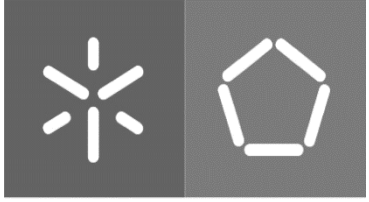
Universidade do Minho
Escola de Engenharia

André Antunes Oliveira

Multicore Embedded Virtualization Architecture Exploiting ARM TrustZone

Guimarães, May of 2017



Universidade do Minho
Escola de Engenharia

André Antunes Oliveira

Multicore Embedded Virtualization Architecture Exploiting ARM TrustZone

Dissertação de Mestrado em Engenharia Eletrónica Industrial e Computadores

Trabalho efectuado sob a orientação do
Professor Doutor Adriano Tavares
Professor Doutor Sandro Pinto

Guimarães, May of 2017

Declaração do Autor

Nome: André Antunes Oliveira
Correio Eletrónico: a65319@alunos.uminho.pt
Cartão de Cidadão: 14243023

Titulo da dissertação: Multicore Embedded Virtualization Architecture Exploiting ARM TrustZone
Ano de conclusão: 2017
Orientador: Professor Doutor Adriano Tavares

Designação do Mestrado: Ciclo de Estudos Integrados Conducentes ao Grau de Mestre em Engenharia Eletrónica Industrial e Computadores
Área de Especialização: Sistemas Embebidos
Escola de Engenharia
Departamento de Eletrónica Industrial

Universidade do Minho, 27/04/2017

Assinatura: André Antunes Oliveira

Agradecimentos

Faço uso desta secção da minha dissertação para agradecer a todos os que nela tiveram influência, direta ou indiretamente. Assim como aos que tiveram influência no meu percurso académico que culminou nesta dissertação.

Quero agradecer a todos os meus orientadores, pois sem eles não conseguiria terminar esta dissertação, pelo menos com o mesmo nível.

Ao meu co-orientador, o Professor [e dentro de dias] Doutor Sandro Pinto, queria agradecer a orientação, profissionalismo e ética de trabalho, sempre incontestáveis, fazendo tudo para encontrar tempo para me ajudar. Pelos conhecimentos e principalmente metodologia que me transmitiste, são incalculáveis. A minha vontade de querer sempre fazer bem e fazer melhor foi me transmitida por ti. Além da orientação, nunca hei de esquecer a amizade, o companheirismo, a hospitalidade, as brincadeiras e tudo o mais nas horas em que as mesmas podiam ser feitas. Hás de ser sempre uma pessoa que marcará a minha vida como uma inspiração, um exemplo a seguir, e como uma pessoa extraordinária em todos os aspetos. Todo o meu trabalho nesta dissertação e a minha evolução nos últimos 3 anos, devo-o a ti.

Ao meu orientador, orientador do meu co-orientador, Professor Doutor Adriano Tavares, por me encaminhar na direção certa, ainda que esta parecesse inalcançável, querendo sempre que trabalhássemos para nós e por nós, para que o nosso nome como engenheiro fosse merecido. Por estar sempre atento às necessidades de cada um de nós apesar de todo o seu trabalho, alunos e projetos sobre a sua orientação.

Um especial obrigado aos meus dois orientadores por me darem a oportunidade de evoluir como pessoa e como futuro engenheiro. A sua colaboração para me orientarem no caminho certo foi certamente o melhor que alguma vez poderia pedir. Agradeço ainda a oportunidade concedida, oportunidade de uma vida, de poder realizar Erasmus na Tailândia, onde a nível pessoal e profissional considero que tive uma evolução brutal.

Aos meus amigos Eduardo Mendes e João "Jony" Silva por todo o apoio demonstrado ao longo destes 6 anos (já são 12 para o Eduardo). Ao Eduardo por conseguir decifrar ou pelo menos tentar o meu pensamento emaranhado e muitas vezes conseguir que este fizesse sentido, até para mim. Pela amizade que muito

aprecio e por todos os conselhos que me deste, que nunca deixes de os dar. Ao Jony pela companhia naquele laboratório até ao nascer do dia seguinte, por me ouvires quando precisava e principalmente pela amizade e companheirismo que nunca hei de esquecer.

À minha namorada, Ana Guimarães, pela paciência que teve de me aturar enquanto trabalhava na dissertação. Por esperar por mim 6 meses, por me aturar quando estava mal disposto ou até com a cabeça noutro lado, pela companhia, pelo apoio incondicional e por no fim de tudo ainda ter a disponibilidade de me ajudar. Sabes que te amo!

Quero agradecer aos meus pais e aos meus irmãos por sempre me apoiarem independentemente do desafio, dificuldade ou problema. Ainda que não tivesse dado o devido valor, sinto que o apoio incondicional que eles me dão todos os dias foi fundamental para que pudesse chegar onde cheguei.

O meu último agradecimento vai para todos os meus colegas de laboratório que sempre se mostraram disponíveis para me ajudar e para me ouvir, quer numa fase inicial quer na fase final da dissertação, ao Carlos, ao Rapha, ao Monte e ao Filipe o meu obrigado. Um especial obrigado ao tMR. Gomes, pelo companheirismo, pelas piadas, pelas viagens e principalmente pelo desbloqueio de que precisava naquela hora em Zhuhai. Outro agradecimento especial não pode deixar de ir para o Jorge Pereira, sempre disponível para me co-orientar nas alturas que necessitava, obrigado pelo divertimento e amizade que me proporcionaste, nunca o esquecerei.

A todas os meus amigos que mesmo indiretamente me ajudaram nesta dissertação e percurso académico

O meu sincero Obrigado!

Resumo

O mercado e a própria aplicabilidade de sistemas embebidos têm-se expandido exponencialmente nos últimos anos, levando a uma crescente complexidade e sofisticação dos mesmos. A estes é agora cada vez mais exigido que integrem características próprias de diferentes classes de sistemas operativos – o cariz de tempo-real dos sistemas operativos de Tempo-Real (RTOS) e as interfaces gráficas dos sistemas operativos de Propósito Geral (GPOS). Uma das soluções que permite a coexistência de ambientes heterogéneos numa mesma plataforma de hardware e que garante, ao mesmo tempo, o isolamento dos requisitos de tempo-real face às interferências introduzidas pelas características de propósito geral é a utilização da tecnologia de virtualização. Existe um interesse generalizado por parte da indústria e academia em investigar soluções de virtualização assistidas por hardware uma vez que estas apresentam vantagens ao nível de desempenho e esforço de engenharia quando comparadas com as técnicas e soluções tradicionais.

A crescente necessidade de integração de um maior número de funcionalidades e complexidade nos sistemas embebidos atuais tem sido acompanhada por melhorias, na sua quase totalidade correspondentes, de performance por parte das plataformas *single-core*. No entanto esse crescimento está a tornar-se gradualmente insuficiente, levando em muitos casos a consumos energéticos exagerados por parte destas plataformas devido ao aumento insustentável da performance. A única solução viável para aumentar a performance sem comprometer o consumo é a migração para plataformas *multicore*.

Neste contexto a presente dissertação propõe a expansão de uma *framework* de virtualização assistida por hardware numa configuração *single-core* para uma configuração *multicore*. A tecnologia ARM TrustZone é explorada e utilizada pela *framework* já existente como uma extensão de virtualização do próprio processador, garantindo a execução simultânea de um GPOS e de um RTOS. Esta dissertação tem por objetivo implementar uma configuração *Asymetric MultiProcessing* (AMP) numa abordagem direta: dual *guest*, dual core. Será também explorada a implementação de um mecanismo de comunicação inter-partição por forma a potencializar as características das partições integrantes na *framework*.

Palavras Chave: Sistemas Embebidos, Virtualização, *Multicore*, Comunicação e ARM TrustZone.

Abstract

The embedded systems' market and its own applicability has expanded exponentially these last few years, leading to a growth in their complexity and sophistication. They have been increasingly demanded to integrate features of different operating systems classes – the real-time requirements of Real Time Operating Systems (RTOS) and the graphical interfaces of General Purpose Operating Systems (GPOS). One of the solutions which allows the coexistence of heterogenous environments in a same hardware platform and at the same time enforces the isolation of the real-time requirements against the interferences introduced by the general-purpose features is the use of virtualization technology. There is a general interest by both the industry and the scientific community to explore the hardware assisted virtualization solutions, since they present a better performance level and engineering effort when compared with traditional solutions.

The growing need for integration of a multiple number of features as well as complexity levels in embedded systems has been followed by improvements in performance rates, almost correspondently, by single-core platforms. Nevertheless, these improvements are becoming gradually insufficient, leading in many cases to an exaggerated energy consumption by those platforms due to the unsustainable need for performance levels. The only viable solution to the aforementioned problem without compromising the energy consumption is the migration to multicore platforms.

In this context, the present dissertation purposes the expansion of a single-core hardware assisted virtualization framework to a multicore configuration. The existent framework explores the use of the technology ARM TrustZone as virtualization extensions of the processor, thus granting the simultaneous execution of a RTOS and a GPOS. This dissertation has as its main goal the implementation of an Asymmetric MultiProcessing configuration in a straightforward approach: dual guest, dual core. Additionally the implementation of inter-partition communication mechanisms will be explored in order to potentiate the capabilities of the integrating parts of the framework.

Keywords: Embedded Systems, Virtualization, Multicore, Communication and ARM TrustZone.

Índice

Lista de Figuras

Lista de Tabelas

Lista de Listagens

Introdução

1.1 Contextualização

Os sistemas embebidos têm vindo a tornar-se parte integrante do quotidiano das sociedades modernas, tendo agora uma maior abrangência em termos de mercado do que há uns anos atrás, nomeadamente mercados como sistemas de transporte, área medicinal e até mesmo dispositivos eletrónicos de consumo. Este aumento da área de aplicabilidade de sistemas embebidos levou consequentemente a um aumento de desafios criados pelas exigências por parte do mercado, havendo uma necessidade de integração de diferentes funcionalidades num mesmo sistema de recursos limitados concluindo num aumento de complexidade dos mesmos.

De uma forma geral, todos os sistemas embebidos apresentam as seguintes métricas: performance, tamanho e peso, time-to-market, consumo energético e lista de materiais (BOM – *bills of materials*). Além destas métricas, que eventualmente se transformarão em requisitos, existem também certas restrições das quais se destacam o cariz de tempo-real, a fiabilidade e/ou a segurança em relação à resistência a ataques externos, que necessitam de ser considerados pelos projetistas dos sistemas embebidos.

Existe um amplo número de soluções capazes de mitigar os problemas acima referidos e até mesmo de auxiliar o cumprimento dos requisitos impostos. Soluções como a utilização de FPGAs, utilização de tecnologia *multicore* e técnicas de virtualização estão entre as tecnologias *state-of-the-art* atualmente utilizadas na área de sistemas embebidos.

De entre as soluções supracitadas a utilização de técnicas de virtualização está entre as soluções que mais se tem destacado ultimamente. Apesar de já ser utilizada recorrentemente em sistemas de propósito geral, esta solução apenas mais recentemente tem sido utilizada em sistemas embebidos [4]. Recorrer a técnicas de virtualização assistidas por software, ou em alguns casos assistidas por hardware, tem sido uma constante para implementação de múltiplas classes de sistemas operativos num mesmo dispositivo hardware, permitindo assim otimizar o BOM, sendo que muitas vezes pode afetar métricas como a performance e *time-to-market*, especialmente quando implementada em software.

Recorrendo ao suporte oferecido por extensões em hardware do próprio processador, estas soluções de virtualização conseguem mitigar as limitações de quebra de performance e *time-to-marker* acima referidas. Comparativamente às suas vantagens inerentes este género de solução tem tido uma aderência relativamente escassa. Esta escassez de soluções de virtualização assistidas por hardware justifica-se pelo preço, sendo que os processadores nas quais se encontram este tipo de extensões de hardware são tipicamente processadores de gama alta e até de cariz não determinístico e por vezes pouco apreciados.

É nesta conjuntura, de processadores de gama média-baixa para os quais existe uma necessidade de técnicas de virtualização e onde não existe um suporte próprio em hardware para a sua implementação, que surge a tecnologia ARM® TrustZone®. Desenvolvida em 2003 pela ARM, este conjunto de extensões de segurança em hardware tem o propósito de isolar partes críticas do sistema de possíveis fontes de vulnerabilidade nos sistemas operativos. Apesar desta conceção original, a tecnologia tem sido explorada para a implementação de soluções de virtualização assistidas por hardware, aparecendo diversos projetos como [5, 6, 7, 8, 9, 10, 11, 12, 13].

Dos projetos supracitados destaca-se a *framework* desenvolvida *in-house* pelo ESRG [5]. A *framework* explora as extensões de segurança em hardware ARM TrustZone para implementar um hypervisor assistido por hardware. A *framework* apresenta uma arquitetura de virtualização que possibilita a integração de um RTOS (Real Time Operating System) e de um GPOS (General Purpose Operating System) numa mesma plataforma.

1.2 Motivação e Objetivos

As tradicionais técnicas de virtualização, isto é, as técnicas de virtualização assistidas exclusivamente por software, apesar do seu recente ímpeto e crescente popularidade em sistemas embebidos [4], representam sempre um *trade-off* entre a flexibilidade e a performance. A virtualização por emulação tem um custo no desempenho do sistema, enquanto que a para-virtualização representa um grande esforço de engenharia. Neste contexto, várias entidades introduziram no hardware dos seus SoCs extensões de virtualização.

A tecnologia ARM TrustZone tem ganho particular importância pela sua predominância em processadores do domínio embebido quando comparada com as extensões de virtualização, sendo inclusivamente vistas como a única solução para processadores ARM privados das ARM Virtualization Extension (VE). Apesar de

se tratarem de extensões de hardware direcionadas para segurança, as mesmas têm sido exploradas para a implementação de hypervisores *dual-OS* assistidos por hardware, como é exemplo a *framework* em expansão [5].

Contudo, a solução [5], por prioritizar as características de tempo-real do sistema, apresenta uma limitação, destacada por Ngabonziza *et al.* em [14]. Quando o RTOS tem um fator de carga demasiado elevado, pode, como consequência, nunca ceder o controlo do processador. Nesse caso o GPOS incorre no fenómeno de *starvation*, não tendo tempo de execução no processador.

A migração da *framework* para uma configuração *multicore* é vista como a solução para o fenómeno supracitado. A paralelização do processamento da *framework* permitiria solucionar a influência temporal do RTOS sobre o GPOS, eliminando por completo a sua limitação conhecida.

Neste sentido surge a presente dissertação que tem como objetivo expandir a *framework* para uma configuração *multicore*. Adicionalmente, esta dissertação pretende munir a *framework* de mecanismos de comunicação, tornando-a no geral, numa solução robusta e flexível, obtendo assim a versatilidade para ser utilizada numa ampla gama de problemas existentes no domínio embebido.

Abaixo estão sumarizados os objetivos individuais que esta dissertação pretende cumprir:

- Expansão da *framework* para uma configuração *multicore* AMP, dual-core, dual-OS;
- Implementação de um mecanismo de comunicação versátil e escalável, suportado em ambas as configurações *single-core* e *multicore*;
- Avaliação da configuração *multicore* implementada comparativamente com a *framework* original. Avaliação realizada em termos de *footprint* de memória, métricas *real-time* e desempenho global dos *guests* inclusos;
- Caracterização do mecanismo de comunicação implementado. Este deverá ser avaliado em termos de latência, *throughput* e impacto geral no sistema para ambas as configurações.

1.3 Organização da Dissertação

A presente dissertação está dividida em 6 capítulos diferentes. No presente capítulo pretendeu-se contextualizar o problema abordado assim como revelar as

motivações para a implementação do mesmo, tendo sido posteriormente revelados os objetivos propostos na realização da dissertação.

No capítulo 2 serão abordadas as tecnologias utilizadas no decorrer da realização da dissertação, com destaque para as tecnologias de virtualização, *multicore* e OpenAMP. Estas são reveladas como a solução para as tendências e desafios dos sistemas embebidos. Será feita uma análise comparativa entre as tecnologias supramencionadas e as demais analisadas como solução dos diferentes problemas com que se deparam os sistemas embebidos.

O capítulo 3 é responsável por descrever detalhadamente a arquitetura ARM, com um foco na arquitetura do processador presente na plataforma utilizada no desenvolvimento da dissertação, o processador Cortex-A9. Será abordada em detalhe a tecnologia ARM TrustZone e como a mesma se correlaciona com a *framework* de virtualização. Posteriormente serão referenciadas as ferramentas utilizadas no desenvolvimento da dissertação. Por último é descrita a arquitetura inicial da *framework* de desenvolvimento, bem como a arquitetura especificada pelo OpenAMP.

O capítulo 4 foca-se nos detalhes de implementação da expansão da *framework* para uma configuração *multicore* e ainda no suporte integrado para mecanismos de comunicação inter-partição. Inicialmente será feita uma análise meticulosa estrutural da configuração inicial da *framework* para que seja mais fácil para o leitor entender as alterações realizadas e respetivos impactos na mesma.

Os resultados obtidos da avaliação global das soluções propostas são apresentados no capítulo 5. Os resultados debruçar-se-ão sobre o impacto das diferentes configurações do LTZVisor nos sistemas operativos *guests* e nas suas principais características comparativamente com as suas versões nativas. Será ainda realizada uma caracterização da solução de comunicação implementada.

Finalmente, o capítulo 6 apresentará as conclusões retiradas do trabalho exposto, as melhorias e limitações introduzidas na *framework*. Serão feitas recomendações de possíveis melhorias ou adições de funcionalidades ao sistema como trabalho futuro a esta dissertação.

Literatura e Estado da Arte

2.1 Sistemas Embebidos

Os sistemas embebidos estão omnipresentes, estando neste momento altamente disseminados pela sociedade moderna. Os domínios dos sistemas embebidos são realmente extensos: desde sistemas de entretenimento nos quais é exigido cada vez mais uma melhor performance, como nos dispositivos eletrónicos de consumo (*e.g.* telemóveis, televisões, etc.), a sistemas de suporte vital nos quais é exigida a máxima fiabilidade, sendo estes bastante comuns em sistemas de transporte (*e.g.,* aviação, automóvel, espaço, etc.) ou na área da saúde.

A sua evolução tecnológica e consequentemente a constante mudança de funcionalidades e de hardware incluso, assim como a mudança das características requeridas nos mesmos, tornam difícil encontrar uma definição que reúna consenso. No entanto, apesar desta mudança de paradigma constante, existe algo comum a todas as gerações de sistemas embebidos. A sua especificidade e respetiva utilização. Estes são sistemas de propósito específico incorporados num sistema maior (geralmente incluindo hardware adicional e/ou partes mecânicas) com a finalidade de o controlar, monitorar ou acrescentar-lhe funcionalidades [15].

Os sistemas embebidos possuem geralmente limitações ao nível de recursos presentes no seu hardware, realçando-se a escassez de memória e a *performance*. Aliado a estas limitações ou mesmo pela própria aplicabilidade/finalidade dos sistemas, existem determinados requisitos que têm ditado a tendência na sua conceção e no seu aprimoramento: a) melhor performance, b) baixo consumo energético, c) tamanho e peso reduzido, d) rápido *time-to-market* e e) baixo *bills of materials* (BOM). Simultaneamente existem outros requisitos próprios de sistemas embebidos de tempo-real ou de finalidades específicas dos próprios sistemas que necessitam também de ser levados em consideração pelos projetistas de sistemas embebidos. Entre eles destacam-se as características de tempo-real, a fiabilidade e a segurança. Estes requisitos têm ganho particular importância devido ao advento da interconectividade em sistemas embebidos, proporcionada pela era da *Internet of Things*, bem como a junção de diferentes classes de sistemas operativos numa mesma plataforma.

Para combater este conjunto de desafios ambas a comunidade científica e indústria têm-se virado para soluções baseadas em tecnologias *state-of-the-art*: *Multicore*, FPGA e técnicas de virtualização. Vanderleest et al. em [16], refere estas soluções como a base para o futuro da tecnologia aviônica onde a maioria dos requisitos supramencionados se aplicam. A presente dissertação invoca o uso da tecnologia *Multicore* e extensões de hardware para utilização de técnicas de virtualização de forma a expandir a framework [5] no sentido de combater os requisitos impostos frequentemente a designers de sistemas embebidos.

2.2 Virtualização

A definição de virtualização em termos computacionais é a criação de uma abstração integral ou parcial em software do sistema computacional hardware. A forma mais comum de virtualização é a de criação de uma abstração do processador inteiro, criando assim um número variável de processadores virtuais (VCPU). O principal objetivo desta abstração é a possibilidade de alojar simultaneamente num mesmo processador físico (de forma *time-sliced*) mais do que um sistema operativo (SO), isto é, permitindo que múltiplos sistemas operativos isolados espacial e temporalmente co-habitem a mesma plataforma.

A camada de software que possibilita a abstração dos recursos do hardware e que é responsável pelo cumprimento do isolamento temporal e espacial é normalmente denominada de hypervisor ou Virtual Machine Monitor (VMM). Para que o isolamento seja garantido, esta camada deverá ser executada num modo com um nível de privilégio acima dos sistemas operativos virtualizados, também denominados de Virtual Machines (VM) [12]. Usualmente ao modo de privilégio superior do sistema está associado o controlo exclusivo de certos periféricos que são utilizados para a aplicação do isolamento temporal e espacial, como temporizadores e controladores de memória (*e.g.* MMU, MPU, entre outros). A atribuição exclusiva do modo superior de privilégio ao hypervisor é considerado um dos maiores desafios na implementação de técnicas de virtualização em sistemas embebidos, nos quais os processadores possuem tipicamente apenas dois modos de privilégio.

O hypervisor deverá ser responsável na maioria dos casos, à semelhança de um sistema operativo, por realizar o escalonamento dos sistemas operativos. Assim como um sistema operativo deverá também realizar o *context switch* e alterações necessárias, se o sistema em que o hypervisor está inserido assim o exigir, como mudanças de estados em recursos partilhados. O hypervisor deverá ainda fortalecer e aplicar o isolamento espacial e temporal entre as diferentes partições. Os

hypervisores podem ser divididos em dois tipos, de acordo com a metodologia de como são implementados:

- **Tipo 1:** O hypervisor, como observado na Figura 2.1 (denominado de *Camada de Virtualização*), é executado de forma nativa/*baremetal*, isto é, diretamente no hardware. Este tipo de hypervisor é o mais comum na área de sistemas embebidos;
- **Tipo 2:** O hypervisor executa dentro do contexto de um sistema operativo, sendo tratado pelo SO *host* como um processo normal. Estes hypervisores são frequentemente denominados de *hosted hypervisors*. Dentro deste tipo de hypervisores destacam-se as distribuições comerciais Virtualbox e VMWare.

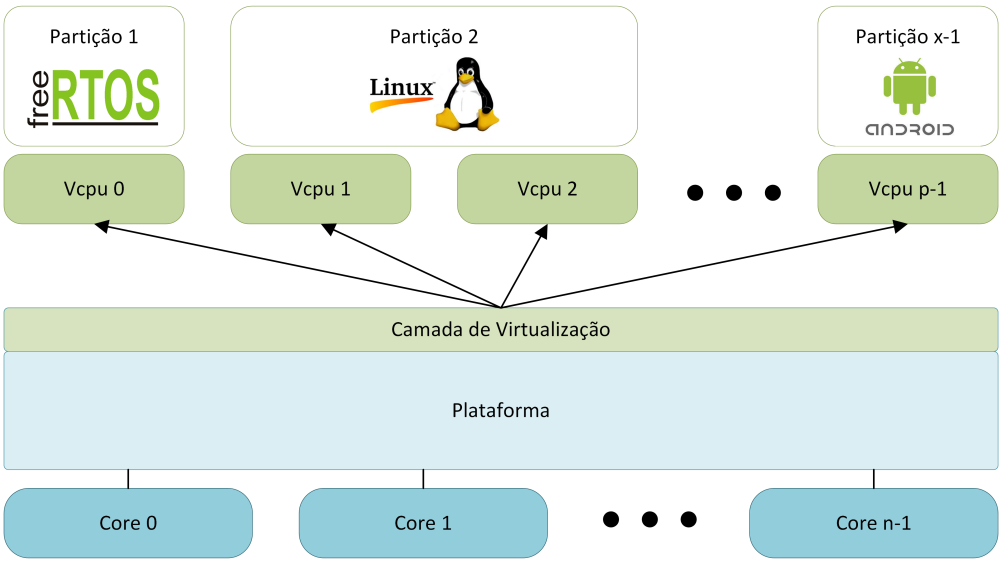


Figura 2.1: Virtualização *multicore* de uma plataforma fisíca

A Figura 2.1 retrata uma solução de virtualização bastante usual em sistemas embebidos onde é utilizado um hypervisor de tipo 1, uma solução existente em *frameworks* como a presente no projecto MultiPARTES [17] ou Xen [18]. A Figura 2.1 representa uma técnica de virtualização aplicável a uma plataforma física com número variável de cores. A virtualização neste caso cria um número variável de VCPUs, uma representação virtual de um core do processador. A cada core pode estar associado vários VCPUs, sendo nesse caso cada VCPU uma representação virtual de um *time-slice* desse mesmo core. Na figura pode-se ver representado várias partições, podendo cada uma delas conter um ou mais VCPUs e estar associada um *guest*/SO. Esses VCPUs podem pertencer a cores diferentes e correr em simultâneo observando-se uma partição multicore, como podem ser representações de um mesmo core e correr em alternância originando uma partição

single-core com uma frequência de execução mais elevada no core representado pelos VCPUs pertencentes.

Existem várias soluções de virtualização normalmente divididas em dois tipos de acordo com o suporte usado para a sua implementação: as soluções assistidas unicamente por software e as soluções que utilizam suporte em hardware. Neste subcapítulo serão expostas algumas das vantagens e desvantagens dos dois tipos de soluções sendo feita uma comparação entre ambas.

2.2.1 Virtualização Assistida por *Software*

A virtualização assistida por software apresenta inúmeros desafios bem conhecidos, cujas soluções apresentam sempre vantagens e desvantagens obrigando sempre a um *trade-off* das características inerentes aos sistemas embebidos.

O principal desafio consiste no isolamento espacial e temporal das VMs participantes na virtualização. A chave para a resolução deste problema é, como supramencionado, a de executar o VMM de forma exclusiva nos modos de execução do processador de maior nível de privilégio, tendo assim acesso exclusivo aos componentes que aplicam o isolamento, levando a que qualquer possível quebra ou ataque ao isolamento por parte das VMs seja filtrada obrigatoriamente pelo hypervisor. Não obstante, esta solução apresenta uma óbvia desvantagem: as partições perdem um ou mais modos de execução (os que sejam definidos como de maior privilégio) para o hypervisor, acrescentando por si só um novo desafio.

Tabela 2.1: Modos de processamento e respetivo privilégio em processadores ARM

Modo de Execução	Privilégio	
	User	Kernel
User	x	
System		x
Abort Exception		x
Undefined Exception		x
IRQ		x
FIQ		x
Supervisor		x

Tipicamente um processador opera em vários modos (*e.g.irq*, *user*, supervisor, *system*, entre outros), no entanto estes podem ser divididos em dois níveis de privilégio como observado na Tabela 2.1: *user* e *kernel*, também denominados de modo *user* e modo *kernel*. O modo *user* é caracterizado por conter as aplicações de utilizador, apresentando bastantes restrições de maneira a que uma falha de

uma destas aplicações não comprometa o sistema inteiro. Por outro lado, o modo *kernel*, que como indica o nome, é onde é executado o *kernel* do sistema operativo, usualmente garante ao código nele executado um nível de privilégio superior sem restrições, no qual é possível realizar as tarefas de maior importância e de maior impacto no sistema.

Concluindo, a impossibilidade de execução do sistema operativo *guest* no modo de execução de maior nível de privilégio do processador resume-se num desafio para o funcionamento correto do mesmo, que assim fica impedido de executar instruções exclusivas ao modo *kernel*. Das soluções encontradas para este desafio destacam-se a para-virtualização e a virtualização completa (*full-virtualization*):

(i) ***Full Virtualization***. Neste tipo de virtualização os sistemas operativos guests são integrados sem qualquer tipo de alteração, consequentemente traduzindo-se num rápido *time-to-market*. O facto de o guest não necessitar de qualquer alteração deve-se a uma tarefa executada pelo hypervisor denominada de *trap and emulate*. Esta tarefa tem como objetivo capturar qualquer instrução exclusiva ao modo privilégio no qual o hypervisor se encontra e posteriormente emula-la, pois ao *guest* não lhe é permitido executar nesses mesmos modos. Todavia, apesar de apresentar vantagens ao nível do *time-to-market* esta solução tem uma desvantagem muitas vezes indesejada ou até insustentável: uma degradação da performance devido ao *overhead* introduzido no sistema [12] levando muitas vezes a ter de se recorrer a um tipo diferente de virtualização.

(ii) ***Paravirtualização***. Para além da *full-virtualization* outro género de virtualização comum assistida por software é a paravirtualização, também denominada de *"paravirtualization"*. Neste tipo de virtualização, contrariamente à *full-virtualization*, os sistemas operativos guests são modificados de maneira a suportarem *hypercalls* – de maneira semelhante a uma *system call*, – isto é, o *guest* faz um pedido explícito ao hypervisor para que este execute a instrução exclusiva ao modo de privilégio no qual o próprio hypervisor se encontra. Estas alterações eliminam o *overhead* introduzido pela *full-virtualization* na captura deste género de instruções. No entanto, o esforço de engenharia e mesmo de manutenção relacionado com estas alterações aos sistemas operativos guests levam a um aumento do custo de engenharia do produto e a atrasos consideráveis no *time-to-market*. Estas desvantagens tornam a solução de virtualização pouco atrativa para projetos que requeiram um rápido *time-to-market* e uma solução *ready to use*.

Uma outra desvantagem comum à virtualização em software é a de tipicamente existirem periféricos que possuem a capacidade de ultrapassar o isolamento garantido pelo hypervisor, tais como o DMA (*direct memory access*) ou o GPU (*graphical processing unit*) [19]. Daí que seja necessário que estes periféricos sejam controlados pelo hypervisor. Essa necessidade traduz-se, para além de um esforço extra de engenharia, numa degradação da performance.

2.2.2 Virtualização Assistida por *Hardware*

Face ao gradual crescimento da sua popularidade mesmo com as conhecidas desvantagens adjacentes às soluções de virtualização assistidas por software, o desenvolvimento e utilização de técnicas de virtualização assistida por hardware ganhou um novo foco. Estas técnicas fazem uso de extensões em hardware integradas na própria arquitetura do processador para implementar soluções de virtualização sem a totalidade ou parte das desvantagens supracitadas.

As extensões caracterizam-se usualmente por adicionarem ao processador um novo nível de privilégio, superior aos existentes e conhecidos modo *user* e modo *kernel*, no qual será executado o hypervisor. Ao novo nível de privilégio adicionam-se outras primitivas em hardware capazes de mitigar um dos maiores desafios da virtualização: o isolamento espacial e temporal. Dependendo da própria extensão, estas primitivas podem incluir adições ou alterações aos componentes do próprio processador de maneira a produzir autênticos CPUs virtuais em hardware. Entre eles destacam-se os periféricos da plataforma, registos de propósito geral ou de coprocessadores, MMU, caches e a própria memória ou controlador de memória.

Dentro das extensões de hardware para a utilização de virtualização no domínio de sistemas embebidos destacam-se as ARM Virtualization Extensions (ARM VE) e as Intel Virtualization Technology (Intel VT).

2.2.2.1 Intel *Virtualization Technology*

Criada em 2005, a Intel Virtualization Technology (Intel VT) também denominada de Vanderpool numa fase inicial do seu desenvolvimento, é uma extensão de virtualização em hardware presente numa vasta gama de processadores da Intel. A tecnologia VT introduz nos processadores da Intel um novo modo de privilégio denominado *"root mode"* (mesma propóstio do modo *"hypervisor mode"* da figura Figura 2.2.b)), possibilitando assim que a virtualização seja implementada sem perdas significativas de performance ou sem que existam alterações obrigatórias aos sistemas operativos *guests*.

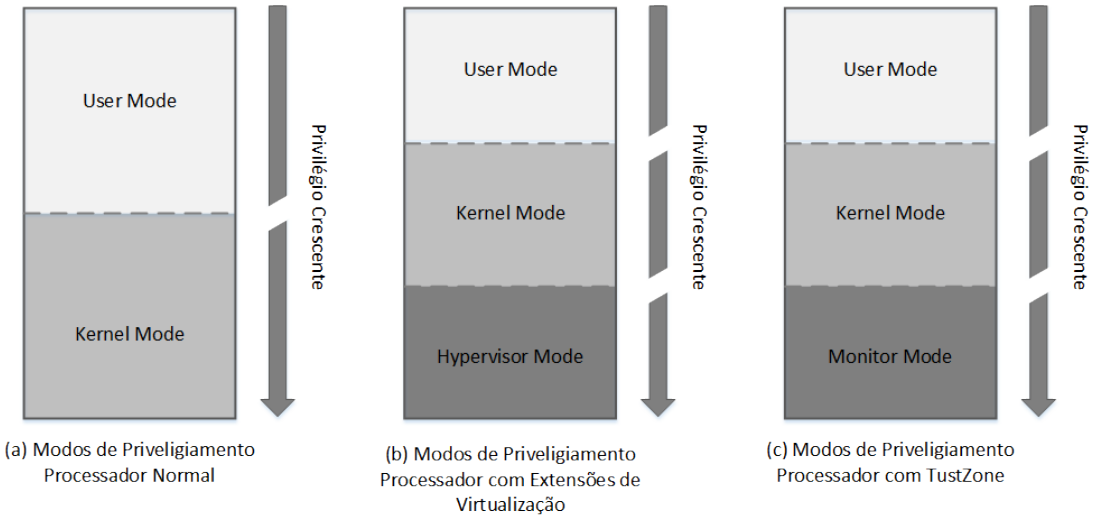


Figura 2.2: Modos de privilégio vistos em vários tipos de processadores ARM

Em relação a versões anteriores da mesma tecnologia, a versão x (VT-x) permite, para além de uma migração fácil de ambientes não virtualizados e até de gerações da mesma tecnologia anterior, a virtualização da *page table* em hardware (e respetivo context switch) assim como atribuições de periféricos específicos a um determinado sistema operativo guest.[1]

2.2.2.2 ARM *Virtualization Extensions*

Criadas em 2010 pela ARM, as extensões de virtualização ARM VE tiveram como objetivo adicionar o suporte para a implementação de técnicas de virtualização nas arquiteturas mais avançadas dos processadores ARM, uma vez que estas, devido a limitações do próprio design, apresentavam falhas que dificultavam a implementação de soluções de virtualização assitida por software.[2]

Estas extensões surgem depois da criação das extensões de segurança ARM TrustZone, estando por isso intrinsecamente ligadas às mesmas. A tecnologia ARM VE adiciona um modo de privilégio superior ao hypervisor denominado de *hypervisor mode*. De maneira idêntica à Intel VT-x a tecnologia ARM VE permite também a virtualização de periféricos em hardware assim como mecanismos para a virtualização da memória.

[1] https://www-ssl.intel.com/content/www/us/en/virtualization/virtualization-technology/intel-virtualization-technology.html

[2] https://genode.org/documentation/articles/arm_virtualization

2.2.2.3 ARM TrustZone Security Extensions

A tecnologia ARM TrustZone foi desenvolvida com o propósito de acrescentar características de segurança ao processador adicionando-lhe hardware adicional. Com esse objetivo o hardware adicionado divide o processador e alguns dos seus recursos em dois mundos virtuais conceptualmente chamados de mundo seguro e mundo não-seguro. O objetivo principal é o de criar um ambiente de execução seguro para o mundo virtual seguro, tendo por isso o primeiro mundo níveis de privilégio e acessos superiores ou únicos em relação ao outro mundo. A tecnologia TrustZone adiciona também ao processador um modo de execução de privilégio superior aos modos de privilégio *kernel*. Neste modo o processador consegue realizar exclusivamente a comutação de mundos virtuais em execução. Este modo possui a característica única de poder controlar os periféricos privilegiados e primitivas adicionadas pela tecnologia TrustZone independentemente do mundo em que se encontra, não necessitando de executar no mundo seguro para controlar e aplicar o isolamento entre os dois mundos virtuais.

Estas características adicionadas pelas extensões de segurança ARM TrustZone são bastante semelhantes a extensões de virtualização em certos aspetos, podendo por isso mesmo ser utilizada para esse efeito com um esforço minimizado quando comparado a soluções de virtualização assistidas por software. A utilização do modo monitor incluído pelas extensões para a implementação do hypervisor permite que os sistemas operativos *guests* possam executar no modo de privilégio *"kernel mode"*. Os *guests* podem assim usufruir das instruções exclusivas ao modo *kernel* sem que o isolamento seja comprometido (por parte do mundo não-seguro) e sem que sejam necessárias alterações aos mesmos (excetuando o caso de acessos a periféricos partilhados). Desta forma retira-se um dos maiores problemas intrinsecamente ligados à implementação de virtualização assistida por software, que seria a eliminação do modo kernel do ponto de vista dos *guests*.

Para além da vantagem supramencionada, a divisão do processador em dois mundos virtuais, torna propícia a implementação de uma configuração de virtualização denominada de Dual-OS. Nesta configuração, a cada mundo virtual estará associado um sistema operativo *guest*, estando o isolamento espacial e temporal garantido pelo hardware incluso pela tecnologia TrustZone e pelo próprio hypervisor. No entanto, para que esta premissa seja válida, o *guest* incluído no mundo virtual seguro terá de fazer parte da *Trusted Computing Base* (TCB), isto é, deverá ser considerado seguro e capaz de não interferir com os recursos do mundo não-seguro, uma vez que este é capaz de ultrapassar facilmente o isolamento imposto pelo hardware [12].

Assim como em extensões em hardware próprias para a implementação de virtualização, a divisão do processador criada pela tecnologia TrustZone insere também o banqueamento de grande parte dos registos do coprocessador 15 – esses registos não necessitam de ser salvaguardados – e permite também ainda a coexistência de dados separadamente de ambos os mundos/guests na cache – deixa de ser necessária a limpeza ou invalidação da mesma – possibilitando assim um *context switch* entre guests batante mais rápido do que o realizado numa arquitetura de virtualização assistida por software [19, 12, 9].

As extensões TrustZone permitem ainda o encaminhamento das diversas exceções (*abort exception*, IRQ, FIQ) para o modo monitor, podendo assim ser tratadas pelo hypervisor. Uma das vantagens associadas a esta característica da tecnologia é a possibilidade de garantir as características de tempo-real de um dos sistemas operativos *guests* fazendo o encaminhamento das interrupções FIQ diretamente para o mundo onde o mesmo se encontra [5].

Face aos avanços das extensões de virtualização, as soluções de virtualização sofreram enormes progressos, porém a sua empregabilidade ainda é escassa devido ao limitado número de processadores capazes de suportar estas extensões, quer pelo custo adicional, quer pela forma como foram originalmente desenhados ou até mesmo pelas suas características de recursos escassos. No entanto com o anúncio da ARM de estender a tecnologia ARM TrustZone a todos processadores da série Cortex-A e com planos para expandir o seu uso aos processadores da gama Cortex-M[3], a exploração desta tecnologia como extensão de virtualização ganhou destaque, tendo surgido projetos como o SafeG criado pela TOPPERS [13], o ViMoExpress [9], o SierraTEE criado pela Sierraware[4], o projeto SASP [20], assim como própria *framework* em expansão nesta dissertação [5] ou até projetos como [12, 11].

2.3 Multicore

A crescente necessidade de integração de um maior número de funcionalidades e complexidade nos sistemas embebidos atuais tem sido acompanhada por melhorias, na sua quase totalidade correspondentes, de performance por parte das plataformas *single-core*. No entanto esse crescimento está a tornar-se gradualmente insuficiente ou até incomportável, levando em muitos casos a consumos

[3]https://www.arm.com/products/security-on-arm/trustzone
[4]http://www.sierraware.com

energéticos exagerados por parte destas plataformas devido ao aumento insustentável da performance. A única solução viável para aumentar a performance sem comprometer o consumo de energia é a migração para plataformas *multicore*.

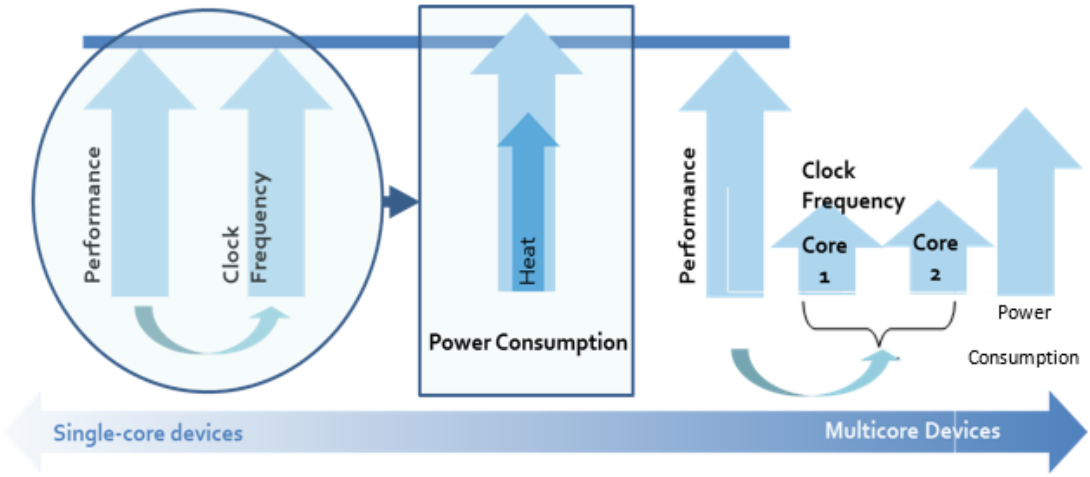


Figura 2.3: Compromisso entre performance e consumo energético

Através da Figura 2.3 é possível observar uma ilustração do compromisso entre performance e consumo energético. Tipicamente, num sistema *single-core* a forma de obter uma performance superior é através de um aumento da frequência de relógio do processador, o qual se traduzirá num aumento de consumo de energia e de potência dissipada pelo processador. Esse aumento de energia consumida e potência dissipada pode muitas vezes ser insustentável ou até mesmo prejudicial no mundo de sistemas embebidos. Por essa razão, recorre-se à migração para um sistema *multicore*, no qual o aumento de performance é realizado recorrendo ao processamento paralelizado através da utilização de mais de um core. Existindo um aumento da frequência de relógio dos processadores pertencentes ao sistema, este será repartido pelos CPUs existentes evitando assim os problemas supramencionados. A migração para sistemas *multicore* é vista na área de sistemas embebidos como inevitável.

Para além do problema do consumo energético associado ao aumento da frequência do processador surge ainda o problema da própria frequência de relógio do processador que pode muitas vezes ser prejudicial no sistema embebidos em que se insere. Um exemplo concreto desta situação será no domínio espacial onde a frequência do processador é limitada devido à presença de radiação, levando por isso a que exista uma maior necessidade de migração para plataformas *multicore* [17].

Esta tecnologia apesar de relativamente recente tem sido amplamente adoptada pela indústria, produtores e comunidade científica. É uma tecnologia que permite adicionar flexibilidade e escalabilidade ao *design* de um sistema embebido. Permite que a performance seja aumentada pelo aumento de cores participantes no sistema como permite ao mesmo tempo diferentes configurações (confinando diferentes camadas de software a diferentes cores).

A tecnologia *multicore*, também usualmente designada de processamento paralelo, apenas pode ser executada em plataformas que o permitam (plataformas *multicore*). Estas plataformas contém mais de um core no seu *System-on-Chip* (SoC). O processamento executado em cada core é feito independentemente, sendo que os cores processam literalmente em simultâneo, levando muitas vezes a problemas de concorrência.

Os cores revelam-se normalmente independentes uns dos outros, excetuando a partilha de certos recursos, como a memória RAM principal e certos periféricos de I/O. Existe ainda um nível de cache (tipicamente L2) que é também partilhado entre os cores incluídos na plataforma, sendo que as caches mais importantes (L1) são particulares a cada core. Nestes níveis de cache associados a cada core pode usualmente ocorrer um problema de coerência: as caches de diferentes cores podem conter valores diferentes de um mesmo endereço da memória principal. Nesta situação recorre-se usualmente a soluções de hardware como a SCU nos processadores da ARM ou recorrendo a soluções baseadas em software responsáveis por manter a coerência transversalmente entre as caches de cada core.

As plataformas *multicore* podem-se dividir em dois grupos: i) plataformas homogéneas e ii) plataformas heterogéneas. O primeiro grupo caracteriza-se por todos os cores pertencentes à plataforma serem idênticos, enquanto que no segundo grupo os cores poderem ser diferentes e até mesmo de diferentes famílias de processadores. O SoC Zynq-7000 é um exemplo das plataformas homogéneas contendo dois processadores idênticos ARM Cortex-A9. Por sua vez o SoC Zynq UltraScale, um exemplo de plataformas heterogéneas, contém até quatro processadores ARM Cortex-A53 e dois processadores ARM Cortex-R5.

O processamento paralelo e recursos partilhados são desafios importantes na configuração de um sistema embebido que utilize uma plataforma *multicore* e são muitas vezes as fontes de problemas de migração de aplicações *single-core*. Estes problemas são geralmente mitigados recorrendo a mecanismos de sincronização e comunicação, por vias da implementação de um sistema virtualizado ou recorrendo a certas características do hardware incluso na plataforma *multicore*. Existem diferentes tipos de configuração *multicore* sendo que cada um contém os seus

próprios desafios, revelados em parte neste subcapítulo.

2.3.1 Symetric Multiprocessing

A configuração SMP (Symetric Multiprocessing) é apenas vista em plataformas homogéneas e caracteriza-se pelo papel de cada core ser determinado dinamicamente por um mesmo sistema operativo, isto é, existe apenas um sistema operativo que comanda os cores da plataforma pertencentes à configuração SMP.

Qualquer aplicação, processo ou tarefa, que não possua afinidade a um dos cores, pode ser executado em qualquer core, sendo o trabalho do *scheduler* do sistema operativo de migrar essas mesmas tarefas para os diferentes cores. O objetivo principal será o de atingir uma carga de trabalho ótima transversalmente a todos os cores, através da migração das diferentes tarefas. Essa migração, no entanto, não deverá ser muito frequente pois a mudança de tarefas entre cores, poderá afetar a performance da Cache.

Na Figura 2.4 observa-se a arquitetura de um processador *multicore* com uma configuração SMP, na qual um sistema operativo comanda a totalidade de processadores presentes na plataforma. Esta configuração *multicore* é a que apresenta o menor *footprint* de memória pois os diferentes cores correm a mesma imagem do sistema operativo. Todos os cores têm a mesma visão da memória e do hardware partilhado. Usualmente a um dos cores será atribuída a responsabilidade por realizar o boot do sistema operativo assim como garantir o arranque dos restantes cores e ainda eventualmente deverá 'comandar' o acesso a periféricos *I/O* partilhados.

Num sistema operativo *single-core* é necessário recorrer a mecanismos de sincronismo para o correto funcionamento das tarefas que partilham os mesmos recursos, pois as mesmas executam em paralelo, *i.e.*, podem ser escalonadas intervaladamente. Num sistema operativo *multicore* SMP a situação é agravada pelo facto de a execução ser verdadeiramente paralela, sendo necessário a implementação de mecanismos de sincronismo próprios para *multicore* (por exemplo, *spinlock*) de maneira a evitar problemas de concorrência de tarefas.

Nesta configuração em que os cores têm acesso à mesma memória (*i.e.,* a memória principal será um partilhada entre os cores) é necessário recorrer a hardware próprio para manter a coerência entre os dados em Cache e na memória principal.

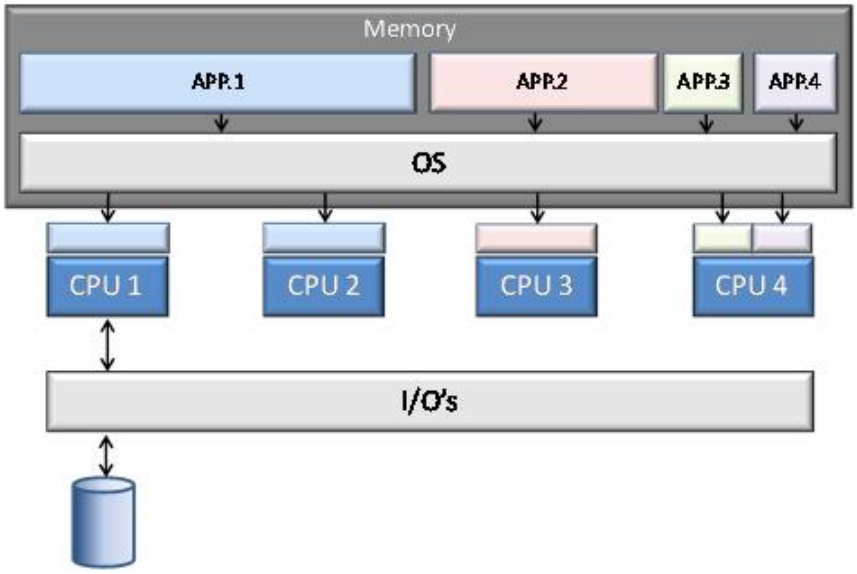


Figura 2.4: Symetric multiprocessing (SMP) num processador quadcore

2.3.2 Asymetric Multiprocessing

A configuração AMP (Asymetric Multiprocessing) define-se por tratar cada core de forma individual, isto é, cada core executa independente dos outros cores, correndo em cada uma sua versão individual de um SO. A configuração AMP pode ser considerada homogénea se cada um dos cores correr uma cópia individual do mesmo sistema operativo ou heterogénea se cada core tiver um sistema operativo diferente dos executados nos restantes cores.

Na Figura 2.5 pode-se observar a arquitetura de um processador *multicore* com configuração AMP. Esta configuração é caracterizada por aumentar bastante o *footprint* de memória, pois cada core terá uma cópia própria da sua versão do sistema operativo que executa.

Cada core pode ter uma visão diferente da memória disponível e do hardware partilhado. Não necessitam de mecanismos de coerência de dados, pois cada core teria a sua própria memória, Cache e MMU, salvo claro nas situações de recursos partilhados onde teriam de ser implementados mecanismos de sincronismo ou comunicação.

Tipicamente, os sistemas operativos não podem equilibrar a carga de trabalho efetuada por cada core (através de por exemplo migração de tarefas entre cores), pois tratam-se de cores independentes e com sistemas operativos diferentes, levando em certos casos a níveis de carga de trabalho bastante diferentes e por vezes comprometedoras da estratégia de *multicore* escolhida. Esta situação é facilmente ultrapassada recorrendo a mecanismos de comunicação RPC como vistos

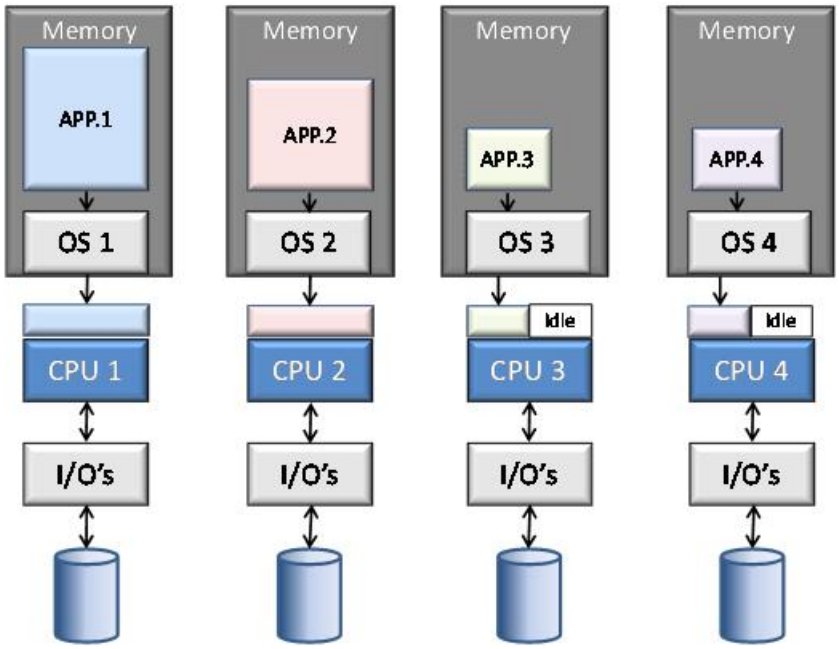


Figura 2.5: Asymetric multiprocessing (AMP) num processador quad-core

na arquitetura *multicore* em [2], referidos na Subsecção 2.4.

Do ponto de vista de cada core, o mesmo corre como se estivesse numa configuração *single-core*. Isto é, devido à natureza da configuração (cada core tem o seu OS), nenhum core sabe da existência dos outros (à excepção dos mecanismos de sincronização e comunicação). Esta particularidade permite ter um ambiente de execução idêntico ao de uma configuração *single-core* facilitando assim a migração de aplicações *legacy*.

2.3.2.1 AMP Supervisionado

Similarmente à tecnologia de virtualização, a configuração AMP *multicore* é uma outra forma de obter diferentes classes de sistemas operativos dentro de uma mesma plataforma física. Encontra-se, no entanto, limitado à utilização de um número de sistemas operativos igual ao número de núcleos existentes na plataforma *multicore*.

Comparativamente à técnica de virtualização dual-*guest*, a utilização da configuração AMP permite níveis de desempenho melhorados uma vez que o tempo de processamento não é partilhado, mas sim paralelizado. Esta paralelização contribui para a mitigação de problemas de *starvation*. Atribuindo a cada core um diferente sistema operativo garante-se um isolamento temporal suficiente para impedir o problema supracitado. Este problema reflete-se em sistemas virtualizados,

quando um dos sistemas operativos, pelo tipo de escalonamento utilizado, não dispõe de tempo necessário para cumprir prazos de tempo-real ou em último caso de tempo necessário para sequer executar.

A atribuição de um sistema operativo a um determinado core pode ocorrer de forma estática em *design time* ou de forma dinâmica, recorrendo a mecanismos de comunicação inter-core. Apesar do isolamento temporal fornecido pela configuração assimétrica, esta carece de um isolamento espacial. Este isolamento, como supracitado, é imposto de forma estática ou através de mecanismos de comunicação inter-core. No entanto, ambas as soluções carecem de um isolamento próprio, uma vez que em casos de funcionamento defeituoso de uma das partes integrantes, o isolamento é quebrado.

A solução encontrada para esta falta de isolamento é a virtualização do sistema *multicore*. Esta solução pode também ser vista como uma migração de um sistema virtualizado para uma arquitetura *multicore*. Esta solução, também designada de AMP supervisionado, é uma integração da tecnologia *multicore* (AMP) e das técnicas de virtualização, como observado na Figura 2.6. Esta técnica permite obter os níveis de desempenho atribuídos às plataformas *multicore* e ao mesmo tempo um isolamento seguro entre os *guests* imposto pelo hypervisor.

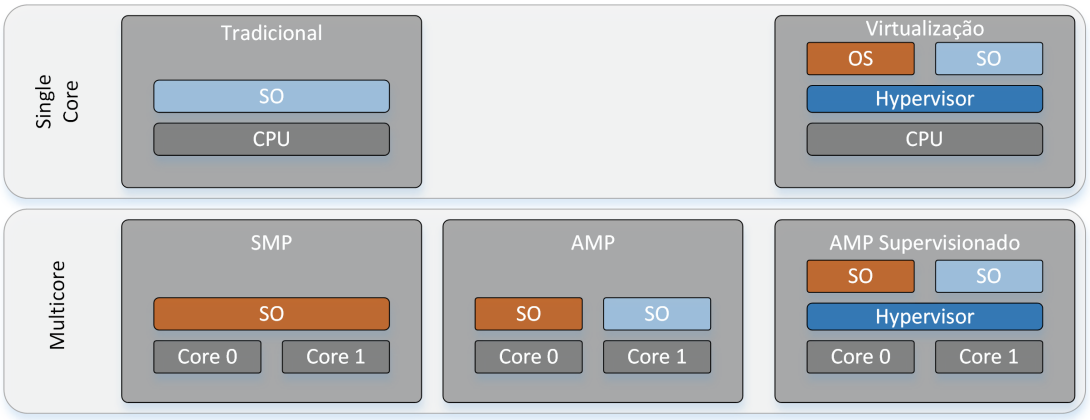


Figura 2.6: Configurações de virtualização e técnicas *multicore* plataformas *single-core* e *multicore*

Do ponto de vista de um sistema AMP nativo, para além do isolamento a nível de segurança e requisitos de tempo-real, a virtualização do mesmo vem reduzir a complexidade dos próprios sistemas operativos habitantes. Esta redução deve-se à partilha dos diferentes recursos entre os diferentes cores passar a ser gerida pelo hypervisor.

Apesar de ser uma tecnologia relativamente jovem e de ainda haver múltiplos desafios à implementação da virtualização *multicore* em sistemas embebidos, esta,

pelas suas garantias, flexibilidade e acima de tudo escalabilidade tem sido uma das tecnologias mais procuradas. De entre os trabalhos realizados na área destacam-se a presença de hypervisores comerciais como o Mentor Embedded Hypervisor, Wind River Hypervisor ou Xen e ainda hypervisores provindos do âmbito académico como o Xtratum [17] e o SASP [20] que procuram obter o máximo proveito das capacidades oferecidas pela plataforma *multicore*.

2.4 Comunicação Inter-Partição

Em sistemas virtualizados a comunicação é um dos componentes vitais para o aproveitamento das potencialidades do próprio sistema. Apesar de não ser obrigatória, a sua presença em sistemas virtualizados é uma constante dado as diversas vantagens inerentes à capacidade de comunicação das partições. A comunicação pode ser realizada entre as próprias partições do sistema, como pode ser realizada entre as partições e o próprio hypervisor, tendo ambas funcionalidades distintas.

Quando realizada entre partição e hypervisor, a comunicação tem como objetivo garantir à partição o acesso a um qualquer dispositivo presente no sistema. Esta comunicação deverá ser realizada através do hypervisor de forma a que o mesmo possa controlar os acessos ao dispositivo e inviabilizar aqueles que possam comprometer o isolamento do sistema. Em casos específicos como no hypervisor Xen, a gestão de dispositivos é migrada para a partição de maior prioridade, "dom0", sendo requerido às restantes partições a comunicação com o "dom0" e não com o hypervisor. Também frequentemente denominada de para-virtualização de dispositivos, esta forma de comunicação é implementada usualmente através de uma abordagem própria ao autor, como observado nas para-drivers desenvolvidas pela XenProject, com uso exclusivo no seu hypervisor Xen. No entanto, surgiu uma tecnologia que tem vindo a ser considerada como “de-facto” standard na virtualização de dispositivos [21], utilizada em hypervisores como o lguest [22], kvm [23] e ainda outros projectos [24].

A comunicação inter-partição ocorre entre partições de um sistema virtualizado seja ele *single-core* ou *multicore*, ou entre partições de um sistema *multicore* não supervisionado. Este tipo de comunicação tem um espectro de funcionalidades mais amplo:

i) **Nivelação da carga de trabalho**: Esta situação é vista em sistemas *multicore* assimétricos virtualizados e não virtualizados, onde a cada core é

atribuído um sistema operativo próprio. Se um sistema operativo estiver sobrecarregado e o nível de carga de trabalho dos restantes for diminuto, pode-se utilizar a comunicação inter-partição para que os restantes executem parte do seu trabalho, se houver essa compatibilidade. Esta aplicação da comunicação permite criar uma nivelação de cargas de trabalho transversalmente aos cores da plataforma, permitindo potencializar o subaproveitamento de um ou mais cores.

ii) **Cooperação entre partições**: Esta cooperação será definida pela aplicação, tendo consequentemente uma abrangência mais ampla. A motivação mais usual para esta cooperação é o aproveitamento de características únicas dos *guests* integrados nas diferentes partições. Nestes casos, existindo sistemas operativos heterogéneos de classes distintas, os mesmos através da comunicação podem usufruir das capacidades uns dos outros. Isto é, um sistema operativo de tempo-real poderá fazer uso das propriedades gráficas do sistema através da cooperação com o sistema operativo de propósito geral, como observado na Figura 2.7. Esta cooperação pode ser motivada em casos esporádicos pela atribuição exclusiva de um recurso a uma partição por parte do hypervisor, sendo necessária a comunicação inter-partição para a utilização desse recurso por parte de partições não autorizadas.

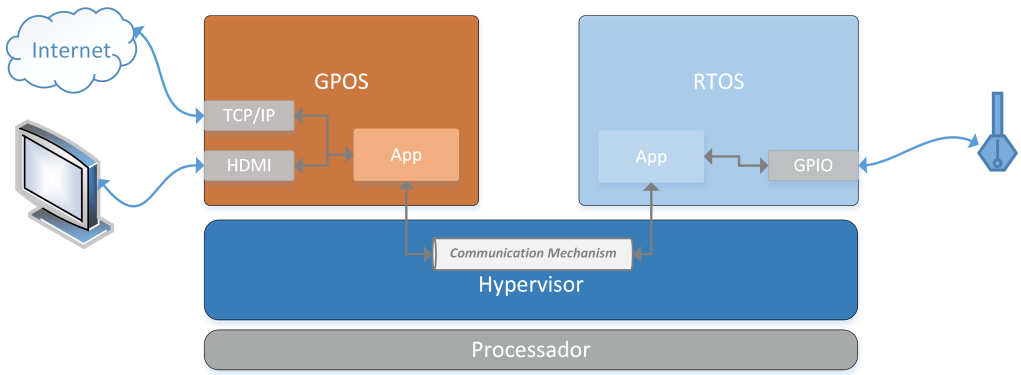


Figura 2.7: Exemplo de cooperação entre diferentes partições

Em sistemas virtualizados, a comunicação inter-partição pode ser realizada de diversas maneiras. Pode ser dividida de acordo com o seu sincronismo e de acordo com a existência ou não de memória partilhada assim como o seu nível de privilégio.

Quanto ao seu sincronismo, existem duas opções possíveis:

- **Comunicação Síncrona**: Também denominada de comunicação com bloqueio, é caracterizada por cada partição necessitar de esperar pela resposta

da outra parte integrante na comunicação. Este tipo de comunicação devido à sua ineficiência é menos comum em sistemas virtualizados, principalmente em sistemas de tempo-real em que o bloqueio causado pela comunicação poderá ter como última consequência o incumprimento de prazos de tempo-real. Por outro lado, esta é também o tipo de comunicação considerado mais seguro e usado inclusivamente nas especificações de comunicação de TEEs (Global Platform's TEE Client API Specifications).

- **Comunicação Assíncrona**: Também denominada de comunicação por eventos, é caracterizada por utilizar eventos para notificar as partes integrantes de que existe uma receção de mensagem. Esta abordagem é considerada a mais eficiente pois permite às partições que continuem a executar enquanto que esperam por uma resposta/mensagem. Isto é possível separando o canal de dados do canal de eventos e recorrendo ao uso de *callbacks* para servirem as mensagens recebidas fora do tempo esperado. No entanto, se não for corretamente implementada, esta abordagem pode ser alvo de diverso ataques, entre os quais o DoS (*"Denial-of-Service"*) onde uma partição impede o funcionamento normal de outra através do envio sucessivo/excessivo de notificações.

Do ponto de vista do canal de dados existem também duas opções, sendo que cada abordagem representa um *trade-off* entre performance e segurança. O canal de dados é implementado com recurso a:

- **Memória partilhada**: Esta primeira abordagem representa uma comunicação mais rápida e de menor segurança, sendo o seu principal alvo de ataque a própria memória partilhada. Esta pode ter um controlo de acesso (realizado pela MMU) ao nível de privilégio de execução, podendo ser apenas acedida em modo *kernel* ou em modo *user* (implementação da comunicação no SafeG ainda que com certos requisitos impostos [25]). Apesar de ter controlo de acesso, se contornado significa uma porta aberta para a quebra do isolamento entre partições. A gestão da memória partilhada neste tipo de comunicação é usualmente feito a partir de uma abordagem própria, porém existe uma tendência em criar um standard centrado na tecnologia VirtIO, como demonstram os trabalhos [2, 26, 24].

- **Cópia de memória, realizada pelo hypervisor**: Esta abordagem proporciona uma maior segurança em comparação com a primeira abordagem, em detrimento da performance. A diminuição de performance deve-se sobretudo a um excessivo número de cópia de dados assim como um aumento do

número de trocas de contexto (*"context-switch"*) [27, 28]. No entanto, esta abordagem permite evitar ataques à comunicação relacionados com falhas de memória causadas pela remoção de páginas partilhadas ou corrupção da própria memória.

Nas próximas subsecções serão discutidas algumas das diferentes comunicações inter-partição implementadas em diversos hypervisores ou sistemas supervisionados. Será dado um enfase extra à especificação OpenAMP e correspondente implementação *open-source* desenvolvida em conjunto pela Xilinx e Mentor Graphics.

2.4.1 Xen

O Hypervisor Xen [18] é um hypervisor com suporte *multicore* e *multiguest*, permitindo a conjugação de diferentes sistemas operativos numa mesma plataforma. O mecanismo de comunicação implementado no hypervisor é utilizado para comunicação inter-partição *per se*, assim como para a para-virtualização de dispositivos realizada através da partição de maior privilégio denominada de "dom0".

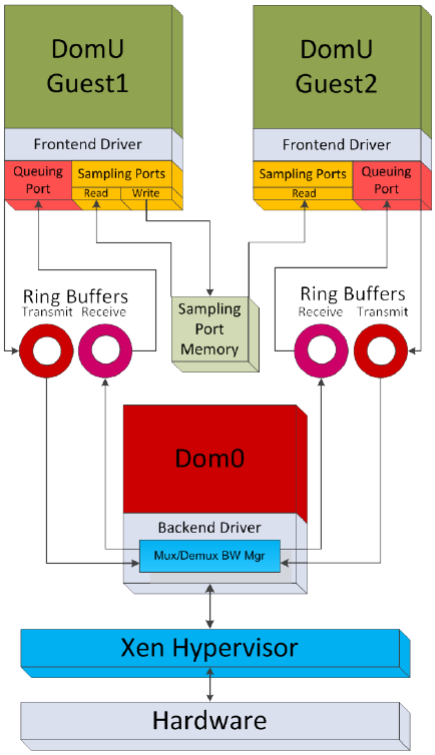


Figura 2.8: Exemplo de comunicação no hypervisor Xen direcionado para a indústria aviónica [1]

A Figura 2.8 representa uma implementação da comunicação direcionada para a industria aviónica [1]. A utilização de buffers circulares unidirecionais e de portas de amostragem é implementada fazendo uso de memória partilhada ao nível

do modo *kernel*. Como supramencionado, apenas a partição de maior privilégio possui acesso aos dispositivos da plataforma através de para-virtualização, sendo a comunicação necessária para que as restantes partições façam uso desses dispositivos.

A comunicação possibilita ainda a comunicação inter-partição entre qualquer uma das partições, não sendo necessariamente exclusiva com o "dom0". O mecanismo de comunicação é suportado pelo "*Xen bus*", implementado especificamente para a arquitetura de hypervisores Xen.

2.4.2 OpenAMP e Linux Drivers

A Texas Instruments (TI) desenvolveu duas tecnologias denominadas de Remoteproc e RPMsg. O Remoteproc permite a um sistema operativo fazer a gestão de ciclo de execução dos diferentes cores integrantes na plataforma, enquanto que o RPMsg define um protocolo para comunicação entre cores. Estas duas tecnologias em conjunto permitem a interação entre diferentes sistemas operativos hospedados em diferentes cores.

Essas tecnologias foram desenvolvidas para o sistema operativo Linux (versão *open-source*) e para sistemas operativos pertencentes à TI (versão comercial), estando portanto limitadas ao software pertencente à TI assim como o próprio hardware. Visto com grande potencialidade para uso em plataformas heterogéneas e homogéneas, a tecnologia desenvolvida pela TI para o SO Linux foi complementada pela especificação OpenAMP, criada pela Multicore Association[5]. Orientado para sistemas operativos de recursos limitados como RTOS e sistemas *baremetal* (sistemas nativos), esta especificação tem como principal objetivo criar um standard ao nível de comunicação inter-core e gestão de ciclo de execução de cores. Esta tecnologia tem ainda a vantagem de ser compatível com outra especificação criada pela Multicore Association, MCAPI (Multicore Communications API), provando por isso ser um método de comunicação eficaz e escalável.

Para reforçar a importância dada à potencialidade da especificação, a mesma obteve uma implementação em formato *open-source* desenvolvida em conjunto pela Multicore Association, Mentor Graphics e Xilinx. A implementação desta especificação garante ainda facilidade de portabilidade quer entre SOs, quer entre diferentes classes de processadores, como se pode ver na Figura 2.9 através das camadas “Environment” e “HIL” respetivamente. As partes integrantes desta especificação serão posteriormente analisadas na Subsecção 3.3.2.3.

[5]http://www.multicore-association.org/workgroup/oamp.php

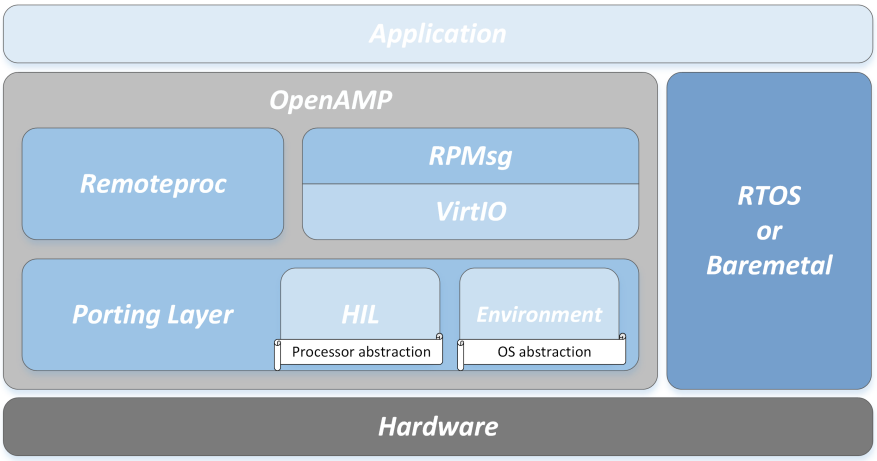


Figura 2.9: Arquitetura da Implementação OpenAMP

2.4.3 Mentor Embedded Multicore Framework (MEMF)

Para além da implementação *open-source*, a especificação OpenAMP obteve ainda uma implementação comercial por parte da Mentor Graphics. Essa implementação encontra-se inserida na sua *framework* denominada Mentor Embedded Multicore Framework (MEMF) [2], cujo objetivo é facilitar e providenciar ferramentas para a implementação e migração para plataformas heterogéneas. Entre estas ferramentas, contam-se a inclusão de uma implementação comercial do OpenAMP e ainda ferramentas de ajuda no desenvolvimento e depuração de tecnologias *multicore* em plataformas heterogéneas.

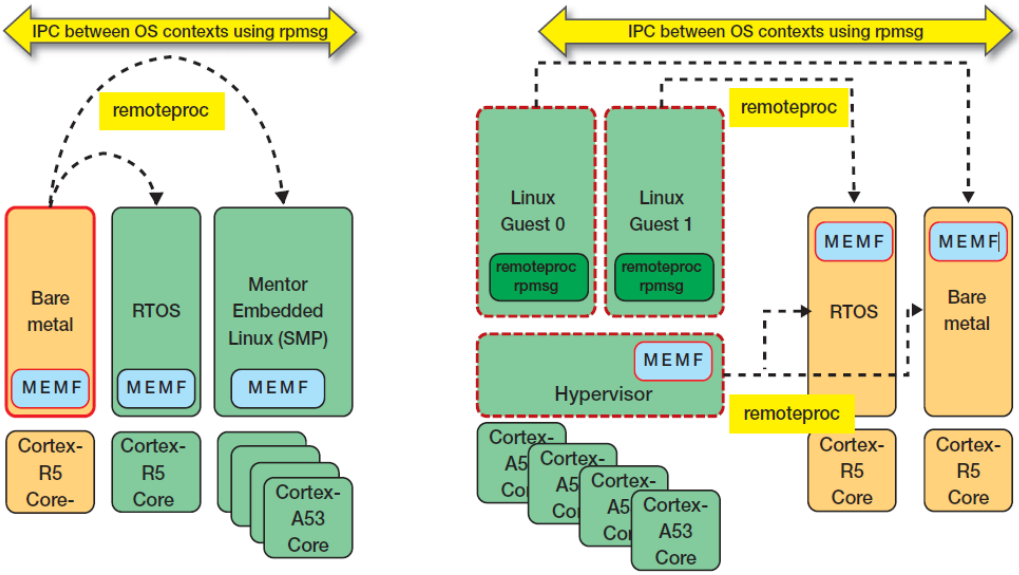


Figura 2.10: Arquitetura do MEMF na plataforma Zynq UltraScale [2]

A ferramenta OpenAMP conta com uma implementação *open-source* por parte

da Xilinx e Mentor Graphics, no entanto, esta ainda se encontra num estágio de desenvolvimento inicial. A implementação *open-source* inclui unicamente o modo *slave* da comunicação. Adversativamente o SO Linux implementa apenas o modo *master* da comunicação Remoteproc/RPMsg, pois foi assumido implicitamente que esse seria o design mais frequente, especialmente ao nível do Remoteproc.

Apesar de fazer parte do grupo de desenvolvimento da implementação *open-source*, a Mentor Graphics quis incluir na sua *framework* MEMF todas as características explicítas na especificação OpenAMP através de uma implementação comercial própria e adaptada, concedendo a utilização a todos os *guests* de todos os modos de comunicação/interação inter-core. Esta inclusão engloba a adição do modo *slave* à sua versão do Linux (Mentor Linux) e do modo *master* à versão *baremetal*/RTOS, possibilitando uma maior diversidade de designs, dotando a sua *framework* de uma maior versatilidade.

Esta implementação comercial do OpenAMP conta ainda com a utilização das funcionalidades da especificação OpenAMP entre os diferentes *guests* de um sistema virtualizado, nomeadamente através do hypervisor *Mentor Embedded Hypervisor* [6]. Através da Figura 2.10 podemos observar a arquitetura do MEMF numa plataforma heterogénea Zynq UltraScale. Como referido, é possível o uso da framework OpenAMP num ambiente *multicore* virtualizado (*Mentor Embedded Hypervisor*) assim como num ambiente *multicore* supervisionado (MEMF), provando a escalabilidade da tecnologia OpenAMP.

[6] https://www.mentor.com/embedded-software/hypervisor/

Especificação do Sistema

3.1 Arquitetura ARM

Para a realização da presente dissertação foi escolhida a utilização de tecnologias presentes em processadores ARM, nomeadamente a tecnologia ARM TrustZone. Esta escolha deve-se à proliferação dos processadores ARM no ambiente de sistemas embebidos, obtendo um crescimento exponencial especialmente nos últimos anos. Este crescimento revela um impacto ainda maior quando se fala particularmente do mundo da tecnologia móvel, em que a presença de processadores ARM contabiliza uns impressionantes 95% da quota de mercado.[1]

Os processadores ARM seguem uma arquitetura RISC (*Reduced Instruction Set Computer*), tal como indica o seu próprio nome (ARM - *Advanced RISC Machine*). Contrariamente à arquitetura CISC (*Complex Instruction Set Computer*), a filosofia da arquitetura RISC traduz-se numa redução da complexidade das instruções realizadas em hardware, migrando muita da capacidade de execução para o nível do software, providenciando assim uma maior flexibilidade. Esta característica permite reduzir a complexidade e tamanho do hardware, reduzindo o consumo de energia assim como a área de silício drasticamente em detrimento da performance. A redução do consumo energético e do próprio preço (área de silício reduzida) são características apreciadas no mundo dos sistemas embebidos como referido na Secção 2.1.

A redução da complexidade do hardware e consequente reduzida capacidade de execução pode traduzir-se numa redução de performance. Para combater esta desvantagem as arquiteturas RISC utilizam técnicas como o *pipeline*, registos e instruções separadas para acessos à memória. A primeira técnica consiste na divisão do processamento das instruções em pequenas partes, para que estas possam ser executadas simultaneamente, garantindo assim um melhor rendimento. A segunda e terceira técnica consistem em realizar operações sobre registos de propósito geral, cujo acesso é bastante rápido e de utilização genérica. Efetuando o armazenamento destes registos na memória principal em diferentes alturas permite uma eficiência maior no processamento.

[1] https://www.arm.com/markets/mobile

3.1.1 Arquitetura ARM - Conceitos Básicos

Dentro dos processadores ARM, existem várias arquiteturas disponíveis, arquiteturas com propósitos diferentes. Em termos de evolução das mesmas podemos observar as várias versões das arquiteturas ARM: ARMv1 a ARMv8. Devido à evolução da tecnologia envolvida nos processadores ARM surgiu a necessidade de dividir o desenvolvimento dos mesmos de acordo com os seus propósitos. Esta divisão consiste em três diferentes propósitos: geral, baixo consumo energético e determinismo. Na Tabela 3.1 pode-se observar os diferentes propósitos e finalidades das diferentes arquiteturas dentro da versão ARMv7, também denominada de processadores Cortex-X (em que X representa a indicação dos seus propósitos).

Tabela 3.1: Diferentes processadores dentro da arquitetura ARMv7

Arquiteturas ARMv7	Propósitos	Finalidade	Exemplos de Utilização
Cortex-A	Propósito Geral	Performance em Sistemas Embebidos	Industria Móvel
Cortex-R	Determinismo	Sistemas de Tempo Real	Sensores e Cálculos
Cortex-M	Baixo Consumo Energético	Microcontroladores	Sistemas Autosuficientes

O trabalho realizado na presente dissertação foi desenvolvido sobre um processador Cortex-A9, o qual incorpora uma arquitetura ARMv7-A. Esta arquitetura, bem como o processador Cortex-A9, está preparada para ser utilizada em plataformas *multicore* homogéneas. Uma plataforma *multicore* que utilize processadores Cortex-A9 pode possuir até 4 cores no seu SoC, podendo existir as diferentes topologias: single-core, dual-core e quadcore. No caso específico da plataforma utilizada, a plataforma contém dois cores Cortex-A9, conforme será explicado na Secção 3.2.

Os processadores pertencentes à arquitetura ARMv7-A, à semelhança de processadores em sistemas embebidos direcionados para a performance (propósito geral), são acompanhados de alguns periféricos com um alto impacto e influência no seu comportamento geral: i) caches, ii) controladores de memória e ainda iii) a existência de coprocessadores. As caches, blocos de memória de rápido acesso inseridas entre o processador e a memória principal, contribuem para uma melhoria de performance pois permitem o armazenamento temporário de dados pertencentes à memória principal. Os controladores de memória, nomeadamente a MPU (*Memory Processing Unit*) e a MMU (*Memory Management Unit*) oferecem uma ajuda na organização e proteção de zonas de memória usadas pelas aplicações,

sendo que grande parte dos sistemas operativos de propósito geral requerem a sua existência para que possam executar. Os coprocessadores podem ter uma variedade de funções enormes, usualmente aumentando o conjunto de instruções do processador. Podem aumentar o número de funcionalidades de um processador como lhe proporcionar uma interface de controlo de certos periféricos (*e.g., cp 15*).

Processadores com a arquitetura ARMv7-A possuem tipicamente dezasseis registos de propósito geral, que, como supramencionado, são de mais rápido acesso em comparação com a memória principal. Apesar do seu propósito geral e acesso indiscriminado, existem três registos que contém particularidades em relação aos outros: os registos *r13, r14 e r15*. Estes registos têm uma funcionalidade específica, nomeadamente: (i) ***r13*** – *stack pointer*, armazena o endereço do topo da stack do modo de execução actual; (ii) ***r14*** – *linker register*, contém o endereço de retorno do programa actual; (iii) ***r15*** – *program counter*, atualizado a cada ciclo de relógio com o valor do endereço da instrução a ser executada. Para além destes dezasseis registos de propósito geral, os processadores Cortex-A possuem ainda dois registos específicos denominados de *cpsr – current program status register* e *spsr – saved program status register*. Estes registos contêm informações acerca do modo de execução do processador atual e guardado pela última vez, respetivamente. Estes últimos registos contêm ainda informações sobre o resultado de operações que alterem o valor das *flags* condicionantes (*Zero, oVerflow, Carry e Negative*, assim como informações sobre o estado dos bits de interrupção (*FIQ e IRQ*).

Tabela 3.2: Modos de execução de um processador ARM e respetivos níveis de privilégio

Processor Mode	Exception Level
User	PL0
System	PL1
Supervisor	PL1
Abort	PL1
Undefined	PL1
FIQ	PL1
IRQ	PL1
Monitor	PL1
Hypervisor	PL2

Os processadores com a arquitetura ARMv7-A contemplam atualmente até 9 modos de operação distintos (dependendo se têm implementadas as extensões opcionais). Na Tabela 3.2 estão sumarizados a totalidade desses modos, sendo que o último, *hypervisor mode*, não se encontra presente no processador Cortex-A9. O

modo atual do processador é dado pelos primeiros 4 bits (definidos como Mode Field) do registo Current Program Status Register (CPSR).

Cada modo tem um nível de privilégio diferente. Cada nível de privilégio confere ao modo em execução diferentes permissões: desde acessos a determinados periféricos a execuções de diferentes instruções. Na Tabela 3.2 observa-se a relação entre o nível de privilegio (*Priviledge Level – PL*) e o modo de execução do processador.

O processador entra nos diferentes modos mediante a ocorrência de exceções específicas ou através da escrita direta no registo CPSR. O processador entra em *abort mode* quando ocorre uma falha num acesso à memória, em *undefined mode* quando tenta executar uma instrução que não lhe é permitida ou não existe, e em *interrupt mode* (*FIQ ou IRQ*) quando ocorre uma interrupção do nível correspondente. Modos como *supervisor, system, user e monitor* são ativados por instruções especificas ou através da escrita direta no CPSR.

As aplicações executam geralmente em modo *user* (PL0). Neste modo a memória encontra-se protegida pela MPU ou MMU, se as mesmas existirem. Para o processador comutar de modo, será necessária a utilização explícita de uma instrução denominada SVC ou através da ocorrência de uma das exceções supracitadas. Os restantes modos são normalmente denominados de modos privilegiados. O processador, após *reset*, encontra-se em modo *supervisor*, sendo este o modo em que tipicamente operam os sistemas operativos.

Tabela 3.3: Mapa de registos do processador Cortex-A9

User/System Mode	Supervisor Mode	Abort Mode	Undefined Mode	FIQ Mode	IRQ Mode	Monitor Mode
r1						
r2						
r3						
r4						
r5						
r6						
r7						
r8				*r8_fiq*		
r9				*r9_fiq*		
r10				*r10_fiq*		
r11				*r11_fiq*		
r12				*r12_fiq*		
r13 (sp)	*sp_svc*	*sp_abt*	*sp_und*	*sp_fiq*	*sp_irq*	*sp_mon*
r14 (lr)	*lr_svc*	*lr_abt*	*lr_und*	*lr_fiq*	*lr_irq*	*lr_mon*
r15 (pc)						
CPSR						
-	*SPSR_svc*	*SPSR_abt*	*SPSR_und*	*SPSR_fiq*	*SPSR_irq*	*SPSR_mon*

De forma idêntica, cada modo de execução garante ao processador acesso a registos próprios também denominados de registos banqueados (definidos com a côr cinzenta na Tabela 3.3). Isto é, cada registo banqueado é acedido de maneira idêntica, no entanto existe uma cópia do mesmo que apenas é passível de ser acedida no modo respetivo, sendo as cópias respetivas dos outros modos salvaguardadas e inacessíveis. Na Tabela 3.3 pode-se verificar os registos próprios a cada modo de execução. O modo *system* apesar de ser um modo privilegiado, tem acesso aos mesmos registos de um modo não priveligiado (*user*), no entanto contém permissões extras em relação a este último. Esta característica foi implementada pela ARM para que o modo fosse utilizado nas rotinas de serviço a interrupções que possibilitem reentrância de interrupções ou até de chamadas SVC, evitando assim a corroboração de registos como o *lr*.

Na Tabela 3.2 está também representado um modo denominado de *monitor mode*. Este modo é introduzido pelas extensões de segurança TrustZone, já aqui referenciadas como opcionalmente exploradas para a implementação de soluções de virtualização (subsecção 2.2.2.3). Não obstante o nível de privilégio, este modo é diferente dos restantes por só estar disponível no estado seguro do processador, definido pelas extensões ARM TrustZone. O modo *hypervisor*, introduzido pelas extensões ARM VE, apresenta um nível de privilégio superior, no entanto, o mesmo só está disponível no estado não-seguro do processador, sendo teoricamente um modo menos privilegiado do que o modo monitor. Na próxima secção pretende-se explorar o hardware e funcionalidades adicionadas pela tecnologia TrustZone.

3.1.2 ARM TrustZone

A arquitetura ARMv7-A possui extensões em hardware no próprio processador direcionadas para a segurança, denominadas de ARM TrustZone. As extensões de segurança pretendem munir o processador de mecanismos capazes de ajudar a combater os problemas de segurança que têm vindo a assolar o mundo dos sistemas embebidos, particularmente em áreas de grande interconectividade. Uma das utilizações mais populares deste tipo de extensões de segurança verifica-se na tecnologia "Touch ID" dos Iphones, uma das bases da integridade do smartphone[2], que usa um sistema personalizado baseado na tecnologia ARM TrustZone.

[2]http://support.apple.com/kb/ht5949

3.1.2.1 Processador

Esta tecnologia divide o processador em dois ambientes de execução, focando a segurança num dos ambientes. Esses ambientes de execução, pelo isolamento conferido pela ARM TrustZone são designados de autênticos processadores virtuais, designados de mundo seguro e mundo não-seguro. Com a adição de dois mundos virtuais o nível de privilégio dos modos de execução deixa de ser linear, passando a existir um modo ortogonal. Os modos de execução do mundo seguro irão ter de forma parcial um privilégio superior aos modos inseridos no mundo não-seguro.

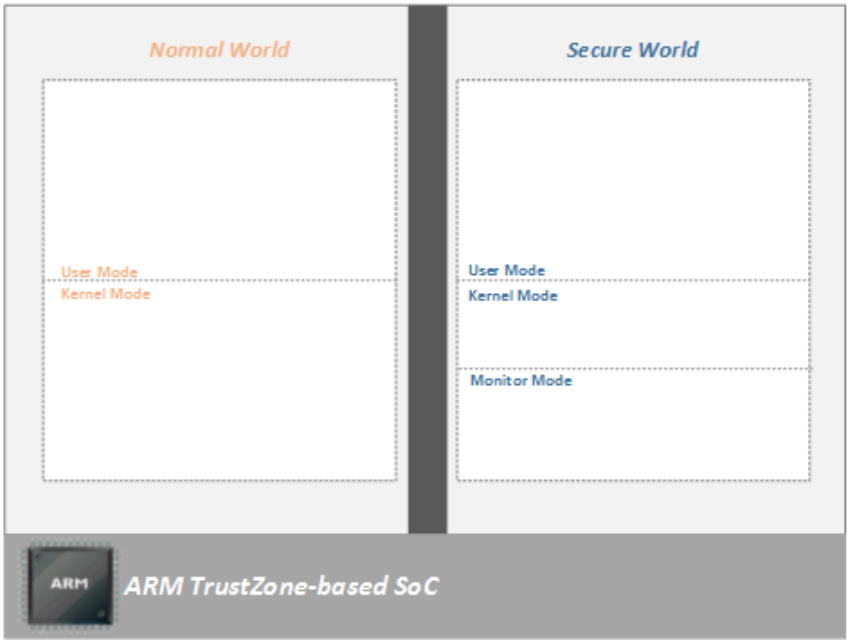


Figura 3.1: Divisão entre cores virtuais no mesmo core físico

Através da Figura 3.1 pode-se observar a divisão do processador assim como a adição de um novo modo de execução denominado de *monitor mode*. Este modo, à semelhança de outros modos privilegiados, possui registos banqueados, nomeadamente o *SPSR, r14 (lr)* e *r13 (sp)*. O modo monitor tem privilégios idênticos aos dos modos de execução de nível PL1 que executem no mundo seguro, no entanto este modo possui três particularidades em relação aos restantes:

- O modo monitor executa sempre no mundo seguro independentemente do valor da variável SCR-NS. Este pode aceder a valores de registos ou memória de acordo com o valor atribuído a essa mesma variável;
- A adição de uma nova instrução para entrada em modo monitor ao ISA do ARMv7-A – *Secure Monitor Call* (SMC). Esta instrução permite a qualquer mundo, desde que execute em modo privilegiado, entrar em modo monitor, de forma idêntica a uma *supervisor call* (svc);

- O monitor é capaz de fazer o *trap* de qualquer exceção que ocorra num dos mundos. Esta particularidade permite realizar o redireccionamento de interrupções para os mundos adequados. Isto é, se o monitor for programado para realizar o *trap* de FIQs no mundo normal, poderá, mesmo que a mesma ocorra fora do ambiente de execução segura, permitir ao mundo seguro servir essa mesma interrupção com um tempo de latência reduzido.

3.1.2.2 Memória

Para garantir o isolamento entre mundos o processador baseia-se no valor de um bit chamado NS (*non-secure*) bit que terá o valor de 1 se o ambiente de execução for pertencente ao mundo não-seguro e 0 se pertencente ao mundo seguro. Este bit é propagado em todos os barramentos do processador, possibilitando a proteção de todos os recursos do processador. Este bit, nas arquiteturas de 32-bit da ARM é muitas vezes denominado de 33º bit [19].

Recorrendo a este bit, o processador banqueia alguns registos importantes do coprocessador 15, sendo o acesso aos registos não banqueados, considerados críticos, negado ao mundo não-seguro, assegurando o isolamento entre mundos. Entre estes periféricos encontram-se a MMU, as Caches, o barramento AXI, entre outros, sendo que existe ainda a possibilidade de configurar em *run-time* a segurança de outros periféricos.

A memória principal encontra-se protegida pelo TrustZone Address Space Controller (TZASC) acedido apenas por software em execução no mundo seguro. Este controlador possui a capacidade de atribuir níveis de segurança a blocos de memória predefinidos. A dimensão dos blocos de memória é definida pelo vendedor da plataforma. Isto permite que cada região de memória possa ser configurada como segura ou não-segura, possibilitando o particionamento da memória. Este isolamento apenas é realizado para o mundo não-seguro, uma vez que o mundo seguro possui permissão para aceder a memoria não-segura. O controlador TrustZone Memory Adapter (TZMA) proporciona as mesmas funcionalidades, mas para a memória *on-chip* (OCM). A granularidade deste controlador é usualmente mais fina, limitando-se no entanto a atribuições estáticas.

A MMU, um dos periférico mais importante no controlo de acessos à memória do ponto de vista de cada sistema operativo, possui extensões próprias que a dividem a sua interface virtualmente em duas, uma para cada processador virtual. Isto permite que cada mundo possua a sua própria tabela de mapeamentos de páginas virtuais da memória. Os descritores de cada tabela conterão o bit identificador do estado de segurança da memória, negando o acesso a dados seguros ao mundo

não-seguro. As tabelas são próprias a cada mundo e trocadas assim que acontece uma comutação de mundos. O conjunto de TLBs (*Translation Lookaside Buffers*) permite a coexistência de resultados da MMU de ambos os mundos, sendo o seu acesso coerente com o estado de segurança. Isto irá permitir uma aceleração no processo de troca de contextos dos mundos.

As Caches foram extendidas de modo a suportarem o armazenamento do 33º bit nas suas *"lines"*. Esta modificação permite a coexistência de dados do mundo seguro e do mundo não-seguro numa mesma cache, sendo que o seu acesso é sempre controlado: o mundo não-seguro apenas pode aceder a *"Cache lines"* com o bit não-seguro ativo vice-versa. Esta coexistência permite acelerar mais uma vez os *context-switches* entre os mundos virtuais, pois o isolamento é imposto por hardware ao nível das caches.

Como referido anteriormente existem periféricos cuja segurança pode ser dinamicamente modificada. Usualmente estes periféricos contêm interfaces próprias no processador que permitem realizar esta modificação ou então ela é realizada pelo periférico em si (no caso de periféricos inteligentes programados na FPGA). De qualquer forma, esta modificação apenas pode ser respeitada devido à propagação e respetivo suporte dentro do barramento AXI que realiza a interface com a maioria dos periféricos num processador ARM Cortex-A9. O controlador TrustZone Protection Controller (TZPC) pode ser colocado no barramento APB (compatível com AXI) para realizar o controlo de acessos de outros componentes no SoC. Qualquer acesso não autorizado a estes dispositivos, protegido pelos mecanismos das extensões TrustZone, resulta no desencadeamento de uma exceção *Undefined Exception*.

Não obstante existem certos periféricos cuja segurança não pode ser modificada, por serem considerados demasiado influentes no funcionamento do sistema e poderem facilmente quebrar o isolamento entre mundos. Ou ainda, por uma questão de design, serem considerados apenas e unicamente seguros, tendo que ser feita uma espécie de para-virtualização dos mesmos se necessária e autorizada a sua utilização por parte do mundo não-seguro.

3.2 Ambiente de Desenvolvimento

Neste subcapítulo irão ser realizadas as análises das diversas ferramentas utilizadas ao longo do desenvolvimento da parte prática da dissertação.

3.2.1 Zynq-7000

Para a implementação da parte prática da dissertação foi escolhida a plataforma Zedboard, vendida pela Digilent, Inc e Avnet. Esta *board* satisfaz na totalidade as condições para a realização do trabalho proposto, visto conter no seu interior o SoC Zynq-7000, um chip que integra dois cores Cortex-A9 e uma FPGA (*Field Programable Gate Array*). Aliado à tecnologia *multicore* o chip garante ainda a capacidade de uso da tecnologia ARM TrustZone, muitas vezes não implementado em plataformas com diferentes SoCs mas incluindo o mesmo processador (Cortex-A9), produzido por outros vendedores.

Este chip, *Zynq-7000 All Programable SoC*, está disponível noutras plataformas, nomeadamente nas plataformas ZC706 e ZC702, e ainda na Zybo, plataformas produzidas pela própria Xilinx e Digilent, Inc. respetivamente. A framework em desenvolvimento [5] foi implementada na *board* ZC702. Nesta dissertação, devido às semelhanças de arquitetura, foi efetuado facilmente o *porting* para a plataforma Zedboard.

A Xilinx, produtora do SoC Zynq-7000, disponibiliza ainda diversos programas e materiais de suporte à realização de projetos nas suas plataformas (incluindo a Zedboard), tornando-se também este um aspecto favorável na escolha da *board*. Entre eles, destacam-se as ferramentas de desenvolvimento Xilinx SDK e Xilinx Vivado e ainda documentação técnica de suporte [3, 29]. Estes manuais foram fundamentais na compreensão da tecnologia TrustZone na plataforma assim como da própria plataforma.

Na Figura 3.2 está representada a arquitetura do hardware presente no SoC Zynq-7000. Durante a realização da dissertação foi dado maior destaque à PS (*Processing System*) do que à PL (*Programmable Logic*), pois esta última não servia os propósitos do trabalho realizado na dissertação. Pode-se observar na figura a constituição da PS, formada pelo bloco APU (*Application Processor Unit*) e pelos restantes periféricos. O bloco da APU integra dois cores ARM Cortex-A9, sendo que cada um tem os sues próprios periféricos integrados: MMU, Cache L1, FPU e NEON. Adicionalmente encontram-se os periféricos partilhados pelos cores, os periféricos I/O, Caches L2, DMA, a memória OCM e memória principal, assim como as interfaces para a PL. De destacar a presença dominante do barramento AXI entre os diferentes periféricos e a APU, sendo que este barramento como referido anteriormente é TrustZone-*aware*, *i.e.,* suporta as extensões de segurança TrustZone.

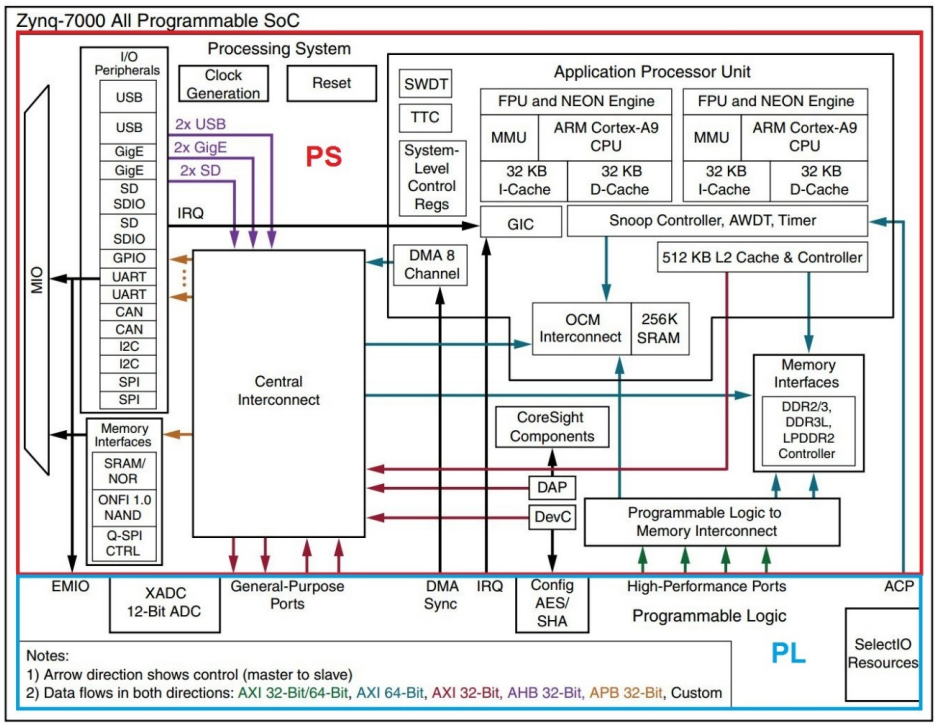


Figura 3.2: Arquitetura do Zynq-7000 AP SoC [3]

3.2.2 Xilinx Vivado e XSDK

Como supramencionado, a Xilinx disponibiliza, devido à sua complexidade, ferramentas próprias para a depuração e programação das plataformas que integram o seu SoC (Zynq-7000). Essas ferramentas são o Xilinx Vivado assim como Xilinx SDK, usualmente referido como XSDK. Na realização desta dissertação foram utilizadas as versões 2015.4.1 das ferramentas por serem as mais recentes aquando a iniciação do trabalho. Apesar de haver atualizações durante a realização do mesmo, optou-se por se fixar com as versões 2015.4.1 por serem mais maduras e não apresentarem erros influentes no projeto (algo que poderia acontecer numa versão recente – a versão 2015.4.1 é antecedida pela versão 2016.1).

3.2.2.1 Xilinx Vivado

A ferramenta de desenvolvimento Xilinx Vivado, formalmente conhecida como *Xilinx Vivado Design Suite*, direciona-se para o desenvolvimento e depuração de programas em hardware, nomeadamente nas suas FPGAs. Devido à natureza da dissertação, esta vertente das ferramentas não foi explorada excetuando a configuração do hardware da PS através da mesma (frequência do próprio processador e dos *clocks* de certos periféricos).

3.2.2.2 Xilinx SDK

O ambiente de desenvolvimento Xilinx SDK (*Software Development Kit*) é baseado no ambiente de desenvolvimento *open-source* Eclipse. É um IDE bastante completo possibilitando a realização de design, depuração (através da integração do JTAG *debug*) e análise de performance de aplicações em plataformas *multicore* homogéneas e heterogéneas. Engloba ainda uma interface com o Vivado Design Suite, proporcionando uma integração e configuração completa de ambos os componentes hardware e software das plataformas Xilinx.

Para além do ambiente de desenvolvimento IDE fornecido pelo Xilinx SDK, este integra ainda a sua própria *toolchain*. A *toolchain* integrada neste subsistema foi utilizada para a compilação completa das partes integrantes e da própria *framework* LTZVisor. Esta *toolchain* inclui duas componentes importantes denominadas com o prefixo de *arm-xilinx-linux-gnueabi-* e *arm-xilinx-eabi-*, cada uma responsável respetivamente pela compilação do próprio sistema operativo Linux e programas integrantes, e ainda de programas *baremetal* como o RTOS e hypervisores.

Como referido anteriormente a produtora dos SoC e distribuidora do software XSDK, Xilinx, teve uma participação muito ativa na promoção do seu chip, incluindo uma documentação diversificada assim como exemplos de uso de diferentes tecnologias presentes no mesmo. Entre esses exemplos destaca-se a implementação da especificação OpenAMP realizada em conjunto com a Mentor Graphics e a Multicore Association. Devido à sua participação nesse projeto, as plataformas que integravam os seus SoCs obtiveram um *porting* dessa mesma implementação, sendo as plataformas oficiais de testes do projeto. Apesar de ser um projeto *open-source* e de estar numa fase inicial, podendo observar-se algumas incoerências e partes incompletas, foi uma grande vantagem e considerado o ponto de partida para implementação da comunicação inter-partição, explicado na Secção 4.3.

De realçar ainda o *port* e a manutenção de um sistema operativo de propósito geral muito apreciado em sistemas embebidos, o Linux, acompanhando e atualizando as várias versões do mesmo. Desde a versão 3.3, utilizada na framework em desenvolvimento, à versão atual 4.10. Ademais, inclui ainda nos seus portefólios o porte do sistema operativo de Tempo Real FreeRTOS em várias versões.

3.2.3 ARM Fast Models

Foram analisadas diversas ferramentas de simulação de plataformas contendo processadores ARM para a depuração e desenvolvimento do trabalho prático da

dissertação. A escolha recaiu sobre o ARM Fast Models, uma ferramenta de desenvolvimento poderosa, capaz de simular até quatro processadores ARM, tendo a possibilidade de simular os processadores presentes na plataforma física, os processadores Cortex-A9. Além disso, uma das principais motivações para a sua escolha foi a de suportar a tecnologia de extensões de segurança ARM TrustZone, a única de entre as diversas analisadas.

A ferramenta ARM Fast Models permite para além da depuração, ainda o trace e a realização do *profiling*, integrando APIs próprias e sistemas de visualização próprios, resultando numa ferramenta bastante completa e crucial no desenvolvimento de tecnologia *multicore* com aplicação sobre a tecnologia TrustZone.

3.3 Arquitetura do Sistema

Será descrita neste subcapítulo a arquitetura do sistema, nomeadamente a arquitetura inicial da framework LTZVisor [5] numa plataforma contendo o SoC Zynq-7000. Será ainda descrita a arquitetura da comunicação especificada pelo OpenAMP, na qual a dissertação se irá basear para a implementação da comunicação inter-partição na *framework* LTZVisor.

3.3.1 LTZVisor

O LTZVisor (*Ligthweight TrustZone-assisted hyperVisor*) foi um projeto desenvolvido na Universidade do Minho com o propósito de demonstrar as funcionalidades das extensões de segurança, ARM TrustZone, nomeadamente na sua exploração para implementação de soluções de virtualização. A *framework* faz parte de um projeto maior cuja finalidade é a exploração e compreensão da tecnologia TrustZone nos variados âmbitos de projetos de virtualização, sendo o LTZVisor, como o nome indica, uma versão *lightweight*.

A Figura 3.3 demonstra a arquitetura inicial do LTZVisor anterior ao trabalho desenvolvido sobre o mesmo. Sendo uma versão reduzida e embrionária, este hypervisor apenas incorpora uma topologia *dual-OS*, com a inclusão de dois sistemas operativos de classes diferentes, GPOS e RTOS. Os sistemas operativos escolhidos foram o Linux 3.3 e o FreeRTOS 7.0.2 para desempenhar os papéis de GPOS e RTOS respetivamente. Como referido anteriormente o LTZVisor faz uso da tecnologia TrustZone para garantir o suporte de virtualização. A *framework* aproveita os mundos virtuais criados pela tecnologia para particionar o sistema em dois, colocando cada SO *guest* numa dessas partições.

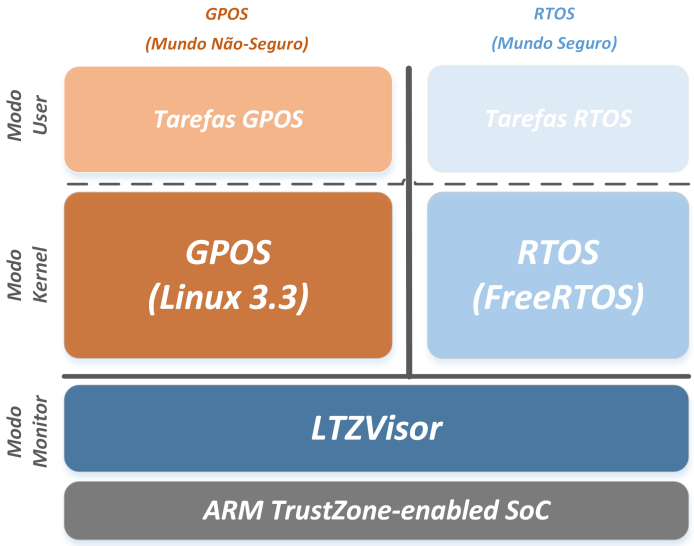


Figura 3.3: Arquitetura *single-core* do LTZVisor

3.3.1.1 Monitor

O LTZVisor aproveita o novo modo introduzido pela tecnologia TrustZone para a implementação da camada de abstração da virtualização, a qual terá visão e controlo sobre a totalidade do sistema. Como representado na Figura 3.3 a camada do hypervisor será uma camada mais *'ligth'*, denominada de monitor, isto porque as capacidades do hypervisor na *framework* LTZVisor são reduzidas.

Esta é uma *framework* de virtualização para sistemas de tempo-real, dando prioridade de execução ao RTOS que executa no mundo seguro, salvaguardando as características de tempo-real do sistema.

Hypervisor Tempo Real

As razões para a redução de funcionalidades devem-se à sua configuração *"lightweight"* assim como à política de escalonamento incorporada, denominada de *idle-scheduler*, que visa proteger as características de tempo-real do RTOS. Esta política assenta na ideia de que o hypervisor terá uma prioridade mais baixa de execução do que a do RTOS, prevendo uma maior facilidade do cumprimento dos prazos de tempo-real do mesmo. Nesta configuração, o monitor e consequentemente o *context-switch* para o GPOS apenas será autorizado a executar pela tarefa de *idle* do RTOS, através do uso da instrução *smc*.

A tecnologia TrustZone possibilita o *"trap"* de interrupções, sejam elas FIQ ou IRQ. Em concordância com a política de escalonamento que visa proteger o RTOS, parte da TCB, o monitor atribui as interrupções IRQ ao GPOS e as

interrupções FIQ ao RTOS. Fazendo esta atribuição e realizando o *"trap"* das FIQ enquanto o processador executa no ambiente não-seguro, permite realizar o redireccionamento das interrupções diretamente para o mundo seguro com a menor latência possível, salvaguardando as características de tempo-real do RTOS. Estas duas funcionalidades (política de escalonamento e o *trap*) podem ter como consequência o *"starving"* do GPOS. Isto ocorrerá se a carga de trabalho do RTOS for demasiada, de maneira a que o GPOS não possua tempo de execução suficiente, sendo uma das limitações visíveis da *framework* de virtualização.

Context-Switch

O *context-switch* efetuado pelo monitor é bastante reduzido comparando com técnicas de virtualização baseadas em emulação, isto uma vez que a própria tecnologia cria cópias diferentes de diversos registos e estados do processador para os diferentes mundos (incluindo do coprocessador 15). No entanto é necessário realizar a troca de alguns registos, salvaguardando-os na VMCB ("*Virtual Machine Control Block*"). Esta é composta por 27 registos dos bancos de registos dos diferentes modos de execução do processador e por 20 registos do coprocessador 15, dos quais se destacam o SCTLR, ACTLR e VBAR, assim como das TTB.

O monitor, aproveitando as características do sistema, implementa uma política de troca de mundos denominada de *lazy context-switch*. Esta política permite reduzir ainda mais o tempo de *context-switch* no geral e consequentemente reduzir o tempo de latência no serviço das interrupções FIQ por parte do RTOS. O monitor aproveita o caso de o RTOS incluído não utilizar grande parte dos modos de execução do processador realizando apenas a troca dos modos comuns aos dois SOs. Por exemplo, o monitor não salvaguarda os registos dos modos *IRQ* e *FIQ* de ambos os mundos, uma vez que estes são mutualmente exclusivos a cada partição (atribuído a cada mundo um tipo de interrupção diferente, mundo seguro -RTOS, mundo normal - GPOS).

Durante a execução do *context-switch* de mundos é realizado a mudança de estado do bit NS do registo SCR (Secure Control Register). Este bit define o ambiente de execução do processador, se é um ambiente seguro ou se é não-seguro, *i.e.*, se o processador se encontra a executar no mundo virtual seguro ou não-seguro. Ainda no mesmo registo é modificado o estado do bit FIQ, que define se o surgimento de uma interrupção FIQ desencadeia uma mudança do modo de execução do processador para o modo monitor.

Memória

Para se poder ter um sistema virtualizado, o isolamento entre as diferentes partições necessita de ser completo: isolamento temporal e isolamento espacial. O isolamento temporal, como descrito acima, é realizado recorrendo a uma politica de escalonamento *idle* ou assimétrica. O isolamento espacial, isto é, o isolamento do espaço de memória utilizada pelas diferentes partições é aplicado recorrendo a uma das propriedades da TrustZone, a configuração do estado de segurança de diferentes blocos de memória. A designação da segurança de cada bloco de memória é realizada pelo periférico TZASC. Na Figura 3.4 observa-se o particionamento da memória para cada componente do sistema (o espaço vazio encontra-se reservado para futura utilização – *e.g.,* comunicação inter-partição). No caso de o *guest* habitante no mundo normal tentar aceder à memória atribuída ao mundo seguro (configurada como memória seguro através do TZASC) irá despoletar uma exceção *Undefined*.

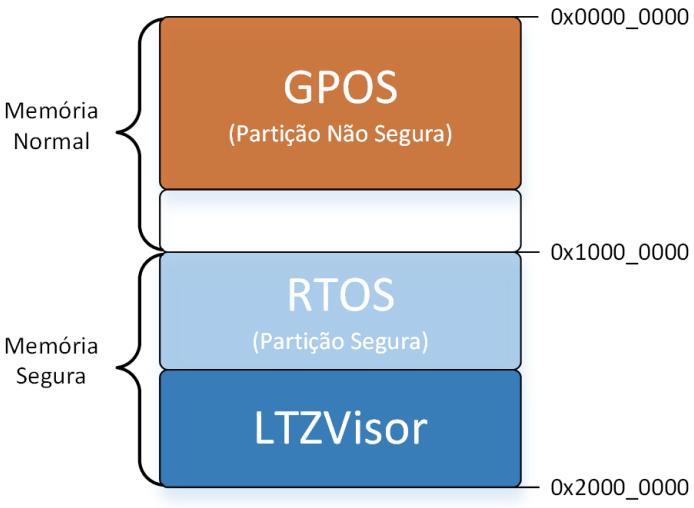


Figura 3.4: Divisão da memória em termos de segurança e partições

Nas plataformas SoC Zynq-7000, a granularidade dos blocos de memória é de 64 Mb, isto é, num espaço de memória de 1 Gb como a Zedboard é possível definir o estado de segurança de 16 blocos predefinidos de memória contígua.

Dispositivos

A tecnologia TrustZone permite que os dispositivos tenham o seu estado de segurança configurados de forma estática ou dinâmica, podendo ser considerados

seguros ou não-seguros. Desta forma, é possível realizar o particionamento dos periféricos da plataforma por hardware.

O LTZVisor implementa a virtualização de dispositivos utilizando uma política de *passthrough*. Isto significa que os dispositivos são geridos diretamente pelos próprios *guests*. A adoção desta política apenas é possível pelas características dos *guests* inclusos no LTZVisor, por não partilharem nenhum dos dispositivos do sistema. O particionamento dos dispositivos é então feito de forma direta, os dispositivos atribuídos ao RTOS são configurados como seguros e os dispositivos atribuídos ao GPOS configurados como não seguros.

Este particionamento é realizado também ao nível do GIC, sendo necessária a configuração da segurança das interrupções originárias dos dispositivos previamente particionados. O estado de segurança destas interrupções define a distribuição das mesmas para os diferentes mundos, isto é, uma interrupção configurada como segura será apenas desencadeada quando o processador se encontrar no mundo seguro e vice-versa. O monitor, como referido anteriormente, pode ainda ser configurado para fazer o *"trap"* destas interrupções.

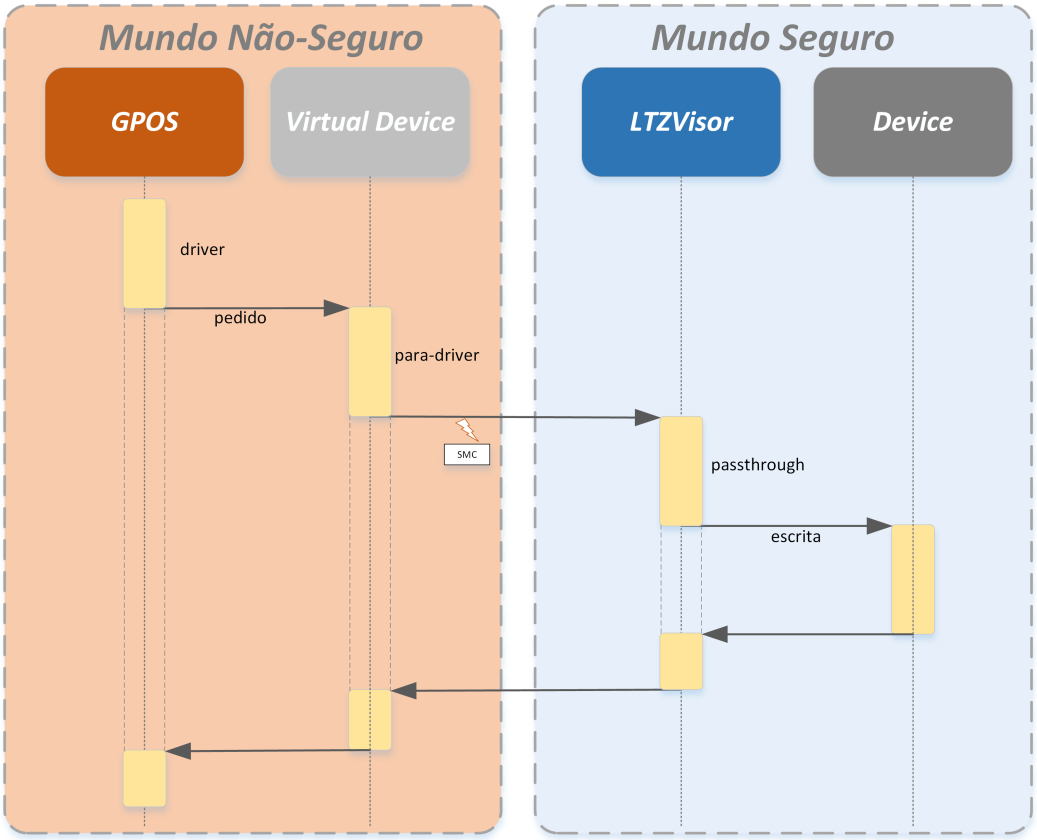


Figura 3.5: Exemplo de acesso a um dispositivo seguro no LTZVisor por parte do GPOS utilizando a técnica para-TrustZone

No entanto, a utilização da política *passthrough* não é exclusiva a todos os dispositivos. Devido à importância atribuída a certos dispositivos por parte da tecnologia TrustZone, estes não possuem um estado de segurança configurável. Dispositivos como o TTC0 e as PLLs são considerados demasiado influentes no sistema global, capazes de quebrar o isolamento do mundo seguro, e por isso são sempre considerados como dispositivos seguros.

Para que o *guest* da partição não-segura possa aceder a estes dispositivos, foi implementada uma técnica de acessos a dispositivos denominada de para-TrustZone. Esta técnica é uma junção da política *passthrough* com a técnica de para-virtualização, encontrando-se ilustrada na Figura 3.5.

O dispositivo é de igual forma gerido exclusivamente pelo GPOS, no entanto a configuração e acessos ao mesmo são realizados através do monitor. Através da utilização da instrução "smc", o GPOS faz um pedido ao monitor para que este aceda ao dispositivo diretamente. O monitor restringe a utilização desta técnica a acessos a dispositivos não partilhados com o RTOS e que estejam incluídos na VMCB do *guest* não-seguro.

3.3.1.2 Mundo Seguro

O *guest* em execução no mundo seguro na arquitetura do sistema LTZVisor deverá ser preferencialmente um RTOS. Devido à complexidade e funcionalidades reduzidas inerentes aos RTOS, trata-se de um SO usualmente mais seguro e de relativa facilidade em ser certificado. Esta característica torna-se necessária nas arquiteturas dual-OS baseadas na tecnologia ARM TrustZone, pois este *guest* deverá ser parte da TCB (*Trusted Computing Base*), uma vez que o mesmo tem controlo sobre todo o sistema. Para além desta necessidade, as características do LTZVisor, desde a política de escalonamento até ao *trap* de interrupções FIQ favorecem a integração de um RTOS no mundo seguro, contrariamente ao mundo não-seguro.

Qualquer SO integrado no mundo seguro deverá conter uma tarefa de menor prioridade, usualmente denominada de *idle task*. Esta tarefa será responsável por "escalonar" o hypervisor, que consequentemente irá realizar o *context-switch* para o mundo não-seguro, onde o outro SO irá executar. Qualquer tarefa deverá ter uma prioridade superior à da tarefa *idle*, caso contrário a mesma não será escalonada pelo sistema operativo. A *idle task* apenas deverá executar quando não restar nenhuma tarefa em estado *ready*.

Devido às características da *framework* LTZVisor, as tarefas de tempo-real do RTOS têm os seus requisitos de tempo-real protegidos em relação a interferências

do sistema virtualizado. O tempo que uma rotina de serviço a uma interrupção terá de esperar até ser executada será definido pelo seu WCET (*worst case execution time*). Essa latência será de 2 vezes o tempo de um *context-switch* entre mundos, como demonstrado pela Figura 3.6. Este caso ocorrerá quando uma interrupção ocorrer imediatamente a uma chamada do monitor por parte da *idle tasks*. Excetuando o caso de estar a executar no seu próprio ambiente de execução (mundo seguro) onde essa latência é dada pelos próprios mecanismos e políticas de escalonamento do próprio RTOS.

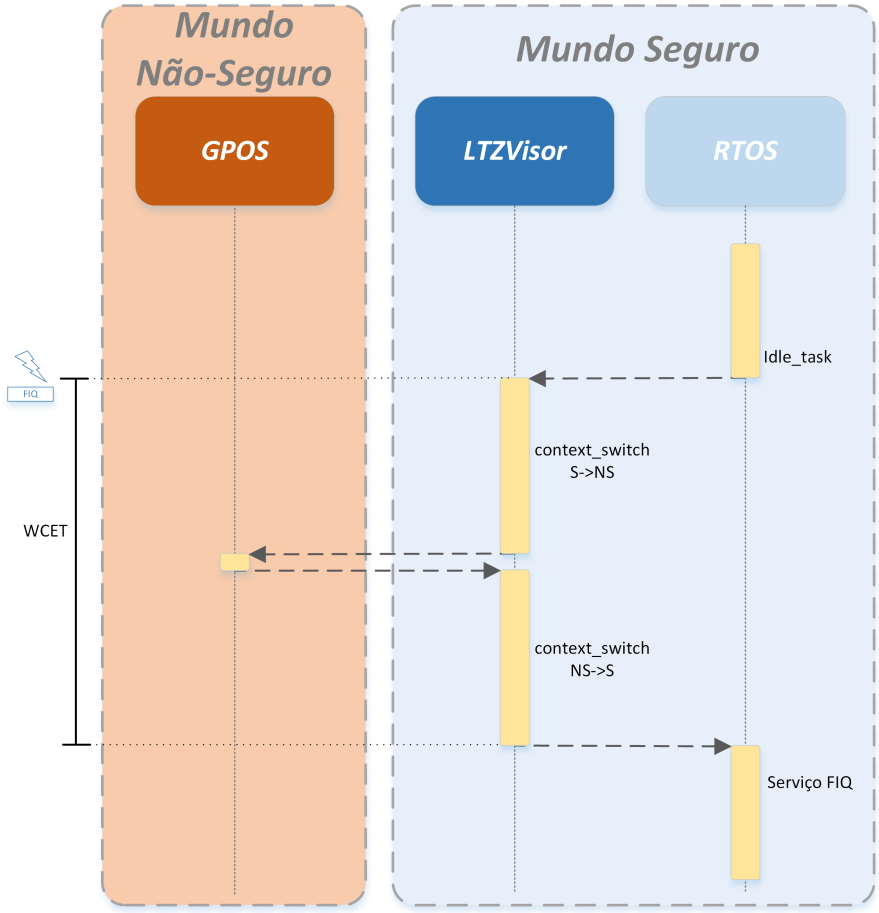


Figura 3.6: *Worst Case Estimated Time* (WCET) da latência de serviço a uma FIQ

Na implementação do LTZVisor, foi escolhido o sistema operativo de Tempo Real FreeRTOS para executar no mundo seguro, tendo sido realizado o *porting* da sua versão 7.0.2. Trata-se de um RTOS desenhado para plataformas de recursos escassos, escrito maioritariamente em C, sendo por isso consequentemente um SO de alta portabilidade (o seu Kernel mínimo é composto por 3 ficheiros, task.c, list.c e port.c). Este RTOS aplica uma política de escalonamento baseada em prioridades (com reentrância), sendo que em casos de prioridades iguais o RTOS

segue um modelo de escalonamento *round-robin*. A popularidade deste SO deve-se sobretudo à sua vertente *open-source* e desenvolvimento ativo, assim como à sua compactação, fiabilidade, e fácil portabilidade.

3.3.1.3 Mundo Não-Seguro

No mundo não-seguro poderá executar qualquer um SO de qualquer classe (GPOS ou RTOS). No entanto, de maneira a aproveitar as funcionalidades do sistema e da própria plataforma aconselha-se o uso de um GPOS, um SO capaz de tirar proveito das funcionalidades oferecidas por plataformas de propósito geral (contendo processadores de performance – Cortex-A9). Para além da potencialização da plataforma, o GPOS é, por norma, um sistema operativo que lida bem com a perda de interrupções e com o não cumprimento de prazos de tempo-real, algo crucial, dadas as características acima referidas da *framework* LTZVisor.

Um GPOS caracteriza-se pela sua capacidade em aproveitar todas as funcionalidades de uma plataforma com vista a facultar ao utilizador a habilitação de executar qualquer tipo de programa. A utilização inata de propriedades gráficas e correspondente interface gráfica com o utilizador é uma das particularidades mais apelativas dos GPOS. Outra peculiaridade dos GPOS, que os torna imensamente populares, é a sua capacidade em executar programas transferidos de outras fontes, quer devido à sua enorme camada de abstração como à sua facilidade em conexão ao mundo exterior (habilitando a descarga destes programas).

A *framework* LTZVisor integra na sua implementação a versão 3.3 do SO Linux. O Linux é um dos mais influentes sistemas operativos no mundo de hoje, obtendo um impacto ainda maior na área de sistemas embebidos. Um SO baseado em Unix maioritariamente escrito em C, excetuando os ficheiros de dependentes da arquitetura onde se observa um misto de linguagem C com Assembly. O seu conceito de *free open-source* assim como as inúmeras contribuições, tanto por parte de grandes empresas como simples utilizadores, destacaram a sua popularidade entre os sistemas operativos de propósito geral. Em relação ao *guest* do mundo seguro destaca-se a sua gestão de memória avançada (através do uso do periférico MMU), a utilização de *device drivers*, o uso de contas de utilizador e a sua conetividade.

Devido ao ambiente de execução em que se encontra inserido, o mundo não-seguro, o Linux encontra-se de certa forma limitado, tanto no uso de periféricos externos como no uso da própria memória. Devido à ação do TZASC, que configura parte da memória como segura, e ao particionamento de dispositivos, o Linux deve ter a sua ação restringida aos recursos atribuídos ao mundo não-seguro, para que não incorra em exceções *Undefined* em acessos não autorizados. Deve-se ter esta

restrição em conta na compilação do Linux para as zonas de memória atribuídas e na compilação da respetiva *Device Tree*, que contém os dispositivos atribuidos ao sistema operativo Linux.

3.3.2 Comunicação OpenAMP

A implementação futura da *framework* prevê o uso de uma configuração *multicore* ou *single-core*. A implementação de um mecanismo de comunicação entre os *guests* incorporados na *framework* deverá suportar ambas as configurações referidas e ainda antever um certo nível de segurança coerente com o género de *framework* na qual está inserida. Aparte destes requisitos funcionais deverá ter-se em conta o suporte para uma possível futura configuração da *framework* envolvendo *multiguest*, isto é, mais de dois *guests*.

Na Secção 2.4 foram analisadas algumas abordagens de comunicação inter-partição standardizadas no domínio embebido. Entre os mecanismos analisados, a comunicação inter-*core* inter-*guest* providenciada na *framework* do MEMF, uma implementação comercial da especificação OpenAMP, foi a que mais se assemelhou à comunicação requerida, destacando-se as seguintes características:

- **Comunicação *multicore***. Um dos requisitos do mecanismo de comunicação a implementar é a de compatibilidade com a configuração *multicore* implementada, recaindo por isso a escolha numa comunicação nativamente inter-*core*.

- **Escalabilidade**. A presença das funcionalidades da especificação do OpenAMP em *guests* em ambientes virtualizados ou ambientes *multicore* supervisionados, demonstram que a tecnologia pode ser adaptada para o sistema virtualizado LTZVisor. Adicionalmente, a possibilidade de comunicação entre mais de um par de *guests* provam que uma implementação similar seria escalável para uma futura configuração *multiguest* do LTZVisor;

- **Suporte para os *guests* integrados na atual configuração do LTZVisor, Linux e FreeRTOS**. Apesar de possuírem funcionalidades reduzidas comparativamente com as apresentadas no MEMF, as implementações *open-source* disponíveis do OpenAMP para o FreeRTOS e das próprias drivers do Linux foram um ponto de partida avançado para a implementação de mecanismos de comunicação inter-partição. Esta última característica permite otimizar o tempo de implementação dos mecanismos de comunicação através da utilização de COTS (*commercial off the shelf*).

- **Suporte para uma arquitetura supervisionada**. O suporte para uma arquitetura virtualizada pode ser observada na implementação da comunicação no hypervisor da Mentor Graphics.

Nos próximos subcapítulos serão descritas as várias componentes integradas na comunicação especificada pelo OpenAMP, utilizada na *framework* MEMF e com vista a serem implementadas na comunicação inter-*core* do sistema LTZVisor.

3.3.2.1 Remoteproc

A tecnologia Remoteproc tem dois modos de execução: modo *master* e modo *slave*. No modo *master* o Remoteproc permite que um sistema operativo tenha a capacidade de gerir o ciclo de execução de um *core* pertencente à plataforma na qual se insere. O SO ganha ainda a capacidade de descarregar o *firmware* que será executado nesse mesmo *core*.

No modo *slave* assim como no modo *master* o Remoteproc tem como função a configuração e inicialização das restantes partes integrantes do sistema de comunicação inter-*core*, nomeadamente o RPMsg e o dispositivo virtual VirtIO.

Para que qualquer destas tarefas seja corretamente executada, o Remoteproc, a executar em modo *master*, recorre a uma estrutura estática, denominada de *resource_table*, presente no *firmware* que irá ser descarregado no *core slave*. Esta estrutura deverá ser retirada do *firmware* e usada para obter informações importantes para a configuração do mecanisms de comunicação RPMsg assim como para a descarga do próprio *firmware*. Na Listagem 3.1 abaixo estão alguns dos componentes presentes nessa estrutura.

Listagem 3.1: Estrutura resource_table

```
1 /* Resource table for the given remote */
2 struct remote_resource_table {
3 unsigned int version;
4 unsigned int num;
5 unsigned int reserved[2];
6 unsigned int offset[8];
7
8 /* rpmsg vdev entry */
9 struct fw_rsc_vdev rpmsg_vdev;
10 struct fw_rsc_vdev_vring rpmsg_vring0;
11 struct fw_rsc_vdev_vring rpmsg_vring1;
12 };
13
14 /* rpmsg VirtIO device */
```

```
15 struct fw_rsc_vdev {
16 unsigned int type;
17 unsigned int id;        //VirtIO device id
18 unsigned int notifyid;  //notification id
19 unsigned int dfeatures; //device features
20 unsigned int gfeatures; //place holder
21 unsigned int config_len;
22 unsigned char status;
23 unsigned char num_of_vrings;    //number of vrings
24 unsigned char reserved[2];
25 struct fw_rsc_vdev_vring vring[0]; //array of vrings
26 } __attribute__((__packed__));
27
28 /* VirtIO vring */
29 struct fw_rsc_vdev_vring {
30 unsigned int da;        //device address
31 unsigned int align;     //alignment between consumer and
      producer parts
32 unsigned int num;       //number of buffers supported
33 unsigned int notifyid;  //id of the notification
34 unsigned int reserved;
35 } __attribute__((__packed__));
```

No sistema operativo Linux, o componente Remoteproc é o único com conhecimento da plataforma em que está inserido, necessário para que seja realizada a gestão dos *cores* da plataforma. Aproveitando este fator, o Remoteproc será ainda responsável por configurar as especificidades da plataforma nos restantes componentes (RPMsg e VirtIO), como o método de notificações usado no canal de eventos da comunicação.

3.3.2.2 VirtIO

O componente VirtIO é uma abstração da camada de transporte usado em comunicação. O VirtIO faz a gestão da memória partilhada, ao nível dos dados de controlo assim como ao nível da utilização dos buffers para partilha de dados. Foi originalmente criado para comunicação inter-VM (hypervisor e *guest*) com o propósito de servir para-drivers, no entanto o seu âmbito foi amplamente ultrapassado, sendo inclusivamente usado em comunicação inter-*core* por componentes como RPMsg ou comunicação inter-*guests* [24].

Atualmente existem pelo menos oito hypervisores do tipo 2 suportados pelo sistema operativo Linux, isto é, oito sistemas de virtualização diferentes que executam dentro do contexto de um processo do Linux. Cada um desses sistemas

providencia a sua própria implementação de drivers para para-virtualização das diversas variantes de dispositivos (*e.g.,* consola, bloco, *network*, etc.). O VirtIO foi criado para se tornar um standard no mundo da para-virtualização dentro do SO Linux, criando uma unificação de drivers que pudessem providenciar uma ABI (*"Application Binary Interface"*) comum para a utilização das para-drivers. Isto é, utilizando a camada de abstração VirtIO, poder-se-ia utilizar as mesmas para-drivers em qualquer hypervisor. As para-drivers estão divididas em duas partes: back-end drivers, pertencentes ao hypervisor e responsáveis por emular o dispositivo e aceder ao mesmo. E *front-end* drivers, pertencentes ao *guest*, responsáveis por comunicar com o hypervisor para a realização das tarefas pretendidas sobre o dispositivo em questão. A Figura 3.7 demonstra a hierarquia das drivers de para-virtualização utilizando o VirtIO.

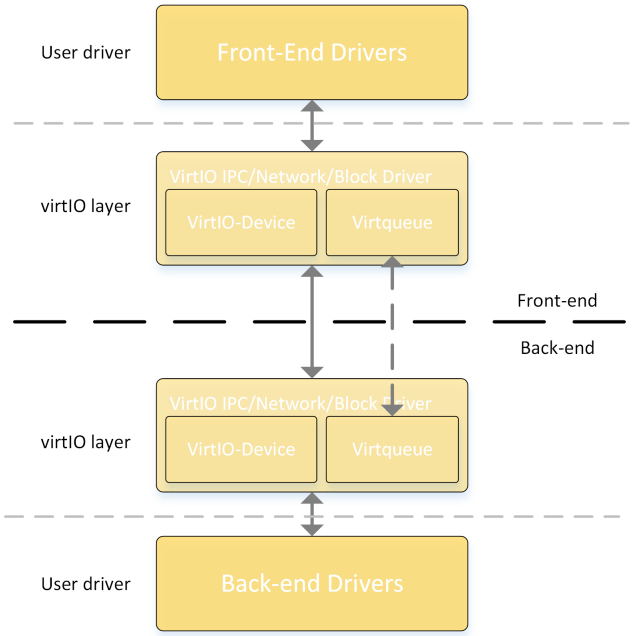


Figura 3.7: Hierarquia de drivers de para-virtualização utilizando o VirtIO

O VirtIO proporciona duas abstrações chave para a implementação das para-drivers: i) abstração do dispositivo e ii) abstração da camada de transporte denominada de *virtqueue*.

A abstração do dispositivo, chamado de *VirtIO_device*, surge através da criação de um dos dispositivos virtuais genéricos à disposição (entre network, block, pci, consola, rpmsg, etc.). Este dispositivo virtual tem a formação base de um dispositivo real, sendo posteriormente necessário às *user front-end* drivers informar as *back-end* drivers das configurações requeridas (*e.g.,* configurar uma porta-séria com bit de paridade, etc.) [30].

A abstração da camada de transporte é realizada através do uso de buffers circulares em memória partilhada (*vrings*), sendo esta uma abordagem normalizada entre os diferentes hypervisores criando uma menor resistência à sua aderência. O VirtIO providencia às *front-end* e *back-end* drivers uma API para o uso da camada de transporte denominada de *virtqueue*, assim como para o uso de notificações a ambas as partes. Como pode ser observado na Figura 3.7, a *virtqueues* ligam conceptualmente as *front-end* drivers às *back-end* drivers.

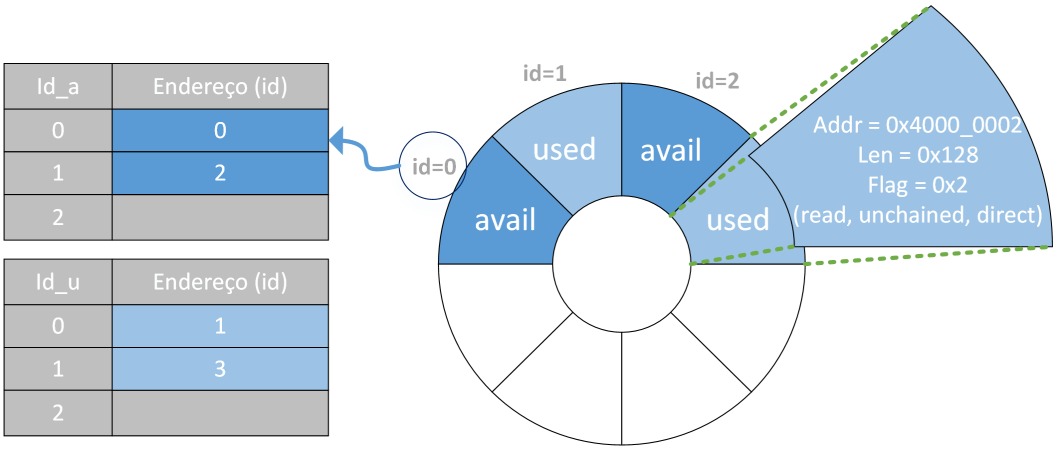


Figura 3.8: Arrays circulares da camada de transporte virtqueue do VirtIO

A camada de transporte do VirtIO pode ser descrita como uma série de apontadores e arrays descritores em memória partilhada. Simplificando o funcionamento da camada de transporte, pode-se resumir o mecanismo de passagem de dados a três arrays circulares. Os arrays são denominados de: *descriptors*, *available* e *used* (respetivamente descritores, disponíveis e usados). A Figura 3.8 demonstra o relacionamento entre os três arrays, em que o descritor está representado como um buffer circular contendo a informação chave. Os restantes arrays (também circulares) são arrays auxiliares da comunicação. Para os diferentes utilizadores do VirtIO (o *guest* - o *master* da comunicação - e o hypervisor - correspondente *slave* da comunicação), os arrays têm um significado diferente:

- ***Descriptor Array*** – apenas pode ser escrito pelo *master* da comunicação. Contém a descrição dos buffers: a) o endereço do buffer na memória partilhada, b) o tamanho do buffer, e ainda c) *flags* que descrevem algumas das características do próprio buffer (leitura ou escrita, endereço direto ou indireto, e se faz parte de uma cadeia de buffers ou não);

- ***Available Array*** – apenas pode ser escrito pelo *master*. Contém informação sobre os buffers considerados disponíveis. Isto é, contém o id da

posição onde esses buffers se encontram no array descritor. Do ponto de vista do *master*, os buffers são considerados disponíveis se já foram usados por ele mesmo (escritos ou limpos) e estão agora prontos para serem usados pelo *slave*. Do ponto de vista do *slave*, contrariamente, os buffers são considerados disponíveis se os mesmos contiverem dados prontos a ser lidos ou se encontram prontos a serem utilizados para a escrita de dados;

- ***Used Array*** – apenas pode ser escrito pelo *slave*. De forma idêntica ao array de buffers disponíveis, este irá ser um array mas de ids dos buffers usados no array descritor. Os buffers são considerados usados se foram usados em ultimo lugar pelo *slave*, isto é, se o mesmo já leu os dados ou se neles escreveu alguma coisa, estando agora prontos a serem reciclados pelo *master* (após a leitura, se requerido).

O VirtIO permite uma comunicação bidirecional sobre o mesmo trio de arrays descritores (*vring*). No entanto, o fluxo é sempre idêntico ao supramencionado. O *master* poderá disponibilizar o buffer vazio (para receber uma mensagem) ou colocar o buffer preenchido já com os dados a enviar, preenchendo as *flags* de acordo (escrita ou leitura).

A Figura 3.9 ilustra o fluxo da comunicação na camada de transporte VirtIO. O *master* da comunicação, obtém os buffers diretamente da memória partilhada, colocando-os no array de descritores após uso. O array de buffers disponíveis será atualizado com o "*id*" desse buffer no array descritor ("*id*" 2 na Figura 3.9). Após a atualização dos buffers é realizada uma barreira de memória (para que o hypervisor leia o valor atualizado nos arrays) e é enviado uma notificação. A notificação irá despoletar uma leitura do "id" ao array dos buffers disponíveis por parte do hypervisor. Esse id revelará a posição das informações do buffer no array de descritores. O hypervisor irá, de seguida, realizar as ações pretendidas, sejam elas ler o buffer recebido ou escrever nele. Posteriormente, o "*id*" desse buffer será passado para o array de usados. Numa fase posterior, o *guest/master* irá buscar os buffers ao array de buffers usados, para leitura ou simplesmente para fazer a sua reciclagem.

Por se tratarem de arrays circulares, o seu tamanho, isto é, o número de buffers que conseguem descrever deve ser uma potência de 2. Isto não significa, no entanto, que a zona de memória partilhada deva conter esse número de buffers, uma vez que os buffers são inseridos explicitamente pelo *master* na forma de descritores. Esta particularidade prevê também a possibilidade de alocação dinâmica do espaço de memória partilhada ou até que a mesma não seja contígua.

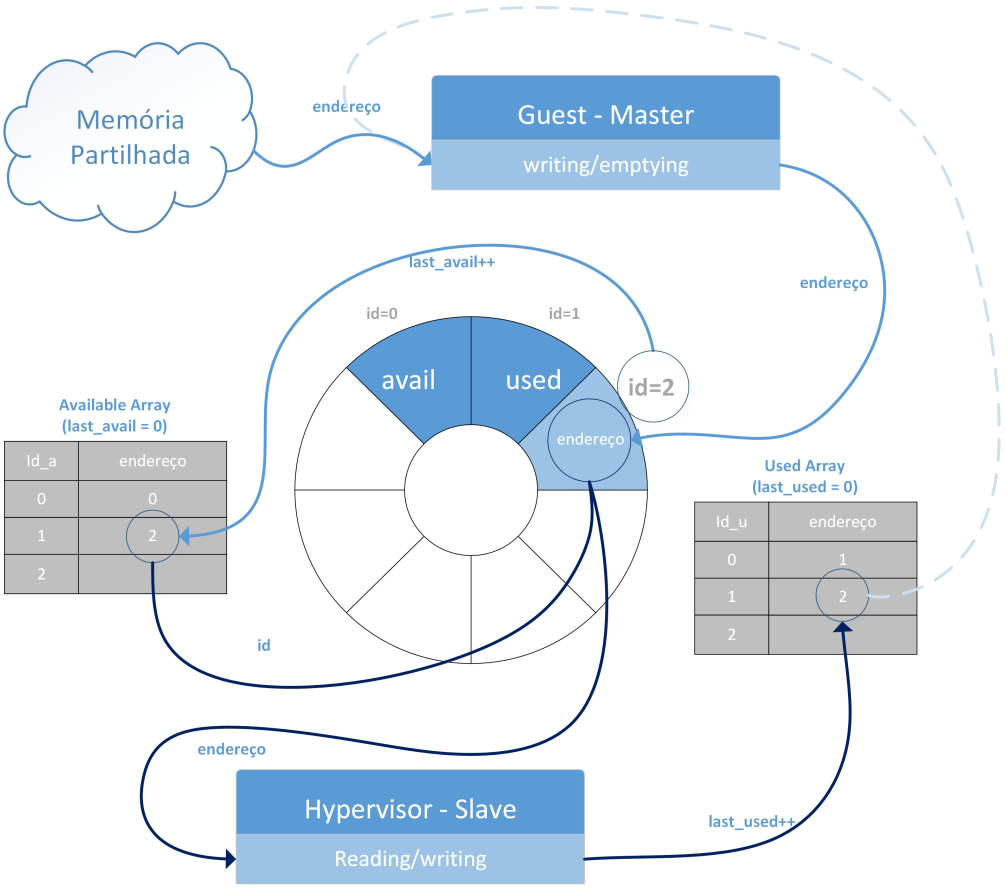


Figura 3.9: Fluxo de uma transferência de dados numa Virtqueue

De ressalvar que a estrutura do VirtIO permite a comunicação inter-*core* sem a necessidade de mecanismos de sincronismo inter-domínio. Esta característica deve-se ao uso de barreiras de memória e ao uso de buffers circulares *single-writer-single-reader*, em que a cada array apenas são dadas permissões de escrita (por parte do driver, não do *hardware* em si) a um dos participantes na comunicação permite. A necessidade de integração de mecanismos de sincronização ou exclusão mútua representariam um desafio enorme, tanto à implementação como à eficiência da comunicação em si. As barreiras de memória, no caso específico do processador em uso (ARM Cortex-A *series*), permitem que qualquer acesso à memória realizado antes da barreira seja finalizado, impedindo leituras (posteriores à barreira) dessa memória com valores inalterados e errados. No caso dos arrays circulares, o *slave* da comunicação (hypervisor numa comunicação inter-VM) terá acesso apenas ao array de buffers usados, sendo os restantes "propriedade" do *master*. O uso de variáveis internas para a atualização do estado do array por parte do participante sem permissão de escrita é um dos fatores que ajuda nesta característica importante da comunicação.

Como descrito o VirtIO é um mecanismo muito atrativo à implementação

de comunicação, seja ela inter-VM, inter-*guest* ou inter-*core*, devido às suas propriedades genéricas, à sua eficiência e ainda pela diversidade de opções disponíveis, tornando-se o mecanismo perfeito para a *"standardização"* da abstração da camada de transporte.

3.3.2.3 RPMsg

A implementação original do VirtIO apenas contemplava a utilização deste mecanismo de abstração para providenciar comunicação entre as drivers *back-end* e as drivers *front-end*. No entanto, devido às suas propriedades genéricas e à sua eficiência, o VirtIO foi utilizado pelo RPMsg para servir de camada de transporte à sua comunicação inter-*core*. Estando a implementação do VirtIO dividida em hypervisor e *guest*, o RPMsg dividiu o seu protocolo de comunicação em modo *master* e *slave*, sendo atribuído ao modo master o mesmo papel que seria atribuído a um *guest*, e ao modo *slave* o papel que seria atribuído a um *guest* na comunicação original inter-VM. Foi criada na interface VirtIO um *VirtIO_device* do tipo *rpmsg_device* cuja configuração ficará a cargo do Remoteproc (driver com informação específica sobre o *hardware*).

Apesar de o VirtIO contemplar uma comunicação bidirecional num só conjunto de arrays e apontadores (*vring*), o RPMsg instancia dois *vrings* para realizar uma comunicação unidirecional em cada um deles (bidirecional no total). Esta característica permite uma melhor divisão entre o significado das notificações. Tratando-se de um *vring* bidirecional, a notificação despoletada pelo *master* pode representar no *slave* a recente disponibilização de um buffer para escrita como de uma mensagem recebida. Tornando o *vring* unidirecional, cada notificação irá ter apenas um significado, usualmente o de mensagem recebida, podendo-se inclusivamente "desligar" as notificações de buffer disponível (apenas necessária em casos específicos). Esta divisão permite que a comunicação surja a qualquer altura por qualquer uma das partes, sendo apenas necessário ao *master* disponibilizar a totalidade dos buffers para o *vring* do *slave* na fase de inicialização.

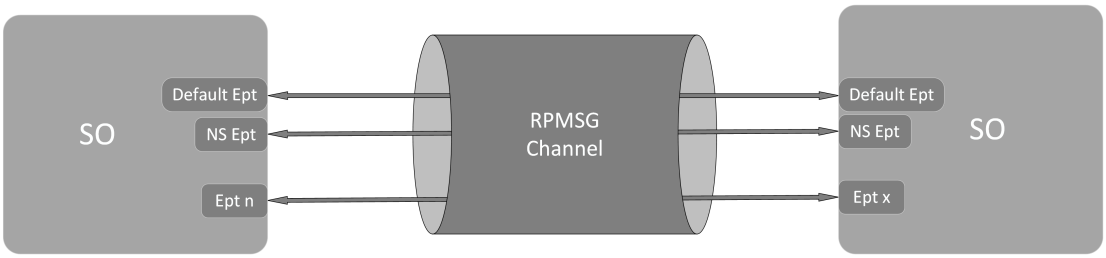


Figura 3.10: Canal RPMsg e respetivos endpoints

O RPMsg representa um protocolo de comunicação não persistente ponto-a-ponto. Isto é, a comunicação é feita entre dois pontos, cada um com funcionalidades diferentes (apenas ao nível da camada VirtIO e *handshake*). Cada ligação ponto-a-ponto é denominada de RPMsg *Channel* (canal). Cada canal tem vários pontos de ligação, denominados de *endpoints*. Tratando-se de uma comunicação assíncrona, esta divisão de cada ponto em vários *endpoints* irá permitir que cada um contenha o seu próprio *callback*, possibilitando assim a comunicação entre diversas aplicações num mesmo canal. Estes dois conceitos introduzidos pela tecnologia RPMsg encontram-se visíveis na Figura 3.10 e definidos abaixo:

- **Canal RPMsg** - Uma comunicação bidirecional entre duas partes no mundo RPMsg é chamada de canal. O canal é caracterizado pelo seu nome e pelo endereço atribuído a cada um dos pontos da comunicação. Um canal pode conter um ou mais *endpoints*. Usualmente um canal contém sempre dois *endpoints*: o *endpoint* por defeito, para que a comunicação possa começar imediatamente, e o *endpoint* de Name Service announcement (NS) usado pelo mecanismo de *handshake*. A Figura 3.10 demonstra a representação gráfica de um canal e respetivos *endpoints*.

- **RPMsg *Endpoint*** - Um endpoint é uma abstração lógica em cada canal que proporciona a infraestrutura para o envio e receção de mensagens. Cada endpoint terá o seu próprio endereço, normalmente relacionado com o endereço do canal em que está inserido. O endpoint contém também o seu próprio callback, permitindo que várias aplicações dentro do mesmo canal possam realizar comunicações diferentes.

A comunicação RPMsg, como referido anteriormente, trata-se de uma comunicação assíncrona, apresentando um canal de dados separado do canal de eventos. Para o canal de dados é utilizada a camada de abstração VirtIO, enquanto que para os eventos são utilizadas interrupções inter-*core* (no caso do processador ARM em uso serão as *Inter Processor Interrupts*, IPIs, também denominadas de *Software Generated Interrupts*, SGIs). O protocolo RPMsg define a utilização de duas interrupções por *vring* (por direção de comunicação), uma para aviso de receção de mensagem e outra para aviso de leitura completa ou buffer disponível para reciclagem.

Visto o protocolo de comunicação estar intrinsecamente ligado ao Remoteproc, existe uma necessidade de sincronismo entre os canais devido à gestão de ciclo dos *cores slave*. O sincronismo é realizado às custas de uma espécie de *handshake*. O *handshake* permite ainda que as partes tenham conhecimento do endereço uma

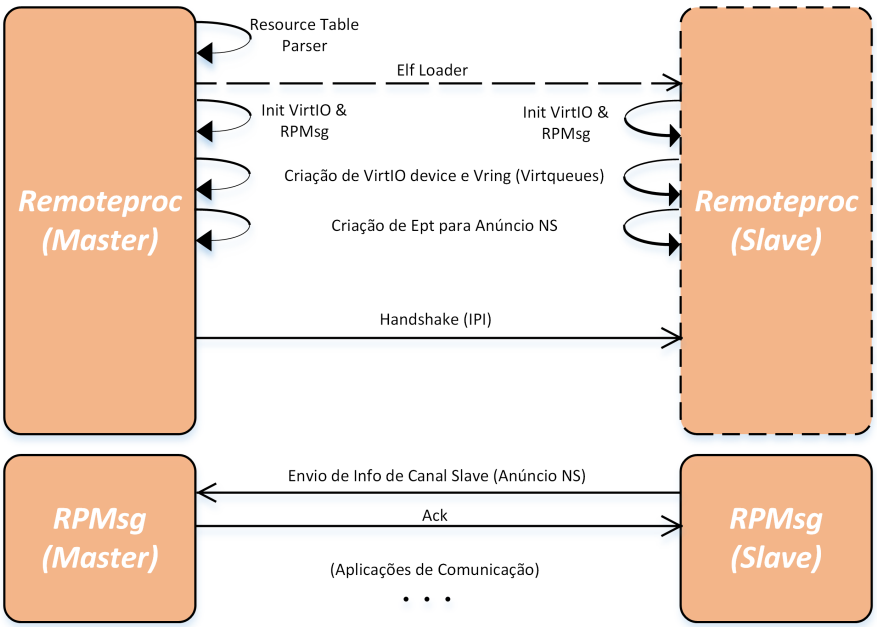


Figura 3.11: *Handshake* numa comunicação Remoteproc/RPMsg

das outras. O *handshake* pode ser observado na Figura 3.11, não se tratando de mais do que uma interrupção inter-*core* seguido de (se o *core* estiver 'acordado') um serviço de anunciação de nome (NS). Este serviço permite que ambos os *cores* participantes tomem conhecimento do endereço de cada um, habilitando a criação de *endpoints* através do *callback* associado à criação do canal.

Uma comunicação não persistente tem a vantagem de poder realizar transferências de dados de forma não ordenada e para diferentes *endpoints* ao mesmo tempo, não sendo necessário ocupar o canal com uma conexão entre dois *endpoints*. No entanto, para que isto seja possível é necessária a inserção de um *header* em cada mensagem enviada. A Listagem 3.2 apresenta os integrantes num *header* de uma mensagem enviada numa comunicação RPMsg. O *header* introduz algum *ovehead* na comunicação, ao nível de memória (o espaço que ocupa no buffer) e ao nível de tempo na sua criação na origem e descodificação no destino.

Listagem 3.2: Estrutura do *header* de uma mensagem RPMsg

```
1 struct rpmsg_hdr {
2 unsigned long src;      //addr of the source
3 unsigned long dst;      //addr of the destination
4 unsigned long reserved;
5 unsigned short len;     //length of the payload
6 unsigned short flags;   //message flags
7 unsigned char data[0];  //len bytes of message payload
8 } __attribute__((packed));
```

Integração

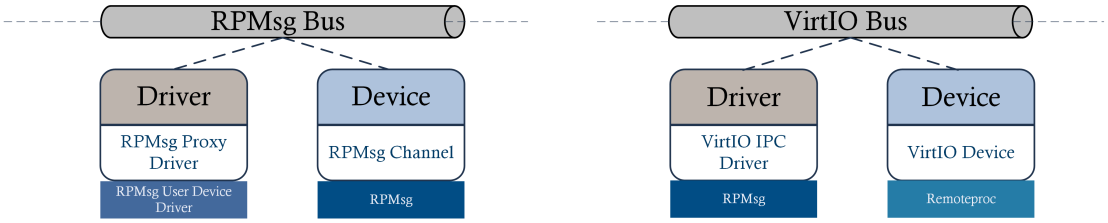


Figura 3.12: Relação entre drivers e devices no OpenAMP

Na Figura 3.12 é possível observar a relação entre as partes integrantes da especificação OpenAMP do ponto de vista da comunicação. O Remoteproc é responsável pela configuração do dispositivo virtual VirtIO (*rpmsg_device*) pois obtém da *resource_table* os valores da memória partilhada para a sua implementação, assim como das opções da plataforma para a implementação das notificações. Do ponto de vista do RPMsg, cada canal será um dispositivo virtual diferente, pois trata-se de um ponto de comunicação diferente, existindo uma necessidade de um tratamento diferente, recorre-se por isso a drivers específicas desses mesmos canais. O RPMsg cria a driver responsável pela interface com o VirtIO que irá fornecer as APIs de acesso à comunicação às drivers específicas de cada canal.

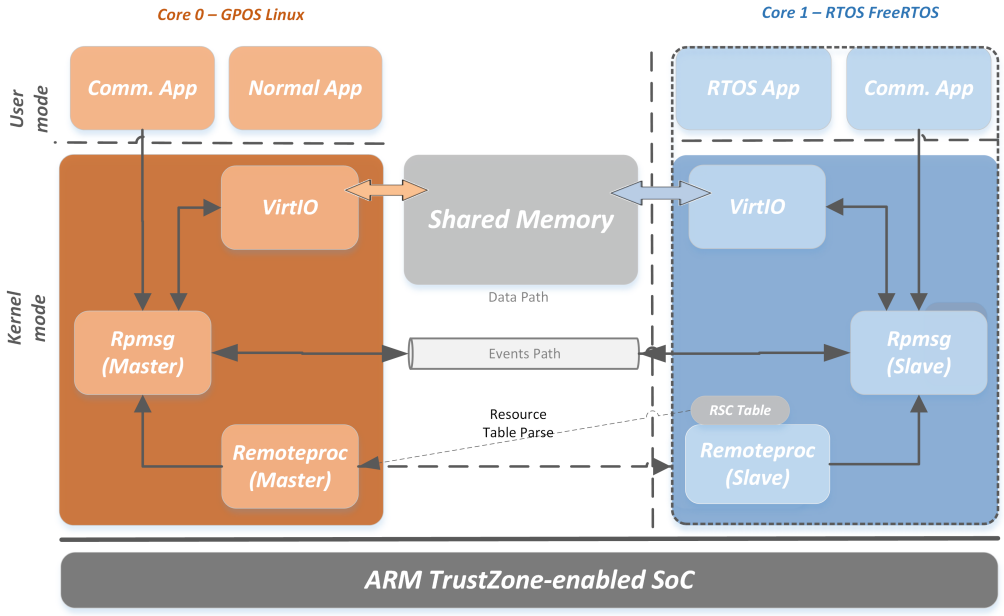


Figura 3.13: Arquitetura de dois SOs utilizando OpenAMP

A figura Figura 3.13 representa a arquitetura de dois SOs utilizando a totalidade das ferramentas providenciadas pelo OpenAMP. O GPOS Linux, inicialmente em configuração SMP, passa para uma configuração *single-core* abdicando

da execução no *core* 1, realizando posteriormente a descarga do RTOS para esse mesmo *core* após ler a sua *resource_table*. Subsequentemente, após a inicialização do *core* 1, os *cores* estarão possibilitados de comunicar entre si assincronamente. A qualquer momento, o modo *master* do Remoteproc, o Linux, é capaz de parar a execução do *core* 1 por parte do *firmware slave* para voltar a utiliza-lo para execução própria. A comunicação como referido anteriormente utiliza memória partilhada gerida pelo VirtIO e um canal de eventos imposto pelo RPMsg.

A especificação OpenAMP trata-se de uma ferramenta bastante interessante para a implementação de interação e comunicação inter-*core* entre *guests* de variadas classes de sistemas operativos. A sua eficiência e capacidade de escalabilidade cimentam a sua posição no mundo da tecnologia embebida *multicore*. Esta tecnologia foi já adotada para arquiteturas diferentes da originalmente concebida como demonstrado na *framework* MEMF.

LTZVisor

Neste capítulo será apresentado o trabalho realizado sobre a *framework* LTZVisor, nomeadamente a expansão da *framework* para uma configuração *multicore* e a adição do suporte para uma comunicação inter-partição. Inicialmente será feita uma análise estrutural da própria *framework* assim como de algumas alterações estruturais necessárias para a conclusão da dissertação. Posteriormente serão analisados os detalhes de implementação do trabalho proposto.

4.1 LTZVisor - Single-core

A *framework* LTZVisor, como referido anteriormente, assume uma arquitetura dual-OS, com duas partições situadas em cada mundo virtual proporcionado pela tecnologia TrustZone num mesmo core. Neste subcapítulo será feita a análise estrutural da *framework* assim como a análise às alterações realizadas à sua configuração standard. Isto é, alterações promovidas para atualizar a *framework* tendo em vista uma futura adição de configurações à mesma (nomeadamente a configuração *multicore* com suporte para a comunicação inter-partição).

4.1.1 Análise Estrutural

Estruturalmente a *framework* do LTZVisor encontra-se dividida em 6 partes, estando cada uma dessas partes associada a uma pasta no diretório principal do projeto, abaixo apresentadas:

- **arch** – A portabilidade do LTZVisor é garantida através da manutenção dos ficheiros deste subdiretório. Nela encontrar-se-á o código *machine-dependent* (código *low-level*), *i.e.*, o código dependente da arquitetura do processador no qual será executada a *framework*. O LTZVisor, como se vê através da Figura 4.1, tem suporte para a arquitetura ARMv7, especificamente para os processadores Cortex-A9 (Zynq-7000) e Cortex-A15 (utilizado na VExpress, simulada pelo Fast Models, devido a erros conhecidos do processador A9 no mesmo modelo de simulação). Este código inclui as inicializações da plataforma, incluindo a inicialização das *stacks*, MMU e caches, e ainda o

código do *context-switch* entre mundos. Na Listagem 4.1 o demonstra-se a inicialização do GIC (controlador de interrupções) assim como a configuração da segurança de algumas das interrupções utilizadas por ambos os *guests* do sistema, depende do processador escolhido;

Listagem 4.1: Inicialização do GIC

```
1 ret = interrupt_distributor_init();
2 if(!ret)
3         goto cpu_init_ret;
4 interrupt_interface_init();
5
6 #ifdef CONFIG_CORTEX_A15        /* Cortex A15*/
7   interrupt_security_config(UART_1_INTERRUPT,Int_NS);
8   interrupt_security_config(TIMER_0_INTERRUPT,Int_NS);
9   interrupt_security_config(UART_2_INTERRUPT,Int_S);
10  interrupt_security_config(TIMER_1_INTERRUPT,Int_S);
11 #else                            /* Cortex A9*/
12  interrupt_security_config(UART_1_INTERRUPT,Int_NS);
13  interrupt_security_config(GLOBAL_TMR_INTERRUPT, Int_NS);
14  interrupt_security_config(TTC1_TTCx_2_INTERRUPT,Int_S);
15 #endif
```

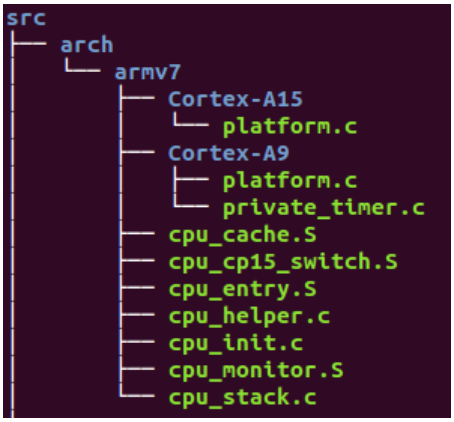


Figura 4.1: Árvore da Subpasta *Arch*

- **core** – Neste directório está incluído o código pertencente ao monitor que não é dependente da plataforma em que está inserido. Nesta divisão encontra-se a API de gestão de *guests*, nomeadamente da cópia da imagem do *guest* não-seguro, e da passagem de controlo do processador para o *guest* do mundo seguro;

- **drivers** – Apesar de pertencer ao código de portabilidade do sistema, as drivers dos diferentes periféricos são incluídas nesta pasta, uma vez que diferentes plataformas usam o mesmo periférico, assim como processadores idênticos (de diferentes plataformas) utilizam periféricos diferentes. A separação entre código *machine-dependent* e *platform-dependent* é necessária para evitar a duplicação de código e possíveis erros de portabilidade. Existem subpastas dedicadas às drivers de cada plataforma (na Figura 4.2 observa-se o suporte para a plataforma VExpress (VE) e para as plataformas contendo o SoC Zynq-7000, nomeadamente a plataforma Zedboard e ZC702), existindo no entanto uma subpasta de drivers de periféricos comuns às plataformas suportadas na implementação do LTZVisor (e.g., GIC - *Generic Interrupt Controller*, periférico partilhado pelas plataformas acima referidas). Nesta pasta é ainda incluída a inicialização da segurança da *board*, uma vez que apesar de ser comum a todos os processadores, muitas das características da tecnologia TrustZone são *implementation-defined* pelo fabricante da própria *board*. No excerto de código abaixo verifica-se algumas das inicializações de segurança, como a configuração do TZASC, uma configuração definida pelo produtor do SoC (nomeadamente a sua granularidade – na Zynq-7000 esta é de 64Mb) assim como pela própria plataforma (através da memória disponível na plataforma – na plataforma Zedboard é de 2Gb). Este diretório incluí ainda o *linker script* também pela última razão referenciada;

Listagem 4.2: Configurações de segurança iniciais

```
1 write32((void *)SLCR_UNLOCK, SLCR_UNLOCK_KEY);
2
3 /*Base Filter Address*/
4 write32((void*)0xF8F00040, 0);
5
6 /* Configure all memory non-secure for this trial*/
7 write32((void *)TZ_DDR_RAM, 0xFFFFFFFF);
8 write32((void *)TZ_DDR_RAM-0x30, 0xFFFFFFFF);
9
10   (...)
```

- **lib** – De forma a manter o código da *framework* relativamente curto (torna a sua verificação e consequentemente certificação mais fácil), foi evitado o uso das bibliotecas standard ANSI C (libc). Em vez da sua inclusão foram replicadas unicamente as funcionalidades utilizadas pelo próprio LTZVisor e pelo *guest* do mundo seguro (parte da TCB). A Figura 4.3 demonstra

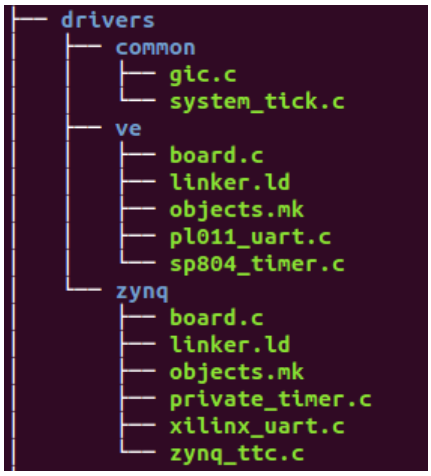


Figura 4.2: Árvore da Subpasta *Drivers*

algumas da bibliotecas replicadas, mas com uma implementação própria (*e.g., mem.c, stdlib.c e stdio.c*);

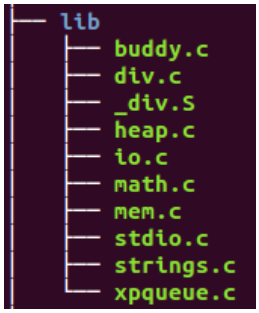


Figura 4.3: Árvore da Subpasta *Lib*

- **guests** – Nesta diretoria será incluído a imagem do *guest* a executar no mundo não-seguro. Encontra-se o código de inclusão do seu binário no binário do LTZVisor, assim como da configuração do posicionamento do próprio *guest* no espaço de memória que lhe é dedicado;

- **secure_guest** – O *guest* do mundo seguro é considerado parte do sistema (parte da TCB) em vez de considerado um simples *guest*. Esta consideração permite que o mesmo seja integrado no sistema LTZVisor da forma apresentada na Figura 4.4. Entre os ficheiros estão incluídos os ficheiros de *porting*, neste caso para a arquitetura ARMv7, compatível com as plataformas alvo contendo Zynq-7000 assim como da plataforma simulada pelo Fast Models, Versatile Express.

De referir ainda a localização do ficheiro Makefile no diretório "mãe". É através da modificação deste ficheiro que se determina as configurações bases da própria

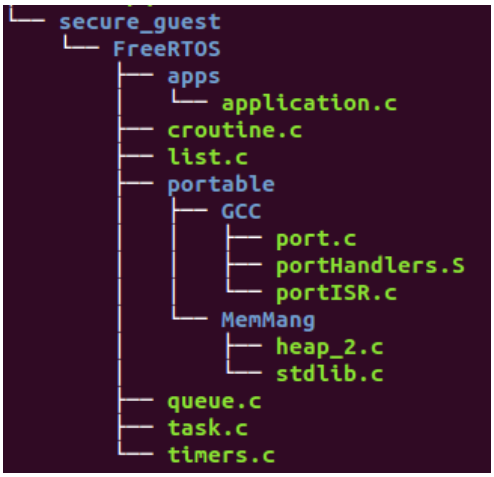


Figura 4.4: Árvore da subpasta *Secure_Guest*

framework incluindo a arquitetura e processador alvo, assim como futuras configurações implementadas como o suporte para uma configuração *multicore* ou até o suporte para comunicação. Na Listagem 4.3 verifica-se um exemplo simples de atribuição da plataforma Zynq no Makefile, e consequentemente mudança de *flags* de compilação.

Listagem 4.3: Seleção de Plataforma no Makefile

```
1 PLATFORM=zynq
2 ifeq ($(PLATFORM), zynq)
3         CC_FLAGS += -DCONFIG_ZYNQ=1
4         ASM_FLAGS += -DCONFIG_ZYNQ=1
5         ARM_A = Cortex-A9
6 endif
```

4.1.2 Fluxo de Execução

Nesta subsecção será retratada a inicialização do sistema LTZVisor, maioritariamente na implementação do suporte para a arquitetura do processador Cortex-A9 presente na plataforma de desenvolvimento Zedboard. O fluxo inicial é observado de forma superficial na Figura 4.5 e detalhado a seguir no decorrer desta secção.

Após os processos de verificação e cópia do binário do cartão de memória SD para a memória principal por parte do FSBL (*First Stage Boot Loader*), o mesmo irá realizar o *handoff* da execução do core 0 do processador para a entrada do LTZVisor. A entrada do sistema é definida pela *label* indicada no *linkerscript* através do comando *ENTRY(label)*, neste caso o *reset_handler*.

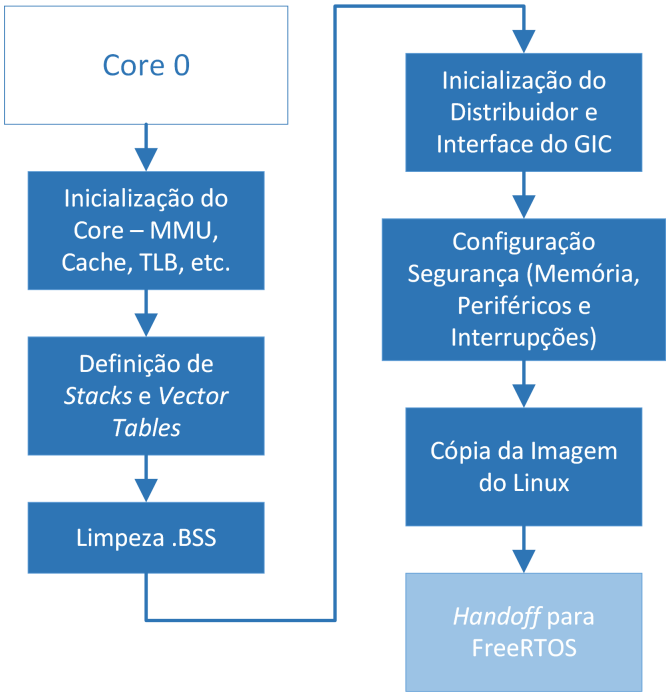


Figura 4.5: Fluxo da Inicialização do Sistema

A primeira parte do código a executar, presente na page 64, determina a validade do core para correr o código, *i.e.*, apenas o core 0 poderá executar aquele código, sendo que os restantes cores ficarão em *loop*. Esta verificação impede que, no caso de o código ser simulado na plataforma Fast Models, os cores secundários o executem, uma vez que todos eles partem do *reset_handler* ao mesmo tempo. No caso das plataformas físicas contendo o SoC Zynq-7000, os cores secundários esperam por instruções extras antes de "executarem" qualquer código, retratado em detalhe mais à frente na secção 4.2.

Listagem 4.4: Algoritmo de verificação de core

```
1 bl      get_cpu_id      @r0 = core_id
2 cmp     r0, #1          @if core_id == 1,
3 bleq    .               @loop
```

De seguida o LTZVisor trata da inicialização do SoC, nomeadamente desabilitando e limpando qualquer periférico capaz de causar problemas numa fase inicial, especificamente a MMU e as Caches. É realizado então a configuração das *stacks* de todos os modos de execução assim como das *vector table* do monitor e do *guest* do mundo seguro, através dos registos VBAR e MVBAR acedidos pela interface do coprocessador 15, observado na page 65. São inicializados diversos registos com o seu valor de *reset* (SCTRL, NSACR e ACTRL) sendo que no registo SCR

é desabilitada a opção de encaminhamento das FIQs para o modo monitor, uma vez que o sistema está em fase de inicialização e se encontra no mundo seguro.

Listagem 4.5: Instanciação das vector table no CP15

```
1 ldr     r0, =_secure_vector_table
2 mcr     p15, 0, r0, c12, c0, 0  @Write to VBAR register
3
4 ldr     r0, =_monitor_vector_table
5 mcr     p15, 0, r0, c12, c0, 1  @Write to MVBAR register
```

A seguir à inicialização da secção BSS (zona de memória onde se encontram as *stacks* e variáveis alocadas estaticamente) o LTZVisor procede à inicialização do sistema dual-OS. Nesta fase será realizada a configuração de segurança da plataforma, através da atribuição de certos periféricos, blocos de memória e interrupções aos respetivos *guests* de acordo com o nível de segurança que lhes é conferido.

Finalmente, o LTZVisor copia a imagem do *guest* do mundo normal para a respetiva zona de memória e conclui-se o processo de inicialização do sistema procedendo-se ao *handoff* da execução do processador para o *guest* do mundo seguro.

A execução normal do sistema foi já escrutinada na subsecção 3.3, sendo que a política de escalonamento do sistema de virtualização, *idle scheduling*, define o comportamento geral do sistema. Isto é, o *guest* do mundo normal é tratado como um processo de menor prioridade do *guest* do mundo seguro. Esta característica juntamente com o encaminhamento de interrupções FIQ do mundo normal para o monitor (e consequentemente mundo seguro) reforça o cumprimento das características de tempo-real do mundo seguro, demonstrando as únicas possibilidades de entrada e saída dos diferentes *guests*.

4.1.3 Para-TrustZone

Durante a fase de design da dissertação foi considerada a implementação de comunicação inter-partição utilizando como base a tecnologia *open-source* OpenAMP (Subsecção 2.4). Esta tecnologia, como referido anteriormente, está disponível para as plataformas contendo o SoC Zynq-7000, posteriormente modificada e utilizada no LTZVisor. Esta tecnologia surgiu como uma resposta às drivers de Linux denominadas de Remoteproc e RPMsg, drivers essas disponíveis a partir da versão 3.4. Este requisito não era compatível com a versão atual da *framework* LTZVisor, uma vez que a mesma apenas incluía a versão 3.3 do Linux. Tendo em conta este

impedimento, foi decidido realizar a atualização do *guest* do mundo normal para a versão 4.0 (uma versão mais madura e com resultados previamente obtidos para a versão do OpenAMP posteriormente modificada e utilizada neste projeto).

Como referido previamente na Subsecção 3.3, a versão em desenvolvimento do LTZVisor permite aos seus *guests* acesso aos dispositivos partilhados do sistema utilizando a técnica de para-TrustZone com uma política de acesso denominada *passthrough*. Este tipo de para-virtualização apenas acontece nos dispositivos que por serem considerados pela tecnologia TrustZone como demasiado influentes no sistema, não oferecem uma configuração de segurança modificável, não podendo por isso ser acedidos diretamente pelo ambiente de execução do mundo normal.

4.1.3.1 Linux 3.3

Na versão 3.3 do Linux apenas o Triple Timer Counter 0 (TTC0) entra no rol de dispositivos considerados essenciais para uma versão minimalista, mas funcional do SO Linux e também como contendo uma configuração, neste caso não modificável, de segurança. Uma vez que este dispositivo não era utilizado pelo mundo seguro, nomeadamente pelo *guest* seguro FreeRTOS, não surgiu qualquer problema na sua para-virtualização, através da técnica para-TrustZone. De referir que dispositivos como a porta-série por não serem considerados influentes no isolamento dos mundos virtuais não possuem configurações de segurança podendo ser acedidos por ambos os mundos, no entanto a interrupção associada à mesma, podendo pertencer apenas a um mundo virtual, foi atribuída ao ambiente de execução do mundo normal.

4.1.3.2 Linux 4.0

A versão 3.3 foi a primeira versão do Linux a ser portada para as plataformas com o SoC Zynq-7000, sendo que se verificaram bastantes melhorias no código *machine-dependent* do Linux desde essa primeira versão até à 4.0, versão atualizada no LTZVisor. Entre as modificações, contemplam-se as mudanças da utilização de diferentes dispositivos para o funcionamento normal do SO. Para que a nova versão executasse normalmente no sistema LTZVisor, os dispositivos essenciais para o funcionamento de uma versão minimalista do Linux tiveram de ser identificados e posteriormente para-virtualizados através da técnica para-TrustZone, ou simplesmente configurados como não-seguros para poderem ser acedidos diretamente pelo SO.

De entre as modificações no código *machine-dependent* denota-se a introdução do suporte para a utilização do modo FIQ dos processadores ARM. Apesar de o

suporte apenas requerer a inicialização do modo (*stack* e FIQ/IRQ bits), este teve de ser removido. A remoção deve-se à política de *context-switch* implementada no sistema (*lazy context-switch*), visto que esta não contempla a troca de contextos dos modos FIQ e IRQ entre os diferentes mundos. Esta remoção não causou impacto no sistema ou no próprio *guest*, uma vez que lhe era requerida única e exclusivamente a utilização do modo IRQ no serviço das interrupções.

Na versão 4.0, os seguintes dispositivos foram identificados como cruciais para o funcionamento correto de uma versão minimalista do Linux: i) o Global Timer da ARM, ii) as PLLs (Phase Locked Loop) da plataforma Zynq-7000 e ainda iii) o acesso ao registo de controlo de energia através do coprocessador 15. Nenhum dos dispositivos acima referidos influenciam o funcionamento do *guest* do mundo seguro sendo possível mais uma vez a aplicação da técnica de para-TrustZone.

A atualização do Linux envolveu a atualização do seu temporizador para a realização de *scheduling* do SO, passando a utilizar o Global Timer da ARM em vez do TTC0. Esta modificação foi benéfica para o sistema, uma vez que o Global Timer possui uma interface nos registos da SCU (Snoop Control Unit) capaz de modificar a sua configuração de segurança dinamicamente, podendo assim ser acedido diretamente pelo *guest* do mundo normal. Na Listagem 4.6 pode-se observar o excerto de código referente às modificações necessárias na inicialização do sistema para que o Global Timer pudesse ser configurado pelo mundo normal no core 0.

Listagem 4.6: Mudança de estado de segurança do Global Timer

```
1 //enables the CPUs to access SCU_NSACR
2 write32((void*)SCU_ACR, 0x3);
3 //enables core 0 to access GT in NS state
4 write32((void*)SCU_NSACR, 0x100);
```

O Linux 4.0, comparativamente à sua versão 3.3 utiliza as PLLs para obter e configurar o valor do rating do clock de alguns periféricos que utiliza, nomeadamente da UART e do temporizador Global Timer. Anteriormente este valor era obtido através do Device Tree Blob (DTB), algo considerado desacertado uma vez que estes podiam ser obtidos diretamente do hardware. Estes periféricos são considerados sempre seguros no SoC Zynq-7000, tendo de ser utilizada a técnica de acesso para-TrustZone. No entanto, dado os acessos às PLLs apenas se verificarem na inicialização do *guest*, o *overhead* na execução normal do Linux ou do próprio sistema global é nulo.

De forma idêntica aos acessos às PLLs do sistema, o registo de controlo de energia apenas é acedido na altura de *boot* do Linux, não causando *overhead* na execução normal do Linux. Este registo apenas é acessível em modo seguro e permite fazer uma gestão do gasto de energia relacionado com a utilização das Caches. Isto é, permite o controlo dinâmico do relógio da Cache. Uma vez que esta se encontra inativa na partição do mundo seguro, esta configuração não terá influência negativa no ambiente de execução seguro.

No geral, a atualização do *guest* do mundo normal verificou-se positiva, uma vez que a utilização da técnica para-TrustZone para a virtualização de certos dispositivos apenas é utilizada durante o *boot* do Linux. Além de se observar uma atualização geral das drivers disponíveis para o sistema operativo, esta versão garante o suporte para a implementação dos mecanismos comunicação inter-partição baseados nas drivers Remoteproc e RPMsg. Esta versão tem ainda a vantagem indireta de ser uma versão mais madura e com eliminação de possíveis bugs/erros, inclusivamente no código *machine-dependent* visto já não se tratar de uma primeira versão (versão alfa).

4.2 LTZVisor - Multicore

As seguintes secções definem as principais modificações ao sistema de virtualização LTZVisor para suportar uma configuração *multicore*. Primeiramente demonstrar-se-á o design escolhido para a arquitetura do sistema quando executando na configuração *multicore*. Por fim, serão explicados os detalhes de implementação da expansão das configurações da *framework*.

4.2.1 Design

A tecnologia TrustZone insere no SoC um isolamento dos diferentes recursos da plataforma, criando dois mundos virtuais. Os recursos são particionados ao nível da plataforma e não ao nível do *core*, ou seja, numa plataforma *multicore* existirão apenas dois mundos virtuais. Isto significa que a tecnologia TrustZone não tem influência no isolamento inter-*core*. Este apenas existirá no contexto da tecnologia TrustZone e apenas em arquiteturas dual-*core*, se a cada *core* for atribuído um ambiente de execução diferente, seguro ou não-seguro.

Devido a este particionamento, a virtualização implementada na *framework* LTZVisor, que impõe um isolamento de forma exclusiva através da atribuição de um ambiente de execução num mundo diferente a cada partição, apenas suportará

a inclusão de duas partições independentemente do número de *cores* disponíveis na plataforma.

Todavia a expansão da *framework* para uma configuração *multicore* significa um acréscimo ao nível de *cores* de processamento utilizados, o que no panorama global corresponderá a uma melhoria do desempenho da *framework*. A configuração *multicore*, devido à limitação supramencionada prevê a utilização do mesmo número de *guests* continuando com uma arquitetura dual-OS.

A plataforma de desenvolvimento utilizada contém apenas dois *cores* no seu SoC. Por essa razão foi escolhida uma arquitetura dual-OS dual-core, em que em cada *core* se encontra hospedado exclusivamente por um *guest*. Esta arquitetura é também comummente denominada de AMP supervisionada. A arquitetura escolhida implica também, que a cada *core* seja conferida uma configuração de segurança única correspondente ao *guest* que nele executa, *i.e.*, o *core* definido como seguro irá hospedar o *guest* do mundo seguro e vice-versa.

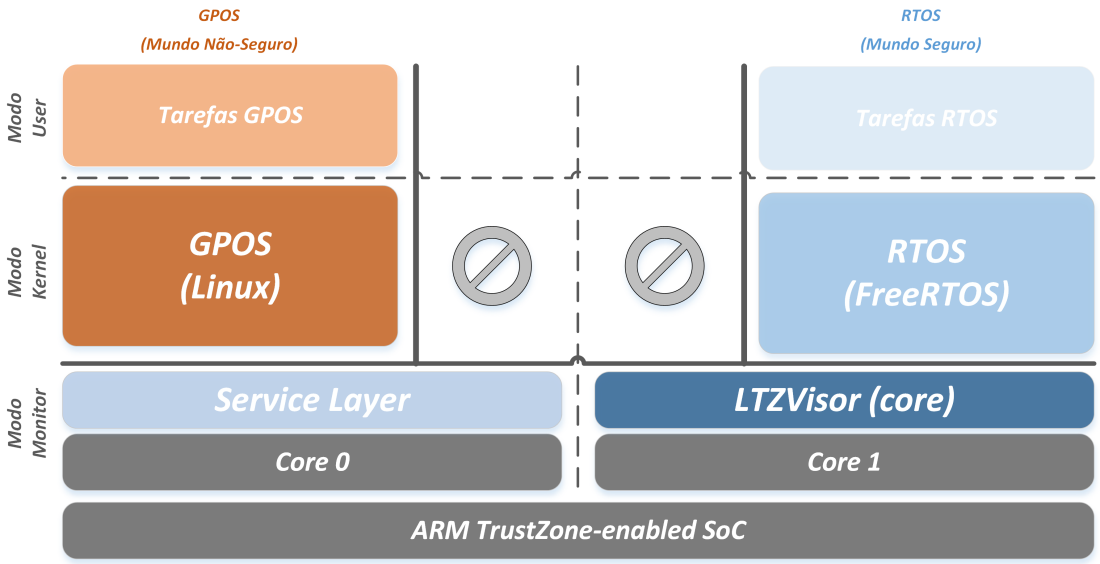


Figura 4.6: Arquitetura do LTZVisor em configuração AMP

A Figura 4.6 representa a arquitetura do LTZVisor em configuração AMP. O *guest* do mundo não-seguro encontra-se hospedado no *core* 0 e o *guest* do mundo seguro hospedado no *core* 1. Mantendo a coerência em relação à segurança conferida a cada *core*, a componente principal do monitor irá executar no *core* 1, determinado como *core* seguro. O *core* 0 irá conter uma pequena camada de *software* a executar no modo monitor, denominada de "service layer" (camada de serviço), responsável pela inicialização do *core* e pela para-virtualização dos dispositivos utilizados pelo *guest* desse *core* através da técnica para- TrustZone. Esta

camada será ainda responsável por dar ao *guest* não-seguro suporte à comunicação inter-*core*, através da virtualização das notificações, explicado na Secção 4.3.

A atribuição da segurança a cada *core* não foi aleatória. Em virtude da maior complexidade e quantidade de código presente no *guest* não-seguro (SO de propósito geral), a tarefa de o portar para executar num *core* diferente do predestinado (*core* 0), através da modificação do seu GIC e código *machine-depending*, seria demorada e complexa. Essa tarefa traduzir-se-ia num *time-to-market* mais lento, algo indesejável no mundo de sistemas embebidos. Por esta razão foi atribuído ao *core* 1 o estado seguro, em que o *port* do *guest* seguro se mostrou relativamente simples.

O *layout* de memória da configuração AMP do LTZVisor é o mesmo da configuração *single-core*, visto que a memória utilizada pelos *guests* é idêntica, excetuando a camada de serviço adicionada, inclusa na zona de memória do monitor.

4.2.2 Implementação

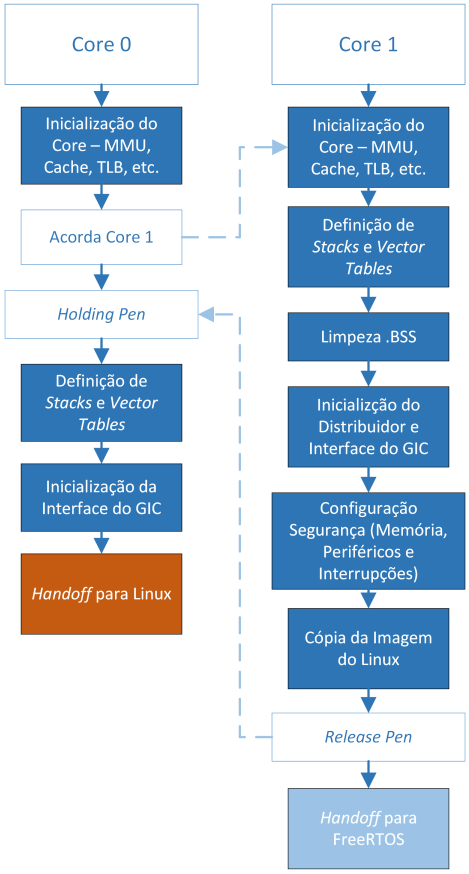


Figura 4.7: Fluxo da inicialização do sistema em configuração AMP

Nesta subsecção serão revelados alguns dos detalhes de implementação da configuração *multicore* assim como detalhes do fluxo de inicialização da plataforma. Esta implementação teve como objetivo manter a configuração *single-core* inalterada ao mínimo em termos de comportamento. Através da ativação da *flag* de compilação *"-DCONFIG_AMP"* no Makefile, a compilação do código do LTZVisor passa de uma configuração *single-core* para uma *multicore*.

Através da Figura 4.7 é possível observar o panorama geral do fluxo de inicialização da plataforma. Como constatado, o *core* 0 tem poucas responsabilidades na inicialização do sistema, estando maioritariamente em "*idle*" até que o *core* 1 conclua as suas tarefas.

4.2.2.1 Arranque do Núcleo Secundário

O *core* designado como *core* não-seguro, arranca inicialmente num estado seguro por forma a realizar as inicializações do próprio *core*. No entanto, este necessita de esperar pelas inicializações conferidas ao *core* seguro, uma vez que este possui as componentes principais do hypervisor para que seja mantida a coerência de segurança. Assim, a primeira tarefa do *core* primário é a de arrancar o *core* secundário, sendo necessária a realização das inicializações base que o permitam fazer de forma correta.

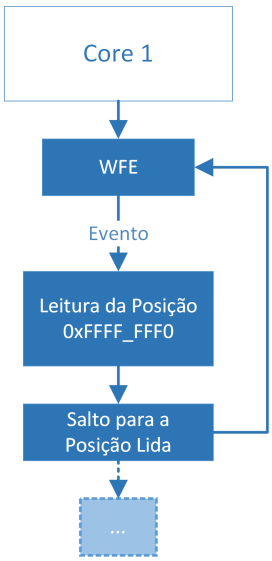


Figura 4.8: Fluxo do Arranque do Core 1

Os *cores* secundários, devido às características do SoC Zynq-7000 encontram-se num estado Wait For Event (WFE). Neste estado particular de inicialização, como demonstrado na Figura 4.8, o *core* secundário sempre que recebe um evento (através da instrução SEV (*Send EVent*)) irá ler o valor presente no endereço de

memória *0xFFFF_FFF0*. O valor por defeito indicado nessa posição de memória irá apontar para a instrução anterior WFE, evitando que o *core* secundário acorde desnecessariamente desencadeado por qualquer evento.

O arranque do *core* secundário é realizado através da escrita do endereço da primeira instrução do código destinado ao *core* 1 na posição *0xFFFF_FFF0*, seguido da instrução SEV. Através da Listagem 4.7, observa-se ainda a colocação de uma barreira (DSB – *Data Synchronization Barrier*) entre a escrita e o desencadeamento do evento, certificando-se que os valores lidos pelo código secundário estão devidamente atualizados, evitando o envio repetido de eventos.

Listagem 4.7: Acordar do core secundário

```
1 ldr     r0, =0xFFFFFFF0          @ Wake up Core 1 by writting
2 ldr     r1, =_reset_handler      @ the start address to
3 str     r1, [r0]                 @ the position 0xFFFFFFF0
4 dsb     @ Barriers so the other cores will see the changes
5 sev     @ Send the event to wake
```

4.2.2.2 Inicialização da Plataforma

O *core* primário, que inicia em modo seguro (modo monitor), encarrega-se de realizar as inicializações específicas ao *core*, similarmente às realizadas em configuração *single-core*, incluindo: i) a inicialização das stacks, ii) dos periféricos próprios (MMU, Caches e TLBs) e iii) colocar o valor de reset nos registos SCTLR, ACTLR, NSACR e SCR.

Este *core*, por uma decisão de design justificada anteriormente, não será responsável pela inicialização dos recursos da plataforma como o distribuidor do GIC nem pelo particionamento dos recursos globais, ficando essa tarefa a cabo do *core* secundário. No entanto, aparte das inicializações do *core* acima descritas, deverá realizar as configurações de segurança ao nível do *core*, como a configuração das interrupções utilizadas pelo GPOS e o particionamento de recursos específicos ao *core*. Posteriormente a estas configurações e inicializações, o *core* primário entra no mundo não-seguro, libertando o processador para o *guest* não-seguro, Linux. A partir deste momento, o *core* primário não voltará a entrar no mundo seguro, daí a designação do *core*, como não-seguro. O *core* apenas entra em modo monitor para a realização da para-virtualização para-TrustZone e para o envio de notificações na comunicação inter-partição.

Antes das inicializações supracitadas, o *core* primário, após arrancar o *core* 1, deverá ficar em estado WFE até que o *core* 1 finalize a inicialização global da plataforma. Este estado é caracterizado na subsecção 4.2.2.3.

O *core* secundário executará sensivelmente as mesmas instruções destinadas ao *core* 0 na configuração *single-core*. Isto é, será responsável pelas inicializações específicas do *core* assim como das globais à plataforma. Esta migração das funções principais do LTZVisor para o *core* 1, incluindo a cópia da imagem do *guest* do mundo normal, correspondem à coerência em atribuir o nível de segurança a esse mesmo *core*. As poucas alterações visíveis são a não configuração da segurança das interrupções (realizado pelo *core* onde executa o *guest* não-seguro) e a introdução de um mecanismo de sincronização para o recomeçar de tarefas do *core* primário revelado na subsecção 4.2.2.3.

Na configuração *multicore* AMP a troca entre mundos não é efetuada, uma vez que cada *core* é utilizado de forma mutualmente exclusiva pelos mundos virtuais disponíveis. Não obstante é necessário realizar a inicialização do contexto do mundo normal, visto que o *core* por ele habitado é inicializado no mundo seguro.

4.2.2.3 Sincronização – *Holding Pen*

Como referido previamente, o *core* primário necessita de esperar por certas configurações do *core* seguro, como a inicialização do distribuidor do GIC e até a cópia da imagem do *guest* não-seguro. Para que o *core* primário não execute inicializações antecipada e erroneamente será necessária a implementação de uma forma de sincronismo inter-*core*. O mecanismo de sincronização escolhido foi o de *holding pen*, conceptualmente idêntico ao mecanismo utilizado pelo SoC Zynq-7000 para evitar o arranque incorreto dos núcleos secundários.

O mecanismo *holding pen* implementado na configuração *multicore* do LTZVisor não é mais do que uma variável numa posição de memória específica inicializada pelo núcleo não-seguro com um determinado valor e reescrita mais tarde pelo núcleo seguro. Durante o desfasamento temporal entre as duas escritas, o *core* não-seguro, assim que chegar ao "ponto de sincronização", irá repetidamente testar a variável, até que a mesma obtenha o valor esperado. De forma a não sobrecarregar o barramento de memória com leituras dispensáveis, o processador entra no estado WFE após a primeira leitura indesejada, similarmente ao estado inicial do núcleo secundário, até que o núcleo seguro atualize a variável e desencadeie um evento (SEV).

A Listagem 4.8 representa as iterações do *core* 0 em "espera" até que receba o valor desejado na variável, neste caso o valor 0. Faz-se uso das características

condicionantes adicionadas às instruções específicas do ISA do ARMv7, sendo que neste caso (uso do sufixo "*-eq*") apenas executam se a *flag* de condição de igualdade (zero no resultado da comparação) estiver ativa.

Listagem 4.8: Mecanismo de sincronismo *Holding Pen*

```
1 1:
2 ldr       r1, =pholding_pen
3 add       r1, r1, r0, lsl #2
4 ldr       r1, [r1]
5 cmp       r1, #0
6 wfeeq
7 beq       1b
```

4.2.2.4 *Guest* do Mundo Seguro

O *porting* do *guest* do mundo seguro manifestou-se relativamente simples, apenas sendo necessário alterar a configuração das interrupções dos temporizadores para utilizarem o processador no qual o FreeRTOS executa. Demonstrado na Listagem 4.9 abaixo apresentada, esta modificação representa uma mudança em ambas as configurações do LTZVisor, uma vez que agora o *guest* é obrigado a obter a identificação do *core* no qual se encontra através de um registo presente no coprocessador 15.

Listagem 4.9: Exemplo de utilização de identificação do núcleo

```
1 interrupt_target_set(TTC1_TTCx_2_INTERRUPT,get_cpu_id(),1);
```

```
1 .global get_cpu_id
2 .func get_cpu_id
3 get_cpu_id:
4 mrc       p15, 0, r0, c0, c0, 5
5 and       r0, r0, #0x03
6 bx        lr
7 .endfunc
```

4.3 LTZVisor - Comunicação Inter-Partição

A decisão de implementação de mecanismos de comunicação na *framework* do LTZVisor deveu-se às características referidas na Subsecção 3.3.2.3, sendo utilizadas as implementações existentes no Linux e *open-source* do OpenAMP

como base. Foram utilizadas as versões das drivers Remoteproc, RPMsg e VirtIO presentes na versão 4.0 do Linux e utilizou-se a biblioteca OpenAMP disponibilizada pela ferramenta de desenvolvimento SDK da Xilinx (2015.4.1). Esta última identifica-se como uma versão da implementação *open-source* das especificações OpenAMP (versão 04.15) portada pela própria Xilinx para as plataformas contendo o SoC Zynq-7000.

No entanto, as ferramentas utilizadas providenciam nativamente mais do que comunicação (interação inter-partição) e encontram-se desprovidas de mecanismos de inserção em ambientes virtualizados. Nas seguintes subsecções serão analisadas e justificadas algumas das modificações realizadas tanto às implementações nos próprios *guests* como na *framework* em si para suportar um mecanismo de comunicação inter-partição baseado nas tecnologias presentes no OpenAMP.

4.3.1 Design

Existem alguns requisitos e premissas que o mecanismo de comunicação deve respeitar, de modo a poder ser inserido na *framework* LTZVisor:

- **Isolamento de Memória** – Para que o GPOS possa utilizar a memória partilhada, a mesma não pode ser configurada pelo TZASC como seguro. No entanto, devem ser garantidas algumas premissas para que a memória não seja suscetível a ser atacada. O *guest* seguro deve ter acesso total e indiscriminado a essa memória. Por outro lado, o *guest* do mundo não-seguro apenas deve poder aceder à memória em modo privilegiado e preferencialmente de forma exclusiva pela driver VirtIO/RPMsg. Estas últimas são conseguidas através da driver *cma (contiguous memory allocator)* pertencente ao Linux, que permite alocar grandes zonas de memória para um só dispositivo, no caso, o *rpmsg_device*, dono do *VirtIO_device*;

- **Características de Tempo Real** – As características de tempo-real do RTOS devem ser respeitadas e mantidas. Esta premissa apenas é garantida se o canal de eventos for controlado pelo hypervisor. Deste modo, um possível ataque do GPOS através do uso inadequado das interrupções é devidamente protegido pelo hypervisor através de um controlo das próprias interrupções. Do lado do RTOS, as tarefas de comunicação devem, quando possível, ter uma prioridade inferior em relação às tarefas de tempo-real;

- **Falha de Memória** – Utilizando uma alocação estática da memória partilhada, na altura da configuração do sistema, permite que a memória esteja

sempre presente e devidamente alocada por parte do RTOS, não existindo possíveis falhas de alocação de memória se a mesma fosse realizada de forma dinâmica.

- **Bloqueio Indeterminado** – De modo a garantir o cumprimento das deadlines de tempo-real do RTOS a comunicação não deve apresentar bloqueios intermináveis ou indevidos. O mecanismo de comunicação, por se tratar de uma comunicação assíncrona, tem como único ponto de bloqueio o *fetch* de buffers por parte do VirtIO. Não existindo buffers disponíveis na memória partilhada o programa bloquearia por um tempo fixo. Para que este seja inexistente, a função deve retornar um erro específico. A próxima sequência de ações é *aplication-defined*, usualmente definida pela espera associada à notificação de disponibilização de buffer (uma das notificações inter-partição disponíveis na comunicação);

Para além das premissas e dos requisitos impostos ao mecanismo de comunicação serão necessárias realizar algumas alterações à especificação OpenAMP de modo a que a mesma possa ser suportada pelo LTZVisor e respeitar o isolamento temporal e espacial imposto pelo mecanismo de virtualização.

Primeiramente deve ser retirado o componente Remoteproc do sistema, uma vez que o mesmo pela sua natureza quebra o isolamento temporal e espacial assim como permissões dos próprios *guests*. Não obstante, esta remoção não representa uma perda de funcionalidades pois estas estão já integradas no monitor do LTZVisor, nomeadamente a gestão de ciclo de execução dos *guests* do sistema. Obviamente as especificações da plataforma associadas ao Remoteproc e as inicializações dos restantes componentes não deixam de ser utilizadas, no entanto a sua utilização será migrada para o RPMsg.

Em segundo lugar, a utilização das interrupções deverá ser gerida pelo hypervisor. Apesar de ser uma fonte de *overhead*, esta gestão representa um *trade-off* entre performance e segurança. Esta modificação permite, como referido anteriormente, a proteção de ambos os *guests* através do controlo de acessos às interrupções quando estas forem realizadas de forma comprometedora. Permite ainda que as interrupções possam ser utilizadas de forma idêntica independentemente da configuração atual da *framework*, isto é, permite a utilização do mesmo canal de eventos (interrupções inter-*core*) mesmo estando em configuração single-core.

Finalmente, serão realizadas alterações nas implementações dos drivers/OpenAMP para que a comunicação siga o design apresentado na Figura 4.9. A alocação estática da *resource_table* incluída em cada *guest* em vez de apenas no *slave*

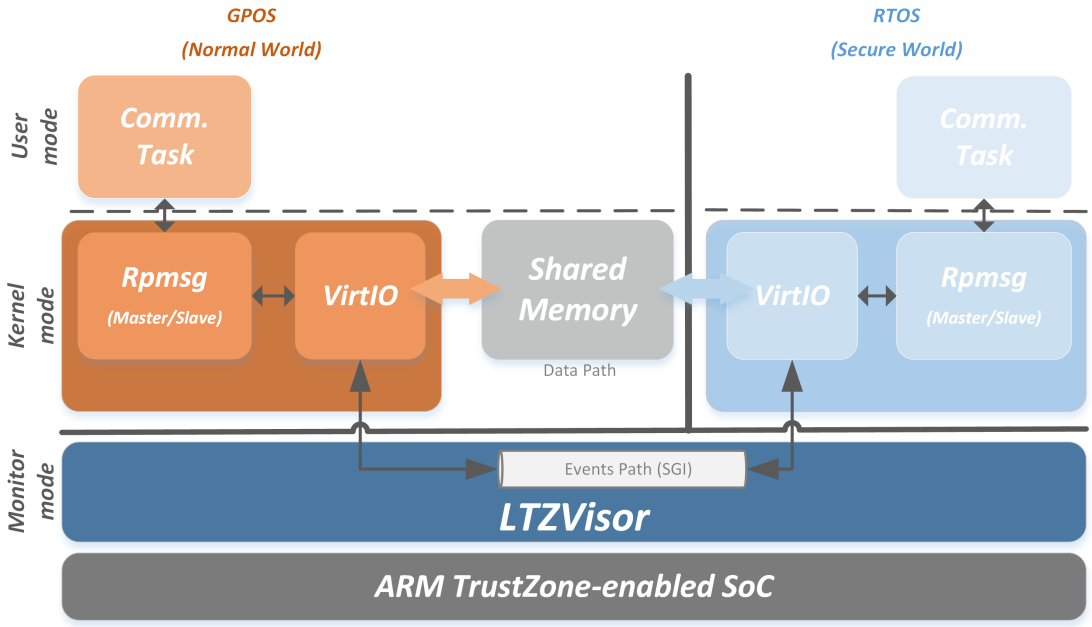


Figura 4.9: Arquitetura da comunicação inter-partição

da comunicação e criação de uma API de acesso à gestão de interrupções do monitor, assim como as implementações não presentes das configurações *master/slave* da comunicação.

De referir que devido à virtualização do canal de eventos, não haverá diferença entre a arquitetura do sistema com suporte para comunicação independentemente da configuração do LTZVisor. Na Figura 4.9 as partições representadas podem estar em *cores* diferentes ou no mesmo *core*, não se revelando diferenças no mecanismo de comunicação, uma vez que a camada de serviço pertencente ao monitor permite a virtualização do canal de eventos no *core* não-seguro.

4.3.2 LTZVisor

A cada canal estão associados dois buffers unidirecionais, isto é, dois *vrings*. Cada um desses canais será utilizado unidireccionalmente por um *guest*, no entanto apenas um deles será responsável pela inicialização dos buffers (*fetch* de buffer à memória principal e colocação no *array* "descritor" e "disponível"). A cada *vring* estão associadas duas interrupções, quatro por canal:

- **Vring 0, IPI 1** – Interrupção do *master* do canal para avisar o *slave* de que enviou uma mensagem/buffer;

- **Vring 0, IPI 2** – Interrupção que avisa o *slave* de que o *master* recebeu e leu o buffer e que o mesmo se encontra agora disponível no *array* de "disponíveis";
- **Vring 1, IPI 3** – Interrupção que avisa o *master* de receção de mensagem (disponível no *array* de "usados");
- **Vring 1, IPI 4** – Aviso direcionado ao *master* de disponibilização de buffer após leitura do mesmo;

Como supramencionado, as interrupções da comunicação inter-partição devem ser virtualizadas. Esta virtualização ocorre por vias de pedidos de interrupções que serão posteriormente despoletadas a seu devido tempo. O pedido deverá ser feito através da instrução "*smc*" com o id 0 e argumento (*arg0*) 34 como observado Listagem 4.10 onde está representado a macro para a solicitação de interrupções por parte do RTOS.

Listagem 4.10: Requisito de IPI ao monitor - mundo seguro

```
1 #define REQUEST_IPI(ipinr,guest_id) __asm__(\
2 "ldr     r0, =-34          \n\t"\
3 "mov     r1, %0            \n\t"\
4 "mov     r2, %1            \n\t"\
5 "smc     #0        \n\t"\
6 ::"r"((ipinr)),"r"((guest_id)))
```

O desencadeamento das interrupções será realizado de forma diferenciada de acordo com a configuração atual do LTZVisor. Numa configuração single-core o desencadeamento imediato de uma SGI, resultaria numa interrupção ao próprio *guest*. Por esta razão, o pedido deve ser armazenado pelo hypervisor e a interrupção deve ser posteriormente desencadeada, na altura de execução do devido *guest*. O momento apropriado foi definido como a altura de *context-switch* entre mundos. Deste modo a interrupção será desencadeada de forma automática, não sendo necessário qualquer tipo de mecanismo para que o hypervisor desencadeie a interrupção no *timing* certo. Este *timing* permite que a mesma seja servida o quanto antes, assim que esta seja a de maior prioridade. Na Listagem 4.11 podemos ver a macro a executar na troca de mundos que irá chamar a função de desencadeamento de uma interrupção.

Listagem 4.11: Envio de IPI em altura de *context-switch*

```
1 .macro send_ipi_to_ns
2 push    {r0-r12, lr}
3 bl      send_monitor_ipi_to_ns
4 pop     {r0-r12, lr}
5 .endm
```

O mecanismo de armazenamento de pedidos de interrupção escolhido foi o buffer circular. Este mecanismo tem uma política de *first in, first out*, isto é, permite que as interrupções sejam despoletadas pela ordem correta (ordem de chegada). Cada bloco de controlo de cada *guest* (VMCB) irá conter o seu próprio buffer, sendo que os pedidos são realizados pelo *guest* oposto como observado na Figura 4.10.

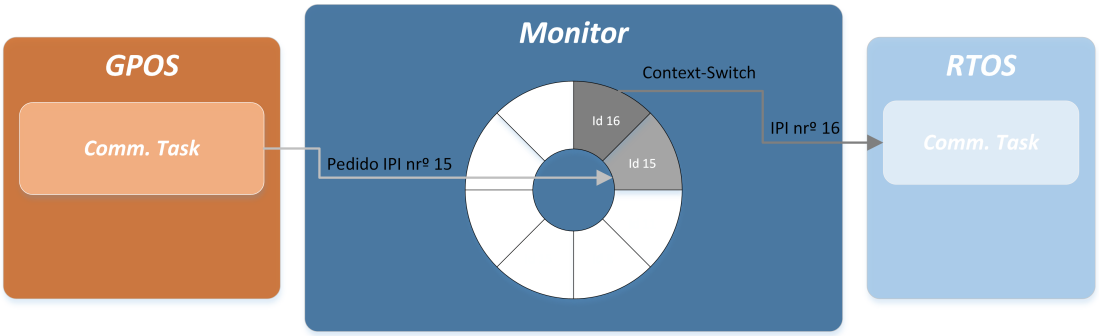


Figura 4.10: Pedido e Desencadeamento de IPI através do Monitor

Ainda nesta configuração devido à natureza das interrupções, será utilizada uma política de apenas uma interrupção numerada por buffer, isto é, cada interrupção poderá aparecer apenas uma vez no buffer. Esta política é possível e necessária pois uma única notificação faz com que o *guest* leia todas as mensagens disponíveis, e, enviando uma interrupção por mensagem em alturas de *context-switch* diferentes resultaria em leituras vazias. Isto otimiza a utilização de interrupções e evita o uso inapropriado das mesmas, mesmo em cenários de mal funcionamento do GPOS.

Na Listagem 4.12 está apresentada a estrutura de controlo de armazenamento e desencadeamento de interrupções, parte da VMCB de cada *guest*. A variável de 16 bits permite informar através da leitura de cada bit se alguma das 16 interrupções disponíveis no processador ARM se encontra no buffer. As duas primeiras variáveis definem o uso da interrupção que necessita de saber o número do *core* e o estado de segurança do mesmo para ser corretamente desencadeada.

Listagem 4.12: Estrutura de gestão de IPI presente nas VMCB

```
1 struct ipi_guest_state
```

```
2 {
3 uint32_t state; //TrustZone Security
4 uint32_t cpu_id;        //Cpu ID
5 uint32_t idx;           //Id of next element to be consumed
6 uint32_t tail;  //Id of the last inserted element
7 uint32_t ipinr[SIZE];   //circular buffer of SIZE 16
8 uint16_t on_buffer;     //if ipinr is already on buffer
9 };
```

Na configuração multicore a interrupção não necessita de ser armazenada visto que os *guests* encontram-se sempre ativos e disponíveis para receber as notificações. No entanto são utilizadas as informações supracitadas na estrutura do armazenamento de IPIs presentes na VMCB, como o estado e número do *core* do *guest* alvo, uma vez que estas são inicializadas pelo monitor em *boot-time*.

Adicionalmente foi inserido um mecanismo extra de segurança na estrutura de gestão de interrupções presentes na VMCB: ativação e desativação de interrupções específicas na sua VMCB. Isto permite que o *guest* não seja incomodado com avisos de disponibilidade de buffers ou outra notificação até que o requisite novamente através da ativação dessa interrupção no buffer circular. Esta funcionalidade apenas está disponível ao RTOS uma vez que este tipo de sensibilidade temporal é característica dele. A implementação passou pela adição de uma variável de 16 bits (cada bit corresponde a uma interrupção numerada) à estrutura de controlo de interrupções do VMCB do *guest* do mundo seguro, assim como uma API acedida através da instrução "*smc*".

De referir ainda que em ambas as configurações da comunicação, cada uma das interrupções utilizadas foi configurada com a segurança respetiva da partição que as recebe. Isto é, as interrupções que pretendem notificar o mundo seguro devem ser configuradas como seguras: Esta é uma razão adicional para a necessidade de virtualização destas interrupções, uma vez que o mundo não-seguro não dispõe de permissões para desencadear uma interrupção considerada segura.

4.3.3 Implementação

Nesta subsecção serão discutidos os detalhes de implementação no sistema global, nomeadamente o *layout* de memória e a remoção do componente Remoteproc. Posteriormente a esta secção serão descritas as implementações próprias a cada *guest*.

4.3.3.1 Memória

Na Figura 4.11 está representado o *layout* da memória do LTZVisor com suporte para comunicação incluído na plataforma Zedboard. A zona de memória partilhada encontrava-se livre, tendo sido previamente reservada para implementações futuras. Este espaço, de momento, encontra-se sobredimensionado com o propósito de servir uma futura expansão da comunicação.

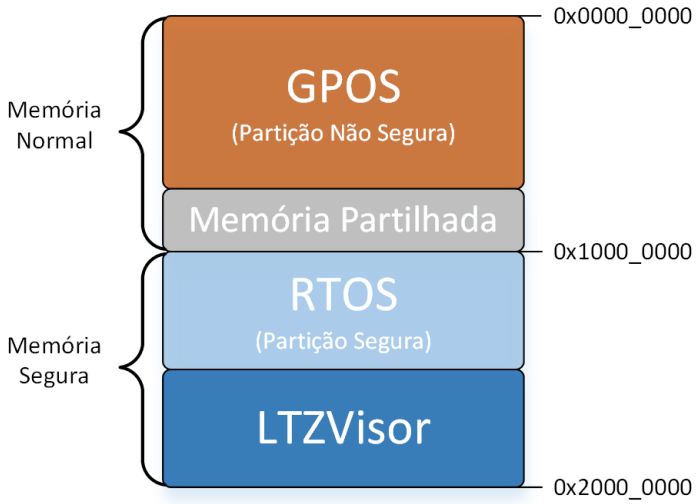


Figura 4.11: *Layout* da memória do LTZVisor com Comunicação

A memória partilhada é constituída pelas estruturas de controlo da comunicação, denominadas de *vrings* e pela zona de buffers a serem utilizados para a partilha de mensagens através da sua adição aos *vrings*. Os buffers podem pertencer a zonas de memória não contígua e dinâmica, e podem ser dinamicamente inseridos em qualquer um dos vrings (apenas movidos pelo *master* da comunicação), todavia, nesta implementação, os buffers encontram-se em zonas de memória contígua e estática, e são também estaticamente atribuídos a um *vring*.

Os parâmetros de configuração da memória partilhada seguem as linhas das implementações atuais do OpenAMP e das drivers existentes no Linux. Cada um dos *vrings* foi configurado para suportar 256 buffers. Este valor deve ser sempre uma potência de 2, visto tratar-se de estruturas formadas por arrays circulares, não sendo, no entanto, obrigatório que o *vring* contenha os 256 buffers, podendo inclusive conter apenas 1. O tamanho de cada buffer foi definido com o valor máximo de 512 bytes, sendo o valor real de dados enviados de 496 bytes devido à inserção do *header* em cada mensagem.

Com os parâmetros definidos podemos obter o valor de *overhead* de memória provocado exclusivamente pela inserção de memória partilhada na *framework*. A zona de memória destinada à alocação de buffers irá ter a dimensão de 262kb

(0x40000). Este valor é obtido através da multiplicação do número de buffers pelo número de vrings (512), tendo cada buffer o tamanho de 512 bytes, o valor total será de 512*512. As estruturas de controlo para o suporte de 256 buffers ocupam cerca de 5kb cada.

Estes valores podem parecer demasiado altos para o tipo de comunicação pretendida, no entanto, os mesmos foram escolhidos para que pudessem suportar qualquer tipo de comunicação futura, uma vez que é mais fácil o redimensionamento da memória partilhada para valores mais baixos do que para valores mais altos, não sendo necessário realocar as memórias atribuídas aos diferentes *guests*. Esta escolha justifica-se também para que a caracterização da comunicação, detalhada na Secção 5.2, pudesse ser mais completa.

4.3.3.2 Remoção Remoteproc

Como referido anteriormente foram realizadas alterações estruturais aos componentes da especificação OpenAMP, nomeadamente a remoção do Remoteproc e migração de algumas das suas funcionalidades de forma estática para o RPMsg. De forma a serem mantidas intactas as drivers originais, este novo componente foi designado de RPMsgSuper (RPMsg Supervisionado) numa alusão à arquitetura virtualizada no qual se insere. Para além da remoção da descarga do ficheiro *.elf* para o *core* secundário e da gestão de execução desse mesmo *core*, foram realizadas outras alterações a essa mesma componente integrada no RPMsg.

A obtenção de informações específicas da configuração do *VirtIO_device* e do *rpmsg_device*, anteriormente obtida da *resource_table* do *firmware* a descarregar no *core* secundário, foi modificada para ser uma estrutura estática em cada *guest*. Esta estrutura não é modificada de canal para canal, uma vez que a configuração dos dispositivos virtuais de comunicação é sempre igual, excetuando claro a sua posição na memória partilhada. Este posicionamento deverá ser efetuado através da obtenção dos valores corretos da DTB (*Device Tree Blob*) e de uma estrutura estática específica, no GPOS e RTOS respetivamente. De salientar que estes valores devem ser coerentes nas suas versões dos diferentes *guests*.

Finalmente foi necessária a modificação da conexão inicial entre os participantes da comunicação. O *handshake* numa arquitetura não supervisionada era sincronizado pelo descarregamento e ativação do núcleo secundário, no entanto este sincronismo desapareceu devido à remoção destas funcionalidades. Na Figura 4.12 está representada a reformulação do *handshake* de forma a respeitar o design da comunicação na arquitetura virtualizada. A interrupção inicial do *handshake* será

reproduzida indefinidamente (com intervalos de tempo definidos) até que seja recebida a confirmação do lado do *guest slave* com a informação NS (*Name Service*). O *guest slave* deverá esperar pela interrupção inicial para enviar o anúncio NS.

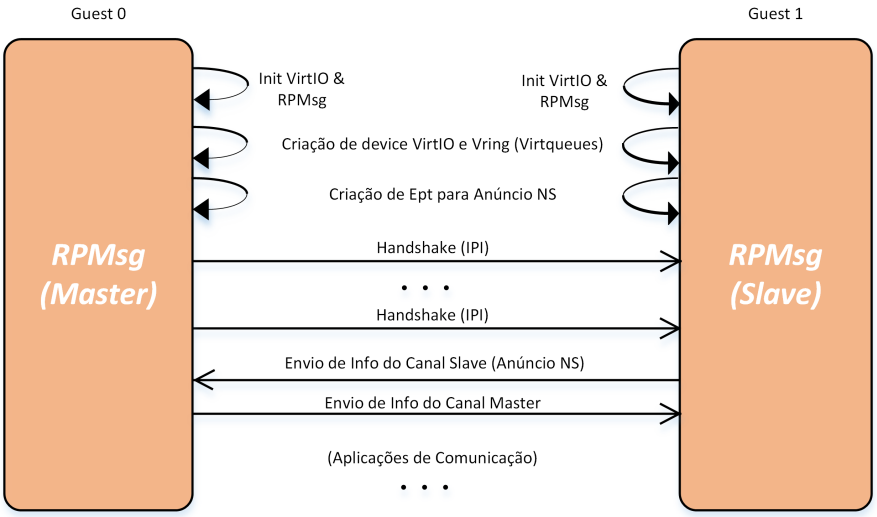


Figura 4.12: Reformulação do Handshake

Na Figura 4.12 é possível observar ainda que em resposta à chegada de uma mensagem NS é enviada o NS respetivo do *guest master*. Esta alteração permite a inclusão de vários canais num só *guest*, sendo que os mesmo são caracterizados pelo nome e endereço. Anteriormente o envio de um simples "Ack" era suficiente pois o *handshake* dependia do sincronismo fornecido pelo Remoteproc. Esta modificação está relacionada com a escalabilidade do sistema assim como com a inclusão do modo *slave* do componenete RPMsg no Linux, sendo considerada uma alteração necessária e coerente com o protocolo de comunicação.

Na arquitetura inicial não supervisionada a ação que despoletaria o *load* do *firmware* e inicialização dos dispositivos era realizada pelo utilizador. Por uma decisão de *design*, a inicialização dos dispositivos virtuais deverá ser automática e começar em *boot-time*, podendo assim o *handshake* ser realizado de seguida. A inicialização automática originou uma necessidade de reestruturação de algumas partes da camada de *software* dos *guests*, maioritariamente no GPOS. Esta reestruturação será analisada pormenorizadamente nas subsecções abaixo, pois cada sistema operativo terá as suas próprias propriedades.

4.3.4 RTOS

As alterações realizadas à implementação da especificação OpenAMP proveniente do XSDK 2015.4.1 foram menos profundas comparativamente às alterações

realizadas ao *guest* do mundo não-seguro.

A *stack* de *software* da comunicação foi incluída no próprio sistema LTZVisor, uma vez que fará parte da TCB e poderá ser utilizada posteriormente por qualquer sistema operativo *guest* do mundo seguro, desde que respeite a classe SO (baremetal ou RTOS). Os elementos respeitantes à classe de SO e ao próprio SO encontram-se localizadas na diretoria do próprio *guest*. As alterações, para além da remoção faseada do Remoteproc, foram realizadas em 4 partes distintas da camada de *software* da biblioteca de comunicação:

- **Notificações por eventos** – As notificações, como referido anteriormente, foram substituídas por pedidos de interrupção em vez do uso direto das mesmas. Os pedidos de interrupção assim como os pedidos de ativação/desativação de notificação são realizadas através da instrução "*smc*";
- **NS *callback*** – de modo a permitir uma inclusão de vários canais num só *guest*, o *callback* associado à chegada de um *Name Service Announcement* irá verificar o nome do canal recebido com o canal associado a esse *callback* de forma a verificar a sua compatibilidade. Posteriormente será enviado o NS do próprio canal para que o canal seja verificado e finalizado em ambos os *guests*;
- ***Resource table*** – visto tratar-se de uma implementação com resultados obtidos na sua versão *slave*, esta incluía já a estrutura *resource_table*. A alteração passou pela remoção do analisador de tabelas associado ao Remoteproc *master* e pela inclusão da estrutura estaticamente;
- **Modo *master* no VirtIO** – apesar de já fornecer a configuração do modo *master* da comunicação, a implementação da especificação do OpenAMP utilizado carecia de pequenas correções devido à falta de testes pelo grupo que desenvolveu a solução. De referir que estas correções foram aplicadas numa versão mais recente desta mesma implementação.

4.3.5 GPOS

As alterações realizadas às drivers do Linux que protagonizam a comunicação especificada pelo OpenAMP foram relativamente mais profundas, quer pela complexidade inerente à classe do SO em causa, quer pela inexistência de suporte à configuração *slave* da comunicação.

4.3.5.1 Interrupções Inter-Core

Como seria de esperar a utilização de SGIs (*Software Generated Interrupts*) no modo single-core do Linux encontram-se inacessíveis. Esta particularidade deve-se à possibilidade de um uso inadequado das mesmas para envio de sinais entre as drivers do Linux (intra-*core*) e ainda devido à utilização destas interrupções em modo SMP para comunicação entre o *scheduler* do *core* principal e o *scheduler* dos *cores* secundários.

Apesar do mecanismo de comunicação implementado não prever o uso direto das interrupções inter-*core*, mas antes a realização de pedidos, o acesso à API das mesmas é imprescindível para a invocação das rotinas de serviço e para providenciar uma implementação genérica da driver RPMsgSuper.

O código responsável pela incorporação da API de gestão de interrupções inter-*core* encontra-se no diretório *machine-dependent* do Linux e é compilado apenas se a *flag* CONFIG_SMP, definida em tempo de compilação, estiver ativa. Este código foi reproduzido num ficheiro apenas compilável dependendo do estado da *flag* CONFIG_RPMSG_SUPER (inversa à CONFIG_SMP e se escolhida pelo utilizador).

No excerto de código abaixo pode-se ver o pedido de interrupção inter-partição ao monitor através da instrução "*smc*". Esta função substituiu a própria invocação na API das interrupções inter-*core* supracitadas.

Listagem 4.13: Requisito de IPI ao monitor - mundo não-seguro

```
1 ENTRY(secure_ipi_request)
2 mov     r2, r1    /*guest id*/
3 mov     r1, r0    /*ipi nr*/
4 ldr     r0, =-33  /* smc call id - request ipi */
5 smc     #0
6 bx      lr
7 ENDPROC(secure_ipi_request)
```

4.3.5.2 RPMsgSuper

Como referido no inicio da secção, uma das partes fulcrais da adaptação do mecanismo de comunicação à arquitetura virtualizada foi a remoção parcial do componente Remoteproc. Esta passou maioritariamente pela remoção da gestão de ciclo de execução dos *cores* secundários e pela remoção dos elementos responsáveis pelo *load* do *firmware* nesses mesmos núcleos. Esta remoção suprimiu a

necessidade de criar um componente novo ou até mesmo um com uma configuração *master/slave*.

As funcionalidades restantes do Remoteproc foram migradas para o componente RPMsgSuper. Separados em dois elementos, estes têm propósitos diferentes:

- **Arquitetura** – Este elemento pertence exclusivamente à parte *machine-dependent* do componente RPMsgSuper. Isto é, irá conter as especificidades da plataforma como a configuração das interrupções e será também responsável pela obtenção dos parâmetros da comunicação variáveis de acordo com a arquitetura do processador. Esta aquisição é realizada através da leitura da DTB que deverá ter o aspeto descrito pelo excerto de código abaixo apresentado. Entre os parâmetros destacam-se a zona de memória partilhada, o número das interrupções a utilizar e o modo da comunicação definido pela variável *role*;

Listagem 4.14: Exemplo de configuração do RPMsgSuper na DTS

```
1         rpmsgsuper0: rpmsgsuper@0 {
2         compatible = "xlnx,zynq_rpmsgsuper"; //driver of this
              virtual dev
3         reg = < 0x08000000 0x000C0000 >;        //shared mem
4         chnnl_name = "rpmsg-openamp-demo-channel";
5         role = <0>;                        //role 0 = slave
6         guest_id = <0>;          //rsvd for future use
7         vring0 = <15>;
8         vring1 = <14>;
9         int_buf = <13>;
10        rcv_buf = <12>;
11        };
```

- ***VirtIO Device*** – A inicialização do dispositivo virtual VirtIO manteve-se inalterada pois os seus parâmetros e âmbito de funcionalidade mantiveram-se inalterados. A única alteração digna de destaque foi a substituição do *callback* de notificação de mensagem do modo *slave*, que teve de ser alterado para que o buffer a inspecionar na procura preliminar de mensagens fosse o correto. Como referido anteriormente também o método de armazenamento de parâmetros do dispositivo VirtIO foi alterado, para uma estrutura estática (*resource_table*).

O componente RPMsgSuper engloba ainda o antigo RPMsg, adaptado a ambos os modos configuráveis da comunicação e à arquitetura virtualizada. Esta

adaptação foi relativamente simples, havendo, no entanto, a necessidade de acrescentar o nome do canal nos processos de reconhecimento e ativação do canal. A criação do dispositivo *rpmsg_channel* teve de ser adiada no modo *slave* sendo apenas invocada no *callback* do NS, para que a driver associada ao canal pudesse ser sondada como se tratasse do *callback* de criação do canal.

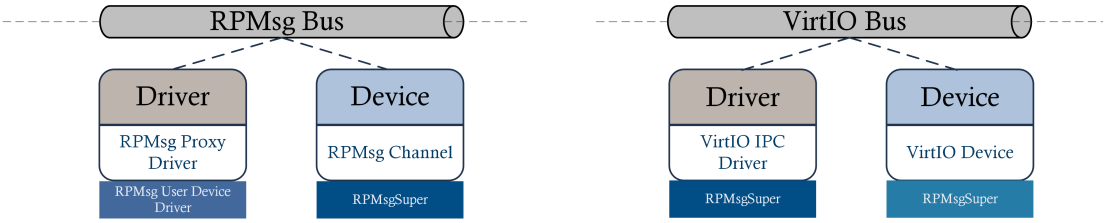


Figura 4.13: Overview da arquitetura da comunicação no Linux

Resumindo, o componente RPMsgSuper é responsável pela criação da maior parte do mecanismo da comunicação. Na Figura 4.13 é possível verificar o impacto no sistema global. O RPMsgSuper cria o *bus* para a criação de múltiplos canais e das respetivas drivers responsáveis pela interface com as aplicações utilizadores desse mesmo canal. Este componente também cria e gere os principais elementos (driver e dispositivo) inseridos no *bus* do VirtIO associados a cada um dos canais instanciados.

4.3.5.3 VirtIO – Slave

O VirtIO, como abstração de transporte direcionada para a para-virtualização de dispositivos em hypervisores baseados em Linux possui uma API para o modo *slave*/hypervisor dessa comunicação. Esta implementação da para-driver é denominada de vhost, sendo que a para-driver complementária tem o nome da tecnologia VirtIO.

Apesar de existir uma implementação do modo *slave* da abstração do transporte da comunicação, o mesmo não foi utilizado. As razões que levaram a essa decisão prenderam-se com a dificuldade de integração deste conjunto de APIs nos módulos existentes do RPMsgSuper, o que levaria à reformulação e até mesmo à recriação de um novo módulo RPMsgSuper compatível com as mesmas.

O vhost, comparativamente à versão utilizada do virtio (*slave*) utilizado no RTOS, oferece um conjunto de funcionalidades extras, como o uso do *scatter gather* (forma de utilizar um conjunto de memória não contigua como se tratasse de memória contigua). Estas funcionalidades não são utilizadas neste tipo de dispositivo VirtIO (*rpmsg_device*), visto que o mesmo utiliza o método "uma mensagem – um evento" traduzindo-se na inserção de um buffer no VirtIO de cada vez.

Concluindo, a utilização do vhost traduzir-se-ia num *trade-off* entre um *time-to-market* excessivo por um conjunto de funcionalidades que não seriam utilizadas. Foi, portanto, adicionado o suporte do modo *slave* ao componente VirtIO.

Conforme analisado na Subsecção 3.3.2.2 a única diferença ao nível da camada de transporte entre o modo *slave* e o modo *master* da comunicação centra-se na utilização dos arrays dos *vrings*. Isto é, o *master* tem permissão de escrita no array principal "descritor" e no array de buffers "disponíveis", contrariamente o *slave* apenas escreve no array de "usados".

O suporte para utilização do VirtIO em modo *slave* passou pela adição das funções *"virtqueue_get_available_buf"* e *"virtqueue_add_consumed_buf"*. Estes são responsáveis respetivamente pelo *fetch* de buffers ao *array* "disponível" e pela inserção de buffers no *array* de "usados". Este suporte apenas ficou completo com a atualização da estrutura interna de controlo de posição nos *arrays* com o suporte para o *array* de "usados".

4.3.6 Escalabilidade

Dutante a fase de conceção, design e implementação do mecanismo comunicação foi sempre tido em conta a possibilidade de escalabilidade dos mesmos. Nesta subsecção serão analisadas algumas das decisões feitas assim como da possível expansão destes mesmos mecanismos.

A cada canal de comunicação estão associadas 4 interrupções, sendo que cada *guest* apenas pode receber 2, as restantes serão as de envio. Na arquitetura atual dual-OS, existindo apenas dois *guests*, estes podem ser unidos por um único canal, principalmente devido à possibilidade de comunicações diferenciadas pelo endereço do *endpoint* respetivo dentro do mesmo canal (através do *header* da mensagem). A existência de um único canal na arquitetura e o facto de o mesmo usualmente apenas necessitar do desencadeamento de uma interrupção por *context-switch* (em *multicore* não há necessidade de armazenamento), faz com que a utilização de um buffer circular para o armazenamento de interrupções se torne obsoleto.

Não obstante, este foi o método escolhido, uma vez que permite a escalabilidade do mecanismo de comunicação para uma futura expansão para uma arquitetura *multiguest*. A Figura 4.14 demonstra como esta estrutura de armazenamento permite a alocação de IPIs de diferentes proveniências duma forma organizada sequencial e temporalmente. A descodificação da origem da interrupção é realizada pela numeração dessa mesma interrupção ao nível de cada *guest*.

Ainda no campo das interrupções inter-partição, são necessárias algumas modificações para que o monitor consiga fazer o encaminhamento das mesmas para a

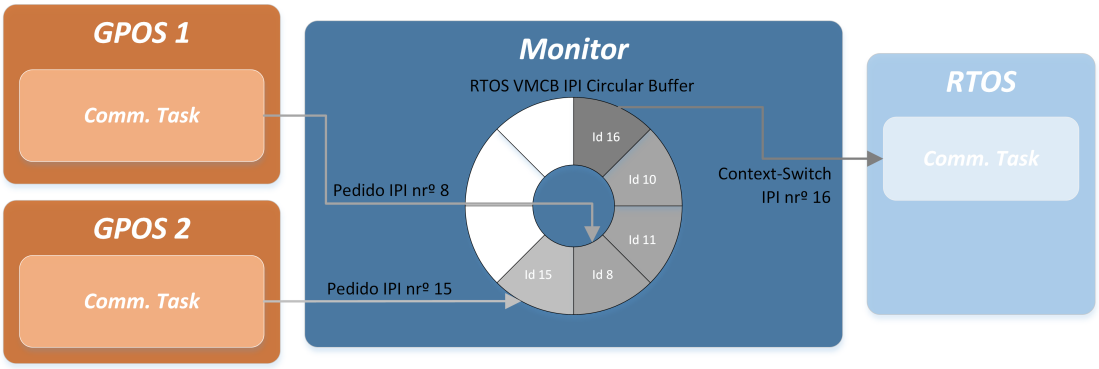


Figura 4.14: Buffer Circular de Armazenamento de IPIs numa Arquitetura *Multiguest*

partição correspondente. Para que este encaminhamento seja possível é necessário que o monitor introduza no bloco de contexto de cada *guest* uma identificação (*guest_id*). Este id será introduzido nas estruturas dos parâmetros de cada canal de cada *guest* e posteriormente propagado juntamente com a requisição de interrupções inter-*core*. Estas requisições já preveem a propagação do *"guest_id"* como observado no excerto de código retirado do *smc handler* de alto nível abaixo apresentado, faltando, no entanto, os mecanismos internos do monitor para o descodificar.

Listagem 4.15: Utilização do *guest_id* reservada para futura utilização

```
1 switch(arg0) {
2 (...)
3 case (-34):
4 #ifdef CONFIG_COMM
5 request_monitor_ipi (arg1, arg2); //arg1 = ipinr; arg2 =
      guest_id;
6 #endif
7 break;
8 (...)
9 };
```

Os mecanismos implementados preveem ainda a possibilidade de reutilização de IPIs, visto que as mesmas se encontram em número limitado nos processadores ARM. Devido ao envio do "*guest_id*" juntamente com o número de IPI, um mesmo *guest* pode utilizar a mesma IPI numerada para notificar diferentes *guests*. Do ponto de vista da receção de IPIs, um *guest* identifica a notificação pela numeração da IPI, não sendo possível que esta seja reutilizada, no entanto um segundo *guest* poderá receber uma IPI com a mesma numeração. A reutilização de IPIs apenas

se pode dar dentro do mesmo *core*, se as mesmas tiverem o mesmo estado de segurança associado, visto que a configuração dinâmica é relativamente lenta.

De uma forma geral, o mecanismo de comunicação está preparado para uma futura expansão *multiguest*. Para provar este conceito foram instanciados múltiplos canais entre o mesmo par de *guests* disponíveis no LTZVisor, como observado na Figura 4.15. Devido à abstração fornecida pelo próprio canal de comunicação, para cada *guest*, cada canal representa um caminho de comunicação com um *guest* diferente. Esta prova de conceito permite-nos garantir a escalabilidade do mecanismo, sendo apenas necessário, como referido anteriormente, a alteração condizente da VMCB de cada *guest*.

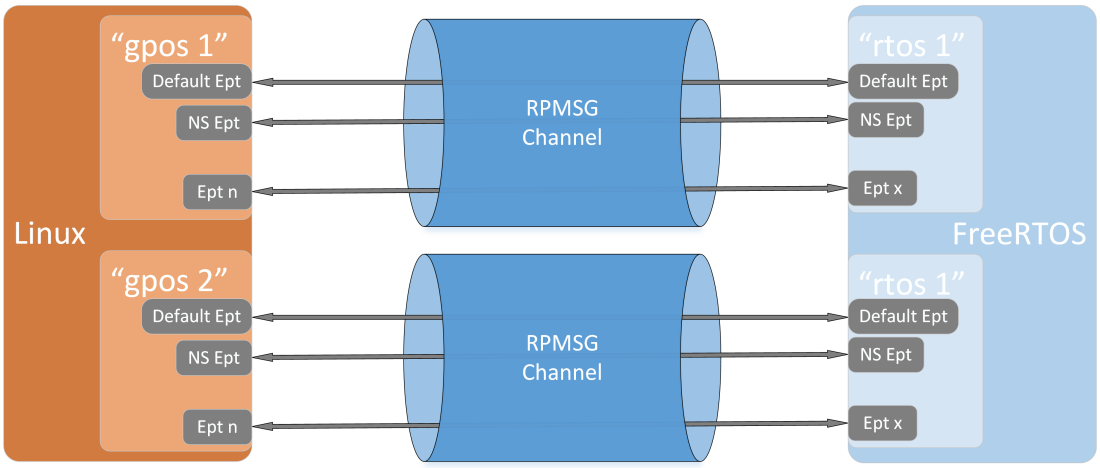


Figura 4.15: Instanciação de dois canais RPMsg num mesmo par de SOs, simulando um sistema *multiguest*

A comunicação especificada pelo OpenAMP tem ainda uma característica apreciada no mundo de sistemas embebidos que é a de suportar a utilização de comunicação RPC (*Remote Procedure Calls*). Esta técnica permite a um dos pontos da comunicação invocar uma função presente no outro ponto da comunicação através de comandos simples como *open, close, read, write* e *rpc*. Existem já aplicações proxy presentes na versão 2015.4.1 do XSDK prontas a serem utilizadas, elevando o nível de abstração da comunicação inter-partição implementada.

Resultados

Neste capítulo serão apresentados os resultados obtidos para as diferentes funcionalidades implementadas no LTZVisor, nomeadamente da configuração *multicore* e da comunicação agora suportada pela *framework*. A caracterização do sistema LTZVisor é realizada apenas com foco nas alterações implementadas, uma vez que a caracterização inicial (pré-alterações) do mesmo pode ser consultada em [5].

A avaliação global do sistema foi obtida com recurso à plataforma de hardware Xilinx Zynq-7000 SoC Zedboard. Referida na Subsecção 3.2, esta plataforma contém dois processadores Cortex-A9 com suporte TrustZone, coerente com as necessidades da dissertação. De entre os recursos disponibilizados, foram utilizados os timers TTC0 e Global Timer, assim como a ferramenta PMU (Performance Monitoring Unit) para a obtenção dos diferentes resultados.

5.1 *Multicore*

Os resultados analisados nesta Secção englobam todas as configurações da *framework*, assim como dos respetivos *guests*. Inicialmente é analisado o impacto das diferentes configurações no sistema LTZVisor em termos de *overhead* de memória.

De seguida serão retirados os resultados de desempenho dos dois *guests* inclusos nas seguintes configurações: i) nativamente, *i.e.,* sem qualquer tipo de virtualização; ii) inseridos na configuração *single-core* do LTZVisor, sendo que o *guest* não alvo da caracterização estará em estado *idle*; iii) inseridos na configuração *multicore* do LTZVisor de forma idêntica ao teste anterior.

Por fim, de forma a demonstrar o potencial da configuração *multicore*, serão repetidos os testes acima referidos em (ii) e (iii), no entanto, o *guest* não envolvido nos *benchmarks* será alvo de um nível de carga de trabalho ajustável. Este teste será apenas repetido para o GPOS, uma vez que o RTOS possui prioridade em relação ao GPOS, não existindo modificações no desempenho deste último.

5.1.1 *Footprint* de Memória

Para a obtenção dos valores de *overhead* de memória, foi utilizada a ferramenta SIZE da *toolchain* ARM GNU. Esta ferramenta revela o tamanho do executável analisado e dos seus diferentes segmentos (.text, .bss e .data). As secções dos executáveis possuem significados diferentes:

- **text** – Esta secção contém o código executável, isto é, as instruções e funções a executar. Contém também o valor das variáveis constantes e as *vector tables*;
- **data** – Nesta secção encontram-se as variáveis inicializadas do sistema;
- **bss** – As variáveis não inicializadas serão colocadas nesta secção. É aqui também que serão incluídas as *stacks* dos diferentes modos de execução, assim como a zona de variáveis dinâmicas denominada de *heap*.

Na Tabela 5.1 estão registados os valores obtidos de *footprint* de memória para as configurações *single-core* e *multicore* da *framework* LTZVisor. De forma a evidenciar o impacto destas configurações, os valores obtidos não contêm o *guest* Linux, uma vez que este, devido à sua complexidade, apresenta valores demasiado grandes comparativamente com os do sistema, o que resultaria numa camuflagem do verdadeiro impacto. Os valores englobam o código correspondente ao *boot* da plataforma Zynq-7000, as drivers da respetiva plataforma, as *vector table* do modo monitor em ambos os *cores* se aplicável, a implementação própria das bibliotecas standard e ainda o *guest* seguro FreeRTOS. O *footprint* de memória encontra-se na Tabela 5.1 em formato decimal e em formato normalizado (em relação à configuração *single-core*) para que a comparação possa ser facilmente interpretada.

Tabela 5.1: *Footprint* de Memória do Sistema LTZVisor + FreeRTOS - *multicore*

Ficheiros	**Secções**			**Tamanho Total**	
	text	**data**	**bss**	**dec**	**hexa**
LTZVisor_single-core	46488	284	423984	470756	72ee4
	1,0000	1,0000	1,0000		1,0000
LTZVisor_multicore	52868	284	452656	505808	7b7d0
	1,1372	1,0000	1,0676		1,0745

Verifica-se, através de uma rápida análise, que a introdução da nova configuração tem um impacto mediano no sistema geral, nomeadamente de 7,45% no valor total. Este aumento, no entanto, por se verificar apenas nas secções .text e

.bss, significa um aumento considerável de código. O aumento de código, justificado pela inclusão de uma segunda *vector table* e a inicialização adicional do *core* secundário, apresenta ainda um aumento considerável, na ordem dos 13,72

Os resultados foram retirados da última versão atualizada do LTZVisor, contendo a versão 4.0 do Linux, recorrendo ao uso de uma variável no Makefile denominada de "*MP_AMP*". Esta variável ativa/desativa a *flag* de compilação referida em secções anteriores como "*CONFIG_AMP*". Permite também que o Makefile não compile certos ficheiros específicos da configuração *multicore*, presentes na Listagem 5.1 abaixo inclusos pela variável "*SRC_ARCH_ARMV7_AMP*".

Listagem 5.1: Compilação dependente da variável *MP_AMP*

```
1 ifeq ($(MP_AMP), y)
2 ${CC} ${CC_FLAGS} ${INCLUDES} -c ${SRC_ARCH_ARMV7}/*.S
3 else
4 ${CC} ${CC_FLAGS} ${INCLUDES} -c $(filter-out ${
      SRC_ARCH_ARMV7_AMP}, $(wildcard ${SRC_ARCH_ARMV7}/*.S))
5 endif
```

5.1.2 RTOS

Uma das características mais apreciadas em sistemas operativos de tempo-real é, como o nome indica, a sua capacidade de garantir as deadlines temporais. Esta característica pode ser medida em termos de desempenho para cada uma das funcionalidades que tenham influência no comportamento do RTOS, e influenciem consequentemente o cumprimento das ditas deadlines.

Para a medição do nível de performance do FreeRTOS foram realizados os micro-*benchmarks* pertencentes à *"Suite"* Thread-Metric disponibilizados de forma *"open-source"* pela Express Logic[1]. Este conjunto de micro-*benchmarks* é facilmente adaptado e portado para diferentes RTOS em diferentes arquiteturas de processadores. Estes micro-*benchmarks* permitem avaliar o desempenho de um determinado RTOS para um determinado serviço, usualmente caracterizado como influente no cumprimento das deadlines de tempo-real.

Os *benchmarks* escolhidos pela *"Suite"* são usualmente observados em aplicações próprias de RTOS. Abaixo encontram-se listadas as respetivas tarefas, bem como a sua denominação no decorrer da demonstração dos respetivos resultados:

- ***Cooperative Contex-Switch*** (TM_Coop_CS) – avalia a troca cooperativa de contextos entre tarefas, isto é, o tempo despendido pelo FreeRTOS

[1] http://rtos.com/PDFs/MeasuringRTOSPerformance.pdf

na comutação de tarefas, que voluntariamente libertam o controlo do processador;

- ***Preemptive Context-Switch*** (TM_Preemp_CS) – avalia a capacidade de troca de contextos do RTOS num contexto de sobreposição de tarefas de diferentes prioridades, isto é, avalia a capacidade do RTOS em interromper tarefas para o escalonamento de tarefas de maior prioridade;

- ***Interrupt Handling*** (TM_FIQ_Hand) – avalia a capacidade do sistema operativo no atendimento a interrupções e posterior escalonamento de tarefas;

- ***Interrupt Preemption*** (TM_FIQ_Preemp) – avalia a capacidade do RTOS em escalonar tarefas ativadas pela própria rotina de serviço à interrupção, com uma prioridade menor da tarefa que despoletou a dita interrupção;

- ***Memory Allocation*** (TM_Mem_Alloc) – avaliação do sistema de alocação de memória através da alocação e libertação sistemática de um mesmo bloco de memória de 128 bytes;

- ***Message Processing*** (TM_Msg_Proc) – apreciação da performance dos sistemas de *queues* de mensagens implementados pelo RTOS a avaliar;

- ***Synchronization Processing*** (TM_Sync_Proc) – avalia a performance dos semáforos implementados no RTOS.

A avaliação do FreeRTOS foi feita nas seguintes configurações: nativo, *single-core* e *multicore*, esta última com a particularidade de ser num *core* diferente. Os resultados obtidos da versão nativa do FreeRTOS permitirão, comparativamente, avaliar o *overhead* introduzido pelo sistema de virtualização LTZVisor. Os tempos retirados para as diferentes configurações permitirão tirar ilações sobre o efeito da mudança de *core* no SO FreeRTOS.

A Figura 5.1 demonstra os resultados obtidos para os micro-*benchmarks* efetuados. Cada resultado de um micro-*benchmark* consistiu na realização consecutiva de 500 testes (cada um com a duração de 30 segundos), sendo posteriormente obtidos os valores de média e variância respetivos. Os valores médios representados no gráfico são os valores dos contadores dos diferentes testes. Estes são responsáveis pela contabilização de passagens do código por uma situação definida pelo próprio teste, e pretendem quantificar assim o desempenho do sistema para o determinado serviço em avaliação.

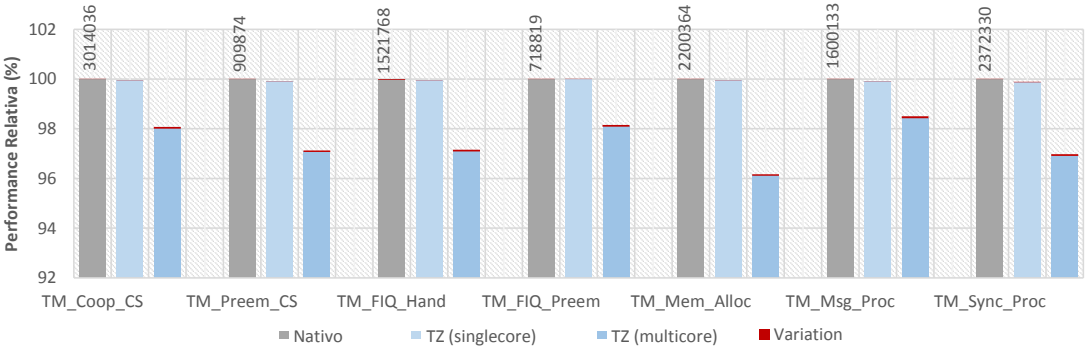


Figura 5.1: Comparação entre o FreeRTOS Nativo e inserido no LTZVisor *single-core* e *multicore*

Através da observação do gráfico presente na Figura 5.1, concluí-se que a presença do sistema de virtualização LTZVisor induz no FreeRTOS um *overhead* insignificante, em média menor de 0,25%. Estes valores justificam-se pela política de escalonamento escolhida para o LTZVisor, assim como pela arquitetura de virtualização assistida pelo *hardware* proporcionado pela tecnologia ARM TrustZone, em que a cada *system tick* o controlo da execução do processador é roteada para o FreeRTOS (se não o for no instante anterior). Esta abordagem permite que o FreeRTOS evite, a não ser que por vontade própria, a perda do controlo do processador, originando assim níveis de performance concorrentes com os observados na sua versão nativa.

Os resultados obtidos para a configuração *multicore* do LTZVisor denotam um acréscimo de *overhead* do sistema operativo de forma geral, situando-se entre os 4% e os 2,75%. Os valores obtidos na configuração *single-core* levam a crer que o *overhead* introduzido não é respeitante ao sistema responsável pela virtualização, mas sim a fatores externos. Entre os fatores externos com possível interferência nos resultados destacam-se:

i) A concorrência aos recursos da plataforma (principalmente o barramento de memória e instruções) resultante da simultaneidade de execução dos dois sistemas operativos. Esta característica é demonstrada nos valores de variância representados e reforçada pelos valores de performance obtidos nos primeiros 30 segundos iniciais, revelados bastante mais baixos (5 a 10%), coincidente com a altura de *boot* do Linux. Esta situação é inerente ao design *multicore*, uma vez que na configuração *single-core* o Linux sofre de *"starvation"*, não sendo executado durante a execução dos micro-*benchmarks* (o nível de carga do FreeRTOS é de 100% não chegando a executar a tarefa *idle*, o ponto de escalonamento do sistema). Testes extras realizados descartam esta hipótese

como fator exclusivo de perda de performance, uma vez que em testes efetuados sem a presença do Linux (FreeRTOS continua no *core* 1), o *overhead* ainda persiste situando-se entre os 1,75% e 3%;

ii) Possível característica do próprio *hardware* da plataforma, em que a latência dos próprios recursos pode ser maior quando executado no *core* secundário (*core* 1). Este fator está presente exclusivamente nos resultados do FreeRTOS inserido na arquitetura *multicore*, pois, por uma questão de design (justificado na Subsecção 3.3), o mesmo foi migrado para o *core* secundário da plataforma.

5.1.3 GPOS

Para a caracterização dos níveis de performance do Linux foi utilizada a ferramenta LMBench [31]. O LMBench é um conjunto de micro-*benchmarks* amplamente utilizado que permite realizar as medições de uma variedade de aspetos de performance do sistema, como a latência e largura de banda. Este conjunto tem como alvo os sistemas UNIX, tendo sido escrito na linguagem portável ANSI-C utilizando as interfaces POSIX. O LMBench 3.0 inclui mais de 40 micro-*benchmarks* categorizados em latência, largura de banda e "outros".

Na avaliação realizada, recorreu-se aos micro-*benchmarks* lat_ops, que, como o nome indica, permitem avaliar a latência das operações aritméticas realizadas pelo CPU. As operações englobam a multiplicação, adição, divisão e *bitwise* para diferentes tipos de variáveis. Este teste permite fazer uma avaliação geral da performance do CPU habitado pelo Linux.

Nas configurações *single-core* e *multicore*, o *guest* seguro, o FreeRTOS, foi mantido em estado *idle*. A única interferência realizada pelo FreeRTOS consiste no *system tick*, responsável pela atualização do *scheduler* do RTOS. Este foi configurado com um intervalo de tempo de 1 milissegundo.

Os resultados encontram-se apresentados na Figura 5.2 sob a forma de performance relativa (normalizada) com acréscimo da respetiva variância dos resultados obtidos. São ainda apresentados os valores absolutos para a configuração nativa do Linux. Para cada teste foram executadas 100 experiências consecutivas, sendo que cada micro-*benchmark* foi configurado para realizar 10 aquecimentos e 1000 repetições (-W 10 -N 1000), englobando um total de 100.000 amostras por barra. As amostras correspondem ao tempo (normalizado) que o processador demorou a executar cada uma das 18 tarefas atribuídas pelo "*lat_ops*".

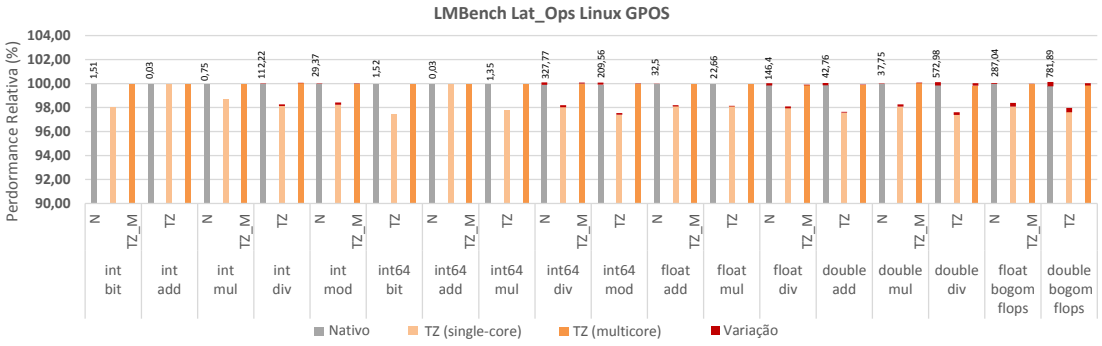


Figura 5.2: Resultados do *Lat_Ops* para as diferentes configurações do Linux

Através de uma observação minuciosa do gráfico presente na Figura 5.2, repara-se que a performance apenas apresenta reduções na configuração *single-core*. Esta redução, apesar de tudo, fixa-se apenas nos 2% e está relacionada com a interferência introduzida pelo *tick* do sistema operativo FreeRTOS. Os valores das operações "int add" e "int64 add", pelo seu valor reduzido (~0,03) e a incerteza correspondente (2 algarismos significativos) devido à resolução do *timer* não permitem acompanhar a tendência dos restantes valores.

Contrariamente ao modo *single-core*, a Figura 5.2 permite concluir que a interferência do sistema global no Linux em modo *multicore* é praticamente nula em comparação com os resultados nativos, podendo desde já prever a vantagem inerente à migração para uma arquitetura *multicore*.

5.1.4 RTOS em Carga

Os resultados obtidos na secção anterior permitem ter uma ideia da vantagem de utilização de uma arquitetura *multicore*, no entanto os valores obtidos não demonstram as potencialidades da mesma. Para que estas pudessem ser reveladas, recorreu-se à medição dos mesmos testes de performance da secção anterior em ambientes de carga diferentes do sistema, *i.e.,* foi variado o nível de carga do FreeRTOS.

Para além do valor do nível de carga do FreeRTOS, foi ainda variado o intervalo de tempo do *tick* do FreeRTOS. Este intervalo de tempo, em RTOS, fixa-se usualmente entre os 500 microssegundos e os 20 milissegundos. Para a realização do teste, foram utilizados os intervalos de tempo 500 μs, 1 ms, 2 ms, 5 ms, 10 ms e 20 ms como *ticks* do FreeRTOS.

Para a variação do nível de carga, foi configurado um segundo *timer* com um valor menor em relação ao *tick* do FreeRTOS. Como observado na Figura 5.3,

o *tick* do FreeRTOS ativa uma tarefa responsável pelo nível de carga (na realidade será um *loop* infinito para que o acesso ao barramento não seja concorrido), ativando também o timer responsável por desativar essa mesma tarefa. O timer foi programado para ter um quarto, dois quartos e três quartos do valor do *tick*, variando assim o nível de carga do FreeRTOS entre os 25%, 50% e 75%.

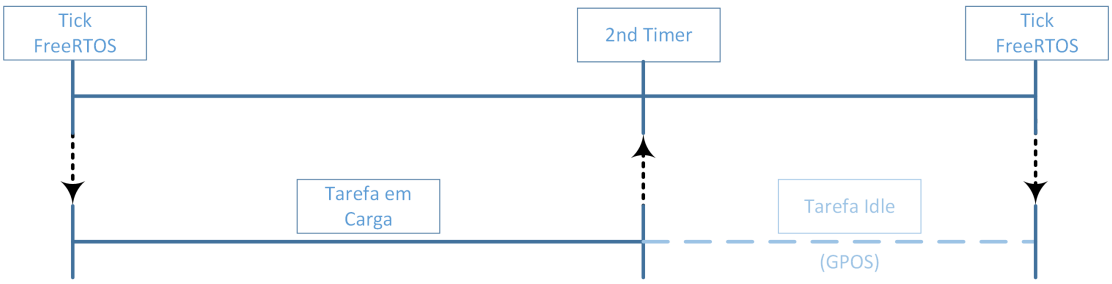


Figura 5.3: Nível de Carga do FreeRTOS dado por um segundo *timer*

O gráfico apresentado na subsecção anterior representava os valores médios normalizados para cada ação do *lat_ops*. O gráfico da Figura 5.4 apresenta a média total desses valores normalizados para cada nível de carga, *tick* do FreeRTOS (escala logarítmica) e configurações diferentes. Isto é, cada valor (representado por um marcador diferente na Figura 5.4) é uma média da performance global do Linux para o *benchmark lat_ops* inserido em ambientes de execução diferentes, totalizando 180.000 amostras por marca (18 operações * 100.000 amostras – cada *"lat_ops"* consiste em 18 operações aritméticas) e 8.640.000 amostras no global (180.000 amostras * 6 *"ticks"* * 4 níveis de carga * 2 configurações).

Os valores globais de performance do Linux, quando inseridos no sistema LTZVisor *single-core*, são inversamente proporcionais aos valores de carga do FreeRTOS. Isto é, para um valor de carga do FreeRTOS de 75%, o Linux apresenta valores de desempenho próximos dos 25%. Este fenómeno está relacionado com a política de prioridade de tempo-real do sistema virtualizado, em que o Linux é interrompido pelo FreeRTOS quando este necessita de tempo de processamento, podendo mesmo chegar a sofrer do fenómeno denominado *"starvation"* (sem tempo de execução algum ou suficiente).

Ademais, quanto menor o intervalo de tempo do *system tick* do FreeRTOS, maior será o número de vezes que o Linux será interrompido, originando uma queda acrescida na performance do mesmo. Este fenómeno é visível através dos diferentes níveis de carga, sendo que no nível de carga 0% a interferência na performance é exclusiva do *system tick* do FreeRTOS. Apesar de exclusiva nesse nível de carga, é também onde apresenta os valores de *overhead* menos acentuados (abaixo dos 2%). Para o nível de carga de 75% o *overhead* do *tick* do FreeRTOS representa até

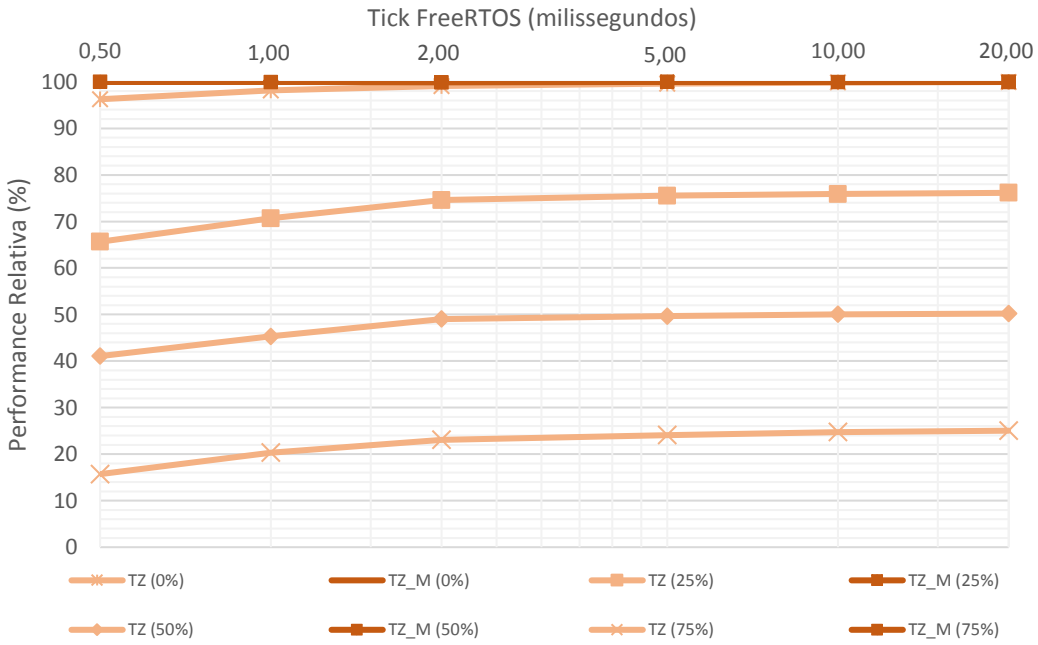


Figura 5.4: Resultados do *Lat_Ops* para as diferentes configurações do Linux, variando a carga e *tick* do FreeRTOS

10% do *overhead* total (para um *tick* de 500 μs), uma vez que neste nível de carga o tempo de execução aumenta, aumentando também o número de interrupções.

A tendência será a dos resultados do desempenho das operações fixarem-se no valor inverso à carga do FreeRTOS (100%-nível de carga), assim que o valor do *tick* do FreeRTOS deixe de ter influência (para 20 milissegundos a sua influência nunca ultrapassa os 0,2%, para 10 os 0,5% e para os 5 millisegundos, o valor máximo situa-se nos 1%).

Como esperado, os resultados do Linux em configuração *multicore* mostram-se independentes, quer do nível de carga, quer do *tick* do FreeRTOS, apresentando valores a rondar os 100%. Estes níveis de performance são apenas condizentes com a utilização de plataformas *multicore* em que o nível de carga de um sistema operativo em nada influencia o outro em designs assimétricos (excetuando possíveis concorrências ao barramento). Esta vantagem é amplamente observável para o nível de carga de 75%, em que o *overhead* observado no Linux se encontra entre os 84,76% e 74,88% na arquitetura *single-core*, passando para 0% na arquitetura *multicore*. Esta diferença será tanto maior quanto o aumento dos valores de carga e frequência do *tick* do FreeRTOS.

5.2 Comunicação

Os resultados expostos nesta secção foram retirados utilizando a ferramenta PMU (Performance Monitoring Unit), excetuando para o *footprint* de memória. Esta ferramenta, arquiteturalmente *tightly-coupled* no processador do SoC Zynq-7000, providencia 6 *timers* de modo a adquirir estatísticas sobre as operações do processador e eventos do sistema. Cada contador permite a contabilização de 58 eventos disponíveis no processador Cortex-A9.

Aparte dos contadores fornecidos, a PMU disponibiliza ainda um contador especial denominado de Cycle Counter (CC). Este contador específico possibilita a contagem de ciclos de relógio entre pontos determinados no código a executar, isto é, a partir da sua utilização é possível medir o tempo de execução de um determinado número de instruções ou operações.

Na Zedboard, o relógio do processador foi programado para uma frequência de 667 MHz, cada ciclo ocorrendo a cada ~1,49 nanossegundos, sendo que uma instrução que demore um ciclo de relógio, demorará aproximadamente 1,49 nanossegundos a executar (a componente CC do PMU irá revelar o valor 1).

5.2.1 *Footprint* de Memória

De forma semelhante à Subsecção 5.1.1, os resultados do *footprint* de memória do suporte de comunicação foram retirados com recurso à ferramenta SIZE presente na *toolchain* utilizada. A Tabela 5.2 mostra os valores obtidos. Estes encontram-se no seu formato original e em formato normalizado em relação às respetivas configurações sem o suporte da comunicação inter-partição.

Tabela 5.2: *Footprint* de Memória do Sistema LTZVisor + FreeRTOS – comunicação

Ficheiros	**Secções**			**Tamanho Total**	
	text	**data**	**bss**	**dec**	**hexa**
LTZVisor_single-core	52868	284	452656	505808	7b7d0
	1,0000	1,0000	1,0000		1,0000
LTZVisor_single-core com comunicação	80548	752	453316	534616	82858
	1,5246	2,6479	1,0015		1,0570
LTZVisor_multicore	46488	284	423984	470756	72ee4
	1,0000	1,0000	1,0000		1,0000
LTZVisor_multicore com comunicação	80312	752	424644	505708	7b76c
	1,7276	2,6479	1,0016		1,0742

Existe a necessidade de dividir a comparação dos resultados por configurações, uma vez que a própria implementação dos mecanismos de comunicação muda de acordo com a configuração inserida, ainda que superficialmente.

Pelos resultados obtidos, revelados na Tabela 5.2, pode-se concluir que a inserção de mecanismos de comunicação teve um impacto global mediano no sistema situando-se nos 5,7 % e 7,42%, para as diferentes configurações *single-core* e *multicore*, respetivamente. A maior alteração observa-se na secção da .data, chegando a ultrapassar um aumento de 160%. Este aumento substancial deriva da utilização de estruturas estáticas para o armazenamento dos parâmetros dos dispositivos virtuais VirtIO (*resource_table*) e o armazenamento das especificidades da plataforma integradas na camada de abstração do processador e utilizadas pelo componente RPMsg_Super.

Verifica-se ainda um aumento de 52,46% e 72,75% na secção .text em relação às configurações *single-core* e *multicore* respetivamente. Isto representa um aumento considerável de código no sistema. A explicação centra-se na inclusão total ou parcial dos componentes Remoteproc, RPMsg e VirtIO, sendo que aos dois últimos foi adicionado o suporte para ambos os modos de comunicação (*master* e *slave*). Na secção de dados dinâmicos (.bss), a alteração é insignificante não chegando a 0,2%.

Contrariamente aos designs de arquitetura suportados pelo LTZVisor, o suporte para mecanismos de comunicação envolveu alterações no *guest* Linux. A Tabela 5.3 representa o *footprint* de memória dessas alterações.

Tabela 5.3: *Footprint* de Memória do Linux– comunicação

Ficheiro	**Secções**			**Tamanho Total**	
	texto	**data**	**bss**	**decimal**	**hexadecimal**
vmlinux	6898491	253388	181876	7333755	6fe77b
	1,0000	1,0000	1,0000		1,0000
vmlinu_comm	6931536	254364	182004	7367904	706ce0
	1,0048	1,0039	1,0007		1,0047

Similarmente aos resultados obtidos no *footprint* de memória do sistema LTZVisor e FreeRTOS, os resultados para o *guest* Linux tornam visível o aumento do código, bem como da memória estática no executável com a inclusão dos componentes de suporte de comunicação. Os valores situam-se em 0,48% e 0,39% respetivamente. Estes valores podem parecer insignificantes pela sua ordem de grandeza, no entanto devido à complexidade do sistema operativo em análise e respetiva dimensão pode-se concluir que são valores com alguma influência na secção das drivers do mesmo.

5.2.2 Desempenho do *Context-Switch*

Excetuando o uso esporádico das instruções *smc* para a execução de algo pretendido pelos SOs *guests*, a única fonte de *overhead* do sistema LTZVisor centra-se na comutação de mundos, isto é, na troca de contextos destes. Esta troca acontece mediante duas ações distintas já reveladas anteriormente: i) ponto de escalonamento forçado para o mundo não-seguro pela tarefa *idle* do *guest* RTOS e ii) através do *trigger* de uma interrupção FIQ enquanto o processador se encontra no estado de execução do mundo não-seguro, despoletando uma comutação para o mundo seguro.

O *overhead* da troca de contextos, apesar de já avaliados em [5], encontram-se entre as métricas mais importantes da caracterização da solução e devido à introdução de *overhead* extra por parte da comunicação deverão ser reavaliados. A implementação do suporte de comunicação introduziu um certo número de instruções responsáveis pelo desencadeamento das interrupções inter-partição durante a altura de comutação de contextos dos *guests*. Esta adição necessita de ser avaliada, de modo a podermos ter uma caracterização total dos mecanismos implementados.

Recorreu-se a três testes distintos para cada tipo de comutação (mundo seguro para mundo não-seguro (1) e o oposto (2)): com a comunicação inter-partição desativada (1.1 e 2.1), com a comunicação inter-partição ativa, em que o buffer de armazenamento de interrupções se encontra vazio (1.2 e 2.2), e ainda outro teste no qual este último se encontra preenchido (1.3 e 2.3). O primeiro teste permite-nos obter um termo de comparação prévio às alterações dos mecanismos de comunicação implementados, sendo que os restantes perfazem o melhor e pior caso possível, sendo o último a fornecer o WCET (*Worst Case Estimated Time*) do tempo de comutação para os dois mundos.

Tabela 5.4: Valores de ciclos de relógio para comutação de mundos em diferentes cenários

Cenário	**Estado Buffer**	μ (**ciclos de relógio**)	δ (**ciclos de relógio**)
Mundo S-Mundo NS (1)	- (1.1)	1675	39,52
	vazio (1.2)	2150	52,79
	preenchido (1.3)	3885	48,74
Mundo NS-Mundo S (2)	- (2.1)	3411	45,74
	vazio (2.2)	3990	49,13
	preenchido (2.3)	5610	54,68

A Tabela 5.4 expõe os valores obtidos diretamente da ferramenta PMU na

medição das trocas de contexto nos 3 cenários supracitados. Os valores representam a média (μ) e a variância (δ) para 20 repetições. Para a obtenção dos valores em unidades de tempo, será necessário a multiplicação dos valores obtidos pelo tempo de cada ciclo de relógio da plataforma em testes. Na Zedboard esse valor é de ~1,49 *ns*.

A comutação para o mundo não-seguro (1.1) revela-se bastante mais rápida, demorando cerca de 2,51 μs. No cenário oposto (2.1), este tempo evolui para 5,12 μs. A adição do mecanismo de comunicação incrementa o *overhead* da comutação, obtendo valores diferentes segundo a necessidade de realmente enviar uma interrupção inter-partição (1.3 e 2.3) ou de não ter essa necessidade (1.2 e 2.2). Neste último caso durante a comutação é apenas feita uma rápida leitura dos valores de índice do buffer, reiniciando de seguida a comutação. Para estes casos, o aumento de *overhead* ficou-se pelos 28,36% e 16,97% para os casos 1.2 e 2.2 respetivamente.

Para os casos 1.3 e 2.3, estes valores são bastante agravados. O envio da interrupção inter-partição e respetiva reconfiguração do bloco de controlo dos diferentes *guests* tem uma interferência significativa na comutação de mundos, representando um aumento de 132% e 64,4% respetivamente. O aumento da variância e consequentemente a deterioração do determinismo têm também um impacto negativo no desempenho global do sistema.

Pelos valores de *overhead* obtidos, pode-se concluir que a adição da virtualização de interrupções inter-partição, como já esperado na altura do design, contribui negativamente no desempenho da própria comunicação, assim como do desempenho global do sistema. Esta decisão, além de necessária na configuração *single-core*, representou um *trade-off* entre performance e segurança no sistema, sendo que esta virtualização evita uma vulnerabilidade que representa também um dos maiores focos de ataque em comunicações inter-partição, as notificações inter-*guest*.

Esta utilização segura das interrupções inter-partição na comunicação vem destacar a importância de migração para plataformas *multicore*, onde a virtualização das interrupções apenas tem um impacto negativo na performance da própria comunicação (quase nulo), não afetando as métricas restantes do sistema global.

5.2.3 Caracterização da Comunicação

A caracterização da comunicação seguiu um formato *best case estimated time*. Esta decisão foi influenciada pelos fatores externos capazes de interferir no desempenho global da comunicação, que pela sua quantidade e diversidade impediriam

uma caracterização adequada dos mesmos. Desta forma, a caracterização significará o *best case scenario* da comunicação, sendo que a utilização em cenários normais terá a sua performance degradada pelos múltiplos fatores abaixo descritos:

- **Número de tarefas no FreeRTOS** – qualquer variação neste número influenciará a própria comunicação, independentemente da prioridade dessas tarefas em relação às de comunicação. Tarefas com uma prioridade inferior serão escalonadas a seguir às tarefas de comunicação, provocando um atraso entre o envio da mensagem e o envio da notificação. Uma tarefa de maior prioridade provocará um atraso no escalonamento das tarefas de comunicação e respetivo suporte;

- **Número de aplicações no Linux** – de igual forma, no GPOS, um número variável de aplicações terá influência nos tempos medidos, uma vez que podem ser escalonadas antes ou até durante o processamento das mensagens de comunicação. Qualquer aplicação com uma interrupção IRQ associada, provocará o mesmo efeito no desempenho da comunicação;

- **Interrupções FIQ** – uma interrupção FIQ no decorrer do processamento de uma mensagem por parte do Linux irá criar um atraso nesse mesmo processamento, resultando em leituras alternáveis da performance da comunicação;

- ***Tick FreeRTOS*** – para além de ser incluída na interferência acima referida por ser uma interrupção FIQ, o próprio *tick* do FreeRTOS poderá indicar a frequência com que uma notificação inter-partição poderá ser despoletada (direção Linux->FreeRTOS). Se esta for a única interrupção ativa no FreeRTOS, será também a única a despoletar a comutação de mundos e consequentemente do envio da notificação. A sua frequência terá influência no desempenho da comunicação, assim como no tempo entre a solicitação de envio de notificação e o envio real.

A caracterização da comunicação seguirá o mesmo padrão em ambas as configurações da *framework*, independentemente das alterações realizadas para a minimização de interferências num determinado design. Os mecanismos de comunicação *multicore* podem ser influenciados pelos dois primeiros fatores supramencionados, isto é, pela prioridade atribuída às tarefas responsáveis pela comunicação e respetivo suporte. No entanto, a interferência inter-*guest*, principalmente vista no GPOS, é completamente anulada pelo design de virtualização assimétrico *multicore*.

Para diminuir ao máximo o número de interferências nos tempos medidos na caracterização da comunicação, o FreeRTOS encontra-se desprovido das suas tarefas de tempo-real, excetuando as tarefas de comunicação necessárias. Estas apenas se encontram ativas enquanto é enviado um determinado número de mensagens ou aquando da receção de interrupções inter-partição (quer de mensagem recebida, quer de buffer disponibilizado).

Com igual propósito, a caracterização foi realizada do lado do FreeRTOS (direção da comunicação FreeRTOS->Linux) pois, desta forma, pode-se adequar o envio da notificação à altura exata do envio da mensagem. Isto deve-se ao facto de na arquitetura *single-core* a notificação ser apenas enviada na altura do *context-switch*. Estando o FreeRTOS com uma única tarefa ativa (comunicação), o envio da mensagem será procedido pela tarefa *idle* e consequentemente, a comutação de mundos. O *tick* do FreeRTOS foi ainda alterado para 150 μs, para que o mesmo não interrompesse o processamento de mensagens do Linux e não interferisse assim nos tempos medidos.

Todos os resultados expostos neste capítulo foram retirados com o mundo seguro configurado como *master* da comunicação. Esta configuração é a mais adequada à arquitetura do sistema, uma vez que o mundo seguro é realmente a partição com maior prioridade no sistema, fazendo inclusivamente parte da TCB. Naturalmente, será apropriado oferecer ao FreeRTOS o controlo sobre a gestão dos buffers e memória.

O *throughput* da comunicação para as diferentes configurações *multicore* e *single-core* foi obtido variando diferentes parâmetros:i) o tamanho da mensagem enviada e ii) o número de mensagens enviadas. Foram realizados múltiplos testes de latência, desde o envio de uma mensagem até à receção da mesma pelo *call-back* associado à mensagem na partição oposta. Os valores de latência obtidos permitirão concluir o *throughput* da comunicação para diferentes parâmetros. O *throughput* é apresentado em bits por segundo (bps).

5.2.3.1 Variação de Bytes Enviados numa Mensagem

Nesta secção da caracterização da comunicação foi enviada uma única mensagem de cada vez, sendo o seu tamanho variado entre os valores mínimo (1 byte) e máximo (512 bytes) possíveis suportados pelo buffer de uma mensagem na comunicação RPMsg. O valor mínimo e máximos são na verdade 17 e 528 bytes, devido ao *header* obrigatório neste tipo de comunicação *connectionless*, no entanto este *header* não será contabilizado na caracterização da comunicação.

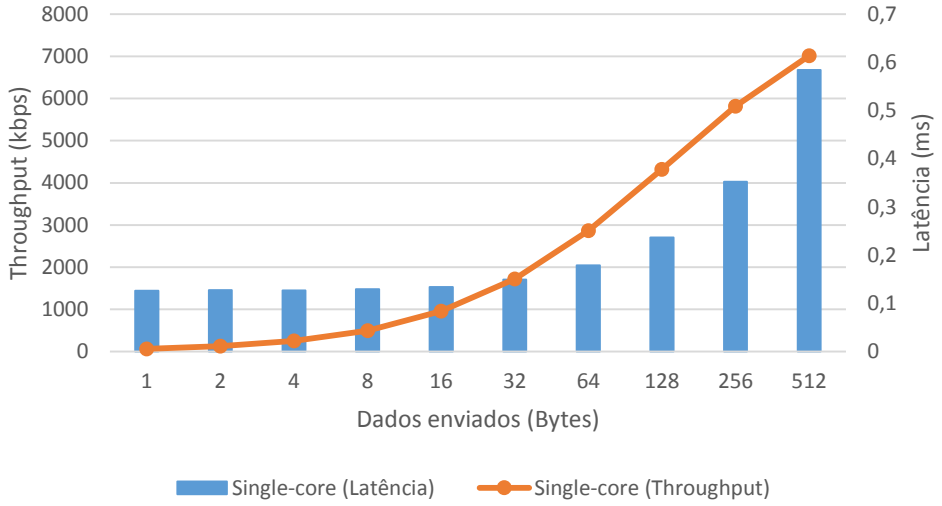


Figura 5.5: Comunicação Singlecore – Latência de envio de uma mensagem de diferentes tamanhos

A latência e taxa de transferência de uma mensagem de diversas dimensões encontram-se representadas na Figura 5.5. Através destas, podemos observar um crescimento quase exponencial de velocidade da comunicação mediante o aumento de dados enviados dentro da mesma mensagem. Este crescimento deve-se ao facto de que, para uma única mensagem, o *overhead* associado é independente do tamanho, à exceção da cópia de dados no início do envio da mensagem. Na verdade, o aumento da latência deve-se única e exclusivamente à ação do *memcpy*.

Os valores de latência variam entre 126 *ns* e 584 *ns*, para 1 e 512 bytes enviados respetivamente. Estes valores traduzem-se numa taxa de transferência de 63 kbps (kbps) e 7Mbps.

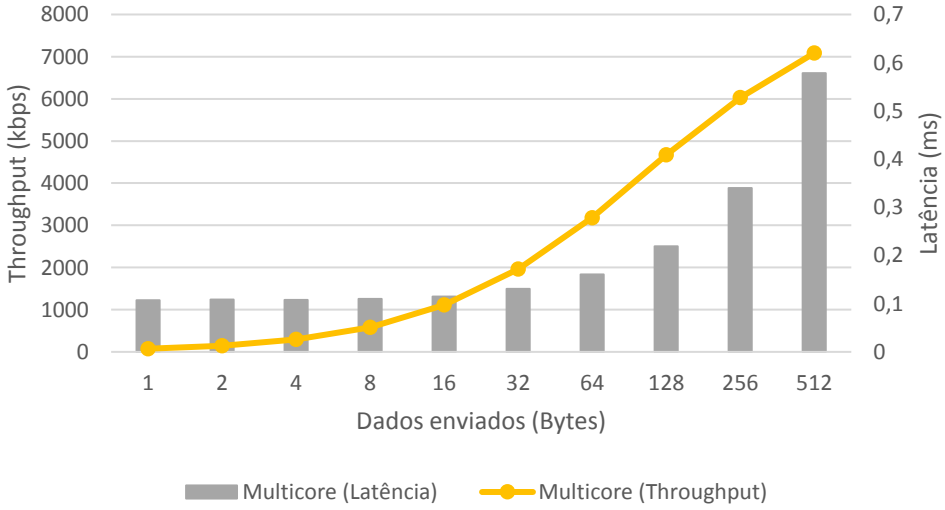


Figura 5.6: Comunicação Multicore – Latência de envio de uma mensagem de diferentes tamanhos

O gráfico da Figura 5.6 representa as mesmas propriedades medidas no teste anterior, mas para a comunicação baseada numa arquitetura assimétrica *multicore*. Os valores para a latência e taxa de transferência são similares formando um padrão semelhante. Estes valores são, no entanto, ligeiramente superiores situando-se entre os 74,79 kbps e os 7,09 Mbps.

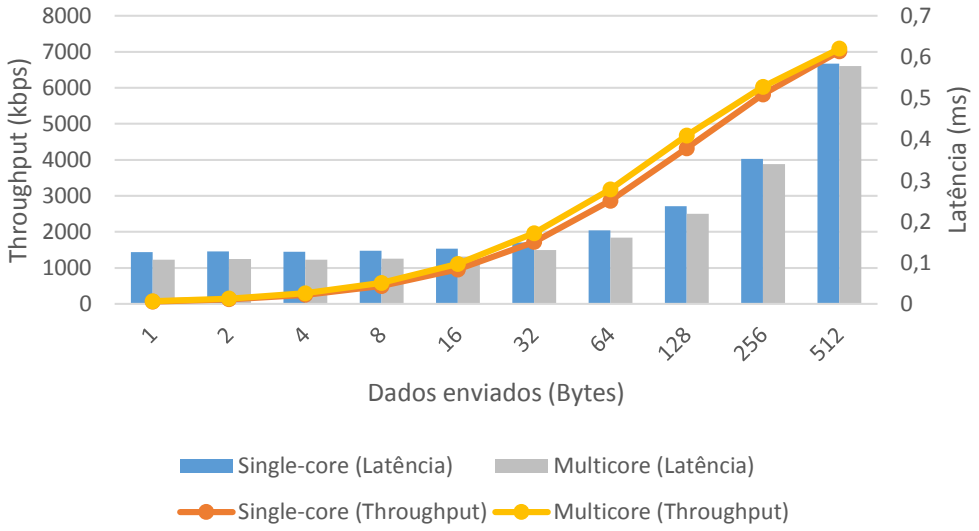


Figura 5.7: Comunicação – Latência de envio de uma mensagem de diferentes tamanhos

O aumento da velocidade pode ser observado na Figura 5.7. Contra intuitivamente ao observado, o aumento percentual mantém-se aproximadamente constante nos primeiros valores (até aos 16 Bytes exisste um aumento de 17% de taxa de transferência na arquitetura *multicore*), diminuindo progressivamente até ao valor final de 512 Bytes, onde o aumento é inferior a 1% (valores observados na Tabela 5.5). A diferença sublime entre os valores obtidos nas distintas configurações da *framework* LTZVisor explicam-se pela virtualização das interrupções e pela própria arquitetura *multicore*.

Tabela 5.5: Diferença entre taxa de transferência da comunicação *single-core* e *multicore*

Bytes enviados	1	2	4	8	16	32	64	128	256	512
Throughput (Mbps) single-core	0,06	0,13	0,25	0,50	0,95	1,72	2,86	4,32	5,82	7,02
Throughput (Mbps) multicore	0,07	0,15	0,30	0,58	1,11	1,96	3,18	4,67	6,03	7,09
Aumento Percentual	**17,87**	**17,13**	**18,01**	**17,11**	**16,49**	**14,10**	**11,19**	**8,11**	**3,65**	**1,00**

Quando a comunicação é realizada sobre a arquitetura *multicore*, as interrupções são roteadas diretamente para a partição oposta. Na versão *single-core*, as interrupções são armazenadas e necessitam de esperar pela comutação de mundos para serem desencadeadas. Esta espera e respetiva comutação (onde é incluída o *trigger* da interrupção) desempenham a fonte de *overhead* extra não existente na configuração *multicore*. À medida que a latência aumenta, este *overhead* vai-se diluindo no valor total, sendo quase impercetível para o envio de uma mensagem com dimensão máxima.

5.2.3.2 Variação do Número de Mensagens Enviadas

Na realização deste teste, foi medido o tempo de envio de diferentes números de mensagens enviadas. Este parâmetro foi variado entre o número mínimo de mensagens enviadas, 1, e o número máximo de buffers disponíveis. Na implementação atual esse valor fixa-se nas 256 mensagens. Cada mensagem enviada utiliza o tamanho máximo de buffer disponível, 512 bytes (não contabilizando o valor do *header*).

O fluxo dos processos pelos quais o Linux passa desde o atendimento à interrupção até ao processamento propriamente dito da mensagem recebida é um fluxo com uma certa variação temporal, devido à falta de determinismo associada a GPOS's. A rotina de serviço à interrupção seria demasiado extensa e não aconselhável pelas normas do Linux, sendo as suas funcionalidades migradas por uma *workqueue* iniciada pela própria rotina. Esta *workqueue*, pela sua menor prioridade comparativamente a um serviço a uma interrupção, pode ser escalonada pelo GPOS num momento posterior e pode até ser interrompida. Esta falta de determinismo tem como consequência uma certa variação nos valores medidos nesta caracterização. Por esta razão, os testes foram repetidos cerca de 1000 vezes para cada parâmetro variado.

O gráfico da Figura 5.8 representa os valores obtidos para a latência e taxa de transferência consoante a totalidade de dados enviados. Para descriminar o número de mensagens enviadas deve-se dividir o total de dados pelo valor de 512 bytes por mensagem. O gráfico apresenta um padrão logarítmico inicial, estagnando em valores próximos de 8 Mbps.

Devido ao design especificado e, de certa forma, imposto pela arquitetura *single-core*, cada grupo de mensagens enviado é processado através de uma única notificação. Esta possibilidade prevista na comunicação RPMsg é benéfica para a comunicação baseada na arquitetura *single-core*, uma vez que não necessita de realizar inúmeros *context-switches* para desencadear as notificações referentes a

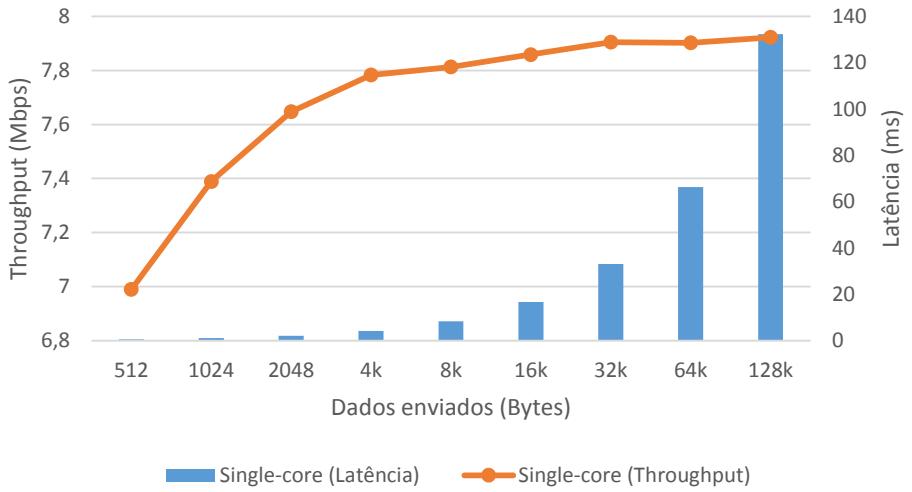


Figura 5.8: Comunicação Singlecore – Latência de envio de várias mensagens de 512 bytes cada (1 a 256 mensagens – 512 a 128k Bytes)

mensagens previamente inseridas no *vring*. Esta característica permite ao Linux o processamento sucessivo das mensagens recebidas de uma só vez, concluindo num aumento ligeiro de performance.

O gráfico presente na Figura 5.9 permite observar os valores de latência e *throughput* obtidos para a comunicação *multicore*, representando um padrão idêntico ao da comunicação baseada na arquitetura *single-core*. No entanto, os valores e rácio de crescimento face ao número de mensagens enviadas é bastante superior. A taxa de transferência passa de uma velocidade de 7,1 Mbps para estabilizar em valores em volta dos 14,5 Mbps, tendo como valor máximo nas 256 mensagens (128kBytes) 14,72 Mbps. Este valor representa um acréscimo de 85% face ao valor máximo obtido na comunicação inter-partição *single-core*.

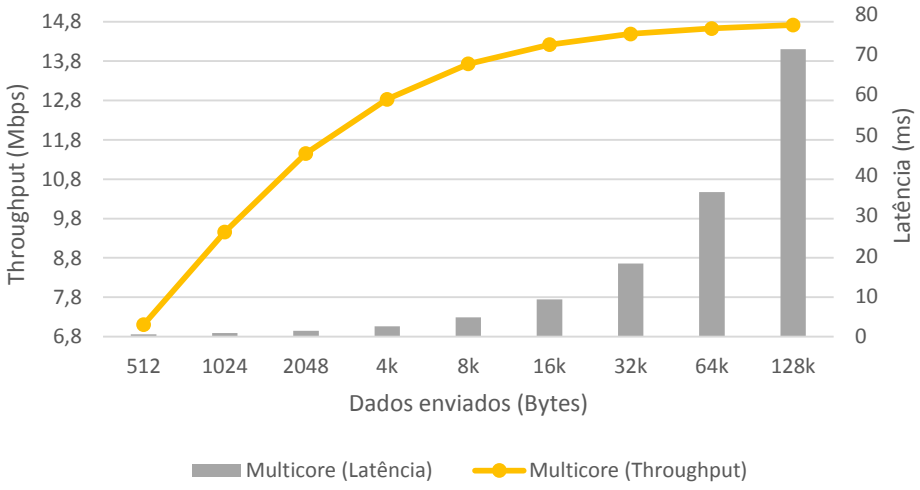


Figura 5.9: Comunicação Multicore – Latência de envio de várias mensagens de 512 bytes cada (1 a 256 mensagens – 512 a 128k Bytes)

A particularidade de uma notificação por diversas mensagens apenas é benéfica na configuração *single-core*. Apesar do *overhead* acrescido do envio de múltiplas notificações, na configuração *multicore*, devido à paralelização do processamento, o envio de uma notificação por mensagem é claramente superior. Enquanto que o FreeRTOS realiza a cópia dos dados a enviar para o buffer, o Linux pode processar mensagens anteriormente enviadas e devidamente notificadas, estando ambos os *guests* em processamento da comunicação (em fases distintas) ao mesmo tempo.

Para reiterar esta suposição, foram realizados alguns testes na comunicação *multicore*, nos quais foram retiradas as notificações inerentes ao envio de uma mensagem, tendo a mesma de ser explicitamente realizada no final do envio das mensagens. Esta alteração torna a comunicação *multicore* com um comportamento sequencial (não paralelizado), idêntico ao observado na arquitetura *single-core*.

Tabela 5.6: Taxa de Transferência para diferentes métodos aplicados na comunicação *multicore* comparativamente com comunicação *single-core*

Bytes enviados		**512**	**1024**	**2048**	**4096**	**8192**	**32k**
Throughput (Mbps) single-core	Not/ X Msg	6,99	7,39	7,65	7,78	7,81	7,90
Throughput (Mbps) multicore	Not/ X Msg	7,10	7,47	7,68	7,82	7,86	7,97
	Not/ 1 Msg	7,10	9,46	11,45	12,83	13,73	14,49

Os valores obtidos, apresentados na Tabela 5.6 para os testes supramencionados, revelam que o método de uma notificação por grupo de mensagens é substancialmente mais lento do que o teste de uma notificação por mensagem. Apesar de este ser o método mais eficiente em comunicação *single-core*, o mesmo não se aplica na sua versão *multicore*. Através do desempenho observado (pelos valores da taxa de transferência) verifica-se que a diferença de valores entre a comunicação baseada na arquitetura *single-core* e a baseada na arquitetura *multicore* com o mesmo método de notificação, centra-se apenas no tempo extra induzido pela comutação de mundos.

5.2.3.3 Conclusão

A comunicação RPMsgSuper, baseada na comunicação RPMsg, trata-se de uma comunicação nativamente *multicore*. Pelos seus mecanismos internos e capacidade de processamento paralelizado, o seu verdadeiro potencial revela-se quando baseada na arquitetura *multicore*, como observado na Figura 5.10.

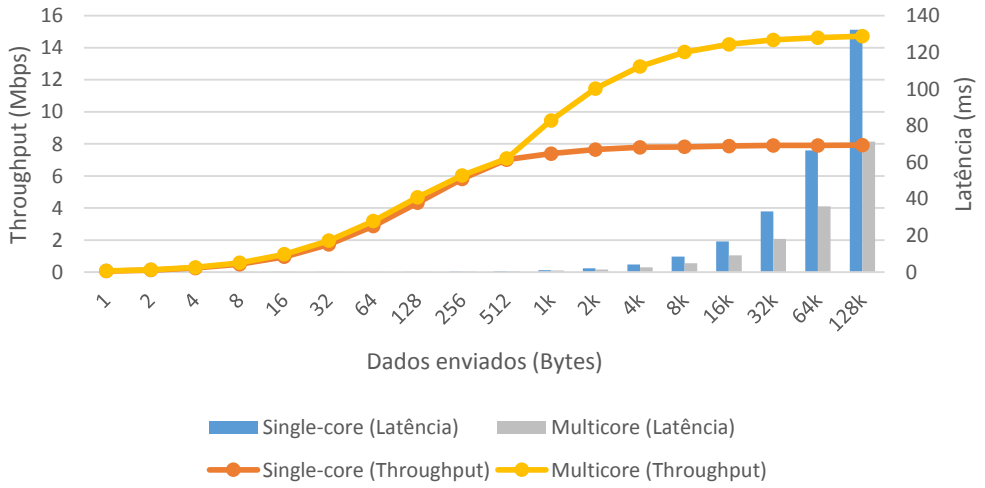


Figura 5.10: Comunicação – Latência de envio de mensagens entre 1 Byte até 128 kBytes

Ambas as configurações da comunicação têm um desempenho idêntico no processamento de uma mensagem com uma quantidade de dados variável, notando-se uma convexão de valores no ponto 512 Bytes, valor máximo suportado pelos mecanismos de comunicação atualmente implementados. Para uma única mensagem o desempenho da comunicação *multicore* é superior à vista na comunicação *single-core*, não acima dos 17%, fixando-se nos 1% para uma mensagem de dimensão máxima.

No entanto, após o aumento do número de mensagens trocado entre as partições, denota-se o favorecimento da comunicação *multicore*. Esta faz uso da sua capacidade de processamento paralelo para impor uma velocidade de transferência mais alta, quando comparada com a arquitetura *single-core*, chegando a valores acima dos 75% a partir do envio de 16 mensagens de forma sucessiva.

Conclusões

Neste último capítulo são apresentadas as conclusões extraídas do trabalho realizado, através das limitações e vantagens encontradas durante a fase de implementação, como através dos resultados obtidos e expostos no Capítulo 5.

6.1 Discussão

Em relação à implementação das mudanças impostas ao LTZVisor, o autor acredita que esta dissertação se tratou de um trabalho extenso e com uma área de abrangência dentro da área de sistemas embebidos enorme, englobando uma grande diversidade de temáticas: desde a compreensão da arquitetura de processadores com a arquitetura ARMv7 até à assimilação de conceitos inovadores na área de segurança e virtualização, nomeadamente a tecnologia hardware ARM TrustZone. Foram ainda aprofundados os conceitos nas áreas de sistemas operativos de tempo-real, bem como no sistema operativo Linux. A área de comunicação inter-partição e *multicore* endereçando de uma forma preferencial a performance (sem descuidar a segurança) foi uma revelação das potencialidades dos próprios sistemas embebidos adquiridas pelo autor.

O conjunto de resultados apresentados na secção 5.1 e 5.2 comprovam a necessidade de suporte da configuração *multicore* por parte da *framework* LTZVisor. Esta, à parte de uma possível ligeira redução de desempenho por parte do FreeRTOS, produziu resultados altamente vantajosos globalmente, tornando assim o sistema LTZVisor uma solução versátil e compatível com diversos problemas associados a sistemas embebidos.

No geral, o mecanismo de comunicação implementado satisfaz as necessidades do sistema LTZVisor, providenciando um modo claro e transparente de envio de dados inter-partição. Os mesmos, por respeitarem ambas as primitivas de segurança e de tempo-real inerentes ao sistema sofrem um pouco com qualquer alteração realizada ao sistema ou às próprias partições. Estes mecanismos, pela forma como foram adaptados ao sistema virtualizado (*single-core*), prejudicam o desempenho da comutação de mundos, sendo uma limitação conhecida dos mesmos. Estes mecanismos permitem uma comunicação transversal às configurações

implementadas no sistema, sendo que a configuração *multicore* tem uma vantagem no seu desempenho pela sua utilização própria dos recursos da plataforma.

6.2 Trabalho Futuro

Os objetivos propostos na presente dissertação (Subsecção 1.2) foram concluídos com enorme sucesso. No entanto existem algumas limitações que necessitam de ser ultrapassadas, bem como funcionalidades que poderiam ser incluídas na *framework* LTZVisor de forma a expandir ainda mais a versatilidade do sistema virtualizado como solução aos problemas inerentes em sistemas embebidos.

Primeiramente, seria necessário aprofundar o conhecimento sobre as limitações encontradas pelos resultados obtidos do FreeRTOS habitando o core 1. Apesar de a perda de performance poder ser considerada parca, esta existe, e, tratando-se da partição de maior prioridade na qual os níveis de performance são considerados essenciais, justificar-se-ia um estudo aprofundado sobre as alterações. À parte do estudo proposto, poder-se-ia realizar ainda o *"porting"* do GPOS Linux para o core 1 possibilitando uma maior flexibilidade da *framework* na disposição dos *guests* habitantes na sua arquitetura assimétrica *multicore*.

Uma outra sugestão partiria pela adaptação do mecanismo de comunicação implementado a sistemas virtualizados *multiguest*. Dentro do mesmo âmbito de virtualização assistida pela tecnologia ARM TrustZone surgiram *in-house*, pelo mesmo autor da *framework* em expansão, diversas outras *frameworks* de virtualização com propósitos diferentes seguindo, no entanto, uma arquitetura semelhante. Estas *frameworks* têm em comum a utilização de mais de um par de *guests*. A flexibilidade e versatilidade projetada na altura de design teve como desígnio uma possível escalação de utilização dos mecanismos de comunicação o que permitiria uma rápida adaptação dos mesmos às *frameworks multiguest* RodosVisor [7] e RTZVisor [8].

Finalmente, ainda no âmbito da comunicação inter-partição, seria interessante a adição de funcionalidades extras ao mecanismo de comunicação implementado. O próprio grupo de desenvolvimento da especificação e implementação OpenAMP tem-se debruçado sobre estas funcionalidades. Entre as discutidas, destacam-se o suporte para comunicação *zero-copy* e envio de mensagens *sampling*. Ambas as funcionalidades são essenciais para sistemas operativos de Tempo Real e até processadores de recursos escassos comparativamente aos Cortex-A. Estas funcionalidades preveem fornecer um nível de versatilidade extra ao LTZVisor, facilitando

um possível *porting* para uma plataforma Cortex-M e invocando a prioridade inerente de tempo-real do mesmo.

Referências Bibliográficas

[1] S. H. VanderLeest, "Arinc 653 hypervisor," in *29th Digital Avionics Systems Conference*, pp. 5.E.2–1–5.E.2–20, Oct 2010.

[2] F. Baum and A. Raghuraman, "Making Full use of Emerging ARM-based Heterogeneous Multicore SoCs." XCell95, October 2015.

[3] Xilinx, "Zynq-7000 All Programmable SoC Technical Reference Manual," vol. 190, pp. 1–1400, 2014.

[4] G. Heiser, "Virtualizing embedded systems: Why bother?," in *Proceedings of the 48th Design Automation Conference*, DAC '11, (New York, NY, USA), pp. 901–905, ACM, 2011.

[5] S. Pinto, D. Oliveira, J. Pereira, N. Cardoso, M. Ekpanyapong, J. Cabral, and A. Tavares, "Towards a lightweight embedded virtualization architecture exploiting arm trustzone," in *Proceedings of the 2014 IEEE Emerging Technology and Factory Automation (ETFA)*, pp. 1–4, Sept 2014.

[6] S. Pinto, D. Oliveira, J. Pereira, J. Cabral, and A. Tavares, "Freetee: When real-time and security meet," in *2015 IEEE 20th Conference on Emerging Technologies Factory Automation (ETFA)*, pp. 1–4, Sept 2015.

[7] S. Pinto, J. Pereira, T. Gomes, M. Ekpanyapong, and A. Tavares, "Towards a trustzone-assisted hypervisor for real time embedded systems," *IEEE Computer Architecture Letters*, vol. PP, no. 99, pp. 1–1, 2016.

[8] S. Pinto, A. Tavares, and S. Montenegro, "Space and time partitioning with hardware support for space applications," *Data Systems In Aerospace, European Space Agency (Special Publication)*, 2016.

[9] S.-C. Oh, K. Koh, C.-Y. Kim, K. Kim, and S. Kim, "Acceleration of dual os virtualization in embedded systems," in *2012 7th International Conference on Computing and Convergence Technology (ICCCT)*, pp. 1098–1101, Dec 2012.

[10] J. Serra, J. Rodrigues, T. R. O. Almeida, and A. P. Coimbra, "Multi-criticality Hypervisor for Automotive Domain," *Atas do INForum 2014*, no. January 2015, pp. 446–456, 2014.

[11] T. Frenzel, A. Lackorzynski, A. Warg, and H. Härtig, "Arm trustzone as a virtualization technique in embedded systems," in *Proceedings of Twelfth Real-Time Linux Workshop, Nairobi, Kenya*, 2010.

[12] M. Cereia and I. C. Bertolotti, "Asymmetric virtualisation for real-time systems," in *2008 IEEE International Symposium on Industrial Electronics*, pp. 1680–1685, June 2008.

[13] D. Sangorrín, S. Honda, and H. Takada, "Dual Operating System Architecture for Real-Time Embedded Systems," *Proc. 6th International Workshop on Operating Systems Platforms for Embedded Real-Time Applications*, pp. 6–15, 2010.

[14] B. Ngabonziza, D. Martin, A. Bailey, H. Cho, and S. Martin, "Trustzone explained: Architectural features and use cases," in *2016 IEEE 2nd International Conference on Collaboration and Internet Computing (CIC)*, pp. 445–451, Nov 2016.

[15] A. S. Berger, *Embedded Systems Design: An Introduction to Processes, Tools, and Techniques*, vol. 2002. 2002.

[16] S. H. VanderLeest and D. White, "Mpsoc hypervisor: The safe amp; secure future of avionics," in *2015 IEEE/AIAA 34th Digital Avionics Systems Conference (DASC)*, pp. 6B5–1–6B5–14, Sept 2015.

[17] A. Crespo, M. Masmano, J. Coronel, S. Peiró, P. Balbastre, and J. Simó, "Multicore partitioned systems based on hypervisor," *IFAC Proceedings Volumes*, vol. 47, no. 3, pp. 12293 – 12298, 2014.

[18] P. Barham, B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, R. Neugebauer, I. Pratt, and A. Warfield, "Xen and the art of virtualization," *SIGOPS Oper. Syst. Rev.*, vol. 37, pp. 164–177, Oct. 2003.

[19] ARM, "ARM Security Technology, Building a Secure System using TrustZone Technoloy," tech. rep., ARM, 2009.

[20] S. W. Kim, C. Lee, M. Jeon, H. Y. Kwon, H. W. Lee, and C. Yoo, "Secure Device Access for Automotive Software," *ICCVE International Conference on Connected Vehicles and Expo*, pp. 177–181, 2013.

[21] R. Russell, "Virtio: Towards a de-facto standard for virtual i/o devices," *SIGOPS Oper. Syst. Rev.*, vol. 42, pp. 95–103, July 2008.

[22] R. Russell and I. OzLabs, "lguest: Implementing the little Linux hypervisor," *Proceedings of the Linux Symposium*, vol. 2, pp. 173–178, 2007.

[23] C. Dall and J. Nieh, "Kvm/arm: The design and implementation of the linux arm hypervisor," in *Proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '14, (New York, NY, USA), pp. 333–348, ACM, 2014.

[24] S. Patni, J. George, P. Lahoti, and J. Abraham, "A zero-copy fast channel for inter-guest and guest-host communication using virtio-serial," in *2015 1st International Conference on Next Generation Computing Technologies (NGCT)*, pp. 6–9, Sept 2015.

[25] D. Sangorrín, S. Honda, and H. Takada, "Reliable and efficient dual-os communications for real-time embedded virtualization," *Information and Media Technologies*, vol. 8, no. 1, pp. 1–17, 2013.

[26] F. Diakhaté, M. Perache, R. Namyst, and H. Jourdren, *Efficient Shared Memory Message Passing for Inter-VM Communications*, pp. 53–62. Berlin, Heidelberg: Springer Berlin Heidelberg, 2009.

[27] A. Gordon, M. Ben-Yehuda, D. Filimonov, and M. Dahan, "Vamos: Virtualization aware middleware," in *Proceedings of the 3rd Conference on I/O Virtualization*, WIOV'11, (Berkeley, CA, USA), pp. 3–8, USENIX Association, 2011.

[28] A. Landau, M. Ben-Yehuda, and A. Gordon, "Splitx: Split guest/hypervisor execution on multi-core," in *Proceedings of the 3rd Conference on I/O Virtualization*, WIOV'11, (Berkeley, CA, USA), pp. 1–7, USENIX Association, 2011.

[29] Xilinx, "TrustZone Architecture on the Xilinx Zynq-7000 All Programmable SoC," vol. 1019, 2014.

[30] M. Jones, "Virtio: An I/O virtualization framework for Linux," tech. rep., IBM, January 2010.

[31] C. Staelin, "lmbench3: Measuring scalability," tech. rep., HP Laboratories Israel, November 2002.