma rotina de serviço à interrupção (RSI), com o tempo usado no regresso ao programa interrompido e com o tempo de execução do cabeçalho mais o rodapé da RSI; como o tempo que se espera até uma interrupção ser servida não é previsível, esse tempo não é aqui considerado;
- $\boldsymbol{FN_i}$ é a frequência de execução do estado programa o_i;
- $\boldsymbol{f1}$ e $\boldsymbol{f2}$ são parâmetros que quantificam o facto de haver ou não múltiplas operações de escrita ou leitura via interrupção:

$$f1, f2 = \begin{cases} 0,1 & \rightarrow \text{ se o número de operações via interrupção} = 1 \\ 1,3 & \rightarrow \text{ se o número de operações via interrupção} > 1 \end{cases} \quad (7.49)$$

Leitura e escrita entre partições de *hardware* adjacentes

Como a comunicação entre *hardware* e *hardware* adjacente se processa através duma ligação dedicada, o protocolo utiliza apenas um endereço para identificar a variável alvo da operação, um sinal para definir se a operação é de escrita ou de leitura e os dados envolvidos na comunicação. O tempo necessário à execução duma operação de leitura ou de escrita entre partições de *hardware* adjacentes vai depender da largura da variável transferida, da largura reservada para dados na ligação entre as duas partições e do tempo que demora a transferência dum valor entre as partições. Embora o largura dos dados trocados entre partições adjacentes possa variar entre zero e o número de sinais da ligação física, em termos de estimação opta-se por um cenário realista em que se distribui de forma fixa o número de pinos pelos vários sinais a usar no protocolo de comunicação. Neste cenário, dos 33 pinos da ligação 16 foram dedicados aos dados ($pDadosF2FPGA$), 6 ao endereço que a partição envia para a partição adjacente ($pAddOutF2FPGA$), 6 ao endereço que a partição recebe da partição adjacente ($pAddInF2FPGA$) e 4 aos sinais de controlo da leitura e escrita ($pRdWrF2FPGA$). Conhecendo a largura do canal de comunicação, falta determinar o tempo que demora a escrever e ler um valor (até $pDadosF2FPGA$ bits) entre partições adjacentes.

O método seguido para obter os tempos de leitura ($TrdHa2H$) e escrita ($TwrH2Ha$) duma partição adjacente consistiu em analisar o caminho típico dos sinais de dados e de controlo envolvidos nessas operações. Tendo por base a figura 7.8 obtém-se um valor aproximado para o atraso nas operações de leitura e escrita.

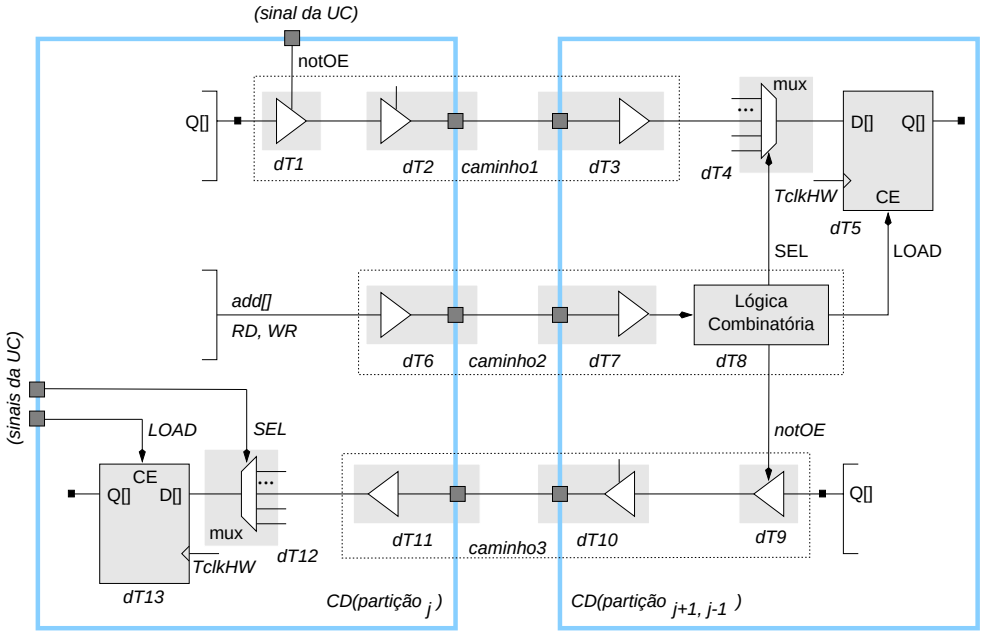


Figura 7.8: Atrasos nos caminhos de dados e de controlo usados nas operações de leitura e de escrita entre duas partições de *hardware* adjacentes.

Quando não se regista o valor lido da partição adjacente, o atraso na operação de leitura da partição $j+1$, ou $j-1$, para a partição j é aproximado pela equação 7.51.

$$dTr_{j+1,j-1\rightarrow j} = dT_{caminho2} + dT_{caminho3} = \ 2 * dT_{ibuf} + 2 * dT_{obuf} + dT_{logic} + dT_{tbuf} \qquad (7.50)$$

$$TrdHa2H \ = \ dTr_{j+1,j-1\rightarrow j} \ = \ \lceil \frac{80ns}{TclkHW} \rceil \ ciclos \qquad (7.51)$$

em que *caminho*3 (*caminho*2) é o caminho seguido pelos dados (sinais de controlo) na operação de leitura e os valores dos diversos atrasos dT estão definidos na tabela 7.4.

Quando se regista o valor lido da partição adjacente, o atraso na operação de leitura da partição $j+1$, ou $j-1$, para a partição j é aproximado pela equação 7.53.

$$dTr_{j+1,j-1\rightarrow j} = MAX(MAX(dT_{caminho2} + dT_{caminho3}, dT_{UC}) + dT_{mux}, dT_{UC}) + dT_{store} =$$

$$= \ dT_{caminho2} + dT_{caminho3} + dT_{mux} + dT_{store} =$$

$$= \ 2 * dT_{obuf} + 2 * dT_{ibuf} + dT_{logic} + dT_{tbuf} + dT_{mux} + dT_{store} \qquad (7.52)$$

$$TrdHa2H \ = \ dTr_{j+1,j-1\rightarrow j} \ = \ \lceil 1 + \frac{110ns}{TclkHW} \rceil \ ciclos \qquad (7.53)$$

Atraso	Descrição do atraso
$dT_1 = dT_9 = dT_{tbuf} \approx 10ns$	Atraso num *buffer tri-state*
$dT_2 = dT_6 = dT_{10} = dT_{obuf} \approx 10ns$	Atraso associado com a saída dum sinal para o exterior duma FPGA
$dT_3 = dT_7 = dT_{11} = dT_{ibuf} \approx 10ns$	Atraso associado com a entrada dum sinal a partir do exterior duma FPGA
$dT_4 = dT_{12} = dT_{mux} \approx 20ns$	Atraso num multiplexador
$dT_5 = dT_{13} = dT_{store} \approx TclkHW + 10ns$	Atraso desde a chegada dos dados à entrada dum registo até os valores aparecerem na saída, na pior situação
$dT_8 = dT_{logic} \approx 30ns$	Valor para o atraso na lógica combinatória que gera os sinais de controlo SEL, $LOAD$ e $notOE$
$dT_{UC} \approx 30ns$	Valor para o atraso na propagação de sinais de controlo gerados pela unidade de controlo (CPLD) duma partição

Tabela 7.4: Valor dos atrasos elementares utilizados no cálculo do tempo de execução das operações de escrita e de leitura entre duas partições de *hardware* adjacentes.

Conhecendo a largura da variável v_k a ler da partição adjacente ($larguraVar(v_k)$) e a largura reservada para dados na ligação entre partições de *hardware* adjacentes ($pDadosF2FPGA$), o número de transacções necessárias para ler v_k é dado por

$$nTransf(v_k) = \lceil \frac{larguraVar(v_k)}{pDadosF2FPGA} \rceil \tag{7.54}$$

Utilizando o valor de $TrdHa2H$ obtido com a equação 7.51 ou 7.53 pode calcular-se o tempo de comunicação associado à leitura da variável v_k por parte dum estado o_i.

$$Tread_i(v_k) = TrdHa2H * nTransf(v_k) \tag{7.55}$$

$$Tcom_i(v_k) = Tread_i(v_k) * numLeituras_i(v_k) * probLeitura_i(v_k) \tag{7.56}$$

O atraso numa operação de escrita na partição $j+1$, ou $j-1$, por parte da partição j é dado por

$$dTw_{j \to j+1,j-1} = MAX(MAX(dT_{caminho1}, dT_{caminho2}) + dT_4, dT_{caminho2}) + dT_5 \tag{7.57}$$

$$dTw_{j \to j+1,j-1} = MAX(dT_{UC} + dT_{tbuf} + dT_{obuf} + dT_{ibuf}, dT_{obuf} + dT_{ibuf} + dT_{logic}) + \\ + dT_{mux} + dT_{store} \tag{7.58}$$

$$TwrH2Ha = dTw_{j \to j+1,j-1} = \lceil 1 + \frac{90ns}{TclkHW} \rceil \ ciclos \tag{7.59}$$

em que o *caminho*1 (*caminho*2) designa o caminho seguido pelos dados (sinais de controlo) na operação de escrita.

Para uma variável v_k com largura $larguraVar(v_k)$ e um canal de comunicação com largura $pDadosF2FPGA$ para dados, o número de transacções necessárias para escrever v_k numa

partição de *hardware* adjacente é definido pela equação 7.54. Utilizando o valor de $TwrH2Ha$ obtido com a equação 7.59 calcula-se o tempo de comunicação associado à escrita da variável v_k, na partição adjacente, por parte dum estado o_i (equação 7.61).

$$Twrite_i(v_k) = TwrH2Ha * nTransf(v_k) \tag{7.60}$$

$$Tcom_i(v_k) = Twrite_i(v_k) * numEscritas_i(v_k) * probEscrita_i(v_k) \tag{7.61}$$

7.4.4 Tempo de Execução dos Estados Programa

Para estimar o tempo de comunicação dum estado programa considera-se o tempo despendido em todas as leituras e escritas por ele efectuadas. Ou seja, soma-se o valor de $Tcom(v)$ para cada uma das variáveis lidas com o valor de $Tcom(v)$ para cada uma das variáveis escritas pelo estado programa. Como se calcula o tempo de comunicação de uma forma incremental, em cada iteração só se corrige o tempo de comunicação de modo a considerar as alterações provocadas pelos objectos deslocados de partição.

Se um estado programa o_i é deslocado duma partição p_o para uma partição de destino p_d, efectuam-se as seguintes correcções no tempo de comunicação:

- ◇ colocar a zero o tempo de comunicação de o_i, exceptuando o tempo de comunicação pelo mecanismo de interrupção associado a outros estados;
- ◇ para cada variável v_k lida ou escrita por o_i, subtrair ao tempo de comunicação por interrupção de o_p[11], o tempo introduzido pela comunicação de o_i com v_k, quando o_i se encontrava na sua partição de origem p_o;
- ◇ para cada variável v_k lida ou escrita por o_i, adicionar ao tempo de comunicação de o_i, o tempo introduzido pela comunicação de o_i com v_k, quando o_i é deslocado para a partição destino p_d.

As tarefas anteriores são executadas pela função $ctlActualizarTcomDObjMudar$, que por sua vez recorre à função $actualizarTcomDevidoObjMudar$, incluída no apêndice G.

Por outro lado, quando uma variável v_k é deslocada da partição p_o para a partição de destino p_d, as correcções a introduzir no tempo de comunicação são as seguintes:

- ◇ em cada estado o_i que lê ou escreve v_k, subtrair ao tempo de comunicação desse estado, o tempo despendido em comunicação com v_k, quando ela se encontrava na sua partição de origem p_o;

[11] Este tempo só existe num estado o_i atribuído a *hardware* e é contabilizado num estado o_p atribuído a *software* e que funciona em paralelo com o_i.

- em cada estado o_i que lê ou escreve v_k, adicionar ao tempo de comunicação desse estado, o tempo despendido em comunicação com v_k, quando ela é deslocada para a partição de destino p_d.

As tarefas anteriores são executadas pela função *ctlActualizarTcomDVarMudar*, que por sua vez recorre à função *actualizarTcomDevidoVarMudar*, descrita no apêndice G.

O tempo de execução final dos estados programa, quando uma variável ou estado programa é deslocado de partição, é (re)calculado pela função *calcularTexecObjecto* após terem sido actualizados os tempos de comunicação (algoritmo da figura 7.9). As métricas utilizadas neste cálculo possuem o seguinte significado:

- $\boldsymbol{Texec(o_i)}$ designa o tempo de execução do estado programa o_i;
- $\boldsymbol{TexecSW(o_i)}$ ($\boldsymbol{TexecHW(o_i)}$) é o tempo de computação de o_i em *software* (*hardware*);
- $\boldsymbol{TcomLocal(o_i)}$ representa o tempo despendido por o_i na comunicação interna à partição a que está atribuído;
- $\boldsymbol{TexecRSI(o_i)}$ contabiliza o tempo despendido em comunicação, através do mecanismo de interrupção, atribuído ao estado o_i;
- $\boldsymbol{TcomViaSW(o_i)}$ designa o tempo que o_i despende em comunicação efectuada com intervenção do *software*, quando este estado se encontra atribuído a *hardware*;
- $\boldsymbol{TcomDir(o_i)}$ ($\boldsymbol{TcomEsq(o_i)}$) representa o tempo despendido por o_i em comunicação com a partição adjacente à direita (esquerda).

calcularTexecObjecto (o_i) $\equiv$

se $(atribuicao(o_i) = SW)$ então
  $Tcomp = TexecSW(o_i)$
  $Tcomm = TcomLocal(o_i)$
  $Texec(o_i) = Tcomp + Tcomm + \frac{TexecRSI(o_i)}{FN(o_i)}$
senão
  $Tcomp = TexecHW(o_i)$
  $Tcomm = maximo(TcomLocal(o_i), TcomViaSW(o_i), TcomDir(o_i), TcomEsq(o_i))$
  $Texec(o_i) = Tcomp + Tcomm$
fse
devolver $(Texec(o_i))$

Figura 7.9: Cálculo do tempo de execução dum estado programa.

7.4.5 Frequência de Execução dos Estados Programa

Para estimar a frequência de execução dos estados programa ($FN(o_i)$ ou FN_i) começa por efectuar-se uma análise estática, ou uma análise dinâmica (simulação), tendo em vista determinar o número de execuções dos ciclos (nc) e a probabilidade de ramificação nos construtores condicionais (pr). A análise estática aplica-se quando o número de execuções dos ciclos e a probabilidade de ramificação nos construtores condicionais assumem valores determinísticos. Caso contrário, funciona-se com os valores médios de nc e pr conseguidos com a análise dinâmica. Numa segunda fase determina-se um sistema de equações em que as variáveis são os valores para a frequência com que o fluxo de controlo passa pelos estados programa (FN_i) e pelos arcos que os interligam ($FA_{i,j}$). Na terceira fase, resolve-se o sistema de equações, por exemplo com o método de eliminação de Gauss, obtendo-se os valores dos vários FN_i e $FA_{i,j}$.

Para determinar o sistema de equações aplica-se o seguinte procedimento:

- ao nodo[12] de entrada do grafo PSMfg do sistema é atribuída uma frequência de execução $FN_0 = 1$, assim como à frequência com que o fluxo de controlo passa pelo arco que liga este nodo ao resto do sistema ($FA_{0,i} = 1$);

- a frequência com que o fluxo de controlo passa por um arco $a_{i,j}$ ($FA_{i,j}$), que parte do nodo o_i para o_j, é dada por $FA_{i,j} = FN_i * pr_{i,j}$, em que $pr_{i,j}$ designa a probabilidade de ramificação associada ao arco $a_{i,j}$;

- o somatório da probabilidade do fluxo de controlo passar pelos vários arcos $a_{k,i}$ que chegam ao nodo o_i é igual ao somatório da probabilidade do fluxo de controlo passar pelos vários arcos $a_{i,l}$ que partem de o_i, que por sua vez é igual à frequência de execução do nodo o_i (FN_i):

$$FN_i \quad (i \neq 0) = \sum_{a_{k,i} \in inEdges_i} FA_{k,i} = \sum_{a_{i,l} \in outEdges_i} FA_{i,l} \tag{7.62}$$

  sendo $inEdges_i$ e $outEdges_i$ o conjunto de arcos que chegam e partem do nodo o_i, respectivamente;

- num ciclo cujo corpo é executado nc vezes, a probabilidade de ramificação associada ao arco que entra no ciclo é $pr = \frac{nc}{nc+1}$ e a probabilidade de ramificação associada ao (único) arco que sai do ciclo é $pr = \frac{1}{nc+1}$; se existirem várias possibilidades de sair do ciclo, a soma da probabilidade de ramificação associada aos arcos de saída é que passa a ser $pr = \frac{1}{nc+1}$; em VHDL esta situação não é permitida;

[12] A designação "nodo" pode ser interpretada como "estado programa".

- conhecida a probabilidade associada a cada uma das n ramificações dum construtor condicional, de forma estática ou por simulação, a sua soma tem que ser 1:

$$pr_1 + pr_2 + ... + pr_n = 1 \tag{7.63}$$

7.4.6 Sincronismo entre Estados Programa

Os pontos de sincronismo entre estados programa podem introduzir não-determinismos no cálculo do tempo de execução dum grafo PSMfg, logo tornam-se num dos aspectos a analisar cuidadosamente quando se estima o desempenho dum sistema. Um ponto de sincronismo explícito resulta dum construtor `wait` presente no modelo PSM do sistema e um ponto de sincronismo implícito resulta duma mudança de partição no fluxo de controlo, introduzida pelo processo de partição. A estimativa do tempo de execução após um ponto de sincronismo exige que se conheça o tempo de execução de todos os estados intervenientes no sincronismo, ou seja, o tempo de execução em vários pontos do grafo PSMfg. O tratamento dos pontos de sincronismo será apresentado em pormenor na secção 7.4.8.

7.4.7 Escalonamento do Sistema

O escalonamento do grafo PSMfg obtido em cada iteração do processo de partição é afectado de forma determinante pela atribuição dos objectos às partições e pelas dependências escondidas na funcionalidade dos estados programa.

Dada a complexidade do problema de escalonar um grafo de fluxo de controlo, influenciado pelas dependências de dados que este grafo não representa explicitamente, optou-se por não efectuar um escalonamento explícito durante a estimação do desempenho do sistema. Contudo, o funcionamento da estratégia aplicada no cálculo do desempenho do sistema corresponde a um escalonamento implícito próximo do que se obtém com o algoritmo ASAP. Em consequência desta simplificação obtêm-se estimativas pessimistas para o desempenho, mas o tempo de cálculo é significativamente inferior. O paralelismo considerado nas estimativas do desempenho resulta apenas do escalonamento dos estados programa, das mudanças de partição introduzidas pelo processo de partição e da descrição do sistema.

7.4.8 Estratégia de Estimação do Desempenho dum Sistema

Para estimar o desempenho do sistema concebeu-se um modelo detalhado que representa as características do tipo de modelação utilizada com os sistemas. Como a modelação dos sistemas se faz com o meta-modelo PSM e a funcionalidade dos estados programa dos modelos

PSM é descrita em VHDL, o modelo de estimação do desempenho permite estimar o tempo de execução de construtores condicionais, ciclos, construtores paralelos e ciclos de espera.

Como se afirmou, não se utiliza escalonamento explícito ao nível do sistema, apenas ao nível do estado programa, resultando daqui uma redução do tempo de cálculo e estimativas tendencialmente pessimistas para o tempo de execução.

Sempre que possível, ou viável, em cada iteração do processo de partição utilizam-se estimativas obtidas por actualização, em vez de estimativas que exigem refazer os cálculos na totalidade.

Um aspecto relevante na estratégia de estimação tem a ver com a forma, simples e adequada, como o tempo de execução associado às quebras no fluxo de controlo do grafo é incluído na estimação do desempenho. As quebras representam mudanças de partição, entre dois nodos vizinhos, introduzidas no fluxo de controlo pelo processo de partição. Uma mudança de partição implica a inserção dum par de nodos do tipo *comutacao* e *sincronizacao* no fluxo de controlo. Em termos de estimação, esta inserção não é efectuada porque não é necessária nem recomendável. Não é necessária porque para considerar as quebras no fluxo basta apenas comparar a partição a que os nodos vizinhos estão atribuídos e não é recomendável porque (i) implicava um tempo de cálculo adicional para alterar o grafo sempre que, durante o processo de partição iterativo, uma quebra fosse introduzida ou removida e (ii) aumentava a complexidade do processo de partição, uma vez que se introduzia no grafo nodos especiais que, por exemplo, não poderiam ser seleccionados para deslocamento.

Como não se exige conhecer o tempo de execução em todos os nodos do grafo, a estratégia foi calcular o tempo de execução dum caminho do grafo apenas nos nodos em que se exige o valor do tempo de execução acumulado desde o início do grafo.

Nos próximos parágrafos aborda-se o método utilizado na estimação do tempo de execução dos construtores permitidos pela modelação dos sistemas.

Tempo de execução dum construtor condicional

Para obter o tempo de execução acumulado desde o início do grafo até ao final dum construtor condicional, aplica-se ao tempo de execução de cada ramo do construtor a probabilidade de ramificação que lhe está associada. O tempo de execução até ao nodo que define o fim do construtor condicional ($Texec_{FI}$ na equação 7.64) obtém-se somando os tempos pesados nos dois ramos do construtor com o tempo até ao início do construtor condicional ($Texec_{II}$).

$$Texec_{FI} = Texec_{II} + prob(ramo_1) * Texec(ramo_1) + prob(ramo_2) * Texec(ramo_2) \qquad (7.64)$$

Tempo de execução dum ciclo

O método aplicado na estimação do tempo de execução no final dum ciclo permite resultados precisos mas exige um tempo de cálculo elevado. O método consiste em simular duas iterações do ciclo e com os valores obtidos extrapolar o tempo de execução para a última iteração do ciclo. A explicação para o facto de se simularem duas iterações do ciclo e não uma, tem a ver com precisão e ciclos de espera. Como na primeira iteração dum ciclo, o fluxo de controlo vem do exterior do ciclo e nas restantes vem do próprio ciclo, a primeira iteração pode ter uma duração diferente das posteriores. A diferença entre a primeira iteração e as restantes ocorre quando o ciclo inclui ciclos de espera. Tal como o nodo que representa o controlo do ciclo é executado mais uma vez do que o corpo do ciclo, também o tempo de execução desse nodo é calculado mais uma vez do que o corpo do ciclo, ou seja, três vezes. A equação 7.65 define o tempo de execução no final dum ciclo com $nCiclos$ iterações, com base no tempo da segunda e da terceira iterações para o nodo que representa o controlo do ciclo.

$$Texec_{CC}[nCiclos] = Texec_{CC}[3] + (nCiclos - 2) * (Texec_{CC}[3] - Texec_{CC}[2]) \tag{7.65}$$

Tempo de execução dum construtor paralelo

Segundo o meta-modelo PSM, o fluxo de execução só avança dum construtor paralelo para o estado que se lhe segue quando todos os ramos do construtor terminarem a sua execução. Este comportamento tem como consequência que o tempo de execução dum construtor paralelo coincide com o maior tempo de entre todos os ramos do construtor. Deste modo, o tempo de execução no final dum construtor paralelo com $nRamos$ ramos é calculado com a equação 7.66, sendo $Texec_{IP}$ o tempo de execução no início do construtor paralelo.

$$Texec_{FP} = Texec_{IP} + maximo\ {}^{nRamos}_{nr=1}\ [Texec(ramo_{nr})] \tag{7.66}$$

Tempo de execução dum ciclo de espera

Os ciclos de espera explícitos resultam de construtores *wait* presentes no modelo do sistema e assumem uma de três alternativas: *wait*, *wait until f(s_1,s_2,...,s_n)* ou *wait on* s_1,s_2,...,s_n. Excluindo a alternativa *wait*, que representa um fluxo de controlo que termina nesse construtor, o tempo de execução das outras alternativas depende não só do tempo de execução do estado programa que representa o ciclo de espera, mas também dos estados que geram eventos nos sinais s_1 a s_n e tornam a expressão $f(s_1, s_2, ..., s_n)$ verdadeira. Em termos de representação PSMfg um ciclo de espera explícito é traduzido num estado programa do tipo *espera* e um ou mais estados do tipo *activacao*. Por seu lado, os ciclos de espera implícitos resultam de mudanças de partição no fluxo de controlo, introduzidas pelo processo de partição. Num grafo PSMfg, um ciclo de espera implícito é representado por um estado do tipo *sincronizacao* e

um ou mais estados do tipo *comutacao*. Do ponto de vista da estimação, os dois tipos de ciclo são tratados exactamente do mesmo modo.

A funcionalidade dum ciclo de espera também pode ser vista como um ponto de sincronismo entre o estado de *espera* (*sincronizacao*) e os estados de *activacao* (*comutacao*).

Mostrou-se que a estimação do tempo de execução dum ciclo de espera exige o conhecimento dos estados que activam os sinais intervenientes na expressão desse ciclo. Numa modelação de alto nível, a realização desta tarefa não é trivial, pelo facto de exigir bastante informação de nível inferior: o tipo de expressão utilizada no ciclo de espera, os sinais incluídos nessa expressão, os valores esperados em cada sinal, que objectos escrevem os valores esperados nos sinais e em que instante os valores são escritos. Os nodos do tipo *espera*, *sincronizacao*, *activacao* e *comutacao* foram incluídos no meta-modelo PSMfg com o objectivo de facilitar a tarefa de estimação do tempo de execução nos ciclos de espera. Para complementar esta ajuda (i) caracterizaram-se as variantes de ciclo de espera para as quais o tratamento automático se mostrou exequível e (ii) definiu-se um conjunto de regras para identificar os estados que intervêm em cada ciclo de espera.

Considere-se que o estado programa que impõe o ciclo de espera é E_s, que o tempo de execução do fluxo de E_s até este estado é $Texec'(E_s)$ (excluindo o tempo de espera) e que o tempo de execução do fluxo de E_s, até ao instante em que este estado termina a sua execução, é $Texec(E_s)$. Neste contexto e tendo em consideração a expressão utilizada no ciclo de espera, caracterizaram-se os seguintes **tipos de sincronismo**:

- ◇ tipo ***single***
  - ciclo de espera: *wait until* $s_1 = valor_1$ (ou *wait on* s_1);
  - definição: o estado programa E_s espera pela próxima activação do sinal s_1 com o valor $valor_1$ (ou com o valor complementar do actual);
  - $Texec(E_s)$: é o menor tempo de execução dum estado programa que seja superior a $Texec'(E_s)$, de entre a lista de estados do tipo *activacao* (ou *comutacao*) que sincronizam com E_s; as regras que definem os estados que sincronizam com E_s, ou seja, os estados que participam no ciclo de espera imposto por E_s, serão apresentadas mais à frente;
- ◇ tipo ***multi-AND***
  - ciclo de espera: *wait until* $s_1 = valor_1$ *and* $s_2 = valor_2$ *and ... and* $s_n = valor_n$ (ou *wait on* $s_1, s_2, ..., s_n$);
  - definição: o estado programa E_s espera até todos os sinais s_1 a s_n terem sido activados com os valores $valor_1$ a $valor_n$ esperados (ou activados com o conjunto de

valores complementares dos actuais), desde o instante em que os sinais começaram a ser "observados" por E_s;

- $Texec(E_s)$: definindo $Tmin_i$ como sendo o menor tempo de execução superior a $Texec'(E_s)$ para todos os estados do tipo *activacao* (ou *comutacao*) que sincronizam com E_s e escrevem $valor_i$ (ou o valor complementar do valor actual de s_i) no sinal s_i, o valor de $Texec(E_s)$ coincide com o maior valor no conjunto de tempos $\{ Tmin_1, ..., Tmin_n \}$;

◇ tipo ***multi-OR***

- ciclo de espera: *wait until* $s_1 = valor_1$ *or* $s_2 = valor_2$ *or ...* *or* $s_n = valor_n$;
- definição: o estado programa E_s espera até que pelo menos um sinal s_i do conjunto $\{s_1, ..., s_n \}$ seja activado com o valor esperado ($valor_i$), desde o instante em que os sinais começaram a ser "observados" por E_s;
- $Texec(E_s)$: definindo $Tmin_i$ como sendo o menor tempo de execução superior a $Texec'(E_s)$ para todos os estados do tipo *activacao* (ou *comutacao*) que sincronizam com E_s e escrevem $valor_i$ no sinal s_i, o valor de $Texec(E_s)$ coincide com o menor valor no conjunto de tempos $\{ Tmin_1, ..., Tmin_n \}$;

◇ tipo ***mixed***

- ciclo de espera: *wait until* $s_1 = valor_1$ *and* $s_2 = valor_2$ *and ...* *or* $s_p = valor_p$ *or* $s_q = valor_q$ *or ...*;
- dada a elevada complexidade exigida pelo tratamento rigoroso deste tipo de expressão, optou-se por considerar este tipo de sincronismo como *multi-AND*; deste modo, a estimativa de $Texec(E_s)$ assume o valor esperado na pior situação.

A identificação dos estados do tipo *activacao* ou *comutacao* que intervêm no ciclo de espera imposto por um estado do tipo *espera* ou *sincronizacao*, aplica as seguintes regras:

Regra EpS1 *Correspondência no tipo de estados*

Se o estado que impõe um ciclo de espera é do tipo *espera* (*sincronizacao*), só os estados do tipo *activacao* (*comutacao*) podem intervir nesse ciclo.

Regra EpS2 *Correspondência nos sinais operados*

Se o estado que impõe um ciclo de espera inspecciona os sinais s_1 a s_n, só os estados que activam pelo menos menos um dos sinais inspeccionados podem intervir nesse ciclo.

Regra EpS3 *Não dependência entre estados*

> Se o estado que impõe um ciclo de espera está inserido num ramo r_j ($1 \leq r_j \leq nr$) dum construtor paralelo com nr ramos, só os estados inseridos num ramo distinto de r_j e simultaneamente um ramo que pertence ao mesmo construtor paralelo, podem intervir nesse ciclo. Um ciclo de espera só faz sentido se o estado que impõe o ciclo estiver inserido num construtor paralelo.

Regra EpS4 *Precedência entre inspecção e activação dos sinais*

> Se o estado que impõe um ciclo de espera inicia a inspecção dos sinais no instante T_e, só os estados que activem um ou mais sinais após o instante T_e podem intervir nesse ciclo.

Regra EpS5 *Correspondência entre valor esperado e activado*

> Se um estado impõe um ciclo em que se espera até o sinal s_1 possuir o valor v_1 e/ou o sinal s_2 possuir o valor v_2 e/ou ... e/ou o sinal s_n possuir o valor v_n, apenas os estados que escrevem v_1 no sinal s_1 e/ou v_2 no sinal s_2 e/ou ... e/ou v_n no sinal s_n, podem intervir nesse ciclo.

Esta regra foi relaxada para permitir que estados, em que não se conhece estaticamente o valor que escrevem nos sinais, possam intervir nos ciclos de espera.

Para garantir segurança no cálculo do tempo de execução dos ciclos de espera, deve verificar-se a seguinte condição:

Regra EpS6 *Adequação entre o número de inspecções e de activações dos sinais*

> Se o estado que impõe um ciclo de espera é executado fn_e vezes, só os estados que são executados pelo menos fn_e vezes intervêm nesse ciclo.

7.4.9 Implementação da Estimação do Desempenho dum Sistema

Nesta secção descrevem-se os aspectos mais relevantes na implementação da estimação do desempenho dum sistema. Começa por apresentar-se as definições utilizadas nesta implementação.

Ponto de paragem

Um ponto de paragem é todo o nodo do grafo do sistema em que é necessário calcular o tempo de execução acumulado desde o início do grafo. Os pontos de paragem são nodos de todos os

tipos, excepto varáveis (porque não representam estados programa) e nodos do tipo *normal* (porque não afectam o fluxo de controlo e não intervêm nos ciclos de espera). O número de pontos de paragem é calculado pela função $searchNumberGraphPP$ e a informação relevante para o seu cálculo é (i) a identificação do fluxo (ou caminho) em que ocorre cada ponto de paragem, (ii) a frequência de execução de cada ponto de paragem, obtida com o contributo dos construtores condicionais e ciclos em que se insere o ponto de paragem e (iii) a indicação de que o ponto de paragem está inserido, ou não, no corpo dum ciclo.

Fluxo de controlo dum grafo

Um fluxo de controlo, ou simplesmente fluxo, é um caminho dum grafo definido por todos os nodos e arcos encontrados (i) desde o início até ao fim dum ramo dum construtor paralelo ou (ii) desde o início até ao fim do grafo. O número de fluxos dum grafo é calculado pela função $calcMaxFluxId$ e para cada fluxo, a função $calcDescendentFluxes$ obtém os fluxos que descendem doutro fluxo.

Fluxo descendente

Um fluxo descendente é um fluxo totalmente inserido dentro do fluxo de que se diz descender.

Sub-fluxo dum grafo

Um sub-fluxo designa um segmento de fluxo, definido por todos os nodos e arcos desde um ponto de paragem (exclusive) até ao próximo ponto de paragem (inclusive) desse fluxo. A definição de sub-fluxo é ilustrada na figura 7.10.

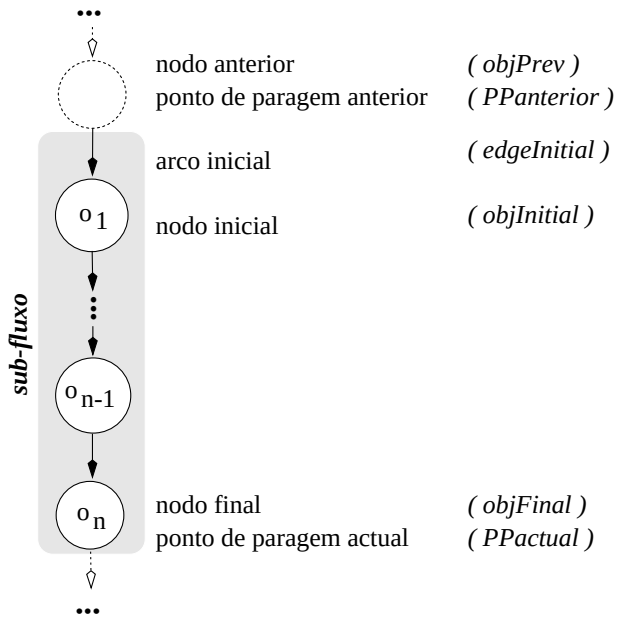


Figura 7.10: Definição de sub-fluxo dum grafo.

Ponto de sincronismo

Tal como foi mencionada anteriormente, um ponto de sincronismo corresponde a um ciclo de

espera imposto por um estado do tipo *espera* ou *sincronizacao* e no qual participam um ou mais estados de *activacao* ou *comutacao*. O número de pontos de sincronismo é calculado pela função $searchNumberGraphSP$ e o número de estados que participam em todos os pontos de sincronismo é obtido pela função $searchNumberGraphESeAC$. Para determinar quais os estados de *activacao* ou *comutacao* que intervêm em cada ponto de sincronismo, bem como o ponto do grafo em que o fluxo que conduz a esses estados diverge do fluxo que conduz ao estado que impõe o sincronismo, utiliza-se a função *calcSynchPoints* incluída na figura 7.11. A função *calcSynchPoints* aplica as regras EpS1 a EpS6, excepto a regra EpS4, tendo a regra EpS6 sido convertida em duas regras mais restritivas: EpS6a e EpS6b.

```
calcSynchPoints () ≡

 para (cada nodo objES do grafo) fazer
  tipoES = tipoObj(objES)
  se ((tipoES=espera) OU (tipoES=sincronizacao)) então // ponto de sincronismo
   revPath[objES] = caminho inverso desde objES até ao início do grafo
   para (cada variável objVar lida por objES) fazer
    para (cada nodo objAC que escreve a variável objVar) fazer // regra EpS2
     tipoAC = tipoObj(objAC)
     se ( ((tipoES=espera) E (tipoAC=activacao)) OU
        ((tipoES=sincronizacao) E (tipoAC=comutacao)) ) então // regra EpS1
      se ((valorEscritoAC é desconhecido) OU ((valorEscritoAC=valorEsperadoES)
         E (valorEscritoAC é conhecido) ) ) então // regra EpS5
       se (revPath[objAC] não foi calculado) então
        revPath[objAC] = caminho inverso desde objAC até ao início do grafo
       fse
       split = ponto de divergência entre revPath[objES] e revPath[objAC]
       se (split != NULL) então // regra EpS3
        se (FN(objES) = FN(objAC)) então // regra EpS6a
         nES = número de construtores que contribui para FN(objES)
         nAC = número de construtores que contribui para FN(objAC)
         se (nES = nAC) então
          iguais = verdade
          para (n=1 até nES) fazer
           se (n-ésimo construtor que contribui para FN(objES) !=
              n-ésimo construtor que contribui para FN(objAC)) então
            iguais = falso
           fse
          fpara
          se (iguais = verdade) então // regra EpS6b
           adicionar objAC à lista de participantes no sincronismo com objES
          fse
         fse
        fse
       fse
      fse
     fse
    fpara
   fpara
  fse
 fpara
```

Figura 7.11: Função *calcSynchPoints* que determina que estados do tipo *activacao* ou *comutacao* participam em cada ponto de sincronismo.

Regra EpS6a *Igualdade entre o número de inspecções e de activações dos sinais*

> Se o estado que impõe um ciclo de espera é executado fn_e vezes, só os estados que são executados fn_e vezes podem intervir nesse ciclo.

Regra EpS6b *Equivalência entre a estrutura dos caminhos de inspecção e de activação*

> Para que um estado do tipo *activacao* ou *comutacao* possa intervir num ciclo de espera, a estrutura do caminho definido desde o início do grafo do sistema até este estado tem que ser igual à estrutura do caminho definido desde o início do grafo do sistema até ao estado que impõe o ciclo.

Por estrutura dum caminho deve entender-se a sequência de construtores por que passa o fluxo de controlo, desde o início até ao fim do caminho. Os construtores relevantes para a regra EpS6b são aqueles que afectam a frequência de execução dos estados do tipo *activacao* ou *comutacao*, ou seja, os ciclos e os construtores condicionais.

Estimar o tempo de execução dum sistema equivale a calcular o tempo de execução no fluxo principal[13] do grafo desse sistema. Dispondo da função $calcTexecFlux$, que calcula o tempo de execução dum fluxo, e da informação relativa a pontos de paragem, pontos de sincronismo e fluxos, o procedimento $calcTexecSystem$ usado na estimação do tempo de execução do sistema limita-se a aplicar a função $calcTexecFlux$ ao fluxo principal do grafo do sistema (figura 7.12).

calcTexecSystem () ≡

Limpar a informação resultante do cálculo anterior do tempo de execução
$fluxId$ = fluxo iniciado no arco de saída do nodo de entrada no sistema

$Texec$ = calcTexecFlux ($fluxId$)

devolver ($Texec$)

Figura 7.12: Função $calcTexecSystem$ que calcula o tempo de execução dum sistema.

Estimar o tempo de execução dum fluxo

No início de cada fluxo, a função $calcTexecFlux$, apresentada na figura 7.13, considera três alternativas: o tempo de execução do fluxo já foi totalmente calculado, o cálculo do tempo de execução do fluxo não foi iniciado ou então o tempo de execução do fluxo foi parcialmente calculado. Na primeira alternativa só é preciso devolver um valor previamente calculado.

[13] Fluxo principal é o fluxo que começa no nodo de entrada do grafo e termina no nodo de saída do grafo.

```
calcTexecFlux (fluxId) ≡

  se (Texecução do fluxo fluxId já foi totalmente calculado) então
    devolver (Texecução do fluxo fluxId obtido na iteração anterior)

  senão se (cálculo de Texecução do fluxo fluxId não foi iniciado) então
    Texec = Texecução de PPanterior // ver figura 7.10

  senão se (cálculo de Texecução do fluxo fluxId já foi iniciado) então

    se (último nodo calculado no fluxo não é do tipo inicioParalelo) então
      Definir um sub-fluxo a partir do último nodo calculado no fluxo

    senão // último nodo calculado no fluxo é do tipo inicioParalelo
      para ( (cada fluxo dFluxId descendente do fluxo actual e
           iniciado no nodo do tipo inicioParalelo) E
           (enquanto o nodoProcurado não for atingido) ) fazer
        Texec = calcTexecFlux (dFluxId)
      fpara

      se (construtor paralelo foi totalmente calculado) então
        Texec = maximo (Texecução para cada ramo do construtor)
      senão // Terminou-se o cálculo do fluxo num nodo do tipo activacao
            // ou comutacao interveniente num ponto de sincronismo
        devolver (Texec)
      fse
    fse
  fse
  enquanto ( (não se chega ao fim do fluxo actual) OU
      (não se atinge o ponto de sincronismo procurado) ) fazer
    // De acordo com o tipo do nodo final do sub-fluxo (TNF), utiliza-se a função
    // que calcula Texecução num sub-fluxo e é adequada ao tipo de nodo TNF.
    // As diferentes funções encontram-se nas figuras H.2 a H.9
    Texec = calcTexecSubFlux_TNF (Texec)
  fenquanto

  devolver (Texec)
```

Figura 7.13: Função *calcTexecFlux* que calcula o tempo de execução no final dum fluxo do grafo.

Quando for necessário estimar o tempo de execução no final do fluxo, a função calcula o tempo de execução em cada um dos sub-fluxos que compõem o fluxo, até atingir o fim do fluxo ou o nodo procurado. A função pára num nodo do tipo *activacao* ou *comutacao*, participante num ponto de sincronismo, quando é invocada com o objectivo de devolver o tempo de execução nesse nodo, ou então pára no final do fluxo quando o objectivo é calcular o tempo de execução no final do fluxo. A função funciona de forma recursiva sempre que encontra um nodo que representa o início dum construtor paralelo, sendo aplicada a cada um dos fluxos que descende do fluxo actual e começando neste nodo. Como os nodos do modelo PSMfg são bastante heterogéneos, o cálculo do tempo de execução em cada sub-fluxo faz-se com o procedimento adequado ao tipo do nodo final desse sub-fluxo:

- *calcTexecSubFlux_II* quando o sub-fluxo termina num nodo do tipo *inicioIf*;

- ◇ $calcTexecSubFlux\_FI$ quando o sub-fluxo termina num nodo do tipo $fimIf$;
- ◇ $calcTexecSubFlux\_CC$ quando o sub-fluxo termina num nodo do tipo *controloCiclo*;
- ◇ $calcTexecSubFlux\_A\_C$ quando o sub-fluxo termina num nodo do tipo *activacao* ou *comutacao*;
- ◇ $calcTexecSubFlux\_E\_S$ quando o sub-fluxo termina num nodo do tipo *espera* ou *sincronizacao*;
- ◇ $calcTexecSubFlux\_IP$ quando o sub-fluxo termina num nodo do tipo *inicioParalelo*;
- ◇ $calcTexecSubFlux\_FP$ quando o sub-fluxo termina num nodo do tipo $fimParalelo$;
- ◇ $calcTexecSubFlux\_FS$ quando o sub-fluxo termina num nodo do tipo $fimSistema$.

As tarefas relevantes executas pelas funções $calcTexecSubFlux\_II$ a $calcTexecSubFlux\_FS$ são:

- ◇ chamar a função $calcTexecSubFlux$, descrita no apêndice H, que adiciona o tempo de execução de todos os nodos do sub-fluxo com o tempo associado a todas as mudanças de partição que ocorrem no sub-fluxo; o tempo envolvido numa mudança de partição é obtido com o procedimento $TpartitionComutation$ do apêndice H, que aplica o principio apresentado na figura 7.4: a duração duma mudança de partição inclui o tempo de execução dum nodo do tipo *comutacao*, na partição de origem da mudança, e o tempo de execução dum nodo do tipo *sincronizacao*, na partição de destino da mudança; em concreto, o nodo de *comutacao* escreve um valor v_m num sinal s_m localizado na partição de origem (ou destino) e o nodo de *sincronizacao* espera até o sinal remoto (ou local) s_m conter o valor v_m;
- ◇ gerir os valores do tempo de execução, relativos ao nodo final do sub-fluxo[14] e obtidos em cada ramo que chega ao nodo ou em cada iteração em que se visita o nodo; gerir significa guardar os valores intermédios e combinar estes valores de modo a obter o tempo de execução no final dum construtor condicional, dum ciclo ou dum construtor paralelo; nesta tarefa aplica-se a equação 7.64, 7.65 ou 7.66;
- ◇ preparar a função $calcTexecFlux$ para o próximo sub-fluxo, ajustando os nodos e os pontos de paragem ilustrados na figura 7.10;
- ◇ executar outras tarefas específicas do tipo de nodo final do sub-fluxo; por exemplo, $calcTexecSubFlux\_IP$ calcula o tempo de execução em cada um dos ramos do construtor paralelo e $calcTexecSubFlux\_E\_S$ actualiza o tempo de execução do sub-fluxo

[14] $objFinal$ na figura 7.10.

com o tempo despendido pelo nodo no ciclo de espera, recorrendo para isso à função $calcTexecAfterSyncESnode$ do apêndice H.

Para ilustrar o cálculo do tempo de execução num sub-fluxo, apresenta-se na figura 7.14 a função $calcTexecSubFlux\_II$, enquanto as restantes funções podem ser consultadas no apêndice H.

```
calcTexecSubFlux_II () ≡

  se (nodo final do sub-fluxo está inserido num ciclo) então
    [ ... ]

  senão // O nodo final do sub-fluxo não está inserido num ciclo
    dT = calcTexecSubFlux()
    sTexec = Texec[PPanterior] + dT
    Texec[PPactual] = sTexec
    Avançar para o próximo sub-fluxo
  fse
  devolver (sTexec)
```

Figura 7.14: Parte da função $calcTexecSubFlux\_II$ que calcula o tempo de execução dum sub-fluxo terminado por um nodo do tipo $inicioIf$.

Estimar o tempo de execução num ciclo de espera

Para identificar os estados que afectam o tempo de execução do ciclo de espera, imposto por um estado E_s do tipo *espera* ou *sincronizacao*, é necessário (i) conhecer as variáveis[15] inspeccionadas pelo estado E_s, (ii) seleccionar os estados que escrevem nestas variáveis e podem participar no sincronismo com E_s, aplicando as regras EpS1-EpS3, EpS5 e EpS6[16] e (iii) refinar a selecção de estados, aplicando a regra EpS4.

Para estimar o tempo de execução do ciclo de espera associado a E_s consideram-se duas contribuições: uma resulta dos estados do tipo *activacao* ou *comutacao* que escrevem valores determinísticos nas variáveis inspeccionadas por E_s (tempo T_{det}) e a outra resulta dos estados do tipo *activacao* ou *comutacao* que escrevem valores não determinísticos nas variáveis inspeccionadas por E_s (tempo T_{nonDet}). A estimativa de $Texec(E_s)$, na pior situação e para cada tipo de ciclo de espera, é assim calculada:

- ciclo de espera do tipo *single*

$$Texec(E_s) = Tsync(single, E_s, s_1, v_1) = maximo\ [T_{det}(E_s, s_1, v_1), T_{nonDet}(E_s, s_1)] \quad (7.67)$$

[15] Em termos de modelação em VHDL o tipo de estrutura de dados envolvida nos ciclos de espera tem que ser um sinal, não uma variável. Ao longo deste documento utiliza-se o termo "variável", para designar uma estrutura de dados abstracta, desde que não seja relevante identificar a correspondência ao "sinal" ou à "variável" do VHDL.

[16] A selecção foi previamente efectuada pela função $calcSynchPoints$.

em que

- $\boldsymbol{T_{det}(E_s, s_1, v_1)}$ é o tempo de execução dos estados E_a^i, seleccionados pelas regras EpS1-EpS4 e EpS6 e que escrevem o valor determinístico v_1 na variável s_1;

$$T_{det}(E_s, s_1, v_1) = minimo_i\ [Texec(E_a^i)] \tag{7.68}$$

- $\boldsymbol{T_{nonDet}(E_s, s_1)}$ é o tempo de execução dos estados E_a^j seleccionados pelas regras EpS1-EpS4 e EpS6 e que escrevem um valor não determinístico na variável s_1;

$$T_{nonDet}(E_s, s_1) = maximo_j\ [Texec(E_a^j)] \tag{7.69}$$

◇ ciclo de espera do tipo *multi-AND*

$$\begin{aligned} Texec(E_s) &= Tsync(multiAND, E_s, s_1, v_1, ..., s_n, v_n)\ = \\ &= maximo_{s_k=s_1}^{s_n}\ [Tsync(single, E_s, s_k, v_k)] \end{aligned} \tag{7.70}$$

em que $Tsync(single, E_s, s_k, v_k)$ é calculado com a equação 7.67;

◇ ciclo de espera do tipo *multi-OR*

$$\begin{aligned} Texec(E_s) &= Tsync(multiOR, E_s, s_1, v_1, ..., s_n, v_n)\ = \\ &= minimo_{s_k=s_1}^{s_n}\ [Tsync(single, E_s, s_k, v_k)] \end{aligned} \tag{7.71}$$

em que $Tsync(single, E_s, s_k, v_k)$ é calculado com a equação 7.67;

◇ ciclo de espera do tipo *mixed*

$$\begin{aligned} Texec(E_s) &= Tsync(mixed, E_s, s_1, v_1, ..., s_n, v_n)\ = \\ &= Tsync(multiAND, E_s, s_1, v_1, ..., s_n, v_n) \end{aligned} \tag{7.72}$$

em que $Tsync(multiAND, E_s, s_1, v_1, ..., s_n, v_n)$ é calculado com a equação 7.70.

Aplicando os princípios apresentados nesta secção, a função $calcTexecAfterSyncESnode$ do apêndice H obtém uma estimativa para o tempo de execução dum nodo do tipo *espera* ou *sincronizacao*.

7.5 Estimação de Métricas ao Nível do Estado Programa

Com o processo de estimação ao nível de abstracção do estado programa pretende obter-se estimativas para métricas relativas aos objectos, ou seja, métricas para os estados programa e as variáveis do modelo PSMfg do sistema. Para cada estado programa estima-se o tempo de computação em *software*, o espaço ocupado em *hardware* pelas unidades funcionais, o espaço ocupado em *hardware* pelos multiplexadores a colocar na entrada das variáveis escritas, o tempo de computação em *hardware*, as variáveis lidas, escritas e que precisam dum multiplexador na sua entrada. Por seu lado, para cada variável calcula-se o espaço ocupado em *hardware* e os estados programa que a lêem e escrevem. Estas estimativas são de nível inferior às que se obtêm ao nível do sistema e estão envolvidas no seu cálculo. A estratégia de estimação que se empregou neste nível teve por objectivo conseguir estimativas de elevada precisão.

7.5.1 Estimação de Métricas de *Software*

A estimação de métricas de *software*, ao nível de abstracção do estado programa, decorre nas duas tarefas a seguir apresentadas.

Tarefa T1

Para estimar o **tempo de computação em *software*** dum estado programa o_i efectua-se o escalonamento das operações deste estado para uma implementação em *software*, respeitando as dependências de dados existentes entre as operações presentes no código que descreve o comportamento de o_i. Ou seja, qualquer operação op_p incluída numa declaração $decl_p$, só pode ser escalonada quando todas as operações op_a, que escrevem algum dos argumentos de op_p e estão incluídas numa declaração anterior a $decl_p$, já tiverem sido escalonadas. Esta condição também pode ser definida à custa dum grafo de fluxo de dados, em que os nodos representam as operações e os arcos os argumentos das operações: uma operação op_p, representada pelo nodo n_p, só pode ser escalonada quando qualquer das operações representada por um nodo antecessor de n_p tiver sido escalonada.

Optando pelo modelo de *software* simples e uma arquitectura alvo uni-processador, cada operação é escalonada numa etapa de controlo distinta; se se aplicar o modelo de *software* melhorado e/ou uma arquitectura alvo multi-processador, o número de operações escalonadas na mesma etapa de controlo é determinado pela duração das operações, pelas dependências de dados, pelo número de processadores, pelos níveis de *pipeline* e pelo grau de superescalaridade de cada processador.

Se o estado o_i contiver ciclos e/ou construtores condicionais, aplica-se um procedimento similar

ao da secção 7.4.5 para calcular a frequência de execução das operações $op_{i,j}$ de o_i ($Fop_{i,j}$).

Tarefa T2

O tempo de computação do estado programa em *software* (***TexecSW*** (o_i)) é calculado com base: (i) no escalonamento e na frequência de execução $Fop_{i,j}$ das operações do estado o_i obtidos com a tarefa T1, (ii) no factor de optimização de código (relativamente ao tempo de execução) para o processador a usar na implementação (λ_T), (iii) no tempo de execução em *software* das operações, $TexecSW(op_{i,j})$ disponível no ficheiro tecnológico para o processador em causa e (iv) na frequência de relógio do processador ($F_{relogioCPU}$). Ao aplicar-se o modelo de *software* simples e uma arquitectura alvo uni-processador, o tempo de computação do estado o_i em *software* é definido pela equação 7.73.

$$TexecSW(o_i) = \frac{\lambda_T}{F_{relogioCPU}} * \sum_{op_{i,j} \in o_i} [\ Fop_{i,j} * TexecSW(op_{i,j})\ ] \tag{7.73}$$

O escalonamento só é relevante para o cálculo de $TexecSW(o_i)$ quando se usa o modelo de *software* melhorado e/ou uma arquitectura alvo multi-processador. Nessa situação, para calcular $TexecSW(o_i)$ é preciso aplicar um procedimento mais complexo do que aquele que conduziu à equação 7.73. Nomeadamente, o valor de $TexecSW(o_i)$ obtém-se somando o tempo de execução da operação mais lenta escalonada em cada etapa de controlo do escalonamento de o_i (equação 7.74).

$$TexecSW(o_i) =$$

$$= \frac{\lambda_T}{F_{relogioCPU}} * \sum_{etapa_k \in schedule(o_i)} maximo_{op_{i,j} \in etapa_k} [\ Fop_{i,j} * TexecSW(op_{i,j})\ ] \tag{7.74}$$

O método de estimação proposto reduz o tempo necessário para calcular o tempo de computação em *software* mas apresenta duas limitações: (i) o modelo simples considera de forma simplificada os mecanismos equivalentes às optimizações de código efectuadas pelos compiladores, de modo a usufruir-se das potencialidades dos processadores actuais e (ii) ambos os modelos, simples e melhorado, tratam a nível local (do estado programa) o problema do escalonamento que, para ser resolvido da forma mais correcta, deveria ser tratado ao nível do sistema. Ou seja, para obter estimativas mais precisas para o tempo de computação em *software* deveria tratar-se o problema totalmente ao nível do sistema, de modo a considerar a interacção entre os vários objectos atribuídos a *software*. Este procedimento obrigaria a refazer todos os cálculos em cada iteração do processo de partição, tendo como consequência um aumento significativo do tempo de cálculo.

7.5.2 Estimação de Métricas de *Hardware*

A estimação de métricas de *hardware* ao nível de abstracção do estado programa é apresentado sob a forma de um conjunto de tarefas executadas segundo uma ordem que não viola as dependências entre métricas.

Tarefa T3

A estimação de algumas métricas dos estados programa exige um escalonamento prévio das operações desses estados, para uma implementação em *hardware*. O escalonamento implementado aplica a estratégia ASAP, mas poderia ser um método baseado em listas de prioridade como o que se apresentou na figura 4.2, respeita as dependências de dados, considera que não existe partilha de unidades funcionais e que os recursos são ilimitados[17] e identifica o número de leituras, ou escritas, de variáveis externas à partição através das regras RW.

Para identificar o número de vezes que é preciso ler, ou escrever, uma variável do exterior duma partição definiram-se as seguintes regras:

Regras RW *Identificação do número de leituras (escritas) das variáveis externas*

Regra RW.1

As variáveis compostas[18] são lidas, ou escritas, do exterior da partição uma vez por cada iteração do ciclo em que a operação se inclui. A leitura ou escrita será efectuada apenas uma vez por ciclo se o elemento acedido não variar durante a totalidade das iterações do ciclo.

Regra RW.2

Num estado programa, uma variável simples só é lida, ou escrita, do exterior da partição uma única vez. A leitura ou escrita única mantém-se válida mesmo que o acesso se insira num ciclo, num ciclo aninhado dentro de outro ciclo ou num construtor condicional.

Regra RW.3

Quando uma variável, simples ou composta, é escrita e lida pelo mesmo estado programa, se a primeira operação de leitura for posterior à primeira operação de escrita dessa variável, a variável não precisa ser lida do exterior da partição.

[17] Apenas se considera que os recursos são ilimitados ao nível do estado programa.

[18] Em código VHDL sintetizável, o acesso a variáveis compostas ocorre essencialmente sob a forma de leitura ou escrita de um elemento dum *array*.

A estratégia de **escalonamento ASAP** aplicada a um grafo $G = \{V, E\}$ escalona cada nodo do grafo o mais cedo possível, ou seja, imediatamente após todos os nodos antecessores desse nodo terem sido escalonados. A figura 7.15 apresenta uma implementação do algoritmo de escalonamento ASAP [GDWL92]. Para cada nodo v_i do grafo, o algoritmo calcula o índice da etapa de controlo em que esse nodo deve ser escalonado ($indiceASAP_i$). Ou seja, se $indiceASAP_5 = 2$ o nodo v_5 deve ser escalonado na etapa de controlo s_2. O algoritmo começa por escalonar os nodos sem antecessores na etapa de controlo s_1 e iniciar a informação que indica quais os nodos já escalonados (*scheduled*). Para que o algoritmo funcione exige-se que a estrutura de dados $antecessores(v_i)$ contenha o conjunto de nodos antecessores imediatos de v_i. Após a fase inicial, o algoritmo percorre todos os nodos não escalonados e se todos os antecessores dum nodo v_i já estiverem escalonados, situação que ocorre quando a condição $allNodesScheduled()$ é verdadeira, o nodo é escalonado na etapa de controlo (mais cedo possível) calculada pela expressão $indiceASAP_i = max(antecessores(v_i), indiceASAP) + 1$. A função *max* devolve o índice da etapa de controlo mais posterior em que algum dos nodos antecessores imediatos de v_i foi escalonado.

Tarefa T4

Combinando a selecção de recursos resultante do escalonamento efectuado pela tarefa T3 com a informação do ficheiro tecnológico para a família de FPGAs a usar na implementação do caminho de dados das partições, estima-se o espaço ocupado pelas unidades funcionais necessárias ao estado programa o_i ($\boldsymbol{areaUF(o_i)}$). O ficheiro tecnológico, que contém informação sobre o espaço e a temporização de operações executadas em FPGAs da família XC4000X/XL, está sintetizado na tabela 7.7.

$$areaUF(o_i) = \sum_{op_{i,j} \in o_i} areaOp\ (typeOp\ (op_{i,j})) \tag{7.75}$$

sendo $areaOp(typeOp(op_{i,j})$ o espaço ocupado por uma unidade funcional que executa as operações do tipo $typeOp(op_{i,j})$.

Tarefa T5

Aplicando às operações $op_{i,j}$ dum estado programa um procedimento idêntico ao descrito na secção 7.4.5, estima-se a **frequência de execução das operações** ($\boldsymbol{Fop_{i,j}}$).

Tarefa T6

Com o escalonamento para *hardware* obtido pela tarefa T3, a frequência de execução das operações que se obteve com a tarefa T5 ($Fop_{i,j}$), o tempo de execução em *hardware* das operações disponível na tabela 7.7 ($TexecHW(op_{i,j})$) e a frequência de relógio utilizada no *hardware*

($FclkHw$), o tempo de computação do estado programa o_i em *hardware* ($\boldsymbol{TexecHW(o_i)}$) coincide com o maior atraso de entre todos os caminhos do grafo que descreve a funcionalidade de o_i (equação 7.76). Considera-se que se o estado programa incluir pontos de sincronismo, eles só podem ocorrer no final da descrição do estado programa. Esta é uma pré-condição para a definição dos estados programa do meta-modelo PSMfg.

$$TexecHW(o_i) = \\ = \frac{1}{FclkHw} * \sum_{etapa_k \in schedule(o_i)} maximo_{op_{i,j} \in etapa_k} [\, Fop_{i,j} * TexecHW(op_{i,j}) \,] \qquad (7.76)$$

```
escalonamentoASAP (V) ≡

  // V é o conjunto de nodos do grafo a escalonar
  para (v_i ∈ V) fazer
    se (antecessores(v_i) = ∅) então
      indiceASAP_i = 1
      V = V − {v_i}
      scheduled_i = Verdade
    senão
      scheduled_i = Falso
    fse
  fpara
  enquanto (V ≠ ∅) fazer
    para (v_i ∈ V) fazer
      se (allNodesScheduled(antecessores(v_i), scheduled) = Verdade) então
        indiceASAP_i = max(antecessores(v_i), indiceASAP) + 1
        V = V − {v_i}
        scheduled_i = Verdade
      fse
    fpara
  fenquanto
  devolver (indiceASAP)

allNodesScheduled (antecessores, scheduled) ≡
  allTrue = Verdade
  para (v_i ∈ antecessores) fazer
    se (scheduled_i = Falso) então
      allTrue = Falso
    fse
  fpara
  devolver (allTrue)

max (antecessores, indiceASAP) ≡
  i1 = índice do primeiro nodo no conjunto antecessores
  indiceMax = indiceASAP_i1
  para (v_i ∈ antecessores) fazer
    se (indiceASAP_i > indiceMax) então
      indiceMax = indiceASAP_i
    fse
  fpara
  devolver (indiceMax)
```

Figura 7.15: Algoritmo de escalonamento ASAP.

Tarefa T7

Conhecendo o grafo escalonado do estado programa o_i, obtido na tarefa T3, o qual representa as operações de leitura de variáveis, a aplicação das regras RW permite calcular as variáveis que o estado o_i precisa de ler ($\boldsymbol{readVAR(o_i)}$).

A leitura duma variável, ou sinal, pode ocorrer em VHDL nas seguintes situações: no lado direito das atribuições, nas condições dos construtores condicionais, como parâmetro das funções, como parâmetro de entrada ou bidireccional dos procedimentos, no controlo dos ciclos, na "lista de sensibilidade" dos processos e na condição associada a um ciclo de espera.

Tarefa T8

Como o grafo escalonado do estado programa o_i representa as operações de escrita de variáveis, aplicando as regras RW determinam-se as variáveis que o estado o_i precisa de escrever ($\boldsymbol{writeVAR(o_i)}$).

A escrita de variáveis, ou sinais, pode ocorrer em VHDL nas seguintes situações: no lado esquerdo das atribuições, para guardar o valor devolvido pelas funções e como parâmetro de saída ou bidireccional dos procedimentos.

Tarefa T9

Depois de conhecer as variáveis escritas num estado programa o_i, lista $writeVAR(o_i)$ obtida na tarefa T8, a aplicação das regras MV indica que variáveis escritas por o_i precisam dum multiplexador na sua entrada ($\boldsymbol{muxes(o_i)}$) e o número de sinais de entrada no multiplexador da variável v_j ($\boldsymbol{muxes(o_i).inputsMUX(v_j)}$). O número de sinais de selecção do multiplexador a colocar na entrada da variável v_j, $muxes(o_i).ncontrol(v_j)$, é obtido pela equação 7.77.

$$muxes(o_i).ncontrol(v_j) \;=\; \lceil log_2\; muxes(o_i).inputsMUX(v_j) \rceil \tag{7.77}$$

Regras MV *Identificação das variáveis com multiplexador na entrada*

Regra MV.1

Não é obrigatório uma variável, escrita por um estado programa em mais de um local, exigir um multiplexador na sua entrada de modo a suportar as múltiplas operações de escrita.

Regra MV.2

Exige-se um multiplexador com ni entradas na entrada duma variável, escrita por um estado programa, se no grafo desse estado existirem ni ($ni \geq 2$) caminhos distintos, entre a mesma entrada e a mesma saída do grafo, que na sua saída resultem em valores/expressões distintos para essa variável e em que os valores/expressões são atribuídos ao longo dos caminhos.

Exemplos sobre as Regras MV

◇ Em algumas situações, todos os ni caminhos distintos terminam com valores/expressões distintos atribuídos à variável. Esta situação ocorre nos casos (i), (ii) e (iv) da figura 7.16, onde o número de entradas do multiplexador ($ni = 2$) coincide com o número de caminhos distintos.

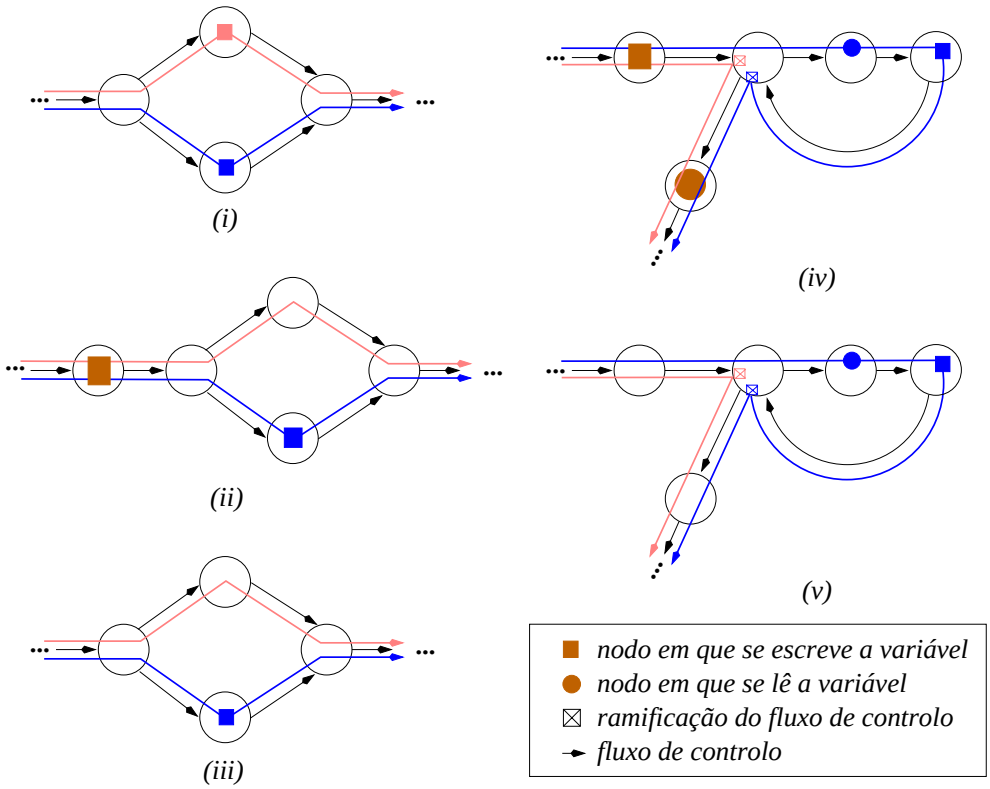


Figura 7.16: Exemplos sobre as regras MV, ilustrando os caminhos distintos entre a entrada e a saída do grafo dum estado programa e as atribuições a uma variável. Nos exemplos (i) a (iii) o grafo representa um construtor condicional e nos exemplos (iv) e (v) um ciclo.

◇ Noutras situações, dos np caminhos distintos apenas nd ($nd < np$) terminam com valores/expressões distintos atribuídos à variável durante os caminhos. Deste modo, o número de entradas do multiplexador é nd. Os restantes $nd - np$ caminhos ou não alteram a variável ou atribuem-lhe o mesmo valor/expressão que outro caminho já contabilizado. Os casos (iii) e (v) da figura 7.16 são exemplos desta situação, onde dos $np = 2$ caminhos apenas $nd = 1$ contribui para as entradas do multiplexador.

Tarefa T10

Para estimar o espaço ocupado em *hardware* pelos elementos de interligação considera-se, como já foi indicado na tarefa T3, que não há partilha de unidades funcionais por parte das operações. Deste modo, não são necessários multiplexadores na entrada das unidades funcionais. Com base na lista de multiplexadores obtida na tarefa T9 ($muxes(o_i)$), o espaço ocupado pelos multiplexadores a colocar na entrada de variáveis escritas em o_i (***areaMUXES(o_i)***) é definido por

$$areaMUXES(o_i) \; =$$

$$= \sum_{v_j \in muxes(o_i)} area(MUX[muxes(o_i).inputsMUX(v_j), 1, larguraVar(v_j)]) \qquad (7.78)$$

em que $area(MUX[muxes(o_i).inputsMUX(v_j), 1, larguraVar(v_j)])$ designa o espaço ocupado por um multiplexador com $muxes(o_i).inputsMUX(v_j)$ entradas, uma saída e em que as entradas possuem $larguraVar(v_j)$ bits. O valor do espaço ocupado por um multiplexador genérico $MUX[ni, 1, nb]$ é definido na equação 7.88 ou 7.89.

Tarefa T11

Conhecido o espaço ocupado em *hardware* pelas unidades funcionais de o_i ($areaUF(o_i)$), o espaço para os elementos de interligação de o_i ($areaMUXES(o_i)$) e excluindo o espaço para elementos de armazenamento, o espaço do estado programa o_i em *hardware* é aproximado por

$$areaHW(o_i) = areaUF(o_i) + areaMUXES(o_i) \qquad (7.79)$$

Tarefa T12

O espaço ocupado em *hardware* por uma variável v_j (***areaHWvar(v_j)***), dado o seu número de elementos ($numElementosVar(v_j)$) e o número de bits por elemento ($larguraVar(v_j)$), é calculado com a equação 7.80 no caso duma variável simples e com a equação 7.81 no caso duma variável composta. O valor de $areaRAM(ne, nb)$ encontra-se definido na equação 7.92.

$$areaHWvar(v_j) = larguraVar(v_j) * area(FF\ D) \qquad (7.80)$$

$$areaHWvar(v_j) = areaRAM(numElementosVar(v_j), larguraVar(v_j)) \qquad (7.81)$$

Tarefa T13

Com a informação sobre quais as variáveis lidas ($readVAR(o_i)$) e escritas ($writeVAR(o_i)$) por cada estado programa o_i, é possível obter o conjunto dos estados programa que lêem (***OBJread(v_j)***) e escrevem (***OBJwrite(v_j)***) cada variável v_j do sistema.

Enquanto os tempos de comunicação[19] e o tempo de execução final ($Texec(o_i)$) associados aos estados programa são actualizados em cada iteração do algoritmo de partição, as outras métricas[20] dos estados programa e variáveis permanecem inalteradas durante todo o processo de partição.

7.6 Métricas de *Hardware* de Baixo Nível

Na estimação ao nível do estado programa e do sistema recorre-se a métricas de *hardware* de baixo nível, fortemente dependentes das características dos componentes disponíveis na arquitectura alvo. Exemplos destas métricas são o espaço ocupado por multiplexadores, operadores aritméticos ou lógicos e elementos de memória. Nesta secção apresentam-se estimativas para estas métricas, válidas para uma implementação com as FPGAs da plataforma EDgAR-2. A função de custo do processo de partição também exige que se conheça a quantidade de recursos que é permitido usar na implementação dos sistemas.

Começa por definir-se os limites que determinam os recursos a utilizar em cada FPGA do EDgAR-2. O limite de portas lógicas equivalentes duma FPGA, que é permitido usar como lógica ou memória, é dado por

$$limiteLogica = \alpha_1 * CLBs * portasPorCLB \tag{7.82}$$

$$limiteRAM = \alpha_1 * CLBs * celulasRAMporCLB * equivalencia1Bit \tag{7.83}$$

e o limite, por FPGA, que se aplica ao número de CLBs[21], LUTs de 3 entradas, LUTs de 4 entradas e *flip-flops*, é dado dado por

$$limiteCLBs = \alpha_2 * CLBs \tag{7.84}$$

$$limiteLUTs3 = \alpha_2 * CLBs * LUTs3PorCLB \tag{7.85}$$

$$limiteLUTs4 = \alpha_2 * CLBs * LUTs4PorCLB \tag{7.86}$$

$$limiteFFs = \alpha_2 * (CLBs * FFsPorCLB + FFsLivresIOBs) \tag{7.87}$$

em que

- $\boldsymbol{\alpha_1}$ é a percentagem de lógica utilizável na FPGA (0.8);

[19] $TcomLocal(o_i)$, $TcomViaSW(o_i)$, $TcomDir(o_i)$, $TcomEsq(o_i)$ e $TexecRSI(o_i)$.
[20] $readVAR(o_i)$, $writeVAR(o_i)$, $muxes(o_i)$, $areaMUXES(o_i)$, $areaUF(o_i)$, $TexecHW(o_i)$, $TexecSW(o_i)$, $FN(o_i)$, $areaHWvar(v_j)$, $OBJread(v_j)$ e $OBJwrite(v_j)$.
[21] Célula base das FPGAs.

- ***CLBs*** é o número de CLBs numa FPGA (576);
- ***portasPorCLB*** é o número típico de portas lógicas equivalentes por CLB (28.5) [Xil97];
- ***celulasRAMporCLB*** é o número de células de RAM disponíveis num CLB (32);
- ***equivalencia1Bit*** é o número de portas lógicas equivalentes a 1 bit de RAM (4);
- $\boldsymbol{\alpha_2}$ é a percentagem de CLBs/LUTs/FFs utilizável na FPGA (0.9);
- ***LUTs3PorCLB*** e ***LUTs4PorCLB*** são o número de LUTs de 3 e de 4 entradas por CLB (1 e 2);
- ***FFsPorCLB*** é o número de *flip-flops* ou *latches* por CLB (2);
- ***FFsLivresIOBs*** é o número de *flip-flops* disponíveis nos blocos de entrada/saída - IOBs[22] (96).

Optou-se por um valor de α_2 (90%) superior ao de α_1 (80%) porque as métricas expressas nas equações 7.84 a 7.87 estão sobre-estimadas em relação às métricas definidas pelas equações 7.82 e 7.83. A quantificação dos limites anteriores é apresentada na tabela 7.5.

Limite de portas lógicas equivalentes (aplicável a lógica)	576 * 28.5 * 0.8	= 13.100
Limite de portas lógicas equivalentes (aplicável a memória)	576 * 32 * 4 * 0.8	= 58.980
Limite de CLBs	576 * 0.9	= 518
Limite de LUTs de 4 entradas	576 * 2 * 0.9	= 1.036
Limite de LUTs de 3 entradas	576 * 0.9	= 518
Limite de *flip-flops* ou *Latches*	(576 * 2 + 96) * 0.9	= 1.123

Tabela 7.5: Recursos, por FPGA do EDgAR-2, que é permitido usar na implementação dos sistemas.

O espaço ocupado por componentes de lógica elementares, exigido pelo cálculo de diversas métricas que foram surgindo ao longo deste capítulo, está resumido na tabela 7.6.

Componente	***Portas Lógicas Equivalentes***
NAND2	1
AND2	2
OR2	3
XOR2	5
MUX2:1	4
FF D	6
FF D com *set* (*reset*)	8
FF D com *set* (*reset*) e *enable*	12
Célula de memória RAM (ROM)	4

Tabela 7.6: Espaço ocupado por alguns componentes de lógica elementares.

Para estimar o espaço ocupado por um **multiplexador** pode optar-se por sinais de selecção codificados ou descodificados. Segundo a opção habitual, os multiplexadores possuem sinais

[22] Célula de entrada/saída nas FPGAs.

de selecção codificados, ou seja, o valor binário definido pelos sinais de selecção indica qual a entrada que é colocada na saída do multiplexador. Quando se faz a síntese de sistemas com um caminho de dados e uma unidade de controlo, pode ser vantajoso optar por multiplexadores com sinais de selecção descodificados, em que o número de sinais de selecção é igual ao número de entradas. Um multiplexador com sinais de selecção codificados exige menos sinais, mas ocupa mais espaço e a geração destes sinais é mais complexa.

O espaço ocupado por um multiplexador com ni entradas ($ni > 1$), uma saída, entradas com nb bits e sinais de selecção codificados é definido pela equação 7.88; enquanto o espaço para o mesmo tipo multiplexador, mas com sinais de selecção descodificados, é calculado com a equação 7.89.

$$\begin{aligned} area(MUX[ni,1,nb]) &= (\lceil log_2\ ni \rceil\ +\ 3 * nb * (ni-1)) * area(NAND2)\ + \\ &+\ ni * (\lceil log_2\ ni \rceil - 1) * area(AND2) \end{aligned} \tag{7.88}$$

$$area(MUX[ni,1,nb])\ =\ 3 * nb * (ni-1) * area(NAND2) \tag{7.89}$$

O desempenho conseguido e os recurso necessários à implementação de **operações aritméticas e lógicas** em FPGA, encontram-se na tabela 7.7. Os valores destas métricas foram obtidos com um processo de síntese. Entre as operações consideradas estão a soma, a subtracção, a multiplicação, a comparação e o deslocamento com operandos inteiros.

Além de operações aritméticas e lógicas, as FPGAs também implementam elementos de armazenamento de informação, nomeadamente **memória RAM** ou **ROM**. Convém por isso possuir estimativas para este tipo de elementos. Para FPGAs da família 4000X/XL da Xilinx, uma memória com tamanho $size$ ($size = 2^n$ e $size > 32$) e palavras de $nbits$ bits, o espaço ocupado em número de CLBs é dado pela equação 7.90. Para memórias com um tamanho de 16 ou 32 palavras, o espaço ocupado em número de CLBs é definido na equação 7.91.

$$\begin{aligned} areaRAMclbs(size,nbits)\ &=\ nbits\ *\ area(RAM32X1)\ *\ \frac{size}{32}\ + \\ &+\ nbits\ *\ \sum_{i=0}^{log_2(size/64)} 2^i\ *\ area(MUX2:1)\ + \\ &+ \begin{cases} 1\ CLB & , \text{ se } size = 64 \\ 2\ CLBs & , \text{ se } size = 128 \\ 4\ CLBs & , \text{ se } size = 256 \\ 16\ CLBs & , \text{ se } size = 512 \\ 32\ CLBs & , \text{ se } size = 1024 \end{cases} \end{aligned} \tag{7.90}$$

Tipo de Operação	***Texec$_{hw}$*** *(ciclos)*	***Maior Atraso$_{hw}$*** *(ns)*	***Recursos de HW***			***Portas Lógicas Equiv.***
			CLBs	***LUTs 4 , 3 entradas***	***FFs ou Latches***	
Deslocar em N posições um registo de 8 bits	N+$\lceil \frac{11}{TclkHw} \rceil$	11+(N+1)* T*clkHw*	11	13 , 9	12(4) (1)	190
Incrementar um registo de 8 bits	$\lceil \frac{11}{TclkHw} \rceil$	11+T*clkHw*	5	10 , 1	8	160
Comparar valores de 8 bits	$\lceil \frac{18}{TclkHw} \rceil$	18+T*clkHw*	5	5 , 0	1	36
Comparar valores de 16 bits	$\lceil \frac{18}{TclkHw} \rceil$	18+T*clkHw*	9	10 , 1	1	70
Subtrair valores de 8 bits	$\lceil \frac{21}{TclkHw} \rceil$	21+T*clkHw*	11	19 , 3	9	180
Multiplicar 6 bits por 2 bits	$\lceil \frac{26}{TclkHw} \rceil$	26+T*clkHw*	19	32 , 5	8	260
Multiplicar 8 bits por 8 bits	$\lceil \frac{60}{TclkHw} \rceil$	60+T*clkHw*	81	161 , 18	16	1,140
Somar 2 bits sem sinal, registando o resultado	$\lceil \frac{8}{TclkHw} \rceil$	8+T*clkHw*	3	5 , 2	3	57
Somar 4 bits sem sinal, registando o resultado	$\lceil \frac{8}{TclkHw} \rceil$	8+T*clkHw*	5	9 , 1	5	90
Somar 8 bits sem sinal, registando o resultado	$\lceil \frac{18}{TclkHw} \rceil$	18+T*clkHw*	12	20 , 2	9	180
Somar 16 bits sem sinal, registando o resultado	$\lceil \frac{60}{TclkHw} \rceil$	60+T*clkHw*	24	40 , 6	17	370
Copiar 8 bits de memória para um registo	2 (2)	23+T*clkHw* (2)	0	0 , 0	0	0
Copiar 16 bits de memória para um registo	2 (2)	20+T*clkHw* (2)	0	0 , 0	0	0
Copiar um registo de 8 bits para memória	2 (2)	23+T*clkHw* (2)	0	0 , 0	0	0
Copiar um registo de 16 bits para memória	2 (2)	20+T*clkHw* (2)	0	0 , 0	0	0
Ler um registo	0	–	0	0 , 0	0	0
Escrever num registo	1	–	0	0 , 0	0	0
Ler memória	1	–	0	0 , 0	0	0
Escrever memória	2	–	0	0 , 0	0	0
RAM 64 x 16 bits	–	–	41	82 , 0	0	4,204 (3)
RAM 64 x 8 bits	–	–	21	42 , 0	0	2,108 (4)

(1) Considerando (excluindo) o registo.
(2) T*clkHw* deve ser $>$ 18 ns, ou seja, a frequência de relógio usada no *hardware* deve ser $<$ 55 MHz.
(3) 4,204 = 4,096 (para memória) + 108 (portas lógicas).
(4) 2,108 = 2,048 (para memória) + 60 (portas lógicas).

Tabela 7.7: Métricas relativas à implementação de operações elementares em FPGAs.

$$areaRAMclbs(size, nbits) \;=\; nbits \;*\; area(RAM32X1) \;*\; \frac{size}{32} \tag{7.91}$$

em que $area(RAM32X1) = 1\ CLB$ e $area(MUX2:1) = \frac{1}{2}\ CLB$.

O espaço duma memória, em portas lógicas equivalentes, obtém-se multiplicando o espaço em número de CLBs pelo número típico de portas lógicas equivalentes por CLB (equação 7.92).

$$areaRAM(size, nbits) \;=\; areaRAMclbs(size, nbits) * portasPorCLB \tag{7.92}$$

com $portasPorCLB = 28.5$.

Aplicando as equações 7.90 e 7.91 construiu-se a tabela 7.8, onde se quantifica o espaço ocupado por algumas RAMs passíveis de ser implementadas numa FPGA com uma dimensão da ordem da dezena de milhar de portas lógicas.

Tipo de RAM	***Número de CLBs***	
RAM Nx8 ($N \leq 16$)	–	4
RAM 32x8	–	8
RAM 64x8	8*2+1*8*1/2+1	= 21
RAM 128x8	8*4+3*8*1/2+2	= 46
RAM 256x8	8*8+7*8*1/2+4	= 96
RAM 512x8	8*16+15*8*1/2+16	= 204
RAM 1024x8	8*32+31*8*1/2+32	= 412
RAM Nx16 ($N \leq 16$)	–	8
RAM 32x16	–	16
RAM 64x16	16*2+1*16*1/2+1	= 41
RAM 128x16	16*4+3*16*1/2+2	= 90
RAM 256x16	16*8+7*16*1/2+4	= 188
RAM 512x16	16*16+15*16*1/2+16	= 392
RAM 1024x16	16*32+31*16*1/2+32	= 792

Tabela 7.8: Espaço ocupado por módulos de RAM numa FPGA do EDgAR-2.

7.7 Resumo e Conclusões

Com a metodologia de estimação apresentada pretende obter-se estimativas precisas e manter-se o tempo de cálculo o mais baixo possível. Para atingir estes objectivos, a estimação funciona em dois níveis de abstracção e utiliza actualização incremental das estimativas. As estimativas obtidas no nível de abstracção do sistema são recalculadas ou actualizadas em todas as iterações do processo de partição, enquanto no nível do estado programa as métricas só são calculadas uma vez. Ao decompor a estimação do tempo de execução dum sistema em dois níveis de abstracção, introduz-se imprecisão nas estimativas porque se trata a nível local o problema global de escalonamento.

Para estimar o desempenho da parte de *software* dos sistemas definiu-se um modelo de *software* simples, específico duma família de processadores e que recorre a um factor, obtido por simulação, para quantificar as optimizações permitidas pelo processador. A maior limitação deste modelo está em exigir uma representação executável para os sistemas e os pontos fortes são a simplicidade de implementação e o tempo de cálculo reduzido. Definiu-se também um modelo melhorado que considera as optimizações sob a forma dum número médio de ciclos por instrução, com a vantagem de este factor poder ser obtido estaticamente.

De acordo com o modelo de *hardware*, o espaço ocupado pelo caminho de dados duma partição inclui as unidades funcionais e os elementos de interligação necessários aos estados programa, os elementos de armazenamento que guardam as variáveis, os elementos de interligação adicionais e os recursos da interface com o exterior da partição. As variáveis locais introduzidas aumentam o espaço ocupado em *hardware*, a fiabilidade e a complexidade das implementações, uma vez que é preciso garantir a coerência entre as várias cópias da mesma variável. O principal contributo para o espaço ocupado pela unidade de controlo duma partição de *hardware* é a máquina de estados, que considera o registo de estado, a lógica de controlo e a lógica do próximo estado. Para estimar o espaço ocupado pela máquina de estados é necessário dispor do escalonamento dos estados programa, da secção de máquina de estados para as tarefas escalonadas em cada etapa de controlo, da informação sobre os sinais de selecção dos multiplexadores e sobre as mudanças de partição.

De entre todas as métricas estimadas, o tempo de comunicação entre partições é aquela que lida mais fortemente com as características da arquitectura alvo das implementações. A comunicação entre *hardware* e *software* pode ser efectuada através dum mecanismo de acesso directo a registo ou memória e por auscultação ou por interrupção, mas o mecanismo de acesso directo à memória não foi implementado. Devido à ausência de funcionalidade *master* no controlador PCI da plataforma EDgAR-2, algumas operações iniciadas em *hardware* são efectuadas sob controlo do *software* e podem decorrer em duas etapas, o que obriga a alterar o código do estado programa que inicia a operação e a criar ou alterar um estado de *software* responsável pela execução da operação. Para reduzir o tempo de cálculo, em cada iteração apenas se corrige o tempo de comunicação dos estados deslocados de partição ou afectados pelas variáveis deslocadas de partição.

O desempenho dum sistema depende do tempo de computação dos estados programa, da atribuição dos objectos, do tempo de comunicação entre estados e variáveis, da frequência de execução dos estados, do sincronismo entre estados e do escalonamento do sistema. Ao não usar escalonamento explícito ao nível do sistema, obtêm-se estimativas pessimistas para o tempo de execução mas reduz-se o tempo de cálculo.

A implementação da estimação do desempenho dum sistema recorre ao conhecimento sobre

o tempo de execução de construtores condicionais, ciclos, construtores paralelos e ciclos de espera. Ao calcular o tempo de execução dum sistema, por análise do seu grafo PSMfg, o tempo de execução é calculado apenas nos nodos em que é preciso guardar o valor acumulado desde o início do grafo.

O tempo de execução após um ciclo de espera é afectado pelo tipo de expressão que define o ciclo, pelas variáveis incluídas na expressão, pelo valor esperado em cada variável, pelos estados que escrevem o valor esperado nas variáveis e pelo instante em que os valores são escritos. Para ajudar a obter esta informação definiram-se as variantes de ciclo de espera a tratar automaticamente e um conjunto de regras que identifica os estados intervenientes em cada ciclo. A estimativa do tempo de execução dum ciclo de espera considera duas contribuições: uma resulta dos estados que escrevem valores determinísticos nas variáveis inspeccionadas e a outra resulta dos estados que escrevem valores não determinísticos nessas variáveis.

Para estimar o tempo de execução dum sistema calcula-se o tempo de execução no fluxo principal do grafo desse sistema, tarefa que para ser concretizada exige a informação relativa a pontos de paragem, pontos de sincronismo e fluxos do grafo. Para obter o tempo de execução dum fluxo, calcula-se o tempo de execução em cada um dos sub-fluxos que o compõem. Por sua vez, o tempo de execução dum sub-fluxo obtém-se a partir do tempo de execução dos nodos e do tempo de execução associado às mudanças de partição.

As estimativas obtidas ao nível do estado programa são de nível inferior às que se obtêm ao nível do sistema, são utilizadas no cálculo das segundas, aplicam-se com métricas relativas aos objectos do sistema, são calculadas apenas uma vez e por isso pretende-se que atinjam um grau de precisão elevado.

O escalonamento das operações dos estados programa, para uma implementação de *software*, só é relevante quando se usa o modelo de *software* melhorado e/ou uma arquitectura alvo multi-processador. O escalonamento de operações para uma implementação de *hardware* aplicou a estratégia ASAP, considerou que não existe partilha de unidades funcionais e que os recursos são ilimitados. Este escalonamento é aplicado no cálculo do tempo de computação e do espaço ocupado pelas unidades funcionais do estado programa.

A estimação ao nível do estado programa e do sistema envolve métricas de *hardware* de baixo nível, como o tempo de execução de operadores aritméticos/lógicos e o espaço ocupado por multiplexadores, operadores aritméticos/lógicos e elementos de memória.

Em conclusão, dada a vastidão de aspectos arquitecturais que era preciso modelar para obter estimativas precisas para as métricas envolvidas na função de custo do processo de partição iterativo, alguns desses aspectos foram tratados de forma simplificada, quer porque a sua importância assim o aconselhava quer porque a exequibilidade deste trabalho o justificava.

Capítulo 8

Validação e Avaliação da Metodologia de Partição

Sumário

Com o objectivo de validar a metodologia de partição proposta, descrevem-se dois casos de estudo: a convolução duma imagem e o algoritmo de criptografia DES. Para cada exemplo estudado apresenta-se uma breve descrição, a melhor solução de partição obtida automaticamente pelo algoritmo de pesquisa tabu e uma implementação hardware/software optimizada que avalia a qualidade dessa solução. A melhor solução de partição é caracterizada pelo modelo PSMfg do sistema, a atribuição dos objectos às partições, as métricas obtidas ao nível do estado programa e do sistema e os parâmetros utilizados no processo de partição. A implementação hardware/software é descrita através do modelo PSM do sistema, a arquitectura das partições de hardware presentes na implementação (caminho de dados e unidade de controlo), as métricas obtidas ao nível do estado programa e do sistema, a precisão das estimativas de métricas e o resultado da inspecção do funcionamento do sistema.

Conteúdo

8.1 Introdução

Para validar a metodologia de partição proposta nos capítulos 6 e 7, utilizou-se como primeiro critério de avaliação a qualidade das soluções de partição obtidas automaticamente. Dado que a qualidade das soluções depende da qualidade das estimativas, seleccionou-se para segundo critério de avaliação a precisão das estimativas. A metodologia pode ainda ser avaliada pelo seu desempenho, ou seja, pelo tempo de cálculo necessário para gerar as soluções de partição e

pelo apoio que presta à implementação destas soluções, nomeadamente à síntese da interface entre partições.

A validação da metodologia foi concretizada através dum estudo detalhado de dois exemplos. Os exemplos seleccionados são sistemas embebidos predominantemente de dados, que em conjunto cobrem as características mais importantes deste tipo de sistema. Não se testou a metodologia com sistemas embebidos predominantemente de controlo porque (i) o paradigma de co-projecto de *hardware* e de *software*, no qual o problema de partição se insere, tem como alvo preferencial os sistemas predominantemente de dados, (ii) dadas as características da arquitectura alvo, estes sistemas colocariam menos problemas à partição/implementação do que os sistemas predominantemente de dados e (iii) considerando que os exemplos disponíveis apresentavam grande interacção com o exterior, levantavam sérios problemas à validação do seu funcionamento. Os maiores problemas levantados pelos sistemas de controlo surgem na sua concepção e não na implementação.

Os exemplos seleccionados foram a aplicação dum filtro a uma imagem (convolução) e o algoritmo de criptografia DES[1]. Os dois casos de estudo completam-se um ao outro, dado que um está aparentemente mais vocacionado para uma implementação em *software* e o outro para uma implementação em *hardware*. Ambos os sistemas possuem uma dimensão da ordem de grandeza dos recursos de *hardware* disponíveis na arquitectura alvo. Para avaliar o comportamento da metodologia de partição, não seria prudente estudar sistemas com dimensão muito superior ao que é permitido pela plataforma EDgAR-2, uma vez que desse modo a relação entre as partes atribuídas a *software* e a *hardware* seria completamente desequilibrada. Como a componente de *software* seria preponderante em qualquer solução de partição, seria difícil observar a influência das decisões tomadas pelo processo de partição sobre a qualidade das soluções por ele obtidas.

8.2 Caso de Estudo 1: Convolução duma Imagem

8.2.1 Descrição do Caso de Estudo

O primeiro exemplo seleccionado para validar a metodologia de partição proposta foi a convolução duma imagem com um filtro [GW87]. Neste exemplo foram usadas imagens com dimensão até 256 por 256 pontos (*pixels*) e um filtro Sobel, um filtro do tipo gradiente, com dimensão de 3 por 3 pontos.

O processo de aplicação dum filtro *mask*, de tamanho *maskX* por *maskY* pontos, a uma imagem I, é ilustrado na figura 8.1. Este processo é efectuado em duas etapas:

[1] ***D**ata **E**ncryption **S**tandard*, na terminologia inglesa.

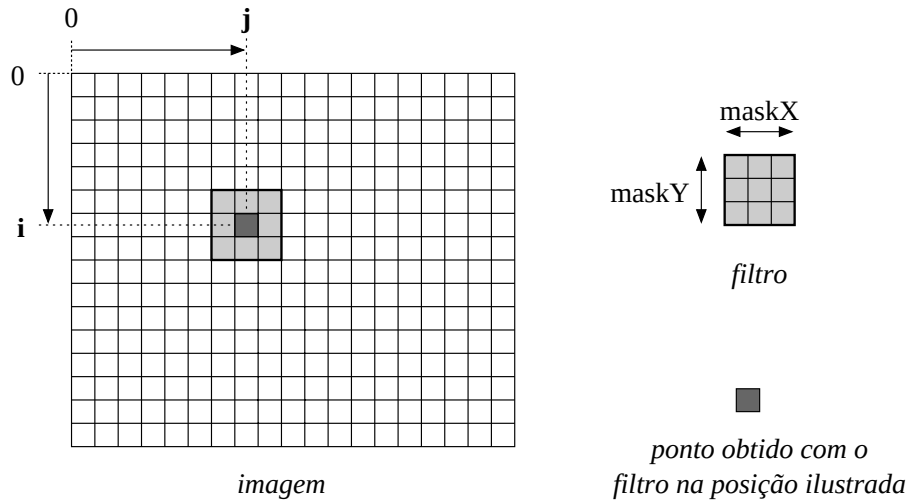


Figura 8.1: Ilustração do processo de aplicação dum filtro de tamanho $maskX$ por $maskY$ a uma imagem, para situação em que se produz o ponto (j, i) da imagem filtrada.

1. Para cada ponto (j, i), com cor $I(j, i)$ na imagem original I, convolve-se uma zona de tamanho igual ao do filtro e centrada no ponto (j, i) com o filtro $mask$, obtendo-se um novo valor $If(j, i)$ para o ponto (j, i); o novo valor $If(j, i)$ é obtido com a equação 8.1;

$$If(j,i) \;=\; \sum_{k=0}^{maskY-1} \; \sum_{l=0}^{maskX-1} I(j - \lfloor \frac{maskX}{2} \rfloor + l \;,\; i - \lfloor \frac{maskY}{2} \rfloor + k) * mask(l,k) \qquad (8.1)$$

   em que as dimensões do filtro, $maskX$ e $maskY$, possuem um número ímpar de pontos;

2. Com o conhecimento do menor e do maior valor na imagem filtrada If, $min(If)$ e $max(If)$ respectivamente, normaliza-se a imagem filtrada para a gama de cores da imagem original ($gama(I)$), resultando daqui a imagem filtrada e normalizada In (equação 8.2).

$$In(j,i) \;=\; \frac{gama(I)}{max(If) - min(If)} * [\; If(j,i) - min(If) \;] \qquad (8.2)$$

Considerando que a imagem e o filtro estão armazenados em memória, uma estrutura unidimensional, a posição de acesso à imagem ou ao filtro é definida à custa dum único valor. Deste modo, a implementação da equação 8.1 com dois ciclos exige que se efectuem os cálculos indicados no conjunto de expressões 8.3.

$$\begin{cases} posI \;=\; (j \;-\; \lfloor \frac{maskX}{2} \rfloor \;+\; l) \;+\; (i \;-\; \lfloor \frac{maskY}{2} \rfloor \;+\; k) \;*\; imgX \\ posM \;=\; l \;+\; k \;*\; maskX \\ posIf \;=\; j \;+\; i \;*\; imgX \\ If(posIf) \;=\; If(posIf) \;+\; I(posI) \;*\; mask(posM) \end{cases} \qquad (8.3)$$

onde $posIf$ é a posição onde se guarda o ponto $If(j,i)$ que está a ser calculado, $posI$ ($posM$) é a posição da imagem (filtro) utilizada numa iteração do cálculo de $If(j,i)$ e $imgX$ é a largura da imagem.

De acordo com a equação 8.1, uma operação presente nas expressões 8.3 e que utiliza l (k) é efectuada $maskY * maskX$ ($maskY$) vezes. Quando se aplica um filtro de dimensão 3x3, para obter um ponto $If(j,i)$ da imagem convolvida não normalizada é necessário efectuar 3*3*4+4 somas, 3*3+7 multiplicações e 2 subtracções. Estas operações contribuem com a maior parcela para o tempo total de execução da convolução. Este facto é confirmado pelo valor estimado para o tempo de execução do estado programa $E_{calc1pt-contrib}$, que consta na figura 8.5. Para melhorar o desempenho da convolução, deve por isso procurar optimizar-se a implementação das expressões 8.3.

O mais forte condicionalismo imposto pela arquitectura alvo à implementação da convolução é não dispor, na plataforma EDgAR-2, de memória RAM em quantidade que permita guardar a totalidade da imagem a convolver e da imagem resultante da convolução.

8.2.2 Solução de Partição Resultante da Descrição Original da Convolução

Para efectuar a partição do sistema, parte-se do modelo PSM do algoritmo de convolução, descrito na figura 8.2, conjuntamente com o código VHDL dos estados programa deste modelo, apresentado na figura 8.3. A representação correspondente, através do meta-modelo PSMfg, encontra-se na figura 8.4.

Para cada estado programa da descrição do sistema, a tabela 8.1 contém as estimativas de métricas obtidas ao nível do estado programa: (i) espaço ocupado pelas unidades funcionais necessárias ao estado programa ($areaUF$), (ii) tempo de computação do estado programa em *software* ($TexecSW$) e (iii) tempo de computação do estado programa em *hardware* ($TexecHW$). As métricas de *hardware* para o estado programa $E_{norm-img}$ não são apresentadas na tabela 8.1, uma vez que a implementação deste estado em *hardware* exige uma multiplicação real e um divisor dum inteiro por um valor não potência de 2, operações de difícil implementação numa FPGA como as que estão disponíveis na arquitectura alvo. Deste modo, o estado programa $E_{norm-img}$ tem como atribuição obrigatória uma partição de *software*.

A estimativa para o espaço $areaHWvar$ ocupado em *hardware* pelas variáveis presentes na descrição da convolução encontra-se na tabela 8.2. Ressalta desta tabela que o espaço necessário às variáveis img_in e img_out excede largamente o espaço disponível nas FPGAs[2], logo deverão ser atribuídas a *software* durante o processo de partição.

A tabela 8.3 apresenta estimativas, para cada estado programa da descrição da convolução,

[2]O espaço utilizável em cada FPGA é definido na tabela 7.5.

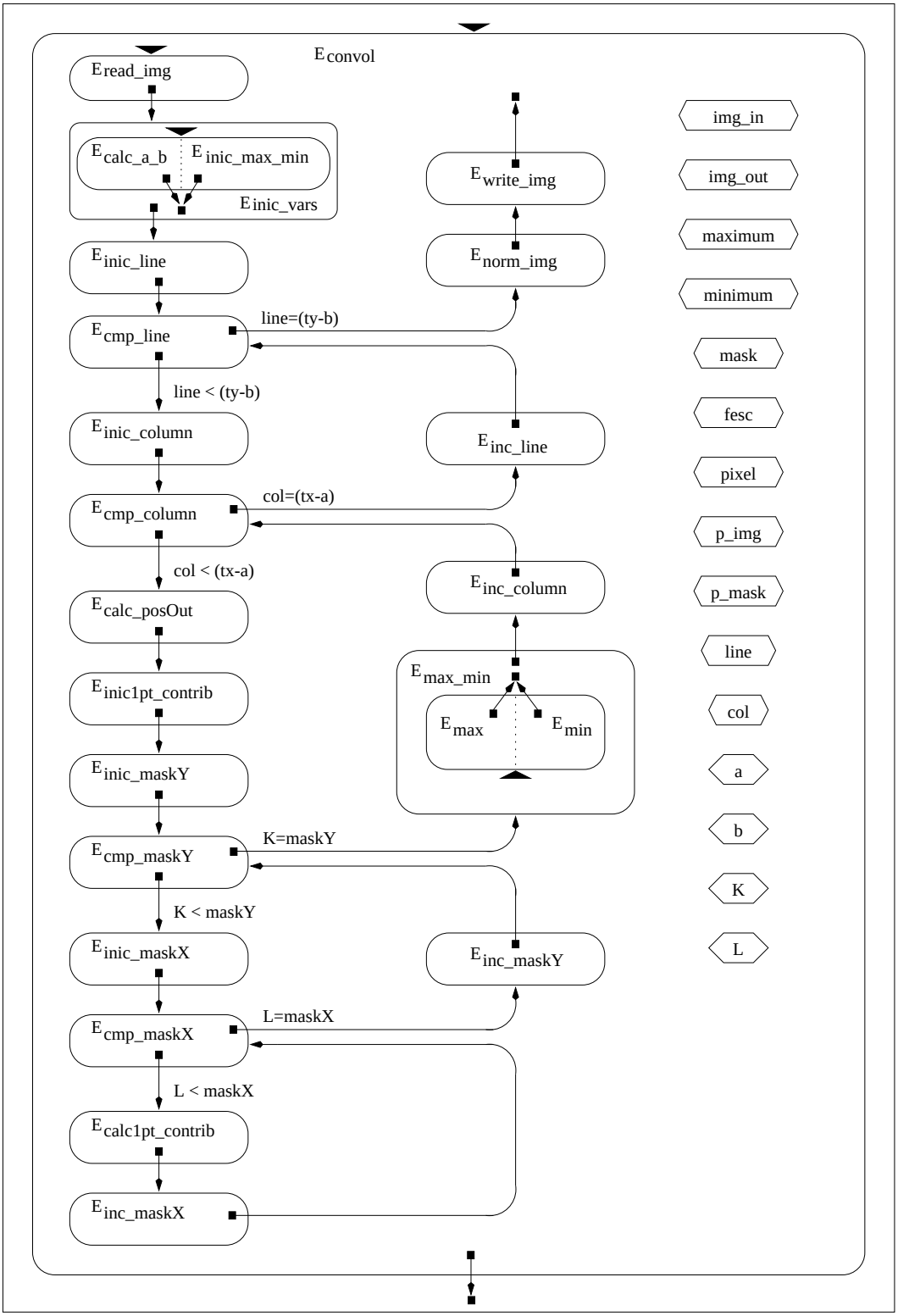


Figura 8.2: Descrição inicial da convolução com o meta-modelo PSM.

```
TYPE int_vector is array (NATURAL range <>) of integer;
variable minimum, maximum: integer;
variable p_img,p_mask,a,b,pixel: integer;
variable fesc: real;
variable line, col, K, L: integer;
signal img_out: int_vector(tx*ty downto 0);
signal img_in: int_vector(tx*ty downto 0);
variable mask: int_vector(maskX*maskY downto 0);
constant tx, ty, maskX, maskY: integer := 256, 256, 3, 3;
```

Ecalc_a_b:
```
=======
   a := maskX/2;
   b := maskY/2;
```

Einic_line:
```
=======
   line := b;
```

Ecmp_line:
```
=======
   L_LINES: while line<(ty-b) loop
      [...]
   end loop L_LINES;
```

Einic_column:
```
==========
   col := a;
```

Ecmp_column:
```
==========
   L_COLUMNS: while col<(tx-a) loop
      [...]
   end loop L_COLUMNS;
```

Einic1pt_contrib:
```
============
   img_out[pixel] <= 0.0;
```

Ecalc_posOut:
```
==========
   pixel := col + (line * tx);
```

Einic_maskY:
```
==========
   K := 0;
```

Ecmp_maskY:
```
==========
   L_MASK: while K < maskY loop
      [...]
   end loop L_MASK;
```

Einic_maskX:
```
==========
   L := 0;
```

Einic_max_min:
```
============
minimum := 0;
maximum := 1;
```

Ecmp_maskX:
```
==========
   C_MASK: while L < maskX loop
      [...]
   end loop C_MASK;
```

Ecalc1pt_contrib:
```
============
   p_img := (col-a+L)+((line-b+K)*tx);
   p_mask := L+K*maskX;
   img_out(pixel) <= img_out(pixel) +
      img_in(p_img) * mask(p_mask);
```

Einc_maskX:
```
==========
   L := L + 1;
```

Einc_maskY:
```
==========
   K := K + 1;
```

Emax:
```
====
   if(img_out(pixel)>maximum) then
      maximum := img_out(pixel);
   end if;
```

Emin:
```
====
   if(img_out(pixel)<minimum) then
      minimum := img_out(pixel);
   end if;
```

Einc_column:
```
==========
   col := col + 1;
```

Einc_line:
```
========
   line := line + 1;
```

Enorm_img:
```
=========
   maximum := maximum-minimum;
   fesc := 255/maximum;
   LINES: for line in b to (ty-b) loop
     COLS: for col in a to (tx-a) loop
       pixel = col + (line * tx);
       img_out(pixel) <=(img_out(pixel)-
          minimum) * fesc;
     end loop COLS;
   end loop LINES;
```

Figura 8.3: Código para a descrição inicial da convolução (estado programa E_{convol}).

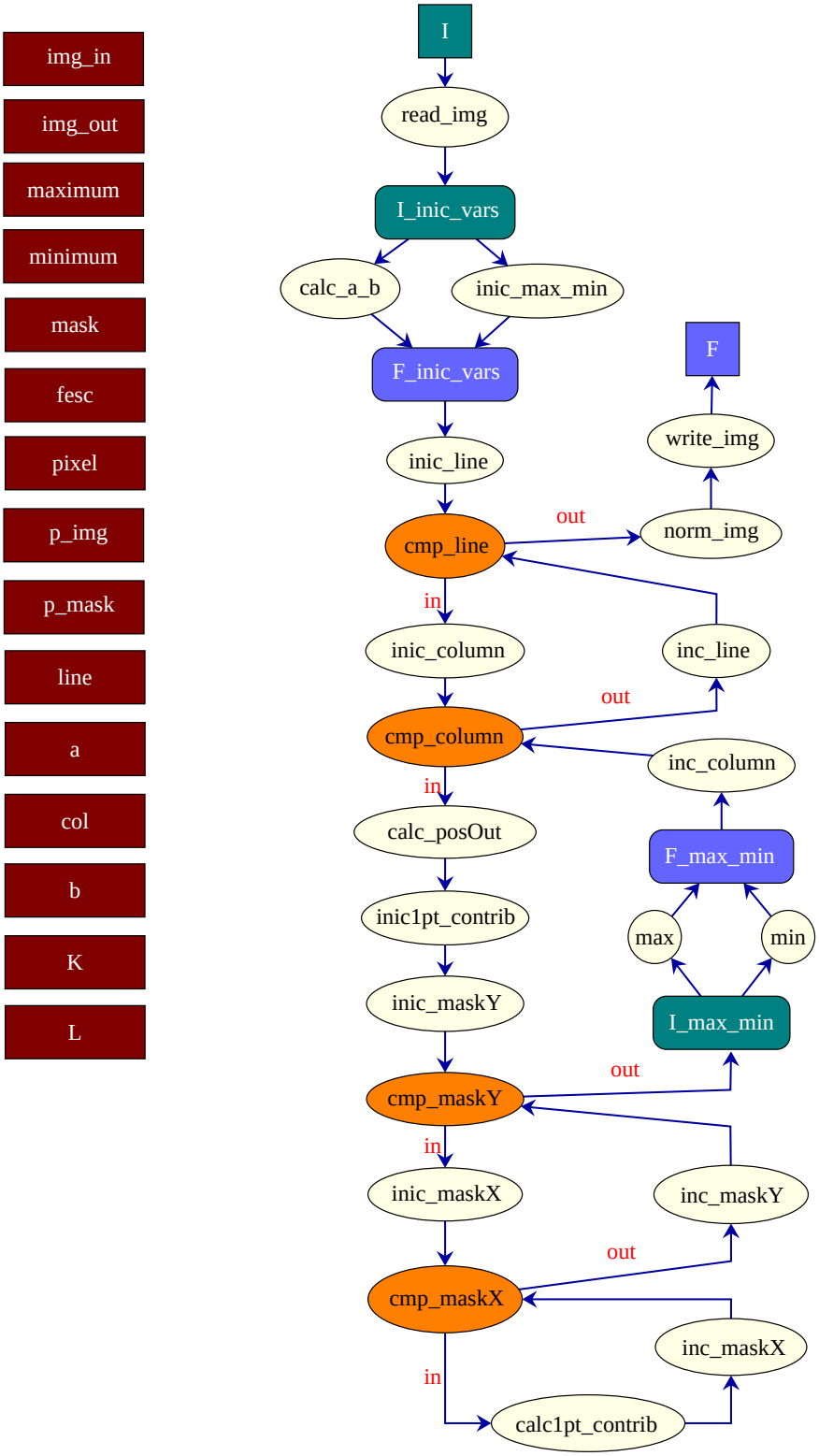


Figura 8.4: Descrição inicial da convolução com o meta-modelo PSMfg.

das variáveis lidas, escritas e das variáveis escritas que necessitam dum multiplexador na sua entrada.

Estado programa	***Tipo de recurso de h/w***	***areaUF*** *(portas lógicas)*	***TexecSW*** *(ciclos do µP)*	***TexecHW*** *(ciclos do h/w)*
Eread_img	–	–	–	–
Ecalc_a_b	2 x deslocação 8-bits	380	8	2
Einic_max_min	–	0	0	0
Einic_line	–	0	0	0
Ecmp_line	comparador 8-bits	36	3	1
Einc_line	contador 8-bits	160	2	1
Einic_column	–	0	0	0
Ecmp_column	comparador 8-bits	36	3	1
Einc_column	contador 8-bits	160	2	1
Ecalc_posOut	somador 16-bits multiplicador 8-bits	1510	15	4
Einic1pt_contrib	–	0	0	0
Einic_maskY	–	0	0	0
Ecmp_maskY	comparador 2-bits	20	3	1
Einc_maskY	contador 2-bits	37	2	1
Einic_maskX	–	0	0	0
Ecmp_maskX	comparador 2-bits	20	3	1
Einc_maskX	contador 2-bits	37	2	1
Ecalc1pt_contrib	2 x somador 16-bits 2 x somador 8-bits somador 4-bits 2 x subtractor 8-bits 2 x multiplicador 8-bits multiplicador 2-bits	3887	57	10
Emax_min	2 x comparador 16-bits	140	6	1
Enorm_img	–	–	58 + 9*254 + 86*254*254	–
Ewrite_img	–	–	–	–

Tabela 8.1: Estimativa das métricas associadas aos estados programa da descrição inicial da convolução.

Supondo uma implementação com um processador a 200 MHz e com componentes de *hardware* a funcionarem com uma frequência de 33 MHz, constata-se que na solução de partição óptima todos os objectos do sistema são atribuídos a *software*. A explicação para esta solução resulta do melhor desempenho que a implementação totalmente em *software* apresenta relativamente a qualquer outra. Como todos os objectos do sistema apresentam um tempo de computação inferior em *software* e o tempo de comunicação é baixo quando se atribuem todos os objectos a *software*, o melhor desempenho ocorre na implementação totalmente em *software*.

O desempenho de solução de *software*, sob a forma de tempo de execução, pode ser obtido a partir do escalonamento esquematizado na figura 8.5. No tempo de execução não se contabiliza nem o tempo de leitura da imagem do disco, estado E_{read_img}, nem o tempo de escrita da imagem em disco, estado E_{write_img}. Este escalonamento, totalmente sequencial, é válido para uma arquitectura alvo uni-processador e para um modelo de estimação simples, que quantifica as optimizações permitidas pelo processador num factor obtido por simulação. A figura 8.5 contém os tempos de execução em *software* para os estados programa do modelo

Variável	***Largura*** *(bits)*	***Número de elementos***	***areaHWvar*** *(portas lógicas)*
img_in	8	256*256	702126
img_out	16	256*256	1402428
maximum	16	1	96
minimum	16	1	96
mask	4	9	57
fesc	32	1	192
pixel	16	1	96
p_img	16	1	96
p_mask	4	1	24
line	8	1	48
col	8	1	48
a	2	1	12
b	2	1	12
K	2	1	12
L	2	1	12

Tabela 8.2: Estimativa das métricas associadas às variáveis da descrição inicial da convolução.

Estado programa	***Variáveis lidas***	***Variáveis escritas***	***Variáveis com multiplexador***
Ecalc_a_b	–	a, b	–
Einic_max_min	–	minimum, maximum	–
Einic_line	b	line	–
Ecmp_line	b, line	–	–
Einc_line	line	line	–
Einic_column	a	col	–
Ecmp_column	a, col	–	–
Einc_column	col	col	–
Ecalc_posOut	col, line	pixel	–
Einic1pt_contrib	pixel	img_out	–
Einic_maskY	–	K	–
Ecmp_maskY	K	–	–
Einc_maskY	K	K	–
Einic_maskX	–	L	–
Ecmp_maskX	L	–	–
Einc_maskX	L	L	–
Ecalc1pt_contrib	col, line, a, b, pixel, mask, img_in, img_out, L, K	p_img, p_mask, img_out	–
Emax	img_out, maximum, pixel	maximum	–
Emin	img_out, minimum, pixel	minimum	–
Enorm_img	maximum, minimum, a, b, line (†), col (§) , img_out (‡)	maximum, fesc, pixel (‡), img_out (‡)	–

(†) 255 vezes (§) 255*254 vezes (‡) 254*254 vezes

Tabela 8.3: Estimativa, para cada estado programa da descrição inicial da convolução, das variáveis lidas, escritas e que precisam dum multiplexador na entrada.

da convolução. Para os dois estados programa com maior tempo de execução, $E_{calc1pt-contrib}$ e E_{norm_img}, é apresentado o seu próprio escalonamento.

A estimativa do tempo de execução da solução totalmente em *software*, obtida manualmente a partir do escalonamento da figura 8.5, é dada por 45.875.576 ciclos de relógio, que para um processador a 200 MHz equivale a 229 ms. O valor obtido automaticamente pela função $calcTexecSystem$ também é 229 ms. Já para a solução totalmente em *hardware*, usando uma frequência de 33 MHz, a estimativa do tempo de execução é aproximada por 452 ms. Para esta tão forte desvantagem, em termos de desempenho, da solução de *hardware* relativamente à de *software* contribuem vários aspectos: (i) a relação entre a frequência dos relógios de *hardware* e de *software* é de 1 para 6, (ii) os tempos de *hardware* estão sobre-estimados porque o tempo de execução das operações elementares é sempre um múltiplo do ciclo do relógio de *hardware*; quanto menor é esta frequência, maior é o grau de sobre-estimação dos tempos de *hardware*, (iii) como a descrição do sistema é praticamente sequencial, não permite explorar o paralelismo natural do *hardware*, (iv) a arquitectura das FPGAs, em conjunto com o facto de as FPGAs e os CPLDs usados na implementação não serem dispositivos de elevado desempenho, traduz-se em atrasos significativos; para compensar os atrasos imprevisíveis que ocorrem nas FPGAs, recorre-se o mais possível a implementações síncronas, o que tende a aumentar ainda mais o tempo de execução.

A estimativa para o desempenho da convolução, obtida com uma implementação totalmente em *software* ou totalmente em *hardware* e usando um relógio de *hardware* a 33 MHz e de *software* a 200 MHz, é apresentada na tabela 8.4.

Tipo de solução de partição	***Estimativa do tempo de execução*** *(ms)*
Solução totalmente em *software*	229
Solução totalmente em *hardware*, excepto E_{norm_img} em *software*	590
Solução totalmente em *hardware*, sem contabilizar E_{norm_img}	424
Solução totalmente em *hardware*, supondo que $TexecHW(E_{norm\_img})$ é igual a $TexecSW(E_{norm\_img})$	452

Tabela 8.4: Estimativa para o desempenho da convolução, obtida com uma implementação totalmente em *software* ou totalmente em *hardware* e usando um relógio de *hardware* a 33 MHz e de *software* a 200 MHz.

Para tentar melhorar o desempenho das soluções com *hardware* e *software*, através da componente de *hardware*, deve alterar-se um ou mais dos quatro aspectos anteriormente citados. Sem recorrer a uma nova arquitectura alvo não é possível melhorar o desempenho intrínseco das FPGAs e dos CPLDs utilizados na implementação. Por outro lado, não se revela adequado usar valores absolutos, em vez de ciclos de relógio, para estimativa do tempo de execução em *hardware* das operações elementares, uma vez que isso equivalia a usar implementações assíncronas que, como se afirmou, apresentam um comportamento imprevisível quando se utilizam

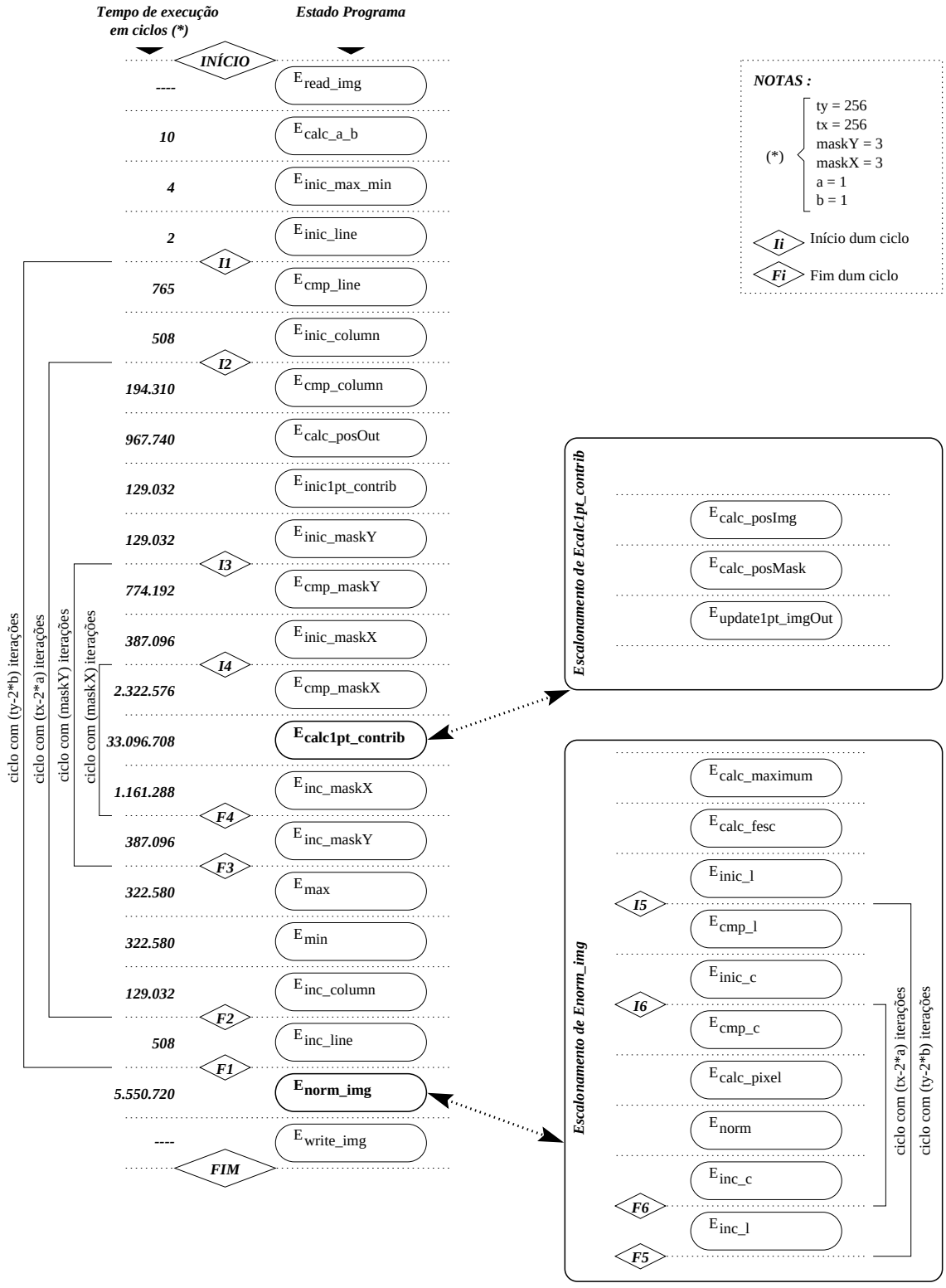


Figura 8.5: Escalonamento da convolução para uma implementação em *software*.

FPGAs como as disponíveis na arquitectura alvo. As alternativas que restam para melhorar o desempenho das soluções, devido ao *hardware*, resumem-se a aumentar a frequência do relógio do *hardware* e a aumentar o grau de paralelismo na descrição do sistema. Ao introduzir mais paralelismo na descrição do sistema permite-se mais paralelismo na implementação.

Quando se aumenta a frequência do relógio do *hardware* de 33 MHz para 50 MHz, a estimativa para o desempenho da implementação da convolução em *hardware* obviamente melhora, como comprovam os valores da tabela 8.5. Contudo, a solução de partição óptima continua a atribuir todos os objectos do sistema a *software*. Considerando que 50 MHz é a frequência máxima que se deseja usar, o desempenho da solução de *software* só será ultrapassado pela solução *hardware*/*software* se for introduzido mais paralelismo na implementação do sistema.

Tipo de solução de partição	***Estimativa do tempo de execução*** *(ms)*
Solução totalmente em *software*	229
Solução totalmente em *hardware*, excepto E_{norm_img} em *software*	451
Solução totalmente em *hardware*, sem contabilizar E_{norm_img}	282
Solução totalmente em *hardware*, supondo que $TexecHW(E_{norm\_img})$ é igual a $TexecSW(E_{norm\_img})$	310

Tabela 8.5: Estimativa para o desempenho da convolução, obtida com uma implementação totalmente em *software* ou totalmente em *hardware* e usando um relógio de *hardware* a 50 MHz e de *software* a 200 MHz.

8.2.3 Solução de Partição após Reestruturar a Descrição da Convolução

Nesta secção apresenta-se a tentativa de melhorar o desempenho da solução de partição, reestruturando a descrição do sistema de modo a aumentar o seu grau de paralelismo explícito. A reestruturação mais natural consiste em decompor o núcleo do algoritmo de convolução, definido pelos quatro ciclos aninhados, em partes. Neste contexto, o grau de paralelismo coincide com o número de partes em que se decompõe o núcleo do algoritmo. Sem pretender viciar as conclusões a retirar do presente exemplo, procurou escolher-se um grau de paralelismo que não dificultasse a obtenção duma solução de partição com qualidade, ou seja, pretendia-se que o paralelismo a introduzir na descrição estivesse perto do paralelismo permitido pela arquitectura alvo.

Uma análise do tempo de computação e do espaço ocupado pelos estados programa, definidos na tabela 8.1, permite concluir que: (i) os estados de cada parte em que se decompõe o núcleo do algoritmo ocupam praticamente o mesmo espaço que o núcleo original (cerca de 7000 portas lógicas), (ii) cada uma das quatro FPGAs da arquitectura alvo dificilmente implementa mais do que uma parte em que se decompõe o núcleo do algoritmo, uma vez que o limite de portas lógicas utilizável por FPGA é aproximadamente 13K e o valor 7K não inclui as variáveis e (iii) o tempo de execução em *software* duma parte em que se decompõe o núcleo do

algoritmo é menos do dobro do tempo de execução em *hardware*. Estas conclusões sugerem uma decomposição do corpo do algoritmo de convolução em cinco partes. Cada parte será responsável pela execução da convolução duma secção distinta da imagem com o filtro de Sobel. Presentemente, a reestruturação da descrição dos sistemas é efectuada manualmente, mas no futuro pretende-se que seja automática.

Na situação em que o corpo do algoritmo é decomposto em cinco partes, a convolução duma imagem de 256x256 pontos com um filtro de Sobel 3x3 é efectuada por cinco tarefas, cada uma processando uma secção de 53x256 pontos com o filtro 3x3. O número de colunas a processar por cada uma das tarefas é $\lceil \frac{256-2*\lfloor 3/2 \rfloor}{5} \rceil + 2 * \lfloor \frac{3}{2} \rfloor = 53$, em que 256 é número de colunas da imagem, 5 é o número de tarefas e 3 é o número de colunas do filtro. A segunda parcela garante que o somatório das cinco convoluções parcelares coincide com a totalidade da imagem.

Partindo da descrição inicial apresentada no grafo PSMfg da figura 8.4, a reestruturação em cinco partes tem como resultado o grafo da figura 8.6. Em cada um dos cinco ramos em que se decompôs o corpo do algoritmo de convolução, o ciclo relativo ao número de colunas da imagem passa de 254 para 51 iterações. Para limitar o custo de comunicação entre partições, replicaram-se as variáveis auxiliares utilizadas nas diversas tarefas. Com o mesmo objectivo, introduziu-se (i) uma variável adicional $fimg_i$ para guardar os valores intermédios do ponto da imagem final que está a ser obtido e (ii) um estado programa $Estore1pt\_img_i$ que, após a conclusão dos cálculos, guarda o conteúdo da variável $fimg_i$ na imagem convolvida (variável *img_out*). Pelo facto de cada tarefa apenas calcular os valores mínimo e máximo na parcela de imagem por ela processada, para se poder normalizar a imagem convolvida foi necessário introduzir na descrição do sistema dois estados programa responsáveis por obter o máximo e mínimo absolutos na imagem convolvida: *Ecalc_maxGlobal* e *Ecalc_minGlobal*, respectivamente.

O código VHDL para os estados programa do modelo PSM, correspondente ao modelo PSMfg da figura 8.6, encontra-se nas figuras 8.7 e 8.8, enquanto as métricas obtidas ao nível do estado programa estão incluídas nas tabelas 8.6, 8.7 e 8.8.

A estimativa do desempenho obtido com a convolução reestruturada é apresentado na tabela 8.9, para as situações em que o sistema é totalmente implementado em *software* ou em *hardware*, para o caso em que se atribui manualmente cada tarefa[3] em que se decompôs a convolução a uma partição distinta e para a solução de partição obtida pelo algoritmo de pesquisa tabu. Como era previsível, a introdução de paralelismo faz com que o tempo de execução para uma implementação puramente de *hardware* seja reduzido de 310 ms para 83 ms. Contudo, a concretização desta solução com a arquitectura EDgAR-2 não é possível porque

[3]Uma tarefa corresponde a um ramo do principal construtor paralelo contido no grafo da figura 8.6.

exige um espaço de *hardware* de 2146000 portas lógicas equivalentes, um valor muito superior ao disponível. A maior contribuição para este espaço advém das variáveis img_in e img_out que guardam a imagem a convolver e a imagem convolvida.

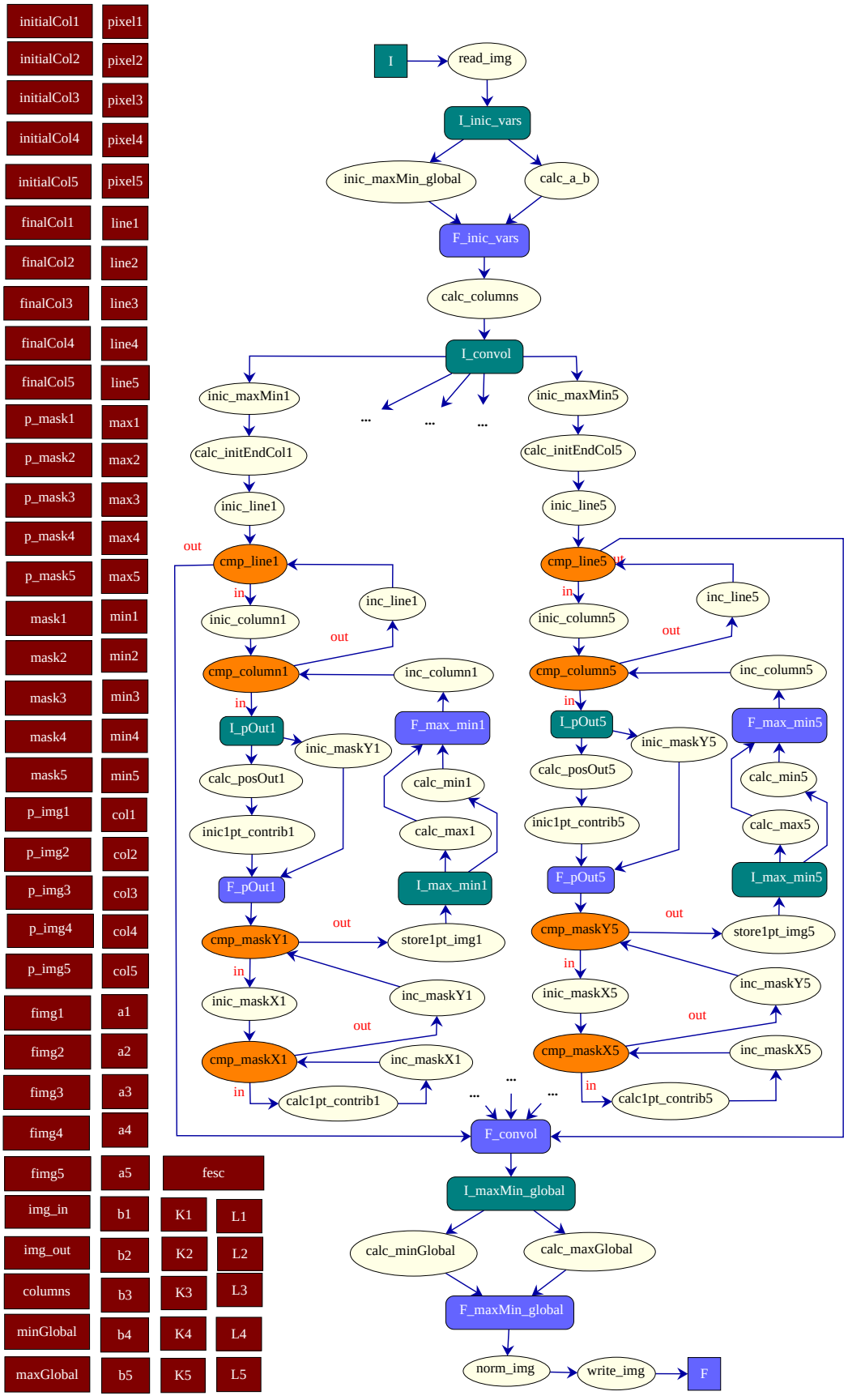


Figura 8.6: Descrição da convolução com o meta-modelo PSMfg após ter sido paralelizada em cinco tarefas.

signal maxGlobal, minGlobal: integer;
signal img_out: int_vector(0 to tx*ty);
variable $line_1$, $line_2$, a_1, a_2, a_3: integer;
variable $line_3$, $line_4$, $line_5$, a_4, a_5: integer;
variable $finalCol_1$, $finalCol_2$: integer;
variable $finalCol_3$, $finalCol_4$: integer;
variable $finalCol_5$, $initialCol_1$: integer;
variable $initialCol_2$, $initialCol_3$: integer;
variable $initialCol_4$, $initialCol_5$: integer;
variable K_1, K_2, K_3, K_4, K_5: integer;
variable p_img_1,p_img_2,columns: integer;
variable p_img_3, p_img_4, p_img_5: integer;
variable fesc: real;

constant tx,ty,maskX,maskY: integer := 256, 256, 3, 3;
signal img_in: int_vector(tx*ty downto 0);
signal min_1, min_2, min_3, min_4, min_5, b_1, b_2, b_3: integer;
signal max_1, max_2, max_3, max_4, max_5, b_4, b_5: integer;
variable col_1, col_2, col_3, col_4, col_5: integer;
variable $pixel_1$, $pixel_2$, $pixel_3$, $pixel_4$, $pixel_5$: integer;
variable L_1, L_2, L_3, L_4, L_5, p_mask_1, p_mask_2: integer;
variable p_mask_3, p_mask_4, p_mask_5: integer;
variable $mask_1$, $mask_2$: int_vector(0 to maskX*maskY);
variable $mask_3$, $mask_4$: int_vector(0 to maskX*maskY);
variable $mask_5$: int_vector(0 to maskX*maskY);
variable $fimg_1$, $fimg_2$, $fimg_3$, $fimg_4$, $fimg_5$: integer;
constant gPar: integer := 5;

Einic_maxMin_global:
==========
minGlobal <= 0;
maxGlobal <= 1;

Ecalc_columns:
==========
columns := tx/gPar+2*a_5;

Ecalc_initEndCol$_1$:
==========
$initialCol_1$:= 0;
$finalCol_1$:= columns;

Ecalc_initEndCol$_3$:
==========
$initialCol_3$:= 2*columns - 5*a_3;
$finalCol_3$:= $initialCol_3$+columns;

Ecalc_initEndCol$_5$:
==========
$initialCol_5$:= 4*columns - 10*a_5;
$finalCol_5$:= $initialCol_5$+columns;

Einc_line$_i$:
==========
$line_i$:= $line_i$ + 1;

Ecmp_column$_i$:
==========
L_COLS: while col_i<$finalCol_i$ loop
[...]
end loop L_COLS;

Ecalc_posOut$_i$:
==========
$pixel_i$:= col_i + ($line_i$ * tx);

Einic_maskY$_i$:
==========
K_i := 0;

Ecmp_maskY$_i$:
==========
L_MASK: while K_i < maskY loop
[...]
end loop L_MASK;

Ecalc_a_b:
==========
a_1:=maskX/2; a_2:=a_1; a_3:=a_1; a_4:=a_1; a_5:=a_1;
b_1:=maskY/2; b_2:=b_1; b_3:=b_1; b_4:=b_1; b_5:=b_1;

Einic_maxMin$_i$: (‡)
==========
min_i <= 0;
max_i <= 1;

Ecalc_initEndCol$_2$:
==========
$initialCol_2$:= columns-3*a_2;
$finalCol_2$:= $initialCol_2$ + columns;

Ecalc_initEndCol$_4$:
==========
$initialCol_4$:= 3*columns-7*a_4;
$finalCol_4$:= $initialCol_4$+columns;

Einic_line$_i$:
==========
$line_i$:= b_i;

Ecmp_line$_i$:
==========
L_LINES: while $line_i$<(ty-b_i) loop
[...]
end loop L_LINES;

Einic_column$_i$:
==========
col_i := $initialCol_i$;

Einc_column$_i$:
==========
col_i := col_i + 1;

Einic1pt_contrib$_i$:
==========
$fimg_i$:= 0.0;

Einc_maskY$_i$:
==========
K_i := K_i + 1;

(‡) Em qualquer estado programa cujo nome inclui o subscrito i, $1 \leqslant i \leqslant gPar$.

Figura 8.7: Código que descreve a convolução paralelizada em cinco tarefas *(parte 1)*.

Ecalc_min$_i$:
========
if($fimg_i$<min_i) then
min_i <= $fimg_i$;
end if;

Ecmp_maskX$_i$:
==========
C_MASK: while L_i < maskX loop
[...]
end loop C_MASK;

Einc_maskX$_i$:
==========
L_i := L_i + 1;

Ecalc1pt_contrib$_i$:
==========
p_img_i := ($col_i - a_i + L_i$) +
(($line_i - b_i + K_i$) * tx);
p_mask_i := L_i + K_i * maskX;
$fimg_i$:= $fimg_i$ + img_in(p_img_i) *
$mask_i$(p_mask_i);

Ecalc_maxGlobal:
==========
maxGlobal <= max_1 ;
L_MAX: for i in 2 to $gPar$ loop
if(maxGlobal > max_i) then
maxGlobal <= max_i;
end if;
end loop L_MAX;

Ecalc_max$_i$:
========
if($fimg_i$>max_i) then
max_i <= $fimg_i$;
end if;

Einic_maskX$_i$:
==========
L_i := 0;

Enorm_img:
========
maxGlobal <= maxGlobal-minGlobal;
fesc := 255/maxGlobal;
LINES: for line in b_5 to (ty-b_5) loop
COLS: for col in a_5 to (tx-a_5) loop
pixel = col + (line * tx);
img_out(pixel) <=(img_out(pixel)-
minGlobal) * fesc;
end loop COLS;
end loop LINES;

Estore1pt_img$_i$:
==========
img_out($pixel_i$) <= $fimg_i$;

Ecalc_minGlobal:
==========
minGlobal <= min_1;
L_MIN: for i in 2 to $gPar$ loop
if(minGlobal < min_i) then
minGlobal <= min_i
end if;
end loop L_MIN;

Figura 8.8: Código que descreve a convolução paralelizada em cinco tarefas *(parte 2)*.

Dado não ser possível implementar mais do que uma tarefa num componente do caminho de dados da arquitectura alvo, a solução de partição em que se atribui manualmente todos os objectos duma tarefa T_i à partição p_i ($1 \leqslant i \leqslant 5$) é uma solução que, embora não seja óptima, corresponde a um mínimo local da função de custo que está próximo da solução óptima. A prova de que é um mínimo local resulta da observação de que o algoritmo de partição tem dificuldade em escapar desta solução. A evidência de que é uma solução quase óptima é comprovada pelo valor de 251 ms estimado para o tempo de execução, um valor pouco superior ao da solução totalmente em *software*, a melhor solução observada.

A melhor solução obtida pelo algoritmo de partição, apresentada na tabela 8.10, atinge um desempenho de 254 ms, o que constitui um erro de 10% relativamente à solução totalmente em *software*. Sem ser excepcional, como o número de objectos da descrição é elevado (217) e como não se utilizou nenhuma intervenção do utilizador na definição da solução inicial, a melhor solução encontrada pelo algoritmo iterativo possui uma qualidade muito aceitável.

Estado programa	***Tipo de recurso de hardware***	***areaUF*** *(portas lógicas)*	***TexecSW*** *(ciclos do µP)*	***TexecHW*** *(ciclos do h/w)*
$Eread\_img$	–	–	–	–
$Ecalc\_a\_b$	2 x deslocação 8-bits	380	26	3
$Ecalc\_columns$	divisor 8x4bits somador 8-bits deslocação 8-bits	–	53	–
$Einic\_maxMin\_global$	–	0	4	1
$Einic\_maxMin_i$ (‡)	–	0	4	1
$Ecalc\_initEndCol_1$	–	0	4	1
$Ecalc\_initEndCol_2$	somador 8-bits multiplicador 2-bits subtractor 8-bits	430	18	3
$Ecalc\_initEndCol_3$	somador 8-bits deslocação 8-bits multiplicador 4-bits subtractor 8-bits	830	22	4
$Ecalc\_initEndCol_4$	somador 8-bits subtractor 8-bits 2 x multiplicador 4-bits	920	30	3
$Ecalc\_initEndCol_5$	somador 8-bits deslocação 8-bits multiplicador 4-bits subtractor 8-bits	830	22	5
$Einic\_line_i$	–	0	2	1
$Ecmp\_line_i$	comparador 8-bits	36	3	1
$Einc\_line_i$	contador 8-bits	160	2	1
$Einic\_column_i$	–	0	2	1
$Ecmp\_column_i$	comparador 8-bits	36	3	1
$Einc\_column_i$	contador 8-bits	160	2	1
$Ecalc\_posOut_i$	somador 16-bits multiplicador 8-bits	1510	15	4
$Einic1pt\_contrib_i$	–	0	2	1
$Einic\_maskY_i$	–	0	2	1
$Ecmp\_maskY_i$	comparador 2-bits	20	3	1
$Einc\_maskY_i$	contador 2-bits	37	2	1
$Einic\_maskX_i$	–	0	2	1
$Ecmp\_maskX_i$	comparador 2-bits	20	3	1
$Einc\_maskX_i$	contador 2-bits	37	2	1
$Ecalc1pt\_contrib_i$	2 x somador 16-bits 2 x somador 8-bits somador 4-bits 2 x subtractor 8-bits 2 x multiplicador 8-bits multiplicador 2-bits	3900	57	10
$Estore1pt\_img_i$	–	0	2	1
$Ecalc\_max_i$	comparador 16-bits	70	3	1
$Ecalc\_min_i$	comparador 16-bits	70	3	1
$Ecalc\_maxGlobal$	4 x comparador 16-bits	280	12	3
$Ecalc\_minGlobal$	4 x comparador 16-bits	280	12	3
$Enorm\_img$	–	–	58+9*254 + 86*254*254	–
$Ewrite\_img$	–	–	–	–

(‡) Em qualquer estado programa cujo nome inclui o subscrito i, $1 \leqslant i \leqslant gPar$.

Tabela 8.6: Estimativa de métricas para os estados programa da descrição da convolução paralelizada em cinco tarefas.

Variável	***Largura*** *(bits)*	***Número de elementos***	***areaHWvar*** *(portas) (lógicas)*	***Variável***	***Largura*** *(bits)*	***Número de elementos***	***areaHWvar*** *(portas) (lógicas)*
img_in	8	256*256	702126	$fesc$	32	1	192
img_out	16	256*256	1402428	$columns$	8	1	48
$maxGlobal$	16	1	96	$pixel_i$ (‡)	16	1	96
$minGlobal$	16	1	96	$line_i$	8	1	48
$mask_i$	4	9	57	max_i	16	1	96
col_i	8	1	48	min_i	16	1	96
a_i	2	1	12	p_img_i	16	1	96
b_i	2	1	12	p_mask_i	4	1	24
$initialCol_i$	8	1	48	K_i	2	1	12
$finalCol_i$	8	1	48	L_i	2	1	12
$fimg_i$	16	1	96				

(‡) Em qualquer variável cujo nome inclui o subscrito i, $1 \leqslant i \leqslant gPar$.

Tabela 8.7: Estimativa de métricas para as variáveis da descrição da convolução paralelizada em cinco tarefas.

Estado programa	***Variáveis lidas***	***Variáveis escritas***	***Variáveis com multiplexador***
$Ecalc\_a\_b$	–	a_1 .. a_5, b_1 .. b_5	–
$Ecalc\_columns$	a_5	columns	–
$Einic\_maxMin\_global$	–	minGlobal, maxGlobal	–
$Einic\_maxMin_i$	–	min_i, max_i	–
$Ecalc\_initEndCol_i$	a_i (*), columns	$initialCol_i$, $finalCol_i$	–
$Einic\_line_i$	b_i	$line_i$	–
$Ecmp\_line_i$	b_i, $line_i$	–	–
$Einc\_line_i$	$line_i$	$line_i$	–
$Einic\_column_i$	$initialCol_i$	col_i	–
$Ecmp\_column_i$	$finalCol_i$, col_i	–	–
$Einc\_column_i$	col_i	col_i	–
$Ecalc\_posOut_i$	col_i, $line_i$	$pixel_i$	–
$Einic1pt\_contrib_i$	–	$fimg_i$	–
$Einic\_maskY_i$	–	K_i	–
$Ecmp\_maskY_i$	K_i	–	–
$Einc\_maskY_i$	K_i	K_i	–
$Einic\_maskX_i$	–	L_i	–
$Ecmp\_maskX_i$	L_i	–	–
$Einc\_maskX_i$	L_i	L_i	–
$Ecalc1pt\_contrib_i$	col_i, $line_i$, a_i, b_i, $fimg_i$, $mask_i$, img_in, L_i, K_i	p_img_i, p_mask_i, $fimg_i$	–
$Estore1pt\_img_i$	$pixel_i$, $fimg_i$	img_out	–
$Ecalc\_max_i$	$fimg_i$, max_i	max_i	–
$Ecalc\_min_i$	$fimg_i$, min_i	min_i	–
$Ecalc\_maxGlobal$	max_1, max_2, max_3, max_4, max_5	maxGlobal (††)	maxGlobal (4 entradas)
$Ecalc\_minGlobal$	min_1, min_2, min_3, min_4, min_5	minGlobal (††)	minGlobal (4 entradas)
$Enorm\_img$	maxGlobal, a_5, b_5, minGlobal, $line_5$ (†), col_5 (§), img_out (‡)	maxGlobal, fesc, $pixel_5$ (‡), img_out (‡)	–

(*) excepto $i = 1$ (†) 255 vezes (††) 4 vezes (§) 255*254 vezes (‡) 254*254 vezes

Tabela 8.8: Estimativa, para cada estado programa da descrição da convolução paralelizada em cinco tarefas, das variáveis lidas, escritas e que precisam dum multiplexador na entrada.

Tipo de solução de partição	***Valor do tempo de execução*** *(ms)*
Solução totalmente em *software*	231
Solução totalmente em *hardware*, supondo que $TexecHW(E_{norm\_img})$ é igual a $TexecSW(E_{norm\_img})$	83
Solução totalmente em *hardware*, sem contabilizar E_{norm_img}	51
Solução em que o ramo i do construtor paralelo é atribuído à partição HW_i ou SW	251
Solução obtida pelo algoritmo de pesquisa tabu	254

Tabela 8.9: Estimativa para o desempenho da convolução paralelizada em cinco tarefas, quando se considera um relógio de *hardware* a 50 MHz e de *software* a 200 MHz.

HW1	***HW2***	***HW3***	***HW4***	***SW***
$Einic\_maxMin_1$	$Einic\_line_2$	$Ecalc\_initEndCol_3$	$Einic\_maxMin_3$	$Ecalc\_posOut_5$
$Einic\_maxMin_2$	$Ecmp\_line_2$	$Ecalc\_initEndCol_5$	$Einic\_maxMin_4$	$Ecmp\_maskY_5$
$Einic\_maxMin_5$	$Einc\_line_2$	$Einic\_line_3$	$Ecalc\_initEndCol_1$	$Einc\_maskY_5$
$Einic\_line_1$	$Einic\_column_2$	$Ecmp\_line_3$	$Ecalc\_initEndCol_2$	$Einic\_maskX_5$
$Einic\_line_5$	$Einic\_column_5$	$Einc\_line_3$	$Ecalc\_initEndCol_4$	$Ecmp\_maskX_5$
$Ecmp\_line_1$	$Ecmp\_column_2$	$Einic\_column_3$	$Einic\_line_4$	$Einc\_maskX_5$
$Ecmp\_line_5$	$Ecmp\_column_5$	$Ecmp\_column_3$	$Ecmp\_line_4$	$Ecalc1pt\_contrib_5$
$Einc\_line_1$	$Einc\_column_2$	$Einc\_column_3$	$Einc\_line_4$	$Estore1pt\_img_5$
$Einc\_line_5$	$Ecalc\_posOut_2$	$Ecalc\_posOut_3$	$Einic\_column_4$	$Ecalc\_max_5$
$Einic\_column_1$	$Einic1pt\_contrib_2$	$Einic1pt\_contrib_3$	$Ecmp\_column_4$	$Ecalc\_min_5$
$Ecmp\_column_1$	$Einic\_maskY_2$	$Einic\_maskY_3$	$Einc\_column_4$	EI_pOut_5
$Einc\_column_1$	$Ecmp\_maskY_2$	$Ecmp\_maskY_3$	$Ecalc\_posOut_4$	EF_pOut_5
$Einc\_column_5$	$Einc\_maskY_2$	$Einc\_maskY_3$	$Einic1pt\_contrib_4$	$EI\_max\_min_5$
$Ecalc\_posOut_1$	$Einic\_maskX_2$	$Einic\_maskX_3$	$Einic\_maskY_4$	$EF\_max\_min_5$
$Einic1pt\_contrib_1$	$Ecmp\_maskX_2$	$Ecmp\_maskX_3$	$Ecmp\_maskY_4$	$Ecalc\_columns$
$Einic1pt\_contrib_5$	$Einc\_maskX_2$	$Einc\_maskX_3$	$Einc\_maskY_4$	$Enorm\_img$
$Einic\_maskY_1$	$Ecalc1pt\_contrib_2$	$Ecalc1pt\_contrib_3$	$Einic\_maskX_4$	$Einic\_maxMin\_global$
$Einic\_maskY_5$	$Estore1pt\_img_2$	$Estore1pt\_img_3$	$Ecmp\_maskX_4$	$Ecalc\_maxGlobal$
$Ecmp\_maskY_1$	$Ecalc\_max_2$	EI_pOut_3	$Einc\_maskX_4$	EI_convol
$Einc\_maskY_1$	$Ecalc\_min_2$	EF_pOut_3	$Ecalc1pt\_contrib_4$	EF_convol
$Einic\_maskX_1$	EI_pOut_2	$EI\_max\_min_3$	$Estore1pt\_img_4$	EI_inic_vars
$Ecmp\_maskX_1$	EF_pOut_2	$EF\_max\_min_3$	$Ecalc\_max_3$	EF_inic_vars
$Einc\_maskX_1$	$EI\_max\_min_2$	$Ecalc\_a\_b$	$Ecalc\_max_4$	EI_maxMin_global
$Ecalc1pt\_contrib_1$	$EF\_max\_min_2$	$mask_3$	$Ecalc\_min_1$	EF_maxMin_global
$Estore1pt\_img_1$	$pixel_2$	col_3	$Ecalc\_min_3$	$fimg_5$
$Ecalc\_max_1$	$mask_2$	$line_3$	$Ecalc\_min_4$	$pixel_5$
EI_pOut_1	col_2	a_3	EI_pOut_4	$mask_5$
EF_pOut_1	$line_2$	a_5	EF_pOut_4	col_5
$EI\_max\_min_1$	a_1	b_2	$EI\_max\_min_4$	$line_5$
$EF\_max\_min_1$	a_2	b_3	$EF\_max\_min_4$	$maxGlobal$
$Ecalc\_minGlobal$	b_1	p_img_4	$pixel_4$, K_4	$fesc$
$mask_1$	p_img_2	p_mask_4	$mask_4$, L_4	max_5
col_1	p_mask_2	K_3	col_4 , $line_4$	min_5
$line_1$	K_2	L_3	$initialCol_1$	b_5
max_1	L_2	$initialCol_3$	$initialCol_2$	p_img_5
max_2	$initialCol_5$	$finalCol_3$	$initialCol_4$	K_5
min_1	$finalCol_2$	$pixel_3$	$finalCol_1$	L_5
min_2	$finalCol_5$	$columns$	$finalCol_4$	img_out
K_1	$fimg_2$		max_3 , max_4	img_in
L_1			min_3 , min_4	
$fimg_1$			p_img_1	
$pixel_1$			p_img_3	
$minGlobal$			p_mask_1	
			p_mask_3 , a_4	
			p_mask_5 , b_4	
			$fimg_3$, $fimg_4$	

Tabela 8.10: Melhor solução de partição obtida pelo algoritmo de pesquisa tabu para a convolução paralelizada em cinco tarefas.

O processo de partição com o algoritmo de pesquisa tabu utilizou os seguintes parâmetros:

- o número de iterações efectuado foi 43400; este valor obtém-se com a equação 6.11, em que a constante $K2ni$ é 10, o número de partições $nPart$ é 5 e o número de objectos $nObj$ da descrição é 217; deste modo, garante-se que em média cada deslocamento é efectuado 10 vezes;
- o limite imposto ao número de iterações sucessivas sem que haja melhoria da solução de partição foi 300;
- a percentagem de objectos que se deslocou para construir a solução inicial duma pesquisa foi 20%;
- a validade do tabu aplicado aos deslocamentos, aos deslocamentos inversos e aos objectos foi de 20, 18 e 15 iterações, respectivamente;
- na terceira alternativa de deslocamento, que funciona como recurso para a definição da próxima solução duma pesquisa, seleccionou-se o deslocamento menos frequente ou o deslocamento de maior qualidade.

A solução de partição que se obteve respeita os condicionalismos impostos ao espaço utilizável pelo caminho de dados e pela unidade de controlo duma partição de *hardware*, como pode ser constatado pelos valores das métricas a seguir introduzidos.

As 4 partições de *hardware* estabelecidas pelo algoritmo de partição exigem um espaço inferior às 13100 portas lógicas permitidas ao caminho de dados de cada partição: o caminho de dados da partição HW1 a HW4 ocupa 10.1K, 7.6K, 9.2K e 10.5K portas lógicas. Além da estimativa do espaço ocupado pelo caminho de dados, a tabela 8.11 contém também o valor dos diferentes contributos para esse espaço. De acordo com a equação 7.7, as parcelas que contribuem para o espaço do caminho de dados são as unidades funcionais, os elementos de armazenamento, os elementos de interligação e os recursos de interface.

O espaço ocupado pela unidade de controlo das partições de *hardware* obtidas pelo algoritmo de pesquisa tabu não excede as 2400 portas lógicas disponíveis. Como se pode observar pela tabela 8.15, a unidade de controlo das partições HW1 a HW4 ocupa 1.69K, 1.10K, 1.25K e 1.28K portas lógicas.

A menos de uma constante, o espaço ocupado pela unidade de controlo duma partição de *hardware* coincide com o espaço ocupado pela respectiva máquina de estados. A equação 7.39, que define o espaço ocupado por uma máquina de estados, considera o registo de estado, a lógica de controlo e a lógica do próximo estado.

Métrica	Solução com o ramo i do construtor paralelo atribuído à partição HW_i				Solução obtida pelo algoritmo de pesquisa tabu			
	$p =$ ***HW1***	$p =$ ***HW2***	$p =$ ***HW3***	$p =$ ***HW4***	$p =$ ***HW1***	$p =$ ***HW2***	$p =$ ***HW3***	$p =$ ***HW4***
$areaEPrograma(CD(p))$	6043	6473	6873	6963	6881	6079	7943	7603
$areaVars(CD(p))$	801	801	801	801	849	669	585	1377
$areaInterface(CD(p))$	560	560	560	560	1516	568	404	814
$areaMuxesExtra(CD(p))$	476	476	476	476	868	252	252	700
areaHW(CD(p))	***7880***	***8310***	***8710***	***8800***	***10114***	***7568***	***9184***	***10494***

Tabela 8.11: Estimativa, para cada partição de *hardware* incluída na implementação da convolução paralelizada em cinco tarefas, das métricas utilizadas no cálculo do espaço ocupado pelo caminho de dados dessas partições.

As tabelas 8.12 a 8.15 contêm as métricas que permitem obter o valor do espaço ocupado pela unidade de controlo das partições de *hardware* presentes na solução obtida pelo algoritmo de pesquisa tabu e na solução em que se atribuem manualmente todos os objectos duma tarefa T_i à partição p_i (**solução comparativa**):

- ◇ os sinais de selecção do multiplexador a colocar na entrada das variáveis escritas pelos estados programa (tabela 8.12);
- ◇ o número de estados ($nE(o)$), de sinais de controlo gerados ($nSC(o)$), de activações dos sinais de controlo ($nASC(o)$), de sinais de estado lidos ($nSE(o)$) e de leituras dos sinais de estado ($nLSE(o)$) necessários à secção de máquina de estados que coordena o funcionamento do estado o (tabelas 8.13 e 8.14);
- ◇ as métricas $nSCmux(p)$ e $nASCmux(p)$, que representam o contributo dos sinais de selecção dos multiplexadores para as métricas nSC e $nASC$ da partição p (tabela 8.15);
- ◇ as métricas nE a $nLSE$ relativas aos estados programa introduzidos para gerir as mudanças de partição no fluxo de controlo; estas métricas são designadas por $nEpart(p)$ a $nLSEpart(p)$ na tabela 8.15.

Com base nas métricas anteriores obtêm-se as métricas $nE(p)$ a $nLSE(p)$ ao nível da partição, o espaço ocupado pelo registo de estado, pela lógica de controlo e pela lógica do próximo estado.

As tabela 8.11 e 8.15 mostram que, para atingir um desempenho idêntico, a solução de partição obtida pelo algoritmo de pesquisa tabu exige mais espaço em *hardware* do que a solução comparativa. Esta situação ocorre porque a função de custo só penaliza as soluções de partição que ultrapassam o espaço disponível para o caminho de dados ou para a unidade de controlo das partições de *hardware*. Ao não exceder qualquer dos condicionalismos impostos ao espaço, a solução obtida pelo algoritmo PT não é penalizada, permitindo que seja considerada tão boa como qualquer outra solução que, embora ocupando menos espaço, atinge o

mesmo desempenho.

Da tabela 8.15 pode constatar-se que a solução obtida pelo algoritmo de partição exige mais estados, mais sinais de controlo e de estado que a solução comparativa, o que obriga a dispor duma máquina de estados com um espaço bastante maior. A explosão do número de estados e de sinais resulta essencialmente duma comunicação mais intensa entre partições do que na solução comparativa. O número elevado de sinais de controlo não é preocupante porque o seu valor é fortemente penalizado pelo facto de as estimativas não considerarem que a comunicação entre duas partições de *hardware* adjacentes utiliza um conjunto de sinais de controlo comum a todas as transferências.

Variável	Solução com o ramo i do construtor paralelo atribuído à partição HW_i				Solução obtida pelo algoritmo de pesquisa tabu			
	HW1	***HW2***	***HW3***	***HW4***	***HW1***	***HW2***	***HW3***	***HW4***
$minGlobal$					1			
$line_1$	1				1			
$line_2$		1				1		
$line_3$			1				1	
$line_4$				1				1
$line_5$					1			
col_1	1				1			
col_2		1				1		
col_3			1				1	
col_4				1				1
max_1	1				1			
max_2		1			1			
max_3			1					1
max_4				1				1
min_1	1				1			
min_2		1			1			
min_3			1					1
min_4				1				1
L_1	1				1			
L_2		1				1		
L_3			1				1	
L_4				1				1
K_1	1				1			
K_2		1				1		
K_3			1				1	
K_4				1				1
$fimg_1$	1				1			
$fimg_2$		1				1		
$fimg_3$			1				1	
$fimg_4$				1				1
Total de sinais de selecção	***7***	***7***	***7***	***7***	***11***	***5***	***5***	***9***

Tabela 8.12: Solução de partição para a convolução paralelizada em cinco tarefas: sinais de selecção do multiplexador a colocar na entrada de cada variável escrita pelos estados programa atribuídos às partições de *hardware*.

Para atingir uma solução de partição mais próxima da solução óptima, seria necessário executar um número de iterações muito superior e dispor dum algoritmo de pesquisa tabu com

todos os parâmetros afinados para este problema. Não se efectuaram sessões de partição com um número de iterações muito superior ao indicado porque o tempo de cálculo se revelou elevado. Para o tempo de cálculo contribui de forma preponderante o elevado tempo despendido na estimação do tempo de execução do sistema. O peso desta contribuição no tempo de cálculo global é potenciado pelo facto de a estimação ser efectuada aproximadamente 200 vezes por iteração[4]. Contudo, com uma média de 1000 iterações e menos de 10 minutos conseguem-se soluções de partição que apresentam um desempenho apenas 10% inferior ao obtido com 43400 iterações.

Estado programa	***Solução em que o ramo i do construtor paralelo é atribuído a*** HW_i				
	nE	***nSC***	***nASC***	***nSE***	***nLSE***
$Einic\_maxMin_i$	1	1	1	0	0
$Ecalc\_initEndCol_1$	4	2	3	1	1
$Ecalc\_initEndCol_2$	8	6	7	1	1
$Ecalc\_initEndCol_{3,4,5}$	9	3	4	1	1
$Einic\_line_i + Ecmp\_line_i + Einc\_line_i$	4	2	4	1	1
$Einic\_column_i + Ecmp\_column_i + Einc\_column_i$	4	2	4	1	1
$Ecalc\_posOut_i$	4	1	1	0	0
$Einic1pt\_contrib_i$	1	1	1	0	0
$Einic\_maskY_i + Ecmp\_maskY_i + Einc\_maskY_i$	4	2	4	1	1
$Einic\_maskX_i + Ecmp\_maskX_i + Einc\_maskX_i$	4	2	4	1	1
$Ecalc1pt\_contrib_i$	15	4	5	1	1
$Estore1pt\_img_i$	3	1	2	1	1
$Ecalc\_max_i$	3	1	1	1	1
$Ecalc\_min_i$	3	1	1	1	1
$EI\_pOut_i + EF\_pOut_i$	8	6	12	6	6
$EI\_max\_min_i + EF\_max\_min_i$	6	4	8	4	4
outros estados	–	–	–	–	–

Tabela 8.13: Estimativa, para cada estado programa da descrição da convolução paralelizada em cinco tarefas, das métricas envolvidas no cálculo do espaço ocupado pela unidade de controlo das partições de *hardware* (solução comparativa).

Como mostra a estimativa do desempenho da solução totalmente em *hardware*, pare se obter uma solução de partição com elevado desempenho para um sistema predominantemente de dados seria necessário dispor de memória em quantidade significativa e de rápido acesso por parte do *hardware*. Obviamente que ao dispor-se de módulos de memória directamente acessíveis aos componentes de *hardware* este objectivo é conseguido. Outra alternativa para reduzir a comunicação com a memória, consiste em implementar um mecanismo equivalente à **hierarquia de memória** presente nos sistemas computacionais baseados em microprocessadores. Ou seja, ao dispor-se próximo dos componentes de *hardware* de uma memória de dimensão inferior à disponível no *software* e que pode ser preenchida de uma só vez, através dum mecanismo de transferência mais rápido que a transacção palavra-a-palavra, consegue

[4]O número de vezes que se calcula o tempo de execução por iteração é próximo do número de objectos presentes na descrição do sistema, que no caso da convolução é 217.

reduzir-se o tempo de comunicação. O principio descrito é similar ao modo como funciona a hierarquia de memória constituída pela memória *cache* e pela memória RAM de nível inferior nos computadores. A secção 8.2.4 mostra como este principio foi aplicado na obtenção duma implementação eficiente para a convolução, envolvendo a plataforma EDgAR-2.

Estado programa	***Solução obtida pelo algoritmo de pesquisa tabu***				
	nE (†)	***nSC*** (†)	***nASC*** (†)	***nSE*** (†)	***nLSE*** (†)
$Einic\_maxMin_i$	1, 1, 1, 1, 6	1, 1, 1, 1, 1	1, 1, 1, 1, 4	0, 0, 0, 0, 1	0, 0, 0, 0, 2
$Ecalc\_initEndCol_i$	4, 13, 6, 9, 20	2, 4, 6, 3, 12	3, 7, 6, 4, 22	1, 3, 0, 1, 0	1, 3, 0, 1, 0
$Einic\_line_i$ + $Ecmp\_line_i$ + $Einc\_line_i$	16, 16, 4, 4, 19	10, 10, 2, 2, 6	16, 16, 4, 4, 12	1, 1, 1, 1, 7	1, 1, 1, 1, 7
$Einic\_column_i$ + $Ecmp\_column_i$ + $Einc\_column_i$	16, 7, 4, 4, 10	10, 3, 2, 2, 4	16, 6, 4, 4, 8	1, 2, 1, 1, 3	1, 2, 1, 1, 3
$Ecalc\_posOut_i$	4, 4, 4, 4, -	1, 1, 1, 1, -	1, 1, 1, 1, -	0, 0, 0, 0, -	0, 0, 0, 0, -
$Einic1pt\_contrib_i$	1, 1, 5, 1, 3	1, 1, 4, 1, 1	1, 1, 8, 1, 2	0, 0, 0, 0, 1	0, 0, 0, 0, 1
$Einic\_maskY_i$ + $Ecmp\_maskY_i$ + $Einc\_maskY_i$	4, 4, 4, 4, 3	2, 2, 2, 2, 1	4, 4, 4, 4, 2	1, 1, 1, 1, 1	1, 1, 1, 1, 1
$Einic\_maskX_i$ + $Ecmp\_maskX_i$ + $Einc\_maskX_i$	4, 4, 4, 4, -	2, 2, 2, 2, -	4, 4, 4, 4, -	1, 1, 1, 1, -	1, 1, 1, 1, -
$Ecalc1pt\_contrib_i$	37, 19, 33, 23, -	20, 7, 17, 10, -	33, 12, 32, 19, -	1, 1, 1, 1, -	1, 1, 1, 1, -
$Estore1pt\_img_i$	3, 3, 9, 3, -	1, 1, 5, 1, -	2, 2, 8, 2, -	1, 1, 1, 1, -	1, 1, 1, 1, -
$Ecalc\_max_i$	3, 13, 3, 3, -	1, 8, 1, 3, -	1, 14, 1, 3, -	1, 1, 1, 3, -	1, 1, 1, 3, -
$Ecalc\_min_i$	19, 13, 3, 3, -	12, 8, 1, 1, -	20, 14, 1, 1, -	1, 1, 1, 1, -	1, 1, 1, 1, -
EI_pOut_i + EF_pOut_i	8, 8, 8, 8, -	6, 6, 6, 6, -	12, 12, 12, 12, -	6, 6, 6, 6, -	6, 6, 6, 6, -
$EI\_max\_min_i$ + $EF\_max\_min_i$	6, 6, 6, 6, -	4, 4, 4, 4, -	8, 8, 8, 8, -	4, 4, 4, 4, -	4, 4, 4, 4, -
$Ecalc\_a\_b$	15	12	24	0	0
$Ecalc\_minGlobal$	15	9	13	0	0
outros estados	–	–	–	–	–

(†) Métrica para i = *1,2,3,4,5*

Tabela 8.14: Estimativa, para cada estado programa da descrição da convolução paralelizada em cinco tarefas, das métricas utilizadas no cálculo do espaço ocupado pela unidade de controlo das partições de *hardware* (solução obtida pelo algoritmo PT).

8.2.4 Implementação *Hardware/Software* Optimizada

No contexto duma abordagem ao problema de partição, a implementação dos sistemas tem por objectivo: (i) medir o desempenho do sistema, para avaliar a qualidade da solução de partição obtida automaticamente pelo algoritmo de pesquisa tabu, (ii) avaliar a precisão das estimativas e (iii) validar a correcção da descrição do sistema. Deste modo, esta secção apresenta uma implementação *hardware/software* eficiente da convolução com a plataforma EDgAR-2, aplicando uma filosofia idêntica à da hierarquia de memória dos computadores.

A descrição a utilizar na implementação é idêntica à que se utilizou na secção 8.2.3. O núcleo do algoritmo de convolução foi decomposto em quatro partes, mas poderia ter-se usado

cinco partes tal como acontece na secção 8.2.3, correspondendo cada parte a uma partição de *hardware*. Posteriormente, a descrição foi sujeita a alguns refinamentos, com o objectivo de

- ◇ melhorar a relação entre a computação e a comunicação nas partições de *hardware*;
- ◇ incluir na descrição os estados programa que implementam as mudanças de partição que ocorrem no fluxo de controlo.

Métrica	Solução em que o ramo i do construtor paralelo é atribuído à partição HW_i				Solução obtida pelo algoritmo de pesquisa tabu			
	$p =$ ***HW1***	$p =$ ***HW2***	$p =$ ***HW3***	$p =$ ***HW4***	$p =$ ***HW1***	$p =$ ***HW2***	$p =$ ***HW3***	$p =$ ***HW4***
$\sum_{o \in p} nE(o)$	64	68	69	69	153	103	122	120
$\sum_{o \in p} nSC(o)$	30	34	31	31	80	55	75	60
$\sum_{o \in p} nASC(o)$	52	56	53	53	137	98	137	101
$\sum_{o \in p} nSE(o)$	19	19	19	19	29	20	16	28
$\sum_{o \in p} nLSE(o)$	19	19	19	19	30	20	16	28
$nEpart(p)$	3	3	3	3	36	12	12	18
$nSCpart(p)$	1	1	1	1	1	1	1	1
$nASCpart(p)$	2	2	2	2	24	8	8	12
$nSEpart(p)$	1	1	1	1	1	1	1	1
$nLSEpart(p)$	1	1	1	1	12	4	4	6
$nSCmux(p)$, $nASCmux(p)$	7	7	7	7	11	5	5	9
$nE(p)$	67	71	72	72	189	115	134	138
$nSC(p)$	38	42	39	39	92	61	81	70
$nASC(p)$	61	65	62	62	172	111	150	122
$nSE(p)$	20	20	20	20	30	21	17	29
$nLSE(p)$	20	20	20	20	42	24	20	34
areaRegEstado(p)	402	426	432	432	1134	690	804	828
areaLogicaCtl(p)	69	69	69	69	240	150	207	156
areaLProxEstado(p)	91	90	90	90	166	109	89	141
areaHWfsm(UC(p))	562	585	591	591	1540	949	1100	1125
areaHW(UC(p))	***712***	***735***	***741***	***741***	***1690***	***1099***	***1250***	***1275***

Tabela 8.15: Estimativa, para cada partição de *hardware* incluída na implementação da convolução paralelizada em cinco tarefas, das métricas envolvidas no cálculo do espaço ocupado pela unidade de controlo dessas partições.

Para melhorar a relação entre a carga computacional e a comunicação nas partições de *hardware* recorreu-se a uma estratégia idêntica à duma hierarquia de memória. O *hardware* é mais rápido a processar a imagem, mas como não se consegue armazenar toda a imagem em *hardware*, o seu desempenho é penalizado pelo tempo de leitura e de escrita dos pontos da imagem a partir de *software*. Para atenuar esta penalidade replica-se em *hardware* uma parte da imagem. No caso concreto em que se aplica um filtro 3x3 à imagem, optou-se por guardar em *hardware* três linhas da imagem, que permitem calcular uma linha da imagem final. Esta alteração obriga a que, por cada linha da imagem, o algoritmo de convolução efectue as seguintes tarefas:

- ◇ o *software* envia três linhas da imagem original para as quatro partições de *hardware*, recebendo cada uma a respectiva parcela das três linhas; para uma imagem de 256x256 as parcelas de imagem definidas incluem as colunas [0:65], [63:128], [127:192] e [190:255];
- ◇ cada partição de *hardware* processa a parcela das três linhas recebidas com o filtro Sobel de dimensão 3x3, produzindo uma parcela de linha da imagem filtrada não normalizada; a partição actualiza os valores máximo e mínimo que vai obtendo para a imagem filtrada;
- ◇ quando as quatro partições de *hardware* tiverem gerado uma linha completa da imagem filtrada, o *software* lê-a para a memória do sistema hospedeiro.

Após a convolução de todas as linhas da imagem, a partição de *software* normaliza a imagem filtrada, utilizando para isso o maior (menor) de entre os quatro máximos (mínimos) calculados pelas partições de *hardware*.

De acordo com as alterações descritas, o algoritmo de convolução é representado pelo modelo PSM da figura 8.9, o qual está organizado em cinco estados programa concorrentes: o estado E_{rw_file}, correspondente a uma partição de *software* e os estados $E_{pconvol1}$ a $E_{pconvol4}$, correspondentes a quatro partições de *hardware*. O código VHDL que define a funcionalidade da partição de *software* é apresentado nas figuras 8.10 a 8.12, enquanto o código relativo às partições de *hardware* se encontra nas figuras 8.13 e 8.14.

O modelo da figura 8.9 já contém os estados programa que suportam as mudanças de partição que ocorrem no fluxo de controlo. Tal como foi explicado no capítulo 7, uma mudança de partição é modelada por um estado programa do tipo *comutacao*, que atribui um valor a um sinal e por um estado do tipo *sincronizacao* que espera por um evento nesse sinal. Dois exemplos desta situação, incluídos na figura 8.9, são os pares ($Eset\_rdone$,$Ewait\_rdone$) e ($Eset\_accept3l$,$Ewait\_accept3ls$). As mudanças de partição, modeladas desta forma, são directamente implementadas com o mecanismo de auscultação, mas nada impede que sejam implementadas com o mecanismo de interrupção.

A figura 8.15 ilustra o resultado do escalonamento manual, efectuado ao nível do sistema, do modelo PSM incluído na figura 8.9, contendo apenas os estados programa da partição de *software* e de uma partição de *hardware*[5]. Considerando uma arquitectura alvo uni-processador e um modelo de estimação simples, o escalonamento da partição de *software* é perfeitamente sequencial. As mudanças de partição no fluxo de controlo têm como consequência a introdução de pontos de sincronismo, ou seja, ciclos de espera. Os pontos de sincronismo presentes na figura 8.15 estão assinalados por um losango, contendo o texto $synch_i$, a ligar os estados que intervêm no sincronismo. Por exemplo, $synch_1$ indica que existe sincronismo entre $Eset\_rdone$ e $Ewait\_rdone$.

[5] O escalonamento das quatro partições de *hardware* é idêntico.

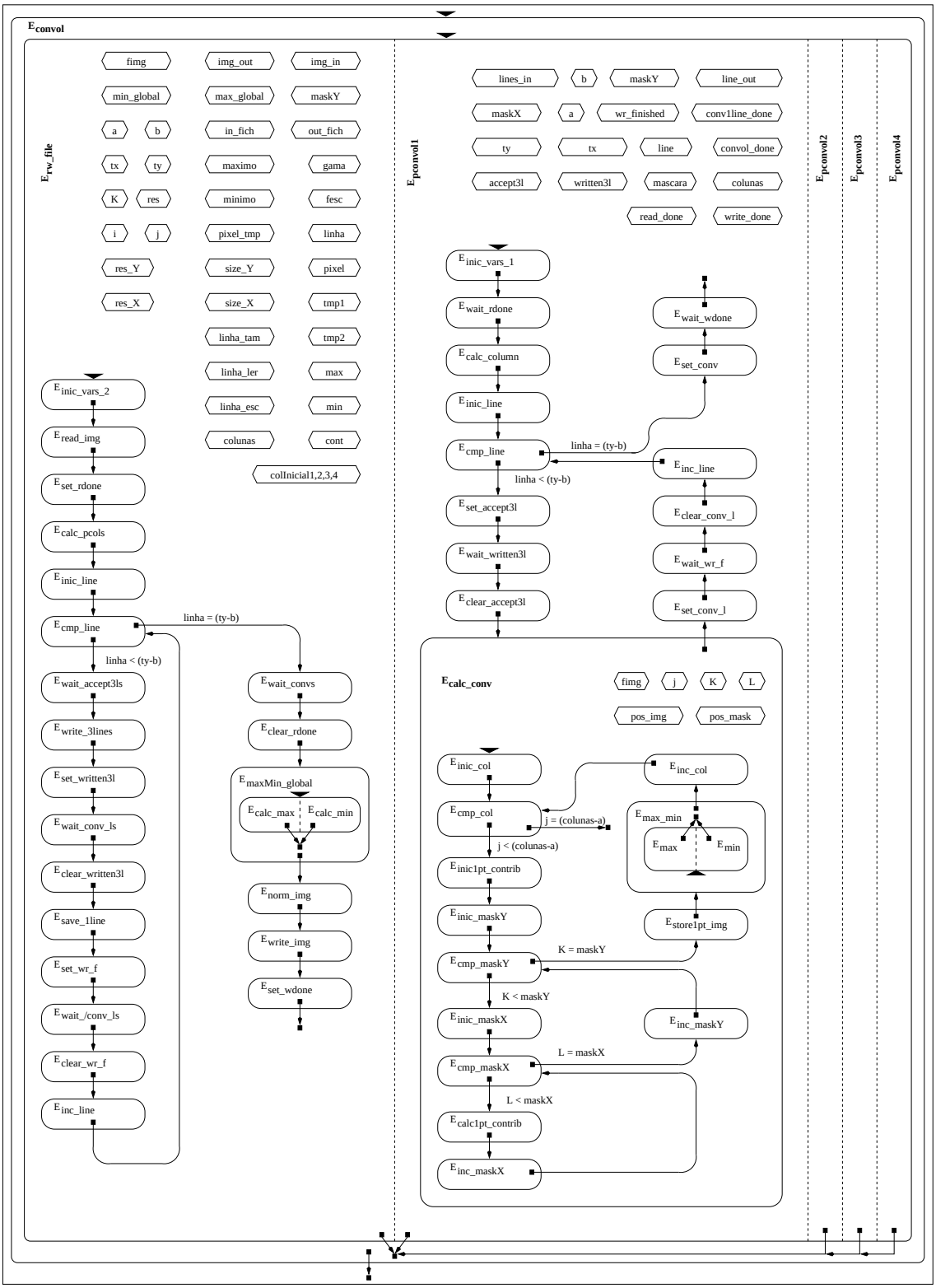


Figura 8.9: Modelo PSM da convolução aplicado na implementação *hardware/software* optimizada.

```
TYPE mat is array (0 to 255, 0 to 255) of integer;
TYPE quadInteger is array (1 to 4) of integer;
signal max_global, min_global, tx, ty: integer;
file in_fich, out_fich: TEXT;
variable linha_tam: line;
variable res, res_X, res_Y: integer;
variable maximo, minimo: integer;
variable size_Y: std_ulogic_vector (11 downto 0);
variable pixel: std_ulogic_vector (31 downto 0);
variable size_X: std_ulogic_vector (11 downto 0);
variable pixel_tmp: std_ulogic_vector (7 downto 0);
signal img_in, img_out: mat;
signal max, min: quadInteger;
constant maskY: integer := 3;
variable a, b, colunas: integer;
variable linha_esc, linha_ler: line;
variable gama, fimg, i, j, k,: integer;
variable tmp1, tmp2, cont, linha: integer;
variable colInicial1, colInicial2: integer;
variable colInicial3, colInicial4: integer;
variable fesc: real;
```

Einic_vars_2:

```
==========
    read_done <= false;
    write_done <= false;
    wr_finished <= false;
    written3l <= false;
    max_global <= 1;
    min_global <= 0;
    maximo := 1; minimo := 0;
    a := maskX/2;
    b := maskY/2;
```

Eread_img:

```
========
    FILE_OPEN(in_fich,"emma_hex.raw",
       READ_MODE);
    readline (in_fich, linha_ler);
    hread(linha_ler,size_X,leu);
    assert leu
       report "Erro na leitura de TX do ficheiro"
       severity ERROR;
    res_X := stdulogic2int(size_X);
    tx <= res_X;
    hread (linha_ler,size_Y,leu);
    assert leu
       report "Erro na leitura de TY do ficheiro"
       severity ERROR;
    res_Y := stdulogic2int(size_Y);
    ty <= res_Y;
    cont := 0;
    loop
       if endfile(in_fich) then
          assert false
             report "Fim de Leitura do Ficheiro"
             severity NOTE;
          exit;
       end if;
       readline(in_fich,linha_ler);
       ciclo2: for i in 0 to (tx-1) loop
          hread(linha_ler,pixel_tmp,leu);
          res:=stdulogic2intuns(pixel_tmp);
          img_in(cont,i) <= res;
       end loop ciclo2;
       cont := cont+1;
    end loop;
    FILE_CLOSE(in_fich);
```

Eset_rdone:

```
========
    read_done <= true;
```

Ecalc_pcols:

```
========
    colunas := (tx/4)+2*a;
    colInicial1 := 0;
    colInicial2 := (tx/4)-a;
    colInicial3 := (2*(tx/4))-a;
    colInicial4 := (3*(tx/4))-2*a;
```

Einic_line:

```
=======
    linha := b;
```

Ecmp_line:

```
=======
    LINHAS: while linha < (ty - b) loop
       [...]
    end loop LINHAS;
```

Ewait_accept3ls:

```
==========
    wait until ((accept3l(1)=true) and
       (accept3l(2)=true) and (accept3l(3)=true)
       and (accept3l(4)=true) );
```

Ewrite_3lines:

```
==========
    k := 0;
    W3LINHAS: for j in (linha-b) to
          (linha+b) loop
       for i in 0 to (colunas-1) loop
          lines_in1(k) <= img_in(j,i+colInicial1);
          lines_in2(k) <= img_in(j,i+colInicial2);
          lines_in3(k) <= img_in(j,i+colInicial3);
          lines_in4(k) <= img_in(j,i+colInicial4);
          k := k+1;
       end loop;
    end loop W3LINHAS;
```

Eset_written3l:

```
==========
written3l <= true;
```

Figura 8.10: Código do estado E_{rw_file} usado na implementação *hardware/software* da convolução *(parte 1)*.

Ewait_conv_ls:
```
==========
    wait until ((conv1line_done(1)=true)
       and (conv1line_done(2)=true) and
       (conv1line_done(3)=true) and
       (conv1line_done(4)=true) );
```

Esave_1line:
```
=========
    for j in 1 to (colunas-2) loop
       img_out(linha,j+colInicial1) <= line_out1(j);
       img_out(linha,j+colInicial2) <= line_out2(j);
       img_out(linha,j+colInicial3) <= line_out3(j);
       img_out(linha,j+colInicial4) <= line_out4(j);
    end loop;
```

Eclear_wr_f:
```
==========
    wr_finished <= false;
```

Ewait_convs:
```
=========
    wait until ((convol_done(1)=true)
       and (convol_done(2)=true)
       and (convol_done(3)=true)
       and (convol_done(4)=true) );
```

Ecalc_max:
```
========
    if(max(2) > max(1)) then
       tmp1 := max(2);
    else
       tmp1 := max(1);
    end if;
    if(max(4) > max(3)) then
       tmp2 := max(4);
    else
       tmp2 := max(3);
    end if;
    if(tmp2 > tmp1) then
       tmp1 := tmp2;
    end if;
    maximo := tmp1;
    max_global <= maximo;
```

Eset_wdone:
```
=========
    write_done <= true;
```

Eclear_written3l:
```
============
   written3l <= false;
```

Eset_wr_f:
```
========
    wr_finished <= true;
```

Ewait_/conv_ls:
```
============
    wait until ((conv1line_done(1)=false)
       and (conv1line_done(2)=false)
       and (conv1line_done(3)=false)
       and (conv1line_done(4)=false));
```

Einc_line:
```
========
    linha := linha+1;
```

Eclear_rdone:
```
==========
    read_done <= false;
```

Ecalc_min:
```
========
    if(min(2) < min(1)) then
       tmp1 := min(2);
    else
       tmp1 := min(1);
    end if;
    if(min(4) < min(3)) then
       tmp2 := min(4);
    else
       tmp2 := min(3);
    end if;
    if(tmp2 < tmp1) then
       tmp1 := tmp2;
    end if;
    minimo := tmp1;
    min_global <= minimo;
```

Enorm_img:
```
=========
    gama := maximo - minimo;
    fesc := (real(graylevels)-1.0)/real(gama);
    CICLO_N1: for cont in 0 to (ty-1) loop
       CICLO_N2: for i in 0 to (tx-1) loop
          fimg := img_out(cont,i) - minimo;
          fimg := integer(real(fimg) * fesc);
          img_out(cont,i) <= fimg;
       end loop CICLO_N2;
    end loop CICLO_N1;
```

Figura 8.11: Código do estado E_{rw_file} usado na implementação *hardware/software* da convolução *(parte 2)*.

```
Ewrite_img:
=========
    FILE_OPEN(out_fich,"emma_out_hex.raw",WRITE_MODE);
    hwrite(linha_tam,size_X,LEFT,4);
    hwrite(linha_tam,size_Y,LEFT,4);
    writeline(out_fich,linha_tam);
    CICLO_W1: for cont in 0 to (ty-1) loop
       CICLO_W2: for i in 0 to (tx-1) loop
          pixel := int2stdulogic(fimg);
          hwrite(linha_esc,pixel,LEFT,9);
       end loop CICLO_N2;
       writeline(out_fich,linha_esc);
    end loop CICLO_N1;
    FILE_CLOSE(out_fich);
```

Figura 8.12: Código do estado E_{rw_file} usado na implementação *hardware/software* da convolução *(parte 3)*.

```
TYPE imagem is array (0 to 197) of integer;
TYPE mask is array (0 to 8) of integer;
TYPE linha_out is array (1 to 64) of integer;
signal read_done, write_done: boolean;
signal wr_finished, written3l: boolean;
signal convol_done, accept3l: boolean;
constant mascara: mask := (-2,-2,0,-2,0,2,0,2,2);
signal lines_in: imagem;
signal line_out: linha_out;
signal conv1line_done: boolean;
variable colunas, line, a, b: integer;
constant tx, ty: integer := 256, 256;
constant maskX: integer := 3;
constant maskY: integer := 3;
```

```
Einic_vars_1:
==========
   convol_done <= false;
   conv1line_done <= false;
   accept3l <= false;
   max <= 1; min <= 0;
   b := maskY/2; a := maskX/2;

Einic_line:
========
   line := b;

Eset_accept3l:
==========
   accept3l <= true;

Ewait_written3l:
===========
   wait until written3l = true;

Ewait_wr_f:
=========
   wait until wr_finished = true;

Einc_line:
========
   line := line + 1;

Ewait_wdone:
==========
   wait until write_done = true;

Ewait_rdone:
=========
   wait until read_done = true;

Ecalc_column:
==========
   colunas := tx/4 + 2*a;

Ecmp_line:
========
   L_LINE: while line< (ty-b) loop
      [...]
   end loop L_LINE;

Eclear_accept3l:
===========
   accept3l <= false;

Eset_conv_l:
=========
   conv1line_done <= true;

Eclear_conv_l:
===========
   conv1line_done <= false;

Eset_conv:
========
   convol_done <= true;
```

Figura 8.13: Código dos estados $E_{pconvol1,2,3,4}$ utilizados na implementação *hardware/software* da convolução.

```
variable j, K , L, fimg: integer;
variable pos_img, pos_mask: integer;

Einic_col:
=======
   j := a;

Einc_col:
=======
   j := j + 1;

Einic_maskY:
==========
   K := 0;

Ecmp_maskY:
==========
   L_MY: while K < maskY loop
      [...]
   end loop L_MY;

Ecalc1pt_contrib:
============
   pos_img := (j-a+L)+ K*colunas;
   pos_mask := L+(K*maskX);
   fimg := fimg+lines_in(pos_img)*
           mascara(pos_mask);

Estore1pt_img:
===========
   line_out(j) <= fimg;

Emax:
====
   if (fimg>max) then
      max <= fimg;
   end if;

Ecmp_col:
========
   L_COL: while j < (colunas-a) loop
      [...]
   end loop L_COL;

Einic1pt_contrib:
============
   fimg := 0;

Einic_maskX:
==========
   L := 0;

Ecmp_maskX:
==========
   L_MX: while L < maskX loop
      [...]
   end loop L_MX;

Einc_maskX:
=========
   L := L + 1;

Einc_maskY:
=========
   K := K + 1;

Emin:
====
   if (fimg<min) then
      min <= fimg;
   end if;
```

Figura 8.14: Código do estado E_{calc_conv} utilizado na implementação *hardware/software* da convolução.

A partir do escalonamento do sistema e por combinação da secção de máquina de estados dos estados programa escalonados na mesma etapa de controlo obtém-se a máquina de estados que coordena o funcionamento das partições de *hardware*. Para os estados mais complexos atribuídos a *hardware* é recomendável analisar o escalonamento das suas operações, para ter a certeza que se obtém uma máquina de estados correctamente projectada. Foi o que se fez com o estado $E_{calc1pt\_contrib}$, cujo o escalonamento está esquematizado na figura 8.16. A figura 8.17 descreve a máquina de estados para a unidade de controlo duma partição de *hardware*.

8.2.5 Resultados Obtidos com a Implementação *Hardware/Software*

Além de estimativas, a implementação *hardware/software* da convolução permite obter medições para as métricas envolvidas no processo de partição: o desempenho e o espaço ocupado pelo caminho de dados e pela unidade de controlo das partições de *hardware*. Deste modo, é

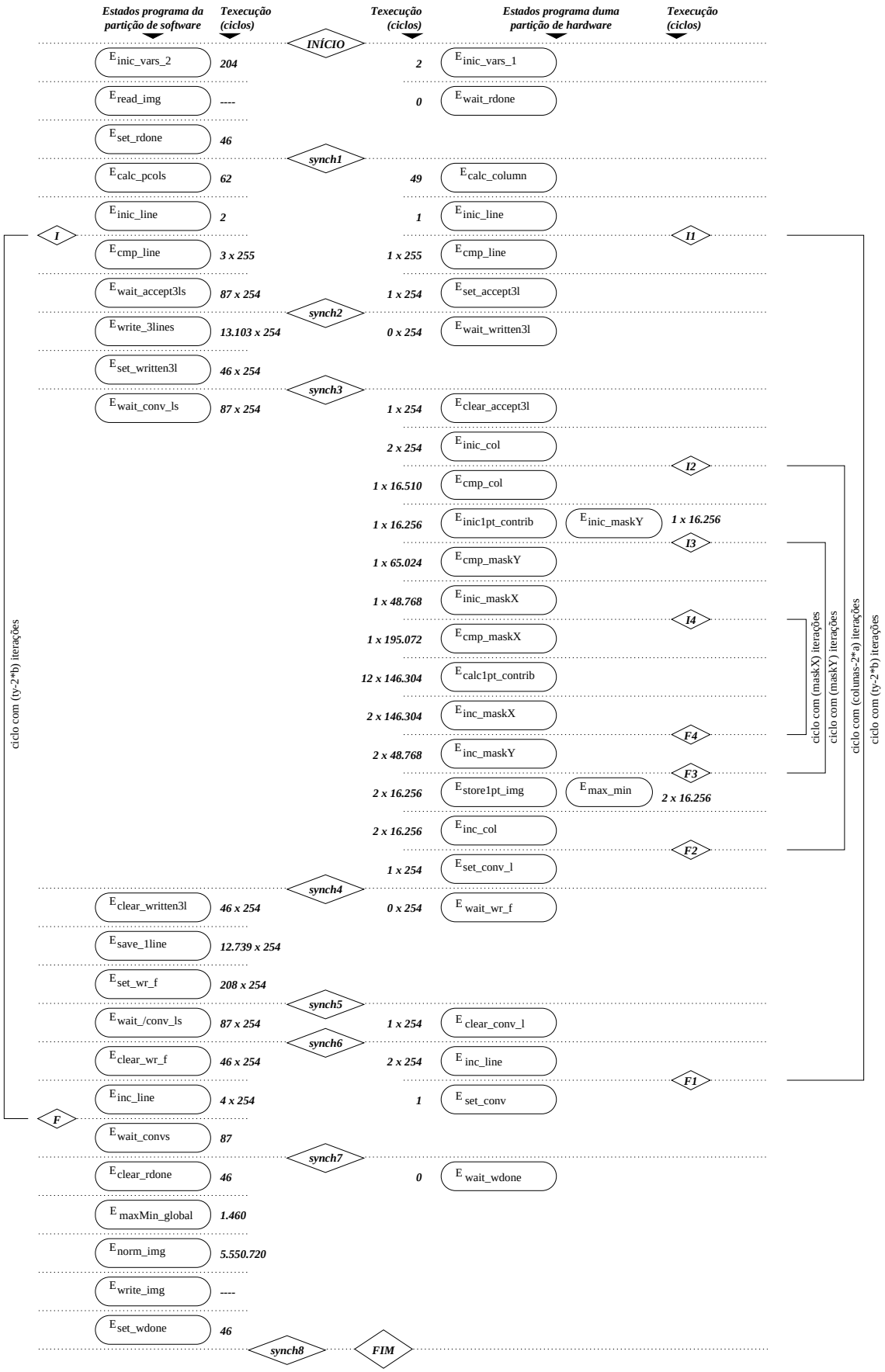


Figura 8.15: Escalonamento ao nível do sistema do grafo da convolução para a implementação *hardware/software*.

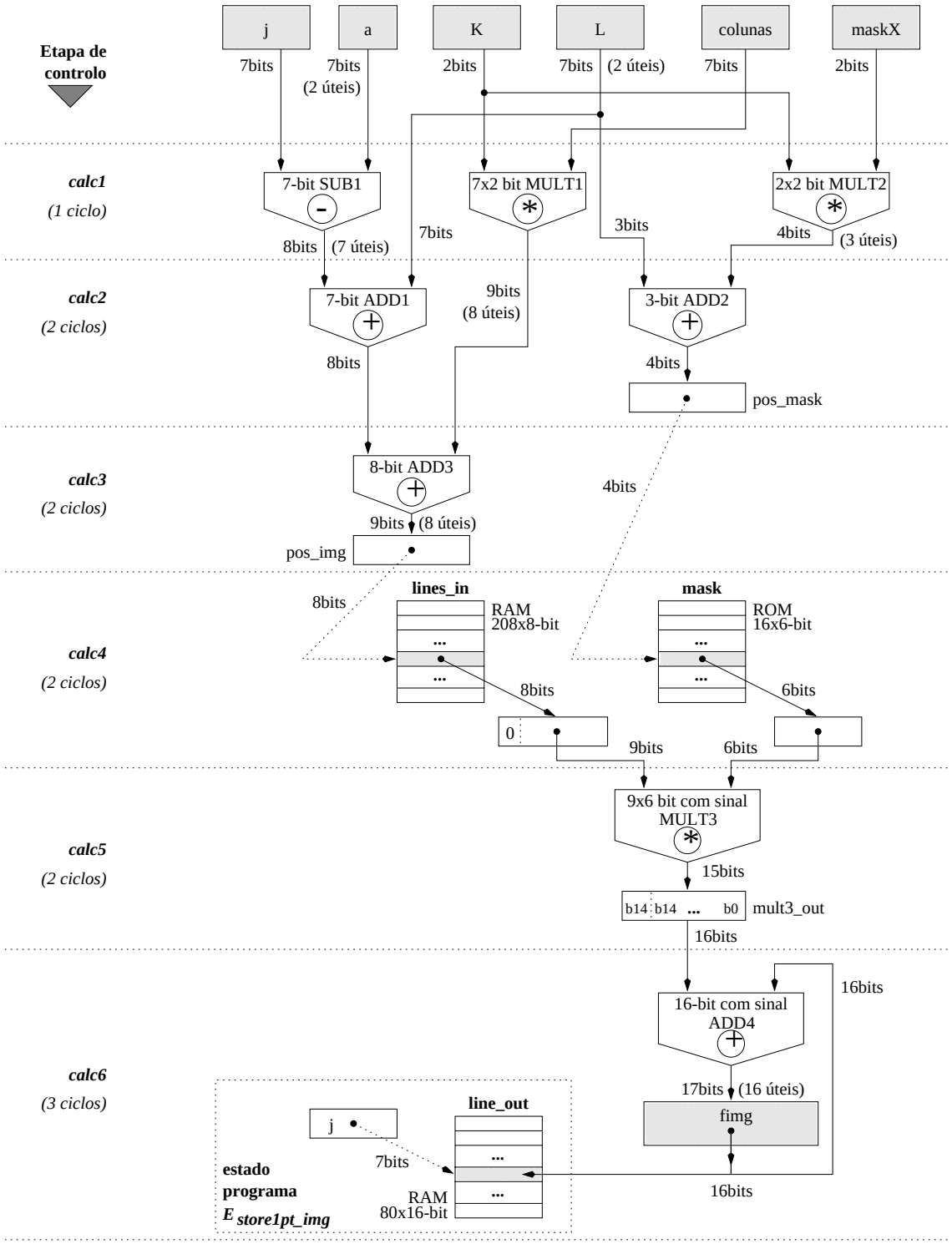


Figura 8.16: Escalonamento das operações do estado programa $E_{calc1pt\_contrib}$.

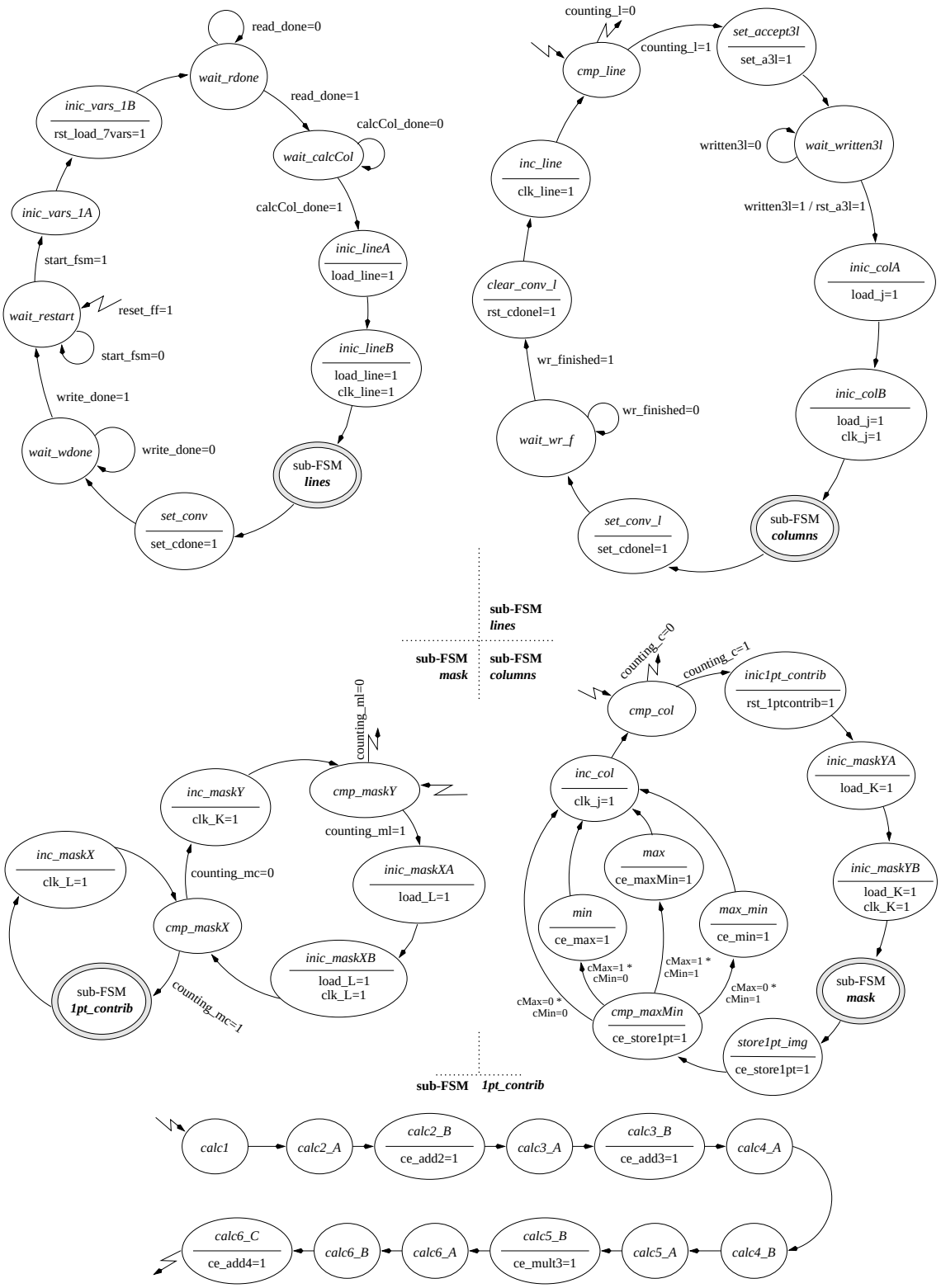


Figura 8.17: Máquina de estados para cada partição de *hardware* da convolução (correspondente ao estado programa $E_{pconvol1,2,3,4}$).

possível avaliar a precisão das estimativas geradas pela metodologia de partição.

Desempenho do sistema

Começa por apresentar-se a estimativa do desempenho da implementação *hardware/software* para duas frequências do relógio de *hardware* (33 e 50 MHz), dois tipos de código (optimizado e não optimizado) e duas alternativas de comunicação entre *hardware* e *software* (tabela 8.16). A forma mais simples de modelar a comunicação entre *hardware* e *software* é pressupor que as partições comunicam ponto a ponto; contudo na arquitectura EDgAR-2 pode estabelecer-se uma comunicação simultânea entre *software* e as quatro partições de *hardware*. A implementação da convolução utiliza comunicação simultânea entre *software* e as quatro partições de *hardware*. A tabela 8.16 mostra que a comunicação simultânea reduz o tempo de execução entre 35 e 41%, relativamente às implementações com comunicação ponto a ponto. Como seria de esperar, o aumento da frequência do relógio de *hardware* de 33 MHz para 50 MHz diminui o tempo de execução, mas não na percentagem do aumento de frequência, especialmente porque o desempenho da parte de *software* não acompanha a melhoria introduzida no *hardware*.

Tipo de estimativa	***Tempo de execução*** *(ms)*
Frequência de hardware = 33 MHz	
Com comunicação simultânea entre *software* e as 4 partições de *hardware* e código não optimizado	139
Com comunicação ponto a ponto entre *software* e *hardware* e código não optimizado	224
Com comunicação simultânea entre *software* e as 4 partições de *hardware* e código optimizado	118
Com comunicação ponto a ponto entre *software* e *hardware* e código optimizado	181
Frequência de hardware = 50 MHz	
Com comunicação simultânea entre *software* e as 4 partições de *hardware* e código não optimizado	122
Com comunicação ponto a ponto entre *software* e *hardware* e código não optimizado	207
Com comunicação simultânea entre *software* e as 4 partições de *hardware* e código optimizado	101
Com comunicação ponto a ponto entre *software* e *hardware* e código optimizado	164

Tabela 8.16: Estimativa do desempenho da implementação *hardware/software* optimizada da convolução, considerando que o *software* funciona com um relógio de 200 MHz.

Para calcular as estimativas da tabela 8.16 aplicou-se um factor de optimização relativamente ao tempo de execução em *software* (λ_T), com o valor de 0.57143. Este valor foi calculado a partir do desempenho obtido com a simulação da solução totalmente em *software*, utilizando código optimizado e não optimizado.

O desempenho da implementação de *software* e da implementação *hardware/software* optimizada encontra-se na tabela 8.17. Verifica-se que mesmo optimizando a implementação *hardware/software* o seu desempenho (143 ms) não é melhor que o da implementação em

software (140 ms), quando se utiliza código optimizado relativamente ao tempo de execução. Embora não testado, fica claro que a implementação *hardware/software* com um relógio de 50 MHz apresentaria um desempenho superior (cerca de 120 ms[6]) ao da solução puramente de *software* (140 ms).

Métrica	***Condição de medição***	***Estimativa*** *(ms)*	***Medição*** *(ms)*	***Precisão*** (%)
Tempo de execução da convolução em *software*	Código não optimizado, *Windows* 2000, P200	230	245	94
Tempo de execução da convolução em *software*	Código optimizado relativamente ao tempo de execução, *Windows* 2000, P200	131	140	94
Tempo de execução da convolução em *hw/sw*	Código não optimizado, *Windows* 2000, P200 e plataforma EDgAR-2 a 33 MHz	139	170	82
Tempo de execução da convolução em *hw/sw*	Código optimizado relativamente ao tempo de execução, *Windows* 2000, P200 e plataforma EDgAR-2 a 33 MHz	118	143	83

Tabela 8.17: Desempenho da convolução quando implementada em *software* ou em *hardware/software*.

Para melhorar o desempenho da implementação *hardware/software* poderia ainda recorrer-se às seguintes optimizações:

- alterar o *device driver* da plataforma EDgAR-2, de modo a suportar a comunicação por acesso directo à memória; estimou-se que quando a comunicação entre *hardware* e *software* se processa através do mecanismo de acesso directo à memória e o *hardware* funciona a 33 MHz, o tempo de execução da convolução diminui de 143 ms para 121 ms, o que representa uma redução de 15%;
- reduzir a duração de algumas etapas de controlo no escalonamento do estado programa $E_{calc1pt\_contrib}$ (figura 8.16), que é o responsável pela maior contribuição para o tempo de execução da convolução; a eliminação do primeiro ciclo, de entre os atribuídos a $E_{calc1pt\_contrib}$, equivale a retirar aproximadamente 5 ms (3,7%) aos 143 ms do tempo de execução da convolução.

A tabela 8.17 mostra ainda que a precisão das estimativas do desempenho do sistema varia entre 82 e 94%, sendo as estimativas de *software* mais precisas que as de *hardware*. Este facto deve-se muito a optimizações, introduzidas na implementação *hardware/software*, que não estão presentes no modelo de estimação. Para o exemplo estudado, a fidelidade das estimativas é de 83%, uma vez que em 5 dos 6 pares de soluções disponíveis, a diferença nas estimativas do desempenho segue a tendência que se verifica na diferença entre as medições. O valor da fidelidade das estimativas não é muito elevado, mas confere confiança à precisão

[6] O valor de 120 ms resulta de se considerar uma estimativa de 101 ms, incluída na tabela 8.16, e uma precisão média de 83%, obtida com as soluções *hardware/software* da tabela 8.17.

das estimativas. No entanto, há que ter em atenção que o número de amostras usado no cálculo da fidelidade das estimativas é baixo.

Embora a implementação *hardware/software* da convolução esteja optimizada em relação à solução obtida pelo algoritmo de partição é lícito estabelecer uma comparação entre as duas soluções. Na implementação *hardware/software*, a situação que mais se aproxima da solução obtida pelo algoritmo de partição é aquela em que o *hardware* funciona com um relógio de 50 MHz e utiliza comunicação ponto a ponto entre *software* e *hardware*. Nestas condições, a tabela 8.16 indica para estimativa do desempenho da implementação *hardware/software* o valor de 164 ms. Considerando este valor e uma precisão de 83%, que é o valor médio obtido nas estimativas do desempenho das implementações de *hardware/software* incluídas na tabela 8.17, pode extrapolar-se o valor de 198 ms para medição do desempenho.

Usando como medida do desempenho o valor de 198 ms e como estimativa o valor de 254 ms apresentado na tabela 8.9, a solução obtida pelo algoritmo de partição atinge 72% do desempenho conseguido pela implementação *hardware/software* optimizada. Ou seja, embora a solução obtida pelo algoritmo esteja longe de poder ser considerada óptima em absoluto, dadas as optimizações incluídas na implementação, é uma solução de partição com qualidade.

Espaço ocupado pelo caminho de dados

O valor medido e estimado para o espaço ocupado pelo caminho de dados de cada partição de *hardware* presente na implementação *hardware/software* da convolução é descrito nas tabelas 8.18 e 8.19. O valor estimado é 9377 portas lógicas equivalentes, enquanto para valor medido se considerou 9180. Ou seja, a precisão da estimativa é de 98%. Contudo, há que explicar o porquê de se escolher 9180 para valor medido, quando a tabela 8.18 contém três alternativas:

- a primeira alternativa que se pode usar para medida do espaço é o número de portas lógicas equivalentes indicado pelas ferramentas de síntese do fabricante das FPGAs presentes na arquitectura alvo, ou seja, 15182;
- a segunda alternativa consiste em utilizar uma medida indirecta do espaço, obtida multiplicando o número de CLBs usados na implementação pelo número típico de portas lógicas equivalentes por CLB [Xil97], ou seja, 9180;
- na terceira alternativa a medida do espaço resulta duma soma pesada do número de *flip-flops*, *LUTs* de 4 entradas e *LUTs* de 3 entradas usados na implementação, em que o peso dos termos é o valor máximo de portas lógicas equivalentes por *flip-flop*, *LUT* de 4 entradas e *LUT* de 3 entradas [Xil97], ou seja, 7290.

Métrica	***Medição*** ***p = HW1,2,3,4***		
CLBs	322	(em 576)	56%
Flip-flops de CLBs	131	(em 1152)	11%
LUTs de 4 entradas	528	(em 1152)	46%
LUTs de 3 entradas	161	(em 576)	28%
IOBs	29	(em 160)	18%
Buffers tri-state	51	(em 1248)	4%

Portas lógicas equivalentes indicadas pelas ferramentas do fabricante das FPGAs	15182
Portas lógicas equivalentes calculadas com base no número de CLBs usados e no número típico de portas lógicas equivalentes por CLB (28.5)	9180
Portas lógicas equivalentes calculadas com base no número de FFs, de LUTs de 3 e de 4 entradas e nos valores máximos de portas lógicas equivalentes por FF (12) ou por LUT (6 e 9)	7290

Tabela 8.18: Valores medidos, para cada partição de *hardware* da implementação *hardware/software*, das métricas relacionadas com o espaço ocupado pelo caminho de dados das partições de *hardware*.

Métrica	***Estimativa*** ***p = HW1,2,3,4***	***Medição*** ***p = HW1,2,3,4***
$areaEPrograma(CD(p))$	3596	–
$areaVars(CD(p))$	4375	–
$areaInterface(CD(p))$	899	–
$areaMuxesExtra(CD(p))$	507	–
$\boldsymbol{areaHW(CD(p))}$	***9377***	***9180***

Tabela 8.19: Estimativa, para cada partição de *hardware* da implementação *hardware/software*, das métricas envolvidas no cálculo do espaço ocupado pelo caminho de dados dessas partições.

A primeira alternativa não é considerada porque o espaço por ela definido inclui todos os recursos utilizados na implementação (CLBs, IOBs, *buffers tri-state*, entre outros), enquanto na estimação do espaço apenas se consideram elementos de lógica maioritariamente implementados com os CLBs, tais como as unidades funcionais, os registos e os multiplexadores. Deste modo, o valor indicado pelas ferramentas de síntese (15182) é bastante superior à lógica quantificada nas estimativas obtidas com a metodologia de estimação desenvolvida (9377).

A segunda e terceira alternativas são hipóteses válidas para definir uma medição de espaço passível de ser comparada com as estimativas obtidas. A terceira alternativa constitui uma medida por defeito ao considerar os 3 tipos de recursos mais importantes dos CLBs, mas deixa de fora alguma da lógica utilizada destas células. Como tem por base o número típico de portas lógicas equivalentes por CLB, a segunda alternativa constitui assim uma boa solução de compromisso. O valor obtido com a segunda alternativa (9180) é o que se aproxima mais do valor estimado (9377). O erro das três alternativas, da primeira até à terceira, é 38%, 2% e 29%.

Espaço ocupado pela unidade de controlo

A medida para o espaço ocupado pela unidade de controlo de cada partição de *hardware* ne-

cessária à implementação da convolução é obtida de forma indirecta a partir de duas métricas fornecidas pelas ferramentas de síntese: (i) o número de macro-células e (ii) o número de produtos[7] do CPLD usados (tabela 8.20). Nenhuma das métricas garante medidas precisas para o espaço ocupado no CPLD. Por um lado, o número de macro-células tende a fornecer medidas por excesso e por outro, o número de produtos gera medidas por defeito. As medidas são por excesso porque a lógica das macro-células pode ser parcialmente usada e é totalmente considerada e são por defeito porque as implementações podem envolver lógica, além dos produtos, que não é contabilizada.

Métrica	***Medição*** p = ***HW1,2,3,4***	
Macro-células	34 (em 108) 31%	
Produtos	107 (em 540) 20%	
Portas lógicas equivalentes calculadas com base no número de macro-células usadas		744
Portas lógicas equivalentes calculadas com base no número de produtos usados		480

Tabela 8.20: Valores medidos, para cada partição de *hardware* incluída na implementação *hardware/software* da convolução, das métricas relacionadas com o espaço ocupado pela unidade de controlo das partições de *hardware*.

A estimativa do espaço ocupado pela unidade de controlo das partições de *hardware*, calculada a partir das métricas elementares incluídas nas tabelas 8.21 e 8.22, é sumariada na tabela 8.23. A tabela 8.23 permite ainda comparar a estimativa com o valor medido para as métricas espaço da unidade de controlo, número de estados $nE(p)$, número de sinais de controlo gerados $nSC(p)$ e respectivo número de activações $nASC(p)$, número de sinais de estado lidos $nSE(p)$ e número de leituras $nLSE(p)$ associado. Constata-se que a estimativa e o valor medido para as métricas $nE(p)$, $nSE(p)$ e $nLSE(p)$ são praticamente coincidentes, enquanto a estimativa das métricas $nSC(p)$ e $nASC(p)$ apresenta um erro significativo: (i) estimou-se que o valor de $nSC(p)$ seria 35 sinais e o valor observado foi 22 sinais e (ii) a estimativa para $nASC(p)$ foi 44 activações mas o valor medido limitou-se a 34 activações. A principal explicação para a imprecisão verificada nas duas estimativas reside na estratégia que se empregou na contabilização dos sinais de selecção dos multiplexadores de entrada nas variáveis. Por questões de simplificação, a estimativa automática do número de entradas dum multiplexador é calculada a partir do número de operações que escrevem valores distintos na variável em questão. Acontece que em determinadas situações a implementação exige uma entrada a menos do que o previsto em cada multiplexador, dado que uma das operações de escrita é efectuada pelo mecanismo de *set* ou *reset* dos *flip-flops*. Embora o impacto desta imprecisão sobre a estimativa do espaço da unidade de controlo não seja dramático, é um aspecto a rever no futuro.

[7] *Product terms*, na terminologia inglesa.

Variável	***HW1***	***HW2***	***HW3***	***HW4***	***Variável***	***HW1***	***HW2***	***HW3***	***HW4***
$line_1$	1				max_1	1			
$line_2$		1			max_2		1		
$line_3$			1		max_3			1	
$line_4$				1	max_4				1
min_1	1				j_1	1			
min_2		1			j_2		1		
min_3			1		j_3			1	
min_4				1	j_4				1
L_1	1				$fimg_1$	1			
L_2		1			$fimg_2$		1		
L_3			1		$fimg_3$			1	
L_4				1	$fimg_4$				1
K_1	1				$convol\_done_1$	1			
K_2		1			$convol\_done_2$		1		
K_3			1		$convol\_done_3$			1	
K_4				1	$convol\_done_4$				1
$conv1line\_done_1$	2				$accept3l_1$	2			
$conv1line\_done_2$		2			$accept3l_2$		2		
$conv1line\_done_3$			2		$accept3l_3$			2	
$conv1line\_done_4$				2	$accept3l_4$				2
Total de sinais de selecção	***12***	***12***	***12***	***12***					

Tabela 8.21: Implementação *hardware/software* da convolução: estimativa dos sinais de selecção do multiplexador a colocar na entrada de cada variável escrita pelos estados programa atribuídos às partições de *hardware*.

Comparando o valor 524 estimado para o espaço da unidade de controlo com as medidas indirectas 744 e 480 incluídas na tabela 8.20, verifica-se uma precisão de 70% ou 91% na estimativa.

Correcção da solução

Tendo aplicado o filtro de Sobel a várias imagens de 256x256 pontos verifica-se um funcionamento correcto por parte da implementação *hardware/software* do algoritmo de convolução. Como curiosidade, o funcionamento da implementação é ilustrado através das imagens, original e filtrada, da figura 8.18.

Além da validar a maioria das opções adoptadas na metodologia de partição e estimação, o exemplo descrito mostrou que embora a arquitectura alvo possua recursos limitados para armazenar informação em *hardware*, permite obter resultados interessantes na implementação dum sistema predominantemente de dados. Provou-se que uma boa solução de partição consegue atenuar o impacto negativo do tempo de comunicação entre *hardware* e *software* sobre o tempo de execução global, mas que uma plataforma como o EDgAR-2 dificilmente permite implementações de *hardware/software* muito eficientes para sistemas que manipulem muitos dados.

Estado programa	***nE***	***nSC***	***nASC***	***nSE***	***nLSE***
$Einic\_vars\_1_i$	2	1	1	0	0
$Ewait\_rdone_i$	1	0	0	1	1
$Ecalc\_column_i$	1 (†)	0	0	1	1
$Einic\_line_i + Ecmp\_line_i + Einc\_line_i$	4	2	4	1	1
$Eset\_accept3l_i$	1	1	1	0	0
$Ewait\_written3l_i$	1	0	0	1	1
$Eclear\_accept3_i$	1	1	1	0	0
$Einic\_col_i + Ecmp\_col_i + inc\_col_i$	4	2	4	1	1
$inic1pt\_contrib_i$	1	1	1	0	0
$Einic\_maskY_i + Ecmp\_maskY_i + Einc\_maskY_i$	4	2	4	1	1
$Einic\_maskX_i + Ecmp\_maskX_i + Einc\_maskX_i$	4	2	4	1	1
$Ecalc1pt\_contrib_i$	12	4	4	0	0
$Estore1pt\_img_i$	1	1	2	0	0
$Emin_i + Emax_i$	4	3	3	2	2
$Eset\_conv\_l_i$	1	1	1	0	0
$Ewait\_wr\_f_i$	1	0	0	1	1
$Eclear\_conv\_l_i$	1	1	1	0	0
$Eset\_conv_i$	1	1	1	0	0
$Ewait\_wdone_i$	1	0	0	1	1

(†) Aplicou-se uma optimização para reduzir o número de estados de 49 para 1.

Tabela 8.22: Estimativa, para os estados programa da descrição utilizada na implementação *hardware/software* da convolução, das métricas envolvidas no cálculo do espaço ocupado pela unidade de controlo das partições de *hardware*.

	Estimativa	***Medição***
Métrica	*p = HW1,2,3,4*	*p = HW1,2,3,4*
$\sum_{o \in p} nE(o)$	46	–
$\sum_{o \in p} nSC(o)$	23	–
$\sum_{o \in p} nASC(o)$	32	–
$\sum_{o \in p} nSE(o)$	11	–
$\sum_{o \in p} nLSE(o)$	11	–
$nEpart(p)$	0	–
$nSCpart(p)$	0	–
$nASCpart(p)$	0	–
$nSEpart(p)$	0	–
$nLSEpart(p)$	0	–
$nSCmux(p)$, $nASCmux(p)$	12	–
$nE(p)$	46	45
$nSC(p)$	35	22
$nASC(p)$	44	34
$nSE(p)$	11	10
$nLSE(p)$	11	10
areaRegEstado(p)	276	–
areaLogicaCtl(p)	27	–
areaLProxEstado(p)	71	–
areaHWfsm(UC(p))	374	–
areaHW(UC(p))	***524***	***480***

Tabela 8.23: Estimativa, para cada partição de *hardware* incluída na implementação *hardware/software* da convolução, das métricas envolvidas no cálculo do espaço ocupado pela unidade de controlo dessas partições.

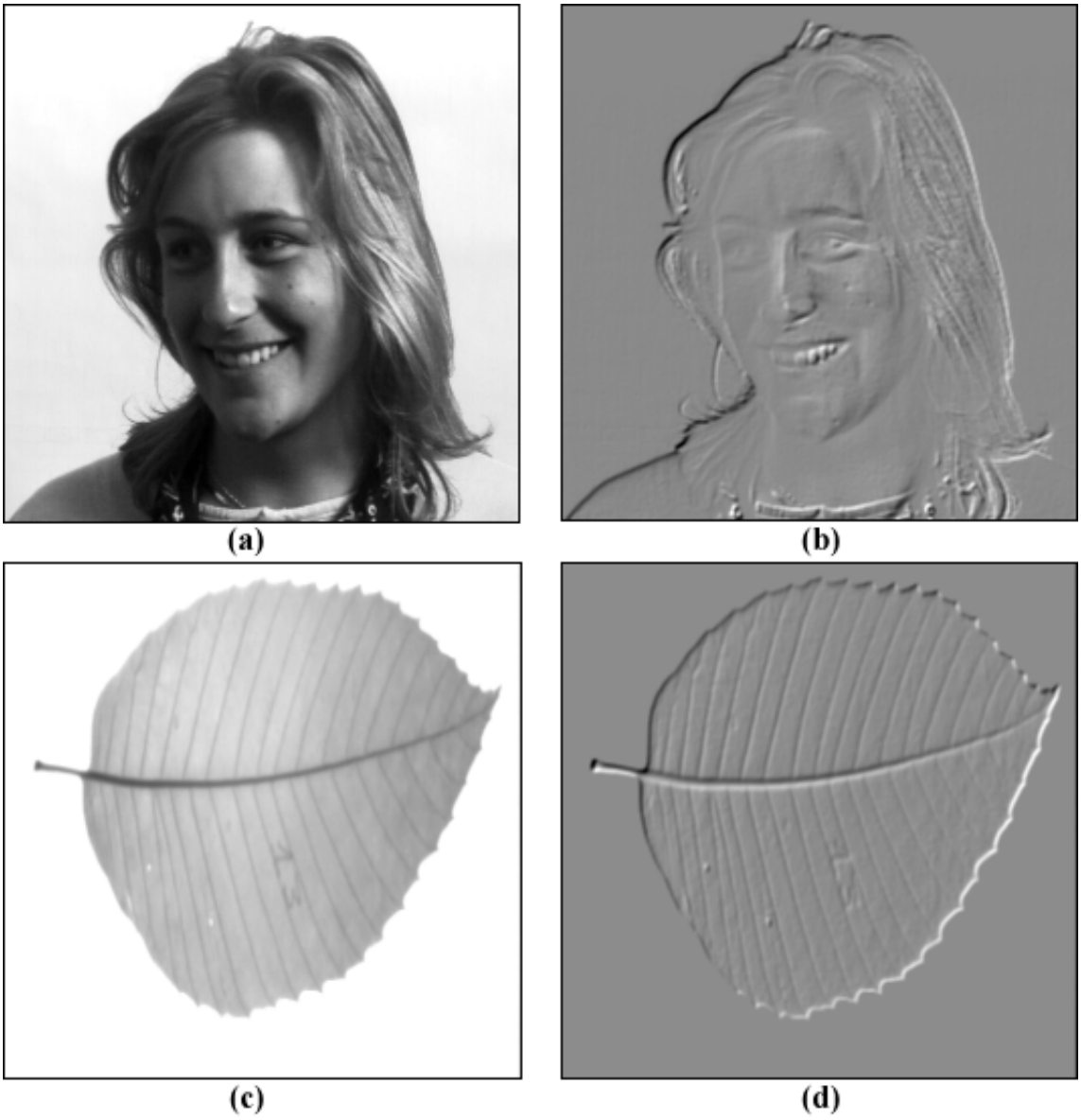


Figura 8.18: Resultado da convolução duma imagem com um filtro de Sobel: (a) imagem original `emma`, (b) imagem `emma` filtrada, (c) imagem original `folha`, (d) imagem `folha` filtrada.

8.3 Caso de Estudo 2: Sistema de Criptografia DES

Para seleccionar o segundo caso de estudo estabeleceram-se os seguintes critérios: (i) o sistema deve estar aparentemente vocacionado para uma implementação em *hardware*, complementando a orientação para *software* do exemplo da convolução, (ii) a dimensão do sistema deve ser da ordem de grandeza dos recursos de *hardware* disponíveis na arquitectura alvo e (iii) como critério secundário, dá-se preferência a um sistema com interface externa simples, de modo a facilitar a validação do seu correcto funcionamento.

A selecção dum sistema embebido predominantemente de dados, exigindo grande capacidade de armazenamento de informação, estava fora de questão dado que estas características estão

presentes no primeiro caso de estudo. Optou-se assim por um sistema embebido predominantemente de dados, que não exigisse grande capacidade de armazenamento de informação. Os sistemas de criptografia encaixam nesta classe de sistemas. Dos resultados apresentados em [RH99] e [EYCP00], relativos à implementação em FPGAs dos algoritmos candidatos ao *advanced encryption standard* (AES), concluiu-se que estes não se enquadravam nos critérios de selecção do caso de estudo: não estão vocacionados para uma implementação de *hardware* e exigem uma quantidade de lógica muito superior à disponível nas FPGAs da plataforma EDgAR-2. A escolha do caso de estudo recaiu assim sobre o algoritmo DES, a norma actual de cifragem. O algoritmo DES possui os três requisitos atrás enumerados: está vocacionado para uma implementação de *hardware*, o espaço exigido numa implementação de *hardware* é apenas o dobro da lógica disponível no EDgAR-2 e a interface com o exterior é simples.

8.3.1 Descrição do Caso de Estudo

O algoritmo DES que se vai implementar aplica um conjunto de transformações aos dados de entrada (amostra), dependentes destes dados e duma chave secreta que, também ela vai sendo alterada ao longo das várias iterações do processo de cifragem ou decifragem. A amostra e a chave, antes e após o processamento, possuem 64 bits, sendo que 8 bits da chave secreta são usados como bits de paridade para detecção de erros. Os bits de paridade estão localizados no bit mais significativo de cada um dos 8 bytes que compõem a chave. O processo de decifrar é o inverso do processo de cifrar, usando-se a mesma chave secreta de entrada e aplicando-se praticamente as mesmas operações. A única diferença, como se explicará mais à frente, reside no tamanho e no tipo dos deslocamentos que se aplicam à chave secreta: deslocamentos à esquerda no processo de cifragem e deslocamentos à direita no processo de decifragem [DK99] [KS98].

Uma amostra a (de)cifrar passa por uma permutação inicial (IP), por um conjunto de transformações dependentes da chave secreta e finalmente por uma permutação final (FP ou IP^{-1}), que é inversa de IP (figura 8.19). O conjunto de transformações dependentes da chave secreta é definido através duma **função de cifragem** f e duma função de **escalonamento da chave** KS. A função f é composta pela expansão E, as tabelas de substituição *S-box* e a permutação P. A informação resultante da permutação inicial IP é dividida em duas metades de 32 bits: a parte menos significativa (R) alimenta a função f e a parte mais significativa (L) é aplicada na entrada dum operador OU-exclusivo[8]. Após uma passagem[9], as duas metades da amostra a (de)cifrar são trocadas entre si e a passagem repete-se. O algoritmo evolui em 16 passagens de modo a efectuar a "circulação" da amostra a (de)cifrar. Após as 16 passagens, cada bit da amostra (de)cifrada depende de todos os bits da amostra inicial e de todos os bits da chave

[8] Uma operação OU-exclusivo equivale a uma soma módulo 2 bit a bit.

[9] Uma iteração do algoritmo DES é designada em inglês por ***round*** e neste documento por **passagem**.

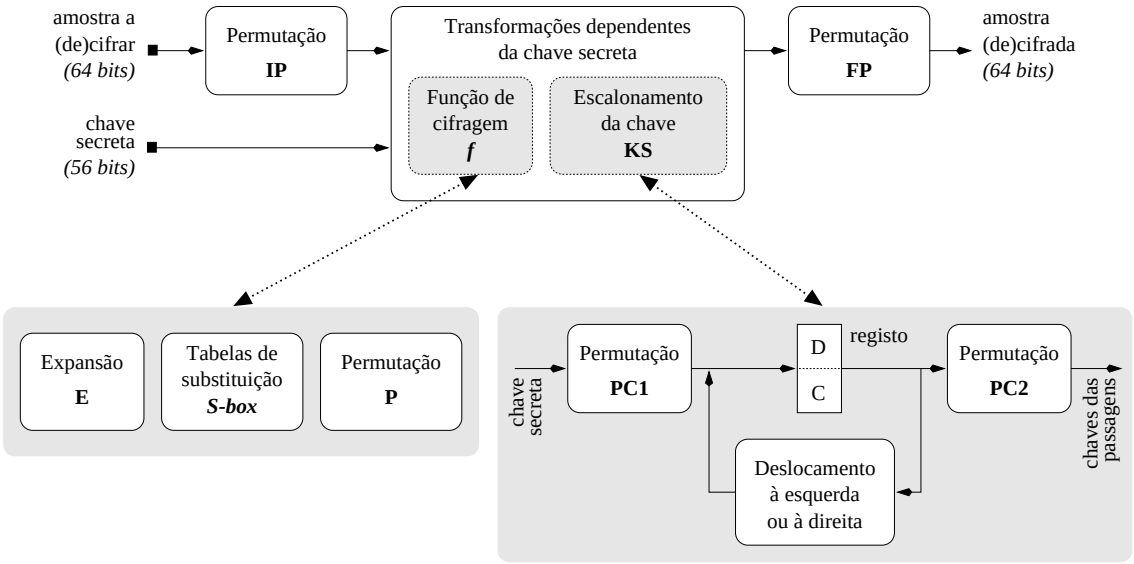


Figura 8.19: Diagrama de blocos do sistema de criptografia DES.

secreta, não existindo correlação entre a amostra produzida e a de entrada.

Apresenta-se a seguir uma breve descrição dos principais componentes do algoritmo DES.

Expansão *E*

O operador E efectua a expansão duma entrada de 32 bits para 48 bits, sendo que o valor expandido é depois aplicado numa soma módulo 2 com a chave da passagem actual.

Tabelas de substituição *S-box*

As tabelas de substituição *S-box* constituem uma transformação não linear, o que faz com que o algoritmo DES seja não linear. O algoritmo utiliza 8 tabelas, cada uma com 64 valores organizados em forma de matriz com 4 linhas e 16 colunas. As tabelas guardam valores de 4 bits e são indexadas por uma posição com 6 bits ($2^6 = 64$): 2 bits identificam a linha e 4 bits a coluna a seleccionar na matriz. Cada linha de 16 posições contém permutações dum número entre 0 e 15, não existindo por isso números repetidos nessa linha. A tabela coloca na saída o valor da posição seleccionada pelos 6 bits aplicados na sua entrada. Com os 8 valores colocados em simultâneo na saída pelas 8 tabelas forma-se um padrão de 32 bits.

Permutações

As permutações IP, FP e P constituem um operador linear definido por uma matriz 8x8, nos dois primeiros casos, ou por uma matriz 8x4, no último caso. O operador funciona da seguinte forma: para qualquer posição (x, y) da matriz, se o conteúdo dessa posição for o valor $v(x, y)$, o bit a colocar na posição (x, y) da saída é o bit que estiver na posição $v(x, y)$ da entrada. A finalidade da permutação P é misturar os bits da amostra, de modo a evitar que as tabelas de substituição reconstituam os bits iniciais.

Escalonamento da chave *KS*

O escalonamento da chave KS gera uma chave de 48 bits distinta para cada uma das 16 passagens do algoritmo DES, misturando de forma linear os 56 bits da chave secreta. Os componentes envolvidos no escalonamento da chave são ilustrados na figura 8.19: a permutação $PC1$, um registo, a permutação $PC2$ e o operador de deslocamento de bits à esquerda ou à direita, consoante o processo é de cifragem ou de decifragem. O escalonamento KS, esquematizado na parte direita da figura 8.20, evolui de acordo com os seguintes passos:

(i) a chave secreta é sujeita à permutação $PC1$, que reorganiza os bits da chave;

(ii) a chave permutada é dividida em duas metades C_i e D_i (com $i = 0$);

(iii) no processo de cifragem (decifragem), as duas metades C_i e D_i são deslocadas à esquerda (direita) numa quantidade pré-definida, gerando C_{i+1} e D_{i+1}; a quantidade pré-definida para o deslocamento a aplicar nas 16 passagens é:

passagem	▶	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
deslocamentos à esquerda	▶	1	1	2	2	2	2	2	2	1	2	2	2	2	2	2	1
deslocamentos à direita	▶	0	1	2	2	2	2	2	2	1	2	2	2	2	2	2	1

(iv) os valores C_{i+1} e D_{i+1} são sujeitos a uma segunda permutação $PC2$, que reorganiza novamente os bits da chave e produz a chave K_{i+1} da passagem $i + 1$ (figura 8.20);

(v) para obter a chave K_{i+2} da passagem seguinte, C_{i+1} e D_{i+1} são deslocadas à esquerda (direita) da quantidade pré-definida para essa passagem, obtendo-se C_{i+2} e D_{i+2}; os valores C_{i+2} e D_{i+2} são sujeitos à permutação $PC2$, produzindo-se a chave K_{i+2} da passagem $i + 2$;

(vi) o procedimento descrito em (v) é repetido para gerar as restantes chaves, ou seja, até $i + 2$ atingir o valor 16.

Combinando o diagrama de blocos da figura 8.19 com a informação sobre os vários componentes que a compõem, atinge-se a arquitectura do algoritmo DES ilustrada na figura 8.20.

8.3.2 Solução de Partição

A descrição puramente comportamental do sistema de criptografia, aplicada no processo de partição automática, é apresentada parcialmente no modelo PSMfg da figura 8.21. A descrição inclui paralelismo entre as operações relativas ao processamento inicial, final e às 16 passagens do algoritmo DES. Deste modo, favorece-se a exploração de paralelismo nas soluções obtidas

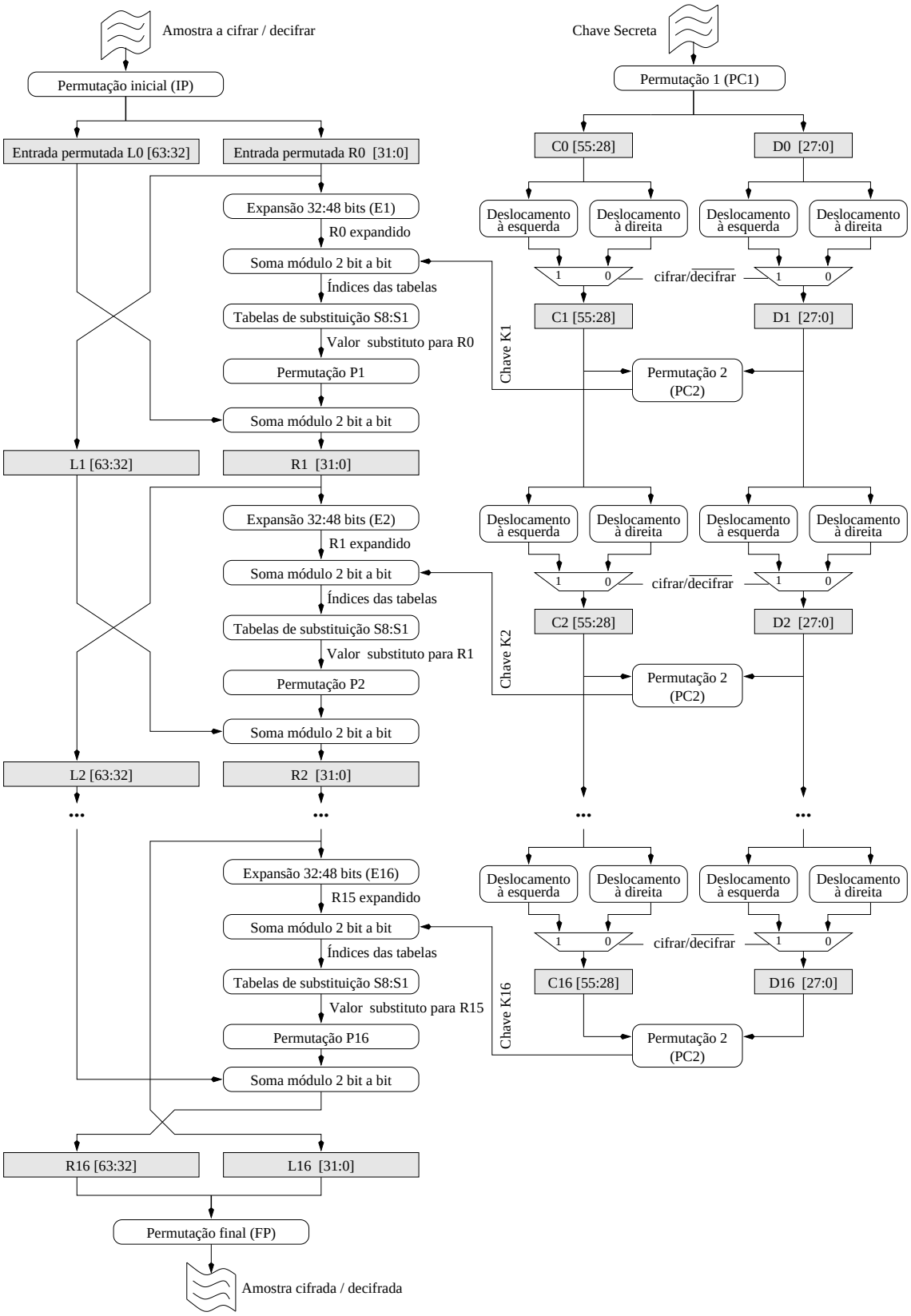


Figura 8.20: Arquitectura do sistema de criptografia DES.

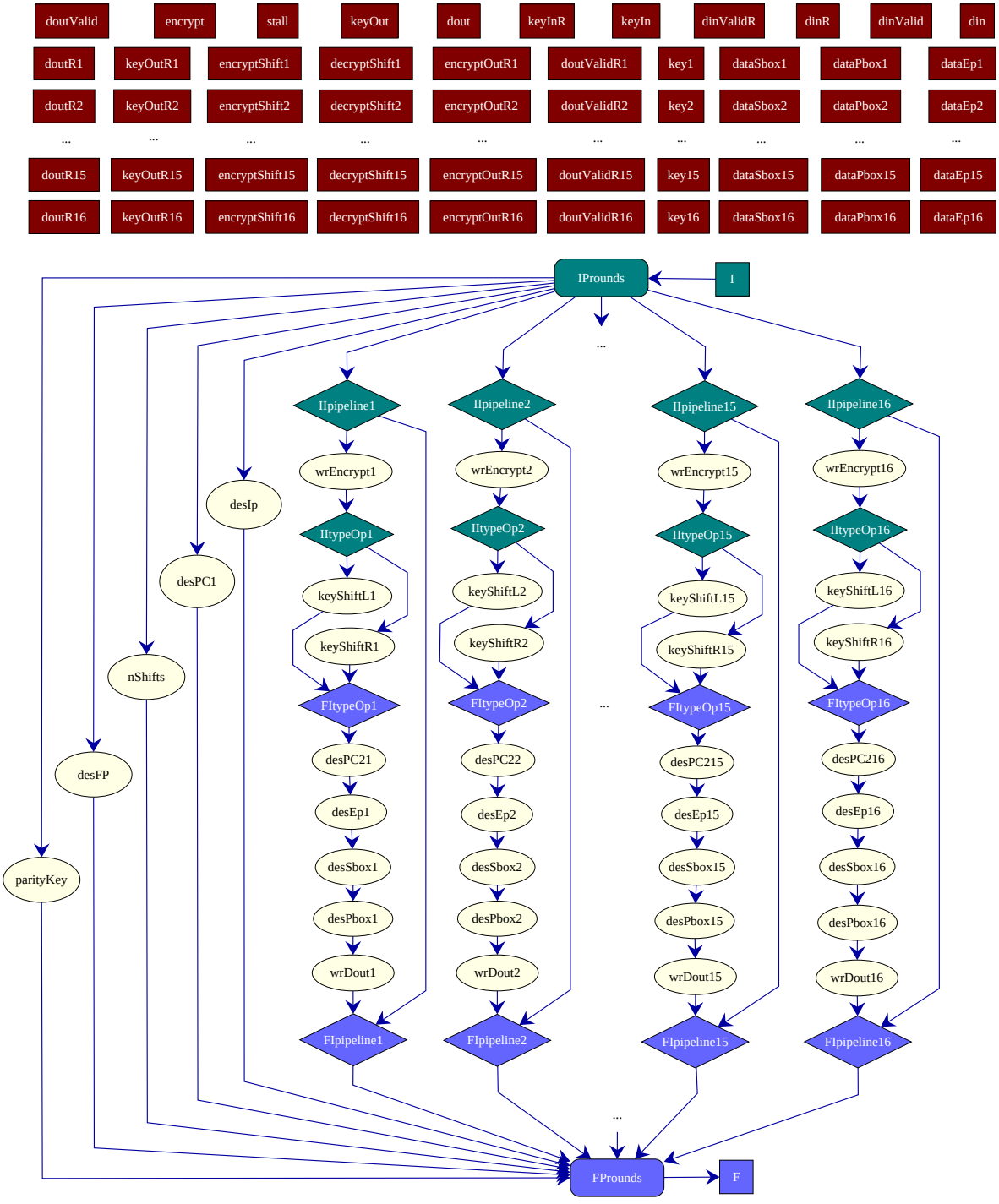


Figura 8.21: Descrição, com o meta-modelo PSMfg, aplicada no processo de partição do sistema de criptografia DES.

pelo algoritmo de partição. O modelo PSMfg[10] concretiza a arquitectura do algoritmo DES esquematizada na figura 8.20. No início de cada passagem adicionou-se uma condição que verifica se foi pedida a paragem da cadeia de *pipeline* definida pelas 16 passagens do algoritmo. No final duma passagem regista-se o resultado parcial do processamento das amostras, o qual alimenta a passagem seguinte do algoritmo.

As soluções de partição baseiam-se nas métricas estimadas ao nível do estado programa. As métricas associadas aos estados programa são (i) o espaço ocupado pelas unidades funcionais ($areaUF$), o tempo de computação em *hardware* ($TexecHW$) e em *software* ($TexecSW$) definidos na tabela 8.24 e (ii) as variáveis lidas ($rdVAR$), escritas ($wrVAR$) e que precisam dum multiplexador na entrada ($muxes$) contidas na tabela 8.25. Para as variáveis estima-se apenas o espaço ocupado em *hardware*, designado por $areaHWvar$ na tabela 8.26.

Estado programa	***Tipo de recurso de H/W***	***areaUF*** *(portas lógicas)*	***TexecSW*** *(ciclos do μP)*	***TexecHW*** *(ciclos do h/w)*
$desIp$	–	0	512	0
$desPc1$	–	0	444	0
$nShifts$	–	0	0	0
$desFp$	–	0	551	0
$parityKey$	–	0	84	0
$IIpipeline_i$	–	0	3	0
$FIpipeline_i$	–	0	0	0
$wrEncrypt_i$	–	0	0	0
$IItypeOp_i$	–	0	3	0
$FItypeOp_i$	–	0	0	0
$keyShiftL_i$	–	0	24 (†) , 36 (‡)	0
$keyShiftR_i$	–	0	24 (†) , 36 (‡)	0
$desPc2_i$	–	0	362	0
$desEp_i$	48 x OU-exclusivo	240	376	1
$desSbox_i$	8 x ROM64X4 bits	2504	156 (§)	0
$desPbox_i$	32 x OU-exclusivo	160	254	1
$wrDout_i$	–	0	2	0

(†) $i \in \{1, 2, 9, 16\}$ (‡) $i \ni \{1, 2, 9, 16\}$ e $(1 \leqslant i \leqslant 16)$ (§) valor médio

Tabela 8.24: Estimativa para as métricas associadas aos estados programa da descrição do sistema de criptografia DES.

A atribuição de estados programa e variáveis às partições, para a melhor solução de partição gerada pelo algoritmo de pesquisa tabu, encontra-se na tabela 8.27. Para obter esta solução utilizaram-se os seguintes parâmetros com o algoritmo de pesquisa tabu:

- o número de iterações efectuado foi 74400; este valor obtém-se com a equação 6.11, em que a constante $K2ni$ é 10, o número de partições $nPart$ é 5 e o número de objectos $nObj$ é 372;
- o limite imposto ao número de iterações sucessivas sem que haja melhoria da solução de partição foi 400;

[10] Em rigor deveria dizer-se "modelo PSM".

Estado programa	***Variáveis lidas***	***Variáveis escritas***	***Variáveis com multiplexador***
$desIp$	din, $dinValid$	$dinR$, $dinValidR$	–
$desPc1$	$keyIn$, $dinValid$	$keyInR$	–
$nShifts$	–	$encryptShift_{1..16}$, $decryptShift_{1..16}$	–
$desFp$	$doutR_{16}$	$dout$	–
$parityKey$	$keyOutR_{16}$	$keyOut$	–
$IIpipeline_i$	$stall$	–	–
$FIpipeline_i$	–	–	–
$wrEncrypt_i$	$encrypt$ (†), $encryptOutR_{i-1}$ (‡)	$encryptOutR_i$	–
$IItypeOp_i$	$encrypt$ (†), $encryptOutR_{i-1}$ (‡)	–	–
$FItypeOp_i$	–	–	–
$keyShiftL_i$	$encryptShift_i$, $keyInR$ (†), $keyOutR_{i-1}$ (‡)	$keyOutR_i$	–
$keyShiftR_i$	$decryptShift_i$, $keyInR$ (†), $keyOutR_{i-1}$ (‡)	$keyOutR_i$	–
$desPc2_i$	$keyOutR_i$	key_i	–
$desEp_i$	key_i, $dinR$ (†), $doutR_{i-1}$ (‡)	$dataEp_i$	–
$desSbox_i$	$dataEp_i$	$dataSbox_i$	–
$desPbox_i$	$dataSbox_i$, $dinR$ (†), $doutR_{i-1}$ (‡)	$dataPbox_i$	–
$wrDout_i$	$dataPbox_i$, $doutR_{i-1}$ (‡), $dinR$ (†), $dinValidR$ (†), $doutValidR_{i-1}$ (‡)	$doutR_i$ $doutValidR_i$ (§), $doutValid$ (*)	–

(†) $i = 1$ (‡) $2 \leqslant i \leqslant 16$ (§) $1 \leqslant i \leqslant 15$ (*) $i = 16$

Tabela 8.25: Estimativa, para cada estado programa da descrição do sistema de criptografia DES, das variáveis lidas, escritas e que precisam dum multiplexador na entrada.

Variável	***Largura*** *(bits)*	***Número de elementos***	***areaHWvar*** *(portas) (lógicas)*	***Variável***	***Largura*** *(bits)*	***Número de elementos***	***areaHWvar*** *(portas) (lógicas)*
din	64	1	384	$dinValidR$	1	1	6
$dinValid$	1	1	6	$doutValid$	1	1	6
$dinR$	64	1	384	$doutR_i$ (†)	64	1	384
$encrypt$	1	1	6	$encryptShift_i$	5	1	30
$keyIn$	64	1	384	$decryptShift_i$	5	1	30
$keyInR$	56	1	336	$encryptOutR_i$	1	1	6
$dout$	64	1	384	$doutValidR_i$	1	1	6
$keyOut$	64	1	384	$keyOutR_i$	56	1	336
$stall$	1	1	6	$dataSbox_i$ (‡)	32	1	0
key_i	48	1	0	$dataPbox_i$	32	1	0
$dataEp_i$	48	1	0				

(†) Em qualquer variável cujo nome inclui o subscrito i, $1 \leqslant i \leqslant 16$.
(‡) As variáveis com espaço nulo são utilizadas para guardar resultados temporários e que não precisam de ser implementadas em *hardware*.

Tabela 8.26: Estimativa para as métricas associadas às variáveis da descrição do sistema de criptografia DES.

- a percentagem de objectos que se deslocou para construir a solução inicial duma pesquisa foi 20%;
- a validade do tabu aplicado aos deslocamentos, aos deslocamentos inversos e aos objectos foi 25, 22 e 20 iterações, respectivamente;
- na terceira alternativa de deslocamento, que funciona como recurso para a definição da próxima solução duma pesquisa, seleccionou-se o deslocamento menos frequente.

SW	***HW1***	***HW2***	***HW3***	***HW4***
$doutValid$, $dout$	$keyOutR_1$	$keyInR$	$dinValidR$	$doutR_4$
$encrypt$, $stall$	$doutValidR_1$	$doutR_1$	$dinR$	$encryptOutR_1$
din, $dinValid$	$doutValidR_3$	$doutR_2$	$doutR_3$	$encryptOutR_4$
$keyIn$, $keyOut$	key_2	$encryptOutR_2$	$keyOutR_2$	$doutValidR_4$
$doutR_{5..16}$	key_5	$encryptOutR_3$	$keyOutR_3$	$dataSbox_5$
$keyOutR_{5..16}$	$dataSbox_2$	$doutValidR_2$	$keyOutR_4$	$dataPbox_4$
$encryptShift_{1..16}$	$dataSbox_3$	key_1	key_3	$dataPbox_5$
$decryptShift_{1..16}$	$dataPbox_2$	$dataSbox_1$	key_4	$dataEp_5$
$encryptOutR_{5..16}$	$dataEp_2$	$dataPbox_1$	$dataSbox_4$	$IIpipeline_{1,3,4,5}$
$doutValidR_{5..16}$	$desFp$	$dataPbox_3$	$dataEp_4$	$FIpipeline_{1,3,4,5}$
$key_{6..16}$	$desPc1$	$dataEp_1$	$desIp$	$wrEncrypt_{4,5}$
$dataSbox_{6..16}$	$wrEncrypt_2$	$dataEp_3$	$wrEncrypt_1$	$IItypeOp_{2,5}$
$dataPbox_{6..16}$	$IItypeOp_1$	$IIpipeline_2$	$desEp_3$	$FItypeOp_{2,5}$
$dataEp_{6..16}$	$IItypeOp_3$	$FIpipeline_2$	$desEp_4$	$keyShiftL_{2,3,5}$
$parityKey$	$FItypeOp_1$	$wrEncrypt_3$	$desSbox_4$	$keyShiftR_{2,3,5}$
$nShifts$	$FItypeOp_3$	$IItypeOp_4$		$desPc2_{2,3,5}$
$IIpipeline_{6..16}$	$keyShiftL_1$	$FItypeOp_4$		$desEp_5$
$FIpipeline_{6..16}$	$keyShiftR_1$	$keyShiftL_4$		$desSbox_5$
$wrEncrypt_{6..16}$	$desPc2_1$	$keyShiftR_4$		$desPbox_{4,5}$
$IItypeOp_{6..16}$	$desEp_2$	$desPc2_4$		$wrDout_{4,5}$
$FItypeOp_{6..16}$	$desSbox_2$	$desEp_1$		
$keyShiftL_{6..16}$	$desPbox_2$	$desSbox_1$		
$keyShiftR_{6..16}$	$desPbox_3$	$desSbox_3$		
$desPc2_{6..16}$	$wrDout_2$	$desPbox_1$		
$desEp_{6..16}$		$wrDout_1$		
$desSbox_{6..16}$		$wrDout_3$		
$desPbox_{6..16}$				
$wrDout_{6..16}$				

Tabela 8.27: Melhor solução de partição obtida pelo algoritmo de pesquisa tabu para o sistema de criptografia DES.

A estimativa do espaço ocupado pelo caminho de dados e pela unidade de controlo das partições de *hardware* que compõem a melhor solução de partição está quantificada nas tabelas 8.28 e 8.29, respectivamente. As tabelas anteriores contêm também estimativas das métricas elementares que contribuem para o espaço ocupado pelo caminho de dados e pela unidade de controlo. As partições de *hardware* desta solução exigem um espaço para o caminho de dados que é ligeiramente superior às 13100 portas lógicas permitidas por partição: o caminho de dados da partição $HW1$ a $HW4$ ocupa 13.4K, 13.4K, 13.1K e 13.7K. Embora o condicionalismo imposto ao espaço seja ligeiramente desrespeitado, a solução continua a ser exequível uma vez que o condicionalismo representa apenas 80% das 16400 portas lógicas disponíveis por FPGA. Para as mesmas partições, o espaço ocupado pela unidade de controlo não excede as 2400 portas lógicas disponíveis. Como se pode observar pela tabela 8.29, a unidade de controlo das partições $HW1$ a $HW4$ ocupa 1085, 972, 527 e 1627 portas lógicas equivalentes.

	Estimativa			
Métrica	***p = HW1***	***p = HW2***	***p = HW3***	***p = HW4***
$areaEPrograma(CD(p))$	11464	11408	10184	11464
$areaVars(CD(p))$	348	1122	1782	402
$areaInterface(CD(p))$	1443	725	1177	1338
$areaMuxesExtra(CD(p))$	169	169	0	507
areaHW(CD(p))	***13424***	***13424***	***13143***	***13711***

Tabela 8.28: Melhor solução de partição obtida pelo algoritmo de pesquisa tabu para o sistema de criptografia DES: estimativa do espaço ocupado pelo caminho de dados das partições de *hardware*.

	Estimativa			
Métrica	***p = HW1***	***p = HW2***	***p = HW3***	***p = HW4***
$nE(p)$	96	80	42	151
$nSC(p)$	39	37	16	63
$nASC(p)$	135	132	47	207
$nSE(p)$	9	5	6	24
$nLSE(p)$	17	12	9	36
areaRegEstado(p)	576	480	252	906
areaLogicaCtl(p)	288	285	93	432
areaLProxEstado(p)	71	57	32	139
areaHWfsm(UC(p))	935	822	377	1477
areaHW(UC(p))	***1085***	***972***	***527***	***1627***

Tabela 8.29: Melhor solução de partição obtida pelo algoritmo de pesquisa tabu para o sistema de criptografia DES: estimativa do espaço ocupado pela unidade de controlo das partições de *hardware*.

De acordo com as estimativas da tabela 8.29, a solução obtida pelo algoritmo de partição exige um número elevado de estados e de sinais de controlo. Um grau de comunicação elevado, para aceder a variáveis ou para implementar as mudanças de partição que ocorrem no fluxo de controlo, é a principal explicação para a explosão do número de estados e de sinais de controlo. O número de sinais de controlo é ainda penalizado pelo facto de na sua estimação se usar um método simples, que não considera a partilha dos sinais envolvidos nas várias transacções entre as mesmas partições de *hardware* adjacentes.

A estimativa para o desempenho do sistema DES, implementado de acordo com a melhor solução de partição obtida pelo algoritmo de pesquisa tabu, traduz-se num débito de 4.18 Mbits/s ou numa latência de aproximadamente 58 μs (tabela 8.30). A tabela 8.30 permite comparar o desempenho da melhor solução de partição com outras alternativas de implementação: o desempenho é melhor que os 2.67 Mbits/s da implementação totalmente em *software*, mas é muito pior que os 66.7 (ou 100) Mbits/s da implementação totalmente em *hardware* a 33 (ou 50) MHz. A implementação totalmente em *hardware* atinge um desempenho muito superior ao da solução obtida pelo algoritmo de pesquisa tabu porque, no primeiro caso as 16 passa-

gens do algoritmo são implementadas em *hardware*, enquanto no segundo caso apenas 4 a 5 passagens são implementadas em *hardware*. Embora a implementação totalmente em *hardware* atinja um desempenho muito superior ao da solução obtida pelo algoritmo de pesquisa tabu, exige uma quantidade de recursos não disponível na arquitectura alvo. A bibliografia referencia implementações extremamente optimizadas que aplicam as técnicas de *pipeline* e de *loop unrol* e atingem um desempenho até 400 Mbits/s [KP98].

Solução totalmente em *software*, código não optimizado, *Windows* 2000, P200			
Métrica	***Estimativa***	***Medição***	***Precisão (%)***
Latência da (de)cifragem (μs)	105	114	92
Débito da (de)cifragem *(Mbits/s)*	0.61	0.56	91

Solução totalmente em *software*, código optimizado relativamente ao tempo de execução, *Windows* 2000, P200			
Métrica	***Estimativa***	***Medição***	***Precisão (%)***
Latência da (de)cifragem (μs)	24	26.0	92
Débito da (de)cifragem *(Mbits/s)*	2.67	2.46	91

Solução totalmente em *hardware*, relógio de *hardware* a 33 MHz	
Métrica	***Estimativa***
Latência da (de)cifragem (μs)	6.3
Débito da (de)cifragem *(Mbits/s)*	66.7

Solução totalmente em *hardware*, relógio de *hardware* a 50 MHz	
Métrica	***Estimativa***
Latência da (de)cifragem (μs)	4.2
Débito da (de)cifragem *(Mbits/s)*	100.0

Melhor solução de partição obtida pelo algoritmo de pesquisa tabu, código optimizado relativamente ao tempo de execução, P200 e EDgAR-2 a 33 MHz	
Métrica	***Estimativa***
Latência da (de)cifragem (μs)	58.5
Débito da (de)cifragem *(Mbits/s)*	4.18

Melhor solução de partição obtida pelo algoritmo de pesquisa tabu, código optimizado relativamente ao tempo de execução, P200 e EDgAR-2 a 50 MHz	
Métrica	***Estimativa***
Latência da (de)cifragem (μs)	57.9
Débito da (de)cifragem *(Mbits/s)*	4.18

Tabela 8.30: Desempenho para várias soluções de partição/implementação do sistema de criptografia DES.

8.3.3 Implementação *Hardware/Software* Optimizada

Descrição da implementação

De acordo com a arquitectura do sistema DES apresentada na figura 8.20, uma forma de obter um desempenho elevado é dispor de lógica para efectuar em simultâneo todas as 16 passagens do algoritmo. Tendo por base os condicionalismos impostos pela arquitectura da plataforma EDgAR-2, ao nível da comunicação entre *hardware* e *software* e dos recursos de *hardware* disponíveis, a implementação que se utilizou na validação da solução de partição obtida pelo

algoritmo PT inclui apenas 4 estágios de *pipeline* em vez dos 16 sugeridos pela arquitectura da figura 8.20. Para simplificar a implementação, quando não se dispõe de recursos para executar as 16 passagens em simultâneo, o número de estágios de *pipeline* deve ser um sub-múltiplo de 16, ou seja, 8, **4** ou 2. Como as estimativas obtidas anteriormente mostraram não ser possível implementar duas passagens em cada uma das quatro FPGAs da arquitectura alvo, o valor seleccionado para o número de estágios de *pipeline* foi 4.

A arquitectura utilizada na implementação do sistema DES com 4 estágios de *pipeline* pode ser ilustrada pelo caminho de dados simplificado da figura 8.22. A figura apresenta apenas o fluxo das amostras a (de)cifrar, ficando de fora o fluxo das chaves aplicadas ao longo do processo de cifragem. Cada estágio é responsável por efectuar uma passagem do algoritmo DES, acrescendo para o estágio de entrada (0) e de saída (3) da cadeia do *pipeline* as tarefas iniciais (permutações IP e $PC1$) e finais (permutação FP/IP^{-1} e a inserção de paridade na chave de saída) do algoritmo, respectivamente. Cada estágio possui a sua cópia registada da informação à entrada e à saída do estágio. Por exemplo, o estágio 1 regista a amostra e a chave de entrada em $dinR1$ e $keyInR1$, sendo a amostra e a chave de saída guardadas em $doutR1$ e $keyOutR1$. A entrada para os estágios 1 a 3 é a saída do estágio anterior (0 a 2), enquanto a entrada para o estágio 0 pode ser a saída do estágio 3 ou uma nova amostra a processar.

Uma das formas que se pode usar para descrever o funcionamento da implementação do sistema de criptografia DES é através do fluxo de informação ao longo do processo de cifragem. A figura 8.23 esquematiza as transformações que as amostras sofrem desde que entram (no estágio 0) até saírem da cadeia de *pipeline* (pelo estágio 3). Durante as primeiras 4 passagens após o arranque do sistema, decorre o preenchimento da cadeia de *pipeline*. Deste modo, na primeira passagem funciona apenas o estágio 0, na segunda funcionam os estágios 0 e 1 e na terceira funcionam os estágios 0, 1 e 2. A partir da quarta passagem a cadeia de *pipeline* funciona na totalidade.

Dado que o algoritmo DES decorre em 16 passagens, a implementação com quatro estágios de *pipeline* recebe e gera 4 amostras por cada 16 passagens. Se a variável *passagem* funcionar como contador do número de passagens, no início das passagens 0 a 3 o estágio 0 recebe quatro amostras a processar e no final das passagens 15, 0, 1 e 2 o estágio 3 produz quatro amostras processadas. Como exemplo, a figura 8.23 destaca as transformações por que passa a amostra $v2$, introduzida na cadeia de *pipeline* na passagem 1, até produzir a saída $v2c$, 16 passagens mais tarde. A amostra $v2$ assume o valor $v2(1)$ após ter sido processada uma vez, $v2(2)$ depois de ter sido processada duas vezes e $v2(16)$ após a série de 16 passagens.

De uma forma sintética, a temporização das acções efectuadas pelos quatro estágios da implementação do sistema DES pode ser definida à custa dos diagramas de estados da figura 8.24.

O estágio 0, aquele que recebe a informação a processar, espera que o sinal $doutValid$ do estágio anterior[11] seja activado antes de ler as novas amostra e chave a processar. Quando o sinal $dinValid$ do presente estágio fica activo significa que o estágio anterior acabou de processar uma amostra e uma chave, ou então que existe uma amostra e uma chave vindas do exterior prontas a ser processadas. Depois de ler a informação a processar, desactiva-se o sinal $doutValid$ do estágio anterior e inicia-se o processamento correspondente a uma passagem do algoritmo DES. Para registar os resultados do processamento, é necessário confirmar se o estágio seguinte já não precisa dos resultados produzidos na passagem anterior pelo presente estágio. Esta condição ocorre quando o sinal $doutValid$ fica inactivo. Após registar a amostra e a chave processadas, activa-se o sinal $doutValid$ para indicar ao estágio seguinte que as pode ler e incrementa-se o contador de passagens *passagem* (figura 8.24 (i)).

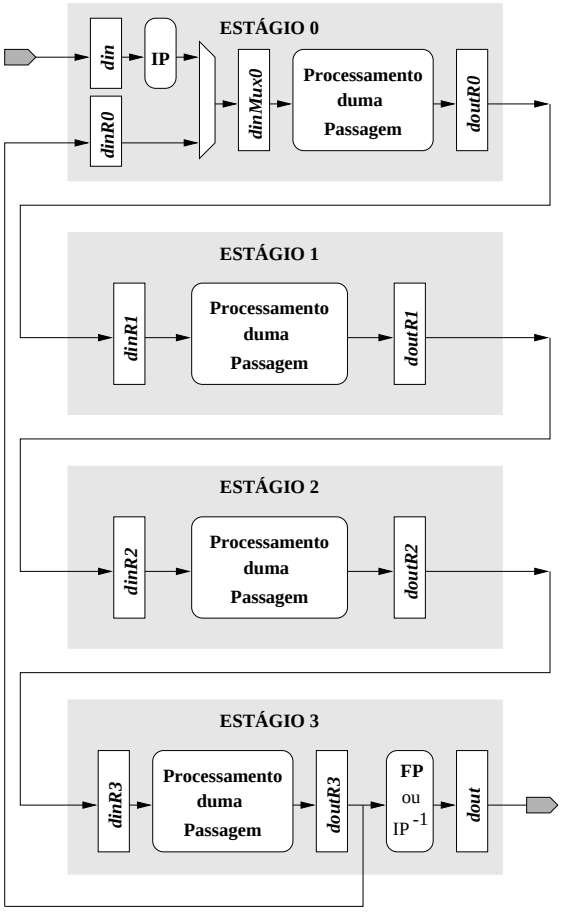


Figura 8.22: Caminho de dados simplificado definido na implementação do sistema de criptografia DES com quatro estágios de *pipeline*.

Para os estágios 1 a 3 a sequência de acções sofre uma pequena alteração relativamente ao estágio 0. Esta alteração resulta de o estágio 0 funcionar desde a primeira passagem, enquanto os restantes entram em funcionamento um de cada vez na segunda, terceira e quarta passagens, como mostra a figura 8.23. Para garantir um funcionamento correcto durante o preenchimento

[11] O sinal $doutValid$ do estágio anterior coincide com o sinal $dinValid$ do estágio presente.

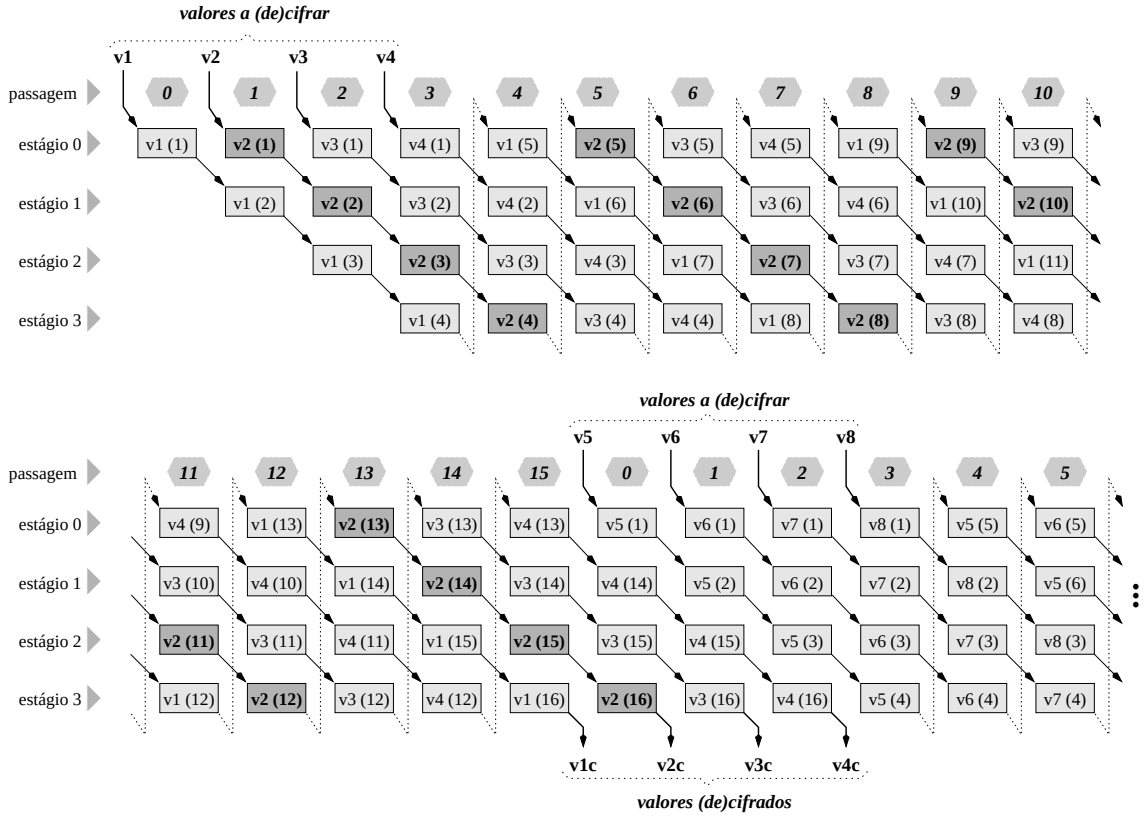


Figura 8.23: Fluxo de dados durante o processo de (de)cifragem quando a implementação do sistema de criptografia DES recorre a quatro estágios de *pipeline*.

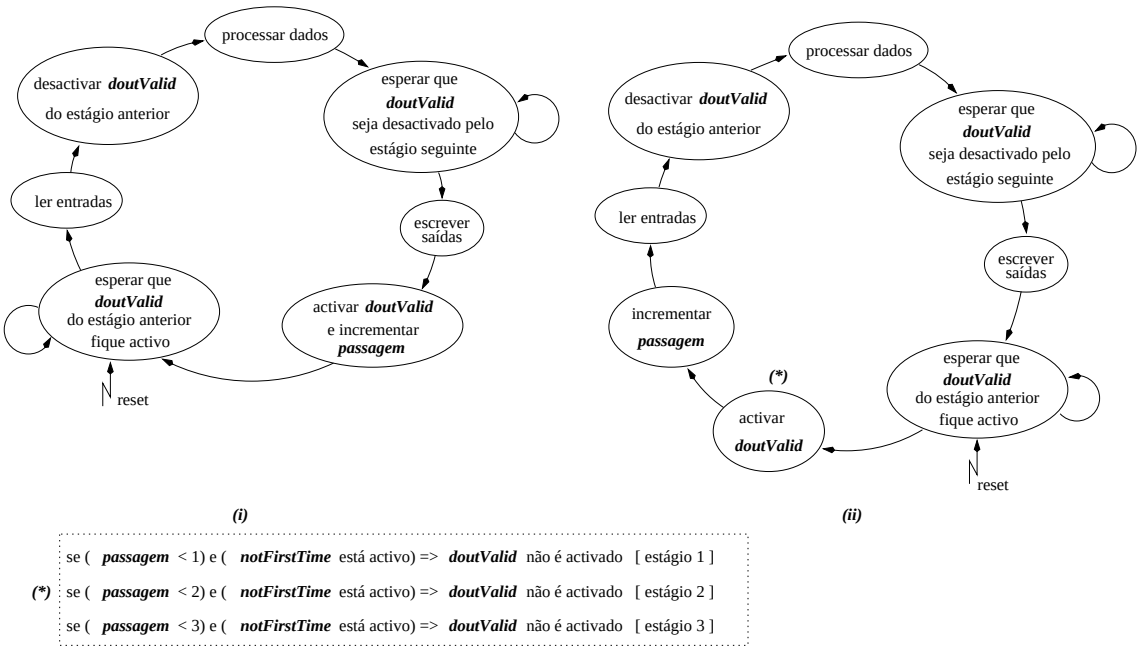


Figura 8.24: Unidade de controlo simplificada para cada um dos quatro estágios utilizados na implementação do sistema de criptografia DES: (i) estágio 0 e (ii) estágios 1 a 3.

da cadeia de *pipeline* e nas passagens posteriores: (i) o contador de passagens para cada um dos estágios 1, 2 e 3 arranca com o valor 0, 1 e 2 e (ii) a activação do sinal $doutValid$ e a actualização do contador de passagens não se processa imediatamente após o estágio ter registado os resultados do processamento, mas sim após esse instante e apenas quando o estágio anterior activar o seu $doutValid$, ou seja, depois de activar o $dinValid$ do presente estágio (figura 8.24 (ii)). Para garantir que os estágios 1 a 3 não entram cedo de mais em funcionamento utiliza-se a condição de guarda assinalada com "*" na figura 8.24 (ii). Esta condição emprega o valor do contador de passagens e o sinal $notFirstTime$, que no arranque do sistema está activo e fica inactivo a partir do momento em que a cadeia de *pipeline* esteja preenchida.

A funcionalidade do sistema de criptografia DES é traduzida no modelo PSM da figura 8.25. Cada um dos cinco estados principais, que operam em paralelo neste modelo, é composto pelos estados programa e variáveis atribuídos a uma partição distinta da implementação: o estado programa E_{DESsw} constitui a partição de *software* e os estados E_{DEShw1} a E_{DEShw4} correspondem às partições de *hardware* $HW1$ a $HW4$. A descrição correspondente, através do meta-modelo PSMfg, foi incluída na figura 8.26. O modelo PSM, ou PSMfg, contém vários pontos de sincronismo: (i) o estado $waitDinValid0$ espera por (um evento emitido por) $writeData$, (ii) o estado $waitRoundDinValid_{1,2,3}$ espera por $setRoundDoutValid_{0,1,2}$, (iii) o estado $waitToRead$ espera por $assertToRead3$ e $waitReadDone3$ espera por $readDone$. Os pontos de sincronismo modelam parte da temporização imposta às acções efectuadas pelos quatro estágios da implementação.

Apresenta-se a seguir um resumo da funcionalidade associada aos estados do modelo PSM:

- ***writeData*** envia para o estágio 0 uma nova amostra e chave a usar na (de)cifragem;
- ***waitToRead*** espera até o estágio 3 ter concluído a última passagem do processamento duma amostra;
- ***readData*** lê a última amostra processada a partir do estágio 3 para o exterior;
- ***readDone*** indica ao estágio 3 que a amostra processada já foi lida;
- ***resetHW*** e $\boldsymbol{resetLogic_i}$ definem o estado inicial de toda a lógica sequencial;
- $\boldsymbol{comLogic_i}$ - operações que geram os sinais $selectIn$ e $selectOut$, utilizados para indicar em que passagens se lê uma amostra a partir do exterior e se envia uma amostra processada para o exterior, respectivamente (figura 8.27);
- ***waitDinValid0*** - o estágio 0 espera que o sinal $dinValidR0$ seja activado a partir do exterior ($dinValid$) ou pelo estágio 3 ($doutValidR3$);

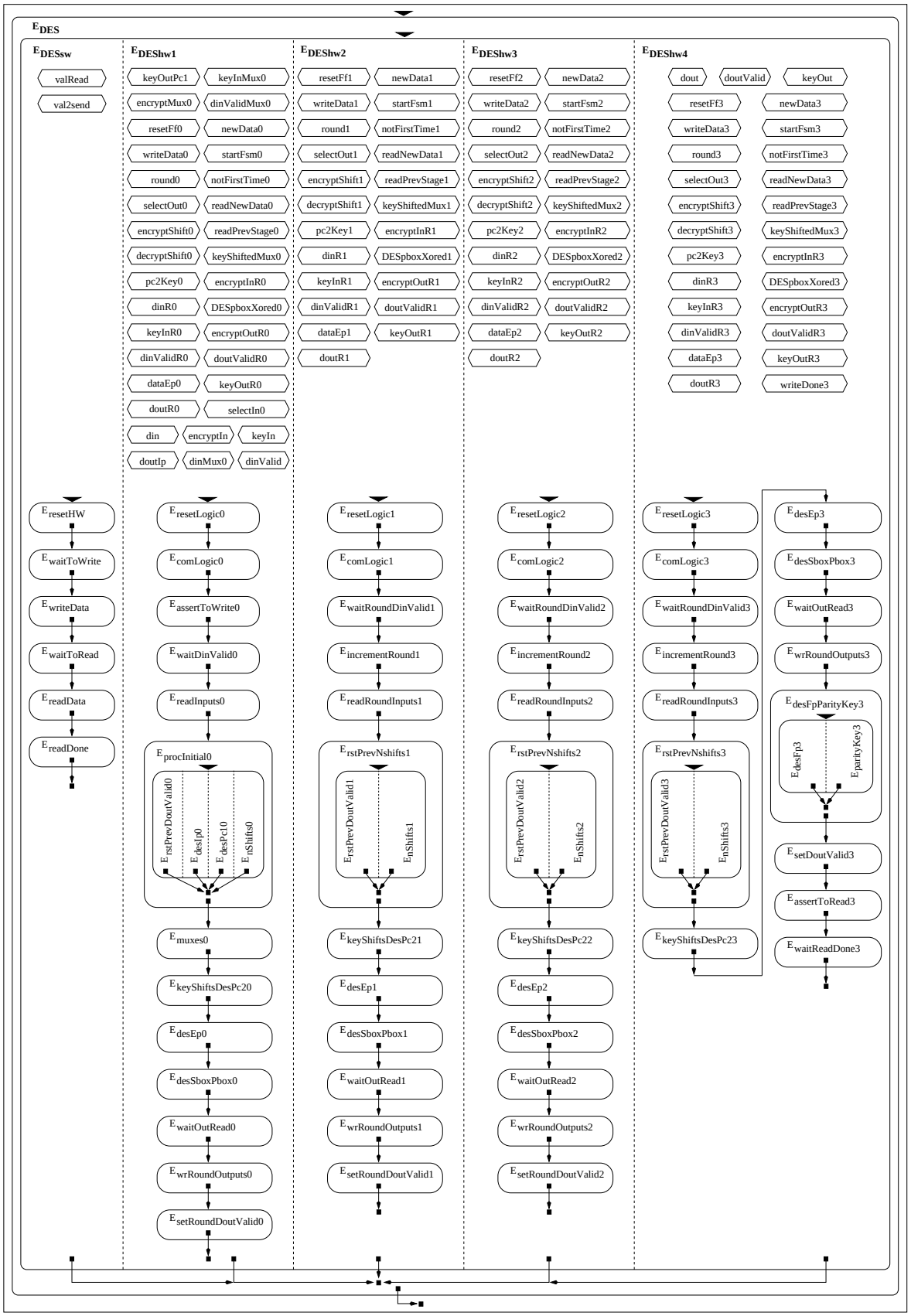


Figura 8.25: Descrição, com o meta-modelo PSM, aplicada na implementação *hardware/software* do sistema de criptografia DES com quatro estágios de *pipeline*.

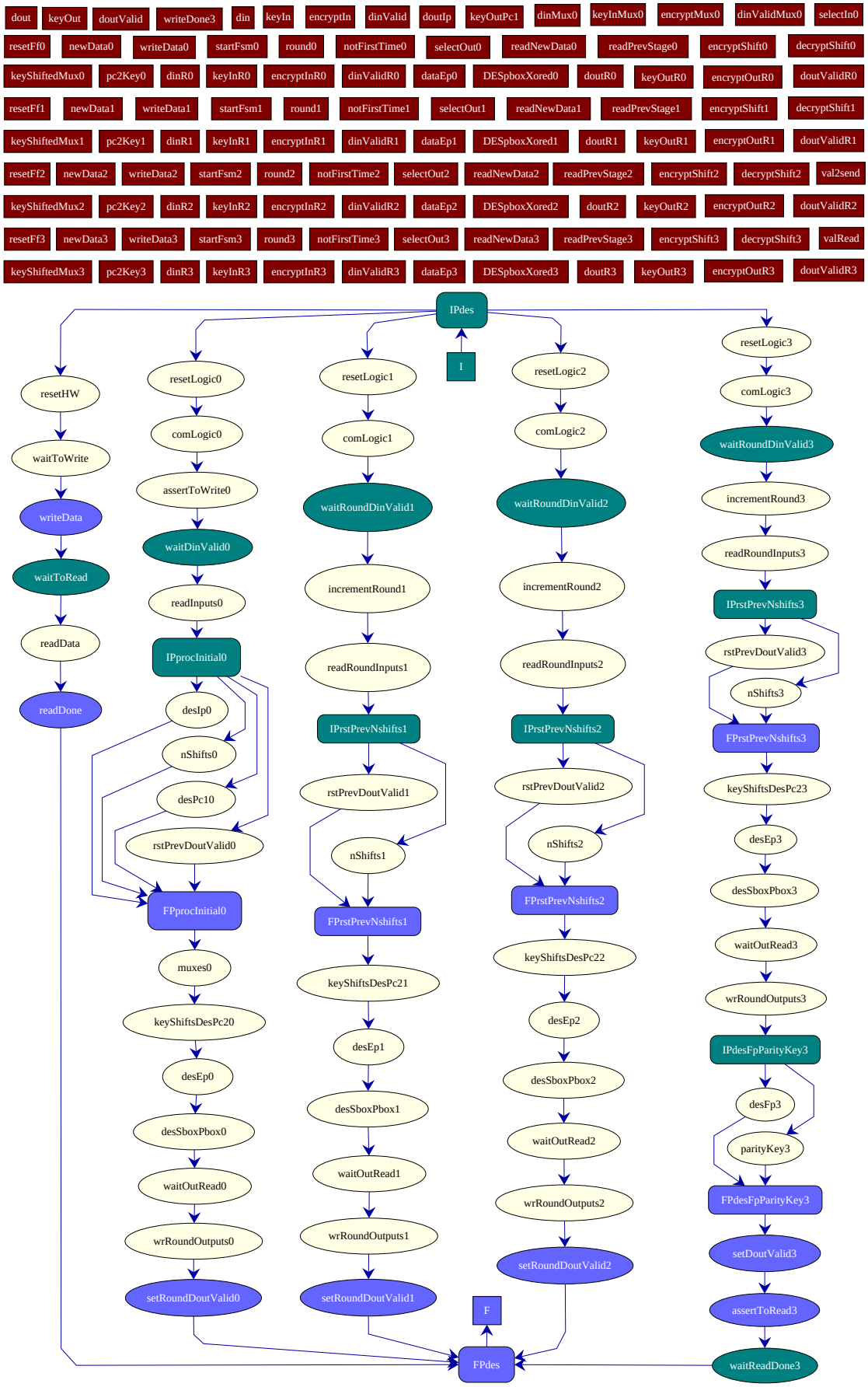


Figura 8.26: Descrição, com o meta-modelo PSMfg, aplicada na implementação *hardware/-software* do sistema de criptografia DES com quatro estágios de *pipeline*.

- ***waitRoundDinValid*$_{1,2,3}$** - o estágio 1,2,3 espera que o sinal $doutValidR_{0,1,2}$ seja activado pelo estágio 0,1,2;
- ***readInputs*0** lê uma nova amostra a processar a partir do exterior ou então lê do estágio 3 o resultado da última passagem executada ($doutR3$, $keyOutR3$ e $encryptOutR3$);
- ***readRoundInputs*$_{1,2,3}$** lê do estágio anterior (0,1,2) o resultado da última passagem executada ($doutR_{0,1,2}$, $keyOutR_{0,1,2}$ e $encryptOutR_{0,1,2}$);
- ***rstPrevDoutValid*$_i$** - o estágio i desactiva o sinal $doutValidR_{i-1}$ do estágio anterior;
- ***desIp*0** - o estágio 0 executa a permutação inicial (IP) sobre a amostra;
- ***desPc*10** - o estágio 0 efectua a permutação 1 ($PC1$) sobre a chave secreta;
- ***nShifts*$_i$** - o estágio i calcula o número de deslocamentos a aplicar à chave;
- ***muxes*0** selecciona as entradas para as passagens executadas pelo estágio 0;
- ***keyShiftsDesPc2*$_i$** - o estágio i executa a permutação 2 ($PC2$) sobre a chave secreta;
- ***desEp*$_i$** aplica a expansão E (ou EP) à amostra;
- ***desSboxPbox*$_i$** aplica a tabela de substituição ($S-box$) e a permutação P (ou $P-box$) sobre a amostra;
- ***waitOutRead*$_i$** espera que o sinal $doutValidR_i$ seja desactivado pelo estágio seguinte;
- ***wrRoundOutputs*$_i$** regista os resultados da passagem actual ($doutR_i$, $keyOutR_i$ e $encryptOutR_i$);
- ***setRoundDoutValid*$_{0,1,2}$** activa o sinal $doutValidR_{0,1,2}$ e no caso do estágio 0 incrementa o contador de passagens;
- ***incrementRound*$_{1,2,3}$** incrementa o contador de passagens;
- ***desFp*3** - o estágio 3 executa a permutação final IP^{-1} (ou FP) sobre a amostra;
- ***parityKey*3** - o estágio 3 junta à chave manipulada os bits de paridade;
- ***setDoutValid*3** - o estágio 3 activa o sinal $doutValid$, que indica ao exterior a conclusão do processamento duma amostra;
- ***assertToRead*3** - o estágio 3 indica ao exterior que pode ler o resultado do processamento duma amostra ($dout$ e $keyOut$);
- ***waitReadDone*3** - o estágio 3 espera que o exterior sinalize que a última amostra processada já foi lida.

O código VHDL completo, representando a funcionalidade dos estados programa atribuídos à partição de *hardware* $HW1$, pode ser consultado em [Est01]. Neste código inclui-se a descrição do caminho de dados e da unidade de controlo da partição $HW1$, a qual implementa o estágio 0 da cadeia de *pipeline*. Como as partições $HW2$ a $HW4$, e em especial as partições $HW2$ e $HW3$, possuem uma funcionalidade muito semelhante, o código que descreve essa funcionalidade não está disponível.

Caminho de dados das partições de *hardware*

O caminho de dados definido pela arquitectura das partições de *hardware*, ilustrado na figura 8.27, contém toda a lógica necessária às operações executadas pelos estados programa do modelo PSM. A lógica sombreada do caminho de dados implementa as tarefas iniciais (permutações IP e $PC1$) e finais (permutação FP e inserção de paridade na chave de saída) do algoritmo DES, sendo por isso incluída apenas no caminho de dados da partição $HW1$ e $HW4$, respectivamente. Os sinais din, $keyIn$, $encryptIn$ e $dinValid$ funcionam como entrada no estágio 0 a partir do exterior do sistema e as entradas $dinR$, $keyInR$, $encryptInR$ e $dinValidR$ são a entrada proveniente do estágio anterior. Os estágios 1 a 3 apenas recebem dados a partir do estágio anterior. Deste modo, apenas o estágio 0 exige multiplexadores para seleccionar a origem dos dados a processar. A ligação entre a saída do estágio 0,1,2,3 e a entrada do estágio 1,2,3,0 faz-se através dos sinais $doutR$, $keyOutR$, $encryptOutR$ e $doutValidR$, enquanto os resultados após as 16 passagens de processamento sobre as amostras saem do estágio 3 para o exterior do sistema por $dout$, $keyOut$, $encryptOut$ e $doutValid$.

Unidade de controlo das partições de *hardware*

O diagrama de estados detalhado relativo à unidade de controlo que gera os sinais que coordenam as actividades do caminho de dados de cada partição de *hardware* $HW1$ a $HW4$ é esquematizado nas figuras 8.28 e 8.29. Estas máquinas de estados constituem um refinamento dos diagramas introduzidos na figura 8.24. Pode por isso estabelecer-se uma ligação entre os macro-estados da figura 8.24 e os estados das máquinas refinadas. Apresenta-se como exemplo a máquina de estados da partição $HW1$ ilustrada na figura 8.28:

- o conjunto definido pelos estados $enableRnewData$ e $enableRprevStage$ até ao estado $read5dinValidR$ implementa o ciclo em que se espera até o sinal $dinValidMux$ ficar activo; como este sinal resulta da multiplexagem de $dinValid$ (sinal disponível na partição porque provém do exterior do sistema) e de $dinValidR$ (sinal coincidente com $doutValidR$ do estágio anterior), a secção de estados tem que estar preparada para coordenar a leitura de $doutValidR$ a partir do estágio anterior[12];
- os estados $incCntReads$, $tstCntReads$ e $read1others$ até $read5others$ representam o

[12] Ler do estágio anterior equivale a uma leitura duma partição de *hardware* adjacente.

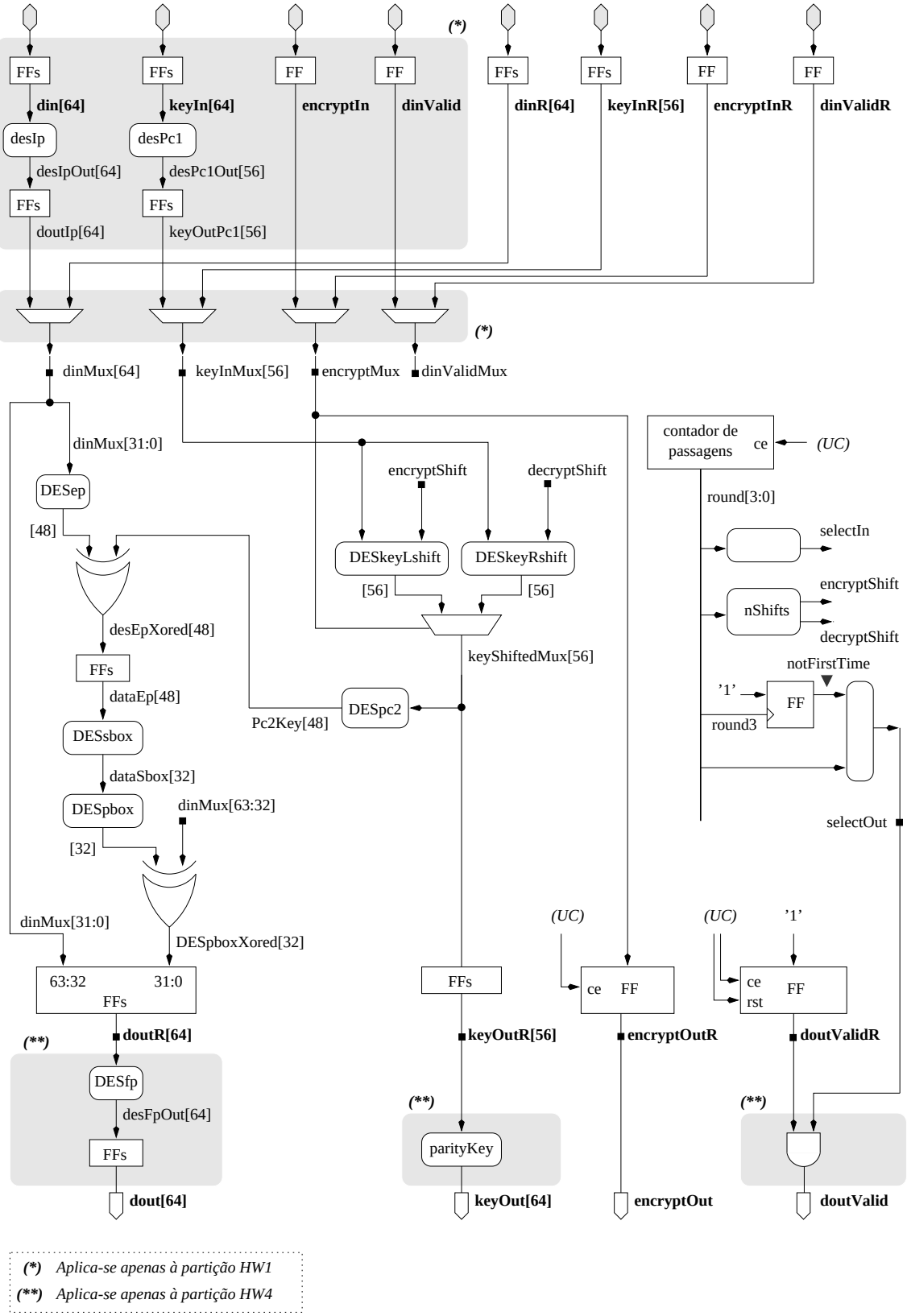


Figura 8.27: Caminho de dados das partições de *hardware* ***HW1*** a ***HW4*** incluídas na implementação *hardware/software* do sistema de criptografia DES.

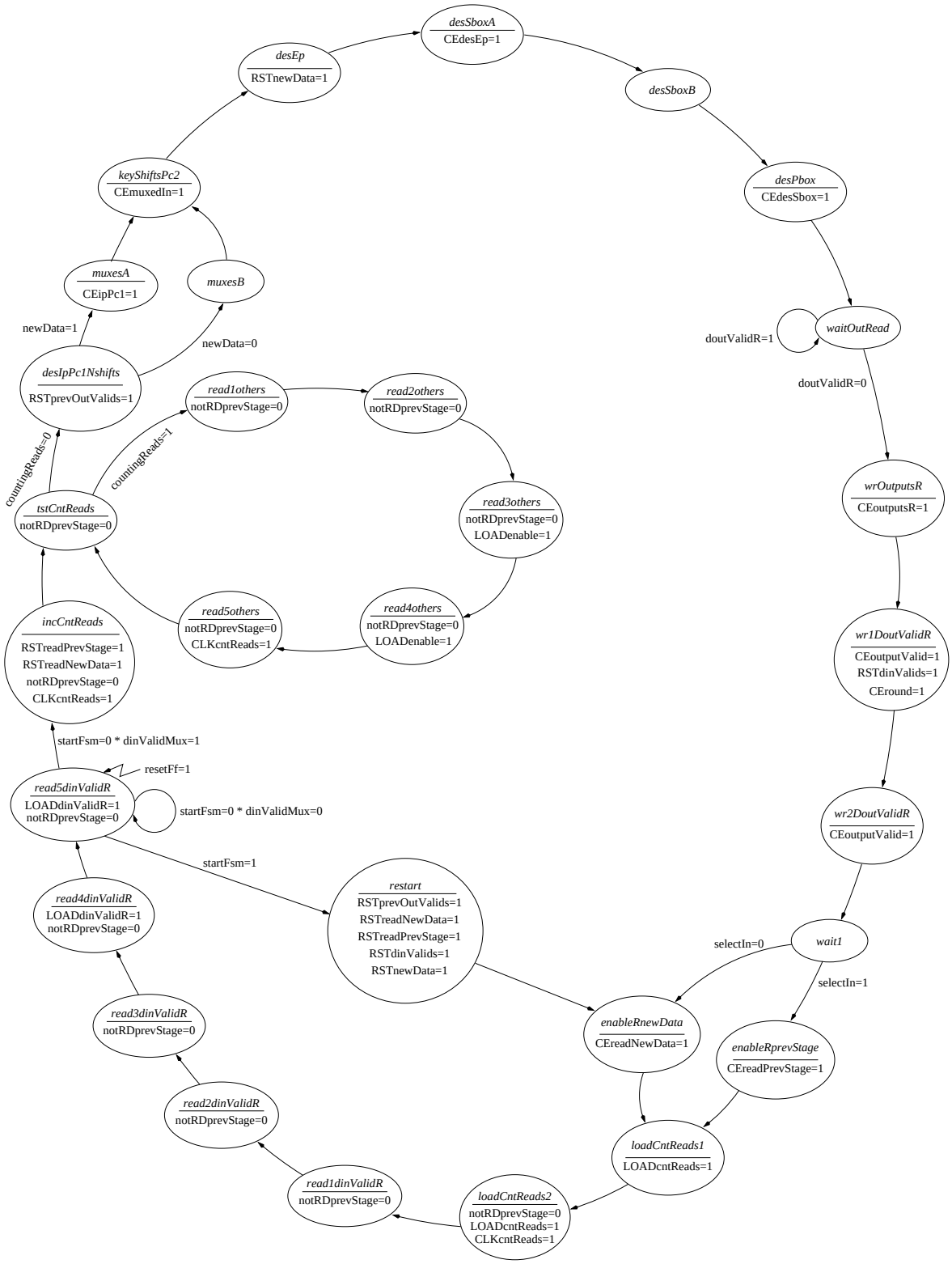


Figura 8.28: Unidade de controlo da partição de *hardware* ***HW1*** envolvida na implementação *hardware/software* do sistema de criptografia DES.

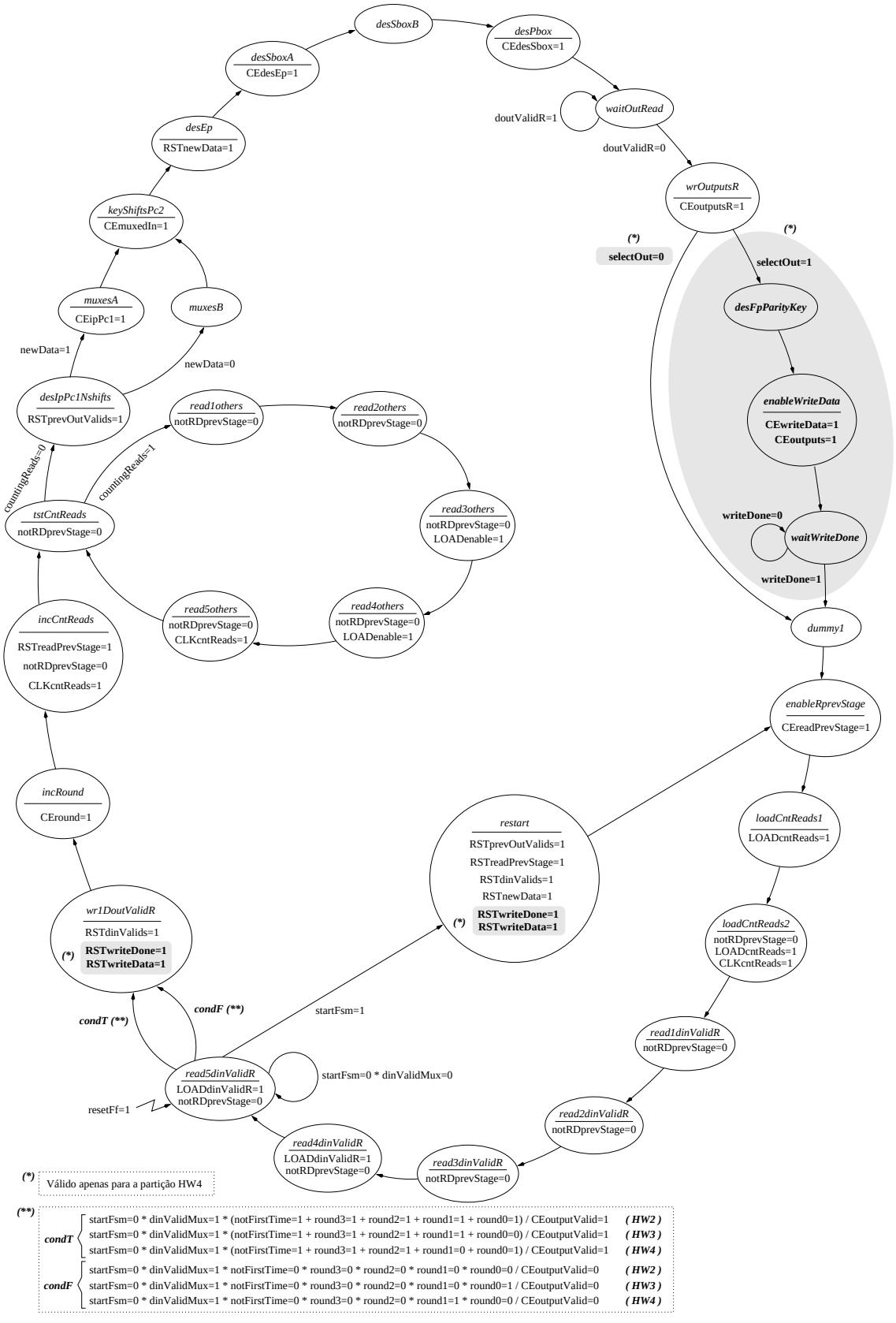


Figura 8.29: Unidade de controlo das partições de *hardware* ***HW2*** a ***HW4*** envolvidas na implementação *hardware/software* do sistema de criptografia DES.

ciclo em que se lêem os dados a processar a partir do estágio anterior; estes estados só são úteis quando a passagem actual é diferente de 0, ou seja, quando os dados não provêm do exterior do sistema;

- ◇ o estado $desIpPc1Nshifts$ permite desactivar o sinal $doutValidR$ no estágio anterior;
- ◇ a secção que vai desde os estados $muxesA$ e $muxesB$ até ao estado $desPbox$ corresponde ao processamento de dados relativo a uma passagem;
- ◇ o estado $waitOutRead$ implementa o ciclo em que se espera que o sinal $doutValidR$ seja desactivado pelo estágio seguinte;
- ◇ com o sinal activado no estado $wrOutputsR$ habilita-se o registo dos dados resultantes do processamento da passagem actual;
- ◇ os estados $wr1DoutValidR$ a $wait1$ permitem que se active o sinal $doutValidR$ e se incremente o valor da passagem actual.

As máquinas de estados correspondentes à unidade de controlo das partições $HW2$ e $HW3$ são iguais entre si e muito semelhantes à máquina de estados da partição $HW1$. Por seu lado, a unidade de controlo da partição $HW4$ inclui todos os estados presentes na unidade de controlo das partições $HW2$ ou $HW3$, mais os estados $desFparityKey$, $enableWriteData$ e $waitWriteDone$, responsáveis pelo controlo das tarefas finais do algoritmo DES: executar a permutação FP, inserir a paridade na chave de saída, registar o resultado das duas tarefas anteriores e esperar que os dados registados sejam lidos para o exterior do sistema. As condições $condT$ e $condF$, assinaladas com "**" na figura 8.29, garantem que no arranque do sistema os estágios 1 a 3 não entram cedo de mais em funcionamento.

Estimativa de métricas ao nível do estado programa

As estimativas das métricas, obtidas ao nível do estado programa, estão incluídas nas tabelas 8.31 a 8.34: (i) a tabela 8.31 contém estimativas para o espaço ocupado em *hardware*, o tempo de computação em *hardware* e o tempo de computação em *software* dos estados programa da descrição do sistema, (ii) nas tabelas 8.32 e 8.33 encontram-se as estimativas das variáveis lidas, escritas e que necessitam dum multiplexador na sua entrada e (iii) na tabela 8.34 é apresentada a estimativa do espaço ocupado em *hardware* pelas variáveis.

8.3.4 Resultados Obtidos com a Implementação *Hardware/Software*

Tal como foi definido no exemplo da convolução, os objectivos pretendidos com a implementação *hardware/software* do sistema de criptografia são: (i) medir o desempenho do sistema,

Estado programa	***Tipo de recurso de H/W***	***areaUF*** *(portas lógicas)*	***TexecSW*** *(ciclos do μP)*	***TexecHW*** *(ciclos do h/w)*
resetHW	–	–	0	–
waitToWrite	–	–	12	–
writeData **(A)**	–	–	48	–
waitToRead **(E)**	–	–	12	–
readData	–	–	48	–
readDone **(A)**	–	–	0	–
*resetLogic*0	–	0	–	0
*comLogic*0	lógica combinatória	159	–	1
*assertToWrite*0	–	0	–	0
*waitDinValid*0 **(E)**	–	0	–	8
*readInputs*0	–	0	–	38
*rstPrevDoutValid*0	–	0	–	1
*desIp*0	–	0	–	0
*desPc*10	–	0	–	0
*nShifts*0	lógica combinatória	21	–	1
*muxes*0	–	0	–	1
*keyShiftsDesPc*20	–	0	–	0
*desEp*0	OU-exclusivo	240	–	1
*desSboxPbox*0	ROM, OU-exclusivo	3194	–	2
*waitOutRead*0	–	0	–	1
*wrRoundOutputs*0	–	0	–	0
*setRoundDoutValid*0 **(A)**	contador 4 bits	72	–	3
*resetLogic*1,2	–	0	–	0
*comLogic*1,2	lógica combinatória	157	–	1
*waitRoundDinValid*1,2 **(E)**	–	0	–	4
*incrementRound*1,2	contador 4 bits	72	–	2
*readRoundInputs*1,2	–	0	–	0
*rstPrevDoutValid*1,2	–	0	–	1
*nShifts*1,2	lógica combinatória	21	–	1
*keyShiftsDesPc*21,2	–	0	–	0
*desEp*1,2	OU-exclusivo	240	–	1
*desSboxPbox*1,2	ROM, OU-exclusivo	3194	–	2
*waitOutRead*1,2	–	0	–	1
*wrRoundOutputs*1,2	–	0	–	0
*setRoundDoutValid*1,2 **(A)**	–	0	–	1
*resetLogic*3	–	0	–	0
*comLogic*3	lógica combinatória	159	–	1
*waitRoundDinValid*3 **(E)**	–	0	–	4
*incrementRound*3	contador 4 bits	72	–	2
*readRoundInputs*3	–	0	–	0
*rstPrevDoutValid*3	–	0	–	1
*nShifts*3	lógica combinatória	21	–	1
*keyShiftsDesPc*23	–	0	–	0
*desEp*3	OU-exclusivo	240	–	1
*desSboxPbox*3	ROM, OU-exclusivo	3194	–	2
*waitOutRead*3	–	0	–	1
*wrRoundOutputs*3	–	0	–	0
*desFp*3	–	0	–	0
*parityKey*3	–	0	–	0
*setDoutValid*3	–	0	–	0
*assertToRead*3 **(A)**	–	0	–	0
*waitReadDone*3 **(E)**	–	0	–	0

(A) Estado programa do tipo *activacao*.
(E) Estado programa do tipo *espera*.

Tabela 8.31: Implementação *hardware/software* do sistema de criptografia DES: estimativa para as métricas associadas aos estados programa da descrição.

para avaliar a qualidade da solução obtida pelo algoritmo de partição, (ii) avaliar a precisão das estimativas e (iii) validar a correcção da descrição do sistema.

Estado programa	***Variáveis lidas***	***Variáveis escritas***	***Variáveis com multiplexador***
resetHW	–	*resetFf*0 (2 x), *startFsm*0	–
waitToWrite	*readNewData*0 (2 x), *valRead* (3 x)	*valRead* (4 x)	–
writeData	*val2send* (30 x)	*din*, *keyIn*, *encryptIn*, *dinValid*, *val2send* (16 x)	–
waitToRead	*writeData*3 (2 x), *valRead* (3 x)	*valRead* (4 x)	–
readData	*dout*, *valRead* (14 x)	*valRead* (14 x)	–
readDone	–	*writeDone*3	–
*resetLogic*0	–	*doutValidR*0, *newData*0, *dinValidR*0, *writeData*0, *startFsm*0, *readNewData*0, *readPrevStage*0	–
*comLogic*0	*round*0, *notFirstTime*0	*selectOut*0, *selectIn*0, *notFirstTime*0	–
*assertToWrite*0	–	*readNewData*0	–
*waitDinValid*0	*dinValid*	–	–
*readInputs*0	–	–	–
*rstPrevDoutValid*0	–	*doutValidR*3	–
*desIp*0	*din*	*doutIp*	–
*desPc*10	*keyIn*	*keyOutPc*1	–
*nShifts*0	*round*0	*encryptShift*0, *decryptShift*0	–
*muxes*0	*doutIp*, *keyOutPc*1, *encryptIn*, *dinValid*	*dinMux*0, *keyInMux*0, *encryptMux*0, *dinValidMux*0	*dinMux*0, *keyInMux*0, *encryptMux*0, *dinValidMux*0
*keyShiftsDesPc*20	*encryptShift*0, *decryptShift*0, *keyShiftedMux*0, *keyInMux*0	*pc2Key*0, *keyShiftedMux*0	*keyShiftedMux*0
*desEp*0	*dinMux*0, *pc2Key*0	*dataEp*0	–
*desSboxPbox*0	*dataEp*0, *dinMux*0	*DESpboxXored*0	–
*waitOutRead*0	*doutValidR*0	–	–
*wrRoundOutputs*0	*DESpboxXored*0, *keyShiftedMux*0, *dinMux*0, *encryptMux*0	*doutR*0, *keyOutR*0, *encryptOutR*0	–
*setRoundDoutValid*0	*round*0	*doutValidR*0, *round*0	–

Tabela 8.32: Implementação *hardware/software* do sistema de criptografia DES: estimativa, para cada estado programa atribuído às partições SW e HW1, das variáveis lidas, escritas e que precisam dum multiplexador na entrada.

Desempenho do sistema

O desempenho medido e estimado, para três alternativas de implementação do tipo *hardware/software*, encontra-se na tabela 8.35. Comparando os valores obtidos nas implementações que usam código optimizado com os valores da solução totalmente implementada em *software*, incluídos na tabela 8.30, verifica-se que, embora o tempo necessário para obter a primeira amostra processada piore bastante, ao subir de 26.0 μs para 45.2 ou 39.6 μs, o débito de amostras processadas melhora numa proporção ainda superior, ao subir de 2.46 Mbits/s

Estado programa	***Variáveis lidas***	***Variáveis escritas***	***Variáveis com multiplexador***
$resetLogic1,2$	–	$doutValidR1,2$, $newData1,2$, $writeData1,2$, $dinValidR1,2$, $readNewData1,2$, $readPrevStage1,2$, $startFsm1,2$	–
$comLogic1,2$	$round1,2$, $notFirstTime1,2$	$selectOut1,2$, $notFirstTime1,2$	–
$waitRoundDinValid1,2$	$doutValidR0,1$	$dinValidR1,2$	–
$incrementRound1,2$	$round1,2$	$round1,2$	–
$readRoundInputs1,2$	$keyOutR0,1$, $doutR0,1$, $encryptOutR0,1$	$keyInR1,2$, $dinR1,2$, $encryptInR1,2$	–
$rstPrevDoutValid1,2$	–	$doutValidR0,1$	–
$nShifts1,2$	$round1,2$	$encryptShift1,2$, $decryptShift1,2$	–
$keyShiftsDesPc21,2$	$keyInR1,2$, $keyShiftedMux1,2$, $encryptShift1,2$, $decryptShift1,2$	$keyShiftedMux1,2$, $pc2Key1,2$	$keyShiftedMux1,2$
$desEp1,2$	$dinR1,2$, $pc2Key1,2$	$dataEp1,2$	–
$desSboxPbox1,2$	$dataEp1,2$, $dinR1,2$	$DESpboxXored1,2$	–
$waitOutRead1,2$	$doutValidR1,2$	–	–
$wrRoundOutputs1,2$	$DESpboxXored1,2$, $keyShiftedMux1,2$, $encryptInR1,2$, $dinR1,2$	$doutR1,2$, $keyOutR1,2$, $encryptOutR1,2$	–
$setRoundDoutValid1,2$	–	$doutValidR1,2$	–
$resetLogic3$	–	$doutValidR3$, $writeDone3$, $writeData3$, $dinValidR3$, $startFsm3$, $readNewData3$, $readPrevStage3$, $newData3$	–
$comLogic3$	$round3$, $selectOut3$, $notFirstTime3$, $doutValidR3$,	$selectOut3$, $notFirstTime3$, $doutValid$	–
$waitRoundDinValid3$	$doutValidR2$	$dinValidR3$	–
$incrementRound3$	$round3$	$round3$	–
$readRoundInputs3$	$doutR2$, $keyOutR2$, $encryptOutR2$	$dinR3$, $keyInR3$, $encryptInR3$	–
$rstPrevDoutValid3$	–	$doutValidR2$	–
$nShifts3$	$round3$	$encryptShift3$, $decryptShift3$	–
$keyShiftsDesPc23$	$keyInR3$, $keyShiftedMux3$, $encryptShift3$, $decryptShift3$	$keyShiftedMux3$, $pc2Key3$	$keyShiftedMux3$
$desEp3$	$dinR3$, $pc2Key3$	$dataEp3$	–
$desSboxPbox3$	$dataEp3$, $dinR3$	$DESpboxXored3$	–
$waitOutRead3$	$doutValidR3$	–	–
$wrRoundOutputs3$	$DESpboxXored3$, $keyShiftedMux3$, $encryptInR3$,$dinR3$	$doutR3$, $keyOutR3$, $encryptOutR3$	–
$desFp3$	$doutR3$	$dout$	–
$parityKey3$	$keyOutR3$	$keyOut$	–
$setDoutValid3$	–	$doutValid$	–
$assertToRead3$	–	$writeData3$	–
$waitReadDone3$	$writeDone3$	–	–

Tabela 8.33: Implementação *hardware/software* do sistema de criptografia DES: estimativa, para cada estado programa atribuído às partições HW2 a HW4, das variáveis lidas, escritas e que precisam dum multiplexador na entrada.

Variável	***Largura*** *(bits)*	***Número de elementos***	***areaHWvar*** *(portas) (lógicas)*
SW			
$valRead$	32	1	–
$val2send$	32	1	–
HW1			
din	64	1	384
$keyIn$	64	1	384
$encryptIn$	1	1	6
$dinValid$	1	1	6
$doutIp$	64	1	384
$keyOutPc1$	56	1	336
$dinMux0$	64	1	384
$keyInMux0$	56	1	336
$encryptMux0$	1	1	6
$dinValidMux0$	1	1	6
$selectIn0$	1	1	6
HW4			
$dout$	64	1	384
$keyOut$	64	1	384
$doutValid$	1	1	6
$writeDone3$	1	1	6
HW1 a HW4			
$resetFf0,1,2,3$	1	1	6
$newData0,1,2,3$	1	1	6
$writeData0,1,2,3$	1	1	6
$startFsm0,1,2,3$	1	1	6
$round0,1,2,3$	4	1	24
$notFirstTime0,1,2,3$	1	1	6
$selectOut0,1,2,3$	1	1	6
$readNewData0,1,2,3$	1	1	6
$readPrevStage0,1,2,3$	1	1	6
$encryptShift0,1,2,3$	2	1	12
$decryptShift0,1,2,3$	2	1	12
$keyShiftedMux0,1,2,3$	56	1	336
$pc2Key0,1,2,3$	48	1	288
$dinR0,1,2,3$	64	1	384
$keyInR0,1,2,3$	56	1	336
$encryptInR0,1,2,3$	1	1	6
$dinValidR0,1,2,3$	1	1	6
$dataEp0,1,2,3$	48	1	288
$DESpboxXored0,1,2,3$	32	1	192
$doutR0,1,2,3$	64	1	384
$keyOutR0,1,2,3$	56	1	336
$encryptOutR0,1,2,3$	1	1	6
$doutValidR0,1,2,3$	1	1	6

Tabela 8.34: Implementação *hardware/software* do sistema de criptografia DES: estimativa para as métricas associadas às variáveis da descrição do sistema.

para 4.56 ou 5.22 Mbits/s. Convém realçar que, na maior parte das aplicações do algoritmo DES, o débito de amostras processadas é bem mais importante que a latência da primeira amostra processada. Outra conclusão a retirar é a de que um aumento de 33 para 50 MHz na frequência de funcionamento do *hardware* faz com que o débito suba de 4.56 para 5.22 Mbits/s, ou seja, a frequência do *hardware* tem uma influência reduzida sobre o desempenho da implementação *hardware/software*. Este resultado é consequência de o desempenho da implementação ser influenciado de forma determinante pela comunicação entre *hardware* e *software*, que praticamente não depende da frequência do *hardware*.

	Solução *hardware/software* com 4 estágios de *pipeline*, código não optimizado, *Windows* 2000, P200 e plataforma EDgAR-2 a 33 MHz		
Métrica	***Estimativa***	***Medição***	***Precisão (%)***
Latência da (de)cifragem (μs)	57.0	55.6	97
Débito da (de)cifragem *(Mbits/s)*	3,76	3.60	96
	Solução *hardware/software* com 4 estágios de *pipeline*, código optimizado relativamente ao tempo de execução, *Windows* 2000, P200 e EDgAR-2 a 33 MHz		
Métrica	***Estimativa***	***Medição***	***Precisão (%)***
Latência da (de)cifragem (μs)	46.0	45.2	98
Débito da (de)cifragem *(Mbits/s)*	4.67	4.56	98
	Solução *hardware/software* com 4 estágios de *pipeline*, código optimizado relativamente ao tempo de execução, *Windows* 2000, P200 e EDgAR-2 a 50 MHz		
Métrica	***Estimativa***	***Medição***	***Precisão (%)***
Latência da (de)cifragem (μs)	40.5	39,6	98
Débito da (de)cifragem *(Mbits/s)*	5.08	5.22	97

Tabela 8.35: Implementação *hardware/software* do sistema de criptografia DES: estimativa e medição das métricas de desempenho.

De acordo com os resultados da tabela 8.35 verifica-se um grau de fidelidade de 100% entre as estimativas e as medições do desempenho. Ou seja, para qualquer par de soluções a diferença entre as estimativas do desempenho segue a mesma tendência que a diferença entre as respectivas medições. Contudo, como o número de amostras disponíveis é reduzido, a fidelidade das estimativas está sujeita a uma margem de erro elevada. A precisão das estimativas também é muito elevada, variando entre 96 e 98%, o que seria de esperar em estimativas efectuadas sobre uma descrição refinada com pormenores da implementação.

O factor de optimização relativamente ao tempo de execução em *software* (λ_T), que se empregou nas estimativas, foi 0.22845. Este valor foi calculado a partir do desempenho obtido com a simulação da solução totalmente em *software*, utilizando código optimizado e não optimizado (tabela 8.30).

Com base no valor 4.18 Mbits/s estimado para o débito da melhor solução obtida pelo algoritmo de partição (tabela 8.30) e no valor 4.56 (5.22) Mbits/s medido na implementação *hardware/software* com um relógio de *hardware* a 33 (50) MHz (tabela 8.35), resulta que a

melhor solução obtida pelo algoritmo de partição atinge 92% (80%) do desempenho da implementação *hardware/software* optimizada. Estes valores mostram que a solução obtida pelo algoritmo de partição possui qualidade.

Espaço ocupado pelo caminho de dados

Analisando a informação da tabela 8.36 constata-se que o caminho de dados de qualquer das partições de *hardware* ocupa um espaço inferior às 13100 portas lógicas permitidas: o caminho de dados da partição $HW1$ a $HW4$ ocupa 10K, 7.3K, 6.7K e 8K portas lógicas equivalentes. Pode ainda verificar-se que não seria possível implementar duas passagens em cada partição, dado que isso implicaria usar praticamente o dobro da lógica actualmente utilizada com uma passagem, o que ultrapassa a lógica disponível por FPGA. A medição do espaço obtém-se a partir da informação fornecida pelas ferramentas de síntese do fabricante das FPGAs. Além da medição e da estimativa do espaço por partição, a tabela 8.36 contém ainda a estimativa das parcelas que contribuem para esse espaço: as unidades funcionais, os elementos de armazenamento, os elementos de interligação e os recursos de interface. A precisão das estimativas do espaço ocupado pelo caminho de dados varia entre 92 e 99%, o que constitui um resultado muito bom.

	Estimativa			
Métrica	***p = HW1***	***p = HW2***	***p = HW3***	***p = HW4***
$areaEPrograma(CD(p))$	4225	3853	3853	3855
$areaVars(CD(p))$	4902	2664	2664	3444
$areaInterface(CD(p))$	1040	184	184	381
$areaMuxesExtra(CD(p))$	15	7	7	16
areaHW(CD(p))	***10182***	***6708***	***6708***	***7696***
	Medição			
Métrica	***p = HW1***	***p = HW2***	***p = HW3***	***p = HW4***
areaHW(CD(p))	***10029***	***7311***	***6694***	***8049***
Precisão (%)	***98***	***92***	***99***	***96***

Tabela 8.36: Implementação *hardware/software* do sistema de criptografia DES: estimativa e medição, para cada partição de *hardware*, das métricas envolvidas no cálculo do espaço ocupado pelo caminho de dados dessas partições.

Espaço ocupado pela unidade de controlo

Como se afirmou no exemplo da convolução, a medição do espaço ocupado pela unidade de controlo das partições de *hardware* presentes na implementação do sistema de criptografia é obtida de forma indirecta a partir de duas métricas fornecidas pelas ferramentas de síntese dos CPLDs: o número de macro-células e o número de produtos usados (tabela 8.37). Em ambas as estratégias, o espaço da unidade de controlo das partições de *hardware* é inferior às 2400 portas lógicas disponíveis num CPLD da arquitectura alvo.

Segundo o modelo de estimação desenvolvido e a menos dum termo fixo, o espaço ocupado

pela unidade de controlo duma partição coincide com o espaço da máquina de estados que coordena o caminho de dados dessa partição e o espaço ocupado pela máquina de estados inclui o registo de estado, a lógica de controlo e a lógica do próximo estado.

	Medição			
Métrica	***p=HW1***	***p=HW2***	***p=HW3***	***p=HW4***
Macro-células	62	62	62	62
Produtos	95	97	97	97
Portas lógicas equivalentes calculadas com base no número de macro-células usadas	1350	1350	1350	1350
Portas lógicas equivalentes calculadas com base no número de produtos usados	425	435	435	435

Tabela 8.37: Implementação *hardware/software* do sistema de criptografia DES: valores medidos, para cada partição de *hardware*, das métricas relacionadas com o espaço ocupado pela unidade de controlo dessas partições.

As tabelas 8.38, 8.39 e 8.40 contêm as métricas que permitem calcular o valor do espaço ocupado pela unidade de controlo das partições de *hardware* presentes na implementação *hardware/software* do sistema de criptografia:

- os sinais de selecção do multiplexador a colocar na entrada das variáveis escritas pelos estados programa (tabela 8.38);
- o número de estados ($nE(o)$), de sinais de controlo gerados ($nSC(o)$), de activações dos sinais de controlo ($nASC(o)$), de sinais de estado lidos ($nSE(o)$) e de leituras dos sinais de estado ($nLSE(o)$) necessários à secção de máquina de estados que coordena o funcionamento do estado o (tabela 8.39);
- as métricas $nSCmux(p)$ e $nASCmux(p)$, que representam o contributo dos sinais de selecção dos multiplexadores para as métricas da partição p (tabela 8.40);
- as métricas $nEpart(p)$ a $nLSEpart(p)$ relativas a estados programa introduzidos na partição p para gerir as mudanças de partição no fluxo de controlo (tabela 8.40).

A partir das métricas anteriores obtêm-se as métricas $nE(p)$ a $nLSE(p)$ ao nível da partição, o espaço ocupado pelo registo de estado, pela lógica de controlo, pela lógica do próximo estado, pela máquina de estados e finalmente pela unidade de controlo da partição ($areaHW(UC(p))$ na tabela 8.40).

Através da tabela 8.40 pode comparar-se a estimativa com a medição das métricas $nE(p)$ a $nLSE(p)$. Verifica-se que apenas para o número de estados $nE(p)$ e casualmente para o número de activações dos sinais de controlo $nASC(p)$, a precisão das estimativas é boa: 86 a 93% para $nE(p)$ e 89 a 93% para $nASC(p)$. Para as outras métricas a precisão situa-se

no intervalo 40 a 57%. Convém explicar que este fraco resultado se deve a que o processo de estimação das métricas $nSC(o)$, $nASC(o)$, $nSE(o)$ e $nLSE(o)$ foi relaxado intencionalmente. Dado que a influência destas métricas sobre o espaço da unidade de controlo é reduzida, aumentar significativamente a complexidade da sua estimação não seria vantajoso para a relação entre a qualidade das estimativas e o tempo de cálculo. Em contraste com as métricas anteriores, o número de estados afecta de forma determinante o espaço da unidade de controlo, razão pela qual se desejam estimativas mais precisas para a métrica $nE(o)$ (logo $nE(p)$). Situação comprovada pela qualidade das estimativas obtidas para $nE(p)$.

Variável	***HW1***	***HW2***	***HW3***	***HW4***
$dinMux0$	1	0	0	0
$keyInMux0$	1	0	0	0
$encryptMux0$	1	0	0	0
$dinValidMux0$	1	0	0	0
$doutValid$	0	0	0	1
$writeDone3$	0	0	0	1
$startFsm0$	1	0	0	0
$startFsm1$	0	1	0	0
$startFsm2$	0	0	1	0
$startFsm3$	0	0	0	1
$keyShiftedMux0$	1	0	0	0
$keyShiftedMux1$	0	1	0	0
$keyShiftedMux2$	0	0	1	0
$keyShiftedMux3$	0	0	0	1
$dinValidR1$	0	1	0	0
$dinValidR2$	0	0	1	0
$dinValidR3$	0	0	0	1
$doutValidR0$	1	0	0	0
$doutValidR1$	0	1	0	0
$doutValidR2$	0	0	1	0
$doutValidR3$	0	0	0	1
Total de sinais de selecção	***7***	***4***	***4***	***6***

Tabela 8.38: Implementação *hardware/software* do sistema de criptografia DES: estimativa dos sinais de selecção do multiplexador a colocar na entrada das variáveis escritas pelos estados programa atribuídos às partições de *hardware*.

Comparando as estimativas da tabela 8.40 para o espaço da unidade de controlo com as duas medidas indirectas incluídas na tabela 8.37, verifica-se que o número de produtos fornece medições que estão mais de acordo com as estimativas implementadas do que as medições obtidas através do número de macro-células. Esta conclusão confirma os resultados do exemplo da convolução. Considerando para medida do espaço da unidade de controlo aquela que se obtém com o número de produtos, a precisão das estimativas de $areaHW(UC(p))$ varia entre 89 e 96%.

Correcção da solução

O último objectivo da implementação do sistema DES consistia em verificar a correcção da descrição do sistema. Para atingir este objectivo compararam-se os resultado obtidos em três

Estado programa	***nE***	***nSC***	***nASC***	***nSE***	***nLSE***
$resetLogic0$	1	7	7	1	1
$comLogic0$	2	1	1	0	0
$assertToWrite0$	1	1	1	0	0
$waitDinValid0$	5	2	6	1	1
$readInputs0$	5	2	6	1	1
$rstPrevDoutValid0$	4	4	8	0	0
$desIp0$	1	1	1	0	0
$desPc10$	1	1	1	0	0
$nShifts0$	1	0	0	0	0
$muxes0$	1	0	0	0	0
$keyShiftsDesPc20$	1	0	0	0	0
$desEp0$	2	1	1	0	0
$desSboxPbox0$	3	1	1	0	0
$waitOutRead0$	1	0	0	1	1
$wrRoundOutputs0$	1	3	3	0	0
$setRoundDoutValid0$	4	2	2	0	0
$resetLogic1,2$	1	8	8	1	1
$comLogic1,2$	2	1	1	0	0
$waitRoundDinValid1,2$	6	3	7	1	1
$incrementRound1,2$	3	1	1	0	0
$readRoundInputs1,2$	5	2	6	1	1
$rstPrevDoutValid1,2$	4	4	8	0	0
$nShifts1,2$	1	0	0	0	0
$keyShiftsDesPc21,2$	1	0	0	0	0
$desEp1,2$	2	1	1	0	0
$desSboxPbox1,2$	3	1	1	0	0
$waitOutRead1,2$	1	0	0	1	1
$wrRoundOutputs1,2$	1	3	3	0	0
$setRoundDoutValid1,2$	1	1	1	0	0
$resetLogic3$	1	8	8	1	1
$comLogic3$	2	1	1	0	0
$waitRoundDinValid3$	6	3	7	1	1
$incrementRound3$	3	1	1	0	0
$readRoundInputs3$	5	2	6	1	1
$rstPrevDoutValid3$	4	4	8	0	0
$nShifts3$	1	0	0	0	0
$keyShiftsDesPc23$	1	0	0	0	0
$desEp3$	2	1	1	0	0
$desSboxPbox3$	3	1	1	0	0
$waitOutRead3$	1	0	0	1	1
$wrRoundOutputs3$	1	3	3	0	0
$desFp3$	1	1	1	0	0
$parityKey3$	1	1	1	0	0
$setDoutValid3$	1	1	1	0	0
$assertToRead3$	1	1	1	0	0
$waitReadDone3$	1	0	0	1	1

Tabela 8.39: Implementação *hardware/software* do sistema de criptografia DES: estimativa, para cada estado programa da descrição, das métricas utilizadas no cálculo do espaço ocupado pela unidade de controlo das partições de *hardware*.

situações: a solução totalmente em *software*, a simulação em VHDL da descrição utilizada na implementação e a própria implementação de *hardware/software*. Obtendo-se valores iguais no processo de (de)cifragem para as três situações, garante-se em boa medida que a descrição do sistema de criptografia está correcta. O correcto funcionamento do algoritmo DES obriga a que ao decifrar um valor $Cout$, resultante de cifrar o valor Vin com uma chave Kin, com a mesma chave Kin, se obtenha o valor original Vin (tabela 8.41).

	Estimativa				***Medição***			
Métrica	***p = HW1***	***p = HW2***	***p = HW3***	***p = HW4***	***p = HW1***	***p = HW2***	***p = HW3***	***p = HW4***
$\sum_{o \in p} nE(o)$	34	31	31	35				
$\sum_{o \in p} nSC(o)$	26	25	25	28				
$\sum_{o \in p} nASC(o)$	38	37	37	40				
$\sum_{o \in p} nSE(o)$	4	4	4	5				
$\sum_{o \in p} nLSE(o)$	4	4	4	5				
$nEpart(p)$	0	0	0	0				
$nSCpart(p)$	0	0	0	0				
$nASCpart(p)$	0	0	0	0				
$nSEpart(p)$	0	0	0	0				
$nLSEpart(p)$	0	0	0	0				
$nSCmux(p)$, $nASCmux(p)$	7	4	4	6				
$nE(p)$	34	31	31	35	30	29	29	32
$nSC(p)$	33	29	29	34	21	20	20	22
$nASC(p)$	45	41	41	46	42	37	37	43
$nSE(p)$	4	4	4	5	7	10	10	12
$nLSE(p)$	4	4	4	5	7	10	10	12
areaRegEstado(p)	204	186	186	210				
areaLogicaCtl(p)	36	36	36	36				
areaLProxEstado(p)	15	15	15	20				
areaHWfsm(UC(p))	255	237	237	266				
areaHW(UC(p))	***405***	***387***	***387***	***416***	***425***	***435***	***435***	***435***
Precisão (%)	***95***	***89***	***89***	***96***				

Tabela 8.40: Implementação *hardware/software* do sistema de criptografia DES: estimativa, para cada partição de *hardware*, das métricas envolvidas no cálculo do espaço ocupado pela unidade de controlo dessas partições.

Chave de entrada	***Amostra a cifrar***	***Amostra cifrada, Amostra a decifrar***	***Amostra decifrada***
0b1a0d2a 332b2b19	ca2b5652 32ad54b7	e22750d4 ae59e505	ca2b5652 32ad54b7
050d0615 1915150c	e515ab29 1956aa5b	f2511162 8a64e45f	e515ab29 1956aa5b
0246034a 0c4a4a46	f28ad594 8cab552d	840916fb 59104372	f28ad594 8cab552d
01634125 46656563	79456aca 4655aa96	a90fba88 c49bc783	79456aca 4655aa96
40312052 23323231	bca2b565 232ad54b	9fafe6e1 d65cd8dd	bca2b565 232ad54b
20585069 51195958	de515ab2 91956aa5	baf803aa c0bd6885	de515ab2 91956aa5
502c6834 284c2c2c	ef28ad59 48cab552	071bf736 c4412255	ef28ad59 48cab552
2816341a 54665656	f79456ac a4655aa9	4c73d063 7a3781af	f79456ac a4655aa9

Tabela 8.41: Implementação *hardware/software* do sistema de criptografia DES: exemplo dos valores hexadecimais obtidos no processo de (de)cifragem.

8.3.5 Melhorar a Implementação *Hardware/Software*

Com o objectivo de melhorar o desempenho da implementação *hardware/software* do sistema de criptografia, substituiu-se o *pipeline* por paralelismo, ou seja, uma solução com quatro estágios de *pipeline* é alterada para uma solução com quatro sub-sistemas sem *pipeline*, mas que funcionam em paralelo. O modelo PSMfg que descreve o sistema DES composto por quatro sub-sistemas operando em paralelo e sem *pipeline* é ilustrado na figura 8.30. Deste modo, usufrui-se da vantagem que constitui a comunicação simultânea entre *software* e as várias partições de *hardware*. Na solução com *pipeline*, o processamento duma amostra implica 18 leituras de *hardware* para *software* e 8 operações de escrita do *software* para *hardware*, enquanto na solução com paralelismo, este número de transacções permite processar quatro amostras. Outra vantagem é a redução da comunicação entre partições de *hardware* adjacentes. Esta vantagem resulta de as variáveis lidas e escritas por uma partição de *hardware*, com excepção das variáveis que representam as entradas e saídas de cada sub-sistema, estarem todas atribuídas a essa partição (tabela 8.42). O tempo de processamento por amostra é assim reduzido devido à eliminação de comunicação entre passagens do sistema DES, ou seja, entre estágios na solução com *pipeline*.

Comparando as estimativas de desempenho incluídas na tabela 8.43 com as estimativas da tabela 8.35, verifica-se que a implementação do sistema de criptografia DES com quatro sub-sistemas, operando em paralelo e sem *pipeline*, apresenta uma melhoria de desempenho muito significativa em relação à implementação com quatro níveis de *pipeline*. Para os dois casos analisados, a melhoria de desempenho situa-se entre 152 e 235%, o que equivale a dizer que o débito passa de 4.67 (5.08) Mbits/s para 13.9 (17.0) Mbits/s, quando a plataforma EDgAR-2 funciona a 33 (50) MHz.

Por outro lado, o espaço ocupado pelas partições de *hardware* sofre um aumento entre 9% e 64% em relação à implementação com quatro níveis de *pipeline*. Embora o espaço ocupado por partição seja penalizado, este espaço mantém-se inferior aos recursos disponíveis nos componentes da arquitectura alvo. Como mostra a tabela 8.44, a estimativa do espaço ocupado pelo caminho de dados de cada partição de *hardware* $HW1$ a $HW4$ sobe para 11027 portas lógicas equivalentes. Embora superior, o valor estimado para o espaço ocupado pela unidade de controlo das partições de *hardware* não é muito diferente do que se obteve na implementação com quatro níveis de *pipeline*.

Das estimativas apresentadas conclui-se que a implementação com quatro sub-sistemas, operando em paralelo e sem *pipeline*, é exequível e atinge um desempenho muito superior ao da implementação com quatro níveis de *pipeline*.

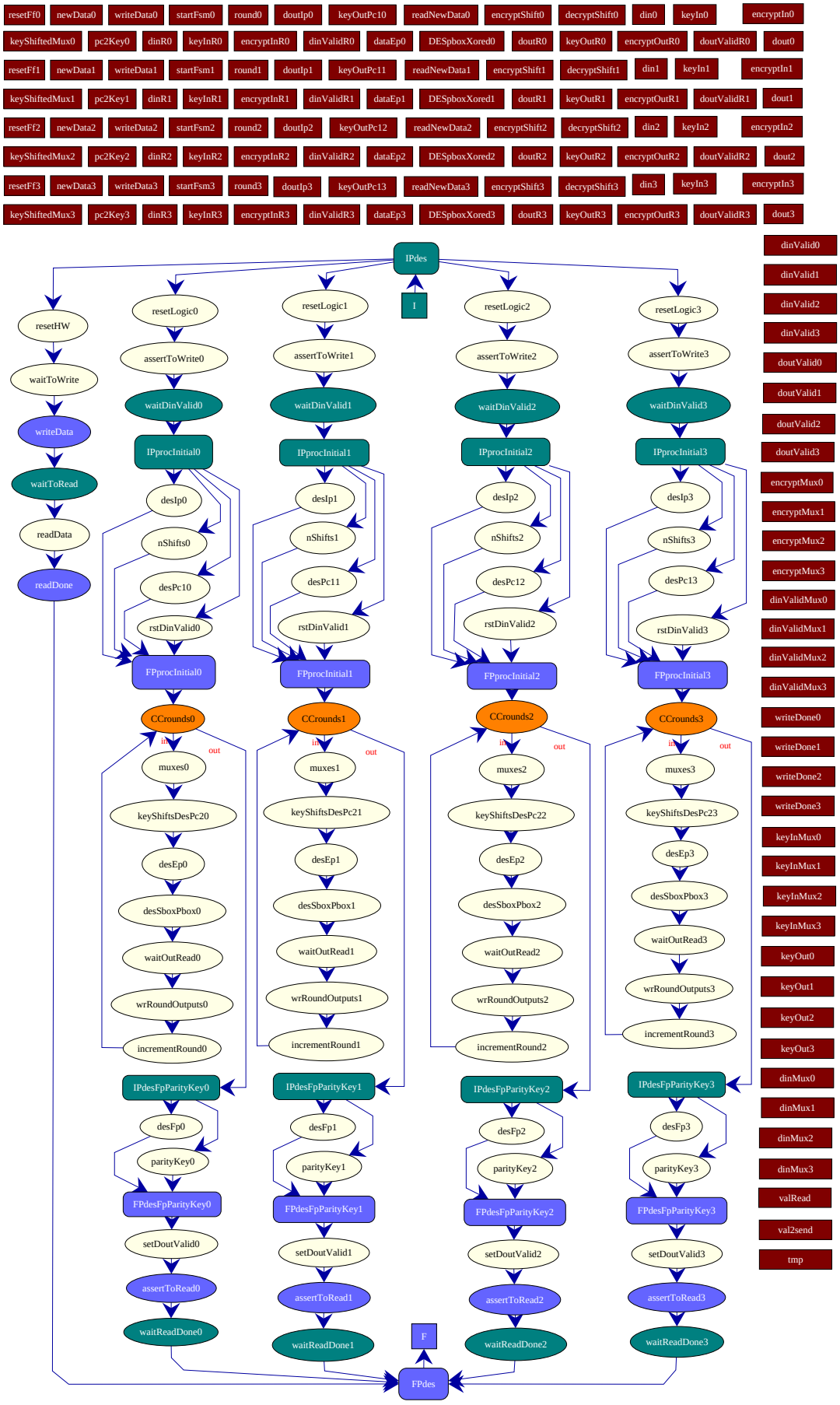


Figura 8.30: Descrição, com o meta-modelo PSMfg, do sistema de criptografia DES quando implementado com quatro sub-sistemas operando em paralelo e sem *pipeline*.

Estado programa	***Variáveis lidas***	***Variáveis escritas***	***Variáveis com multiplexador***
$resetHW$	–	$resetFf_{0,1,2,3}$ (2 x), $startFsm_{0,1,2,3}$	–
$waitToWrite$	$readNewData_{0,1,2,3}$ (2 x), $valRead$ (3 x)	$valRead$ (4 x)	–
$writeData$	$val2send$ (52 x), tmp (2 x)	$din_{0,1,2,3}$, $keyIn_{0,1,2,3}$, $encryptIn_{0,1,2,3}$, $dinValid_{0,1,2,3}$, $val2send$ (53 x), tmp (2 x)	–
$waitToRead$	$writeData_{0,1,2,3}$ (2 x), $valRead$ (3 x)	$valRead$ (4 x)	–
$readData$	$valRead$ (50 x), $dout_{0,1,2,3}$, tmp (4 x)	$valRead$ (52 x), tmp (4 x)	–
$readDone$	–	$writeDone_{0,1,2,3}$	–
$resetLogic_i$ (‡)	–	$doutValidR_i$, $newData_i$, $dinValidR_i$, $writeData_i$, $startFsm_i$, $readNewData_i$	
$assertToWrite_i$	–	$readNewData_i$	–
$waitDinValid_i$	$dinValid_i$	–	–
$desIp_i$	din_i	$doutIp_i$	–
$nShifts_i$	$round_i$	$encryptShift_i$, $decryptShift_i$	–
$desPc1_i$	$keyIn_i$	$keyOutPc1_i$	–
$rstDinValid_i$	–	$dinValid_i$	–
$CCrounds_i$	$round_i$	–	–
$muxes_i$	$doutIp_i$, $doutR_i$, $keyOutPc1_i$, $keyOutR_i$, $encryptIn_i$, $encryptOutR_i$, $dinValid_i$, $doutValidR_i$	$dinMux_i$, $keyInMux_i$, $encryptMux_i$, $dinValidMux_i$	$dinMux_i$, $keyInMux_i$, $encryptMux_i$, $dinValidMux_i$
$keyShiftsDesPc2_i$	$encryptShift_i$, $decryptShift_i$, $keyShiftedMux_i$, $keyInMux_i$	$pc2Key_i$, $keyShiftedMux_i$	$keyShiftedMux_i$
$desEp_i$	$dinMux_i$, $pc2Key_i$	$dataEp_i$	–
$desSboxPbox_i$	$dataEp_i$, $dinMux_i$	$DESpboxXored_i$	–
$waitOutRead_i$	$doutValidR_i$	–	–
$wrRoundOutputs_i$	$DESpboxXored_i$, $keyShiftedMux_i$, $dinMux_i$, $encryptMux_i$	$doutR_i$, $keyOutR_i$, $encryptOutR_i$, $doutValidR_i$	–
$incrementRound_i$	$round_i$	$round_i$	–
$desFp_i$	$doutR_i$	$dout_i$	–
$parityKey_i$	$keyOutR_i$	$keyOut_i$	–
$setDoutValid_i$	–	$doutValid_i$	–
$assertToRead_i$	–	$writeData_i$	–
$waitReadDone_i$	$writeDone_i$	–	–

(‡) Em qualquer estado programa cujo nome inclui o subscrito i, $0 \leqslant i \leqslant 3$.

Tabela 8.42: Sistema de criptografia DES quando implementado com quatro sub-sistemas operando em paralelo e sem *pipeline*: estimativa, para cada estado programa, das variáveis lidas, escritas e que precisam dum multiplexador na entrada.

Solução com 4 sub-sistemas operando em paralelo e sem *pipeline*, código optimizado relativamente ao tempo de execução, *Windows* 2000, P200 e EDgAR-2		
	EDgAR-2 a 33 MHz	
Métrica	***Estimativa***	***Melhoria*** (%)
Latência da (de)cifragem (μs)	18.2	152
Débito da (de)cifragem *(Mbits/s)*	13.9	198
	EDgAR-2 a 50 MHz	
Métrica	***Estimativa***	***Melhoria*** (%)
Latência da (de)cifragem (μs)	15.1	168
Débito da (de)cifragem *(Mbits/s)*	17.0	235

Tabela 8.43: Sistema de criptografia DES quando implementado com quatro sub-sistemas operando em paralelo e sem *pipeline*: estimativa das métricas de desempenho.

	Estimativa
Métrica	***p = HW1,2,3,4***
$areaEPrograma(CD(p))$	4066
$areaVars(CD(p))$	5658
$areaInterface(CD(p))$	1283
$areaMuxesExtra(CD(p))$	20
areaHW(CD(p))	***11027***

Tabela 8.44: Sistema de criptografia DES quando implementado com quatro sub-sistemas operando em paralelo e sem *pipeline*: estimativa, para cada partição de *hardware*, das métricas utilizadas no cálculo do espaço ocupado pelo caminho de dados dessas partições.

8.4 Resumo e Conclusões

Os dois exemplos analisados neste capítulo pertencem à classe dos sistemas embebidos predominantemente de fluxo de dados, são sistemas medianamente complexos pois possuem uma dimensão da ordem de grandeza dos recursos de *hardware* disponíveis na arquitectura alvo e complementam-se um ao outro: enquanto a convolução está vocacionada para uma implementação em *software*, porque exige uma capacidade de armazenamento de informação muito superior à disponível em *hardware* e executa operações sobre valores reais, o sistema de criptografia tem como implementação preferencial o *hardware*, porque exige um espaço para armazenamento de informação compatível com os recursos de *hardware* e como envolve muitas operações ao nível do bit (permutações), o algoritmo apresenta um desempenho superior em *hardware*.

O tempo de cálculo necessário para atingir a melhor solução de partição revelou-se elevado. Como depende do número de objectos do sistema, o tempo de cálculo torna-se tanto maior quanto maior for a dimensão do sistema (tabela 8.45). Este inconveniente é minorado pelo facto de as primeiras pesquisas dum processo de partição, executadas num intervalo da ordem de grandeza de dez minutos, encontrarem normalmente uma solução de partição próxima da melhor solução obtida com a totalidade das pesquisas. A tabela 8.45 mostra que o tempo de

cálculo também depende das características do próprio sistema em partição.

O tempo de cálculo é afectado de forma determinante pelo tempo envolvido na estimação do tempo de execução do sistema. Para reduzir o tempo despendido na estimação do tempo de execução dum sistema, deve optimizar-se a implementação desta estimativa, a qual é baseada na invocação intensiva de rotinas recursivas.

Número de objectos do sistema (nObj)	***Tempo de cálculo*** (*minutos*)
45	2
177	55
185	66
217	88
372	100

Tabela 8.45: Exemplos do tempo de cálculo necessário para efectuar 10000 iterações do processo de partição dum sistema, numa máquina com um processador P600 e *Windows* 2000.

Para quantificar a complexidade temporal do processo de partição comparou-se o comportamento do tempo de cálculo, em função do número de objectos do sistema (tabela 8.45), com o comportamento das funções $f1(nObj) = H1 * nObj$ e $f2(nObj) = H2 * nObj^2$, em que $nObj$ é o número de objectos do sistema e $H1, H2$ são constantes calculadas de modo a que, para $nObj = 177$, o tempo de cálculo medido e indicado pelas funções $f1$ e $f2$ seja igual (figura 8.31). Com a excepção da situação em que o número de objectos é 45, calcular $H1$ e $H2$ com qualquer outro valor de $nObj$ incluído na tabela 8.45, conduz a conclusões idênticas sobre a complexidade do processo de partição. A figura 8.31 mostra que a complexidade temporal do processo de partição iterativo é $\mathcal{O}(nObj)$, ou seja, o tempo de cálculo varia linearmente com a dimensão do sistema. Este resultado era esperado porque:

- o algoritmo de pesquisa tabu implementado possui a mesma estrutura do algoritmo PT discutido na secção 3.4 e a complexidade temporal deste algoritmo, $\mathcal{O}(nObj * nPart)$, resultou de o tamanho da vizinhança pesquisada em cada iteração ser aproximadamente $nObj * (nPart - 1)$, em que $nPart$ é o número de partições permitido;
- ao pesquisar apenas uma subvizinhança com tamanho $nObj$, mantendo-se a estrutura do algoritmo PT da secção 3.4, esperava-se que a complexidade temporal passasse de $\mathcal{O}(nObj * nPart)$ para $\mathcal{O}(nObj)$, reflectindo assim a alteração de $nObj * (nPart - 1)$ para $nObj$ no tamanho da vizinhança pesquisada.

A metodologia de partição também pode ser avaliada pelo apoio prestado à implementação dos sistemas, especialmente à síntese da interface entre partições. Neste momento, este apoio não está disponível automaticamente, mas como se dispõe de modelos de estimação detalhados, este apoio poderá ser disponibilizado sem grande dificuldade. Para gerar o código responsável

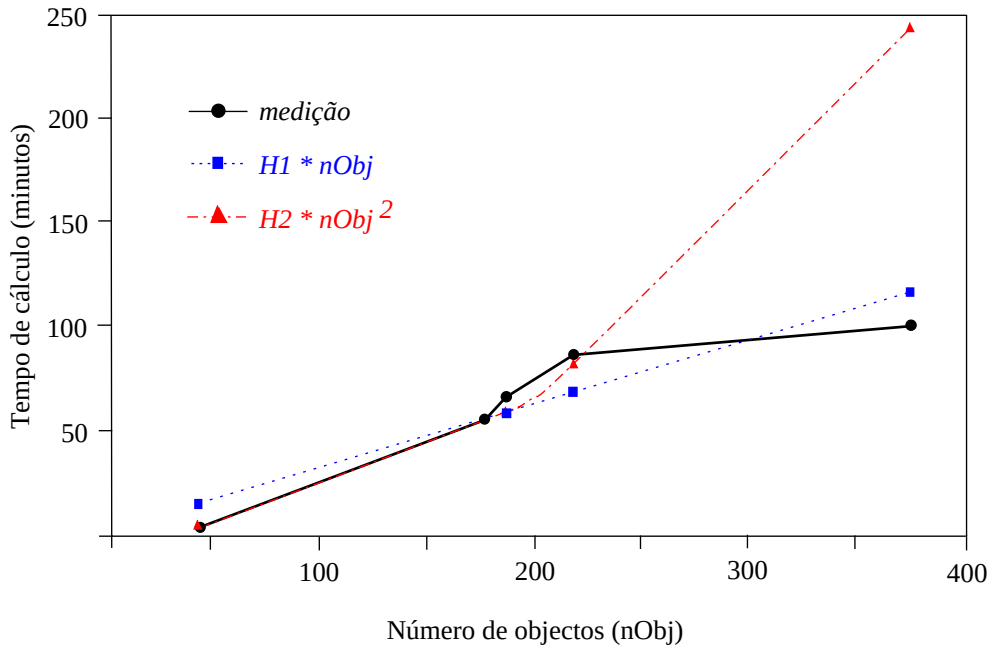


Figura 8.31: Método experimental utilizado para quantificar a complexidade temporal do processo de partição iterativo.

pela interface entre partições, é preciso recorrer à informação utilizada na estimação (i) do espaço ocupado pelos recursos da interface entre o caminho de dados das partições de *hardware* e o seu exterior (secção 7.3.1) e (ii) do tempo de comunicação entre partições (secção 7.4.3). Quando se discutiu a estimação do tempo de comunicação entre partições, apresentou-se o essencial das alterações a efectuar na descrição do sistema, de modo a adaptar as trocas de informação, descritas em alto nível, às características da arquitectura alvo.

No exemplo da convolução, a melhor solução obtida pelo algoritmo de partição atingiu um desempenho apenas inferior em 10% ao da solução teoricamente óptima. A mesma solução atingiu 72% do desempenho obtido com a implementação *hardware/software* optimizada. Dadas as optimizações incluídas na implementação, a solução de partição possui uma qualidade muito aceitável. A precisão das estimativas do desempenho do sistema variou entre 82 e 94%, enquanto a fidelidade foi de 83%. Devido às optimizações introduzidas na implementação *hardware/software*, as quais não são consideradas nos modelos de estimação, as estimativas para a solução de *software* revelaram-se mais precisas do que no caso da implementação *hardware/software*. A precisão das estimativas do espaço ocupado pelo caminho de dados das partições de *hardware* foi de 98%. A medida do espaço do caminho de dados pode ser (i) indicada pela ferramenta de síntese, (ii) calculada com base no número de CLBs usados ou (iii) calculada com base no número de *flip-flops*, de LUTs de 3 e de 4 entradas usados, mas foi a segunda alternativa aquela que proporcionou os melhores resultados. Por seu lado, a precisão das estimativas do espaço ocupado pela unidade de controlo das partições de *hardware*

foi de 91%. A medida do espaço ocupado pela unidade de controlo pode ser obtida a partir do número de macro-células ou do número de produtos usados, mas a segunda alternativa conduz a estimativas mais precisas.

No exemplo de criptografia, o desempenho da melhor solução conseguida no processo de partição atinge 80 a 92% do desempenho da implementação *hardware/software* optimizada. Este resultado, que é melhor do que o conseguido no exemplo da convolução, mostra que a solução obtida pelo algoritmo de partição é de qualidade. A precisão das estimativas do desempenho do sistema, para a solução totalmente em *software*, situou-se nos 92%. No caso da implementação *hardware/software*, a precisão das estimativas do desempenho sobe para valores entre 96 e 98% e atinge-se um grau de fidelidade de 100%. Estes valores são melhores do que os obtidos no exemplo da convolução porque o sistema DES apresenta uma complexidade inferior, o que permite ao modelo de estimação aproximar-se mais do sistema modelado. A precisão das estimativas do espaço ocupado pelo caminho de dados das partições de *hardware* variou entre 92 e 99%, um resultado muito bom e próximo do conseguido no exemplo da convolução.

Embora as estimativas de algumas das métricas que intervêm no cálculo do espaço da unidade de controlo das partições de *hardware* apresentem uma precisão baixa, ao contribuírem pouco para este espaço permitem que a precisão da estimativa do espaço ocupado pela UC se mantenha elevada. A métrica mais influente para o espaço da UC é o número de estados, razão pela qual se procurou que a sua estimativa fosse precisa, como mostra a precisão de 86 a 93% obtida. A precisão das estimativas do espaço ocupado pela unidade de controlo das partições de *hardware* variou entre 89 e 96%, valores muito próximos dos que se verificaram no exemplo da convolução. Tal como aconteceu no exemplo da convolução, o número de produtos forneceu uma medida mais próxima das estimativas calculadas.

A qualidade das soluções de partição obtidas automaticamente, embora muito aceitável porque atingiu entre 72 e 92% do desempenho da implementação *hardware/software* optimizada, é condicionada (i) pelas optimizações efectuadas aquando da implementação, (ii) pela granulosidade dos objectos que, ao ser demasiado fina produz uma relação comunicação/computação baixa quando ocorrem mudanças de partição no fluxo de controlo, (iii) por algumas limitações dos modelos de estimação e (iv) em termos absolutos, a qualidade é fortemente penalizada pelas limitações da arquitectura alvo considerada: comunicação excessiva entre *software* e *hardware*, intervenção do *software* em determinadas transacções não explicitadas e recursos de *hardware* bastante limitados, em espaço e em desempenho.

Verificou-se que o processo de partição gerou sempre uma solução que respeita os condicionalismos impostos ao recursos de *hardware*, o que denota um controlo adequado por parte das

funções de proximidade e de custo aplicadas.

Os parâmetros utilizados com o algoritmo de partição estão sumariados na tabela 8.46 e foram seleccionados de acordo com as seguintes recomendações:

- ◇ o número de iterações a efectuar durante o processo de partição é definido pela equação 6.11, aplicando uma constante $K2ni$ com valor 10;
- ◇ para percentagem de objectos a deslocar, quando se constrói a solução inicial das pesquisas, optou-se pelo valor de 20% recomendado;
- ◇ a recomendação a aplicar na selecção do número de pesquisas sem melhoria da solução de partição, ao fim do qual se deslocam os objectos menos vezes deslocados para uma partição seleccionada de forma aleatória, aquando da construção da solução inicial duma pesquisa, indica que ele deve ser tanto maior quanto maior for a diversificação que se quer introduzir na pesquisa; introduzir diversificação na construção da solução inicial de 25% das pesquisas, ou seja de 4 em 4 pesquisas, corresponde ao valor escolhido;
- ◇ a regra que ajuda a definir o número de iterações a efectuar, sem que haja melhoria da solução de partição, indica que ele não deve ser demasiado alto para evitar que se desperdicem iterações em torno dum mínimo local, nem demasiado baixo para aumentar a possibilidade de convergência para um mínimo local ou absoluto da função de custo; por outro lado, o seu valor deve depender do número de objectos do sistema; com base nestes princípios, escolheu-se o valor de 300 e 400 iterações para o exemplo da convolução e criptografia, respectivamente;

Dimensão do exemplo	***Convolução***	***Criptografia***
Número de partições	5	5
Número de objectos	217	372
Parâmetro		
Número de iterações	43400	74400
Número de iterações sem melhoria da solução de partição	300	400
Percentagem de objectos a deslocar na construção da solução inicial duma pesquisa	20	20
Número de pesquisas sem melhoria da solução de partição ao fim do qual se efectuam deslocamentos "aleatórios" para construir a solução inicial da próxima pesquisa	4	4
Validade do tabu aplicado aos deslocamentos	20	25
Validade do tabu aplicado aos deslocamentos inversos	18	22
Validade do tabu aplicado aos objectos	15	20
Terceira alternativa de deslocamento	deslocamento menos frequente ou com mais qualidade	deslocamento menos frequente

Tabela 8.46: Parâmetros aplicados no processo de partição com o algoritmo de pesquisa tabu nos exemplos estudados.

- para validade do tabu a aplicar aos objectos e aos deslocamentos recomenda-se um valor entre 5 e 10% do número de objectos do sistema; por outro lado, com um tipo de tabu mais restritivo deve aplicar-se uma validade menor; os valores escolhidos para a validade do tabu, definidos na tabela 8.46, variam entre 5 e 9%; como o tabu aplicado aos objectos é mais restritivo do que o tabu dos deslocamentos, seleccionou-se um valor menor para a validade do tabu aplicado aos objectos;

- a estratégia seguida na terceira alternativa de deslocamento considerada pelo algoritmo PT tem pouca influência sobre a qualidade da solução de partição final, isto porque se constatou que, tal como foi implementado, o algoritmo recorre muito poucas vezes a esta alternativa; deste modo, sem ter sido efectuado um estudo comparativo das diferentes estratégias, optou-se por uma das estratégias mais consensuais: efectuar o deslocamento menos frequente.

Capítulo 9

Conclusões

Sumário

Este capítulo final sintetiza as ideias fortes que resultaram do trabalho desenvolvido. Apresentam-se conclusões sobre (i) a modelação relevante para a tarefa de partição, (ii) a arquitectura alvo considerada na partição, (iii) os algoritmos de partição seleccionados, (iv) as funções de proximidade e de custo escolhidas para coordenar a evolução do processo de partição e (v) os estimadores de métricas. Sumariam-se os resultados da avaliação da metodologia de partição e identificam-se os contributos do presente trabalho, no âmbito dos algoritmos de partição, da estimação de métricas, entre outros. Finalmente, apontam-se áreas para futura investigação e desenvolvimento relacionadas com o trabalho realizado.

Conteúdo

9.1 Conclusões

Modelação

Da revisão sobre o tipo de modelação empregue com sistemas embebidos concluiu-se que os meta-modelos heterogéneos, ao integrarem características de vários meta-modelos, são uma boa solução para modelar sistemas embebidos complexos. Outra boa alternativa é a modelação dos sistemas através de várias vistas, aplicando-se a modelação mais adequada a cada vista dos sistemas e a cada fase do desenvolvimento desses sistemas, como acontece na abordagem MOOSE.

Para modelar os sistemas à entrada para o processo de partição seleccionou-se o meta-modelo heterogéneo PSM. A escolha deveu-se à adequação do PSM para descrever sistemas embebidos com alguma complexidade, uma vez que consegue modelar os aspectos mais importantes deste

tipo de sistema: hierarquia funcional, concorrência, transições de estado explicitas, complexidade ao nível das estruturas de dados e do controlo, indicação de fim dos blocos funcionais e a possibilidade de incluir tratamento de excepções. A favor da modelação com PSM pode ainda apontar-se o facto de ela ser intuitiva, o que facilita a tarefa do projectista, e de se poder optar pela HLL ou HDL mais familiar ao projectista ou mais adequada ao projecto. No caso presente optou-se por VHDL. A linguagem VHDL é mais limitada do que uma HLL, como o C, C++ ou Java, para representar estruturas de dados e comportamentos algoritmicamente muito complexos, mas facilita a incorporação nos modelos de requisitos não funcionais associados com desempenho, permite validar os modelos em relação ao cumprimento destes requisitos de desempenho e modela de forma mais elegante o paralelismo e a comunicação entre comportamentos concorrentes.

Metodologia de Partição Desenvolvida

A abordagem proposta para resolver o problema de partição, concretizada na ferramenta *parTiTool*, é do tipo funcional, inter-componentes e automática, por ser esta a estratégia mais adequada para obter soluções de partição com maior qualidade, com sistemas complexos e com qualquer número de componentes. Efectua-se a partição com o objectivo de atingir um desempenho pré-definido com os recursos disponíveis na arquitectura alvo, diferente da maioria das abordagens revistas em que se procura minimizar os recursos seleccionados. A arquitectura alvo considerada é constituída por um sistema hospedeiro e uma plataforma EDgAR-2. A organização da metodologia desenvolvida inclui um módulo de conversão de modelos, os algoritmos de partição construtivo e iterativo, as funções de proximidade e de custo e os estimadores de métricas.

Arquitectura Alvo

As maiores limitações da arquitectura alvo considerada no processo de partição são a disponibilidade dum espaço reduzido nos componentes de *hardware*, a ausência de memória em quantidade significativa na plataforma EDgAR-2 e a impossibilidade de estabelecer certas ligações, entre componentes de *hardware*, sem intervenção do *software*. Estas limitações afectam negativamente o desempenho das implementações. Entre os pontos fortes da arquitectura alvo, relacionados com a plataforma EDgAR-2, incluem-se a boa correspondência entre o meta-modelo usado na descrição dos sistemas e o meta-modelo de computação da plataforma, a possibilidade de estender a arquitectura e as potencialidades que fornece para a exploração do problema de partição em múltiplos componentes diferenciados.

Algoritmos de Partição

A partir do momento que se decidiu que o processo de partição incluía uma fase em que se constrói a solução inicial e outra fase em que sucessivamente se procura melhorar esta solução, não se exigia um algoritmo construtivo tão sofisticado como nos casos em que se aplica apenas um algoritmo para procurar a solução de partição óptima. Foi por esta razão que se optou por um algoritmo construtivo de fácil implementação e que apresenta um tempo de cálculo reduzido, como é o caso do algoritmo de crescimento de grupos. Embora seja um algoritmo simples, a sua capacidade para gerar soluções de partição com qualidade depende da função de proximidade envolvida.

De entre as alternativas ponderadas, o algoritmo iterativo seleccionado para melhorar as soluções de partição geradas pelo algoritmo de crescimento de grupos foi o algoritmo de pesquisa tabu. Relativamente ao algoritmo de *simulated annealing*, um dos algoritmos de optimização mais utilizados, o algoritmo de pesquisa tabu dispõe duma capacidade idêntica para atingir soluções de qualidade, exige um tempo de cálculo inferior e em termos de dificuldade de implementação e de utilização equivalem-se.

Os tipos de tabu implementados no algoritmo PT são aqueles que classificam como sendo tabu o deslocamento, o objecto deslocado e o deslocamento inverso. Para se poder desautorizar a classificação tabu, implementaram-se os critérios de aspiração por objectivo e por defeito. Concluiu-se que no início de cada pesquisa se deve limpar o historial de deslocamentos e de objectos deslocados, para evitar um efeito de dispersão provocado pela bonificação utilizada na segunda alternativa de deslocamento.

Para seleccionar um deslocamento, o algoritmo PT pesquisa apenas numa subvizinhança simples definida a partir do melhor deslocamento para cada objecto do sistema. Pesquisar apenas uma parcela de vizinhança e efectuar um único deslocamento por iteração reduz o tempo de cálculo, mas representa uma exploração do espaço de projecto mais incompleta. Deslocar uma percentagem de objectos, no início de cada pesquisa, contribui para o sucesso da fuga a mínimos locais da função de custo e a possibilidade de atribuir os objectos a deslocar a uma partição seleccionada aleatoriamente, constitui uma capacidade acrescida para evitar os mínimos locais.

O algoritmo de pesquisa tabu implementado dispõe de diversos mecanismos que favorecem a convergência para a solução de partição óptima e que ajudam a evitar mínimos locais da função de custo: (i) definir a subvizinhança com base no melhor deslocamento para cada objecto constitui uma estratégia de intensificação, (ii) a segunda alternativa de evolução do algoritmo, onde se aplica uma bonificação que favorece o deslocamento de objectos que não são deslocados regularmente, introduz diversificação na pesquisa, (iii) construir a solução inicial

duma pesquisa a partir da melhor solução encontrada é um forma de intensificação, enquanto deslocar os objectos menos vezes deslocados introduz diversificação, (iv) os métodos utilizados na escolha do deslocamento que gera a terceira alternativa de evolução do algoritmo também produzem efeitos de diversificação ou intensificação.

Funções de Proximidade e de Custo

A função de proximidade utilizada durante o processo de partição construtivo adapta-se ao tipo de objecto em atribuição e tem por objectivo atribuir cada objecto à partição (i) que apresente a maior intensidade de comunicação com esse objecto, (ii) em que o tempo de computação do objecto é menor e (iii) em que o espaço disponível é mais elevado. A função selecciona de forma prioritária a partição que conduz ao melhor desempenho e em segundo lugar a partição que esteja menos preenchida.

A função de custo seleccionada para orientar o processo de partição iterativo inspirou-se na função de custo da abordagem SpecSyn, por ser aquela que traduz os objectivos pretendidos: considerar como óptima uma solução de partição que respeita os condicionalismos impostos pela arquitectura alvo e atinge o desempenho exigido ao sistema. As alterações efectuadas na função da abordagem SpecSyn traduziram-se na introdução dum termo relacionado com o espaço ocupado pela unidade de controlo das partições de *hardware*, a eliminação dos termos relativos ao respeito pelo condicionalismo imposto ao número de componentes e à largura da ligação entre componentes e o tratamento do desempenho ao nível do sistema em vez do objecto.

Estimadores de Métricas

Verificou-se que as métricas mais frequentes nas abordagens ao problema de partição são o espaço ocupado em *hardware*, a largura das ligações entre componentes, o espaço que o código e os dados ocupam em memória e o tempo de execução em *hardware* e em *software*. Porque o objectivo da estimação de métricas é a possibilidade de comparar alternativas de partição, é mais importante dispor de estimativas com um grau de fidelidade elevado do que com uma precisão elevada. É de esperar que a uma precisão elevada corresponda um grau de fidelidade também elevado. Concluiu-se ainda que na generalidade das abordagens a estimação opera sobre uma representação do sistema em forma de grafo, a maioria assume um modelo de *hardware* com um caminho de dados e uma unidade de controlo e um modelo de *software* que exclui as optimizações de código efectuadas pelos compiladores e que estão relacionadas com as potencialidades dos estágios de *pipeline*, da superescalaridade do processador e da hierarquia de memória. Este procedimento é aceitável para a maioria dos problemas de partição que operam sobre sistemas embebidos. Constatou-se que uma boa parte das abordagens não

estima, ou não releva, métricas associadas à comunicação e que, para estimar o tempo de execução, é recorrente percorrer-se o grafo do sistema à procura do caminho com o maior tempo de execução acumulado.

Em primeiro lugar, a metodologia de estimação implementada pretendia obter estimativas precisas e em segundo lugar manter o tempo de cálculo o mais baixo possível. Para obter estimativas precisas desenvolveram-se modelos detalhados dos recursos utilizados na implementação, especialmente os modelos de *hardware* e de comunicação, e a estimação processa-se em dois níveis de abstracção (sistema e estado programa). Para a redução do tempo de cálculo contribuem a actualização incremental das estimativas e a estimação a dois níveis.

No nível de abstracção do estado programa obtêm-se estimativas de mais baixo nível aplicadas no cálculo das estimativas ao nível do sistema, os cálculos só são efectuados uma vez por cada sessão de partição e as estimativas são mais precisas. No nível do estado programa calculam-se estimativas para métricas relativas a cada objecto do sistema, tais como o tempo de computação em *software* e em *hardware*, o espaço ocupado por unidades funcionais, multiplexadores e variáveis, as variáveis lidas, escritas e que precisam dum multiplexador na entrada, e os estados que lêem e escrevem cada variável.

No nível de abstracção do sistema obtêm-se estimativas de nível superior, os cálculos são repetidos em cada iteração do processo de partição, as estimativas são menos precisas e, sempre que possível, efectua-se apenas a actualização das estimativas em vez de se refazerem todos os cálculos. No nível do sistema estima-se o desempenho do sistema e o espaço ocupado pelo caminho de dados e pela unidade de controlo das partições de *hardware*.

Foram apresentados dois modelos de *software*: um modelo simples, específico para a família de processadores Intel Pentium, que considera as optimizações efectuadas pelos compiladores sob a forma dum factor obtido por simulação e um modelo melhorado, genérico, em que as optimizações são traduzidas num número médio de ciclos por instrução obtido estaticamente. O modelo simples, o único implementado, é utilizado no cálculo dos tempos de execução em *software*.

O modelo de *hardware* é detalhado e intervém no cálculo do desempenho do sistema e do espaço ocupado pelas partições de *hardware*. Segundo este modelo, para o espaço duma partição contribuem o espaço do caminho de dados e o espaço da unidade de controlo. O caminho de dados considera as unidades funcionais, os elementos de armazenamento, os recursos de interligação e os recursos de interface com o exterior da partição. A menos duma constante, o espaço da unidade de controlo coincide com o espaço da máquina de estados associada ao caminho de dados da partição, ou seja, é composto pelo espaço do registo de estado, da lógica de controlo e da lógica do próximo estado. Os exemplos estudados confirmaram que o registo

de estado é o termo preponderante no espaço da máquina de estados, representando entre 60 e 80% do seu espaço.

O modelo de comunicação participa na estimação do desempenho do sistema e do espaço ocupado pelas partições de *hardware* e define os tempos de comunicação e os recursos envolvidos nessa comunicação. O modelo suporta o mecanismo de acesso a registo, por auscultação e por interrupção. Os tempos de comunicação não são recalculados em cada iteração do processo de partição, mas antes actualizados e apenas para os objectos afectados pelos objectos deslocados de partição.

No cálculo do desempenho dum sistema utiliza-se escalonamento explícito ao nível do estado programa mas não ao nível do sistema. Deste modo, reduz-se o tempo de cálculo e obtêm-se estimativas tendencialmente pessimistas. O desempenho dum sistema depende do tempo de computação dos estados programa, da atribuição dos objectos, do tempo de comunicação entre estados e variáveis, da frequência de execução dos estados e do sincronismo entre estados. Para estimar o desempenho dum sistema é necessário calcular o tempo de execução em construtores condicionais, ciclos, construtores paralelos e ciclos de espera. Para estimar o tempo de execução num ciclo de espera, o construtor de mais difícil tratamento, definiu-se um conjunto de cenários para o qual o cálculo do tempo de execução é automático e que, nos casos onde se verificam não-determinismos, gera uma estimativa para a pior situação. A implementação do cálculo do tempo de execução do sistema funciona de forma recursiva, percorre todos os fluxos (ou caminhos) do grafo e calcula (i) o tempo de execução no final de cada fluxo à custa do tempo de execução dos sub-fluxos que o compõem e (ii) o tempo de execução no final de cada sub-fluxo a partir do tempo de execução dos estados programa e das mudanças de partição que o integram.

Resultados da Avaliação da Metodologia de Partição

A metodologia de partição foi avaliada através de dois sistemas embebidos predominantemente de fluxo de dados, medianamente complexos e que se complementam um ao outro: a convolução está vocacionada para uma implementação em *software* e o sistema de criptografia tem como implementação preferencial o *hardware*.

O primeiro, e mais relevante, critério de avaliação da metodologia de partição é a qualidade das soluções de partição que consegue gerar. Tendo como referencial uma implementação *hardware/software* optimizada, o desempenho da melhor solução obtida pelo processo de partição atingiu 72 a 92% do desempenho dessa implementação optimizada, sendo superior no exemplo de criptografia. Este resultado, considerado muito aceitável, não é melhor precisamente porque se efectuaram optimizações na implementação, porque se usaram objectos de

pequena dimensão que apresentam uma relação comunicação/computação desfavorável e porque os modelos de estimação, especialmente o modelo de *software* simples, mantêm algumas limitações. As diversas experiências efectuadas com os exemplos estudados resultaram sempre em soluções de partição exequíveis, ou seja, soluções que respeitam os condicionalismos impostos pela arquitectura alvo, o que denota um controlo adequado por parte das funções de proximidade e de custo aplicadas.

O segundo aspecto avaliado na metodologia desenvolvida foi a precisão e a fidelidade das estimativas do desempenho e do espaço ocupado em *hardware*. A precisão das estimativas do desempenho do sistema situou-se no intervalo entre 82 e 98%, sendo superior no exemplo de criptografia devido a uma complexidade menor. Uma fidelidade entre 83 e 100%, praticamente coincidente com o intervalo de variação da precisão, traduz-se num elevado grau de confiança nas estimativas obtidas. Já a precisão das estimativas do espaço ocupado pelo caminho de dados das partições de *hardware* variou entre 92 e 99%, sendo idêntica nos dois exemplos estudados. Obtiveram-se melhores resultados quando se utilizou como medida do espaço do caminho de dados um valor calculado com base no número de CLBs usados. Por seu lado, a precisão das estimativas do espaço ocupado pela unidade de controlo das partições de *hardware* variou de 89 a 96%, sendo os valores obtidos com os dois exemplos muito próximos. Utilizar como medida do espaço ocupado pela unidade de controlo um valor obtido a partir do número de produtos usados, traduziu-se numa precisão superior. Verificou-se ainda que o espaço da unidade de controlo depende maioritariamente do espaço do registo de estado, que por sua vez é proporcional ao número de estados.

Como os valores da precisão para os dois exemplos são muito próximos, a sua consistência confere confiança às estimativas obtidas para as várias métricas. No conjunto de todas as métricas e exemplos, a precisão e a fidelidade das estimativas situaram-se sempre acima de 82%, o que constitui um resultado muito compensador.

O terceiro aspecto avaliado foi o desempenho da ferramenta que concretiza a metodologia de partição. Este desempenho revelou-se baixo, sobretudo porque não se optimizou a estimação do tempo de execução do sistema. A complexidade temporal $\mathcal{O}(nObj)$, prevista para o algoritmo de partição iterativo, foi comprovada experimentalmente. Como o tempo de cálculo varia linearmente com o número de objectos do sistema, em sistemas de grande dimensão o tempo necessário para encontrar a melhor solução de partição é elevado. Contudo, na maior parte dos casos, as primeiras pesquisas do processo de partição geram uma solução de qualidade.

Por fim, importava identificar o apoio prestado pela metodologia de partição à implementação de sistemas. Embora não tenha sido implementada, a síntese automática da interface entre

partições poderá ser disponibilizado sem grande dificuldade, recorrendo à informação envolvida na estimação do espaço ocupado pelos recursos da interface entre partições e do tempo de comunicação entre partições.

Ao conceber e implementar o núcleo duma metodologia de partição que gera soluções de qualidade passíveis de serem implementadas na arquitectura alvo pré-definida, foi atingido o objectivo maior a que se propunha esta tese.

9.2 Contributos do Trabalho

Esta secção resume os contributos que resultaram do trabalho desenvolvido. Começa por enunciar-se os contributos considerados mais relevantes:

- ◇ efectuar partição tendo por alvo uma arquitectura composta por pares de componentes (FPGA,CPLD) que implementam directamente o par (CD,UC) de cada partição de *hardware* do sistema em projecto;
- ◇ ter desenvolvido o meta-modelo PSMfg, bem como a ferramenta de edição que lhe está associada, para representar os sistemas durante o processo de partição; este meta-modelo garante uma óptima continuidade com o meta-modelo PSM utilizado na interacção entre o processo de partição e o resto da metodologia de desenvolvimento.

No âmbito dos algoritmos de partição identificaram-se os seguintes contributos:

- ◇ concepção duma função de proximidade que permite ao algoritmo de crescimento de grupos (i) construir soluções de elevada qualidade a aplicar no processo de partição iterativo e (ii) reduzir o tempo de cálculo necessário ao processo iterativo;
- ◇ introdução de elementos novos no algoritmo de pesquisa tabu, nomeadamente (i) efectuar as pesquisas apenas numa subvizinhança da solução actual, de modo a reduzir o tempo de cálculo necessário à execução das iterações, (ii) dispor de mais estratégias de evolução quando não há soluções de qualidade elegíveis, (iii) aplicar uma bonificação mais eficaz quando nenhum deslocamento melhora o custo da solução actual e (iv) estar reforçado com elementos de diversificação e intensificação, os quais representam uma capacidade adicional para evitar mínimos locais e convergir para a solução de partição óptima.

Os contributos relacionados com a estimação de métricas são os seguintes:

- ◇ efectuar a estimação em dois níveis de abstracção, com o objectivo de reduzir a complexidade e o tempo de cálculo da estimação, mantendo elevada a precisão das estimativas de mais alto nível;

- ◇ o procedimento desenvolvido para estimar o tempo de execução dum grafo; os aspectos a realçar são: gera estimativas precisas, considera praticamente a totalidade da semântica do meta-modelo com que se descrevem os sistemas (PSM), não exige o escalonamento explícito do grafo, dispensa as alterações do grafo para que o tempo de execução associado às mudanças de partição seja contabilizado e trata de forma muito aceitável o sincronismo existente entre dois ou mais nodos do grafo;

- ◇ o método que se aplicou na estimação do espaço duma máquina de estados, o qual assenta no cálculo das métricas número de estados, número de sinais de controlo gerados, número de activações de sinais de controlo, número de sinais de estado lidos e número de leituras de sinais de estado.

9.3 Trabalho Futuro

Para terminar identificam-se áreas para futura investigação e desenvolvimento, relacionadas com o trabalho realizado. Parte das áreas apontadas resulta de tarefas não concretizadas, correspondendo-lhe uma forte componente de investigação, enquanto a outra parte representa melhoramentos a efectuar na metodologia de partição, sendo dominante a componente de desenvolvimento. Entre as tarefas não concretizadas incluem-se:

- ◇ para aumentar o sucesso do processo de partição implementaram-se, de forma optimizada, os algoritmos que se julgou serem os mais adequados, concretamente os algoritmos de crescimento de grupos e de pesquisa tabu; outra forma de tentar melhorar os resultados da partição consiste em implementar vários algoritmos, especialmente iterativos; ao dispor de algoritmos com estratégias de optimização distintas, é possível que cada um consiga melhores resultados que os outros em determinados exemplos;

- ◇ estudar as implicações para a metodologia proposta que resultariam de a arquitectura alvo deixar de ser pré-definida para poder ser seleccionada pelo projectista e adaptar a metodologia a este cenário bem mais interessante;

- ◇ desenvolver mecanismos de reestruturação da descrição dos sistemas por forma a potenciar o sucesso do processo de partição, nomeadamente a introdução ou incremento do paralelismo em descrições essencialmente sequenciais; uma forma de aumentar o paralelismo numa descrição, resulta do desdobramento dum ciclo em vários ciclos parcelares que possam funcionar em paralelo;

- ◇ automatizar a conversão de modelos, necessária para passar (i) da descrição PSM que alimenta a partição para a representação PSMfg utilizada internamente no processo de

partição e (ii) da representação PSMfg para o modelo PSM de cada componente incluído na solução de partição.

Os melhoramentos sugeridos para a metodologia de partição são os seguintes:

- ◇ estudar em maior profundidade a viabilidade de efectuar o escalonamento dos grafos PSMfg, com o objectivo de considerar mais paralelismo nas estimativas, especialmente na estimativa do desempenho do sistema;
- ◇ implementar o mecanismo de comunicação por acesso directo à memória, tendo em vista um aumento no desempenho das soluções de partição;
- ◇ optimizar a implementação da estimação do desempenho do sistema, de modo a diminuir o tempo de cálculo e consequentemente melhorar o desempenho da ferramenta que apoia a partição;
- ◇ suportar de forma mais adequada a comunicação simultânea entre *software* e as várias partições de *hardware*;
- ◇ permitir que haja partilha de sinais de controlo nas várias transacções entre as mesmas partições de *hardware* adjacentes, quando se estimam as métricas elementares necessárias ao cálculo do espaço da unidade de controlo das partições de *hardware*;
- ◇ alterar a estimação do número de entradas em cada multiplexador, de modo a eliminar a entrada relativa à operação de escrita passível de ser implementada com o mecanismo de *set* ou *reset* dos *flip-flops*.

Considera-se que deixar várias e relevantes portas abertas a trabalho futuro, assentes na concretização do núcleo da metodologia de partição, é um bom indicador da validade do trabalho desenvolvido.

Parte III

Apêndices

Apêndice A

Algoritmos de Partição

Este apêndice descreve em maior detalhe alguns dos algoritmos introduzidos no capítulo 3, concretamente os algoritmos de Kernighan/Lin, *simulated annealing*, pesquisa binária condicionada, PACE e GCLP.

A.1 Kernighan/Lin

O princípio que está subjacente ao algoritmo de Kernighan/Lin é a comutação de um objecto incluído num grupo (ou partição) com um objecto doutro grupo, por forma a optimizar a função de custo associada ao conjunto dos grupos. No algoritmo base, a solução inicial que se pretende optimizar deve ser constituída por duas partições com igual número de objectos [EKP98a]. O algoritmo base utiliza o seguinte critério de evolução:

> *Os dois objectos que comutam de grupo são aqueles que provocam a maior descida da função de custo*[1] *ou então a menor subida da função.*

Ao critério de evolução definido corresponde uma estratégia do tipo *hill-climbing*, uma vez que o algoritmo pode evoluir numa direcção em que a função de custo piora, evitando-se assim alguns mínimos locais. Por **mínimo local** entende-se uma solução cujo valor da função de custo não é melhorado com a comutação de um único par de objectos, mas apenas com uma sequência de comutações de pares de objectos.

Cada iteração do algoritmo analisa todos os objectos (de ambos os grupos) ainda não deslocados, comutando o par de objectos que validar o critério de evolução. Após a execução da comutação de um par de objectos, obtém-se uma nova solução de partição que é guardada temporariamente. Quando todos os objectos tiverem sido deslocados, ou não for possível validar o critério de evolução, a iteração termina. Cada objecto só é deslocado uma vez por iteração, para evitar bloqueios no algoritmo. No final da iteração, guarda-se a solução de

[1] No caso duma função de custo que se pretende mínima.

partição que, de entre todas as obtidas nesta iteração, apresentar o melhor valor da função de custo.

As tarefas definidas como uma iteração são repetidas, utilizando como entrada a melhor solução de partição obtida na iteração anterior, enquanto o custo da solução de partição obtida na iteração corrente for inferior ao da solução de partição obtida na iteração anterior.

O algoritmo de Kernighan/Lin foi pensado para efectuar partição estrutural de circuitos, em que a única métrica incluída na função de custo representava a largura da ligação entre partições. Quando aplicado na partição dum sistema composto por N objectos, a complexidade temporal do algoritmo é $\mathcal{O}(N^3)$, mas após algumas alterações do algoritmo assume o valor $\mathcal{O}(N^2 * log(N))$.

O algoritmo pode ser estendido por forma a aceitar e gerar duas partições de tamanho diferente. Esta extensão, designada por **KL/não-balanceado**, obriga a que em cada passo do algoritmo se desloque um objecto em vez de se comutar um par de objectos [GVNG94]. A filosofia do algoritmo passa a ser o deslocamento de um objecto dum grupo para outro, em vez da comutação de dois objectos, de modo a optimizar a função de custo. O critério de evolução que corresponde a esta variante do algoritmo é:

> *O objecto a ser deslocado dum grupo para outro é aquele que provocar a maior descida da função de custo ou então a menor subida da função.*

Quando aplicado na partição dum sistema composto por N objectos, a complexidade temporal da extensão KL/não-balanceado é $\mathcal{O}(N^2)$.

Outra extensão do algoritmo de Kernighan/Lin, muito idêntica à variante KL/não-balanceado, foi proposta por Fiduccia/Mattheyeses [FM82]. Além da alteração proposta em KL/não-balanceado, a versão do algoritmo de Fiduccia/Mattheyeses (**KL/FM**) alterou a estrutura de dados onde se guarda a informação relativa aos objectos do sistema. Com uma estrutura de dados mais eficiente, o deslocamento dum objecto é executado num tempo constante[2] e a complexidade da extensão KL/FM do algoritmo passa a ser $\mathcal{O}(N)$.

O algoritmo de Kernighan/Lin também foi alterado para efectuar a partição funcional de sistemas em mais de dois componentes de *hardware* e/ou de *software* [VL97], originando a extensão **KL/funcional**. As alterações incluídas na extensão KL/funcional são as seguintes:

- ◇ como é aplicada na partição funcional de sistemas, contrariamente ao algoritmo inicial que está vocacionado para a partição estrutural de sistemas, implicou a introdução de novas métricas na função de custo; assim, em vez de se usar como única métrica a

[2] O tempo de cálculo não depende do número de objectos do sistema.

largura da ligação entre partições, utilizam-se também métricas de desempenho (como por exemplo o tempo de execução) ou métricas relacionadas com o espaço ocupado pelas partições;

- ◇ o algoritmo aceita e gera partições não balanceadas, desde que em cada passo do algoritmo se desloque um objecto em vez de se comutar um par de objectos;
- ◇ utiliza-se uma estrutura de dados eficiente, idêntica à de Fiduccia/Mattheyeses; como neste caso o cálculo das métricas é mais complexo, as tarefas associadas ao deslocamento de um objecto são executadas num tempo que varia logaritmicamente com o número de objectos do sistema;
- ◇ para reduzir o tempo de cálculo, o critério de paragem foi relaxado; em vez de se parar a execução quando, de uma iteração para seguinte, não se melhora a função de custo

$$Fcusto(iteracao_i) \geq Fcusto(iteracao_{i-1}) \tag{A.1}$$

pára-se a execução quando, de uma iteração para seguinte, a melhoria da função de custo não for superior a um determinado valor positivo ($NivelParagem$)

$$Fcusto(iteracao_i) \geq [Fcusto(iteracao_{i-1}) - NivelParagem] \tag{A.2}$$

Quando se efectua a partição dum sistema composto por N objectos em duas partições, a complexidade temporal da extensão KL/funcional do algoritmo é $\mathcal{O}(N * log(N))$.

A.2 *Simulated Annealing*

O funcionamento do algoritmo de *simulated annealing* é inspirado no processo de temperar materiais, no qual o material é sujeito a uma temperatura superior ao seu ponto de fusão e depois a temperatura é reduzida lentamente até ao estado de energia mínima. O estado de energia mínima é atingido, desde que para cada temperatura se atinja o ponto de equilíbrio [KGV83]. No problema de partição, a variável energia usada no processo de temperar os materiais é substituída pela variável custo da solução de partição e o estado de energia mínima equivale à solução de partição óptima.

O algoritmo SA apresenta maiores potencialidades para atingir soluções óptimas do que os algoritmos do tipo *greedy* e mesmo do que o algoritmo de Kernighan/Lin. Isto deve-se à forma como se processa a evolução do algoritmo. Primeiro, ao permitir o deslocamento do mesmo objecto mais do que uma vez durante uma iteração, consegue-se uma pesquisa mais exaustiva

do espaço de projecto. Segundo, ao seleccionar aleatoriamente os objectos a deslocar entre partições e ao fazer depender a aceitação dos deslocamentos dum número aleatório, consegue evitar-se mais mínimos locais da função de custo.

```
algSA (P, Fcusto, TemperaturaInicial) ≡

    // P é a solução de partição inicial
    T = TemperaturaInicial
    custo = Fcusto(P)
    enquanto CondicaoParagem() = FALSO fazer
        enquanto Equilibrio() = FALSO fazer
            P_deslocamento = MovimentoAleatorio(P)
            custo_deslocamento = Fcusto(P_deslocamento)
            Δcusto = custo_deslocamento − custo
            se Aceitacao(Δcusto, T) > Aleatorio(0, 1) então
                P = P_deslocamento
                custo = custo_deslocamento
            fse
        fenquanto
        T = ReducaoTemperatura(T)
    fenquanto
```

Figura A.1: Algoritmo de *simulated annealing* (SA).

A figura A.1 apresenta, de forma simplificada, o algoritmo de *simulated annealing* em pseudo-código [GVNG94].

O funcionamento do algoritmo começa com uma solução de partição P, gerada por um dos algoritmos construtivos e uma temperatura inicial $TemperaturaInicial$. A temperatura é depois diminuída lentamente, correspondendo a cada temperatura de simulação (T) uma nova iteração. Para diminuir a temperatura utiliza-se a função $ReducaoTemperatura$ que possui a funcionalidade definida pela equação A.3.

$$T_{nova} \;=\; \alpha * T_{antiga} \qquad [\text{ com } \; (0 < \alpha < 1) \;] \tag{A.3}$$

Uma iteração do algoritmo, à qual corresponde uma temperatura fixa, envolve as seguintes tarefas:

- gerar aleatoriamente uma série de deslocamentos de objectos, com a função $MovimentoAleatorio$, até se atingir o ponto de equilíbrio; o ponto de equilíbrio pode ser definido como uma situação em que após uma série de deslocamentos, a melhoria do custo da solução de partição é inferior a um nível de decisão (função $Equilibrio$); contrariamente ao que acontecia no algoritmo de Kernighan/Lin, aqui é permitido deslocar o mesmo objecto mais de uma vez durante uma iteração;

- ◇ para cada deslocamento gerado, calcular o custo da solução que daí resulta ($custo_{deslocamento}$);
- ◇ para cada deslocamento gerado, verificar se ele é aceite; um deslocamento é aceite se a variação do custo satisfizer o critério de aceitação de deslocamentos ($Aceitacao >$ número aleatório entre 0 e 1), em que $Aceitacao$ é definido na equação A.4.

$$Aceitacao(\Delta custo, T) \;=\; minimo(1, e^{-\Delta custo/T}) \tag{A.4}$$

quando o deslocamento dum objecto se traduzir numa solução de partição com custo inferior ao da solução anterior a esse deslocamento ($\Delta custo < 0$), a deslocação do objecto é sempre aceite; mesmo que a solução de partição apresente um custo superior ao da solução anterior ao deslocamento ($\Delta custo > 0$), a probabilidade de o deslocamento ser aceite varia directamente com a temperatura T e inversamente com a diferença de custo $\Delta custo$; ou seja, quanto pior for a solução obtida com o deslocamento do objecto, o mesmo é dizer quanto mais elevado for o $\Delta custo$, maior será a probabilidade de a solução ser rejeitada; por outro lado, quando menor for a temperatura T, menor é a probabilidade de se aceitarem maus deslocamentos de objectos.

O algoritmo de *simulated annealing* possui dois tipos de controlo: (i) um ciclo externo relativo à variação de temperatura, que faz terminar o algoritmo quando a temperatura T for aproximadamente nula ou quando o sistema estiver num estado que não evolui mais (função $CondicaoParagem$) e (ii) um ciclo interno que possibilita a execução de vários deslocamentos de objectos, para a mesma temperatura (função $Equilibrio$). Por cada deslocamento é gerado um número aleatório entre 0 e 1 que condiciona a aceitação desse deslocamento (função $Aleatorio$).

O utilizador pode jogar com o número de iterações a executar por cada valor da temperatura de simulação ($N_{iteracoes}$) e com a taxa de redução da temperatura (parâmetro α), para obter boas soluções de partição (com a contrapartida de um elevado tempo de cálculo) ou soluções rapidamente calculadas (com a contrapartida de serem soluções menos boas). Teoricamente, para se garantir uma solução óptima a partir de qualquer ponto inicial, o valor de $N_{iteracoes}$ é proporcional ao quadrado do espaço de projecto. Como o espaço de projecto varia exponencialmente com o número de objectos do sistema, o valor de $N_{iteracoes}$ também varia exponencialmente com o número de objectos do sistema. Com esta complexidade, o algoritmo não tem grande aplicabilidade na partição de sistemas, razão pela qual é necessário encontrar implementações eficientes que reduzam o tempo de cálculo mas mantenham elevada a probabilidade de obter soluções quase-óptimas [GL95].

Apresentam-se agora alguns ajustes para melhorar a eficiência do algoritmo SA. Convém lembrar que, para o algoritmo apresentado na figura A.1, a complexidade temporal depende das funções *CondicaoParagem*, *Equilibrio*, *Aceitacao* e *ReducaoTemperatura*, que por sua vez depende da constante α.

Em primeiro lugar, a temperatura inicial deve ser suficientemente elevada para garantir que, independentemente da solução de partição inicial, se chega a uma solução óptima. Isto porque, quanto maior for a temperatura maior será a probabilidade de se aceitarem maus deslocamentos.

Provou-se que valores elevados para a taxa de redução da temperatura, parâmetro α da equação A.3, conduzem o algoritmo SA a melhores resultados [GL95]. Valores elevados para α, comummente no intervalo $[0.8, 0.99]$, implicam uma variação lenta da temperatura. Em vez de um parâmetro α fixo, também é possível aplicar um parâmetro que depende da temperatura de simulação. Por exemplo, revela-se adequado um parâmetro α que assume um valor mais elevado nas temperaturas intermédias.

Como a utilização dum valor de $N_{iteracoes}$ a depender exponencialmente do número de objectos do sistema é inviável, é prática comum seleccionar-se um valor de $N_{iteracoes}$ que é uma função polinomial do número de objectos do sistema. Em alternativa, pode optar-se por um valor de $N_{iteracoes}$ que depende da temperatura de simulação, assumindo o valor mais elevado nas temperaturas mais baixas. Deste modo, dispõe-se de uma margem de manobra suplementar para evitar parar em mínimos locais da função de custo. Outra possibilidade, é fazer com que o número de iterações, a executar com uma dada temperatura, dependa dos resultados obtidos com os deslocamentos efectuados com as temperaturas precedentes.

Como foi mencionado, a condição de paragem do algoritmo pode ser verificada quando se atinge o ponto de temperatura nula ou então quando o sistema atinge um estado do qual não evolui mais. Tendo esta definição por base, é possível derivar condições de paragem que reduzem significativamente o tempo de cálculo, sem prejudicar em demasia a qualidade da solução final. Apresentam-se agora alguns exemplos de condição de paragem: (i) parar quando se atinge uma temperatura acima de zero, (ii) parar quando se atinge uma temperatura $T \leq \frac{\delta}{ln[(|S|-1)/\theta]}$, que garante uma probabilidade θ de a solução obtida estar numa vizinhança δ da solução óptima[3], (iii) parar quando se ultrapassa o limite imposto ao número de iterações e/ou de temperaturas seguidas sem melhoria na função de custo, (iv) parar quando a percentagem de objectos realmente deslocados, em relação ao total de deslocamentos tentados, desce além dum determinado limite inferior ou (v) parar quando se ultrapassa o limite imposto ao número total de iterações, para a globalidade das temperaturas.

[3] S designa o espaço de projecto, ou seja, o universo das soluções de partição.

A.3 Pesquisa Binária Condicionada

O algoritmo de pesquisa binária condicionada[4] (PBC) foi pensado para resolver o problema da partição em dois componentes, um de *hardware* e outro de *software*, com o objectivo de identificar a solução que, ocupando o espaço mínimo em *hardware*, atinge o desempenho exigido. O algoritmo PBC em vez de tentar atingir simultaneamente soluções óptimas relativamente à minimização do espaço ocupado em *hardware* e ao desempenho, concentra-se apenas no desempenho [VGG94].

Encarando o problema de partição como uma tarefa em que se procura a solução de partição com custo mínimo e que emprega a menor quantidade de recursos de *hardware* possível, a estratégia do algoritmo de partição é definida por:

> *Com o condicionalismo imposto ao espaço ocupado em hardware a variar desde zero, solução totalmente em software, até ao valor máximo $TamMaxHw$, solução totalmente em hardware, procurar a primeira solução com custo mínimo.*

Com a definição anterior em mente, o objectivo do algoritmo é determinar o menor valor do condicionalismo (c_{tamHw}) que atinge o desempenho exigido. Se um dado valor de c_{tamHw} minimizar a função de custo $Fcusto$, estabelecendo a solução para o problema de partição, qualquer valor maior que c_{tamHw} e menor que a quantidade de recursos de *hardware* disponível ($TamMaxHw$), origina uma solução válida. Desta observação resulta que o problema de minimização do *hardware* respeitando um condicionalismo, se transforma na tentativa de encontrar o primeiro zero na sequência de custo. Por sequência de custo entende-se uma sequência de valores da função de custo, em que a cada zero corresponde uma solução exequível. Deste modo, obtém-se um problema idêntico ao da pesquisa num *array* ordenado, resolúvel por uma **pesquisa binária**.

Em vez de um algoritmo PBC a avançar lentamente, procurando em cada iteração melhorar ligeiramente um dos objectivos sem piorar muito o outro, utiliza-se um algoritmo que "relaxa" um dos objectivos. Assim, em vez de se tentar minimizar a quantidade de recursos de *hardware*, apenas se pretende mantê-la abaixo dum certo valor (c_{tamHw}). A função de custo da equação A.5 traduz estes objectivos.

$$Fcusto(P, Req, c_{tamHw}) =$$

$$= K_{perf} * \sum_{j=1}^{m} SucessoAtingir(r_j) + K_{area} * Violacao(TAM(H), c_{tamHw}) \qquad \text{(A.5)}$$

[4] *Binary Constraint-Search*, na terminologia inglesa.

onde

- $P = H \cup S$ é uma solução de partição;
- $Req = \{r_1, ..., r_m\}$ são os requisitos do projecto;
- $TAM(H)$ designa o espaço ocupado pela partição de *hardware* H;
- $SucessoAtingir(r_j)$ indica o sucesso com que se atinge o requisito r_j;
- $Violacao()$ quantifica o grau de violação do condicionalismo imposto ao espaço ocupado em *hardware*;
- $K_{perf} = K_{area} = 1$.

Deste modo, quando o algoritmo iterativo procura a solução óptima dispõe duma maior flexibilidade para lidar com os dois aspectos envolvidos na partição e o problema de minimização dos recursos de *hardware* assume a seguinte definição:

> *Dada uma solução de partição P, os requisitos Req, o algoritmo de partição iterativo $PartAlg$ e a função de custo $Fcusto$, o problema de minimização dos recursos de hardware e simultâneo respeito pelos condicionalismos consiste em calcular o menor c_{tamHw} para o qual*
>
> $$Fcusto(PartAlg(P, Req, c_{tamHw}, Fcusto()), Req, c_{tamHw}) \; = \; 0 \qquad \text{(A.6)}$$

A figura A.2 ilustra o algoritmo de pesquisa binária condicionada, aplicado a uma sequência de custo definida a partir da gama de valores do condicionalismo imposto aos recursos de *hardware*. As variáveis inf e sup são os dois valores do condicionalismo que delimitam a janela onde pode existir a solução de partição com custo mínimo. A variável *meio* guarda o condicionalismo aplicado em cada iteração, coincidindo com o ponto intermédio da janela anterior. Em cada instante, a melhor solução encontrada pelo algoritmo $PartAlg$ está guardada em P_{melhor}, significando isso que de entre todas as soluções analisadas é aquela que exige o menor condicionalismo. A função $PartAlg$ é um algoritmo de partição iterativo, como por exemplo *simulated annealing*.

Tendo-se constatado que o algoritmo permanece um número elevado de iterações utilizando valores de c_{tamHw} próximos do valor óptimo, é recomendável alterar o algoritmo da figura A.2 para que aceite soluções que estejam muito próximas da solução óptima. Em relação ao algoritmo da figura A.2, o factor de precisão δ é definido por $(sup - inf)/TamMaxHw$. O

```
algPBC (P, Req, TamMaxHw, Fcusto) ≡

    inf = 0, sup = TamMaxHw
    enquanto (inf < sup) fazer
        meio = (inf + sup + 1)/2      // meio é usado como c_tamHw
        P = PartAlg(P, Req, meio, Fcusto(P, Req, meio))
        se Fcusto(P, Req, meio) = 0 então
            sup = meio − 1
            P_melhor = P
        senão
            inf = meio
        fse
    fenquanto
    devolver (P_melhor)
```

Figura A.2: Algoritmo de pesquisa binária condicionada (PBC).

valor de δ representa o tamanho máximo da janela do condicionalismo, que ao conter uma solução com custo mínimo, será considerada a melhor solução de partição. Considerando o factor de precisão, a condição de paragem é alterada de $((sup - inf) \leq 0)$ para $((sup - inf) \leq \delta)$. Como ilustração, um factor de precisão igual a 1% permite obter uma aceleração da ordem dos 250% no cálculo da solução de partição.

No pior dos casos, a complexidade do algoritmo de pesquisa condicionada é igual ao produto da complexidade do algoritmo de partição iterativo ($PartAlg$) pela complexidade do algoritmo PBC. Como a complexidade duma pesquisa binária sobre uma sequência com $TamMaxHw$ elementos é $\mathcal{O}(log_2 TamMaxHw)$ e o factor de precisão reduz esta complexidade a uma constante, a complexidade da pesquisa binária sobre N elementos e com um factor de precisão δ é dada por $\mathcal{O}(log_2(1/\delta))$. Daqui resulta que o algoritmo PBC tem a complexidade do algoritmo de partição $PartAlg$, a menos da constante $log_2(1/\delta)$.

A estratégia do algoritmo PBC garante que a quantidade de recursos de *hardware* seleccionados pelas soluções é quase-óptima. Relativamente às abordagens que aplicam simplesmente o algoritmo *simulated annealing*, pode dizer-se que o algoritmo PBC encontra sempre uma solução que satisfaz o requisito de desempenho e requer menos recursos de *hardware*. O algoritmo é aplicável noutras abordagens ao problema de partição em *hardware* e *software*, onde a métrica a minimizar não seja o espaço ocupado em *hardware* e/ou o requisito não seja o desempenho.

A.4 Partição com Ênfase na Comunicação

O algoritmo PACE[5], um algoritmo específico para partição *hardware/software*, baseia-se em programação dinâmica e apresenta como principal característica a ênfase dada à comunicação

[5] ***P**artitioning **A**lgorithm with **C**ommunication **E**mphasis*, na terminologia inglesa.

entre os objectos do sistema. Os objectos manipulados pelo algoritmo são blocos de código do modelo do sistema, designados por BBEs[6] [Knu95][KM96b]. Para efectuar a partição, cada objecto é anotado com o espaço que ocupa em *hardware*, a aceleração obtida ao deslocá-lo para *hardware* e a aceleração resultante da redução da comunicação entre *hardware* e *software*. Obtém-se uma redução na comunicação entre *hardware* e *software*, quando os objectos adjacentes do objecto deslocado para *hardware* também se encontram em *hardware*. O modelo de comunicação aplicado pelo algoritmo PACE limita a comunicação entre objectos de *hardware*, sem que haja intervenção do *software*, aos objectos adjacentes.

Com uma arquitectura alvo composta por um componente de *hardware* e um microprocessador, o problema de partição é formulado do seguinte modo:

> *O processo de partição é a pesquisa de sequências de objectos não sobrepostas, a atribuir à partição de hardware, que resultem na maior aceleração para o sistema e não ocupem mais do que o espaço disponível no componente de hardware.*

Para determinar o melhor conjunto de sequências a atribuir a *hardware*, utiliza-se o algoritmo incluído na figura A.3. Para evitar que sejam seleccionadas duas ou mais sequências com o mesmo objecto, quando o algoritmo pesquisa as sequências, estas encontram-se organizadas em grupos ordenados. Um grupo é composto por todas as sequências que terminam no mesmo objecto. Para clarificar o funcionamento do algoritmo, a figura A.4 mostra um sistema composto por quatro BBEs e a tabela A.1 define os grupos e a ordenação respeitantes a esse sistema. Na tabela A.1, os valores do espaço ocupado pela sequência $S(o_i : o_j)$ em *hardware* e os valores da aceleração obtida ao deslocar a mesma sequência para *hardware*, são calculados com a equação A.7 e A.8, respectivamente.

$$area(S(o_i : o_j)) = \sum_{k=i}^{j} area(o_k) \tag{A.7}$$

$$aceleracao(S(o_i : o_j)) = \sum_{k=i}^{j} aceleracao(o_k) + \sum_{k=i}^{j-1} aceleracaoLigacao(o_k, o_{k+1}) \tag{A.8}$$

No final do algoritmo é necessário reconstruir o conjunto das sequências que constituem a partição de *hardware*. Com os grupos de sequências definidos e ordenados como se mostra na tabela A.1, procura-se a melhor sequência[7] do grupo com o maior índice (4 no exemplo em discussão) que ocupe um espaço a não superior ao disponível ($1 \leq a \leq maxArea$). Supondo que a sequência do grupo que induz a maior aceleração é $S(o_i : o_4)$, duas situações podem

[6] **B**locos **B**ásicos de **E**scalonamento.

[7] A melhor sequência é a que induz a maior aceleração.

```
algPACE () ≡

// Iniciações
para G = 1 até NumBBEs fazer // G ≡ grupo de sequências
  para A = 0 até maxArea fazer // A ≡ espaço em hardware
    Seleccao_melhor[G][A] = {}
    Aceleracao_melhor[G][A] = 0
  fpara
fpara

// Processo de partição
para G = 1 até NumBBEs fazer
  // Procurar a melhor solução de partição para todos os grupos até G
  BBE_sup = G // todas as sequências do grupo G possuem como índice máximo G

  para BBE_inf = 1 até BBE_sup fazer
    // Analisar todas as sequências de BBEs do grupo G
    AreaSeq = espaço total da sequência S(BBE_inf : BBE_sup)
    AceleracaoSeq = aceleração total da sequência S(BBE_inf : BBE_sup)

    para A = AreaSeq até maxArea fazer
      // O valor de A garante que existe espaço suficiente para a sequência.
      // Assume-se que a sequência é seleccionada para hardware.
      // Se BBE_inf = 1, a sequência é formada por todos os elementos, e
      // não se podem seleccionar mais elementos para hardware.

      se BBE_inf = 1 então
        // Apenas esta sequência pode ser seleccionada
        se AceleracaoSeq > Aceleracao_melhor[G][A] então
          Aceleracao_melhor[G][A] = AceleracaoSeq
          Seleccao_melhor[G][A] = S(BBE_inf : BBE_sup)
        fse
      senão
        // Assumir também que a melhor solução, para as sequências
        // até BBE_inf − 1, é seleccionada para o espaço restante.
        Aceleracao = AceleracaoSeq + Aceleracao_melhor[BBE_inf − 1][A − AreaSeq]
        se Aceleracao > Aceleracao_melhor[G][A] então
          Aceleracao_melhor[G][A] = Aceleracao
          Seleccao_melhor[G][A] = S(BBE_inf : BBE_sup)
        fse
      fse
    fpara
  fpara

  // Para cada espaço, verificar se a melhor solução encontrada nas sequências sem o
  // elemento BBE_sup é melhor que a solução encontrada nas sequências com BBE_sup.
  // Se sim, substituir as soluções agora encontradas pela melhor solução anterior.
  se BBE_sup > 1 então
    para A = 0 até maxArea fazer
      se Aceleracao_melhor[G − 1][A] > Aceleracao_melhor[G][A] então
        Aceleracao_melhor[G][A] = Aceleracao_melhor[G − 1][A]
        Seleccao_melhor[G][A] = Seleccao_melhor[G − 1][A]
      fse
    fpara
  fse
fpara

devolver (Seleccao_melhor[ ],Aceleracao_melhor[ ])
```

Figura A.3: Algoritmo de partição com ênfase na comunicação (PACE).

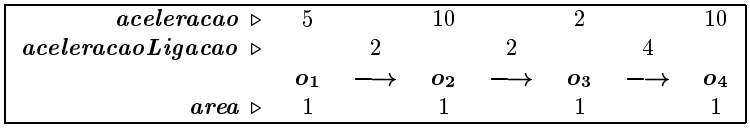


Figura A.4: Um sistema definido por quatro objectos, com as anotações utilizadas pelo algoritmo PACE.

Grupo	*Sequência*	*BBEs*	*Espaço*	*Aceleração*
1	$S(o_1 : o_1)$	o_1	1	5
2	$S(o_1 : o_2)$	o_1, o_2	2	17
	$S(o_2 : o_2)$	o_2	1	10
	$S(o_1 : o_3)$	o_1, o_2, o_3	3	21
3	$S(o_2 : o_3)$	o_2, o_3	2	14
	$S(o_3 : o_3)$	o_3	1	2
	$S(o_1 : o_4)$	o_1, o_2, o_3, o_4	4	35
4	$S(o_2 : o_4)$	o_2, o_3, o_4	3	28
	$S(o_3 : o_4)$	o_3, o_4	2	16
	$S(o_4 : o_4)$	o_4	1	10

Tabela A.1: Definição dos grupos de sequências e da ordenação usados no algoritmo PACE, para o sistema da figura A.4.

ocorrer: (i) o espaço ocupado pela sequência é igual ao espaço disponível ou (ii) o espaço ocupado pela sequência (a) é inferior ao espaço disponível. No primeiro caso, a sequência $S(o_i : o_4)$ será a solução de partição. No segundo caso, procura-se no grupo $i-1$ a melhor sequência, que ocupe um espaço a_r não superior ao restante ($a_r = maxArea - a$). Deste modo garante-se a escolha dum conjunto de sequências sem objectos repetidos. O processo repete-se até se esgotar o espaço disponível. Quando isto ocorrer, a solução de partição é composta pelo conjunto das melhores sequências encontradas.

Na implementação do algoritmo, a pesquisa das melhores sequências é suportada pelas matrizes $Seleccao_{melhor}[N][A]$ e $Aceleracao_{melhor}[N][A]$, em que $N = NumBBEs$ é o número de objectos do sistema e $A = maxArea$ é o espaço disponível em *hardware*. A matriz $Seleccao_{melhor}$ guarda a melhor sequência para cada combinação de valores do par (grupo,espaço), enquanto a matriz $Aceleracao_{melhor}$ guarda a aceleração correspondente à melhor sequência, para cada combinação de valores do par (grupo,espaço). A reconstrução do conjunto das sequências que constituem a partição de *hardware* começa na melhor sequência do grupo $NumBBEs$, que se encontra guardada na posição $Seleccao_{melhor}[NumBBEs][maxArea]$ e cuja aceleração está guardada na posição $Aceleracao_{melhor}[NumBBEs][maxArea]$.

Para o sistema da figura A.4 e no caso de o espaço disponível em *hardware* ser 3, a partição de *hardware* obtida pelo algoritmo coincide com a sequência $S(o_2 : o_4) = \{o_2, o_3, o_4\}$ e a aceleração que lhe está associada é 28.

O algoritmo PACE tanto pode ser aplicado na optimização do espaço de *hardware* com o objectivo de atingir o desempenho exigido, como na optimização do desempenho com o objectivo de não exceder o espaço de *hardware* disponível.

A complexidade temporal do algoritmo PACE é $\mathcal{O}(N^2 * A)$ e a complexidade espacial é $\mathcal{O}(N * A)$, com N a ser o número de objectos (BBEs) e A o espaço disponível em *hardware*. Com esta complexidade temporal, a partição de sistemas de grande dimensão é bastante morosa.

A.5 Partição *Hardware/Software* Direccionada pela Urgência Global e Fase Local

O algoritmo GCLP[8] aplica em cada etapa do processo de partição a função de proximidade mais apropriada a esse momento. A função de proximidade a aplicar depende da maior ou menor necessidade de optimizar o desempenho e da heterogeneidade do sistema. O algoritmo de partição GCLP faz parte duma metodologia de desenvolvimento de sistemas embebidos, para os quais se desejam implementações de baixo custo que obedeçam aos requisitos de desempenho exigidos, usando para isso uma arquitectura alvo heterogénea composta por um processador e *hardware* dedicado (ASIC) [KL94].

Durante a partição, os sistemas são modelados por um grafo acíclico direccionado $G = \{N, A\}$, em que os nodos N representam a computação, com granulosidade ao nível da tarefa ou do processo, e os arcos A definem as precedências entre nodos. As principais características do meta-modelo subjacente ao grafo são as seguintes:

- cada nodo o_i tem associados quatro números não negativos: (i) o espaço ocupado pelos recursos numa implementação de *hardware* (ah_i), (ii) o espaço ocupado pelo código numa implementação de *software* (as_i), (iii) o tempo de execução na implementação de *hardware* (thw_i) e (iv) o tempo de execução na implementação de *software* (tsw_i);
- cada arco $a_{i,j}$ tem associado um inteiro não negativo ($NA_{i,j}$), que representa o número de amostras enviadas do nodo i para o nodo j;
- o custo associado com a comunicação efectuada através da interface *hardware*/*software* é caracterizado por três valores: (i) o espaço ocupado em *hardware* (ah_{com}), (ii) o espaço ocupado em *software* (as_{com}) e (iii) o tempo de comunicação (t_{com}).

O problema de partição a resolver com o algoritmo GCLP é definido do seguinte modo:

> *O processo de partição procura atribuir cada nodo do grafo do sistema a hardware ou a software (atribuição I_i) e o instante de arranque desse nodo (escalonamento ts_i), tendo em consideração as limitações impostas à implementação e o custo de comunicação, com o objectivo de minimizar o espaço ocupado em hardware.*

[8] ***G**lobal **C**riticality/**L**ocal **P**hase*, na terminologia inglesa.

As limitações do projecto são o requisito de desempenho (latência T), a quantidade de recursos de *hardware* (AH) e a quantidade de recursos de *software* (memória disponível AS).

A solução mais imediata para minimizar o espaço ocupado em *hardware* consiste em deslocar o máximo possível de nodos para *software*, estando esta opção limitada pelo requisito de desempenho (T) e pelo condicionalismo de *software* (AS). Outra alternativa consiste em atribuir os nodos aos componentes da arquitectura alvo, de acordo com a minimização duma função de proximidade. Dois exemplos de função de proximidade são: (i) minimizar o tempo de execução do nodo e a comunicação do nodo com os outros nodos e (ii) minimizar a percentagem de recursos de *hardware* seleccionada pelo nodo, em relação aos nodos de *software*. Contudo, os dois objectivos podem ser contraditórios, porque é difícil conciliar a minimização do espaço em *hardware* com o respeito pelos requisitos de desempenho. Deste modo, ao tentar minimizar o atraso (ou o espaço), obtém-se uma solução não exequível devido ao espaço (ou ao atraso). As abordagens que empregam a mesma função de proximidade com todos os nodos, muitas vezes não produzem soluções globalmente óptimas porque param em mínimos locais dessa função.

O algoritmo GCLP aborda o problema de partição de uma maneira diferente, que se traduz na aplicação da função de proximidade mais apropriada a cada etapa do processo de partição. A função é seleccionada de acordo com duas medidas: uma medida que é uma estimativa da urgência temporal global[9] (***CG***) em cada passo do algoritmo e uma medida da heterogeneidade dos nodos (***FL***). O algoritmo aplica a medida CG para manter a exequibilidade da implementação e usa a fase local FL para obter optimização local, tendo em atenção as características dos nodos. Em cada passo, os critérios global e local são sobrepostos pelo mecanismo de nível de decisão, que procura seleccionar a melhor função de proximidade.

O algoritmo GCLP apresenta um ciclo exterior que é executado N vezes, correspondendo a cada execução uma etapa do processo de partição (figura A.5). Em cada etapa, selecciona-se um nodo de entre os que estão prontos[10], com base num critério de urgência do escalonamento. Para o nodo n_i escolhido, é necessário encontrar a atribuição I_i a uma das partições e o instante ts_i em que entra em funcionamento (escalonamento). Para efectuar a atribuição são consideradas as medidas CG e FL.

A medida CG, baseada na percentagem de nodos escalonados e não escalonados e nos requisitos de desempenho, orienta a escolha da função de proximidade que controla a atribuição dos nodos a uma das partições. Se o tempo for crítico, a medida CG favorece a selecção duma função que minimiza o tempo de conclusão da execução, senão favorece a selecção duma função que minimiza o espaço ocupado em *hardware*. Para escolher uma função de entre as

[9] Urgência temporal representa a maior ou menor necessidade de optimizar o desempenho num determinado momento.

[10] Diz-se que um nodo está pronto se todos os nodos precedentes estiverem escalonados.

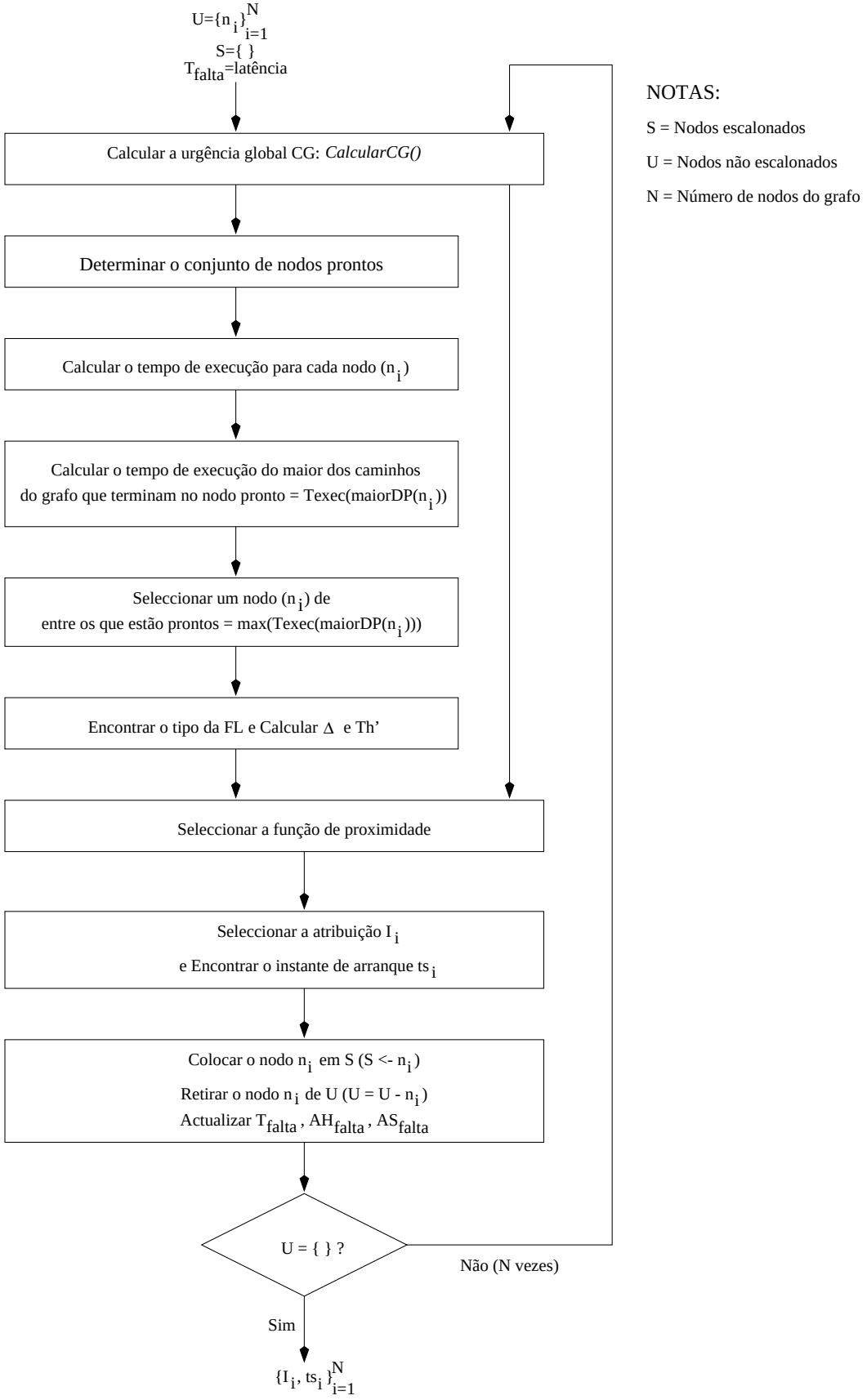


Figura A.5: Algoritmo de partição GCLP.

duas disponíveis, o valor de CG é comparado com um nível de decisão, como se mostra na figura A.6.

Uma atribuição de nodos aplicando apenas a medida CG, raramente é globalmente óptima porque os nodos são heterogéneos e esta medida não depende das características do nodo envolvido em cada etapa. Para obter uma medida que considere a heterogeneidade dos nodos, definiram-se três tipos de nodos: nodos do tipo extremidade, nodos do tipo repelente e nodos do tipo normal. Por exemplo, um nodo que é uma extremidade de *hardware* (mau para *hardware*) requer um espaço elevado em *hardware*, pode ser implementado em *software* com custo reduzido. A preferência local (numa determinada etapa) por um nodo do tipo extremidade, é traduzida no *delta da fase local FL* (Δ) e afecta o nível de decisão (Th') envolvido na selecção da função de proximidade (figura A.6).

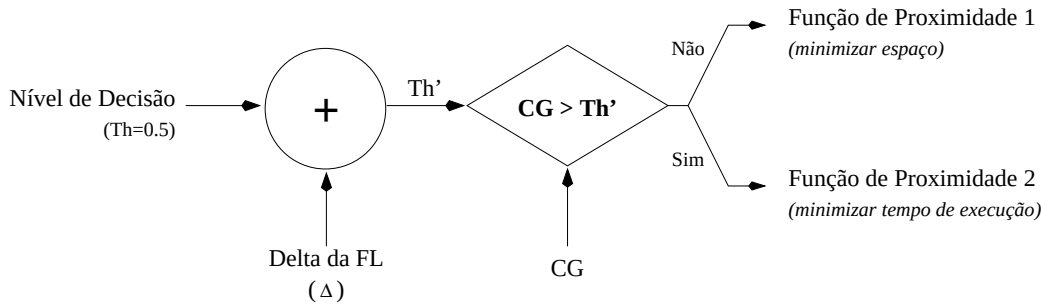


Figura A.6: Selecção da função de proximidade a aplicar em cada etapa do algoritmo GCLP.

A solução de implementação obtida numa etapa, poderá ser melhorada através da deslocação para *hardware* de nodos de *software*, com o objectivo de melhorar o desempenho. Por exemplo, se houver dois candidatos a deslocar para *hardware*, se os dois candidatos tiverem implementação idêntica em *software* e só for possível deslocar um deles, o nodo seleccionado será aquele que apresentar o maior valor de repelente de *software*. Um nodo tem associadas várias propriedades de repelente, as quais estão relacionadas com o ganho, em espaço ou desempenho, que se obtém ao trocar esse nodo duma implementação para a implementação complementar. O *valor efectivo do repelente* (VR) é função das propriedades de repelente. Ao afectar o *delta da FL* (Δ), a medida VR altera o valor do nível de decisão aplicado na comparação com CG (Th' na figura A.6), influenciando assim a escolha da função de proximidade.

A medida $CG(k)$, o mesmo é dizer o valor de CG quando já foram atribuídos k nodos, é definida como a probabilidade de um nodo não escalonado ser atribuído a *hardware*, por forma a atingir a exequibilidade global da solução de partição. Quanto maior for CG, mais nodos deverão ser deslocados para *hardware*, para tornar a solução de partição exequível. A medida CG é obtida pela função $CalcularCG$ descrita na figura A.7, onde o tamanho dos nodos é definido como o número de operações elementares desse nodo. A soma e o produto são operações elementares. Os nodos a deslocar para *hardware* são escolhidos por uma função de

prioridade f_p. Por exemplo, a função de prioridade selecciona os nodos n_i com o maior tempo de execução em *software* (tsw_i), os nodos que apresentam a maior diferença ($tsw_i - thw_i$) ou então os nodos que ocupam menos espaço em *hardware* (ah_i).

CalcularCG () ≡

INICIO:
// Encontrar os nodos a deslocar para *hardware* de modo a respeitar a latência

Estimar o conjunto de nodos a deslocar para *hardware* (H) com base numa
função de prioridade (f_p)
Calcular o tempo de execução actual considerando que o conjunto H foi deslocado para hw
se *(solução não for exequível)* então // Estimando a conclusão da execução
saltar para *INICIO*
fse

// Calcular CG, com U a ser o conjunto de nodos não escalonados

$CG = \frac{\sum_{n_i \in H} tamanho(n_i)}{\sum_{n_i \in U} tamanho(n_i)}$
Actualizar o tempo que resta para perfazer a latência permitida

devolver CG

Figura A.7: Cálculo da medida CG utilizada no algoritmo GCLP.

Apresenta-se agora a definição dos três tipos de nodo que contribuem para a FL.

Um nodo é uma **extremidade** duma implementação I, ou seja, é um mau candidato para essa implementação, se consumir muitos recursos preciosos de I e poucos recursos preciosos da implementação complementar $\overline{I}$. Um nodo com elevado tempo de execução é uma extremidade de *software* e um nodo que ocupa um espaço elevado é uma extremidade de *hardware*. A intensidade com que um nodo é uma extremidade, é quantificada na medida de extremidade.

Para cada nodo, ao qual está associado um sub-grafo, identificam-se algumas propriedades de **repelente**. *MINB*[11] e *nível de precisão* são propriedades características de um nodo repelente de *software*, enquanto *mistura de instruções que usam memória intensivamente* e *mistura de instruções que operam com LUTs* são propriedades características de nodos do tipo repelente de *hardware*.

Por exemplo, quando maior for o *MINB* ($0 \leq MINB \leq 1$) maior é o valor de repelente. Se a dois nodos n_i e n_j corresponderem implementações de *software* ocupando espaços idênticos, mas o nodo n_i tiver um *MINB* maior do que o *MINB* do nodo n_j, então a implementação do nodo n_i em *hardware* ocupa menos espaço do que a implementação do nodo n_j em *hardware*. Neste caso, o nodo n_i é um repelente de *software* relativamente ao nodo n_j.

As outras propriedades do tipo repelente são definidas de forma análoga à de *MINB*. Ao

[11] **M**istura de **I**nstruções que operam ao **N**ível do **B**it - definido como a percentagem de instruções que operam ao nível do bit em relação ao total de instruções do nodo.

combinar o valor das várias propriedades dum nodo obtém-se o valor efectivo do repelente desse nodo. O valor efectivo do repelente para um dado nodo n_i ($-0.5 \leq VR_i \leq 0.5$) é dado por

$$VR_i = \frac{1}{2} * [\sum_{p(n_i):n_i \in H} pesoPropriedade_p * valorNormalizadoPropriedade_{p(n_i)} - \\ - \sum_{p(n_i):n_i \in S} pesoPropriedade_p * valorNormalizadoPropriedade_{p(n_i)}] \quad \text{(A.9)}$$

em que H é o conjunto de nodos a deslocar para *hardware*, S é o conjunto de nodos escalonados e p é uma das propriedades do tipo repelente.

Em conclusão, um nodo n_i é um repelente a uma implementação se não for uma extremidade e possuir um valor de $VR_i \neq 0$. O valor de VR é uma medida dos recursos que se poupam quando se trocam dois nodos idênticos do tipo repelente entre implementações complementares.

Um nodo é considerado **normal**, se não for nem uma extremidade nem um repelente.

As características de cada nodo afectam o nível de decisão aplicado na escolha da função de proximidade ($Th' = 0.5 + \Delta$) através do *delta da fase local* (Δ). Se o nodo é do tipo extremidade, o *delta da fase local* assume um comportamento linear relativamente à medida de extremidade. Nos nodos do tipo repelente o *delta da fase local* assume o valor de VR ($Th' = 0.5 + VR$) e nos nodos normais o *delta da fase local* é nulo ($Th' = 0.5$).

No pior dos casos, a complexidade temporal do algoritmo GCLP é $\mathcal{O}(N * A)$, em que N é o número de nodos e A o número de arcos. Para sistemas predominantemente de dados, como os sistemas de processamento de sinal ou de imagem, em que os número de nodos e de arcos se aproximam, a complexidade do algoritmo é quadrática ($\mathcal{O}(N^2)$).

Apêndice B

Notação para Representar a Complexidade de Algoritmos

Para expressar uma aproximação ao tempo cálculo, ou ao espaço em memória, exigido pela resolução de determinado problema, o qual não é conhecido com exactidão, pode utilizar-se uma representação assimptótica baseada na notação-$\mathcal{O}$ [Knu97][AG94]. Uma representação assimptótica ao tempo de cálculo, ou equivalentemente uma complexidade temporal assimptótica, apenas garante que a precisão da aproximação por ele expressa é elevada quando a dimensão do problema for muito elevada.

B.1 Notação-$\mathcal{O}$

A notação-$\mathcal{O}$, utilizada como aproximação do tempo de cálculo associado com a resolução dum problema, define um limite superior para esse tempo.

A expressão $\mathcal{O}(f(n))$ pode ser usada sempre que $f(n)$ for uma função que depende dum inteiro positivo n e designa uma quantidade que não é conhecida de forma explícita, excepto que a sua magnitude é muito elevada.

> ***Definição:*** o significado de $\mathcal{O}(f(n))$ é o de que existem duas constantes positivas H e n_0 tais que o número x_n, representado por $\mathcal{O}(f(n))$, satisfaz a condição $|x_n| \leq H * |\mathcal{O}(f(n))|$, para qualquer inteiro $n \geq n_0$.

Para ilustrar a notação-$\mathcal{O}$, apresenta-se como exemplo um polinómio de grau menor ou igual a m: $P_m(n) = a_0 + a_1 * n + \ldots + a_m * n^m$. Aplicando a propriedade $s + t \leq |s| + |t|$ e pondo em evidência n^m obtém-se

$$|P_m(n)| \leq |a_0| + |a_1| * n + \ldots + |a_m| * n^m \;=\; (\frac{|a_0|}{n^m} + \frac{|a_1|}{n^{m-1}} + \ldots + |a_m|) * n^m \qquad \text{(B.1)}$$

Como $\frac{1}{n^m} \leq 1$, para qualquer $n \geq 1$, resulta que

$$|P_m(n)| \leq (|a_0| + |a_1| + \ldots + |a_m|) * n^m \qquad (se\ n \geq 1) \tag{B.2}$$

Logo $P_m(n) = \mathcal{O}(n^m)$ e as constantes H e n_0 assumem os valores $H = |a_0| + |a_1| + \ldots + |a_m|$ e $n_0 = 1$, ou qualquer outro par de valores $H = \frac{|a_0|}{k^m} + \frac{|a_1|}{k^{m-1}} + \ldots + |a_m|$ e $n_0 = k$, obtido com $k \in [2 : n]$.

A aproximação representada através da notação-$\mathcal{O}$ define uma igualdade válida apenas num sentido, ou seja, pode exprimir-se que $k1 * n^2 + n = \mathcal{O}(n^2)$ mas não que $\mathcal{O}(n^2) = k1 * n^2 + n$.

Uma declaração, ou fórmula, que envolve $\mathcal{O}(f(n))$ pode ser entendida como um conjunto de funções de n, resultando daqui uma nova definição para a notação-$\mathcal{O}$.

> ***Definição:*** $\mathcal{O}(f(n))$ designa o conjunto de todas as funções g que dependem do inteiro n, para o qual existem duas constantes H e n_0 que garantem a validade da condição $|g(n)| \leq H * |\mathcal{O}(f(n))|$, para todos os inteiros $n \geq n_0$.

Algumas das operações permitidas pela notação-$\mathcal{O}$ são as seguintes:

$$\begin{array}{lcll} f(n) & = & \mathcal{O}(f(n)) & \\ k * \mathcal{O}(f(n)) & = & \mathcal{O}(f(n)) & (\text{se } k \text{ for uma constante}) \\ \mathcal{O}(f(n)) + \mathcal{O}(f(n)) & = & \mathcal{O}(f(n)) & \\ \mathcal{O}(\mathcal{O}(f(n))) & = & \mathcal{O}(f(n)) & \\ \mathcal{O}(f(n)) * \mathcal{O}(g(n)) & = & \mathcal{O}(f(n) * g(n)) & \\ \mathcal{O}(f(n) * g(n)) & = & f(n) * \mathcal{O}(g(n)) & \end{array} \tag{B.3}$$

B.2 Notação-Ω

A notação-Ω define uma aproximação do tempo de cálculo associado com a resolução dum problema, sob a forma de limite inferior.

> ***Definição:*** a declaração $g(n) = \Omega(f(n))$ indica que existem duas constantes positivas L e n_0 que garantem a validade da condição $|g(n)| \geq L * |\Omega(f(n))|$, para qualquer inteiro $n \geq n_0$.

Se o tempo de cálculo despendido por um algoritmo $A1$ na resolução de determinado problema for (aproximado por) $\Omega(n^2)$ e se o tempo de cálculo de outro algoritmo $A2$ for $\mathcal{O}(n * log\ n)$, para valores elevados de n o algoritmo $A2$ é melhor do que $A1$, mas não é possível saber a partir de que valor isso ocorre quando não se conhecem as constantes envolvidas nas notações $\mathcal{O}$ e Ω.

B.3 Notação-Θ

A notação-Θ define com exactidão a "ordem de grandeza" da dependência entre o tempo de cálculo despendido na resolução dum problema e a dimensão desse problema, sem que para isso seja necessário explicitar a constante H ou L exigida pela notação $\mathcal{O}$ ou Ω, respectivamente. Nas definições anteriores, a dimensão do problema é representada por n.

Definição: $g(n) = \Theta(f(n)) \Leftrightarrow g(n) = \mathcal{O}(f(n))$ **e** $g(n) = \Omega(f(n))$.

Apêndice C

Funções de Custo

Este apêndice disponibiliza informação complementar sobre algumas das funções de custo apresentadas na secção 3.9. A formulação das funções terá o aspecto da equação C.1 ou da equação C.2.

$$Fcusto = \sum_i K_i * M_i \tag{C.1}$$

$$Fcusto = \sum_i K_i * f_i(M_i) \tag{C.2}$$

C.1 Função de Custo da Abordagem de Peng e Eles

A função de custo utilizada em [EPD94] [EKP98a] aplica-se a um processo de partição que visa gerar uma partição de *hardware* e outra de *software*. A função favorece o agrupamento dos objectos que possuem mais características em comum e apresentam uma troca de informação elevada.

A abordagem ao problema de partição começa com a reformulação da especificação do sistema, através dum processo em que se extraem as zonas que apresentam a maior carga computacional. Depois desta reformulação gera-se um grafo, em que os nodos coincidem com os objectos da especificação[1] e os arcos representam a comunicação entre objectos. Os nodos o_i do grafo são anotados com as métricas M^i_{n1} e M^i_{n2} e as ligações a_{ij} são anotadas com M^{ij}_{a1} e M^{ij}_{a2}. A partição do grafo em dois componentes, um de *software* (S) e outro de *hardware* (H), processa-se com o algoritmo de *simulated annealing*. O processo de partição é orientado por um função de custo que inclui três termos, como consta na equação C.3.

$$Fcusto = K_1 * M_{comHwSw} + K_2 * M_{compCom} + K_3 * M_{adeqImp} \tag{C.3}$$

[1] Um objecto coincide com um processo da especificação em VHDL do sistema.

Cada um dos três termos associa um peso a uma métrica. Os pesos são as constantes K_1 a K_3 e são escolhidos de modo a estabelecer uma relação equilibrada entre a contribuição das três métricas. O significado das métricas é o seguinte:

- $\boldsymbol{M_{comHwSw}}$ mede a comunicação entre *hardware* e *software* e é definida pela equação C.4;

$$M_{comHWSW} = \sum_{a_{ij} \in CorteHwSw} M_{a1}^{ij} \tag{C.4}$$

  em que $CorteHwSw$ define o conjunto das ligações entre a partição de *hardware* e a partição de *software*; M_{a1}^{ij} depende da intensidade de comunicação entre objectos, e será definida adiante;

- $\boldsymbol{M_{compCom}}$ quantifica o valor médio da relação entre comunicação e computação nos objectos atribuídos a *hardware* e é definida pela equação C.5;

$$M_{compCom} = \frac{\sum_{o_i \in H} \frac{\sum_{\exists\ a_{ij}} M_{a2}^{ij}}{M_{n1}^{i}}}{N_h} \tag{C.5}$$

  em que N_h é o número de objectos na partição de *hardware*; M_{a2}^{ij} mede o número de interacções entre objectos e M_{n1}^{i} mede a carga computacional desse objecto; M_{a2}^{ij} e M_{n1}^{i} serão descritas adiante;

- $\boldsymbol{M_{adeqImp}}$ mede a inadequação dos objectos para o tipo de implementação a que foram atribuídos; quantos mais forem os objectos que, sendo adequados a uma dada implementação (*hardware* ou *software*), estiverem atribuídos à implementação complementar (*software* ou *hardware*), maior será o valor desta métrica; $M_{adeqImp}$ é definida pela equação C.6;

$$M_{adeqImp} = \frac{\sum_{o_i \in S} M_{n2}^{i}}{N_s} - \frac{\sum_{o_i \in H} M_{n2}^{i}}{N_h} \tag{C.6}$$

  em que N_s representa o número de objectos na partição de *software* e M_{n2}^{i} faz o balanço entre as características do objecto o_i, sendo definida adiante.

As métricas M_{a1}^{ij} e M_{a2}^{ij} atribuídas a cada ligação (ou arco) entre dois objectos o_i e o_j dependem da intensidade da comunicação entre os objectos, mas enquanto a primeira mede o total de informação trocada, a segunda mede apenas o número de interacções.

$$M_{a1}^{ij} = \sum_{ch \in Nch} Nbits_{ch} * Freq_{ch} \qquad M_{a2}^{ij} = \sum_{ch \in Nch} Freq_{ch} \tag{C.7}$$

em que Nch é o conjunto de canais utilizados na comunicação, $Nbits_{ch}$ é a largura do canal ch e $Freq_{ch}$ é a intensidade de comunicação no canal ch.

A métrica M^i_{n1}, anotada no nodo que representa o objecto o_i, mede a carga computacional desse objecto, definida à custa do número de execuções e do peso computacional de cada operação contida no objecto o_i.

A métrica M^i_{n2} faz o balanço entre as características do objecto o_i que o tornam adequado para uma implementação de *hardware* e as características que o tornam adequado para uma implementação de *software*. M^i_{n2} é definida por

$$M^i_{n2} = c_1 * M^i_{ccr} + c_2 * M^i_{unif} + c_3 * M^i_{pp} - c_4 * M^i_{sw} \tag{C.8}$$

onde

- $\boldsymbol{M^i_{ccr}}$ mede a carga computacional relativa do objecto o_i e pode ser definida à custa do número de execuções e do peso computacional de cada operação contida no objecto o_i;
- $\boldsymbol{M^i_{unif}}$ é uma medida da uniformidade de operações no objecto o_i, definida como o quociente entre o total de operações e o número de operações distintas no objecto o_i;
- $\boldsymbol{M^i_{pp}}$ é uma medida do potencial de paralelismo existente no objecto o_i, definida como o quociente entre o total de operações e o tamanho do caminho mais longo[2] no objecto o_i;
- $\boldsymbol{M^i_{sw}}$ é uma medida da quantidade de operações do objecto o_i que são adequadas para serem implementadas em *software*, definida como o quociente entre uma soma, para a qual contribuem todas as operações adequadas a *software*, e o total de operações do objecto o_i; a contribuição de cada operação é uma constante que representa o grau de adequação dessa operação para uma implementação em *software*;
- $\boldsymbol{c_1}$ a $\boldsymbol{c_4}$ são constantes.

Durante a pesquisa da solução óptima, o algoritmo iterativo de partição procura minimizar o valor devolvido pela função de custo e simultaneamente respeitar os limites impostos ao espaço ocupado em *hardware* e em *software*.

C.2 Função de Custo para Sistemas Tempo Real

Nos sistemas embebidos tempo real o cumprimento dos requisitos temporais influencia de forma determinante o processo de partição. Nestes sistemas, em vez de se aplicar uma função

[2] O tamanho dum caminho é medido em número de operações.

de custo que combina várias métricas, é pertinente aplicar separadamente um conjunto de métricas (equação C.9), que representam os aspectos temporais ou não funcionais do sistema e controlam a exploração do espaço de projecto. A abordagem [DH94] descreve como é que através duma métrica M_p, designada por factor de exequibilidade, se verifica se um sistema é ou não exequível.

$$Fcusto = \overrightarrow{M} = \begin{bmatrix} M_1 \\ \dots \\ M_p \\ \dots \\ M_n \end{bmatrix} \tag{C.9}$$

O processo de partição em *hardware* e *software* recebe como entradas a descrição do sistema segundo um conjunto de funções do tipo *time-critical* e os requisitos/limitações associados a cada função. A partição será então um processo de selecção de microprocessadores e dispositivos de *hardware* e de atribuição de tarefas aos componentes seleccionados, por forma a optimizar a relação custo/desempenho da implementação e atingir os requisitos.

Nos sistemas de tempo real as características das funções tipo *time-critical* são geralmente especificadas por um trio (a, d, t), em que a é o instante de activação, d é o instante em que a função deve terminar a actividade e t é o intervalo entre repetições da função. Como o cumprimento dos requisitos por parte das funções implementadas em *hardware* é mais fácil de verificar, o esforço concentra-se nas funções implementadas em *software*. Neste caso, pode acontecer que haja várias tarefas a competir pelos mesmos recursos do microprocessador, não se garantindo assim que todas as tarefas sejam concluídas dentro de tempo.

O factor de exequibilidade M_p, para um microprocessador p, é definido por

$$M_p = \begin{cases} \frac{TR_p - TR_L}{TR_U - TR_L} & , \; se \; (TR_p - TR_L) < (TR_U - TR_L) \\ 1 & , \; \text{nos outros casos} \end{cases} \tag{C.10}$$

onde

- $\boldsymbol{TR_p}$ define o débito do microprocessador p;
- $\boldsymbol{TR_T}$ representa o valor mínimo de TR_p para que o microprocessador p consiga escalonar todas as tarefas que lhe foram atribuídas; TR_U e TR_L são o limite superior e inferior para TR_T, respectivamente; os valores de TR_U e TR_L são definidos por

$$TR_U = [N * (2^{1/N} - 1)]^{-1} * \sum_{i=1}^{N} \frac{c_i}{d_i - a_i} \tag{C.11}$$

$$TR_L = max_{n=1}^{N} \left\{ \; \sum_{i=1}^{n} \frac{k_i * c_i}{d_n - a_i} \quad , \quad \sum_{i=1}^{n} \frac{h_i * c_i}{d_n - a_n} \; \right\} \tag{C.12}$$

em que

- $\boldsymbol{N}$ é o número de tarefas atribuídas ao microprocessador p;
- $\boldsymbol{F_i} = (a_i, d_i, t_i)$ é uma função da descrição do sistema, crítica em termos temporais, correspondente à tarefa de *software* T_i com c_i instruções;
- a equação C.12 pressupõe que as tarefas T_i estão ordenadas por ordem crescente do valor de d_i, logo d_n corresponde à tarefa com conclusão mais posterior;
- $\boldsymbol{k_i}$ é quantificado por

$$k_i = \begin{cases} \lceil \frac{d_n - a_i}{t_i} \rceil & , \; se \; \lfloor \frac{d_n - a_i}{t_i} \rfloor * t_i + d_i \leq d_n \\ \lfloor \frac{d_n - a_i}{t_i} \rfloor & , \text{ nos outros casos} \end{cases} \tag{C.13}$$

- $\boldsymbol{h_i}$ é dado por

$$h_i = \begin{cases} k_i - \lceil \frac{a_n - a_i}{t_i} \rceil & , \; se \; (a_i < a_n) \\ k_i & , \text{ nos outros casos} \end{cases} \tag{C.14}$$

Quando $TR_U \leq TR_p$ todas as tarefas atribuídas a p serão escalonadas, mas quando $TR_L > TR_p$ é garantido que nem todas as tarefas serão escalonadas. Se $TR_p > TR_T$ é possível encontrar um escalonamento para todas as tarefas atribuídas a p, senão impõe-se um requisito a TR_T através da inequação $TR_L \leq TR_T \leq TR_U$.

Se $M_p = 1$ o conjunto de tarefas atribuídas a p é exequível, se $M_p \leq 0$ o conjunto de tarefas não é exequível e se $0 < M_p < 1$ a probabilidade do conjunto de tarefas ser exequível é M_p.

O factor de exequibilidade M_p é utilizado como atributo e como requisito no algoritmo de partição. No caso de ser um requisito, as alternativas de partição com $M_p \leq 0$ são consideradas inválidas. No caso de ser um atributo, a alternativa de partição escolhida será aquela que apresentar o maior factor M_p.

C.3 Função de Custo da Abordagem Cosyma

Na abordagem Cosyma, o processo de partição aplica um algoritmo iterativo e parte duma solução exequível em relação aos requisitos de desempenho [EHB93]. O problema de optimização da solução de partição resolve-se aumentando de forma exagerada o custo das soluções que excedem os requisitos de desempenho e diminuindo de forma acentuada o custo das soluções em que o desempenho melhora. Evita-se assim terminar o processo de partição em soluções não exequíveis.

Para efectuar a partição de sistemas muito complexos é conveniente reduzir o espaço de projecto, para tornar o problema tratável. Na abordagem Cosyma, a redução do espaço de

projecto é conseguida através dum processo designado por extracção de *hardware*. A extracção de *hardware* opera com uma função de custo que atribui um custo inferior às soluções que implementam em *hardware* as partes do sistema mais adequadas para esse tipo de implementação. Podem aplicar-se diferentes funções, em paralelo ou sequência, para identificar as partes mais adequadas a diferentes arquitecturas alvo.

Uma das funções de custo desenvolvidas identifica as partes do sistema que apresentam uma carga computacional elevada, nomeadamente ciclos. Por simulação e *profiling* identificam-se os objectos que apresentam a carga computacional mais elevada, depois determina-se o número de vezes que cada objecto é executado, a aceleração obtida com o deslocamento dos objectos para *hardware* e as respectivas perdas devido à comunicação entre *hardware* e *software*.

Em vez de se definir o custo duma solução de partição, define-se o incremento de custo que resulta do deslocamento dum objecto o_i de *software* para *hardware* (equação C.15).

$$\delta Fcusto_i = K_1 * f(M_1) * M_2 * (M_3 + M_4 - M_5 - M_6) \tag{C.15}$$

O incremento de custo varia exponencialmente com o grau de incumprimento do requisito de desempenho T_r, definido pela diferença entre o requisito e a estimativa desse requisito na solução *hardware/software* (T_e). O incremento de custo depende ainda do tempo que o objecto o_i deslocado de *software* para *hardware* demora a executar em n unidades funcionais ($T^i_{execHw}(n)$), do tempo adicional para comunicação com o_i (T^i_{com}), do intervalo em que existia sobreposição entre a execução de o_i e a execução de outros objectos em *software* ($T^i_{parHwSw}$), do tempo que o_i demorava a executar em *software* (T^i_{execSw}) e do número de vezes que o objecto é executado ($Freq_i$). O incremento do custo $\delta Fcusto_i$, resultante do deslocamento do objecto o_i de *software* para *hardware*, é dado pela equação C.16.

$$\delta Fcusto_i = sinal(T_r - T_e) * e^{\frac{T_r - T_e}{K_t}} * Freq_i * (T^i_{execHw}(n) + T^i_{com} - T^i_{parHwSw} - T^i_{execSw}) \tag{C.16}$$

em que K_t é uma constante.

Dado que para estimar a comunicação *hardware/software* adicional, no caso de existirem m objectos, é preciso analisar $(2^m - 1)$ situações, limita-se a análise aos objectos adjacentes no grafo de fluxo de controlo. A partir do conjunto de variáveis acedidas pelo objecto o_i e que são definidas fora dele ($in(o_i)$) e do conjunto de variáveis definidas no objecto ($out(o_i)$), estabelece-se um limite superior para a comunicação introduzida ao deslocar um objecto para *hardware*. Supondo que o objecto o_i foi deslocado para *hardware* e que os objectos que o precedem no grafo de fluxo ficam em *software*, a estimativa para o número adicional de variáveis a transferir de *software* para *hardware* é

$$\delta in(o_i) = in(o_i) \cap \{ \bigcup_{o_k \in \ antecessores(o_i)} out(o_k) \} \tag{C.17}$$

De forma idêntica, a estimativa para o incremento de variáveis a deslocar de *hardware* para *software* é definida por

$$\delta out(o_i) = out(o_i) \cap \{ \bigcup_{o_k \in \ antecessores(o_i)} in(o_k) \} \tag{C.18}$$

O tempo T^i_{com} é aproximado por $\delta in(o_i)$ mais $\delta out(o_i)$, quando se despreza a comunicação dentro do *hardware*.

Em [HE98] é proposta uma função de custo melhorada para a abordagem Cosyma, que tal como a anterior também coopera com um algoritmo de partição iterativo. Como o objectivo é reduzir ao mínimo o tempo de execução (desempenho) e o espaço ocupado em *hardware*, a função inclui um termo para cada uma destas métricas (equação C.19). Nesta equação, B designa o modelo do sistema e $f_2(M_1)$ é um factor aplicado ao espaço (M_2), dependente do desempenho (M_1). Adicionado algum detalhe obtém-se a função de custo da equação C.20.

$$Fcusto(B) = K_1 * f_1(M_1) + K_2 * f_2(M_1) * M2 \tag{C.19}$$

$$Fcusto(B) = K_T * T_{execNorm}(B) + K_{area} * \omega_{area} * \frac{Area_{HW}(B)}{\overline{A}} \tag{C.20}$$

O termo $Area_{HW}(B)$ é a estimativa do espaço ocupado em *hardware*, calculada com a equação 4.24, e $\overline{A}$ é um factor de normalização do espaço ocupado em *hardware*. O termo $T_{execNorm}(B)$ representa o sucesso com que se atinge o desempenho exigido ao sistema e é definido na equação C.21. Nesta equação, $T_{exec}(B)$ é a estimativa para o tempo de execução do sistema e $reqT$ é o requisito de desempenho.

$$T_{execNorm}(B) = \frac{\mid T_{exec}(B) - reqT \mid}{reqT} + 1 \tag{C.21}$$

Os factores K_T e K_{area} são constantes, enquanto ω_{area} é um factor que varia dinamicamente, ao longo do processo de partição, com o grau de aproximação ao desempenho exigido ao sistema. O comportamento de ω_{area}, tal como é definido pela equação C.22, garante que o peso do espaço no custo da solução de partição é nulo quando se está longe do requisito de desempenho ($\omega_{area} = 0$) e que à medida que o tempo de execução se aproxima do requisito de desempenho, o peso do espaço aumenta cada vez mais. Quando se atinge o desempenho exigido ao sistema, o valor de ω_{area} assume o valor máximo e o peso do espaço torna-se

dominante na função de custo. O valor da constante T_{th} é seleccionado de acordo com alguma experiência, mas deve ter-se em atenção que a influência de T_{th} sobre ω_{area} é como se ilustra na figura C.1 e que não faz sentido escolher T_{th} maior do que $reqT$.

$$\omega_{area} = \begin{cases} \frac{T_{exec}(B) - T_{th}}{reqT - T_{th}} & , \text{ se } T_{th} \leq T_{exec}(B) \leq reqT \\ \frac{2 * reqT - T_{th} - T_{exec}(B)}{reqT - T_{th}} & , \text{ se } reqT \leq T_{exec}(B) \leq (2 * reqT - T_{th}) \\ 0 & , \text{ nos outros casos} \end{cases} \tag{C.22}$$

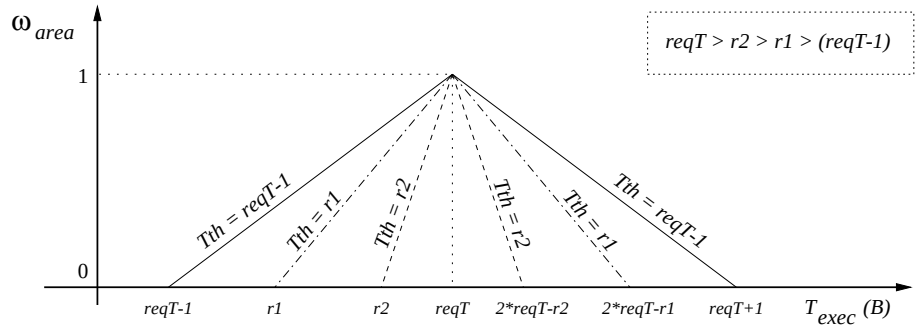


Figura C.1: Comportamento do factor variável ω_{area} aplicado ao espaço em *hardware*, para alguns valores de T_{th}.

Com a função de custo apresentada, na qual se aplica à métrica de espaço um peso que varia dinamicamente com a métrica de desempenho, obtêm-se soluções de partição que exigem menos recursos de *hardware* e atingem o mesmo desempenho que as soluções obtidas com uma função de custo equivalente, que não use o factor variável ω_{area}.

Apêndice D

Ferramenta de Edição e Partição de Grafos PSMfg: *parTiTool*

Este apêndice ilustra de forma resumida as potencialidades da ferramenta *parTiTool* que permite visualizar, editar e efectuar a partição de grafos PSMfg. A ferramenta possui ainda a capacidade de analisar todos os caminhos dum grafo, por forma a detectar erros na sua estrutura, e visualizar o resultado do processo de partição dum grafo (figura D.1). A maioria das operações envolvidas na edição e visualização dos grafos provém das classes `GRAPH` e `GraphWin` da biblioteca LEDA [MN99] [MNSU00].

A figura D.2 mostra como se insere num grafo um nodo correspondente a um estado programa ou variável do sistema. A forma de visualizar ou editar os parâmetros associados a um estado programa ou variável, que na maior parte dos casos correspondem a métricas envolvidas no processo de partição, é ilustrada nas figuras D.3 e D.4, respectivamente. Presentemente, a ferramenta *parTiTool* só obtém de forma automática a atribuição dos objectos às partições e os parâmetros assinalados com "(*)" nas figuras D.3 e D.4. Como se constata destas figuras, alguns parâmetros são guardados sob a forma de lista, que é o que acontece com as variáveis que um estado programa lê, escreve ou exigem um multiplexador na entrada e os estados programa que lêem ou escrevem uma variável. Por exemplo, a edição e visualização das variáveis escritas por um estado programa processa-se como mostram as figuras D.5 e D.6.

Para efectuar a partição dum grafo PSMfg a ferramenta *parTiTool* dispõe duma interface simples onde se inserem os parâmetros necessários à construção da solução de partição inicial e ao posterior processo de partição iterativo. Como ilustra a figura D.7, para uma sessão de partição é possível seleccionar o requisito de desempenho imposto ao sistema, controlar o número de iterações do processo iterativo e o tamanho das pesquisas, introduzir os parâmetros relacionados com a obtenção da solução inicial de cada pesquisa, definir o validade dos tabus e o método a aplicar na escolha da terceira alternativa de deslocamento no algoritmo de pesquisa tabu.

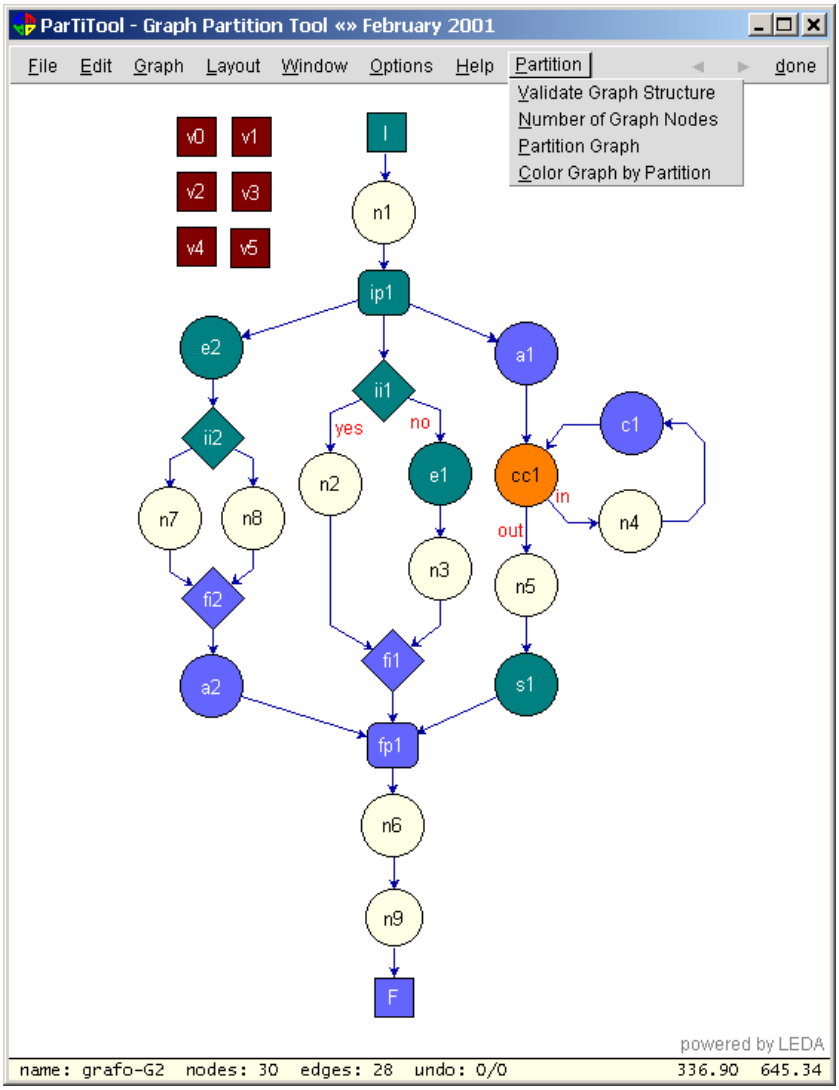


Figura D.1: A janela principal da ferramenta *parTiTool*.

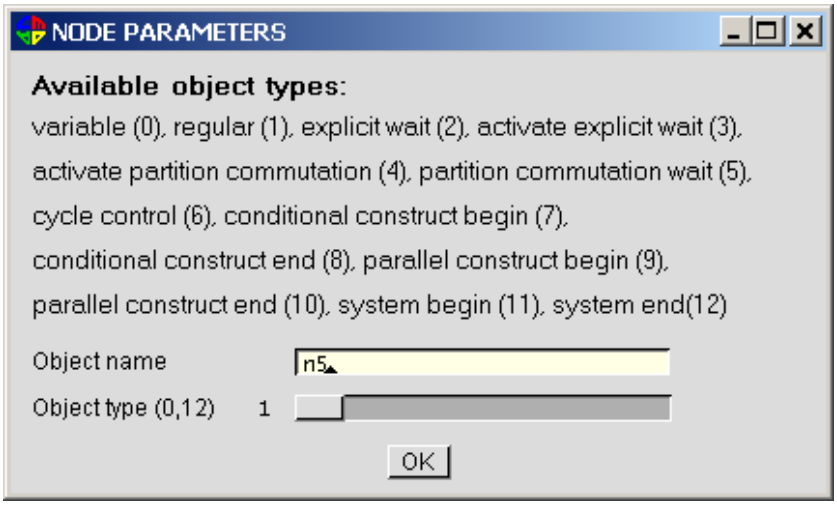


Figura D.2: Criação dum nodo do grafo PSMfg.

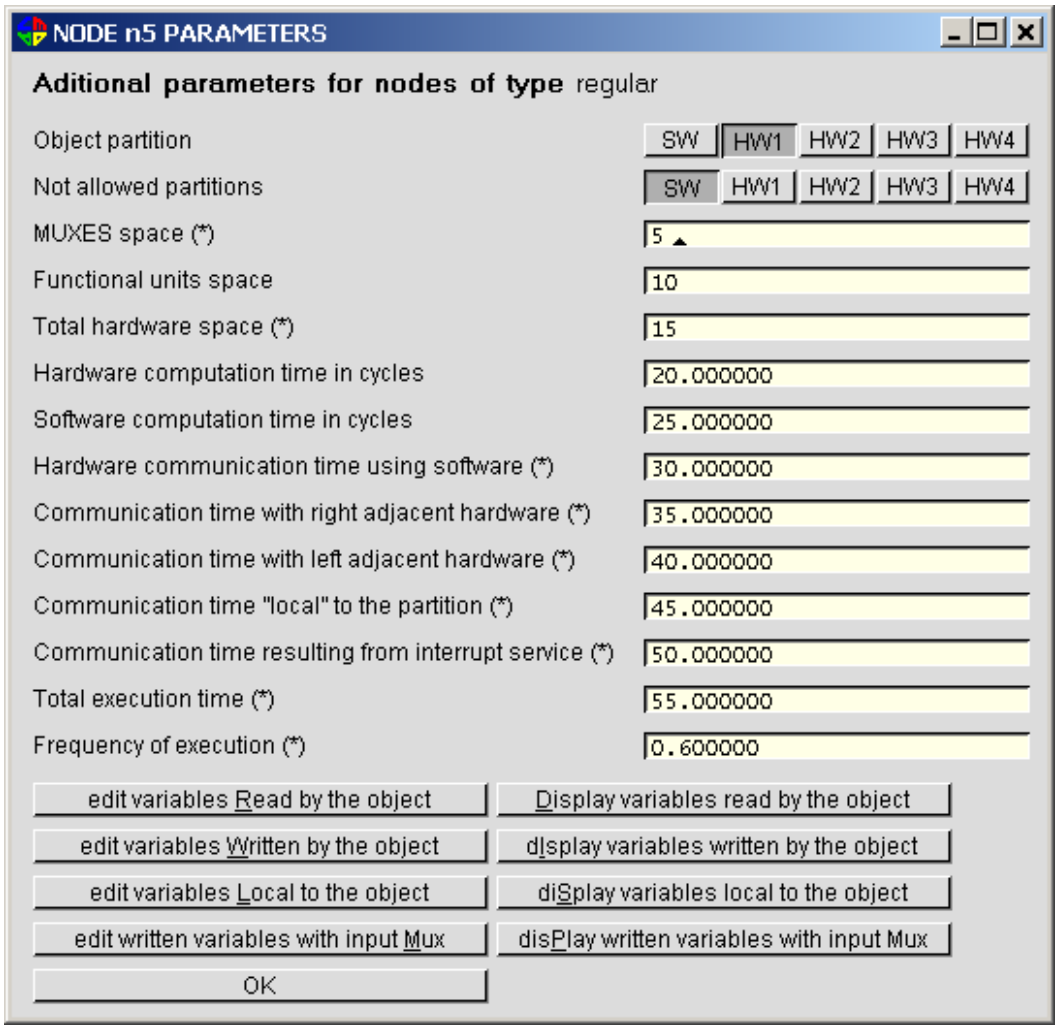


Figura D.3: Visualização e edição dos parâmetros dum nodo do tipo *normal*.

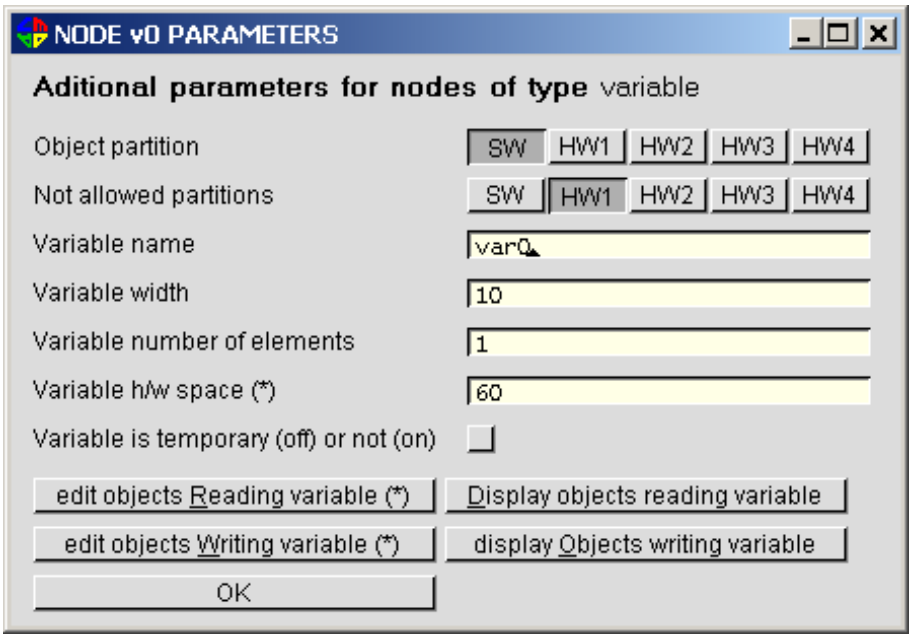


Figura D.4: Visualização dos parâmetros dum nodo do tipo *variavel*.

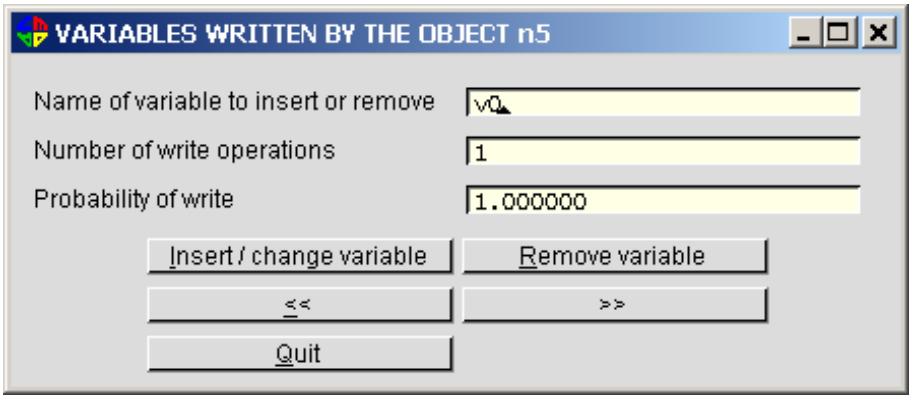


Figura D.5: Edição das variáveis escritas por um nodo do tipo *normal*.

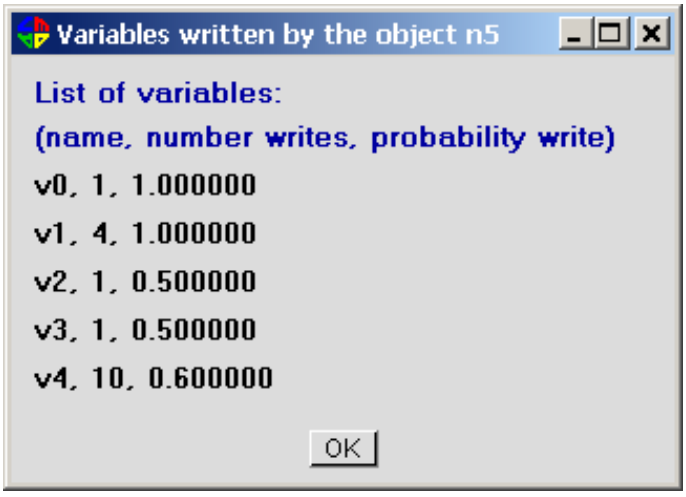


Figura D.6: Visualização das variáveis escritas por um nodo do tipo *normal*.

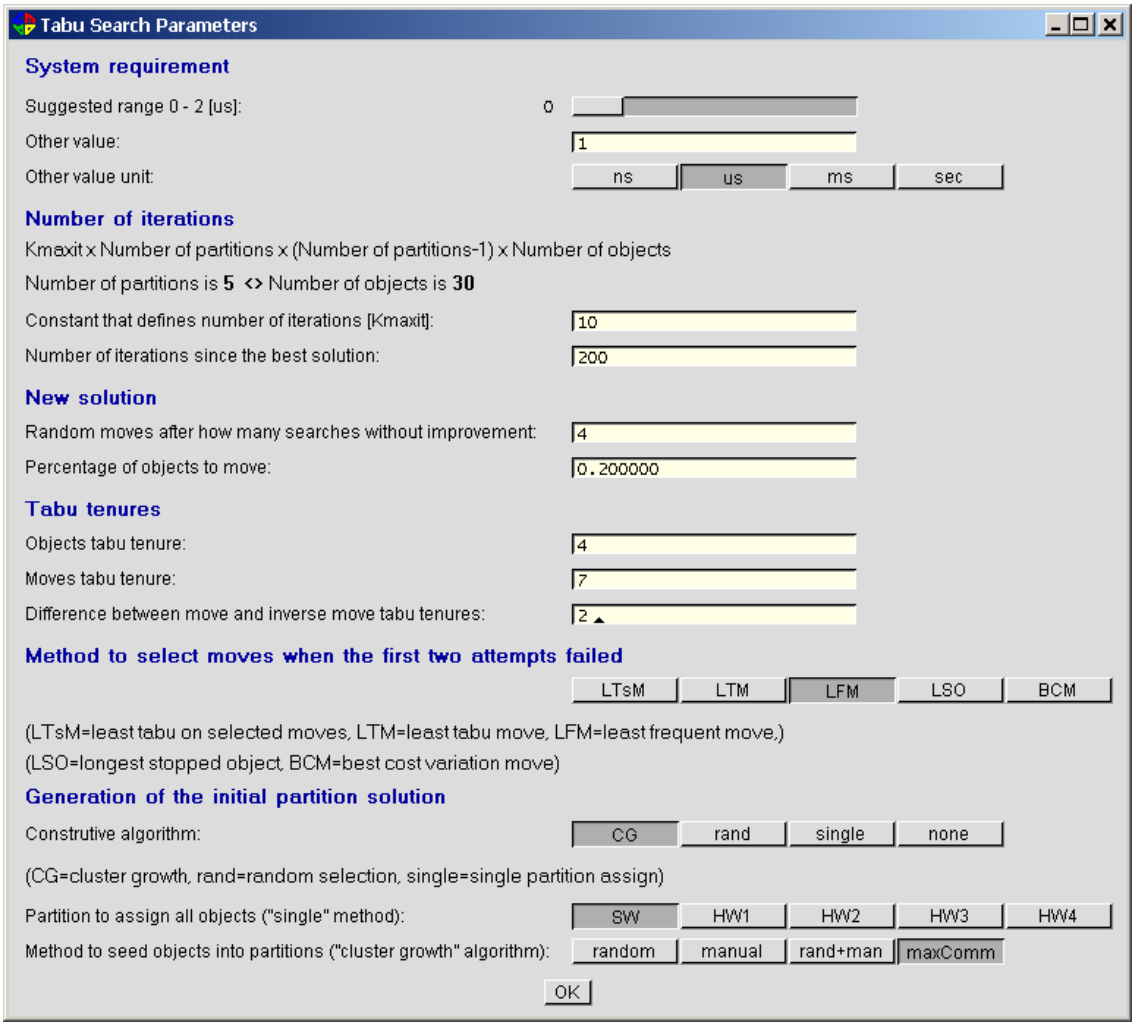


Figura D.7: Selecção dos parâmetros para o processo de partição dum grafo.

Apêndice E

Implementação do Algoritmo de Crescimento de Grupos

E.1 Função de Proximidade

selectBestObjAssign $(O, obj) \equiv$

```
se (obj é uma variável) então

    // O objecto obj é um mau candidato para hardware
    se (areaHWvar(obj) > LIMITE_AREA) E (atribuição de obj a SW não é proibida) então
        pSel = SW

    senão // O objecto obj não é um mau candidato para hardware
        // Calcular a comunicação entre obj e cada uma das partições → epsCG
        rdWrPStatesCost (obj, epsCG)

        // Obter o valor mínimo e máximo da comunicação entre obj e cada uma das partições
        minEps = minimo (epsCG)
        maxEps = maximo (epsCG)
        dEps = maxEps - minEps

        // Usando epsCG calcular a proximidade para cada partição → HWSWvarsCost
        para (cada partição np) fazer
            HWSWvarsCost[np] = (maxEps - epsCG[np]) / dEps
        fpara

        // Ordenar por ordem crescente de valor o array HWSWvarsCost
        // e guardar a ordem das partições no array partHWSWvarsCost
        increaseSortArray (HWSWvarsCost, partHWSWvarsCost)

        // Seleccionar para obj o deslocamento com o menor valor de HWSWvarsCost
        para (cada partição np) fazer
            p = partHWSWvarsCost[np]

            // A melhor atribuição não é proibida
            se (atribuição de obj a p não é proibida) então
                pSel = p
                Sair do ciclo
            fse
        fpara
    fse
```

Figura E.1: Selecção da melhor atribuição para um objecto, durante o processo de construção duma solução de partição com o algoritmo de crescimento de grupos *(parte 1)*.

```
senão // O objecto obj é um estado programa

   // Calcular o espaço ocupado por cada partição de hardware → areaCG
   calcAreaCGrowth (areaCG)

   // Obter o valor mínimo e máximo do espaço ocupado pelas partições de hardware
   minArea = minimo (areaCG)
   maxArea = maximo (areaCG)
   dArea = maxArea - minArea

   // Calcular a comunicação entre obj e cada uma das partições → varsCG
   rdWrVariablesCost (obj, varsCG)

   // Obter o valor mínimo e máximo da comunicação entre obj e cada uma das partições
   minVars = minimo (varsCG)
   maxVars = maximo (varsCG)
   dVars = maxVars - minVars

   // Calcular a proximidade utilizada na selecção da melhor partição de hardware → HWcost
   para (cada partição de hardware np) fazer
      HWcost[np] = CKvars * ((maxVars - varsCG[np])/dVars) +
                   CKarea * ((areaCG[np] - minArea)/dArea )
   fpara

   // Ordenar por ordem crescente de valor o array HWcost e
   // guardar a ordem das partições no array partHWcost
   increaseSortArray (HWcost, partHWcost)

   // Calcular o tempo de computação mínimo e máximo de obj → minTcomp e maxTcomp

   minTcomp = minimo (TexecHW(obj), TexecSW(obj))
   maxTcomp = maximo (TexecHW(obj), TexecSW(obj))
   dTcomp = maxTcomp - minTcomp

   // Calcular a proximidade que se utiliza para optar entre uma partição de software e de hardware
   // → HWSWcost

   para (cada partição np) fazer
      se (np = SW) então
         HWSWcost[np] = CKtcomp * ((TexecSW(obj) - minTcomp)/dTcomp) +
                        CKvars * ((maxVars - varsCG[np])/dVars)
      senão
         HWSWcost[np] = CKtcomp * ((TexecHW(obj) - minTcomp)/dTcomp) +
                        CKvars * ((maxVars - varsCG[np])/dVars)
      fse
   fpara

   // Ordenar por ordem crescente de valor o array HWSWcost e
   // guardar a ordem das partições no array partHWSWcost

   increaseSortArray (HWSWcost, partHWSWcost)
```

Figura E.2: Selecção da melhor atribuição para um objecto, durante o processo de construção duma solução de partição com o algoritmo de crescimento de grupos *(parte 2)*.

```
    // Seleccionar para obj o deslocamento com o menor valor de HWSWcost

    para (cada partição np) fazer
       p = partHWSWcost[np]

       // A melhor atribuição não é proibida
       se (atribuição de obj a p não é proibida) então
          pSel = p
          Sair do ciclo
       fse
    fpara

    // Se a melhor atribuição for uma partição de hardware,
    // seleccionar a partição a que corresponde o menor valor de HWcost

    se (pSel ≠ SW) então
       para (cada partição de hardware np) fazer
          p = partHWcost[np]

          // A melhor atribuição não é proibida
          se (atribuição de obj a p não é proibida) então
             pSel = p
             Sair do ciclo
          fse
       fpara
    fse
fse

devolver (pSel)
```

Figura E.3: Selecção da melhor atribuição para um objecto, durante o processo de construção duma solução de partição com o algoritmo de crescimento de grupos *(parte 3)*.

Apêndice F

Estimação do Tempo de Comunicação dos Estados Programa

Este apêndice complementa a apresentação do processo de estimação do tempo de comunicação entre estados programa e variáveis, iniciada na secção 7.4.3. Recorda-se que a transacção envolvida na leitura ou escrita duma variável recai num dos seguintes tipos:

- ◇ leitura/escrita dentro duma partição de *hardware* ou de *software*;
- ◇ leitura/escrita de *hardware* para/por *software*;
- ◇ leitura/escrita de *software* para/por *hardware*;
- ◇ leitura/escrita de *hardware* não adjacente para/por *hardware*;
- ◇ leitura/escrita de *hardware* adjacente para/por *hardware*.

Sendo que o processo de estimação do tempo de comunicação envolvido nas operações de (i) leitura duma variável localizada em *hardware* para *software*, (ii) leitura duma variável atribuída a *software* para *hardware* e (iii) leitura/escrita entre partições de *hardware* adjacentes já foi apresentado na secção 7.4.3.

Cada variável v_k que seja lida ou escrita por um estado programa o_i, contribui para o tempo de comunicação de o_i com um tempo do tipo

$$Tcom_i(v_k) = Tread_i(v_k) * numLeituras_i(v_k) * probLeitura_i(v_k) \tag{F.1}$$

ou

$$Tcom_i(v_k) = Twrite_i(v_k) * numEscritas_i(v_k) * probEscrita_i(v_k) \tag{F.2}$$

em que $Tread_i(v_k)$ ($Twrite_i(v_k)$) é tempo necessário para efectuar uma operação de leitura (escrita) da variável v_k, $numLeituras_i(v_k)$ ($numEscritas_i(v_k)$) é número de vezes que a

variável v_k é lida (escrita) por o_i e $probLeitura_i(v_k)$ ($probEscrita_i(v_k)$) é a probabilidade de o_i ler (escrever) a variável v_k.

F.1 Leitura Dentro duma Partição de *Hardware*

À leitura duma variável v_k simples, localizada na mesma partição de *hardware* que o estado programa o_i que efectua a leitura, corresponde um tempo de leitura nulo. Um tempo de leitura nulo equivale a um tempo de comunicação também nulo (equação F.3). Quando a variável lida é composta, o tempo necessário a uma leitura corresponde ao tempo que o *hardware* demora a ler uma posição duma memória RAM implementada em *hardware*, THW_ram_read definido na tabela 7.2. O tempo de comunicação resultante é dado pela equação F.5.

$$Tread_i(v_k) = 0 \quad \Rightarrow \quad Tcom_i(v_k) = 0 \tag{F.3}$$

$$Tread_i(v_k) = THW\_ram\_read \tag{F.4}$$

$$Tcom_i(v_k) = THW\_ram\_read * numLeituras_i(v_k) * probLeitura_i(v_k) \tag{F.5}$$

F.2 Escrita Dentro duma Partição de *Hardware*

O tempo necessário para escrever numa variável v_k simples, localizada na mesma partição de *hardware* que o estado o_i que efectua a escrita, coincide com o tempo que o *hardware* demora a escrever num registo de *hardware* ($THW\_register\_write$). No caso de a variável ser composta, o tempo de escrita em *hardware* passa a ser o tempo que demora a escrever numa posição duma memória RAM implementada em *hardware* (THW_ram_write). Com o tempo necessário a uma operação de escrita, $Twrite_i(v_k)$ definido na equação F.6, obtém-se o tempo de comunicação da equação F.7.

$$Twrite_i(v_k) = \begin{cases} THW\_register\_write & , \text{ se } v_k \text{ for uma variável simples} \\ THW\_ram\_write & , \text{ se } v_k \text{ for uma variável composta} \end{cases} \tag{F.6}$$

$$Tcom_i(v_k) = Twrite_i(v_k) * numEscritas_i(v_k) * probEscrita_i(v_k) \tag{F.7}$$

Os valores de $THW\_register\_write$ e THW_ram_write encontram-se na tabela 7.2.

F.3 Leitura e Escrita Dentro duma Partição de *Software*

O tempo necessário para que um estado programa o_i atribuído a *software* leia ($TreadSW2SW$) ou escreva ($TwriteSW2SW$) uma variável v_k também localizada em *software*, coincide com

o tempo que demora uma transacção dentro da partição de *software* (tabela 7.2). O tempo de comunicação associado a uma leitura e a uma escrita é definido pelas equações F.9 e F.11, respectivamente.

$$Tread_i(v_k) = TreadSW2SW \tag{F.8}$$

$$Tcom_i(v_k) = TreadSW2SW * numLeituras_i(v_k) * probLeitura_i(v_k) \tag{F.9}$$

$$Twrite_i(v_k) = TwriteSW2SW \tag{F.10}$$

$$Tcom_i(v_k) = TwriteSW2SW * numEscritas_i(v_k) * probEscrita_i(v_k) \tag{F.11}$$

F.4 Escrita em *Hardware* a Partir do *Software*

O número de transacções exigido por uma operação em que um estado o_i, atribuído a *software*, escreve uma variável v_k localizada em *hardware* é dado pela equação F.12.

$$nTransf(v_k) = \lceil \frac{larguraVar(v_k)}{pFPGA2SW} \rceil \tag{F.12}$$

onde $larguraVar(v_k)$ designa o número de bits da variável e $pFPGA2SW$ é a largura do canal de comunicação entre *hardware* e *software*.

O tempo despendido na operação de escrita é definido pela equação F.13. Aplicando este tempo na equação F.2 obtém-se o tempo de comunicação envolvido na escrita duma variável v_k por parte dum estado o_i.

$$\begin{aligned} Twrite_i(v_k) = nTransf(v_k) * TwriteSW2HW + (nTransf(v_k) - 1) * TandSW \ + \\ + \ nshifts(ordemHW, nTransf(v_k)) * TshiftSW \end{aligned} \tag{F.13}$$

em que

- $\boldsymbol{TwriteSW2HW}$ é o tempo que o *software* demora a escrever num registo de *hardware*, definido na tabela 7.2;
- $\boldsymbol{TandSW}$ e $\boldsymbol{TshiftSW}$ são o tempo de execução das operações *E-lógico* e *deslocação* em *software*, definidos na tabela 7.1;
- $\boldsymbol{nshifts(ordemHW, nTransf(v_k))}$ representa o tamanho dos deslocamentos envolvidos no acesso a uma variável v_k, localizada na partição de *hardware* cuja ordem é $ordemHW$, e em que $nTransf(v_k)$ é o número de transacções necessárias para aceder a essa variável.

F.5 Escrita em *Software* a Partir do *Hardware*

Tal como a leitura de *software* para *hardware*, a operação de escrita em *software* a partir do *hardware* também exige a participação do *software*. Resulta daqui que a análise da comunicação envolvida numa operação de escrita em *software* a partir do *hardware* é idêntica à análise efectuada na secção 7.4.3 para a operação de leitura de *software* por *hardware*. Deste modo, apenas se apresentam os aspectos que distinguem a escrita em *software* da leitura de *software*, em termos de comunicação.

A operação de escrita em *software* a partir do *hardware* continua a poder ser ilustrada pelas figuras 7.6 e 7.7, desde que se tenham em consideração as alterações a seguir apresentadas.

Para começar, a operação $read(v)$ passa a ser $write(v, val)$ em ambas as figuras. Quando o estado o_i que inicia a leitura não funciona em paralelo com outros estados atribuídos a *software*, o código que substitui a operação $write(v, val)$ no estado o_i é o que se inclui na figura F.1.

```
ss_j := val1; // val1=¬ss_j
wait until sh_j=val2;
```

Figura F.1: Código que substitui a operação $write(v, val)$ no estado que inicia uma operação de escrita em *software* a partir de *hardware*, quando se utiliza o mecanismo de auscultação.

O código que define a funcionalidade do estado o_{com} introduzido no fluxo de *software* encontra-se na figura F.2. Convém lembrar que uma operação de escrita em *software* a partir de *hardware* se transforma numa leitura do *hardware* a partir de *software*.

```
(B)  while (readHW(ss_j) != val1);
     v=readHW(addHw1);
     vs=readHW(addHw2); v=v [ ≪ | ≫ ] pFPGA2SW; v=v & vs;
     // bloco (B) nTransf() − 3 vezes
     vs=readHW(addHwN); v=v [ ≪ | ≫ ] pFPGA2SW; v=v & vs;
     writeHW(sh_j,val2);
```

Figura F.2: Código do estado o_{com} criado em *software* para executar a operação de escrita em *software* a partir de *hardware*, através do mecanismo de auscultação.

Para a duração duma operação de escrita em *software* a partir de *hardware* contribuem o tempo de execução das instruções que substituíram a operação $write(v, val)$ no estado o_i e o tempo de execução das instruções do estado o_{com} criado. A soma do tempo de execução dessas instruções resulta na expressão incluída na equação F.14. Aplicando esta expressão na equação F.2 obtém-se o tempo de comunicação envolvido na escrita duma variável v_k de

software por parte dum estado o_i de *hardware*, na situação em que o_i não funciona em paralelo com outros estados atribuídos a *software*.

$$\begin{aligned} Twrite_i(v_k) = {} & TwriteHW2HW + TwriteSW2HW + TcomparacaoSW + \\ + {} & (1 + nTransf(v_k)) * TreadHW2SW + (nTransf(v_k) - 1) * TandSW + \\ & + nshifts(ordemHW, nTransf(v_k)) * TshiftSW \end{aligned} \tag{F.14}$$

onde

- ***TwriteHW2HW*** designa o tempo que o *hardware* demora a escrever num registo de *hardware*, definido na tabela 7.2;
- ***TcomparacaoSW*** é o tempo que demora a comparação de duas variáveis inteiras em *software*, definido na tabela 7.1;
- ***TreadHW2SW*** é o tempo que o *software* demora a ler um registo de *hardware*, definido na tabela 7.2.

Quando a operação de escrita se processa através do mecanismo de interrupção, o código que substitui a operação $write(v, val)$ no estado o_i é apresentado na figura F.3 e o código do estado o_{RSI} que serve a interrupção encontra-se na figura F.4.

```
< idVAR2write := valy; > // Identificar a variável a escrever
interruptReg := enableVal; // Gerar uma interrupção
wait until sh_j=val1;
< idVAR2write := 0; >
```

Figura F.3: Código que substitui a operação $write(v, val)$ no estado que inicia uma operação de escrita em *software* a partir do *hardware*, quando a operação é executada através do mecanismo de interrupção.

O tempo de execução do estado o_{RSI} é definido pela equação F.16 e o tempo de comunicação associado a o_i é obtido com a equação F.18.

$$\begin{aligned} TwriteRSI(v_k) = {} & TgestaoRSI + 2 * TwriteSW2HW + TcomparacaoSW + \\ + {} & (f2 + nTransf(v_k)) * TreadHW2SW + (nTransf(v_k) - 1) * TandSW + \\ & + nshifts(ordemHW, nTransf(v_k)) * TshiftSW \end{aligned} \tag{F.15}$$

$$TexecRSI(v_k) = TwriteRSI(v_k) * numEscritas_i(v_k) * probEscrita_i(v_k) * FN_i \tag{F.16}$$

```
        [ cabeçalho duma RSI ]
        writeHW(interruptReg,disableVal); // Desactivar a interrupção
        < id=readHW(idVAR2write); >
        < if (id==valy) > {
            v=readHW(addHw1);
(B)         vs=readHW(addHw2); v=v [ ≪ | ≫ ] pFPGA2SW; v=v & vs;
            //  bloco (B) nTransf() − 3 vezes
            vs=readHW(addHwN); v=v [ ≪ | ≫ ] pFPGA2SW; v=v & vs;
            }
        writeHW(sh_j,val1); // val1=¬sh_j)
        [ rodapé duma RSI ]
```

Figura F.4: Código do estado o_{RSI} criado em *software* para servir a interrupção associada a um pedido de execução duma operação de escrita em *software* a partir do *hardware*.

$$Twrite_i(v_k) = f1 * TwriteHW2HW + TgestaoRSI + 2 * TwriteSW2HW +$$

$$+ \ TcomparacaoSW + nshifts(ordemHW, nTransf(v_k)) * TshiftSW +$$

$$+ \ (f2 + nTransf(v_k)) * TreadHW2SW + (nTransf(v_k) - 1) * TandSW \tag{F.17}$$

$$Tcom_i(v_k) = Twrite_i(v_k) * numEscritas_i(v_k) * probEscrita_i(v_k) \tag{F.18}$$

em que

- ***TgestaoRSI*** representa a soma do tempo envolvido na invocação duma rotina de serviço à interrupção (RSI), com o tempo usado no regresso ao programa interrompido e com o tempo de execução do cabeçalho mais o rodapé da RSI;
- $\boldsymbol{FN_i}$ é a frequência de execução do estado programa o_i;
- $\boldsymbol{f1}$ e $\boldsymbol{f2}$ são parâmetros que quantificam o facto de haver ou não múltiplas operações de escrita ou leitura via interrupção.

$$f1, f2 \ = \ \begin{cases} 1,0 & \rightarrow \text{ se o número de operações via interrupção} = 1 \\ 3,1 & \rightarrow \text{ se o número de operações via interrupção} > 1 \end{cases} \tag{F.19}$$

F.6 Leitura de *Hardware* Não Adjacente para *Hardware*

A operação de leitura de *hardware* não adjacente para *hardware* exige a participação do *software* e processa-se em duas fases: primeiro o *software* lê do *hardware* não adjacente e depois escreve em *hardware*. A operação de leitura de *hardware* não adjacente para *hardware* também é ilustrada pelas figuras 7.6 e 7.7, com a excepção de parte do código nelas incluído.

O código que substitui a operação $read(v)$ no estado o_i coincide com o código que se apresentou na figura 7.6, enquanto o código que define a funcionalidade do estado o_{com} introduzido no fluxo de *software* encontra-se na figura F.5.

```
(B)  while (readHW(ss_j) != val1);
     v1=readHW(addHw1r); v1=v1 [ ≪ | ≫ ] 2*pFPGA2SW; writeHW(addHw1w,v1);
     // bloco (B) nTransf() − 2 vezes
     v1=readHW(addHwNr); v1=v1 [ ≪ | ≫ ] 2*pFPGA2SW; writeHW(addHwNw,v1);
     writeHW(sh_j,val2);
```

Figura F.5: Código do estado o_{com} criado em *software* para executar uma operação de leitura de *hardware* não adjacente a partir de *hardware*, através do mecanismo de auscultação.

O tempo de execução duma operação de leitura de *hardware* não adjacente para *hardware* é expresso pela equação F.20. Aplicando esta expressão na equação F.1 obtém-se o tempo de comunicação envolvido na leitura duma variável v_k localizada em *hardware* não adjacente para um estado o_i de *hardware*, na situação em que o_i não funciona em paralelo com outros estados atribuídos a *software*.

$$Tread_i(v_k) = TwriteHW2HW + TcomparacaoSW + nTransf(v_k) * TshiftSW +$$
$$+ (nTransf(v_k)+1) * TreadHW2SW + (nTransf(v_k)+1) * TwriteSW2HW \qquad \text{(F.20)}$$

Quando a operação de leitura se processa através do mecanismo de interrupção, o código que substitui a operação $read(v)$ no estado o_i coincide com o código que se apresentou na figura 7.7 e o código do estado o_{RSI}, que serve a interrupção, encontra-se na figura F.6.

```
     [ cabeçalho duma RSI ]
     writeHW(interruptReg,disableVal); // Desactivar a interrupção
     < id=readHW(idVAR2read); >
     < if (id==valx) > {
(B)      v1=readHW(addHw1r); v1=v1 [ ≪ | ≫ ] 2*pFPGA2SW; writeHW(addHw1w,v1);
         // bloco (B) nTransf() − 2 vezes
         v1=readHW(addHwNr); v1=v1 [ ≪ | ≫ ] 2*pFPGA2SW; writeHW(addHwNw,v1);
         }
     writeHW(sh_j,val1); // val1=¬sh_j
     [ rodapé duma RSI ]
```

Figura F.6: Código do estado o_{RSI} criado em *software* para servir a interrupção associada a um pedido de execução duma operação de leitura de *hardware* não adjacente para *hardware*.

O tempo de execução do estado o_{RSI} é definido pela equação F.22 e o tempo de comunicação associado a o_i obtém-se com a equação F.24.

$$TreadRSI(v_k) = TgestaoRSI + nTransf(v_k) * TshiftSW +$$
$$+ (f2 + nTransf(v_k)) * TreadHW2SW + (2 + nTransf(v_k)) * TwriteSW2HW \quad \text{(F.21)}$$

$$TexecRSI(v_k) = TreadRSI(v_k) * numLeituras_i(v_k) * probLeitura_i(v_k) * FN_i \quad \text{(F.22)}$$

$$Tread_i(v_k) = f1 * TwriteHW2HW + TgestaoRSI + nTransf(v_k) * TshiftSW +$$
$$+ (f2 + nTransf(v_k)) * TreadHW2SW + (nTransf(v_k) + 2) * TwriteSW2HW \quad \text{(F.23)}$$

$$Tcom_i(v_k) = Tread_i(v_k) * numLeituras_i(v_k) * probLeitura_i(v_k) \quad \text{(F.24)}$$

com $f1$ e $f2$ a serem definidos pela equação F.19.

F.7 Escrita em *Hardware* Não Adjacente a Partir de *Hardware*

A operação de escrita em *hardware* não adjacente a partir de *hardware* também exige a participação do *software* e decorre em duas fases: na primeira o *software* lê do *hardware* e na segunda escreve no *hardware* não adjacente. Na análise da operação de escrita em *hardware* não adjacente a partir de *hardware* continua a utilizar-se as figuras 7.6 e 7.7, com a excepção de partes do código nelas incluído.

Quando a comunicação se processa com o mecanismo de auscultação, a operação $write(v, val)$ do estado o_i é substituída pelo código da figura F.1. Para efeitos de estimação, o código que define a funcionalidade do estado o_{com}, introduzido no fluxo de *software*, coincide como o código apresentado na figura F.5. Em termos de implementação, a diferença reside apenas nos endereços a usar nas operações de leitura e de escrita.

O tempo de execução duma operação de escrita em *hardware* não adjacente a partir de *hardware* é expresso pela equação F.25. Aplicando esta expressão na equação F.2 obtém-se o tempo de comunicação envolvido na escrita duma variável v_k localizada em *hardware* não adjacente por parte dum estado o_i de *hardware*, na situação em que o_i não funciona em paralelo com outros estados atribuídos a *software*.

$$Twrite_i(v_k) = TwriteHW2HW + TcomparacaoSW + nTransf(v_k) * TshiftSW +$$
$$+ (nTransf(v_k) + 1) * TwriteSW2HW + (nTransf(v_k) + 1) * TreadHW2SW \quad \text{(F.25)}$$

Quando a operação de escrita aplica o mecanismo de interrupção, o código que substitui a operação $write(v, val)$ no estado o_i coincide com o código da figura F.3. Na fase de estimação

do desempenho, pode usar-se para código do estado o_{RSI} o código da figura F.6. Em termos de implementação, as diferenças relativamente à figura F.6 reflectem-se apenas no identificador da variável a escrever ($valx$ em vez de $valy$) e nos endereços a usar nas operações de leitura e de escrita. Deste modo, o tempo de execução do estado o_{RSI} é definido pela equação F.27 e o tempo de comunicação associado a o_i obtém-se através da equação F.29. Comparando a equação F.21 com F.26 e a equação F.23 com F.28, verifica-se que as operações de escrita e de leitura da mesma variável, envolvendo duas partições de *hardware* não adjacentes, apresentam o mesmo tempo de execução.

$$TwriteRSI(v_k) = TgestaoRSI + nTransf(v_k) * TshiftSW + \\ + (f2 + nTransf(v_k)) * TreadHW2SW + (2 + nTransf(v_k)) * TwriteSW2HW \quad \text{(F.26)}$$

$$TexecRSI(v_k) = TwriteRSI(v_k) * numEscritas_i(v_k) * probEscrita_i(v_k) * FN_i \quad \text{(F.27)}$$

$$Twrite_i(v_k) = f1 * TwriteHW2HW + TgestaoRSI + nTransf(v_k) * TshiftSW + \\ + (f2 + nTransf(v_k)) * TreadHW2SW + (nTransf(v_k) + 2) * TwriteSW2HW \quad \text{(F.28)}$$

$$Tcom_i(v_k) = Twrite_i(v_k) * numEscritas_i(v_k) * probEscrita_i(v_k) \quad \text{(F.29)}$$

com $f1$ e $f2$ a serem definidos pela equação F.19.

Apêndice G

Implementação da Estimação do Tempo de Comunicação

G.1 Constantes Necessárias ao Cálculo do Tempo de Comunicação

- $\boldsymbol{pFPGA2SW}$ é a largura da ligação entre uma FPGA e o *software*;
- $\boldsymbol{nFPGAS}$ é o número de FPGAs;
- $\boldsymbol{nPinos_{i,j}}$, $\boldsymbol{nPinos_{j,i}}$ é a largura da ligação entre as FPGAs i e j (16 bits);
- $\boldsymbol{THW\_register\_write}$ é o tempo de execução duma operação de escrita dentro duma partição de *hardware* efectuada sobre uma variável simples;
- $\boldsymbol{THW\_ram\_write}$ é o tempo de execução duma operação de escrita dentro duma partição de *hardware* efectuada sobre uma variável composta;
- $\boldsymbol{THW\_ram\_read}$ é o tempo de execução duma operação de leitura dentro duma partição de *hardware* efectuada sobre uma variável composta;
- $\boldsymbol{TwriteSW2HW}$ é o tempo de execução duma operação de escrita duma variável simples de *hardware* a partir duma partição de *software*;
- $\boldsymbol{TreadHW2SW}$ é o tempo de execução duma operação de leitura de uma variável simples de *hardware* a partir duma partição de *software*;
- $\boldsymbol{TwriteHW2HW} = THW\_register\_write$;
- $\boldsymbol{TwriteSW2SW}$ é o tempo de execução duma operação de escrita dentro duma partição de *software*, considerando uma transacção entre registos;
- $\boldsymbol{TreadSW2SW}$ é o tempo de execução duma operação de leitura dentro duma partição de *software*, considerando uma transacção entre registos;

- ***TwriteHW2HWa*** é o tempo de execução duma operação de escrita a partir duma partição de *hardware* para um registo numa partição de *hardware* adjacente;
- ***TreadHWa2HW*** é o tempo de execução duma operação de leitura a partir dum registo duma partição de *hardware* adjacente para uma partição de *hardware*;
- ***TgestaoRSI*** contabiliza o tempo de invocação duma RSI, o tempo de regresso ao programa interrompido e o tempo de execução do cabeçalho e do rodapé duma RSI;
- ***TshiftSW*** é o tempo de execução duma operação de *deslocação* em *software*;
- ***TandSW*** é o tempo de execução duma operação *E-lógico* em *software*;
- ***TcomparacaoSW*** é o tempo de execução duma comparação de 2 variáveis inteiras em *software*.

G.2 Parâmetros de Entrada das Funções que Calculam o Tempo de Comunicação

- $\boldsymbol{v}$ é a variável deslocada de partição;
- $\boldsymbol{pold_v}$ é a partição a que pertencia a variável v;
- $\boldsymbol{pnew_v}$ é a nova partição da variável v;
- $\boldsymbol{o}$ é o estado programa que lê, ou escreve, a variável v;
- $\boldsymbol{p_o}$ é a partição do estado programa o; p_o coincide com a antiga partição de o se a actualização devido à mudança de variáveis for efectuada antes da actualização devido à mudança de estados programa; p_o coincide com a nova partição de o se a actualização devido à mudança de variáveis for efectuada depois da actualização devido à mudança de estados programa;
- $\boldsymbol{pold_o}$ é a partição a que pertencia o estado programa o;
- $\boldsymbol{pnew_o}$ é a nova partição do estado programa o;
- $\boldsymbol{p_v}$ é a partição da variável v; p_v coincide com a antiga partição de v se a actualização devido à mudança de estados programa for efectuada antes da actualização devido à mudança de variáveis; p_o coincide com a nova partição de v se a actualização devido à mudança de estados programa for efectuada depois da actualização devido à mudança de variáveis;
- ***RDouWR*** indica se o estado programa o lê (RD) ou escreve (WR) a variável v;

- $\boldsymbol{objParalelo[identificacao(o)]}$ é o estado programa do fluxo de *software* que funciona em paralelo com o, ou $NULL$ se não existir esse estado, ao qual se atribui o tempo da comunicação por interrupção efectuada por o;
- $\boldsymbol{nBits_v}$, ou $larguraVar_v$, é o número de bits dum elemento da variável v;
- $\boldsymbol{nWR_v}$, ou $numEscritas_v$, é o número de vezes que o estado programa o escreve a variável v;
- $\boldsymbol{pWR_v}$, ou $probEscrita_v$, é a probabilidade de o estado programa o escrever a variável v;
- $\boldsymbol{nRD_v}$, ou $numLeituras_v$, é o número de vezes que o estado programa o lê a variável v;
- $\boldsymbol{pRD_v}$, ou $probLeitura_v$, é a probabilidade de o estado programa o ler a variável v;
- $\boldsymbol{ordemPartHW}$ é a ordem da partição de *hardware* envolvida na comunicação, ou seja, é a partição do estado programa o ou da variável v;
- $\boldsymbol{nElement_v}$, ou $numElementosVar_v$, é o número de elementos da variável v;
- $\boldsymbol{FN_o}$ é a frequência de execução do estado programa o que lê, ou escreve, a variável v;
- $\boldsymbol{nAcessosIRQ}$ é o número de acessos, para ler ou escrever variáveis distintas, em que se utiliza o mecanismo de interrupção.

G.3 Equações para Calcular o Tempo de Escrita de Variáveis

```
// Hardware escreve em hardware (equação W1)

TcomHWwrHW (nElement_v, nWR_v, pWR_v) ≡
  se (nElement_v = 1) então
    Tcom = THW_register_write * nWR_v * pWR_v
  senão se (nElement_v > 1) então
    Tcom = THW_ram_write * nWR_v * pWR_v
  fse
  devolver (Tcom)
```

Figura G.1: Tempo de comunicação necessário para que um estado programa de *hardware* escreva uma variável de *hardware*.

```
// Software escreve em software (equação D1)

TcomSWwrSW (nWR_v, pWR_v) ≡
   Tcom = TwriteSW2SW * nWR_v * pWR_v
   devolver (Tcom)
```

Figura G.2: Tempo de comunicação necessário para que um estado programa de *software* escreva uma variável de *software*.

```
// Software escreve em hardware (equação D24)

TcomSWwrHW (nBits_v, nWR_v, pWR_v, ordemPartHW) ≡
   nt = ⌈nBits_v/pFPGA2SW⌉ // Número de transacções
   ns = nshifts(ordemPartHW, nt) // Número de deslocamentos
   Twrite = TwriteSW2HW * nt + ns * TshiftSW
   Tcom = Twrite * nWR_v * pWR_v
   devolver (Tcom)
```

Figura G.3: Tempo de comunicação necessário para que um estado programa de *software* escreva uma variável de *hardware*.

```
// Calcular o número de deslocamentos envolvidos no acesso a uma variável
//     op - ordem da partição de hardware envolvida na comunicação
//     nt - número de transacções necessárias para aceder à variável

nshifts (op, nt) ≡
   num_ntIs3[nFPGAS:1] = {3,2,3,2}
   num_ntIs2[nFPGAS:1] = {3,2,1,1}
   se (nt ≥ nFPGAS) então
      ns = nt − 1
   senão se (nt = 3) então
      ns = num_ntIs3[op]
   senão se (nt = 2) então
      ns = num_ntIs2[op]
   senão se (nt = 1) então
      ns = op − 1
   fse
   devolver (ns)
```

Figura G.4: Cálculo do número de deslocamentos envolvidos no acesso a uma variável.

```
// Hardware escreve em software, sem interrupções (equação B27)

TcomHWwrSW (nBits_v, nWR_v, pWR_v, ordemPartHW) ≡
   nt = ⌈nBits_v/pFPGA2SW⌉ // Número de transacções
   ns = nshifts(ordemPartHW, nt) // Número de deslocamentos
   Twrite = TwriteHW2HW + TwriteSW2HW + 2 * TcomparacaoSW + (2 + nt) * TreadHW2SW
   Twrite = Twrite + (nt − 1) * TandSW + ns * TshiftSW
   Tcom = Twrite * nWR_v * pWR_v
   devolver (Tcom)
```

Figura G.5: Tempo de comunicação necessário para que um estado programa de *hardware* escreva uma variável de *software* (sem usar interrupções).

// ***Hardware*** **escreve em** ***software*****, com interrupções** (equação B25)

$\boldsymbol{TcomHWwrSWirq}\ (nBits_v, nWR_v, pWR_v, ordemPartHW, nAcessosIRQ) \equiv$
se $(nAcessosIRQ = 1)$ então
$f1 = 1$
$f2 = 0$
senão se $(nAcessosIRQ > 1)$ então
$f1 = 3$
$f2 = 1$
fse
$nt = \lceil nBits_v / pFPGA2SW \rceil$ // Número de transacções
$ns = nshifts(ordemPartHW, nt)$ // Número de deslocamentos
$Twrite = f1 * TwriteHW2HW + TgestaoRSI + 2 * TwriteSW2HW + TcomparacaoSW$
$Twrite = Twrite + (f2 + nt) * TreadHW2SW + ns * TshiftSW + (nt - 1) * TandSW$
$Tcom = Twrite * nWR_v * pWR_v$
devolver $(Tcom)$

Figura G.6: Tempo de comunicação necessário para que um estado programa de *hardware* escreva uma variável de *software* (usando interrupções).

// ***Hardware*** **escreve em** ***software*****, com interrupções** (equação B26)

$\boldsymbol{TexecORSIHWwrSWirq}\ (nBits_v, nWR_v, pWR_v, ordemPartHW, nAcessosIRQ, FN_o) \equiv$
se $(nAcessosIRQ = 1)$ então
$f2 = 0$
senão se $(nAcessosIRQ > 1)$ então
$f2 = 1$
fse
$nt = \lceil nBits_v / pFPGA2SW \rceil$ // Número de transacções
$ns = nshifts(ordemPartHW, nt)$ // Número de deslocamentos
$Twrite = TgestaoRSI + 2 * TwriteSW2HW + TcomparacaoSW + (f2 + nt) * TreadHW2SW$
$Twrite = Twrite + ns * TshiftSW + (nt - 1) * TandSW$
$Tcom = Twrite * nWR_v * pWR_v * FN_o$
devolver $(Tcom)$

Figura G.7: Tempo de execução da rotina de serviço à interrupção, necessário para que um estado programa de *hardware* escreva uma variável de *software*.

// ***Hardware*** **escreve em** ***hardware*** **não adjacente, sem interrupções** (equação B42)

$\boldsymbol{TcomHWwrHWna}\ (nBits_v, nWR_v, pWR_v) \equiv$
$nt = \lceil nBits_v / pFPGA2SW \rceil$ // Número de transacções
$Twrite = TwriteHW2HW + 2 * TcomparacaoSW + (nt + 2) * TreadHW2SW$
$Twrite = Twrite + (nt + 1) * TwriteSW2HW + nt * TshiftSW$
$Tcom = Twrite * nWR_v * pWR_v$
devolver $(Tcom)$

Figura G.8: Tempo de comunicação necessário para que um estado programa de *hardware* escreva uma variável localizada em *hardware* não adjacente (sem usar interrupções).

```
// Hardware escreve em hardware não adjacente, com interrupções (equação B46)

TcomHWwrHWnaIrq (nBits_v, nWR_v, pWR_v, nAcessosIRQ) ≡
  se (nAcessosIRQ = 1) então
    f1 = 1
    f2 = 0
  senão se (nAcessosIRQ > 1) então
    f1 = 3
    f2 = 1
  fse
  nt = ⌈nBits_v/pFPGA2SW⌉ // Número de transacções
  Twrite = f1 * TwriteHW2HW + TgestaoRSI + (nt + 2) * TwriteSW2HW
  Twrite = Twrite + (f2 + nt) * TreadHW2SW + nt * TshiftSW
  Tcom = Twrite * nWR_v * pWR_v
  devolver (Tcom)
```

Figura G.9: Tempo de comunicação necessário para que um estado programa de *hardware* escreva uma variável localizada em *hardware* não adjacente (usando interrupções).

```
// Hardware escreve em hardware não adjacente, com interrupções (equação B48)

TexecORSIHWwrHWnaIrq (nBits_v, nWR_v, pWR_v, nAcessosIRQ, FN_o) ≡
  se (nAcessosIRQ = 1) então
    f2 = 0
  senão se (nAcessosIRQ > 1) então
    f2 = 1
  fse
  nt = ⌈nBits_v/pFPGA2SW⌉ // Número de transacções
  Twrite = TgestaoRSI + (2 + nt) * TwriteSW2HW + (f2 + nt) * TreadHW2SW + nt * TshiftSW
  Tcom = Twrite * nWR_v * pWR_v * FN_o
  devolver (Tcom)
```

Figura G.10: Tempo de execução da rotina de serviço à interrupção, necessário para que um estado programa de *hardware* escreva uma variável localizada em *hardware* não adjacente.

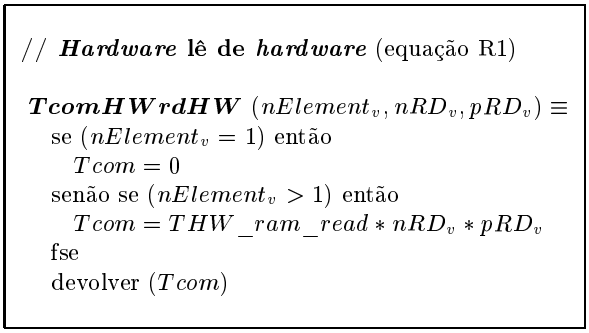
```
// Hardware escreve em hardware adjacente (equação B34)

TcomHWwrHWa (nBits_v, nWR_v, pWR_v, nPinos_i,j) ≡
  nt = ⌈nBits_v/nPinos_i,j⌉ // Número de transacções
  Twrite = nt * TwriteHW2HWa
  Tcom = Twrite * nWR_v * pWR_v
  devolver (Tcom)
```

Figura G.11: Tempo de comunicação necessário para que um estado programa de *hardware* escreva uma variável localizada em *hardware* adjacente.

```
particaoEsquerda (p) ≡
  se (p = 1) então
    pEsq = nFPGAS
  senão
    pEsq = p − 1
  fse
  devolver (pEsq)

particaoDireita (p) ≡
  se (p = nFPGAS) então
    pDir = 1
  senão
    pDir = p + 1
  fse
  devolver (pDir)

particaoNaoAdjacente (p) ≡
  pNA = (p + 2)%nFPGAS
  devolver (pNA)
```

Figura G.12: Cálculo da partição adjacente à esquerda, adjacente à direita e não adjacente duma partição p.

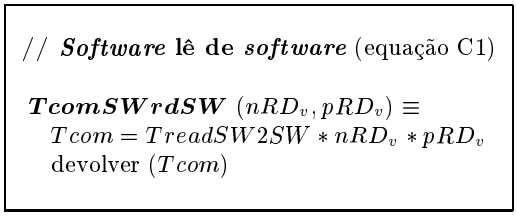
G.4 Equações para Calcular o Tempo de Leitura de Variáveis



// ***Hardware*** **lê de** ***hardware*** (equação R1)

$\boldsymbol{TcomHWrdHW}\ (nElement_v, nRD_v, pRD_v) \equiv$
se $(nElement_v = 1)$ então
$Tcom = 0$
senão se $(nElement_v > 1)$ então
$Tcom = THW\_ram\_read * nRD_v * pRD_v$
fse
devolver $(Tcom)$

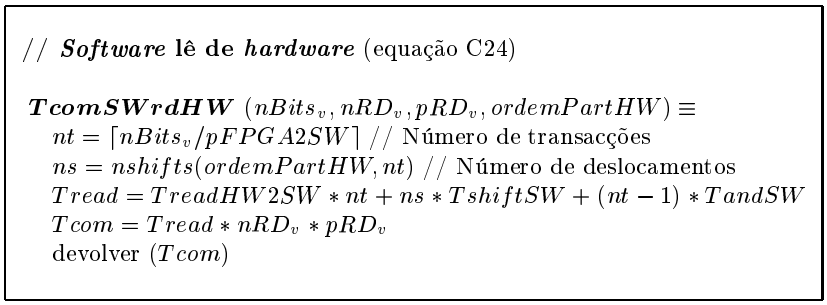
Figura G.13: Tempo de comunicação necessário para que um estado programa de *hardware* leia uma variável de *hardware*.



// ***Software*** **lê de** ***software*** (equação C1)

$\boldsymbol{TcomSWrdSW}\ (nRD_v, pRD_v) \equiv$
$Tcom = TreadSW2SW * nRD_v * pRD_v$
devolver $(Tcom)$

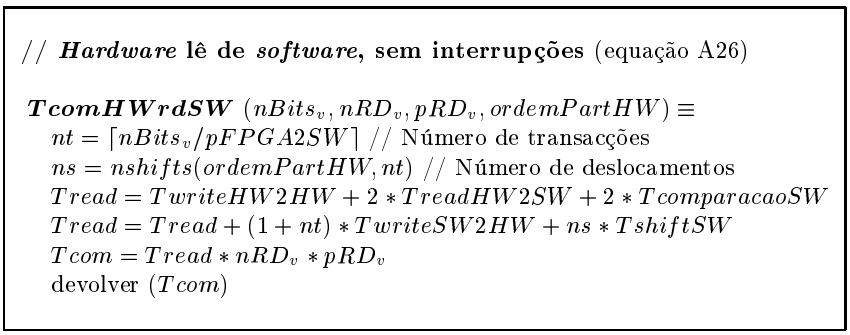
Figura G.14: Tempo de comunicação necessário para que um estado programa de *software* leia uma variável de *software*.



// ***Software*** **lê de** ***hardware*** (equação C24)

$\boldsymbol{TcomSWrdHW}\ (nBits_v, nRD_v, pRD_v, ordemPartHW) \equiv$
$nt = \lceil nBits_v / pFPGA2SW \rceil$ // Número de transacções
$ns = nshifts(ordemPartHW, nt)$ // Número de deslocamentos
$Tread = TreadHW2SW * nt + ns * TshiftSW + (nt - 1) * TandSW$
$Tcom = Tread * nRD_v * pRD_v$
devolver $(Tcom)$

Figura G.15: Tempo de comunicação necessário para que um estado programa de *software* leia uma variável de *hardware*.



// ***Hardware*** **lê de** ***software*****, sem interrupções** (equação A26)

$\boldsymbol{TcomHWrdSW}\ (nBits_v, nRD_v, pRD_v, ordemPartHW) \equiv$
$nt = \lceil nBits_v / pFPGA2SW \rceil$ // Número de transacções
$ns = nshifts(ordemPartHW, nt)$ // Número de deslocamentos
$Tread = TwriteHW2HW + 2 * TreadHW2SW + 2 * TcomparacaoSW$
$Tread = Tread + (1 + nt) * TwriteSW2HW + ns * TshiftSW$
$Tcom = Tread * nRD_v * pRD_v$
devolver $(Tcom)$

Figura G.16: Tempo de comunicação necessário para que um estado programa de *hardware* leia uma variável de *software* (sem usar interrupções).

```
// Hardware lê de software, com interrupções (equação A27)

TcomHWrdSWirq (nBits_v, nRD_v, pRD_v, ordemPartHW, nAcessosIRQ) ≡
  se (nAcessosIRQ = 1) então
    f1 = 1
    f2 = 0
  senão se (nAcessosIRQ > 1) então
    f1 = 3
    f2 = 1
  fse
  nt = ⌈nBits_v/pFPGA2SW⌉ // Número de transacções
  ns = nshifts(ordemPartHW, nt) // Número de deslocamentos
  Tread = f1 * TwriteHW2HW + TgestaoRSI + f2 * TreadHW2SW + TcomparacaoSW
  Tread = Tread + (2 + nt) * TwriteSW2HW + ns * TshiftSW
  Tcom = Tread * nRD_v * pRD_v
  devolver (Tcom)
```

Figura G.17: Tempo de comunicação necessário para que um estado programa de *hardware* leia uma variável de *software* (usando interrupções).

```
// Hardware lê de software, com interrupções (equação A28)

TexecORSIHWrdSWirq (nBits_v, nRD_v, pRD_v, ordemPartHW, nAcessosIRQ, FN_o) ≡
  se (nAcessosIRQ = 1) então
    f2 = 0
  senão se (nAcessosIRQ > 1) então
    f2 = 1
  fse
  nt = ⌈nBits_v/pFPGA2SW⌉ // Número de transacções
  ns = nshifts(ordemPartHW, nt) // Número de deslocamentos
  Tread = TgestaoRSI + f2 * TreadHW2SW + TcomparacaoSW + (2 + nt) * TwriteSW2HW
  Tread = Tread + ns * TshiftSW
  Tcom = Tread * nRD_v * pRD_v * FN_o
  devolver (Tcom)
```

Figura G.18: Tempo de execução da rotina de serviço à interrupção, necessário para que um estado programa de *hardware* leia uma variável de *software*.

```
// Hardware lê de hardware não adjacente, sem interrupções (equação A4R)

TcomHWrdHWna (nBits_v, nRD_v, pRD_v) ≡
  nt = ⌈nBits_v/pFPGA2SW⌉ // Número de transacções
  Tread = TwriteHW2HW + 2 * TcomparacaoSW + (nt + 2) * TreadHW2SW
  Tread = Tread + (nt + 1) * TwriteSW2HW + nt * TshiftSW
  Tcom = Tread * nRD_v * pRD_v
  devolver (Tcom)
```

Figura G.19: Tempo de comunicação necessário para que um estado programa de *hardware* leia uma variável localizada em *hardware* não adjacente (sem usar interrupções).

// ***Hardware* lê de *hardware* não adjacente, com interrupções** (equação A45)

TcomHWrdHWnaIrq $(nBits_v, nRD_v, pRD_v, nAcessosIRQ) \equiv$
  se $(nAcessosIRQ = 1)$ então
    $f1 = 1$
    $f2 = 0$
  senão se $(nAcessosIRQ > 1)$ então
    $f1 = 3$
    $f2 = 1$
  fse
  $nt = \lceil nBits_v / pFPGA2SW \rceil$ // Número de transacções
  $Tread = f1 * TwriteHW2HW + TgestaoRSI + (nt + 2) * TwriteSW2HW$
  $Tread = Tread + (f2 + nt) * TreadHW2SW + nt * TshiftSW$
  $Tcom = Tread * nRD_v * pRD_v$
  devolver $(Tcom)$

Figura G.20: Tempo de comunicação necessário para que um estado programa de *hardware* leia uma variável localizada em *hardware* não adjacente (usando interrupções).

// ***Hardware* lê de *hardware* não adjacente, com interrupções** (equação A47)

TexecORSIHWrdHWnaIrq $(nBits_v, nRD_v, pRD_v, nAcessosIRQ, FN_o) \equiv$
  se $(nAcessosIRQ = 1)$ então
    $f2 = 0$
  senão se $(nAcessosIRQ > 1)$ então
    $f2 = 1$
  fse
  $nt = \lceil nBits_v / pFPGA2SW \rceil$ // Número de transacções
  $Tread = TgestaoRSI + (2 + nt) * TwriteSW2HW + (f2 + nt) * TreadHW2SW + nt * TshiftSW$
  $Tcom = Tread * nRD_v * pRD_v * FN_o$
  devolver $(Tcom)$

Figura G.21: Tempo de execução da rotina de serviço à interrupção, necessário para que um estado programa de *hardware* leia uma variável localizada em *hardware* não adjacente.

// ***Hardware* lê de *hardware* adjacente** (equação A34)

TcomHWrdHWa $(nBits_v, nRD_v, pRD_v, nPinos_{i,j}) \equiv$
  $nt = \lceil nBits_v / nPinos_{i,j} \rceil$ // Número de transacções
  $Tread = nt * TreadHWa2HW$
  $Tcom = Tread * nRD_v * pRD_v$
  devolver $(Tcom)$

Figura G.22: Tempo de comunicação necessário para que um estado programa de *hardware* leia uma variável localizada em *hardware* adjacente.

G.5 Funções para Actualizar os Tempos de Comunicação

ctlActualizarTcomDVarMudar ($changedOBJ, objUpdated, nAcessosIRQ, objParalelo[]$) ≡

// $changedOBJ$ - objectos que mudaram de partição
// $objUpdated = true$ - indica que a actualização devido à mudança
// de estados programa já foi efectuada

para (cada objecto $v \in changedOBJ$) fazer
se ($tipoObj(v) = variavel$) então
// Parâmetros
$pold_v = changedOBJ.oldPartition(v)$
$pnew_v = changedOBJ.newPartition(v)$
$nBits_v = larguraVar(v)$
$nElement_v = numElementosVar(v)$
para (cada objecto $o_r \in OBJread(v)$) fazer
// Parâmetros
se ($o_r \subset changedOBJ$) então
se ($objUpdated = true$) então
$p_o = changedOBJ.newPartition(o_r)$
senão
$p_o = changedOBJ.oldPartition(o_r)$
fse
senão
$p_o = atribuicaoObj(o_r)$
fse
$o_{paralelo} = objParalelo[identificacao(o_r)]$
$nWR_v = writeVAR(o_r).numEscritas(v)$
$pWR_v = writeVAR(o_r).probEscrita(v)$
$nRD_v = readVAR(o_r).numLeituras(v)$
$pRD_v = readVAR(o_r).probLeitura(v)$
$actualizarTcomDevidoVarMudar(v, pold_v, pnew_v, o_r, p_o, RD, o_{paralelo},$
$0, 0, nRD_v, pRD_v, nBits_v, nElement_v, FN(o_r), nAcessosIRQ)$
fpara
para (cada objecto $o_w \in OBJwrite(v)$) fazer
// Parâmetros
se ($o_w \subset changedOBJ$) então
se ($objUpdated = true$) então
$p_o = changedOBJ.newPartition(o_w)$
senão
$p_o = changedOBJ.oldPartition(o_w)$
fse
senão
$p_o = atribuicaoObj(o_w)$
fse
$o_{paralelo} = objParalelo[identificacao(o_w)]$
$nWR_v = writeVAR(o_w).numEscritas(v)$
$pWR_v = writeVAR(o_w).probEscrita(v)$
$nRD_v = readVAR(o_w).numLeituras(v)$
$pRD_v = readVAR(o_w).probLeitura(v)$
$actualizarTcomDevidoVarMudar(v, pold_v, pnew_v, o_w, p_o, WR, o_{paralelo},$
$nWR_v, pWR_v, 0, 0, nBits_v, nElement_v, FN(o_w), nAcessosIRQ)$
fpara
fse
fpara

Figura G.23: Algoritmo que controla a actualização do tempo de comunicação dos estados programa, quando algumas das variáveis por eles acedidas (lidas ou escritas) mudam de partição.

ctlActualizarTcomDObjMudar $(changedOBJ, varUpdated, nAcessosIRQ, objParalelo[]) \equiv$

// $changedOBJ$ - objectos que mudaram de partição
// $varUpdated = true$ - indica que a actualização devido à mudança
// de variáveis já foi efectuada

para (cada objecto $o \in changedOBJ$) fazer
se ($tipoObj(o) \neq variavel$) então
$TcomViaSW(o) = TcomDir(o) = TcomEsq(o) = TcomLocal(o) = 0$
// Parâmetros
$pold_o = changedOBJ.oldPartition(o)$
$pnew_o = changedOBJ.newPartition(o)$
$o_{paralelo} = objParalelo[identificacao(o)]$
para (cada objecto $v_r \in readVAR(o)$) fazer
// Parâmetros
$nBits_v = larguraVar(v_r)$
$nElement_v = numElementosVar(v_r)$
se ($v_r \subset changedOBJ$) então
se ($varUpdated = true$) então
$p_v = changedOBJ.newPartition(v_r)$
senão
$p_v = changedOBJ.oldPartition(v_r)$
fse
senão
$p_v = atribuicaoObj(v_r)$
fse
$nWR_v = writeVAR(o).numEscritas(v_r)$
$pWR_v = writeVAR(o).probEscrita(v_r)$
$nRD_v = readVAR(o).numLeituras(v_r)$
$pRD_v = readVAR(o).probLeitura(v_r)$
$actualizarTcomDevidoObjMudar(o, pold_o, pnew_o, v_r, p_v, RD, o_{paralelo},$
$nWR_v, pWR_v, nRD_v, pRD_v, nBits_v, nElement_v, FN(o), nAcessosIRQ)$
fpara
para (cada objecto $v_w \in writeVAR(o)$) fazer
// Parâmetros
$nBits_v = larguraVar(v_w)$
$nElement_v = numElementosVar(v_w)$
se ($v_w \subset changedOBJ$) então
se ($varUpdated = true$) então
$p_v = changedOBJ.newPartition(v_w)$
senão
$p_v = changedOBJ.oldPartition(v_w)$
fse
senão
$p_v = atribuicaoObj(v_w)$
fse
$nWR_v = writeVAR(o).numEscritas(v_w)$
$pWR_v = writeVAR(o).probEscrita(v_w)$
$nRD_v = readVAR(o).numLeituras(v_w)$
$pRD_v = readVAR(o).probLeitura(v_w)$
$actualizarTcomDevidoObjMudar(o, pold_o, pnew_o, v_w, p_v, WR, o_{paralelo},$
$nWR_v, pWR_v, nRD_v, pRD_v, nBits_v, nElement_v, FN(o), nAcessosIRQ)$
fpara
fse
fpara

Figura G.24: Algoritmo que controla a actualização do tempo de comunicação dos estados programa que mudam de partição.

actualizarTcomDevidoVarMudar $(v, pold_v, pnew_v, o, p_o, RDouWR, o_{paralelo}, nWR_v, pWR_v, nRD_v, pRD_v, nBits_v, nElement_v, FN_o, nAcessosIRQ) \equiv$

// Simplificações: nPinos=16

se $(RDouWR = WR)$ então // ESCRITA

se $(p_o = SW)$ então // PARTIÇÃO DE *SOFTWARE* (tempo comunicação guardado em $TcomLocal_o$)
caso $pnew_v$ seja:
SW => // (D1). *software* escreve no *software*
$Tcom_v = TcomSWwrSW(nWR_v, pWR_v)$ // [EQ. D1]
$TcomOld_v = TcomSWwrHW(nBits_v, nWR_v, pWR_v, pold_v)$ // [EQ. D24]
$TcomLocal_o = TcomLocal_o - TcomOld_v + Tcom_v$

HW1 | HW2 | HW3 | HW4 => // (D2). *software* escreve no *hardware*
$Tcom_v = TcomSWwrHW(nBits_v, nWR_v, pWR_v, pnew_v)$ // [EQ. D24]
$TcomOld_v = TcomSWwrSW(nWR_v, pWR_v)$ // [EQ. D1]
$TcomLocal_o = TcomLocal_o - TcomOld_v + Tcom_v$
fcaso

senão se $(p_o = HW1|HW2|HW3|HW4)$ então // PARTIÇÃO DE *HARDWARE*
// O tempo de comunicação é $MAX[TcomLocal_o, TcomViaSW_o, TcomDir_o, TcomEsq_o]$
// O tempo de comunicação não é calculado nesta função

caso $pnew_v$ seja:
p_o => // (B1). *hardware* escreve dentro da mesma partição de *hardware*

$Tcom_v = TcomHWwrHW(nElement_v, nWR_v, pWR_v)$ // [EQ. W1]
$TcomLocal_o = TcomLocal_o + Tcom_v$

// (B3e). *hardware* escreve *hardware* adjacente à esquerda
se $(pold_v = particaoEsquerda(p_o))$ então
$TcomOld_v = TcomHWwrHWa(nBits_v, nWR_v, pWR_v, nPinos_{p_o, pold_v})$ // [EQ. B34]
$TcomEsq_o = TcomEsq_o - TcomOld_v$

// (B3d). *hardware* escreve *hardware* adjacente à direita
senão se $(pold_v = particaoDireita(p_o))$ então
$TcomOld_v = TcomHWwrHWa(nBits_v, nWR_v, pWR_v, nPinos_{p_o, pold_v})$ // [EQ. B34]
$TcomDir_o = TcomDir_o - TcomOld_v$

// (B4). *hardware* escreve *hardware* não adjacente
senão se $(pold_v = particaoNaoAdjacente(p_o))$ então
se $(o_{paralelo} = NULL)$ então // Não existe paralelismo entre o e *software*
$TcomOld_v = TcomHWwrHWna(nBits_v, nWR_v, pWR_v)$ // [EQ. B42]
$TcomViaSW_o = TcomViaSW_o - TcomOld_v$
senão // Existe paralelismo entre o e $o_{paralelo}$ (do fluxo de *software*)
$TcomOld_v = TcomHWwrHWnaIrq(nBits_v, nWR_v, pWR_v, nAcessosIRQ)$ // [EQ. B46]
$TcomViaSW_o = TcomViaSW_o - TcomOld_v$
$TcomOld_v = TexecORSIHWwrHWnaIrq(nBits_v, nWR_v, pWR_v, nAcessosIRQ, FN_o)$ // [EQ. B48]
$TexecRSI(o_{paralelo}) = TexecRSI(o_{paralelo}) - TcomOld_v$
$nAcessosIRQ = nAcessosIRQ - 1$
fse

Figura G.25: Algoritmo que actualiza o tempo de comunicação dum estado programa, quando uma variável por ele acedida (lida ou escrita) muda de partição *(parte 1)*.

// (B2). *hardware* escreve no *software*
senão se ($pold_v = SW$) então
  se ($o_{paralelo} = NULL$) então // Não existe paralelismo entre o e *software*
    $TcomOld_v = TcomHWwrSW(nBits_v, nWR_v, pWR_v, p_o)$ // [EQ. B27]
    $TcomViaSW_o = TcomViaSW_o - TcomOld_v$
  senão // Existe paralelismo entre o e $o_{paralelo}$ (do fluxo de *software*)
    $TcomOld_v = TcomHWwrSWirq(nBits_v, nWR_v, pWR_v, p_o, nAcessosIRQ)$ // [EQ. B25]
    $TcomViaSW_o = TcomViaSW_o - TcomOld_v$
    $TcomOld_v = TexecORSIHWwrSWirq(nBits_v, nWR_v, pWR_v, p_o,$
                 $nAcessosIRQ, FN_o)$ // [EQ. B26]
    $TexecRSI(o_{paralelo}) = TexecRSI(o_{paralelo}) - TcomOld_v$
    $nAcessosIRQ = nAcessosIRQ - 1$
  fse
fse // Fim de (B1). *hardware* escreve dentro da mesma partição de *hardware*

SW => // (B2). *hardware* escreve no *software*

se ($o_{paralelo} = NULL$) então // Não existe paralelismo entre o e *software*
  $Tcom_v = TcomHWwrSW(nBits_v, nWR_v, pWR_v, p_o)$ // [EQ. B27]
  $TcomViaSW_o = TcomViaSW_o + Tcom_v$
senão // Existe paralelismo entre o e $o_{paralelo}$ (do fluxo de *software*)
  $Tcom_v = TcomHWwrSWirq(nBits_v, nWR_v, pWR_v, p_o, nAcessosIRQ)$ // [EQ. B25]
  $TcomViaSW_o = TcomViaSW_o + Tcom_v$
  $Tcom_v = TexecORSIHWwrSWirq(nBits_v, nWR_v, pWR_v, p_o, nAcessosIRQ, FN_o)$ // [EQ. B26]
  $TexecRSI(o_{paralelo}) = TexecRSI(o_{paralelo}) + Tcom_v$
  $nAcessosIRQ = nAcessosIRQ + 1$
fse

// (B3e). *hardware* escreve *hardware* adjacente à esquerda
se ($pold_v = particaoEsquerda(p_o)$) então
  $TcomOld_v = TcomHWwrHWa(nBits_v, nWR_v, pWR_v, nPinos_{p_o, pold_v})$ // [EQ. B34]
  $TcomEsq_o = TcomEsq_o - TcomOld_v$

// (B3d). *hardware* escreve *hardware* adjacente à direita
senão se ($pold_v = particaoDireita(p_o)$) então
  $TcomOld_v = TcomHWwrHWa(nBits_v, nWR_v, pWR_v, nPinos_{p_o, pold_v})$ // [EQ. B34]
  $TcomDir_o = TcomDir_o - TcomOld_v$

// (B4). *hardware* escreve *hardware* não adjacente
senão se ($pold_v = particaoNaoAdjacente(p_o)$) então
  se ($o_{paralelo} = NULL$) então // Não existe paralelismo entre o e *software*
    $TcomOld_v = TcomHWwrHWna(nBits_v, nWR_v, pWR_v)$ // [EQ. B42]
    $TcomViaSW_o = TcomViaSW_o - TcomOld_v$
  senão // Existe paralelismo entre o e $o_{paralelo}$ (do fluxo de *software*)
    $TcomOld_v = TcomHWwrHWnaIrq(nBits_v, nWR_v, pWR_v, nAcessosIRQ)$ // [EQ. B46]
    $TcomViaSW_o = TcomViaSW_o - TcomOld_v$
    $TcomOld_v = TexecORSIHWwrHWnaIrq(nBits_v, nWR_v, pWR_v,$
                 $nAcessosIRQ, FN_o)$ // [EQ. B48]
    $TexecRSI(o_{paralelo}) = TexecRSI(o_{paralelo}) - TcomOld_v$
    $nAcessosIRQ = nAcessosIRQ - 1$
  fse

// (B1). *hardware* escreve dentro da mesma partição de *hardware*
senão se ($pold_v = p_o$)) então
  $TcomOld_v = TcomHWwrHW(nElement_v, nWR_v, pWR_v)$ // [EQ. W1]
  $TcomLocal_o = TcomLocal_o - TcomOld_v$
fse // Fim de (B2). *hardware* escreve no *software*

Figura G.26: Algoritmo que actualiza o tempo de comunicação dum estado programa, quando uma variável por ele acedida (lida ou escrita) muda de partição *(parte 2)*.

$particaoNaoAdjacente(p_o)$ => // (B4). *hardware* escreve *hardware* não adjacente

se ($o_{paralelo} = NULL$) então // Não existe paralelismo entre o e *software*
$Tcom_v = TcomHWwrHWna(nBits_v, nWR_v, pWR_v)$ // [EQ. B42]
$TcomViaSW_o = TcomViaSW_o + Tcom_v$
senão // Existe paralelismo entre o e $o_{paralelo}$ (do fluxo de *software*)
$Tcom_v = TcomHWwrHWnaIrq(nBits_v, nWR_v, pWR_v, nAcessosIRQ)$ // [EQ. B46]
$TcomViaSW_o = TcomViaSW_o + Tcom_v$
$Tcom_v = TexecORSIHWwrHWnaIrq(nBits_v, nWR_v, pWR_v,$
$nAcessosIRQ, FN_o)$ // [EQ. B48]
$TexecRSI(o_{paralelo}) = TexecRSI(o_{paralelo}) + Tcom_v$
$nAcessosIRQ = nAcessosIRQ + 1$
fse

// (B3e). *hardware* escreve *hardware* adjacente à esquerda
se ($pold_v = particaoEsquerda(p_o)$) então
$TcomOld_v = TcomHWwrHWa(nBits_v, nWR_v, pWR_v, nPinos_{p_o,pold_v})$ // [EQ. B34]
$TcomEsq_o = TcomEsq_o - TcomOld_v$

// (B3d). *hardware* escreve *hardware* adjacente à direita
senão se ($pold_v = particaoDireita(p_o)$) então
$TcomOld_v = TcomHWwrHWa(nBits_v, nWR_v, pWR_v, nPinos_{p_o,pold_v})$ // [EQ. B34]
$TcomDir_o = TcomDir_o - TcomOld_v$

// (B2). *hardware* escreve no *software*
senão se ($pold_v = SW$) então
se ($o_{paralelo} = NULL$) então // Não existe paralelismo entre o e *software*
$TcomOld_v = TcomHWwrSW(nBits_v, nWR_v, pWR_v, p_o)$ // [EQ. B27]
$TcomViaSW_o = TcomViaSW_o - TcomOld_v$
senão // Existe paralelismo entre o e $o_{paralelo}$ (do fluxo de *software*)
$TcomOld_v = TcomHWwrSWirq(nBits_v, nWR_v, pWR_v, p_o, nAcessosIRQ)$ // [EQ. B25]
$TcomViaSW_o = TcomViaSW_o - TcomOld_v$
$TcomOld_v = TexecORSIHWwrSWirq(nBits_v, nWR_v, pWR_v, p_o,$
$nAcessosIRQ, FN_o)$ // [EQ. B26]
$TexecRSI(o_{paralelo}) = TexecRSI(o_{paralelo}) - TcomOld_v$
$nAcessosIRQ = nAcessosIRQ - 1$
fse

// (B1). *hardware* escreve dentro da mesma partição de *hardware*
senão se ($pold_v = p_o$)) então
$TcomOld_v = TcomHWwrHW(nElement_v, nWR_v, pWR_v)$ // [EQ. W1]
$TcomLocal_o = TcomLocal_o - TcomOld_v$
fse // Fim de (B4). *hardware* escreve *hardware* não adjacente

$particaoDireita(p_o)$ => // (B3d). *hardware* escreve *hardware* adjacente à direita

$Tcom_v = TcomHWwrHWa(nBits_v, nWR_v, pWR_v, nPinos_{p_o,pold_v})$ // [EQ. B34]
$TcomDir_o = TcomDir_o + Tcom_v$

// (B3e). *hardware* escreve *hardware* adjacente à esquerda
se ($pold_v = particaoEsquerda(p_o)$) então
$TcomOld_v = TcomHWwrHWa(nBits_v, nWR_v, pWR_v, nPinos_{p_o,pold_v})$ // [EQ. B34]
$TcomEsq_o = TcomEsq_o - TcomOld_v$

Figura G.27: Algoritmo que actualiza o tempo de comunicação dum estado programa, quando uma variável por ele acedida (lida ou escrita) muda de partição *(parte 3)*.

```
// (B4). hardware escreve em hardware não adjacente
senão se (pold_v = particaoNaoAdjacente(p_o)) então
  se (o_paralelo = NULL) então // Não existe paralelismo entre o e software
    TcomOld_v = TcomHWwrHWna(nBits_v, nWR_v, pWR_v) // [EQ. B42]
    TcomViaSW_o = TcomViaSW_o − TcomOld_v
  senão // Existe paralelismo entre o e o_paralelo (do fluxo de software)
    TcomOld_v = TcomHWwrHWnaIrq(nBits_v, nWR_v, pWR_v, nAcessosIRQ) // [EQ. B46]
    TcomViaSW_o = TcomViaSW_o − TcomOld_v
    TcomOld_v = TexecORSIHWwrHWnaIrq(nBits_v, nWR_v, pWR_v,
                  nAcessosIRQ, FN_o) // [EQ. B48]
    TexecRSI(o_paralelo) = TexecRSI(o_paralelo) − TcomOld_v
    nAcessosIRQ = nAcessosIRQ − 1
  fse

// (B2). hardware escreve no software
senão se (pold_v = SW) então
  se (o_paralelo = NULL) então // Não existe paralelismo entre o e software
    TcomOld_v = TcomHWwrSW(nBits_v, nWR_v, pWR_v, p_o) // [EQ. B27]
    TcomViaSW_o = TcomViaSW_o − TcomOld_v
  senão // Existe paralelismo entre o e o_paralelo (do fluxo de software)
    TcomOld_v = TcomHWwrSWirq(nBits_v, nWR_v, pWR_v, p_o, nAcessosIRQ) // [EQ. B25]
    TcomViaSW_o = TcomViaSW_o − TcomOld_v
    TcomOld_v = TexecORSIHWwrSWirq(nBits_v, nWR_v, pWR_v, p_o,
                  nAcessosIRQ, FN_o) // [EQ. B26]
    TexecRSI(o_paralelo) = TexecRSI(o_paralelo) − TcomOld_v
    nAcessosIRQ = nAcessosIRQ − 1
  fse

// (B1). hardware escreve dentro da mesma partição de hardware
senão se (pold_v = p_o)) então
  TcomOld_v = TcomHWwrHW(nElement_v, nWR_v, pWR_v) // [EQ. W1]
  TcomLocal_o = TcomLocal_o − TcomOld_v
fse // Fim de (B3d). hardware escreve hardware adjacente à direita

particaoEsquerda(p_o) => // (B3e). hardware escreve hardware adjacente à esquerda

Tcom_v = TcomHWwrHWa(nBits_v, nWR_v, pWR_v, nPinos_{p_o,pold_v}) // [EQ. B34]
TcomEsq_o = TcomEsq_o + Tcom_v

// (B3d). hardware escreve em hardware adjacente à direita
se (pold_v = particaoEsquerda(p_o)) então
  TcomOld_v = TcomHWwrHWa(nBits_v, nWR_v, pWR_v, nPinos_{p_o,pold_v}) // [EQ. B34]
  TcomDir_o = TcomDir_o − TcomOld_v

// (B4). hardware escreve em hardware não adjacente
senão se (pold_v = particaoNaoAdjacente(p_o)) então
  se (o_paralelo = NULL) então // Não existe paralelismo entre o e software
    TcomOld_v = TcomHWwrHWna(nBits_v, nWR_v, pWR_v) // [EQ. B42]
    TcomViaSW_o = TcomViaSW_o − TcomOld_v
  senão // Existe paralelismo entre o e o_paralelo (do fluxo de software)
    TcomOld_v = TcomHWwrHWnaIrq(nBits_v, nWR_v, pWR_v, nAcessosIRQ) // [EQ. B46]
    TcomViaSW_o = TcomViaSW_o − TcomOld_v
    TcomOld_v = TexecORSIHWwrHWnaIrq(nBits_v, nWR_v, pWR_v,
                  nAcessosIRQ, FN_o) // [EQ. B48]
    TexecRSI(o_paralelo) = TexecRSI(o_paralelo) − TcomOld_v
    nAcessosIRQ = nAcessosIRQ − 1
  fse
```

Figura G.28: Algoritmo que actualiza o tempo de comunicação dum estado programa, quando uma variável por ele acedida (lida ou escrita) muda de partição *(parte 4)*.

```
      // (B2). hardware escreve no software
      senão se (pold_v = SW) então
        se (o_paralelo = NULL) então // Não existe paralelismo entre o e software
          TcomOld_v = TcomHWwrSW(nBits_v, nWR_v, pWR_v, p_o) // [EQ. B27]
          TcomViaSW_o = TcomViaSW_o − TcomOld_v
        senão // Existe paralelismo entre o e o_paralelo (do fluxo de software)
          TcomOld_v = TcomHWwrSWirq(nBits_v, nWR_v, pWR_v, p_o, nAcessosIRQ) // [EQ. B25]
          TcomViaSW_o = TcomViaSW_o − TcomOld_v
          TcomOld_v = TexecORSIHWwrSWirq(nBits_v, nWR_v, pWR_v, p_o,
                        nAcessosIRQ, FN_o) // [EQ. B26]
          TexecRSI(o_paralelo) = TexecRSI(o_paralelo) − TcomOld_v
          nAcessosIRQ = nAcessosIRQ − 1
        fse

      // (B1). hardware escreve dentro da mesma partição de hardware
      senão se (pold_v = p_o)) então
        TcomOld_v = TcomHWwrHW(nElement_v, nWR_v, pWR_v) // [EQ. W1]
        TcomLocal_o = TcomLocal_o − TcomOld_v
      fse // Fim de (B3e). hardware escreve hardware adjacente à esquerda

    fcaso // Fim das alternativas para a nova partição de v

  fse // Fim das alternativas em que a partição do estado programa (o) que escreve v é hardware

senão se (RDouWR = RD) então // LEITURA

  se (p_o = SW) então // PARTIÇÃO DE SOFTWARE (tempo comunicação guardado em TcomLocal_o)
    caso pnew_v seja:
      SW => // (C1). software lê do software
        Tcom_v = TcomSWrdSW(nRD_v, pRD_v) // [EQ. C1]
        TcomOld_v = TcomSWrdHW(nBits_v, nRD_v, pRD_v, pold_v) // [EQ. C24]
        TcomLocal_o = TcomLocal_o − TcomOld_v + Tcom_v

      HW1 | HW2 | HW3 | HW4 => // (C2). software lê do hardware
        Tcom_v = TcomSWrdHW(nBits_v, nRD_v, pRD_v, pnew_v) // [EQ. C24]
        TcomOld_v = TcomSWrdSW(nRD_v, pRD_v) // [EQ. C1]
        TcomLocal_o = TcomLocal_o − TcomOld_v + Tcom_v
    fcaso

  senão se (p_o = HW1|HW2|HW3|HW4) então // PARTIÇÃO DE HARDWARE
    // O tempo comunicação é MAX[TcomLocal_o, TcomViaSW_o, TcomDir_o, TcomEsq_o]
    // O tempo comunicação não é calculado nesta função

    caso pnew_v seja:
      p_o => // (A1). hardware lê da mesma partição de hardware

        Tcom_v = TcomHWrdHW(nElement_v, nRD_v, pRD_v) // [EQ. R1]
        TcomLocal_o = TcomLocal_o + Tcom_v

        // (A3e). hardware lê do hardware adjacente à esquerda
        se (pold_v = particaoEsquerda(p_o)) então
          TcomOld_v = TcomHWrdHWa(nBits_v, nRD_v, pRD_v, nPinos_{p_o,pold_v}) // [EQ. A34]
          TcomEsq_o = TcomEsq_o − TcomOld_v

        // (A3d). hardware lê do hardware adjacente à direita
        senão se (pold_v = particaoDireita(p_o)) então
          TcomOld_v = TcomHWrdHWa(nBits_v, nRD_v, pRD_v, nPinos_{p_o,pold_v}) // [EQ. A34]
          TcomDir_o = TcomDir_o − TcomOld_v
```

Figura G.29: Algoritmo que actualiza o tempo de comunicação dum estado programa, quando uma variável por ele acedida (lida ou escrita) muda de partição *(parte 5)*.

```
// (A4). hardware lê do hardware não adjacente
senão se (pold_v = particaoNaoAdjacente(p_o)) então
  se (o_paralelo = NULL) então // Não existe paralelismo entre o e software
    TcomOld_v = TcomHWrdHWna(nBits_v, nRD_v, pRD_v) // [EQ. A4R]
    TcomViaSW_o = TcomViaSW_o − TcomOld_v
  senão // Existe paralelismo entre o e o_paralelo (do fluxo de software)
    TcomOld_v = TcomHWrdHWnaIrq(nBits_v, nRD_v, pRD_v, nAcessosIRQ) // [EQ. A45]
    TcomViaSW_o = TcomViaSW_o − TcomOld_v
    TcomOld_v = TexecORSIHWrdHWnaIrq(nBits_v, nRD_v, pRD_v,
                  nAcessosIRQ, FN_o) // [EQ. A47]
    TexecRSI(o_paralelo) = TexecRSI(o_paralelo) − TcomOld_v
    nAcessosIRQ = nAcessosIRQ − 1
  fse

// (A2). hardware lê do software
senão se (pold_v = SW) então
  se (o_paralelo = NULL) então // Não existe paralelismo entre o e software
    TcomOld_v = TcomHWrdSW(nBits_v, nRD_v, pRD_v, p_o) // [EQ. A26]
    TcomViaSW_o = TcomViaSW_o − TcomOld_v
  senão // Existe paralelismo entre o e o_paralelo (do fluxo de software)
    TcomOld_v = TcomHWrdSWirq(nBits_v), nRD_v, pRD_v, p_o, nAcessosIRQ) // [EQ. A27]
    TcomViaSW_o = TcomViaSW_o − TcomOld_v
    TcomOld_v = TexecORSIHWrdSWirq(nBits_v, nRD_v, pRD_v, p_o,
                  nAcessosIRQ, FN_o) // [EQ. A28]
    TexecRSI(o_paralelo) = TexecRSI(o_paralelo) − TcomOld_v
    nAcessosIRQ = nAcessosIRQ − 1
  fse
fse // Fim de (A1). hardware lê da mesma partição de hardware

SW => // (A2). hardware lê do software

se (o_paralelo = NULL) então // Não existe paralelismo entre o e software
  Tcom_v = TcomHWrdSW(nBits_v, nRD_v, pRD_v, p_o) // [EQ. A26]
  TcomViaSW_o = TcomViaSW_o + Tcom_v
senão // Existe paralelismo entre o e o_paralelo (do fluxo de software)
  Tcom_v = TcomHWrdSWirq(nBits_v), nRD_v, pRD_v, p_o, nAcessosIRQ) // [EQ. A27]
  TcomViaSW_o = TcomViaSW_o + Tcom_v
  Tcom_v = TexecORSIHWrdSWirq(nBits_v, nRD_v, pRD_v, p_o, nAcessosIRQ, FN_o) // [EQ. A28]
  TexecRSI(o_paralelo) = TexecRSI(o_paralelo) + Tcom_v
  nAcessosIRQ = nAcessosIRQ + 1
fse

// (A3e). hardware lê do hardware adjacente à esquerda
se (pold_v = particaoEsquerda(p_o)) então
  TcomOld_v = TcomHWrdHWa(nBits_v, nRD_v, pRD_v, nPinos_{p_o,pold_v}) // [EQ. A34]
  TcomEsq_o = TcomEsq_o − TcomOld_v

// (A3d). hardware lê do hardware adjacente à direita
senão se (pold_v = particaoDireita(p_o)) então
  TcomOld_v = TcomHWrdHWa(nBits_v, nRD_v, pRD_v, nPinos_{p_o,pold_v}) // [EQ. A34]
  TcomDir_o = TcomDir_o − TcomOld_v

// (A4). hardware lê do hardware não adjacente
senão se (pold_v = particaoNaoAdjacente(p_o)) então
  se (o_paralelo = NULL) então // Não existe paralelismo entre o e software
    TcomOld_v = TcomHWrdHWna(nBits_v, nRD_v, pRD_v) // [EQ. A4R]
    TcomViaSW_o = TcomViaSW_o − TcomOld_v
```

Figura G.30: Algoritmo que actualiza o tempo de comunicação dum estado programa, quando uma variável por ele acedida (lida ou escrita) muda de partição *(parte 6)*.

senão // Existe paralelismo entre o e $o_{paralelo}$ (do fluxo de *software*)
$TcomOld_v = TcomHWrdHWnaIrq(nBits_v, nRD_v, pRD_v, nAcessosIRQ)$ // [EQ. A45]
$TcomViaSW_o = TcomViaSW_o - TcomOld_v$
$TcomOld_v = TexecORSIHWrdHWnaIrq(nBits_v, nRD_v, pRD_v,$
$nAcessosIRQ, FN_o)$ // [EQ. A47]
$TexecRSI(o_{paralelo}) = TexecRSI(o_{paralelo}) - TcomOld_v$
$nAcessosIRQ = nAcessosIRQ - 1$
fse

// (A1). *hardware* lê da mesma partição de *hardware*
senão se ($pold_v = p_o$)) então
$TcomOld_v = TcomHWrdHW(nElement_v, nRD_v, pRD_v)$ // [EQ. R1]
$TcomLocal_o = TcomLocal_o - TcomOld_v$
fse // Fim de (A2). *hardware* lê do *software*

$particaoNaoAdjacente(p_o)$ => // (A4). *hardware* lê do *hardware* não adjacente

se ($o_{paralelo} = NULL$) então // Não existe paralelismo entre o e *software*
$Tcom_v = TcomHWrdHWna(nBits_v, nRD_v, pRD_v)$ // [EQ. A4R]
$TcomViaSW_o = TcomViaSW_o + Tcom_v$
senão // Existe paralelismo entre o e $o_{paralelo}$ (do fluxo de *software*)
$Tcom_v = TcomHWrdHWnaIrq(nBits_v, nRD_v, pRD_v, nAcessosIRQ)$ // [EQ. A45]
$TcomViaSW_o = TcomViaSW_o + Tcom_v$
$Tcom_v = TexecORSIHWrdHWnaIrq(nBits_v, nRD_v, pRD_v, nAcessosIRQ, FN_o)$ // [EQ. A47]
$TexecRSI(o_{paralelo}) = TexecRSI(o_{paralelo}) + Tcom_v$
$nAcessosIRQ = nAcessosIRQ + 1$
fse

// (A3e). *hardware* lê do *hardware* adjacente à esquerda
se ($pold_v = particaoEsquerda(p_o)$) então
$TcomOld_v = TcomHWrdHWa(nBits_v, nRD_v, pRD_v, nPinos_{p_o,pold_v})$ // [EQ. A34]
$TcomEsq_o = TcomEsq_o - TcomOld_v$

// (A3d). *hardware* lê do *hardware* adjacente à direita
senão se ($pold_v = particaoDireita(p_o)$) então
$TcomOld_v = TcomHWrdHWa(nBits_v, nRD_v, pRD_v, nPinos_{p_o,pold_v})$ // [EQ. A34]
$TcomDir_o = TcomDir_o - TcomOld_v$

// (A2). *hardware* lê do *software*
senão se ($pold_v = SW$) então
se ($o_{paralelo} = NULL$) então // Não existe paralelismo entre o e *software*
$TcomOld_v = TcomHWrdSW(nBits_v, nRD_v, pRD_v, p_o)$ // [EQ. A26]
$TcomViaSW_o = TcomViaSW_o - TcomOld_v$
senão // Existe paralelismo entre o e $o_{paralelo}$ (do fluxo de *software*)
$TcomOld_v = TcomHWrdSWirq(nBits_v), nRD_v, pRD_v, p_o, nAcessosIRQ)$ // [EQ. A27]
$TcomViaSW_o = TcomViaSW_o - TcomOld_v$
$TcomOld_v = TexecORSIHWrdSWirq(nBits_v, nRD_v, pRD_v, p_o,$
$nAcessosIRQ, FN_o)$ // [EQ. A28]
$TexecRSI(o_{paralelo}) = TexecRSI(o_{paralelo}) - TcomOld_v$
$nAcessosIRQ = nAcessosIRQ - 1$
fse

// (A1). *hardware* lê da mesma partição de *hardware*
senão se ($pold_v = p_o$)) então
$TcomOld_v = TcomHWrdHW(nElement_v, nRD_v, pRD_v)$ // [EQ. R1]
$TcomLocal_o = TcomLocal_o - TcomOld_v$
fse // Fim de (A4). *hardware* lê do *hardware* não adjacente

Figura G.31: Algoritmo que actualiza o tempo de comunicação dum estado programa, quando uma variável por ele acedida (lida ou escrita) muda de partição *(parte 7)*.

$particaoDireita(p_o)$ => // (A3d). *hardware* lê do *hardware* adjacente à direita

$Tcom_v = TcomHWrdHWa(nBits_v, nRD_v, pRD_v, nPinos_{p_o, pold_v})$ // [EQ. A34]
$TcomDir_o = TcomDir_o + Tcom_v$

// (A3e). *hardware* lê do *hardware* adjacente à esquerda
se $(pold_v = particaoEsquerda(p_o))$ então
  $TcomOld_v = TcomHWrdHWa(nBits_v, nRD_v, pRD_v, nPinos_{p_o, pold_v})$ // [EQ. A34]
  $TcomEsq_o = TcomEsq_o - TcomOld_v$

// (A4). *hardware* lê do *hardware* não adjacente
senão se $(pold_v = particaoNaoAdjacente(p_o))$ então
  se $(o_{paralelo} = NULL)$ então // Não existe paralelismo entre o e *software*
    $TcomOld_v = TcomHWrdHWna(nBits_v, nRD_v, pRD_v)$ // [EQ. A4R]
    $TcomViaSW_o = TcomViaSW_o - TcomOld_v$
  senão // Existe paralelismo entre o e $o_{paralelo}$ (do fluxo de *software*)
    $TcomOld_v = TcomHWrdHWnaIrq(nBits_v, nRD_v, pRD_v, nAcessosIRQ)$ // [EQ. A45]
    $TcomViaSW_o = TcomViaSW_o - TcomOld_v$
    $TcomOld_v = TexecORSIHWrdHWnaIrq(nBits_v, nRD_v, pRD_v,$
                 $nAcessosIRQ, FN_o)$ // [EQ. A47]
    $TexecRSI(o_{paralelo}) = TexecRSI(o_{paralelo}) - TcomOld_v$
    $nAcessosIRQ = nAcessosIRQ - 1$
  fse

// (A2). *hardware* lê do *software*
senão se $(pold_v = SW)$ então
  se $(o_{paralelo} = NULL)$ então // Não existe paralelismo entre o e *software*
    $TcomOld_v = TcomHWrdSW(nBits_v, nRD_v, pRD_v, p_o)$ // [EQ. A26]
    $TcomViaSW_o = TcomViaSW_o - TcomOld_v$
  senão // Existe paralelismo entre o e $o_{paralelo}$ (do fluxo de *software*)
    $TcomOld_v = TcomHWrdSWirq(nBits_v), nRD_v, pRD_v, p_o, nAcessosIRQ)$ // [EQ. A27]
    $TcomViaSW_o = TcomViaSW_o - TcomOld_v$
    $TcomOld_v = TexecORSIHWrdSWirq(nBits_v, nRD_v, pRD_v, p_o,$
                 $nAcessosIRQ, FN_o)$ // [EQ. A28]
    $TexecRSI(o_{paralelo}) = TexecRSI(o_{paralelo}) - TcomOld_v$
    $nAcessosIRQ = nAcessosIRQ - 1$
  fse

// (A1). *hardware* lê da mesma partição de *hardware*
senão se $(pold_v = p_o))$ então
  $TcomOld_v = TcomHWrdHW(nElement_v, nRD_v, pRD_v)$ // [EQ. R1]
  $TcomLocal_o = TcomLocal_o - TcomOld_v$
fse // Fim de (A3d). *hardware* lê do *hardware* adjacente à direita

$particaoEsquerda(p_o)$ => // (A3e). *hardware* lê do *hardware* adjacente à esquerda

$Tcom_v = TcomHWrdHWa(nBits_v, nRD_v, pRD_v, nPinos_{p_o, pold_v})$ // [EQ. A34]
$TcomEsq_o = TcomEsq_o + Tcom_v$

// (A3d). *hardware* lê do *hardware* adjacente à direita
se $(pold_v = particaoEsquerda(p_o))$ então
  $TcomOld_v = TcomHWrdHWa(nBits_v, nRD_v, pRD_v, nPinos_{p_o, pold_v})$ // [EQ. A34]
  $TcomDir_o = TcomDir_o - TcomOld_v$

Figura G.32: Algoritmo que actualiza o tempo de comunicação dum estado programa, quando uma variável por ele acedida (lida ou escrita) muda de partição *(parte 8)*.

```
    // (A4). hardware lê do hardware não adjacente
    senão se (pold_v = particaoNaoAdjacente(p_o)) então
      se (o_paralelo = NULL) então // Não existe paralelismo entre o e software
        TcomOld_v = TcomHWrdHWna(nBits_v, nRD_v, pRD_v) // [EQ. A4R]
        TcomViaSW_o = TcomViaSW_o − TcomOld_v
      senão // Existe paralelismo entre o e o_paralelo (do fluxo de software)
        TcomOld_v = TcomHWrdHWnaIrq(nBits_v, nRD_v, pRD_v, nAcessosIRQ) // [EQ. A45]
        TcomViaSW_o = TcomViaSW_o − TcomOld_v
        TcomOld_v = TexecORSIHWrdHWnaIrq(nBits_v, nRD_v, pRD_v,
                    nAcessosIRQ, FN_o) // [EQ. A47]
        TexecRSI(o_paralelo) = TexecRSI(o_paralelo) − TcomOld_v
        nAcessosIRQ = nAcessosIRQ − 1
      fse

    // (A2). hardware lê do software
    senão se (pold_v = SW) então
      se (o_paralelo = NULL) então // Não existe paralelismo entre o e software
        TcomOld_v = TcomHWrdSW(nBits_v, nRD_v, pRD_v, p_o) // [EQ. A26]
        TcomViaSW_o = TcomViaSW_o − TcomOld_v
      senão // Existe paralelismo entre o e o_paralelo (do fluxo de software)
        TcomOld_v = TcomHWrdSWirq(nBits_v), nRD_v, pRD_v, p_o, nAcessosIRQ) // [EQ. A27]
        TcomViaSW_o = TcomViaSW_o − TcomOld_v
        TcomOld_v = TexecORSIHWrdSWirq(nBits_v, nRD_v, pRD_v, p_o,
                    nAcessosIRQ, FN_o) // [EQ. A28]
        TexecRSI(o_paralelo) = TexecRSI(o_paralelo) − TcomOld_v
        nAcessosIRQ = nAcessosIRQ − 1
      fse

    // (A1). hardware lê da mesma partição de hardware
    senão se (pold_v = p_o)) então
      TcomOld_v = TcomHWrdHW(nElement_v, nRD_v, pRD_v) // [EQ. R1]
      TcomLocal_o = TcomLocal_o − TcomOld_v
    fse // Fim de (A3e). hardware lê do hardware adjacente à esquerda

  fcaso // Fim das alternativas para a nova partição de v

 fse // Fim das alternativas em que a partição do estado programa (o) que lê v é hardware

fse // Fim das alternativas em que a operação de o sobre v é de LEITURA
```

Figura G.33: Algoritmo que actualiza o tempo de comunicação dum estado programa, quando uma variável por ele acedida (lida ou escrita) muda de partição *(parte 9)*.

actualizarTcomDevidoObjMudar $(o, pold_o, pnew_o, v, p_v, RDouWR, o_{paralelo},$
$nWR_v, pWR_v, nRD_v, pRD_v, nBits_v, nElement_v, FN_o, nAcessosIRQ) \equiv$

// Retirar ao tempo despendido em comunicação via interrupção o tempo introduzido
// pela comunicação de o com v, quando o se encontrava na sua partição anterior ($pold_o$)

se ($o_{paralelo} \neq NULL$) então // Existe paralelismo entre o e $o_{paralelo}$ (do fluxo de *software*)

se ($pold_o = HW1|HW2|HW3|HW4$) então // PARTIÇÃO DE *HARDWARE*

se ($RDouWR = WR$) então // OPERAÇÃO DE ESCRITA

se ($p_v = particaoNaoAdjacente(pold_o)$) então // (B4). *hardware* escreve *hardware* não adjacente
$TcomOld_v = TexecORSIHWwrHWnaIrq(nBits_v, nWR_v, pWR_v,$
$nAcessosIRQ, FN_o)$ // [EQ. B48]
$TexecRSI(o_{paralelo}) = TexecRSI(o_{paralelo}) - TcomOld_v$
$nAcessosIRQ = nAcessosIRQ - 1$

senão se ($p_v = SW$) então // (B2). *hardware* escreve no *software*
$TcomOld_v = TexecORSIHWwrSWirq(nBits_v, nWR_v, pWR_v, pold_o,$
$nAcessosIRQ, FN_o)$ // [EQ. B26]
$TexecRSI(o_{paralelo}) = TexecRSI(o_{paralelo}) - TcomOld_v$
$nAcessosIRQ = nAcessosIRQ - 1$
fse

senão se ($RDouWR = RD$) então // OPERAÇÃO DE LEITURA

se ($p_v = particaoNaoAdjacente(pold_o)$) então // (A4). *hardware* lê do *hardware* não adjacente
$TcomOld_v = TexecORSIHWrdHWnaIrq(nBits_v, nRD_v, pRD_v, nAcessosIRQ, FN_o)$ // [EQ. A47]
$TexecRSI(o_{paralelo}) = TexecRSI(o_{paralelo}) - TcomOld_v$
$nAcessosIRQ = nAcessosIRQ - 1$

senão se ($p_v = SW$) então // (A2). *hardware* lê do *software*
$TcomOld_v = TexecORSIHWrdSWirq(nBits_v, nRD_v, pRD_v, pold_o,$
$nAcessosIRQ, FN_o)$ // [EQ. A28]
$TexecRSI(o_{paralelo}) = TexecRSI(o_{paralelo}) - TcomOld_v$
$nAcessosIRQ = nAcessosIRQ - 1$
fse

fse // Fim das alternativas em que a operação de o sobre v é de LEITURA
fse // Fim das alternativas em que a partição anterior de o é *HARDWARE*
fse // Fim das alternativas em que existe paralelismo entre o e $o_{paralelo}$

//– –
// Adicionar ao tempo despendido em comunicação, o tempo introduzido pela
// comunicação de o com v, quando o passa para a sua nova partição ($pnew_o$)

se ($RDouWR = WR$) então // ESCRITA

se ($pnew_o = SW$) então // PARTIÇÃO DE S/W (tempo comunicação guardado em $TcomLocal_o$)
caso p_v seja:
SW => // (D1). *software* escreve no *software*
$Tcom_v = TcomSWwrSW(nWR_v, pWR_v)$ // [EQ. D1]
$TcomLocal_o = TcomLocal_o + Tcom_v$

HW1 | HW2 | HW3 | HW4 => // (D2). *software* escreve no *hardware*
$Tcom_v = TcomSWwrHW(nBits_v, nWR_v, pWR_v, p_v)$ // [EQ. D24]
$TcomLocal_o = TcomLocal_o + Tcom_v$
fcaso

Figura G.34: Algoritmo que actualiza o tempo de comunicação entre um estado programa e uma variável por ele acedida (lida ou escrita), quando este estado muda de partição *(parte 1)*.

senão se ($pnew_o = HW1|HW2|HW3|HW4$) então // PARTIÇÃO DE *HARDWARE*
// O tempo de comunicação é $MAX[TcomLocal_o, TcomViaSW_o, TcomDir_o, TcomEsq_o]$
// O tempo de comunicação não é calculado nesta função

caso p_v seja:
$pnew_o$ => // (B1). *hardware* escreve dentro da mesma partição de *hardware*

$Tcom_v = TcomHWwrHW(nElement_v, nWR_v, pWR_v)$ // [EQ. W1]
$TcomLocal_o = TcomLocal_o + Tcom_v$

SW => // (B2). *hardware* escreve no *software*

se ($o_{paralelo} = NULL$) então // Não existe paralelismo entre o e *software*
$Tcom_v = TcomHWwrSW(nBits_v, nWR_v, pWR_v, pnew_o)$ // [EQ. B27]
$TcomViaSW_o = TcomViaSW_o + Tcom_v$
senão // Existe paralelismo entre o e $o_{paralelo}$ (do fluxo de *software*)
$Tcom_v = TcomHWwrSWirq(nBits_v, nWR_v, pWR_v, pnew_o, nAcessosIRQ)$ // [EQ. B25]
$TcomViaSW_o = TcomViaSW_o + Tcom_v$
$Tcom_v = TexecORSIHWwrSWirq(nBits_v, nWR_v, pWR_v, pnew_o,$
$nAcessosIRQ, FN_o)$ // [EQ. B26]
$TexecRSI(o_{paralelo}) = TexecRSI(o_{paralelo}) + Tcom_v$
$nAcessosIRQ = nAcessosIRQ + 1$
fse

$particaoNaoAdjacente(pnew_o)$ => // (B4). *hardware* escreve *hardware* não adjacente

se ($o_{paralelo} = NULL$) então // Não existe paralelismo entre o e *software*
$Tcom_v = TcomHWwrHWna(nBits_v, nWR_v, pWR_v)$ // [EQ. B42]
$TcomViaSW_o = TcomViaSW_o + Tcom_v$
senão // Existe paralelismo entre o e $o_{paralelo}$ (do fluxo de *software*)
$Tcom_v = TcomHWwrHWnaIrq(nBits_v, nWR_v, pWR_v, nAcessosIRQ)$ // [EQ. B46]
$TcomViaSW_o = TcomViaSW_o + Tcom_v$
$Tcom_v = TexecORSIHWwrHWnaIrq(nBits_v, nWR_v, pWR_v,$
$nAcessosIRQ, FN_o)$ // [EQ. B48]
$TexecRSI(o_{paralelo}) = TexecRSI(o_{paralelo}) + Tcom_v$
$nAcessosIRQ = nAcessosIRQ + 1$
fse

$particaoDireita(pnew_o)$ => // (B3d). *hardware* escreve *hardware* adjacente à direita

$Tcom_v = TcomHWwrHWa(nBits_v, nWR_v, pWR_v, nPinos_{pnew_o, p_v})$ // [EQ. B34]
$TcomDir_o = TcomDir_o + Tcom_v$

$particaoEsquerda(pnew_o)$ => // (B3e). *hardware* escreve *hardware* adjacente à esquerda

$Tcom_v = TcomHWwrHWa(nBits_v, nWR_v, pWR_v, nPinos_{pnew_o, p_v})$ // [EQ. B34]
$TcomEsq_o = TcomEsq_o + Tcom_v$

fcaso // Fim das alternativas para a partição de v

fse // Fim das alternativas em que a partição do estado programa (o) que escreve v é *hardware*

senão se ($RDouWR = RD$) então // LEITURA

se ($pnew_o = SW$) então // PARTIÇÃO DE *S/W* (tempo comunicação guardado em $TcomLocal_o$)
caso p_v seja:
SW => // (C1). *software* lê do *software*
$Tcom_v = TcomSWrdSW(nRD_v, pRD_v)$ // [EQ. C1]
$TcomLocal_o = TcomLocal_o + Tcom_v$

Figura G.35: Algoritmo que actualiza o tempo de comunicação entre um estado programa e uma variável por ele acedida (lida ou escrita), quando este estado muda de partição *(parte 2)*.

HW1 | HW2 | HW3 | HW4 => // (C2). *software* lê do *hardware*
$Tcom_v = TcomSWrdHW(nBits_v, nRD_v, pRD_v, p_v)$ // [EQ. C24]
$TcomLocal_o = TcomLocal_o + Tcom_v$
fcaso

senão se ($pnew_o = HW1|HW2|HW3|HW4$) então // PARTIÇÃO DE *HARDWARE*
// O tempo de comunicação é $MAX[TcomLocal_o, TcomViaSW_o, TcomDir_o, TcomEsq_o]$
// O tempo de comunicação não é calculado nesta função

caso p_v seja:
$pnew_o$ => // (A1). *hardware* lê da mesma partição de *hardware*

$Tcom_v = TcomHWrdHW(nElement_v, nRD_v, pRD_v)$ // [EQ. R1]
$TcomLocal_o = TcomLocal_o + Tcom_v$

SW => // (A2). *hardware* lê do *software*

se ($o_{paralelo} = NULL$) então // Não existe paralelismo entre o e *software*
$Tcom_v = TcomHWrdSW(nBits_v, nRD_v, pRD_v, pnew_o)$ // [EQ. A26]
$TcomViaSW_o = TcomViaSW_o + Tcom_v$
senão // Existe paralelismo entre o e $o_{paralelo}$ (do fluxo de *software*)
$Tcom_v = TcomHWrdSWirq(nBits_v), nRD_v, pRD_v, pnew_o, nAcessosIRQ)$ // [EQ. A27]
$TcomViaSW_o = TcomViaSW_o + Tcom_v$
$Tcom_v = TexecORSIHWrdSWirq(nBits_v, nRD_v, pRD_v, pnew_o,$
$nAcessosIRQ, FN_o)$ // [EQ. A28]
$TexecRSI(o_{paralelo}) = TexecRSI(o_{paralelo}) + Tcom_v$
$nAcessosIRQ = nAcessosIRQ + 1$
fse

$particaoNaoAdjacente(pnew_o)$ => // (A4). *hardware* lê do *hardware* não adjacente

se ($o_{paralelo} = NULL$) então // Não existe paralelismo entre o e *software*
$Tcom_v = TcomHWrdHWna(nBits_v, nRD_v, pRD_v)$ // [EQ. A4R]
$TcomViaSW_o = TcomViaSW_o + Tcom_v$
senão // Existe paralelismo entre o e $o_{paralelo}$ (do fluxo de *software*)
$Tcom_v = TcomHWrdHWnaIrq(nBits_v, nRD_v, pRD_v, nAcessosIRQ)$ // [EQ. A45]
$TcomViaSW_o = TcomViaSW_o + Tcom_v$
$Tcom_v = TexecORSIHWrdHWnaIrq(nBits_v, nRD_v, pRD_v, nAcessosIRQ, FN_o)$ // [EQ. A47]
$TexecRSI(o_{paralelo}) = TexecRSI(o_{paralelo}) + Tcom_v$
$nAcessosIRQ = nAcessosIRQ + 1$
fse

$particaoDireita(pnew_o)$ => // (A3d). *hardware* lê do *hardware* adjacente à direita

$Tcom_v = TcomHWrdHWa(nBits_v, nRD_v, pRD_v, nPinos_{pnew_o, p_v})$ // [EQ. A34]
$TcomDir_o = TcomDir_o + Tcom_v$

$particaoEsquerda(pnew_o)$ => // (A3e). *hardware* lê do *hardware* adjacente à esquerda

$Tcom_v = TcomHWrdHWa(nBits_v, nRD_v, pRD_v, nPinos_{pnew_o, p_v})$ // [EQ. A34]
$TcomEsq_o = TcomEsq_o + Tcom_v$

fcaso // Fim das alternativas para a nova partição de v

fse // Fim das alternativas em que a partição do estado programa (o) que lê v é *hardware*

fse // Fim das alternativas em que a operação de o sobre v é de LEITURA

Figura G.36: Algoritmo que actualiza o tempo de comunicação entre um estado programa e uma variável por ele acedida (lida ou escrita), quando este estado muda de partição *(parte 3)*.

Apêndice H

Implementação da Estimação do Tempo de Execução do Sistema

```
calcTexecFlux (fluxId) ≡

  se (Texecução do fluxo fluxId já foi totalmente calculado) então
    devolver (Texecução do fluxo fluxId obtido na iteração anterior)

  senão se (cálculo de Texecução do fluxo fluxId não foi iniciado) então
    Texec = Texecução de PPanterior // Ver figura 7.10

  senão se (cálculo de Texecução do fluxo fluxId já foi iniciado) então
    se (último nodo calculado no fluxo não é do tipo inicioParalelo) então
      Definir um sub-fluxo a partir do último nodo calculado no fluxo
    senão // Último nodo calculado no fluxo é do tipo inicioParalelo

      para ( (cada fluxo dFluxId descendente do fluxo actual e
           iniciado no nodo do tipo inicioParalelo) E
           (enquanto o nodoProcurado não for atingido) ) fazer
        Texec = calcTexecFlux (dFluxId)
      fpara

      se (construtor paralelo foi totalmente calculado) então
        Texec = maximo (Texecução para cada ramo do construtor)
      senão // Terminou-se o cálculo do fluxo num nodo do tipo activacao
            // ou comutacao interveniente num ponto de sincronismo
        devolver (Texec)
      fse
    fse
  fse
  enquanto ( (não se chega ao fim do fluxo actual) OU
      (não se atinge o ponto de sincronismo procurado) ) fazer
    // De acordo com o tipo do nodo final do sub-fluxo (TNF), utiliza-se a função
    // que calcula Texecução num sub-fluxo e é adequada ao tipo de nodo TNF.
    // As diferentes funções encontram-se nas figuras H.2 a H.9
    Texec = calcTexecSubFlux_TNF (Texec)
  fenquanto

  devolver (Texec)
```

Figura H.1: Função *calcTexecFlux* que calcula o tempo de execução no final dum fluxo do grafo.

```
calcTexecSubFlux_II () ≡

  se (nodo final do sub-fluxo está inserido num ciclo) então
    [ ... ]

  senão // O nodo final do sub-fluxo não está inserido num ciclo
    dT = calcTexecSubFlux()
    sTexec = Texec[PPanterior] + dT
    Texec[PPactual] = sTexec
    Avançar para o próximo sub-fluxo
  fse
  devolver (sTexec)
```

Figura H.2: Função $calcTexecSubFlux\_II$ que calcula o tempo de execução de sub-fluxos que terminam num nodo do tipo $inicioIf$.

```
calcTexecSubFlux_FI () ≡

  // Consideram-se apenas construtores condicionais com 2 ramos
  dT = calcTexecSubFlux ()

  se (nodo final do sub-fluxo está inserido num ciclo) então
    [ ... ]

  senão // O nodo final do sub-fluxo não está inserido num ciclo

    // O número de vezes em que se visita o início do construtor condicional é impar
    se (nVezes[PPactual] % 2 = 1) então
      sTexec = Texec[PPanterior] + dT
      TauxFI[PPactual][0] = sTexec
      Avançar para o início do segundo ramo do construtor condicional

    // O número de vezes em que se visita o início do construtor condicional é par
    senão se (nVezes[PPactual] % 2 = 0) então
      TauxFI[PPactual][1] = Texec[PPanterior] + dT

      prob0 = probabilidade de ramificação associada ao 1º arco de saída do nodo tipo
              inicioIf que define o início do construtor condicional que termina em PPactual

      prob1 = probabilidade de ramificação associada ao 2º arco de saída do nodo tipo
              inicioIf que define o início do construtor condicional que termina em PPactual

      sTexec = TauxFI[PPactual][0] * prob0 + TauxFI[PPactual][1] * prob1
      Texec[PPactual] = sTexec
      Avançar para o próximo sub-fluxo
    fse
  fse
  devolver (sTexec)
```

Figura H.3: Função $calcTexecSubFlux\_FI$ que calcula o tempo de execução de sub-fluxos que terminam num nodo do tipo $fimIf$.

```
calcTexecSubFlux_CC () ≡

  dT = calcTexecSubFlux ()

  se (nodo final do sub-fluxo está inserido num ciclo) então
    [ ... ]

  senão // O nodo final do sub-fluxo não está inserido num ciclo

    // NCICLOS (=2) é o número de passagens pelo corpo dum ciclo necessário
    // para obter estimativas precisas do tempo de execução.
    // nVezesCiclo é o número de vezes que já se visitou o nodo.

    // Primeira vez que o controlo do ciclo é visitado
    se (o resto da divisão de nVezesCiclo por (NCICLOS+1) for 1) então
      sTexec = Texec[PPanterior] + dT
      TauxCC[PPactual][0] = sTexec
      Avançar para o próximo sub-fluxo

    // Última vez que o controlo do ciclo é visitado
    senão se (o resto da divisão de nVezesCiclo por (NCICLOS+1) for 0) então
      TauxCC[PPactual][NCICLOS] = TauxCC[PPanterior][NCICLOS − 1] + dT
      // Texec[PPanterior] já possui o valor esperado na última iteração do ciclo
      sTexec = Texec[PPanterior] + dT
      Texec[PPactual] = sTexec
      Avançar para o próximo sub-fluxo

    // Não é a primeira nem a última vez que o controlo do ciclo é visitado
    senão
      sTexec = TauxCC[PPanterior][(nVezesCiclo%NCICLOS) − 2] + dT
      TauxCC[PPactual][(nVezesCiclo%NCICLOS) − 1] = sTexec
      Avançar para o próximo sub-fluxo
    fse
  fse
  devolver (sTexec)
```

Figura H.4: Função *calcTexecSubFlux_CC* que calcula o tempo de execução de sub-fluxos que terminam num nodo do tipo *controloCiclo*.

```
calcTexecSubFlux_A_C () ≡

  // nVezes é o número de vezes que já se calculou Texecução em PPactual

  se (nodo final do sub-fluxo está inserido num ciclo) então
    [ ... ]

  senão // O nodo final do sub-fluxo não está inserido num ciclo
    dT = calcTexecSubFlux()
    sTexec = Texec[PPanterior] + dT
    Texec[PPactual] = sTexec
    TexecHistoric[PPactual][nVezes − 1] = sTexec
    se (PPactual não é o ponto de sincronismo procurado) então
      Avançar para o próximo sub-fluxo
    fse
  fse
  devolver (sTexec)
```

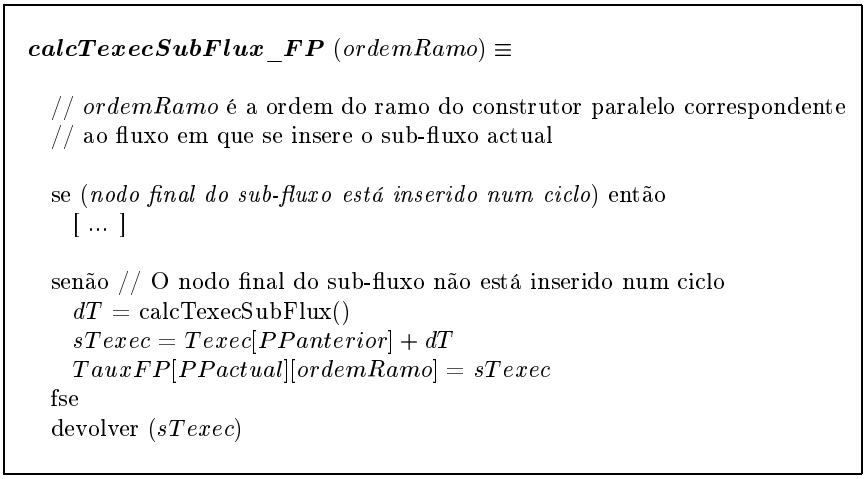
Figura H.5: Função *calcTexecSubFlux_A_C* que calcula o tempo de execução de sub-fluxos que terminam num nodo do tipo *activacao* ou *comutacao*.

```
calcTexecSubFlux_E_S () ≡

  se (nodo final do sub-fluxo está inserido num ciclo) então
    [ ... ]

  senão // O nodo final do sub-fluxo não está inserido num ciclo
    dT = calcTexecSubFlux()

    // TexecPreS é o tempo de execução do sub-fluxo sem contabilizar o tempo
    // que se espera por eventos nos sinais envolvidos no sincronismo
    TexecPreS = Texec[PPanterior] + dT

    // Actualizar o tempo de execução do sub-fluxo com o tempo que
    // se espera por eventos nos sinais envolvidos no sincronismo
    TexecS = calcTexecAfterSyncESnode(TexecPreS)
    Texec[PPactual] = TexecS
    Avançar para o próximo sub-fluxo
  fse
  devolver (TexecS)
```

Figura H.6: Função $calcTexecSubFlux\_E\_S$ que calcula o tempo de execução de sub-fluxos que terminam num nodo do tipo *espera* ou *sincronizacao*.

```
calcTexecSubFlux_IP () ≡

  se (nodo final do sub-fluxo está inserido num ciclo) então
    [ ... ]

  senão // O nodo final do sub-fluxo não está inserido num ciclo
    dT = calcTexecSubFlux()
    sTexec = Texec[PPanterior] + dT
    Texec[PPactual] = sTexec
    Avançar para o sub-fluxo inicial do primeiro fluxo que
      descende do nodo do tipo inicioParalelo
    sTexec = tempo de execução do nodo do tipo inicioParalelo

    // Estimar o Texecução no fim de cada fluxo descendente do nodo do tipo inicioParalelo
    para (cada fluxo que descende do nodo do tipo inicioParalelo) fazer
      sTexec = calcTexecFlux (ordemFluxo)
    fpara

    // A estimação do Texecução no construtor paralelo foi concluída
    se (estimação do Texecução no construtor paralelo foi concluída) então
      // O Texecução dos ramos do construtor paralelo está em TauxFP[ppFP][0 : nRamos − 1],
      // sendo ppFP o ponto de paragem associado ao fim desse construtor
      sTexec = maximo (Texecução para cada ramo do construtor paralelo)
    fse
    Avançar para o próximo sub-fluxo
  fse
  devolver (sTexec)
```

Figura H.7: Função $calcTexecSubFlux\_IP$ que calcula o tempo de execução de sub-fluxos que terminam num nodo do tipo *inicioParalelo*.



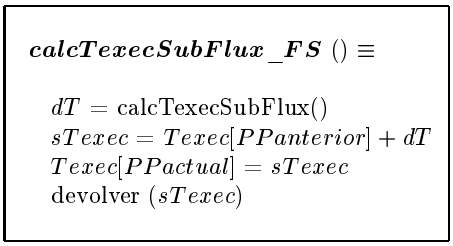
calcTexecSubFlux_FP ($ordemRamo$) ≡

// $ordemRamo$ é a ordem do ramo do construtor paralelo correspondente
// ao fluxo em que se insere o sub-fluxo actual

se (*nodo final do sub-fluxo está inserido num ciclo*) então
[...]

senão // O nodo final do sub-fluxo não está inserido num ciclo
dT = calcTexecSubFlux()
$sTexec = Texec[PPanterior] + dT$
$TauxFP[PPactual][ordemRamo] = sTexec$
fse
devolver ($sTexec$)

Figura H.8: Função $calcTexecSubFlux\_FP$ que calcula o tempo de execução de sub-fluxos que terminam num nodo do tipo $fimParalelo$.



calcTexecSubFlux_FS () ≡

dT = calcTexecSubFlux()
$sTexec = Texec[PPanterior] + dT$
$Texec[PPactual] = sTexec$
devolver ($sTexec$)

Figura H.9: Função $calcTexecSubFlux\_FS$ que calcula o tempo de execução de sub-fluxos que terminam num nodo do tipo $fimSistema$.

```
calcTexecSubFlux () ≡

  // Consultar a figura 7.10
  // anteces é nodo antecessor do nodo inicial do sub-fluxo
  // antecesP é a partição a que o nodo anteces foi atribuída
  // antecesT é o tipo do nodo anteces
  // targetNode(oEdge) devolve o nodo alvo do arco oEdge

  anteces = objPrev
  antecesPv = atribuicaoObj (anteces)
  antecesT = tipoObj (anteces)
  o = targetNode (edgeInitial)
  oP = atribuicaoObj (objInitial)
  oT = tipoObj (objInitial)
  oEdge = arco de saída do nodo o
  dTexec = 0

  fazer
    oTexec = Texec(o)
    dTexec = dTexec + TpartitionComutation (antecesP, oP, antecesT, oT) + oTexec
    anteces = o
    antecesP = oP
    antecesT = oT
    se (o != objFinal) então
      o = targetNode (oEdge)
      oEdge = arco de saída do nodo o
      oT = tipoObj (o)
      oP = atribuicaoObj (o)
    fse
  enquanto (anteces != objFinal)

  devolver (dTexec)
```

Figura H.10: Função *calcTexecSubFlux* que calcula o tempo de execução no final dum sub-fluxo do grafo.

```
TpartitionComutation () ≡

  // prevP é a partição a que o nodo anterior está atribuído
  // p é a partição a que o nodo actual está atribuído
  // prevType é o tipo do nodo anterior
  // type é o tipo do nodo actual

  se ( (prevType=inicioParalelo) OU (prevType=fimParalelo) ) então
    Tcom = 0

  // Mudança de software para hardware
  senão se ((prevP=SW) E ((p=HW1) OU (p=HW2) OU (p=HW3) OU (p=HW4))) então
    Tcom = TwriteSW2HW + 2 * THWclock

  // Mudança de hardware para software
  senão se ((p=SW) E ((prevP=HW1) OU (prevP=HW2) OU (prevP=HW3) OU
      (prevP=HW4)) ) então
    Tcom = 2 * TreadHW2SW + TwriteHW2HW + 2 * TcomparacaoSW

  // Mudança de hardware para hardware adjacente
  senão se ( ((prevP=HW1) OU (prevP=HW2) OU (prevP=HW3) OU (prevP=HW4)) E
      ( (p=particaoEsquerda(prevP)) OU (p=particaoDireita(prevP))) ) então
    Tcom = TwriteHW2HWa + 2 * THWclock

  // Mudança de hardware para hardware não adjacente
  senão se ( ((prevP=HW1) OU (prevP=HW2) OU (prevP=HW3) OU
      (prevP=HW4)) E (p=particaoNaoAdjacente(prevP)) ) então
    Tcom = TwriteHW2HW + 2 * TreadHW2SW + TwriteSW2HW +
             2 * (TcomparacaoSW + THWclock)

  senão
    Tcom = 0
  fse

  devolver (Tcom)
```

Figura H.11: Função *TpartitionComutation* que define o tempo de comunicação associado a uma mudança de partição num fluxo dum grafo.

```
calcTexecAfterSyncESnode (TexecPreSync) ≡

  // TexecPreSync é o tempo de execução no nodo objES, do tipo espera (E) ou
  // sincronizacao (S), antes de contabilizar o tempo que se espera por eventos
  // nos sinais envolvidos no sincronismo
  ppES = ponto de paragem associado ao nodo objES que gera o sincronismo
  // nVezes[ppES] é o número de vezes que o Texecução de ppES já foi calculado

  // Ciclo para todas as variáveis lidas pelo nodo objES
  para (cada variável v ∈ readVAR(objES)) fazer
    Tdet = 0 // Tempo de execução determinístico
    Tnondet = 0 // Tempo de execução não determinístico

    // Ciclo para todos os nodos do tipo A (C) que escrevem a variável lida pelo nodo objES
    // e podem participar no sincronismo porque foram validados pelas regras EpS1-3,5,6
    para ((cada nodo objAC ∈ OBJwrite(v)) E
          (objAC ∈ lista de participantes no sincronismo com objES)) fazer
      ppAC = ponto de paragem associado ao nodo objAC

      se (não é necessário estimar o Texecução do nodo objAC) então
        TexecAC = TexecHistoric[ppAC][nVezes[ppES] − 1]

      senão // É necessário estimar o Texecução do nodo objAC
        // Estimar Texecução até encontrar o nodo objAC e começando no arco splitEdge que
        // define o ponto de divergência entre os fluxos que conduzem a objAC e objES
        calcTexecFlux (objAC, splitEdge)
        TexecAC = TexecHistoric[ppAC][nVezes[ppES] − 1]
      fse

      // O nodo objAC activa a variável v quando o nodo objES está à espera
      se (TexecAC > TexecPreSync) então
        se (valor escrito pelo nodo objAC é determinístico) então
          // Tempo de execução mínimo entre todos os nodos do tipo A (C)
          Tdet = minimo (TexecAC, Tdet)

        // O valor escrito pelo nodo objAC é não-determinístico
        senão
          // Tempo de execução máximo entre todos os nodos do tipo A (C)
          Tnondet = maximo (TexecAC, Tnondet)
        fse
      fse
    fpara

    // Tempo que o nodo objES espera pela variável v
    T1var = maximo (Tdet,Tnondet)

    se (sincronismo é do tipo multi-OR) então
      // Tempo de execução mínimo devido à espera pela primeira das variáveis
      TexecAfterSync = minimo (T1var, TexecAfterSync)

    senão // O sincronismo é do tipo single, multi-AND ou mixed
      // Tempo de execução máximo devido à espera por todas as variáveis
      TexecAfterSync = maximo (T1var, TexecAfterSync)
    fse
  fpara
  devolver (TexecAfterSync)
```

Figura H.12: Função *calcTexecAfterSyncESnode* que actualiza o tempo de execução dum nodo, do tipo *espera* ou *sincronizacao*, com o tempo que o nodo espera por eventos nos sinais envolvidos no sincronismo.

Bibliografia

[AG94] George Almasi e Allan Gottlieb. *Highly Parallel Computing*. The Benjamin/Cummings Publishing Company, 2ª edição, 1994.

[BCO95] G. Borriello, P. Chou e R. Ortega. *Embedded System Co-design: Towards Portability and Rapid Integration*, pág.s 243–264. *Hardware/Software Codesign*, Ed. M. Sami e G. De Micheli. Kluwer Academic Publishers, Boston, USA, 1995.

[Ber98] Michel Berkelaar. lp_solve 3.0 - Solver for Mixed Integer Linear Programming Problems, 1998. `ftp://ftp.es.ele.tue.nl/pub/lp_solve/`.

[BFS95] A. Balboni, W. Fornaciari e D. Sciuto. *TOSCA: A Pragmatic Approach to Co-design of Control-dominated Systems*, pág.s 265–294. *Hardware/Software Codesign*, Ed. M. Sami e G. De Micheli. Kluwer Academic Publishers, Boston, USA, 1995.

[BFS96a] A. Balboni, W. Fornaciari e D. Sciuto. Co-synthesis and Co-simulation of Control-dominated Embedded Systems. *Design Automation for Embedded Systems*, 1(3):257–289, 1996.

[BFS96b] A. Balboni, W. Fornaciari e D. Sciuto. Partitioning and Exploration Strategies in the TOSCA Co-design Flow. Em *Proceedings of the 4th International Workshop on Hardware/Software Codesign*, pág.s 62–69, 1996.

[BFS98] A. Balboni, W. Fornaciari e D. Sciuto. Partitioning of H/w-S/w Embedded Systems: a Metrics-Based Approach. *Integrated Computer-Aided Engineering, John Wiley Interscience Journal*, 5(1), 1998.

[BG92] G. Berry e G. Gonthier. The Esterel Synchronous Programming Language: Design, Semantics, Implementation. *Science of Computer Programming*, 19(2):87–152, 1992.

[Boo94] Grady Booch. *Object-Oriented Analysis and Design with Applications*. The Benjamin/Cummings Publishing Company, Rational, Santa Clara, California, 2ª edição, 1994.

[Bre95] Barry B. Brey. *The Intel 32-Bit Microprocessors: 80386, 80486 and Pentium.* Prentice-Hall, 1995.

[BRX93] Edna Barros, Wolfgang Rosenstiel e X. Xiong. Hardware/Software Partitioning with UNITY. Em *Proceedings of the 2nd International Workshop on Hardware/Software Codesign*, Outubro 1993. Cambridge, Massachusetts.

[BS94] Edna Barros e Augusto Sampaio. Towards Provably Correct Hardware/Software Partitioning using Occam. Em *Proceedings of the 3rd International Workshop on Hardware/Software Codesign*, pág.s 210–217. IEEE Computer Society Press, Setembro 1994. Grenoble, France.

[CA97] Anton V. Chichkov e Carlos B. Almeida. An Hardware/Software Partitioning Algorithm for Custom Computing Machines. Em *Field Programmable Logic and Applications - Proceedings of the 7th International Workshop FPL'97*, pág.s 274–283, Setembro 1997.

[CEG+96] M. Chiodo, D. Engels, P. Giusto, H. Hsieh, A. Jurecska, L. Lavagno, K. Suzuki e A. Sangiovanni-Vincentelli. A Case Study in Computer-Aided Co-design of Embedded Controllers. *Design Automation for Embedded Systems*, 1(1-2):51–67, 1996.

[CGH+93] M. Chiodo, P. Giusto, H. Hsieh, A. Jurecska, L. Lavagno e A. Sangiovanni-Vincentelli. A Formal Specification Model for Hardware/Software Codesign. Relatório Técnico ERL-93-48, University of California - Berkeley, Junho 1993.

[CGH+94] M. Chiodo, P. Giusto, H. Hsieh, A. Jurecska, L. Lavagno e A. Sangiovanni-Vincentelli. A Formal Methodology for Hardware/Software Co-design of Embedded Systems. *IEEE Micro*, Agosto 1994.

[CLL+96] C. Carreras, J. C. López, M. L. López, C. Delgado-Kloos, N. Martínez e L. Sánchez. A Co-Design Methodology Based on Formal Specification and High-Level Estimation. Em *Proceedings of the 4th International Workshop on Hardware/Software Codesign*, pág.s 28–35, Março 1996.

[CLR90] Thomas Cormen, Charles Leiserson e Ronald Rivest. *Introduction to Algorithms.* MIT Press, Cambridge, Massachusetts, USA, Agosto 1990.

[COB95] P. Chou, R. Ortega e G. Borriello. The Chinook Hardware/Software Co-synthesis System. Em *Proceedings of the 8th International Symposium on Systems Synthesis (ISSS)*, pág.s 22–27, 1995.

[CW96] R. Camposano e J. Wilberg. Embedded Systems Design. *Design Automation for Embedded Systems*, 1(1-2):5–50, 1996.

[DH94] Joseph D'Ambrosio e Xiaobo Hu. Configuration-Level Hardware/Software Partitioning for Real-Time Embedded Systems. Em *Proceedings of the 3rd International Workshop on Hardware/Software Codesign*, pág.s 34–41. IEEE Computer Society Press, Setembro 1994. Grenoble, France.

[DJ92] Nikil Dutt e Pradip Jha. Rapid Estimation for Parameterized Components in High-Level Synthesis. Em *Proceedings of the 6th International Workshop on High-Level Synthesis*, 1992.

[DK99] William Daley e Raymond Kammer. Data Encryption Standard (DES). Federal Information Processing Standards publication FIPS PUB 46-3, National Institute of Standards and Technology (NIST), Outubro 1999.

[DMVJ97] J. Daveau, G. Marchioro, C. Valderama e A. Jerraya. *VHDL Generation from SDL Specifications*, pág.s 182–201. Hardware Description Languages and their Applications. Chapman & Hall, 1997.

[EFP97] António Esteves, J.Miguel Fernandes e Alberto Proença. *EDgAR: A Platform for Hardware/Software Codesign*, pág.s 19–32. *Embedded System Applications*, Ed. C.Baron, J.-C. Geffroy e G.Motet. Kluwer Academic Publishers, Boston, USA, 1997.

[EHB93] Rolf Ernst, Jörg Henkel e Thomas Benner. Hardware-Software Cosynthesis for Microcontrollers. *IEEE Design & Test of Computers*, 10(4):64–75, Dezembro 1993.

[EKP98a] Petru Eles, Krkrysztof Kuchcinski e Zebo Peng. *System Synthesis with VHDL*. Kluwer Academic Publishers, 1998.

[EKP+98b] Petru Eles, Krkrysztof Kuchcinski, Zebo Peng, Alexa Doboli e Paul Pop. Process Scheduling for Performance Estimation and Synthesis of Hardware/Software Systems. Em *Proceedings of the 24th EUROMICRO Conference*, 1998.

[EPD94] Petru Eles, Zebo Peng e Alexa Doboli. VHDL System-Level Specification and Partitioning in a Hardware/Software Co-Synthesis Environment. Em *Proceedings of the 3rd International Workshop on Hardware/Software Codesign*, pág.s 49–55. IEEE Computer Society Press, Setembro 1994.

[EPKD97] Petru Eles, Zebo Peng, Krkrysztof Kuchcinski e Alexa Doboli. System Level Hardware/Software Partitioning Based on Simulated Annealing and Tabu Search. *Design Automation for Embedded Systems*, 2(1):5–32, 1997.

[Ess96] R. Esser. *An Object Oriented Petri Net Approach to Embedded System Design.* Dissertation ETH Zurich Nº 11869, 1996.

[Est98] António Esteves. EDgAR-2: Highly Re-configurable Digital Emulator. Relatório Técnico UMDITR9805, Dep. de Informática, Universidade do Minho, Braga, Portugal, Dezembro 1998.

[Est01] António Esteves. Documentação sobre a Tese de Doutoramento. Dep. de Informática, Universidade do Minho, Braga, Portugal, Junho 2001. `http://www.di.uminho.pt/~esteves/phd/phd.html`.

[EVD89] P.H. Eijk, C. Vissers e M. Diaz. The Formal Description Technique LOTOS. *Elsevier Science Publishers B.V.*, 1989.

[EYCP00] AJ Elbirt, W Yip, B Chetwynd e C Paar. An FPGA Implementation and Performance Evaluation of the AES Block Cipher Candidate Algorithm Finalists. Em *Proceedings of the Third Advanced Encryption Standard Candidate Conference*, Abril 2000.

[Fer00] João Miguel Fernandes. *MIDAS: Metodologia Orientada ao Objecto para o Desenvolvimento de Sistemas Embebidos.* Tese de doutoramento, Dep. de Informática, Universidade do Minho, Braga, Portugal, Fevereiro 2000.

[FM82] C. Fiduccia e R. Mattheyeses. A Linear-Time Heuristic for Improving Network Partitions. Em *Proceedings of the Design Automation Conference*, 1982.

[FSV97] L. Ferrandi, D. Sciuto e M. Vincenzi. TOSCA User's Manual, Version 2.0, Setembro 1997. `http://www.cefriel.it/eda/projects/seed/um/mainmenuum.htm`.

[GDWL92] Daniel Gajski, Nikil Dutt, Allen Wu e Steve Lin. *High-Level Synthesis - Introduction to Chip and System Design.* Kluwer Academic Publishers, 1992.

[GGN94] Jie Gong, Daniel Gajski e Sanjiv Narayan. Software Estimation from Executable Specifications. *Journal of Computer and Software Engineering*, 1994.

[GK83] Daniel Gajski e R. Kuhn. Guest's editors introduction: New VLSI Tools. *IEEE Computer*, 16:11–14, 1983.

[GL95] Fred Glover e Manuel Laguna. *Tabu Search*, pág.s 70–150. *Modern Heuristic Techniques for Combinatorial Problems*, Ed. Colin Reeves. McGraw-Hill Inc., 1995.

[GM92] Rajesh K. Gupta e Giovanni De Micheli. System-level Synthesis using Re-programmable Components. Em *Proceedings of the European Conference on Design Automation*, pág.s 2–7, Brussels, Belgium, Fevereiro 1992.

[GM94] Rajesh K. Gupta e Giovanni De Micheli. Constrained Software Generation for Hardware-Software Systems. Em *Proceedings of the 3rd International Workshop on Hardware/Software Codesign*, pág.s 56–63. IEEE Computer Society Press, Setembro 1994.

[GM96] Rajesh K. Gupta e Giovanni De Micheli. A Co-synthesis Approach to Embedded Systems Design Automation. *Design Automation for Embedded Systems*, 1(1-2):69–120, Janeiro 1996.

[GMZ97] Daniel Gajski, G. Marchioro e J. Zhu. *Essential Issues in Codesign*, pág.s 1–45. Kluwer Academic Publishers, 1997.

[Gup93] Rajesh K. Gupta. *Co-synthesis of Hardware and Software for Digital Embedded Systems*. Tese de doutoramento, Stanford University, Dezembro 1993.

[GV95] Daniel Gajski e Frank Vahid. Specification and Design of Embedded Hardware-Software Systems. *IEEE Design & Test of Computers*, (Spring):53–67, 1995.

[GVN94] Daniel Gajski, Frank Vahid e Sanjiv Narayan. A System-Design Methodology: Executable-Specification Refinement. Em *Proceedings of the European Conference on Design Automation*, 1994.

[GVNG94] Daniel Gajski, Frank Vahid, Sanjiv Narayan e Jie Gong. *Specification and Design of Embedded Systems*. Prentice-Hall, 1994.

[GVNG96] Daniel Gajski, Frank Vahid, Sanjiv Narayan e Jie Gong. SpecSyn: An Environment Supporting the Specify-Explore-Refine Paradigm for Hardware/Software Systems Design. Technical Report 96-08, Dept. of Information and Computer Science, University Of California, Irvine, 1996.

[GW87] Rafael Gonzalez e Paul Wintz. *Digital Image Processing*. Addison-Wesley, 2ª edição, 1987.

[GZD+00] D. Gajski, J. Zhu, R. Dömer, A. Gerstlauer e S. Zhao. *SpecC: Specification Language and Methodology*. Kluwer Academic Publishers, Boston, USA, Março 2000.

[Har87] D. Harel. StateCharts: A Visual Formalism for Complex Systems. *Science of Computer Programming*, 1987.

[HE95] Jörg Henkel e Rolf Ernst. A Path-Based Estimation Technique for Estimating Hardware Runtime in HW/SW-Cosynthesis. Em *Proceedings of the 8th International Symposium on System Synthesis (ISSS)*, pág.s 116–121, 1995.

[HE98] Jörg Henkel e Rolf Ernst. High-level Estimation Techniques for Usage in Hardware/Software Co-design. Em *Proceedings of the Asia and South Pacific Design Automation Conference*, pág.s 353–360, Fevereiro 1998.

[Hil85] P. Hilfinger. A High-Level Language and Silicon Compiler for Digital Signal Processing. Em *Proceedings of the Custom Integrated Circuits Conference*, 1985.

[Hoa78] C.A.R. Hoare. Communicating Sequential Processes. *Communications of the ACM*, 21(8):666–677, 1978.

[IEE93] IEEE. *IEEE Standard VHDL Language Reference Manual. IEEE Std. 1076-1993.* IEEE Inc. New York, 1993.

[IEE96] IEEE. *IEEE Hardware Description Language based on the Verilog(TM) Hardware Description Language. IEEE Std. 1364-1996.* IEEE Inc. New York, 1996.

[IJ95] T. B. Ismail e A. A. Jerraya. Synthesis Steps and Design Models for Codesign. *IEEE Computer*, pág.s 44–52, 1995.

[Inm84] Inmos. *OCCAM Programming Manual.* Prentice-Hall, 1984.

[ISO87] ISO. *Estelle (Formal Description Technique based on an Extended State Transition Model).* Standard ISO/DIS 9074, 1987.

[Jen85] Kurt Jensen. An Introduction to High-Level Petri Nets. Em *Proceedings of the International Symposium on Circuits and Systems, Kyoto, Japan*, volume 2, pág.s 723–726. IEEE, 1985.

[JO94] A. A. Jerraya e K. O'Brien. *SOLAR: An Intermediate Format for Hardware for System-Level Modelling and Synthesis. Computer Aided Software/Hardware Engineering*, Ed. J. Rozenblit e K. Buchenrieder. IEEE Press, 1994.

[KCJ98] Lars Kristensen, Soren Christensen e Kurt Jensen. The practitioner's guide to coloured Petri nets. *International Journal on Software Tools for Technology Transfer: Special section on coloured Petri nets*, 2(2):98–132, 1998. `http://sttt.cs.uni-dortmund.de`.

[KdM88] D. Ku e G. de Micheli. HardwareC - A Language for Hardware Design. Relatório Técnico CSL-TR-90-419, Stanford University, 1988.

[KGV83] S. Kirkpatrick, C. Gelatt e M. Vecchi. Optimization by Simulated Annealing. *Science*, 220(4598):671–680, 1983.

[KL93] Asawaree Kalavade e Edward Lee. A Hardware-Software Codesign Methodology for DSP Applications. *IEEE Design & Test of Computers*, pág.s 16–28, 1993.

[KL94] Asawaree Kalavade e Edward Lee. A Global Criticality/Local Phase Driven Algorithm for the Hardware/Software Partitioning Problem. Em *Proceedings of the 3rd International Workshop on Hardware/Software Codesign*, pág.s 42–48. IEEE Computer Society Press, Setembro 1994. Grenoble, France.

[KM96a] Peter Knudsen e Jan Madsen. Aspects of System Modeling in Hardware/Software Partitioning. Em *Proceedings of the 7th International Workshop on Rapid Systems Prototyping*, Junho 1996.

[KM96b] Peter Knudsen e Jan Madsen. PACE: A Dynamic Programming Algorithm for Hardware/Software Partitioning. Em *Proceedings of the 4th International Workshop on Hardware/Software Codesign*, Março 1996.

[KM98] Peter Knudsen e Jan Madsen. Communication Estimation for Hardware/Software Codesign. Em *Proceedings of the 6th International Workshop on Hardware/Software Codesign*, 1998.

[Knu95] Peter Knudsen. Fine-Grain Partitioning in Codesign. Tese de mestrado, Technical University of Denmark, Department of Computer Science, Technical University of Denmark, Lyngby, Denmark, Fevereiro 1995.

[Knu97] Donald E. Knuth. *The Art of Computer Programming, Volume 1, Fundamental Algorithms*. Reading, Massachusetts, Addison-Wesley, 3ª edição, 1997.

[KP98] Jens-Peter Kaps e Christof Paar. Fast DES Implementations for FPGAs and its Application to a Universal Key-Search Machine. Em *Proceedings of the 5th Annual Workshop on Selected Areas in Cryptography (SAC'98)*, Agosto 1998.

[KS98] Sharon Keller e Miles Smid. Modes of Operation Validation System (MOVS): Requirements and Procedures. NIST special publication 800-17, National Institute of Standards and Technology (NIST), Fevereiro 1998.

[LMW95] Yau-Tsun S. Li, Sharad Malik e Andrew Wolfe. Performance Estimation of Embedded Software with Instruction Cache Modeling. Em *Proceedings of the International Conference on Computer-Aided Design*, Novembro 1995.

[LT91] E.D. Lagnese e D.E. Thomas. Architectural Partitioning for System Level Synthesis of Integrated Circuits. *IEEE Transactions on Computer-Aided Design*, 10(7):847–860, Julho 1991.

[Mac00] Ricardo Jorge Machado. *Metodologias de Desenvolvimento em Projectos de Engenharia de Computadores no Suporte à Implementação de Sistemas de Informação Distribuídos Não Convencionais (Industriais).* Tese de doutoramento, Dep. de Sistemas de Informação, Universidade do Minho, Guimarães, Portugal, Novembro 2000.

[MEGT96] Derrick Morris, Gareth Evans, Peter Green e Colin Theaker. *Object Oriented Computer Systems Engineering.* Springer-Verlag, Applied Computing Series, 1996.

[MFES00] Ricardo Machado, João Fernandes, António Esteves e Henrique Santos. *Ch.11 An Evolutionary Approach to the Use of Petri Net based Models: from Parallel Controllers to HW/SW Co-Design*, pág.s 205–222. *Hardware Design and Petri Nets*, Ed. Alex Yakovlev, Luis Gomes e Luciano Lavagno. Kluwer Academic Publishers, Boston, USA, 2000.

[MFP98] Ricardo Machado, João Fernandes e Alberto Proença. An Object-Oriented Model for Rapid Prototyping of Data Path/Control Systems - A Case Study. Em *Proceedings of the 9th IFAC/IFIP Symposium on Information Control in Manufacturing (INCOM)*, volume 2, pág.s 269–274, Junho 1998.

[MGK+97] J. Madsen, J. Grode, P. Knudsen, M. Petersen e A. Axthausen. LYCOS: The Lyngby Co-synthesis System. *Design Automation for Embedded Systems*, 2(2):195–235, 1997.

[MH95] Jan Madsen e Bjarne Halde. An Approach to Interface Synthesis. Em *Proceedings of the International Symposium on Systems Synthesis*, 1995.

[MMMK95] P. Middelhoek, G. Mekenkamp, E. Molenkamp e Th. Krol. VHDL and CDFG Based Transformational Design: a Case Study. Em *Proceedings of the ProRISC/IEEE Workshop on CSSP*, pág.s 203–212, Março 1995.

[MN99] Kurt Mehlhorn e Stefan Näher. *The LEDA Platform of Combinatorial and Geometric Computing.* Cambridge University Press, 1999.

[MNSU00] Kurt Mehlhorn, Stefan Näher, Michael Seel e Christian Uhrig. The LEDA User Manual Version 4.1, 2000. `http://www.mpi-sb.mpg.de/LEDA/leda.html`.

[MQB95] R. Mitra, M. Qadir e A. Basu. A Consistent Labelling Approach to Hardware/Software Partitioning. Em *Proceedings of the International Conference on VLSI Design 95*, pág.s 19–24, 1995.

[MRB96a] R. Mitra, P. Roop e A. Basu. A New Algorithm for Implementation of Design Functions by Available Devices. *IEEE Transactions on VLSI Systems*, 4(2):170–180, 1996.

[MRB96b] R. Mitra, P. Roop e A. Basu. An Overview of MICKEY: An Expert System for Automating the Design of Microprocessor Based Systems. *SADHANA, Journal of the Indian Academy of Science*, 21(Pt.6):719–739, 1996.

[Nav93] Zainalabedin Navabi. *VHDL, Analysis and Modeling of Digital Systems.* McGraw-Hill Inc., 1993.

[NG92a] Sanjiv Narayan e Daniel Gajski. Area and Performance Estimation from System-Level Specification. Technical Report 92-16, Dept. of Information and Computer Science, University Of California, Irvine, 1992.

[NG92b] Sanjiv Narayan e Daniel Gajski. System Clock Estimation based on Clock Slack Minimization. Em *Proceedings of the European Design Automation Conference*, 1992.

[Nie98] Ralf Niemann. *Hardware/Software Co-design for Data Flow Dominated Embedded Systems.* Kluwer Academic Publishers. Boston, USA, 1998.

[NVG92] Sanjiv Narayan, Frank Vahid e Daniel Gajski. System Specification with the SpecCharts Language. *IEEE Design & Test of Computers*, Dezembro 1992.

[Pet81] J. L. Peterson. *Petri Net Theory and Modeling of Systems.* Prentice-Hall, Englewood Cliffs, New Jersey, USA, 1981.

[PH97] D. Patterson e J. Henessy. *Computer Organization and Design: the hardware/software interface.* Morgan Kaufmann Publishers, 2ª edição, 1997.

[Pir96] Marc Pirlot. General Local Search Methods. *European Journal of Operational Research*, pág.s 493–511, 1996.

[RBP+91] J. Rumbaugh, M. Blaha, W. Pemerlani, F. Eddy e W. Lorensen. *Object-Oriented Modeling and Design.* Prentice-Hall, Englewood Cliffs, New Jersey, USA, 1991.

[RH99] M Riaz e H M Heys. The FPGA Implementation of the RC6 and CAST-256 Encryption Algorithms. Em *Proceedings of the IEEE, Canadian Conference on Electrical and Computer Engineering (CCECE)*, Maio 1999.

[RVBM96] K. Van Rompaey, D. Verkest, I. Bolsens e H. De Man. CoWare - A Design Environment for Heterogeneous Hardware/Software Systems. Em *Proceedings of the European Design Automation Conference (EURO-DAC)*, 1996.

[SA95] Tom Shanley e Don Anderson. MindShare, Inc. *PCI System Architecture.* Addison-Wesley, 3ª edição, 1995.

[San96] Henrique Santos. *Metodologias de Especificação e Análise de Sistemas Digitais: desenvolvimento do controlador dum APA (GLiTCH).* Tese de doutoramento, Universidade do Minho, Dep. de Informática, Universidade do Minho, Braga, Portugal, Julho 1996.

[SSV96] Kei Suzuki e Alberto Sangiovanni-Vincentelli. Efficient Software Performance Estimation Methods for Hardware/Software Codesign. Em *Proceedings of the 33rd Design Automation Conference(DAC)*, pág.s 605–610, Junho 1996.

[SVDH01] Stuart Swan, Dirk Vermeersch, Dündar Dumlugöl e Peter Hardee. Functional Specification for SystemC 2.0. Relatório técnico, Synopsys Inc., CoWare Inc. & Frontier Design Inc., Janeiro 2001.

[Vah97] Frank Vahid. Modifying Min-Cut for Hardware and Software Functional Partitioning. Em *Proceedings of the 5th International Workshop on Hardware/Software Codesign*, pág.s 43–48, Março 1997.

[VG92] Frank Vahid e Daniel Gajski. Specification Partitioning for System Design. Em *Proceedings of the 29th Design Automation Conference*, 1992. Anaheim, Germany.

[VG95a] Frank Vahid e Daniel Gajski. Clustering for Improved System-Level Functional Partitioning. Em *Proceedings of the 8th International Symposium on System Synthesis (ISSS)*, 1995.

[VG95b] Frank Vahid e Daniel Gajski. Incremental Hardware Estimation during Hardware/Software Functional Partitioning. *IEEE Transactions on Very Large Scale Integration Systems*, 3(3):459–464, Setembro 1995.

[VG95c] Frank Vahid e Daniel Gajski. SLIF: A Specification-Level Intermediate Format for System Design. Em *Proceedings of the European Design and Test Conference (EDTC)*, pág.s 185–189, 1995.

[VGG94] Frank Vahid, Jie Gong e Daniel Gajski. A Binary-Constraint Search Algorithm for Minimizing Hardware during Hardware/Software Partitioning. Em *Proceedings of the European Design Automation Conference (EURO-DAC)*, 1994.

[VL96] Frank Vahid e Thuy Dm Le. Towards a Model for Hardware and Software Functional Partitioning. Em *Proceedings of the 4th International Workshop on Hardware/Software Codesign*, pág.s 116–123, Março 1996.

[VL97] Frank Vahid e Thuy Dm Le. Extending the Kernighan/Lin Heuristic for Hardware and Software Functional Partitioning. *Design Automation for Embedded Systems*, 2(2):237–261, 1997.

[WC97] J. Wilberg e R. Camposano. VLIW Processor Codesign for Video Processing. *Design Automation for Embedded Systems*, 2(1):79–119, 1997.

[XH96] Z. Xu e K. Hwang. Modelling Communication Overhead: MPI and MPL Performance on the IBM SP2. *IEEE Parallel and Distributed Technology*, (Spring):9–23, 1996.

[Xil97] Xilinx. Gate Count Capacity Metrics for FPGAs, Fevereiro 1997. Application note 059. `http://www.xilinx.com/apps/xapp.htm#xapp059`.

[Xil98] Xilinx. *The Programmable Logic Data Book*. Xilinx, 1998.

[Zav84] P. Zave. The Operational versus the Conventional Approach to Software Development. *Communications of the ACM*, 27(2):104–118, Fevereiro 1984.

[ZDG97] J. Zhu, R. Dömer e D. Gajski. Syntax and Semantics of the SpecC Language. Em *Proceedings of the Synthesis and System Integration of Mixed Technologies Conference*, 1997.

Índice