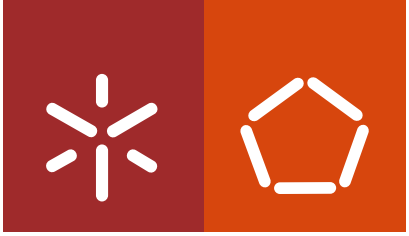
Universidade do Minho
Escola de Engenharia

Adriano Dídimo Machado Carvalho

Embedded-Systems-Oriented Virtualization Framework with Functionality Farming

setembro de 2016

Embedded-Systems-Oriented Virtualization Framework with Functionality Farming
Adriano Dídimo Machado Carvalho
UMinho | 2016



Universidade do Minho
Escola de Engenharia

Adriano Dídimo Machado Carvalho

Embedded-Systems-Oriented Virtualization Framework with Functionality Farming

Tese de Doutoramento em Engenharia Electrónica
e de Computadores

Trabalho realizado sob a orientação do
Professor Doutor Adriano José Conceição Tavares
e do
Professor Doutor Francisco Carlos Afonso

setembro de 2016

STATEMENT OF INTEGRITY

I hereby declare having conducted my thesis with integrity. I confirm that I have not used plagiarism or any form of falsification of results in the process of the thesis elaboration.

I further declare that I have fully acknowledged the Code of Ethical Conduct of the University of Minho.

University of Minho, September 20th, 2016

Full name: *Adriano Dídimo Machado Carvalho*

Signature: Adriano C. Machado

Acknowledgments

I would like to express my sincere gratitude to my adviser Prof. Adriano Tavares for the continuous support of my Ph.D. study and without which this work would not be possible.

I also would like to thank Prof. Francisco Afonso for his comments and encouragement throughout this work.

I would like to thank my Ph.D. proposal defense committee, Prof. Francisco Afonso, Prof. Jorge Cabral, Prof. João Cardoso and Prof. Júlio Martins, for their insightful comments and suggestions.

I would also like to thank everyone, family, colleagues and friends, who have accompanied me during this work, especially during my darkest hours.

My sincere thanks also goes to all of the developers of free and open source software for their dedication and, without which, this work (and many others) would not be possible.

This work has been supported by COMPETE: POCI-01-0145-FEDER-007043 and Fundação para a Ciência e Tecnologia within the Project Scope: UID/CEC/00319/2013. This work was also supported by Fundação para a Ciência e Tecnologia, grant SFRH/BD/81640/2011.

Resumo

Um: O uso de um hipervisor como *kernel* de separação em arquiteturas integradas está a ser considerado, visto que, um hipervisor não só proporciona separação temporal e espacial, mas também compatibilidade com software *legacy*. No entanto, nos dias de hoje, a maior parte dos hipervisores baseiam-se em paravirtualização ou dependem de hardware *high-end*; ambas as abordagens não cumprem os requisitos dos sistema embebidos críticos para a segurança. A paravirtualização, por um lado, não proporciona compatibilidade total com software *legacy*, sendo necessária a sua modificação e adaptação a uma interface especifica do hipervisor utilizado. Hardware *high-end*, por outro lado, apesar de proporcionar compatibilidade total com software *legacy*, dá origem a sistemas de grande dimensão, de elevado peso, com elevado consumo de energia, de elevado custo, etc. Nesta tese, a capacidade da virtualização completa em hardware *low-end* para resolver as limitações dos hipervisores existentes é investigada. Para isso, um hipervisor baseado em virtualização completa em hardware *low-end* é descrito e é apresentada uma avaliação da sua performance e do espaço ocupado em memória.

Dois: Métodos de desenvolvimentos convencionais não são capazes de acompanhar os requisitos dos sistemas embebidos críticos para segurança de hoje em dia. Nesta tese: (a) é apresentada uma abordagem baseada em modelos já existente, mais especificamente, geração de código baseada em modelos; (b) são descritas as modificações aplicadas a um compilador de modelos já existente por forma a que este suporte novas capacidades; e (c) é apresentada uma avaliação sobre a capacidade da geração de código baseada em modelos de reduzir o esforço de engenharia quando comparada com abordagens convencionais.

Três: A maior parte dos sistemas operativos de hoje em dia seguem uma arquitetura monolítica; esta arquitetura, no entanto, está associada a fraca confiabilidade, baixa segurança, esforço de certificação elevado, bem como baixa previsibilidade e escalabilidade. Para colmatar estes problemas, as soluções propostas na literatura apenas contornam a origem do problema, i.e., a elevada dimensão do *kernel* numa arquitetura monolítica, e não o resolvem diretamente. Nesta tese, *functionality farming* é proposto para atacar a origem do problema. *Functionality farming* apenas, no entanto, depende de um esforço de engenharia significativo. Visto isto, esta tese também apresenta FF-AUTO, uma ferramenta capaz de realizar *functionality farming* de forma semi-automática. Por último, esta tese demonstra como *functionality farming* é capaz de melhorar o design e a performance de um *kernel* já existente, e demonstra também como FF-AUTO permite uma redução significativa do esforço de engenharia.

Abstract

First, the use of a hypervisor as the separation kernel on integrated architectures has been considered, as it not only provides time and space partitioning, but it also provides compatibility with legacy software. Nowadays, most hypervisors, however, either rely on paravirtualization or depend on high-end hardware, both of which do not fulfill the requirements of safety-critical embedded systems. Paravirtualization does not provide complete legacy compatibility as it requires legacy software to be modified to fit a hypervisor-specific interface. High-end hardware, on the other hand, even though it provides complete legacy compatibility, it leads to large system size, weight, power consumption, cost, etc. In this thesis, the feasibility of low-end hardware full virtualization to address the limitations of existing hypervisors is investigated. For that, a hypervisor based on low-end hardware full virtualization is described and an evaluation of its performance and footprint is presented.

Second, conventional development methods are unable to keep up with the requirements of nowadays and future safety-critical embedded systems. In this thesis: (a) an existing model-driven engineering approach to address the limitations of conventional development methods is presented; more specifically, a model-driven code generation approach; (b) the modifications applied to an existing model compiler in order for it to support new features are described; and (c) an evaluation of whether or not a model-driven code generation approach leads to lower engineering effort when compared to a conventional approach is presented.

Third, most operating systems, nowadays, follow a monolithic architecture; this, however, leads to poor reliability, weak security, high certification effort, as well as poor predictability and scalability. To address this problem, the solutions proposed in the literature just work around the source of the problem, i.e., the large size of the kernel in a monolithic architecture, and do not address it directly. In this thesis, functionality farming is proposed to tackle the source of the problem. Functionality farming alone, however, depends on a significant engineering effort. To address this problem, this thesis also presents FF-AUTO, a tool which performs functionality farming semi-automatically. At last, this thesis demonstrates how functionality farming is able to improve the design and the performance of an existing kernel, as well as how FF-AUTO enables a significant reduction of the required engineering effort.

Table of Contents

Figures

Tables

Listings

1. Introduction

1.1. Full Virtualization on Low-End Hardware

A system is safety-critical if its failure can cause harm to people or the environment [1]. Such a failure is also known as a catastrophic failure. In a safety-critical system a missed deadline can result in a catastrophic failure, and thus, a safety-critical system is also a hard real-time system. Examples of safety-critical systems include: commercial and military aircraft, automobiles, traffic lights at an intersection, nuclear power plants, medical devices and implants, among many others. Within the domain of safety-critical systems, this thesis focuses on the sub-domain of safety-critical embedded systems. Safety-critical embedded systems can be distinguished from other computer systems by especially tight constrains along many axes, including:

- Size, volume, and weight: embedded systems, as the name suggests, are embedded in larger electrical, mechanical or hydraulic systems, and thus, are constrained to the physical characteristics of a larger system in which they are embedded.
- Power consumption: in many cases, a connection to a continuous power supply is not possible and embedded systems must rely on batteries; therefore, it becomes critical to reduce power consumption in order to extend battery-life, reduce maintenance, and thus, increase availability.
- Cost: embedded systems are often mass produced and reducing their manufacturing cost is critical for profit. In terms of the software, certification, when required, is the most costly development activity and it is critical to reduce its impact. In terms of hardware, memory is usually the most expensive resource, and thus, reducing the software's memory footprint is also important.

In order to assure that a safety-critical system is unlikely to fail (it is not feasible to assure that a system will never fail), it must be certified by a third party before being deployed for its intended application. IEC 61508 [2], for example, is a standard "applicable to all electrical/electronic/programmable electronic safety-related systems irrespective of the application." Similar standards include: DO-178 [3], specific to airborne systems, ISO 26262 [4], for road vehicles, and ISO 62304 [5], for medical devices. In IEC 61508, systems are classified according to a Safety Integrity Level (SIL) depending on the consequences of failure (i.e., “multiple loss of life” down to “minor injuries at worst”) and their likelihood (i.e., “frequent” down to “improbable” and “incredible”); the higher the SIL, the more safety-critical the system is. Certification may require

specific development methods to be followed, such as extensive testing of the hardware and the software; the higher the SIL, the more rigorous the certification process. In IEC 61508, systems classified as SIL 1 (the lowest) can be developed according to quality management standards, such as ISO 9001 [6]. A system classified as SIL 4 (the highest), on the other end, needs to be verified and validated using formal methods and redundancy is mandatory. It is well known that certification of safety-critical systems is a difficult, time consuming and expensive activity, and that it can take up to seven times longer than other development activities [7].

Traditional large-scale safety-critical systems follow a federated architecture [8]–[10], illustrated in Figure 1.1. In a federated architecture, a system is composed of multiple computing units ("CU" in Figure 1.1) connected to one or more buses; each computing unit is assigned with a single, independently developed function. Although providing strong fault containment and isolation between functions through physical separation, a federated architecture, however, has the following disadvantages [8]–[10]:

- a long bill of materials, which leads to large size and volume, high weight, high power consumption, and high cost;
- lack of common line replaceable units, and thus, high maintenance, repair and overhaul costs, as well as obsolescence problems;
- high hardware development and hardware certification effort, and thus, long time to market and high cost;
- high hardware fault rate, low reliability, low availability, and therefore, high maintenance cost;
- few organizations are able to support the life-time costs of the system, leading to a small market and low profit.

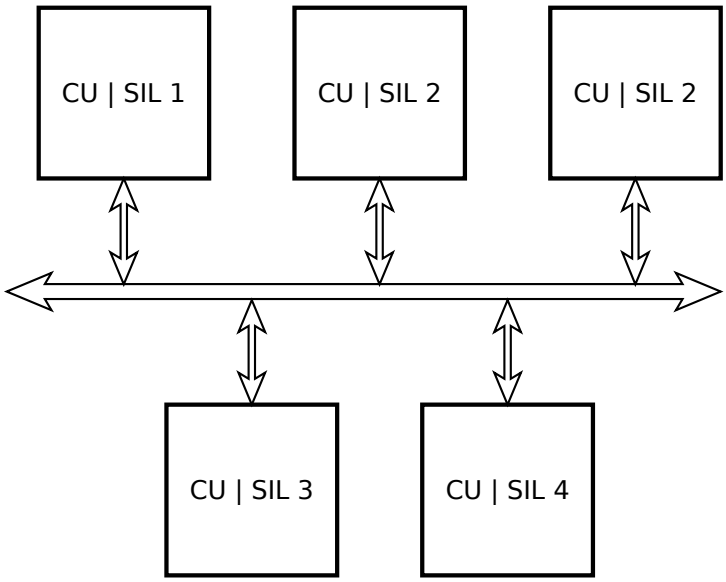


Figure 1.1. Illustration of a federated architecture.

A federated architecture may be well suited for small-scale safety-critical embedded systems with a small number of functions; however, a federated architecture does not scale well for large-scale systems, and it is not adequate for highly-constrained safety-critical embedded systems, especially when the demand for more functions is already high and increasing (e.g., autonomous vehicles).

To address the issues with the federated architecture, integrated architectures have been developed. Examples include: the integrated modular avionics architecture from the avionics industry [11], AUTomotive Open System ARchitecture (AUTOSAR) from the automotive industry [12], and the integrated time-triggered architecture (DECOS) from academia [13]. Similarly to a federated architecture, in an integrated architecture the system is composed by multiple computing units connected to one or more buses. However, in an integrated architecture, each computing unit may be assigned with multiple, independently developed functions with different SIL, as illustrated in Figure 1.2. In this way, an integrated architecture has the following advantages over a federated architecture [8]–[10]:

- shorter bill of materials, and thus, smaller size and volume, lower weight, lower power consumption, and lower cost;
- lower hardware development and hardware certification effort, and therefore, lower time to market and cost;

- lower hardware fault rate, higher reliability and availability, and therefore, lower maintenance costs;
- lower life-cycle costs, and thus, larger market size and higher profit.

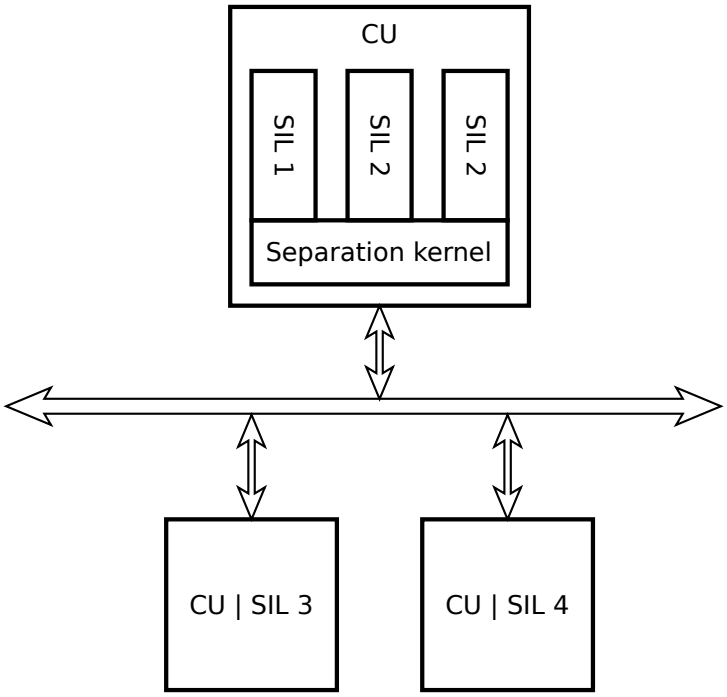


Figure 1.2. Illustration of an integrated architecture.

As illustrated in Figure 1.2, in an integrated architecture, computing units assigned with multiple functions rely on a separation kernel. The separation kernel is responsible for providing predictable execution environments, also called partitions, logically isolated in time (i.e., execution time) and space (i.e., memory and I/O devices) from each other. The separation kernel enforces isolation, through time and space partitioning, that is: by allocating a fixed memory and I/O space as well as a predefined execution time slot to each partition, and by allowing only predefined communication between partitions. The separation kernel guarantees fault containment and isolation by ensuring that partitions do not interfere with each other and, for example, that a failure in a partition assigned with a low-SIL function does not interfere with a partition assigned with a high-SIL function. The separation kernel, nonetheless, must be certified according to at least the highest SIL among the functions in the computing unit.

The separation kernel concept, by itself, however, does not define the actual execution environment available to functions (e.g., the available services, their interface), except that these must be time

and space partitioned. To address this shortcoming and improve the interoperability between different vendors and suppliers, a few standards, such as ARINC 653 [14] for the integrated modular avionics architecture, have been developed, defining the available services and their interface (i.e., the execution environment). Still, the execution environments defined by those standards do not provide compatibility with legacy software. However, significant investments have been made in legacy software and, in many cases, developing anew is not possible (e.g., licensing issues, closed source). To address this problem, using a hypervisor as the separation kernel is being considered [15]–[19].

A hypervisor can be regarded as a specialization of the separation kernel. As illustrated in Figure 1.3, while a separation kernel provides partitions, a hypervisor provides virtual machines. A hypervisor is not only able to do everything that a separation kernel does, such as time and space partitioning, but it also provides compatibility with legacy software, such as real-time and general-purpose operating systems and applications. A hypervisor, therefore, enables the software developed for a legacy federated architecture, for example, to be reused with none or few modification on a newer integrated architecture. Consequently, using a hypervisor as the separation kernel enables the transition from a federated to an integrated architecture to be smoother and less costly. Nowadays, most hypervisors, however, as will be demonstrated in *Section 2.2. Virtualization and Hypervisors*, either (1) do not provide complete legacy compatibility as they rely on paravirtualization, which requires legacy software to be modified to fit a hypervisor-specific, often proprietary, interface, or (2) do provide complete legacy compatibility, but depend on high-end hardware (i.e., hardware with virtualization extensions) which leads to large size and volume, high weight, high power consumption, and high cost. Full virtualization on low-end hardware (i.e., hardware without virtualization extensions), on the other end, has none of those disadvantages.

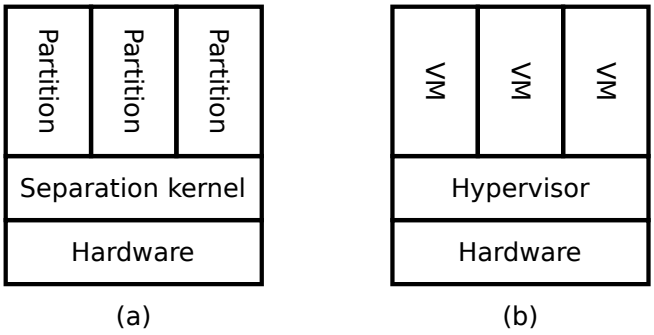


Figure 1.3. Illustration of (a) a separation kernel vs. (b) a hypervisor.

Low-end hardware full virtualization is able to provide the same features as high-end hardware full virtualization, but on low-end hardware. Full virtualization on low-end hardware must be accomplished using mechanisms which were not originally designed for it. Therefore, full virtualization on low-end hardware is not always possible because the hardware may not fulfill the necessary requirements [20]. Furthermore, it is often claimed that it is not feasible due to an unacceptably high virtualization overhead (e.g., [21]); however, we were unable to find real-world quantitative results to support those claims..

In this thesis, performance and footprint (i.e., the size of the trusted computing base and memory footprint) measurements from a case study on low-end hardware full virtualization are presented. More specifically, a case study to evaluate to what extent low-end hardware full virtualization is an alternative to high-end hardware full virtualization and paravirtualization, and provide compatibility with unmodified legacy software with acceptable performance and footprint. At the same time, one processor architecture is evaluated in its ability for the realization of low-end hardware full virtualization, so that the limitations, if any, of this and other similar processor architectures can be addressed in the future.

In this thesis, first, Rodosvisor, a hypervisor featuring full virtualization of the PowerPC 405 [21] (i.e., a representative of low-end hardware), is presented. Second, it is described how Rodosvisor has been integrated with POK [23], an ARINC 653 separation kernel (and real-time operating system), resulting in POK/rodosvisor. Third, an evaluation of the virtualization overhead in a Linux-based operating system is presented, by comparing several benchmarks performed on bare metal and as a guest on POK/rodosvisor. This is similar to what is presented in [24]; there, however, a hypervisor based on paravirtualization is used. Forth, a detailed look at the performance of POK/rodosvisor’s internal operation is presented (i.e., POK/rodosvisor’s performance profile), namely: interrupt handlers and context switching. Fifth and last, the size of POK/rodosvisor’s trusted computing base and memory footprint for various configurations are presented.

1.2. Model-Driven Engineering

In terms of requirements analysis, software design, verification and certification, conventional development methods employ informal methods, prone to misunderstanding, and where the opportunity for automatic generation of the implementation is lost [25], [26]. Furthermore, when it comes to the implementation, conventional development methods depend on a significant amount of manual and error-prone labor, leading to long time-to-market [25], [26]. Knowing that the complexity and the number of safety-critical embedded systems is increasing, it becomes necessary

to apply methods which reduce the accidental complexity, and thus, accelerate development while still maintaining or improving quality. Accidental complexity is the complexity associated with a design or implementation not directly related to the solution space. In the literature many methods have been proposed to improve upon conventional development methods, such as:

- Computer-aided software engineering (CASE) [27], [28]: it emerged when 3rd generation programming languages started to reveal limitations to cope with complexity, and focused on general-purpose graphical programming (e.g., state machines, structure diagrams, data flow diagrams) to express design intent; at the time, however, CASE failed to become widely adopted because the technology was not mature enough, and because programming languages and platforms evolved, alleviating the need for CASE.
- Component-based software engineering (CBSE) [29], [30]: it defends that systems should be designed by assembling software components together; CBSE, however, focuses mostly on reuse of components and not on improving the implementation of the components themselves.
- Software product-line engineering (SPLE) [31], [32]: it enables organizations to built an array of similar software products (applications) from a common (domain specific) and a variable (application specific) pool of resources, and thus, reduce development effort and cost, when compared with conventional single system development; the downside of SPLE is that variability management is still not mature enough, and it requires a significant upfront investment.
- Model-driven engineering (MDE) [28], [33]: it advocates that development should be driven by high-level models which raise the abstraction level (when compared with 3rd generation programming languages) and reduce the distance between the problem space and the solution space, thus facilitating development. MDE technologies are still not mature enough for wide adoption and a significant upfront investment is also required.

POK, mentioned earlier, through Ocarina [34]–[37], supports MDE. Ocarina is a compiler for the Analysis & Architecture Description Language (AADL) [38]. AADL enables the specification of the software and hardware architecture, and thus, the specification of the desired system configuration at a high level of abstraction, and features: processors, memories, processes, threads, subprograms, inter and intra-partition communication, etc. Ocarina is able to transform an AADL model into a POK configuration, composed by C source code and makefiles. Ocarina is also capable of generating partitions, if necessary. Ocarina, however, did not support the features that

have been added to POK, namely: virtual machines and privileged partitions.

In this thesis, it is described how Ocarina has been extended to support virtual machines and privileged partitions. In this way, it is demonstrated: (1) the ability of AADL to represent those features, (2) the ability of Ocarina to support those representations and to generate a POK configuration accordingly, and (3) their ability to replace conventional approaches. In this thesis, it is also demonstrated the ability of AADL-based MDE to reduce the engineering effort when compared to a conventional approach, by comparing the engineering effort required by an AADL-based with a manual development approach.

1.3. Functionality Farming

Nowadays, most operating systems follow a monolithic architecture [39]–[41]. In a monolithic architecture many of the services provided by the operating system (e.g., device drivers and protocol stacks) are deployed in the same address space as the kernel. For example, in the Linux kernel, version 2.6, 88% of the code is related to protocol and device drivers [42]. This, however, leads to a large kernel, which in turn, is associated with the following drawbacks:

- a large number of bugs and low reliability: a conservative estimate indicates that there are six bugs per 1,000 source lines of code, and device drivers have bug rates that are three to seven times higher than normal code [39], [43];
- a large attack surface and weak security: since the kernel is part of the trusted computing base of the entire system, if even a small function in the kernel is compromised, then, the entire system is at risk [42], [44];
- a high certification effort: a large kernel is also harder to certify than a small one [41], [45].

Furthermore, most of those services are not explicitly schedulable and “steal” other schedulable entities' execution time, leading to poor predictability and scalability as well.

To reduce the size of the trusted computing base (TCB), and thus, to improve security and reduce the certification effort, some authors propose the use of architectures based on: (1) a virtual machine monitor (or hypervisor) [42], [44], [46], or (2) a microkernel [41], or (3) a combination of the two (i.e., a microkernel with virtualization support) [47]. In these architecture, the size of the core/root kernel (and thus, of the TCB) is much smaller than the kernel found in most commodity operating systems. In these architectures, commodity operating systems are pushed onto a virtual machine (on a hypervisor-based architecture) or onto one or more user-level servers (on a microkernel-based architecture), reducing the effects that a compromised commodity operating system can have on the

system as a whole. Alongside commodity operating systems, critical services are deployed on other virtual machines (or user-level servers), which depend on a much smaller TCB than that in a commodity operating system. These architectures, however, depend on an additional level of indirection which leads to poor performance. Furthermore, on a hypervisor-based architecture in particular, virtual machines are often coarse-grained and heavyweight leading to high resource usage. On a microkernel architecture, on the other end, there is no compatibility with legacy software, and the porting effort can be very significant. An architecture based on a “microkernel with virtualization support” solves the above issues with the other two architectures, at the cost, however, of a larger size of the kernel. Still, all these architectures depend on the development of a new kernel, and thus, of a significant upfront investment; if the development of the new kernel fails, the cost is huge. In the end, these architectures do not tackle the source of the problem, i.e., the large size of the kernel in commodity operating systems, and just work around it.

In this thesis, functionality farming is proposed, which, instead of a new architecture, consists in partitioning existing kernels by (1) moving functionality out of the kernel and onto the application (or partition) level, to reduce the size of the kernel, and by (2) replacing the functionality being moved with remote procedure calls to the partition level, to bridge the gap between the kernel and the partition level. This is illustrated in Figure 1.4.

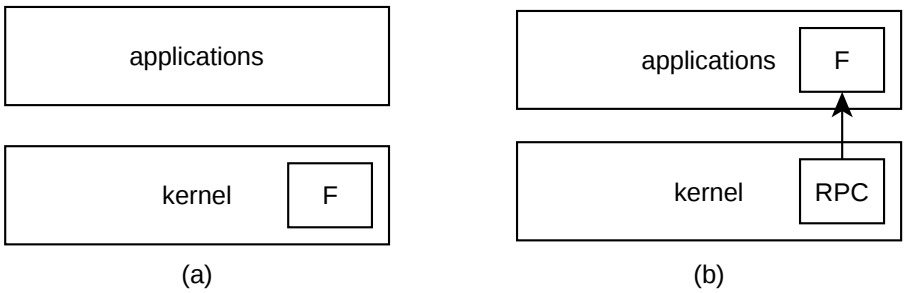


Figure 1.4. Illustration of functionality farming: (a) before functionality farming; (b) after functionality farming.

At the partition level, memory protection is enforced (space partitioning), and, previously non-schedulable entities, become explicitly schedulable (time partitioning). Through space partitioning, it is possible to reduce the size of the kernel, and thus, improve its reliability and security, as well as to reduce the certification effort. Additionally, in case of failure only the faulting partition, and not the entire kernel (and thus, not the entire system), needs to be restarted, leading to higher availability as well. On the other end, through time partitioning, it is possible to improve the

kernel's predictability and scalability by making parts of the kernel explicitly schedulable. At the partition level, moreover, it is easier to distribute the operating system's services across the cores of a multicore processor architecture, leading to improved predictability and scalability on such platforms.

Unlike other works, functionality farming tackles the source of the problem (i.e., the size of the kernel in commodity operating systems). It requires a lower upfront investment, since it enables a progressive reduction of the size of the kernel, instead of an all-or-nothing approach, and thus, it is a more agile approach as it enables some decisions to be postponed closer to delivery time when information about the system's requirements is more precise.

Time and space partitioning an existing kernel, nevertheless, is not an easy task. In some cases, because of a functionality's level of coupling with kernel or its functional requirements (e.g., compatibility with hardware-dependent, kernel-level software), ensuring that time and space partitioning is possible, and at the same time, fulfilling the functionality's functional requirements may depend on a significant engineering effort. To address this issue, functionality farming relies on various partition types. Each partition type provides distinct levels of partitioning (e.g., from time-only partitioning to both time and space partitioning), as well as they fulfill different functional requirements (e.g., from compatibility with hardware-dependent, kernel-level software to compatibility with only hardware-independent software). These different partition types, not only increase the extent to which functionality farming is more easily accomplished, but also enable an even more progressive reduction of the size of the kernel. Thus, these different partition types enable fast design space exploration, and reduce the associated risk. As an example, consider the following. In the beginning, when a functionality is tightly coupled with the kernel, achieving space partitioning may require a significant engineering effort, while achieving time partitioning may not be as hard, then, a partition which provides time-only partitioning can be used. At this stage, it cannot be expected that the size of the kernel will be reduced; nevertheless, as will be shown later, time-only partitioning can reveal interesting design alternatives, as well as it enables bad design alternatives to be ruled out early on. After modifying the functionality to enable space partitioning, it may still depend on compatibility with kernel-level software, then, a partition providing time and space partitioning as well as compatibility with kernel-level software can be used. Lastly, after the functionality is made hardware-independent, then, a lightweight, hardware-independent partition can be used.

Functionality farming alone, however, despite the benefits, still depends on a significant engineering effort, as will be shown later, and its effects are often very hard to predict, meaning that

the associated risk is still high.

This thesis also presents FF-AUTO, a tool which performs functionality farming semi-automatically in POK/rodosvisor. With FF-AUTO, the engineering effort, and thus, the risk associated with functionality farming is significantly reduced, making it also an ideal tool for design space exploration. As explained earlier, POK/rodosvisor supports three partition types:

- ARINC 653 partitions: support only hardware-independent software but enforce both time and space partitioning;
- privileged partitions: support all kinds of software (i.e., hardware-dependent and hardware-independent software), enforce time partitioning but not space partitioning;
- and virtual machines: support all kinds of software, and enforce both time and space partitioning (at the cost, however, of a virtualization overhead).

This thesis also demonstrates how functionality farming is able to improve the design and the performance of POK/rodosvisor, as well as how FF-AUTO enables a significant reduction of the required engineering effort. It was not possible to demonstrate a reduction of the size of the kernel since POK/rodosvisor is already a very small kernel (very close to microkernel). Finally, even though functionality farming and FF-AUTO, currently, target only POK/rodosvisor, its underlying methodology can be applied to any other operating system.

1.4. Thesis Structure

This rest of this thesis is organized as follows.

In the next chapter, *Chapter 2. Rodosvisor*, a hypervisor featuring full virtualization of the PowerPC 405 (i.e., a representative of low-end hardware), is described.

The following chapter, *Chapter 3. POK/rodosvisor*, describes how POK has been extended with support for privileged partitions, and how POK and Rodosvisor have been integrated, becoming POK/rodosvisor. Moreover, the virtualization overhead and the performance profile of POK/rodosvisor are presented. POK/rodosvisor's size of the trusted computing base and memory footprint for various configurations are also presented.

Chapter *4. Model-Driven Engineering using Ocarina* explains how support for privileged partitions and virtual machines has been added to AADL and Ocarina; an evaluation of the reduction of the engineering effort enabled by AADL and Ocarina is presented as well.

In Chapter 5. Functionality Farming, FF-AUTO and its underlying methodology are presented; two

use cases are also presented, demonstrating that functionality farming is able to improve the design and the performance of POK/rodosvisor, and that FF-AUTO contributes to a significant reduction of the required engineering effort, and thus, of the associated risk.

Finally, *Chapter 6. Conclusion*, ends this thesis with a summary of the major findings and contributions.

1.5. Publications

The following publications were performed in the course of this work:

- A. Carvalho, F. Afonso, P. Cardoso, J. Cabral, M. Ekpanyapong, S. Montenegro, and A. Tavares, “Functionality Farming in POK/rodosvisor,” in the International Journal of Computer Science and Software Engineering 5, no. 8 (2016): 161-174.
- A. Carvalho, V. Silva, F. Afonso, P. Cardoso, J. Cabral, M. Ekpanyapong, S. Montenegro, and A. Tavares, “Full Virtualization on Low-End Hardware: a Case Study,” in the 42nd Annual Conference of IEEE Industrial Electronics Society, Florence, Italy, 2016.
- A. Carvalho, F. Afonso, P. Cardoso, J. Cabral, M. Ekpanyapong, S. Montenegro, and A. Tavares, “Cache full-virtualization for the PowerPC 405-S,” in the 11th IEEE International Conference on Industrial Informatics, Bochum, Germany, 2013.
- A. Tavares, A. Carvalho, P. Rodrigues, P. Garcia, T. Gomes, J. Cabral, P. Cardoso, S. Montenegro, and M. Ekpanyapong, “A customizable and ARINC 653 quasi-compliant hypervisor,” in the 2012 IEEE International Conference on Industrial Technology, Athens, Greece, 2012.
- A. Tavares, A. Didimo, S. Montenegro, T. Gomes, J. Cabral, P. Cardoso, and M. Ekpanyapong, “RodosVisor - an Object-Oriented and Customizable Hypervisor: The CPU Virtualization,” in the Conference on Embedded Systems, Computational Intelligence and Telematics in Control, Würzburg, Germany, 2012.

2. Rodosvisor

2.1. Introduction

As explained in the previous chapter, *Section 1.1. Full Virtualization on Low-End Hardware*, the use of a hypervisor as a separation kernel in integrated architectures is being considered, as it not only provides time and space partitioning, but also provides compatibility with legacy software [15]–[19]. However, the hypervisors found in the literature, as will be demonstrated in *Section 2.2. Virtualization and Hypervisors*, either (1) do not provide complete legacy compatibility as they rely on paravirtualization, which requires legacy software to be modified to fit a hypervisor-specific, often proprietary interface, or (2) do provide complete legacy compatibility but depend on high-end hardware (i.e., hardware with virtualization extensions) which leads to large system size and volume, high weight, high power consumption, and high cost. Full virtualization on low-end hardware (i.e., hardware without virtualization extensions), on the other end, has none of those advantages.

In this thesis, Rodosvisor, a hypervisor featuring full virtualization of the IBM PowerPC 405 [22], is presented. The PowerPC 405 has been chosen because (1) it is a simple, low power, low cost processor, especially dedicated to embedded system and thus, a good representative of low-end hardware; and because (2) Xilinx [48], [49] provides good development support, enabling the construction and evaluation of various hardware configurations, such as single-core and dual-core processor configurations. Alternatively, ARM could have been chosen as the processor architecture; however, a PowerPC-405-equivalent ARM processor is much more complex, and thus, too risky for this case study. Consider, for example, that an ARMv7, application profile, virtual memory system architecture implementation, features more than 20 privileged instructions and more than 100 privileged registers; the PowerPC 405, on the other end, features 19 privileged instructions and around 40 privileged registers.

Rodosvisor, similarly to [50] provides “cycle-accurate” virtual processors (VCPU) (i.e., their execution time does not effect the execution time of co-existing virtual processors) compatible with the PowerPC 405 (the CPU). Rodosvisor has not an explicit representation of a virtual machine; instead, a VCPU is configured to enforce the boundaries of the virtual machine, such as its dedicated and shared memory address space, and I/O devices. Moreover, Rodosvisor, by itself, cannot support a running system (e.g., it does not provide a scheduler); it needs to be integrated into a host which provides the services required for a running system. As will be explained in *Section 2.3. Interface with the Host*, the host is responsible for:

- initialization and scheduling of VCPUs;
- loading, resuming, pausing and saving VCPUs;
- and, whenever a VCPU is running, for redirecting interrupts to the current VCPU.

In the next chapter, *Chapter 3. POK/rodosvisor*, among other things, the integration of Rodosvisor with POK [23], an ARINC 653 separation kernel, becoming POK/rodosvisor, is described. An evaluation of the virtualization overhead, POK/rodosvisor's performance profile, as well as POK/rodosvisor's size of the trusted computing base and memory footprint for various configurations are also presented.

Some publications have been made based on a prior version of the Rodosvisor [51]–[53]. In those publications a bare metal version of Rodosvisor is described. In this thesis, unless otherwise noted, when Rodosvisor is referred, it means the newer, host-assisted (a.k.a., hosted) version of Rodosvisor.

2.1.1. Chapter Organization

This chapter is organized as follows.

In the next section, *Section 2.2. Virtualization and Hypervisors*, background on virtualization and hypervisors is given.

In *Section 2.3. Interface with the Host*, the interface with the host and its responsibilities are described.

The methodologies used for full virtualization of the PowerPC 405 are explained in *Section 2.4. Full Virtualization*; this includes virtualization of:

- the instruction set, *Section 2.4.1. Instruction Set*;
- the timers, *Section 2.4.2. Timers*;
- the memory management unit, *Section 2.4.3. Memory Management Unit*;
- and interrupts, *Section 2.4.4. Interrupts*.

In *Section 2.5. I/O virtualization* and *Section 2.6. Paravirtualization*, the various mechanisms available for I/O virtualization and paravirtualization, respectively, are described.

In *Section 2.7. Future Work*, future work is proposed and finally, in *Section 2.8. Summary*, a summary of this chapter is given.

2.2. Virtualization and Hypervisors

In the same way as a separation kernel enforces time and space partitioning between partitions, a hypervisor enforces time and space partitioning between virtual machines, as illustrated in Figure 2.1. The software running “inside” a partition or virtual machine is called “the guest.” As in [20], "a virtual machine is taken to be an efficient, isolated duplicate of the real machine," and, a hypervisor: "first, [...] provides an environment for programs which is essentially identical with the original machine; second, programs run in this environment show at worst only minor decreases in speed; and last, [it] is in complete control of system resources." Furthermore [20] adds:

- "Any program run under the [hypervisor] should exhibit an effect identical with that demonstrated if the program had been run on the original machine directly, with the possible exception of differences caused by the availability of system resources and differences caused by timing dependencies."
- "It demands that a statistically dominant subset of the virtual processor's instructions be executed directly by the real processor, with no software intervention by the [hypervisor]." (This in particular, distinguishes software virtualization from software emulation.)
- "The [hypervisor] is said to have complete control of ... resources if (1) it is not possible for a program running under it in the created environment to access any resource not explicitly allocated to it, and (2) it is possible under certain circumstances for the [hypervisor] to regain control of resources already allocated."

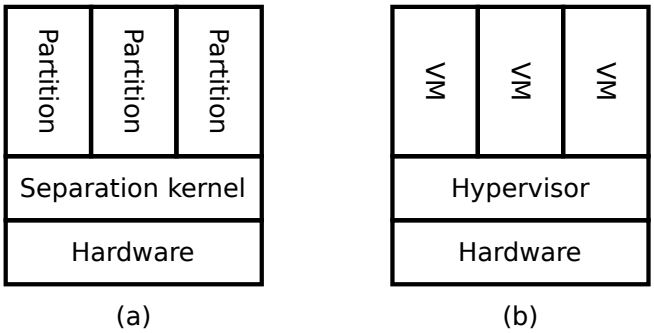


Figure 2.1. Illustration of (a) a separation kernel vs. (b) a hypervisor.

Through time and space partitioning a hypervisor enforces workload isolation, and thus, enables workload consolidation. Workload isolation guarantees that the behavior of a virtual machine, such as a failure, will not affect the rest of the system (i.e., other partitions and the hypervisor).

Furthermore, with workload isolation, the guests can be developed independently of each other, leading to lower development and certification effort. On top of workload isolation, workload consolidation enables a single computing unit to perform the same functions as multiple physically independent computing units, leading to: shorter bill of materials, lower size/volume and weight, lower power consumption, lower hardware development effort and cost, lower life-cycle costs, etc. [9], [54]. For example, a hypervisor enables a single computing unit to hold different operating system with different API (e.g., ARINC 653, OSEK, AUTOSAR, POSIX), leading to improved compatibility with legacy software, improved reuse and higher variability. The distinctive feature of a hypervisor is compatibility with legacy software, from simple applications with no operating system up to full fledged operating systems (e.g., Linux, Windows). As result, a hypervisor can improve reuse and reduce development and certification effort.

There are two types of virtualization, namely: full virtualization and paravirtualization. With full virtualization, the virtual machines established by the hypervisor provide an interface identical to that of the underlying hardware platform (i.e., the physical/real machine); therefore, such a virtual machine is capable of hosting legacy software with no modifications. There are, however, two approaches for the realization of full virtualization: (1) low-end hardware full virtualization, and (2) high-end hardware full virtualization. Low-end hardware full virtualization relies on the mechanisms provided by the underlying hardware platform, which, however, were not originally thought for the realization of full virtualization, and therefore, not all hardware platforms fulfill the necessary requirements [20]. Moreover, it is often claimed that it is not feasible due to an unacceptably high virtualization overhead; (e.g., [21]); however, we were unable to find real-world quantitative results to support those claims. High-end hardware full virtualization, on the other end, depends on hardware with virtualization support [55]–[59], which lead to a large system size, weight, power consumption, and cost. With paravirtualization, an alternative to full virtualization, the (para)virtual machines established by the hypervisor do not provide an interface identical to the underlying hardware platform, and instead, provide a different, more efficient interface. Paravirtualization should provide better performance than low-end hardware full virtualization, and it is the only option on hardware platforms which do not fulfill the requirements for full virtualization. When paravirtualization is used, however, the guests need to be modified to fit a hypervisor-specific, often proprietary interface. In Table 2.1, it can be seen that most hypervisors provide high-end hardware full virtualization and or paravirtualization; only Proteus [60], [61], and Rodosvisor, presented in this thesis, do support low-end hardware full virtualization. The authors of Proteus, however, failed to back their claims by demonstrating compatibility with legacy software,

such as a general-purpose operating system like Linux.

Table 2.1. Comparison of existing hypervisors in terms of low-end hardware full virtualization (LH-FV), high-end hardware full virtualization (HH-FV), and paravirtualization (PV).

Hypervisor	LH-FV	HH-FV	PV
AIR [136], [137]			x
ARLX [19], [135]		x	x
Bruns, 2013 [138]			x
Codezero [140]			x
Denali [121]–[123]			x
Green Hills Integrity Multivisor [18]		x	x
Joe, 2012 [64]		x	
KVM [62], [63]		x	
LynxSecure Separation Kernel Hypervisor [16]		x	x
NOVA [139]		x	
OKL4 [47]		x	x
PikeOS [17]		x	x
Proteus [60], [61]	x		x
QNX Hypervisor [141]		x	
Real Time Systems GbmH Hypervisor [125]		x	x
SierraVisor [134]		x	x
Spumone [126]			x
Wind River VxWorks (Virtualization Profile) [127]		x	x
X-Hyp [128]			x
Xen [129]		x	x
XtratuM [15], [130]–[132]			x
Zampiva, 2015 [133]		x	x

Independently of the type of virtualization, a hypervisor can be classified as bare metal (or type 1), or as hosted (or type 2). Figure 2.2 illustrates the difference between bare metal and hosted hypervisors. A bare metal hypervisor, Figure 2.2(a), sits directly on top of the hardware platform and has full control over it. A hosted hypervisor, Figure 2.2(b), sits on top of an existing operating system, i.e., the host operating system (e.g., Linux, Windows), with which it cooperates. A hosted hypervisor usually requires special support from the host operating system (e.g., kernel modules), usually provided by the same vendor as the hypervisor application. Some authors refer to the

combination of operating system support for virtualization and the hypervisor application as a type 1.5 hypervisor, illustrated in Figure 2.2(c). With complete control over the hardware platform, a bare metal hypervisor is able to provide higher performance than a hosted hypervisor. On the other end, a hosted hypervisor can use the services provided by the host operating system to simplify its design and implementation. Moreover, a hosted hypervisor enables the host operating system to support workloads other than virtual machines. Of all the hypervisors shown in Table 2.1, only KVM [62], [63] and Joe, 2012 [64], are hosted hypervisors; all the others are bare metal hypervisors. Rodosvisor, developed during this work, by itself, can be considered as a hosted hypervisor; POK/rodosvisor, i.e., the result of the integration of Rodosvisor and POK, on the other end, can be considered as a bare metal hypervisor.

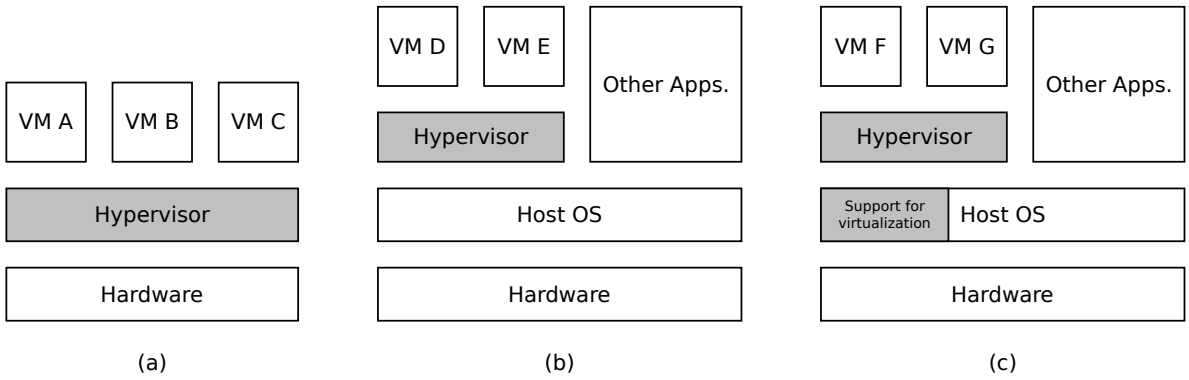


Figure 2.2. Hypervisor types: (a) bare metal, (b) hosted, and (c) type 1.5.

Historically, virtualization was first introduced in the 1960s by IBM with CP/CMS [65]. First, to solve the problem of hardware scarcity, enabling the same hardware to be shared among multiple users, and shortly thereafter to enable reuse of legacy software [66]. It became “meaningless” with the explosion of the personal computer but reappeared on the server side of the Internet to improve resource utilization and reduce power consumption [66], and it is now finding its way to highly-constrained embedded systems.

2.3. Interface with the Host

The expected interface between the host and a VCPU is illustrated in Figure 2.3. When a VCPU is scheduled, its partition window begins, and the host must “load” the VCPU through the corresponding “load” method in the VCPU's interface. The “load” method requires as a parameter the duration of the partition window (a.k.a., quantum). After a VCPU has been loaded, it assumes full control over the CPU and, when the partition window is over, the VCPU is responsible for

returning control back to the host. The VCPU, when requested to load, updates its internal state for the time since it was last unloaded/saved, and sets part of the CPU state to reflect the current state of the VCPU. The part of the CPU state that is set is such that it does not interfere with the operation of the VCPU (e.g., configure the memory management unit but do not enable it); this decreases the effort required for resuming and pausing the VCPU, described below, which need to be performed far more often.

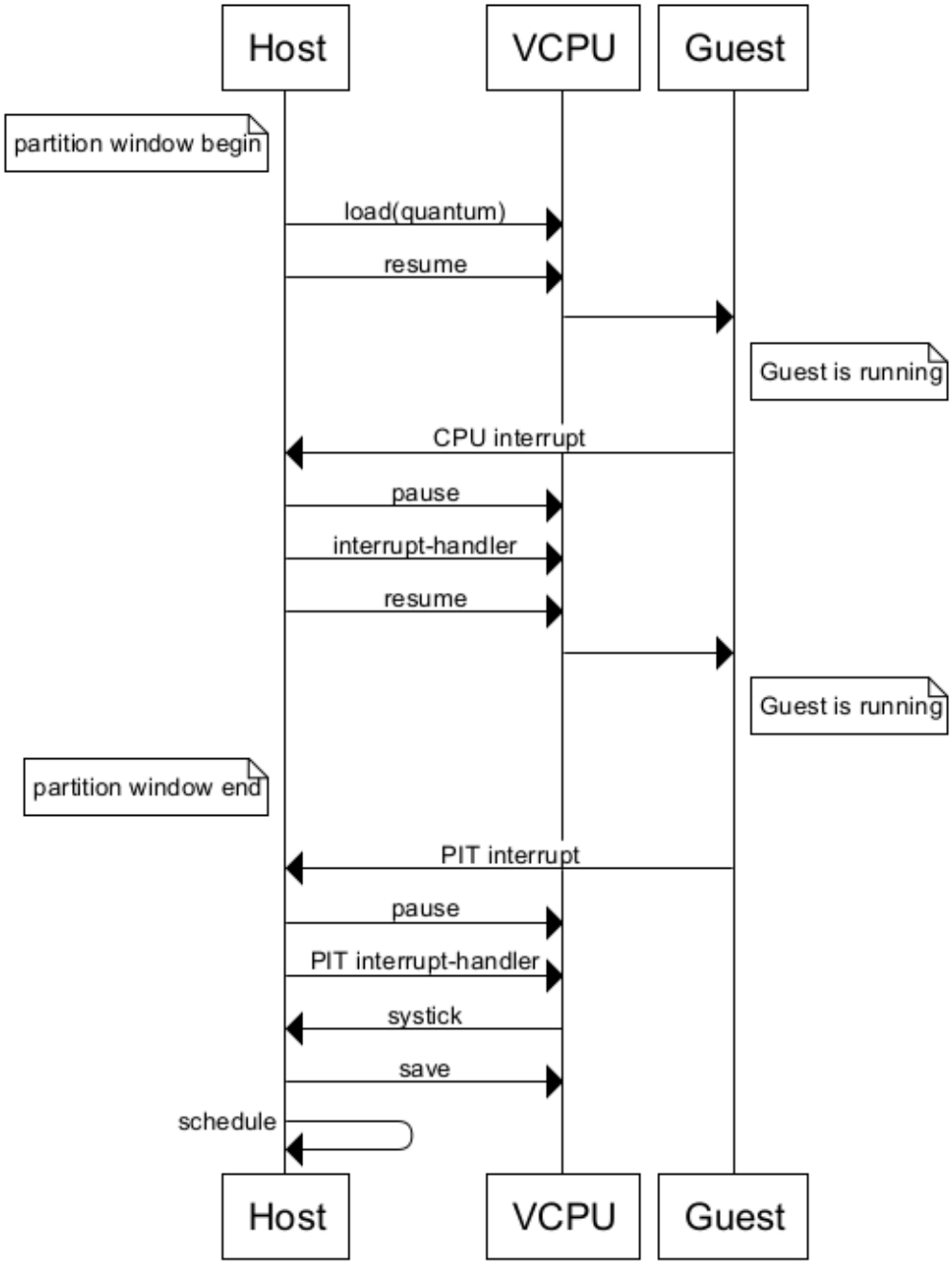


Figure 2.3. Illustration of the interaction between the host, the VCPU and the guest.

After loading the VCPU, or after calling one of the VCPU interrupt-handlers described below, the host must “resume” the VCPU. The VCPU, when requested to resume, updates the remaining state of the CPU (i.e., not updated by “load”) to reflect the current state of the VCPU and resumes the execution of the guest.

After the execution of the guest has been resumed, any CPU interrupt will be first handled by the host. The host is then responsible for:

1. saving the guest's user mode state in the VCPU's register file;
2. pausing the VCPU by calling the corresponding method in the VCPU's interface (“pause” is the inverse of “resume,” i.e., it saves and updates the part of the CPU state which may interfere with the operation of the VCPU);
3. calling the corresponding VCPU interrupt-handler.

A special interrupt is the one that signals the end of the VCPU partition window; in the PowerPC 405 this is the programmable-interval timer (PIT) interrupt, described in more detail in *Section 2.4.2. Timers*. Like any other interrupt, the host is responsible for pausing the VCPU and calling the respective VCPU interrupt-handler. However, upon finding that a particular PIT interrupt is signaling the end of the partition window, the VCPU calls the host's “systick” method. (In *Section 2.4.2 Timers*, it is described how the VCPU finds that a particular PIT interrupt is signaling the end of the partition window.) The host, in turn, is responsible for saving the VCPU and calling the scheduler, effectively returning control back to the host. When a VCPU is requested to “save,” it saves and updates the remaining part of the CPU state to reflect the host's expected state of the CPU.

2.4. Full Virtualization

In the following subsections the methodologies used to accomplish full virtualization of the PowerPC 405 are described. This includes virtualization of the instruction set, *Section 2.4.1. Instruction Set*, of time and of the timers, *Section 2.4.2. Timers*, of the memory management unit, *Section 2.4.3. Memory Management Unit*, and of the interrupts, *Section 2.4.4. Interrupts*.

2.4.1. Instruction Set

The instruction set of the PowerPC 405 is divided in two subsets: (1) the privileged instruction subset, and (2) the user (or non-privileged) instruction subset. The PowerPC 405 also provides two execution modes: privileged mode, and user mode (a.k.a., problem state or non-privileged mode). In privileged mode, the PowerPC 405 allows the execution of the entire instruction set: the privileged

and user instruction subsets. In user mode, however, only the user instruction subset is allowed to be executed, and an attempt to execute a privileged instruction leads to the generation of a program exception (or interrupt). This is illustrated in Figure 2.4.

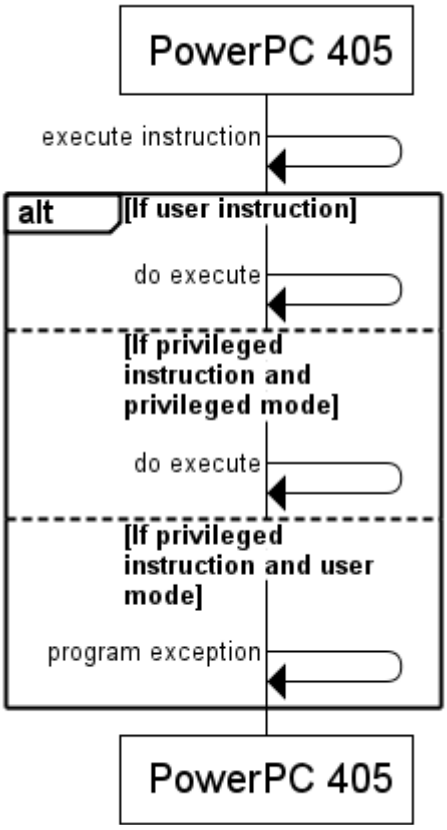


Figure 2.4. A sequence diagram illustrating how the PowerPC 405 handles the user and privileged instruction subsets when in the user and privileged modes.

When a guest is running, the CPU is always in user mode, where execution of privileged instructions is not allowed. In privileged mode, with direct access to the privileged instruction set, the guest would have full control over the CPU, and could violate time and space partitioning. The guest, however, expects access to the privileged instruction set and associated functionality, and thus, access to the privileged instruction set is provided indirectly, while guarantying that time and space partitioning is not violated, as will be explained.

When a guest is running, the CPU is always in user mode and, therefore, whenever the guest executes a user instruction, that instruction is executed directly by the CPU; however, whenever the guest attempts to execute a privileged instruction a program interrupt is generated instead. As explained in *Section 2.3. Interface with the Host*, whenever a CPU interrupt is generated, it is first

handled by the host which, in turn, calls the corresponding VCPU interrupt-handler, in this case, the VCPU program interrupt-handler. As illustrated in Figure 2.5, the VCPU program interrupt-handler, checks the condition which lead to the generation of the program interrupt and then:

C1. If the condition was an attempt to execute a privileged instruction with the CPU configured in user mode, but the VCPU is in privileged mode (condition C1), then, the guest expects the privileged instruction to be executed normally and thus, the program interrupt-handler "manually" fetches, decodes, and emulates the execution of the privileged instruction.

C2. For all other conditions (condition C2), a program interrupt request is issued to emulate the generation of the original program interrupt in the VCPU. Interrupt requests and dispatching are discussed in *Section 2.4.4. Interrupts*.

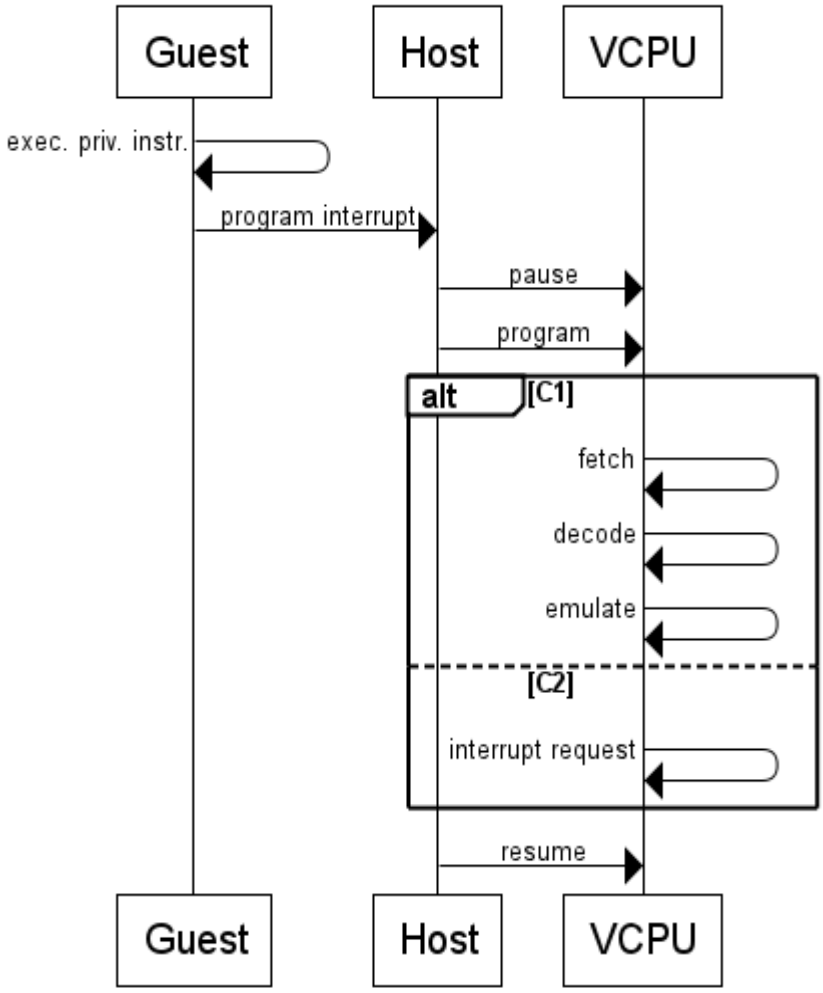


Figure 2.5. A sequence diagram illustrating how emulation of privileged instructions is accomplished.

In this section, it is only explained how privileged instructions are trapped, up to the point where the corresponding emulation routine is called; actual emulation is explained in the following sections.

2.4.2. Timers

The PowerPC 405 provides four timer units, namely: (1) the Time Base, (2) the Programmable-Interval Timer, (3) the Fixed-Interval Timer, and (4) the Watchdog. First, the Time Base, illustrated in Figure 2.6, is a 64-bit counter which increments, depending on the hardware configuration, at the same rate as the CPU's clock or according to an external clock signal. The current value of the Time Base is accessible for reading and writing from two Special-Purpose Registers (SPR): (1) the Time-Base Lower (TBL) for the least significant 32 bits; and (2) the Time-Base Upper (TBU) for the most significant 32 bits.

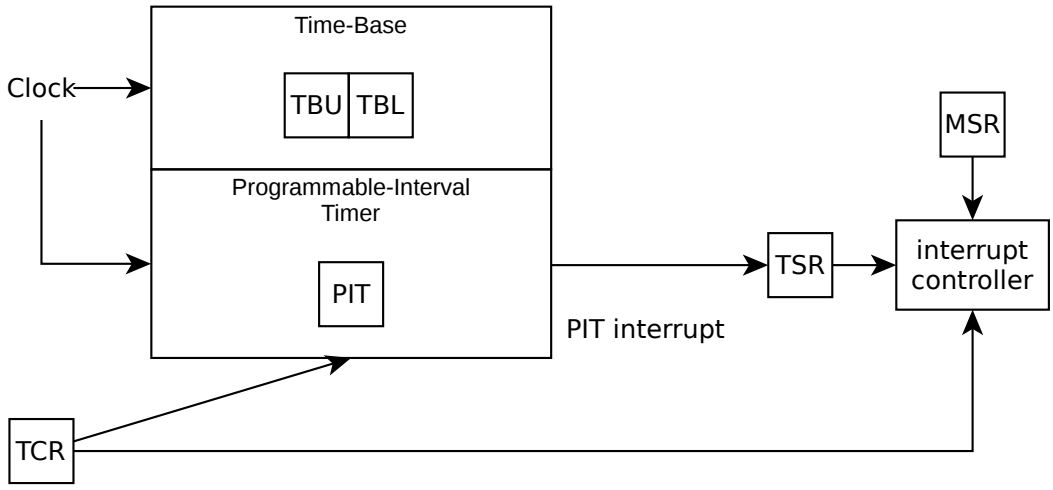


Figure 2.6. The Time Base and the Programmable-Interval Timer in the PowerPC 405.

Second, the Programmable-Interval Timer, also illustrated in Figure 2.6, is a 32-bit timer which decrements at the same rate as the Time-Base, and its current value is accessed through the Programmable-Interval Timer register (PIT). When its current value decrements from one to zero, the timer overflows, the PIT Interrupt Status (PIS) bit field in the Timer Status Register (TSR) is set and, if enabled, a PIT interrupt is generated. The PIT interrupt is enabled through the PIT Interrupt Enable (PIE) bit field in the Timer Control Register (TCR), and by the External interrupt Enable (EE) bit field in the Machine State Register (MSR).

Third, the Fixed-Interval Timer, illustrated in Figure 2.7, keeps track of changes in specific bits of the Time Base; when a selected bit changes from zero to one, the timer “overflows,” the FIT

Interrupt Status (FIS) bit field in the TSR is set and, if enabled, a FIT interrupt is generated. The specific bit of the Time Base is selected through the FIT Period (FP) bit field in the TCR, and the FIT interrupt is enabled through the FIT Interrupt Enable (FIE) bit field in the TCR, and by EE in the MSR.

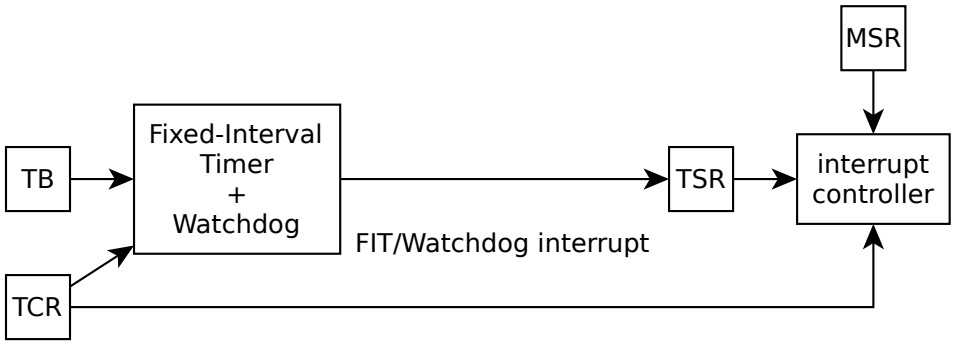


Figure 2.7. The Fixed-Interval Timer and the Watchdog in the PowerPC 405.

Forth and last, the Watchdog, also illustrated in Figure 2.7, similarly to the Fixed-Interval Timer, also keeps track of changes from zero to one (i.e., an “overflow”) on selected bits of the Time Base. When there is an overflow, the Enable Next Watchdog (ENW) bit field in the TSR is set; if there is an overflow and ENW is already set, the Watchdog Interrupt Status (WIS) bit field in the TSR is also set and, if enabled, a Watchdog interrupt is generated; if, however, there is an overflow and ENW and WIS are both set, the CPU will perform one of the following four actions: (1) no action; (2) a core-reset; (3) a chip-reset; or (4) a system-reset. The specific bit of the Time Base is selected through the Watchdog Period (WP) bit field in the TCR, and the action performed by the CPU when there is an overflow and both ENW and WIS are set is selected by the Watchdog Reset Control (WRC) bit field in the TCR. The Watchdog interrupt is enabled through the Watchdog Interrupt Enable (WIE) bit field in the TCR, and by the Critical interrupt Enable (CE) bit field in the MSR.

In summary, the timer units are configured through the following registers: TBL and TBU, TCR, PIT, and MSR; however, with the progression of time the following register are “silently” updated: (1) TBL, TBU, and PIT are updated at the same rate, as explained above; and (2) TSR is updated whenever there is an overflow of the timers.

All of the register mentioned above are accessible through the Move To Special Purpose Register (MTSPR) instruction for writing, and through the Move From Special Purpose Register (MFSPR) instruction for reading. These instructions are considered privileged instructions if the register being

accessed is also a privileged register; otherwise, they are considered user instructions. With the exception of TBL and TBU, all of the register mentioned above are privileged. TBL and TBU have two alias: (1) one read/write privileged register, and (2) one read-only user register.

In the rest of the section, it is described how access to the various registers is emulated and, as a consequence, how the behavior of the PowerPC 405 timer units is emulated. Currently, changing the value of the VCPU's TBL and TBU is not supported, and thus, this is not described. Guests, however, can still read the current value of the TBL and TBU, either through the privileged or the user alias. For improved performance and accuracy, the user alias is preferable as it is executed directly by the CPU. Similarly, emulation of the Watchdog is currently unsupported and it is not described.

2.4.2.1. PIT

As explained in *Section 2.3. Interface with the Host*, the CPU's PIT is used to keep track of the end the partition window and, at the same time, to emulate the behavior of the VCPU's PIT. This is accomplished by configuring the CPU's PIT with the alternative that will generate an interrupt first: (1) the time until the end of the partition window (a.k.a., the quantum), or (2) the VCPU's time until the next PIT interrupt (*Tnpi*). *Tnpi* is infinite, if the VCPU's PIT interrupt is disabled; otherwise, if the VCPU's PIT interrupt is enabled, *Tnpi* is equal to the value of the VCPU's PIT. As shown in Figure 2.8, the CPU's PIT will be loaded with the quantum (Q), if *Tnpi* is larger than the quantum (i.e., the associated interrupt will occur at a later time). Conversely, the CPU's PIT will be loaded with the value of the VCPU's PIT, if the *Tnpi* is smaller than the quantum (i.e., the associated interrupt will occur sooner).

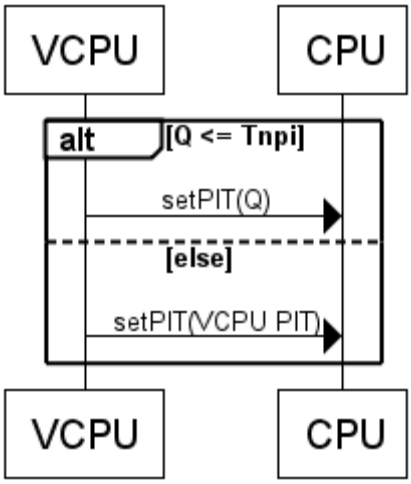


Figure 2.8. A sequence diagram illustrating the configuration of the CPU's PIT with either (1) the time until the end of the partition window (a.k.a., the quantum, or Q), or (2) the VCPU's time until the next PIT interrupt (Tnpi).

After loading the CPU's PIT, when it is necessary to update the quantum or the VCPU's PIT, if the CPU's PIT has been loaded with the quantum (i.e., if Q ≤ Tnpi), then, as illustrated in Listing 2.1:

- The updated quantum (Q') is the current value in the CPU's PIT (i.e., Q' = CPU PIT).
- The updated VCPU's PIT (VCPU PIT'), on the other end, depends on the last known value of the VCPU's PIT (VCPU PIT) (i.e., the value of the VCPU's PIT at the time when the CPU PIT was last loaded). If VCPU PIT is smaller than the time elapsed since the quantum was last loaded (i.e., if VCPU PIT ≤ Q - Q', where Q is the last known quantum, and, as described above, Q' the updated quantum), then, there is an overflow and the VCPU TSR's PIS is set to one. Moreover, if auto-reload is enabled (i.e., if VCPU TSR ARE == 1), the auto-reload value is calculated, and VCPU PIT' is set to that value; otherwise, if auto-reload is disabled, VCPU PIT' is set to zero (i.e., the timer is stopped). If, however, VCPU PIT is larger than the time elapsed since the quantum was last loaded (i.e., if VCPU PIT > Q - Q'), then, the updated VCPU's PIT is: VCPU PIT' = VCPU PIT - (Q - Q').

When, on the other end, it is necessary to update the quantum or the VCPU's PIT, but the CPU's PIT has been loaded with the value of the VCPU's PIT (i.e., if Q > Tnpi), then, as illustrated in Listing 2.1:

- The updated VCPU's PIT (VCPU PIT') is the current value in the CPU's PIT (i.e., VCPU PIT' = CPU PIT).

- The updated quantum is: Q' = Q - (VCPU PIT - VCPU PIT'), where Q' is the updated quantum, Q the last known quantum, VCPU PIT the last known value of the VCPU's PIT, and, as described above, VCPU PIT' the updated value of the VCPU's PIT.

```
1  if Q <= Tnpi
2
3      Q' = CPU PIT
4
5      if VCPU PIT <= Q - Q'
6
7          VCPU TSR PIS = 1
8
9          if VCPU TSR ARE == 1
10           VCPU PIT' = auto_reload()
11         else
12           VCPU PIT' = 0
13         endif
14
15     else
16
17         VCPU PIT' = VCPU PIT - (Q - Q')
18     endif
19
20 else
21
22     VCPU PIT' = CPU PIT
23     Q' = Q - (VCPU PIT - VCPU PIT')
24 endif
```

Listing 2.1. Pseudo-code illustration of the procedure used to update the quantum and the VCPU's PIT.

Hence, when a guest requests the value of the VCPU's PIT, then, as shown in Figure 2.9, the quantum and the VCPU's PIT are updated as described above, and the updated value of the VCPU's PIT is returned to the guest.

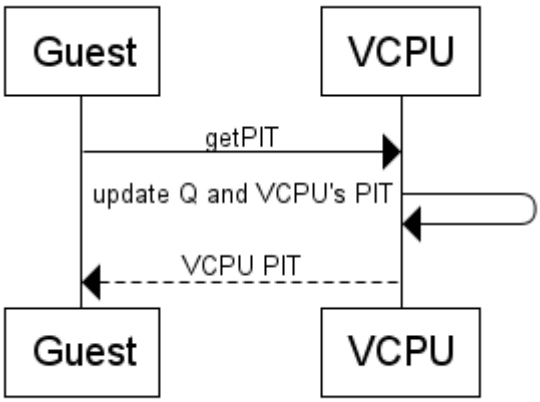


Figure 2.9. VCPU operation when a guest requests the value of the VCPU's PIT.

Conversely, when a guest requests a change to the value of the VCPU's PIT, then, as illustrated in Figure 2.10:

1. The quantum and the VCPU's PIT are updated as described above.
2. The VCPU's PIT is overwritten with the value provided by the guest.
3. Based on the new VCPU's PIT and on the updated quantum, the CPU's PIT is loaded with the configuration which will generated an interrupt sooner (i.e., “loadPIT” in Figure 2.10), as explained earlier.

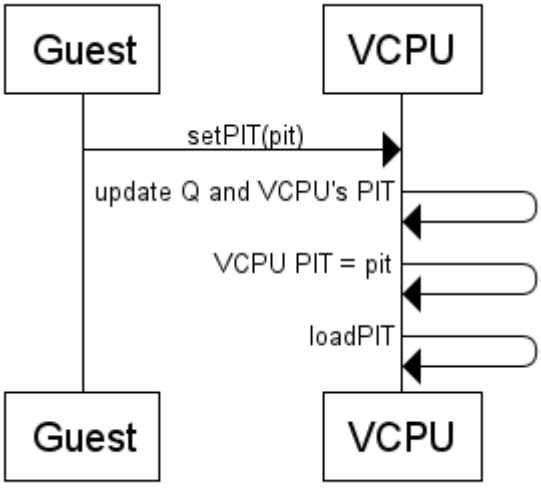


Figure 2.10. VCPU operation when a guest requests a change to the value of the VCPU's PIT.

2.4.2.2. TSR

When a guest request the current value of the VCPU's TSR, then, as illustrated in Figure 2.11:

1. The VCPU TSR's PIS is updated, as explained in the previous section.
2. The VCPU TSR's FIS is updated as follows: if the current CPU TSR's FIS is set, then, set the VCPU TSR's FIS; otherwise, the VCPU TSR's FIS remains unchanged.

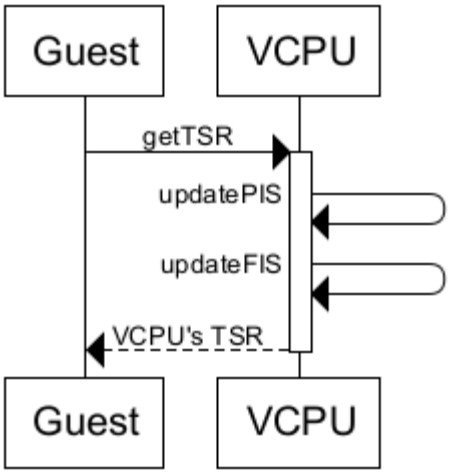


Figure 2.11. VCPU operation when a guest requests the value of the VCPU's TSR.

The TSR is a clear-on-write register: an input bit set to one, clears the corresponding bit in the TSR; an input bit set to zero, keeps the corresponding bit in the TSR unchanged. As shown in Figure 2.12, when a guest writes to the VCPU's TSR:

1. The VCPU TSR's PIS and FIS are updated, as explained above.
2. The input value is used to clear-on-write (COW) the updated VCPU's TSR.
3. The input value is used to clear-on-write only the CPU TSR's PIS and FIS.

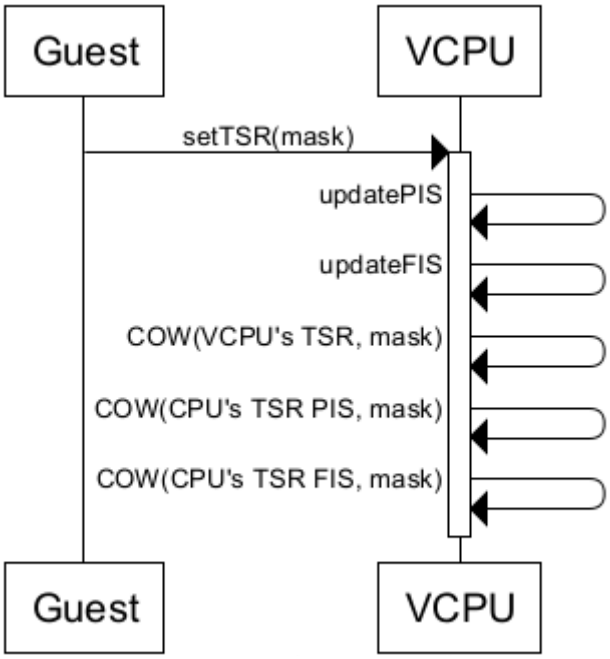


Figure 2.12. VCPU operation when a guest writes to the VCPU's TSR.

2.4.2.3. TCR

When a guest requests a change to the current value of the VCPU's TCR, the CPU's TCR is updated as shown in Table 2.2. The CPU's TCR not only depends on the VCPU's TCR but it also depends on the VCPU's MSR; therefore, whenever the VCPU's MSR is updated, the CPU's TCR is updated as well. The VCPU's TCR is never “silently” updated, therefore, when a guest requests the current value of the VCPU's TCR, then, the current value of the VCPU's TCR is returned without the need for any additional operation.

Table 2.2. Setting of the CPU's TCR based on the VCPU state.

CPU's TCR	Setting
WP (Watchdog Period)	Always set to zero. Setting this field to a value different than zero is currently not supported.
WRC (Watchdog Reset Control)	Always set to zero. Setting this field to a value different than zero is currently not supported.
WIE (Watchdog Interrupt Enable)	Always set to zero. Setting this field to a value different than zero is currently not supported.
PIE (PIT Interrupt Enable)	Always set to one (i.e., PIT interrupts enabled). The CPU's PIT is always set to generate either quantum-related interrupts or VCPU PIT interrupts.
FP (FIT Period)	According to the VCPU TCR's FP.
FIE (FIT Interrupt Enable)	According to the VCPU TCR's FIE if and only if the VCPU MSR's EE is set to one (i.e., external interrupts enabled); otherwise, set to zero.
ARE (Auto Reload Enable)	Always set to zero. Setting this field to a value different than zero is currently not supported.

2.4.2.4. Interrupts

As explained before, when there is a CPU PIT interrupt, and the system is in state P1 (i.e., the CPU's PIT contains the quantum), that interrupt marks the end of the partition window and the host is called to handle the event (i.e., save the VCPU, call the scheduler, etc.). Otherwise, if the system is in state P2 (i.e., the CPU's PIT is an alias of the VCPU's PIT), that interrupt marks a VCPU PIT interrupt and, as illustrated in Figure 2.13, in such case:

1. The CPU TSR's PIS is cleared to prevent further CPU PIT interrupts.
2. The VCPU TSR's PIS is set, and an interrupt request is performed. Interrupt requests and

dispatching are discussed in *Section 2.4.4. Interrupts*.

3. The VCPU's PIT and the quantum are updated, based on the last known value of the VCPU's PIT, and the current time (found in the Time Base), similarly to what has been explained in *Section 2.4.2.1. PIT*.
4. Based on the updated VCPU's PIT and the updated quantum, load the CPU's PIT with the shorter configuration, as explained before.

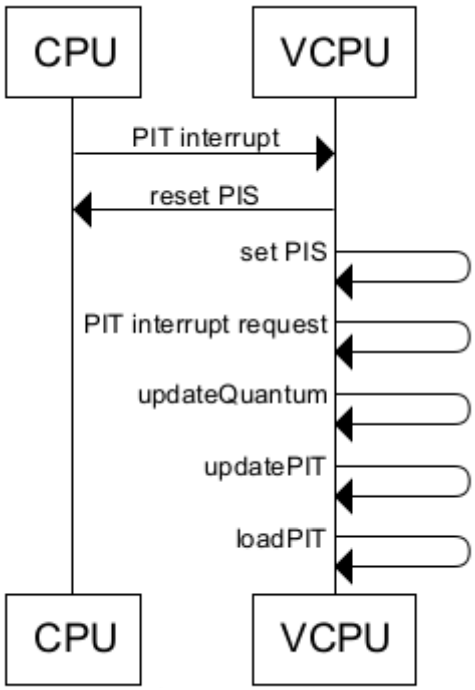


Figure 2.13. VCPU operation when there is a CPU PIT interrupt and the system is in state P2.

A FIT interrupt, on other end, always marks a VCPU FIT interrupt and, as shown in Figure 2.14, when that occurs:

1. The CPU TSR's FIS is cleared to prevent further CPU FIT interrupts
2. The VCPU TSR's FIS is set, and an interrupt request is performed. Interrupt requests and dispatching are discussed in *Section 2.4.4. Interrupts*.

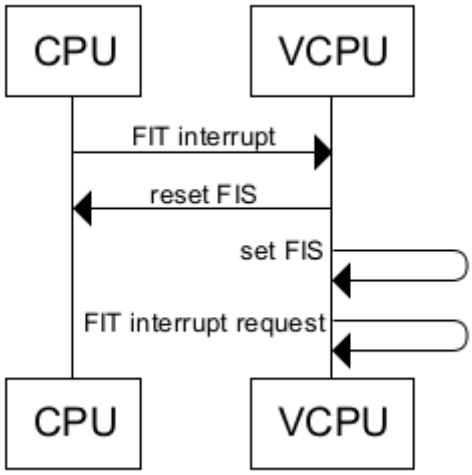


Figure 2.14. VCPU operation when there is a CPU FIT interrupt.

2.4.2.5. Preemption

Whenever a VCPU is not loaded, elapsed time must still be accounted for. Therefore, when a VCPU is saved, the preemption time is saved and then, when a VCPU is loaded again, the time elapsed since the last preemption time is used to update all the timers.

2.4.3. Memory Management Unit

In the following subsections, the methodologies used for the virtualization of the memory management unit (MMU) in the PowerPC 405 are described. First, in *Section 2.4.3.1. PowerPC 405 Memory Management Unit*, the PowerPC 405's MMU is described. Second, an overview of the MMU's virtualization is given in *Section 2.4.3.2. Overview*. Third and forth, the mappings required between the VCPU's real and virtual modes and the CPU's virtual mode, are explained in *Section 2.4.3.3. Real Mode Translation* and *Section 2.4.3.4. Virtual Mode Translation*, respectively.

2.4.3.1. PowerPC 405 Memory Management Unit

In the PowerPC 405 the address specified by an instruction fetch or a data load/store (i.e., an effective address) always goes through the memory management unit (MMU), as illustrated in Figure 2.15. The MMU provides: (1) address translation, (2) access control, and (3) storage attributes. Address translation is enabled for instruction fetches through the instruction relocation (IR) bit field, and for data loads and stores by the data relocation (DR) bit field, both in the Machine State Register (MSR); if set to 1, address translation is enabled; otherwise (i.e., if set to 0), address translation is disabled.

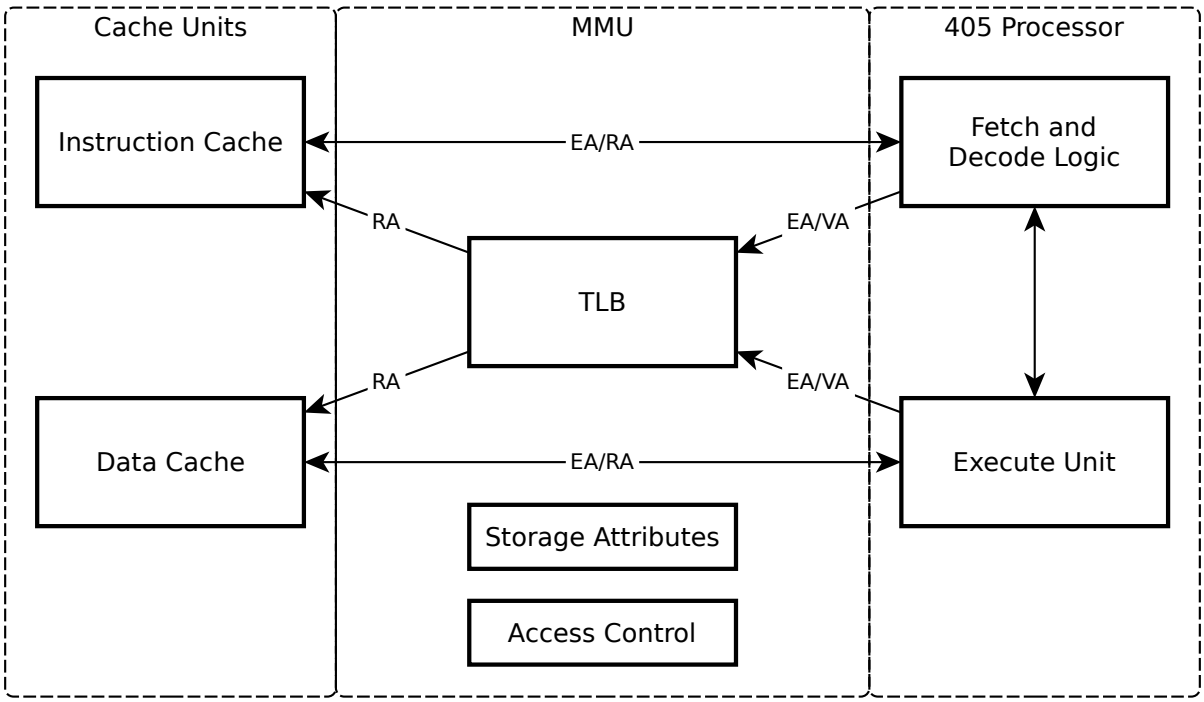


Figure 2.15. The logical connections between the cache units, the memory management unit and the 405 processor within a PowerPC 405.

When address translation is disabled (i.e., real mode), an effective address (EA) is considered a real address (RA) and is used to address the resources outside the CPU (e.g., memory and I/O), as illustrated in Figure 2.15 by the connection between the “Fetch and Decode Logic” and the “Instruction Cache,” and by the connection between the “Execution Unit” and the “Data Cache.” In real mode, the MMU also provides storage attributes, which will be described later in this section.

When address translation is enabled (i.e., virtual mode), on the other end, an effective address (EA) is considered a virtual address (VA) and goes through the Translation Look-aside Buffer (TLB) where it is translated to a real address (RA) which is then used to access the resources outside the CPU; this is illustrated in Figure 2.15 by the connections from the “Fetch and Decode Logic” and the “Execution Unit” to the TLB.

The TLB is composed by 64 entries, and each TLB entry is divided into TLBHI and TLBLO, as shown in Figure 2.16. Each TLB entry specifies a virtual address space and a translation to a corresponding real/physical address space. In Table 2.3, the fields of a TLB entry related to the virtual address space and translation are described; the remaining fields are used for access control and storage attributes and will be described later in this section.

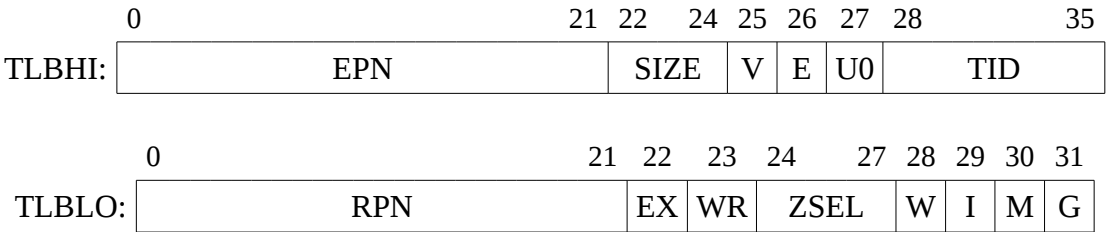


Figure 2.16. TLBHI and TLBLO.

Table 2.3. The bit fields in a TLB entry controlling address translation.

Field	Description
Effective Page Number (EPN)	Defines the start of the virtual address space; it is compared with the most significant bits of the virtual address specified by an instruction fetch or data load/store. If EPN, together with SIZE, does not match the virtual address specified, for all of the TLB entries, then, a TLB-miss interrupt is generated: (1) an instruction TLB-miss interrupt if it is an instruction fetch, or (2) a data TLB-miss interrupt if it is a data load/store.
SIZE	Defines the size of the virtual address space; it also defines how many bits of the virtual address are compared with EPN. Available sizes include: 1 KB, 4 KB, 16 KB, 64 KB, 256 KB, 1 MB, 4 MB, and 16 MB.
Valid (V)	Indicates whether or not the entry is valid, and thus, if it can be used for translation or not.
Translation ID (TID)	If not equal to zero, this field is compared with the value in the Process ID (PID) special-purpose register; if there is no match a TLB-miss interrupt is generated. If equal to zero, no comparison is performed.
Real Page Number (RPN)	When the virtual address matches all the other fields, this field replaces the upper bits of the virtual address to form a real/physical address. The actual number of bits replaced depends on SIZE.

Apart from address translation, the PowerPC 405's MMU also provides access control and storage attributes. Access control provides protection against execution as well as reading and writing from memory; violation of the access control configuration results in the generation of an instruction or

data storage interrupt, depending on the type of access. Access control is available only in virtual mode. Storage attributes, on the other end, enable, among other things, different sections of the address space to be handled differently depending on their contents (e.g., memory or I/O); violation of the restriction associated with some of the storage attributes may also result in the generation of an instruction or data storage interrupt. Storage attributes are available in both real and virtual modes.

In real mode, storage attributes are configured by a set of Special-Purpose Registers (SPR), each dedicated to a specific storage attribute. Each SPR is 32-bit, and each bit controls the storage attribute of a unique 128 MB section (from a total of 4 GB) of the real mode address space, as illustrated in Table 2.4. On the other end, in virtual mode, access control and storage attributes are controlled by the TLB and the Zone Protection Register (ZPR). Each TLB entry, enforces access control and storage attributes to the corresponding virtual address space or, after translation, to the corresponding real address space.

Table 2.4. Bit fields in a SPR controlling a real mode storage attribute and the corresponding 128 MB sections of the real address space affected (big-endian notation).

Bit	Begin	End
0	x'0000 0000' ..	x'07FF FFFF'
1	x'0800 0000' ..	x'0FFF FFFF'
N	*(x'0800 0000' × N)*	*(x'0800 0000' × (N + 1) - 1)*
31	x'F800 0000'	x'FFFF FFFF'

In Table 2.5, access controls available in virtual mode is shown alongside the SPR and or the fields in a TLB entry which control them. Access control is only applicable in virtual mode. Access control is configured by the following:

- TLB[EX], Execute enable in a TLB entry: enables execution of instructions within the associated address space.
- TLB[WR], Write enable in a TLB entry: enables data stores within the associated page
- TLB[ZSEL], Zone Select field in a TLB entry: selects one of 16 protection zones from the ZPR. A protection zone can override the access control configuration specified by TLB[EX]

and TLB[WR], as explained in Table 2.6.

- ZPR, Zone Protection Register: enables the definition of 16 protection zones which may or not override the access control configuration specified by TLB[EX] and TLB[WR], as explained in Table 2.6.

Table 2.5. Access control and its configuration in virtual mode.

Access control	**Virtual mode**
No access	TLB[ZSEL], ZPR
Write-enable	TLB[ZSEL], TLB[WR], ZPR
Execute-enable	TLB[ZSEL], TLB[EX], ZPR

Table 2.6. Bit fields in the Zone Protection Register and their effect in user and privileged modes.

Fields	**Setting**	**User mode**	**Privileged mode**
Z0 -- Z15	b'00'	No access: data load and store operations are not permitted.	Access controlled by TLB[EX] and TLB[WR].
	b'01'	Access controlled by TLB[EX] and TLB[WR].	Access controlled by TLB[EX] and TLB[WR].
	b'10'	Access controlled by TLB[EX] and TLB[WR].	No access restrictions.
	b'11'	No access restrictions.	No access restrictions.

Similarly, Table 2.7 shows that the following storage attributes are available:

- Guarded storage: controls whether or not the processor can perform speculative memory accesses to improve performance; it is controlled by the Storage Guarded Register (SGR), or by the Guarded (G) field in a TLB entry.
- Instruction cacheability: this storage attribute enables or not the instruction cache; it is configured by Instruction Cacheability Control Register (ICCR), or by the Inhibit cache (I)

field in a TLB entry.

- Data cacheability: this storage attribute enables or not the data cache; it is controlled by the Data Cacheability Control Register (DCCR), or by the I field in a TLB entry.
- Write strategy: when the data cache is enabled, this storage attribute controls whether a data store should update only the cache (write-back) or, if both the cache and the external memory should be updated (write-through); it is controlled by the Data Cache Write-through Register (DCWR), or by the Write-through (W) field in a TLB entry.
- Endianess: specifies whether the data is big-endian or little-endian; it is controlled by the Storage Little-Endian Register (SLER), or by the Endian (E) field in a TLB entry.
- User-defined 0: in the PowerPC 405, when this storage attribute is enabled, an interrupt is generated for any data load/store; it is controlled by the Storage User-defined 0 Register (SU0R), or by the User-defined 0 (U0) field in a TLB entry.

Table 2.7. Storage attributes, and their configuration in real mode and virtual mode.

Storage attribute	Real mode	Virtual mode
Guarded storage	SGR	TLB[G]
Instruction cacheability	ICCR	TLB[I]
Data cacheability	DCCR	
Write strategy	DCWR	TLB[W]
Endianess	SLER	TLB[E]
User-defined 0	SU0R	TLB[U0]

In summary, the PowerPC 405's MMU provides address translation, access control and storage attributes; when address translation is disabled, storage attributes are controlled by a set of SPR and there is no access control; when address translation is enabled, virtual-to-real address translation, access control and storage attributes are controlled by the TLB, the PID and the ZPR.

2.4.3.2. Overview

After reset, like the CPU, the VCPU is set to operate in real mode (i.e., instruction and data address

translation disabled). The guest modifies the VCPU's MMU configuration (e.g., mode, address translation, access control, storage attributes) through the execution of privileged instructions which, as explained in *Section 2.4.1. Instruction Set*, lead to the generation of a program interrupt and the execution of the corresponding emulation routine in the VCPU. The guest, however, executes always in the CPU's virtual mode, and the CPU's virtual mode is used to emulate the VCPU's real and virtual modes, as will be explained later. This approach is necessary because the CPU's real mode does not provide access control, and thus, the guest would be able to read, write and execute from anywhere in the memory address space, including other virtual machine's and the hypervisor's code and data. In other words, it is not possible to enforce space partitioning in real mode.

As illustrated in Figure 2.17, whenever the guest requests a change to the VCPU's real mode configuration:

1. The real mode configuration in the VCPU's register file is updated.
2. The VCPU's real mode configuration is translated to a CPU's virtual mode configuration which emulates the behavior of the VCPU's real mode configuration (this translation is described in *Section 2.4.3.3. Real Mode Translation*).
3. If the VCPU is in real mode, the CPU's virtual mode configuration is also updated.

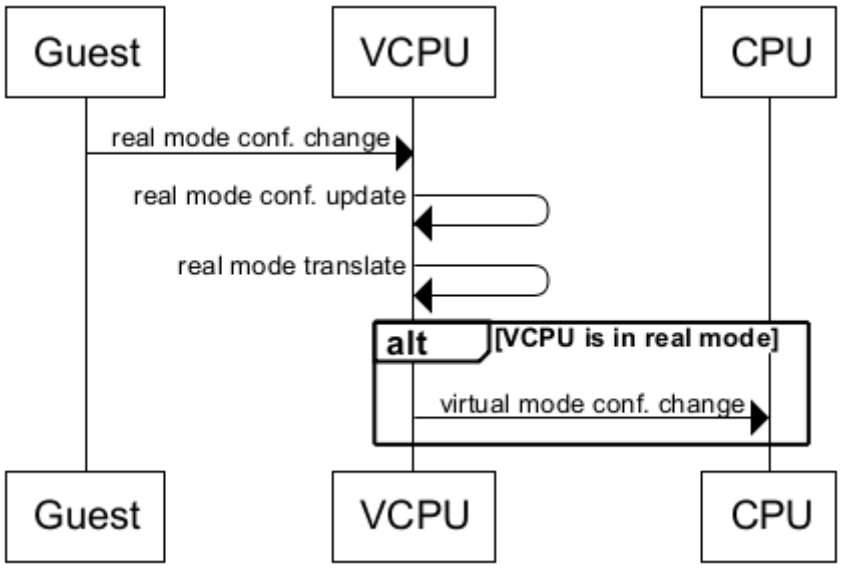


Figure 2.17. Handling of a change to the VCPU's real mode configuration.

Similarly, as illustrated in Figure 2.18, whenever the guest requests a change to the VCPU's virtual

mode configuration:

1. The virtual mode configuration in the VCPU's register file is updated.
2. The VCPU's virtual mode configuration is translated to a CPU's virtual mode configuration which emulates the behavior of the VCPU's virtual mode configuration (this translation is described in *Section 2.4.3.4 Virtual Mode Translation*).
3. If the VCPU is currently in virtual mode, the CPU's virtual mode configuration is also updated.

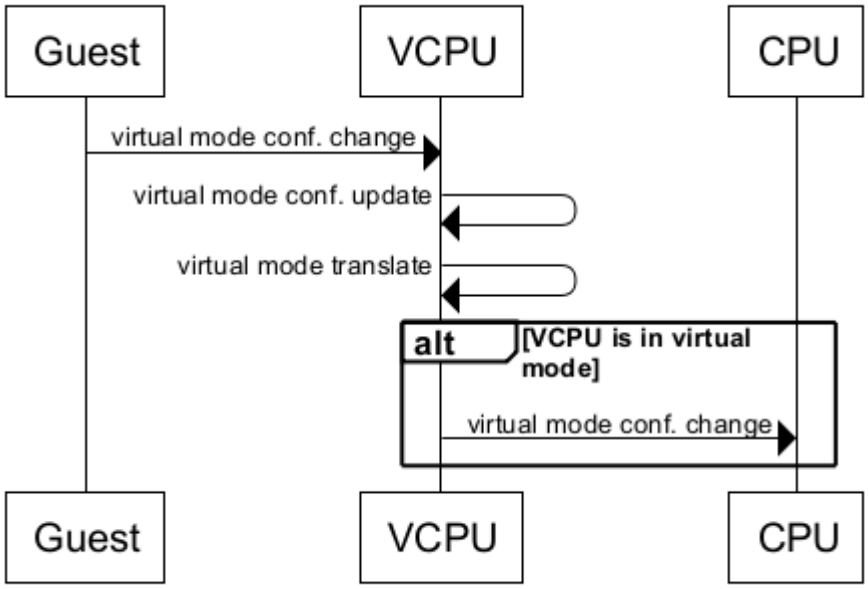


Figure 2.18. Handling of a change to the VCPU's virtual mode configuration.

Unlike the PowerPC 405's MMU, independent control over instruction and data address translation is not supported: either instruction and data translation are both enabled, or both disabled. Independent control over instruction and data address translation would lead to the following issues:

- Knowing that the VCPU's real and virtual modes are emulated by the CPU's virtual mode, a conflict between the two configurations could occur such that more than one TLB entry would be matched for the same effective address.
- Similarly, knowing that the VCPU's real and virtual modes are emulated by the CPU's virtual mode, the CPU's virtual mode would have to be dynamically managed between the two modes.

A throughout study has not been performed to evaluate the true extent of the feasibility of

independent control over instruction and data address translation. We have considered the reasons above enough to stop us from doing so, and postpone that for future work. Furthermore, use of this feature is rare; during this work, we have found that only Linux used this feature, in a routine to flush the cache during initialization; we have easily modified that routine to work around this limitation.

2.4.3.3. Real Mode Translation

The translation between the VCPU's real mode configuration and the CPU's virtual mode configuration is illustrated in Figure 2.19. In Table 2.8 and Table 2.9, the translation between the VCPU's real mode configuration and the CPU's virtual mode configuration is described. Table 2.8 shows the translation between the VCPU's real mode configuration and the CPU's TLB (i.e., realTLB), and Table 2.9 shows the translation between the VCPU's real mode configuration and the CPU's PID and ZPR (i.e., realPID and realZPR). Translation is performed only once, whenever the VCPU's real mode configuration is changed, and it is saved in the VCPU's register file: “realTLB” corresponds to the CPU's TLB, “realPID” to the CPU's PID and “realZPR” to the CPU's ZPR after translation.

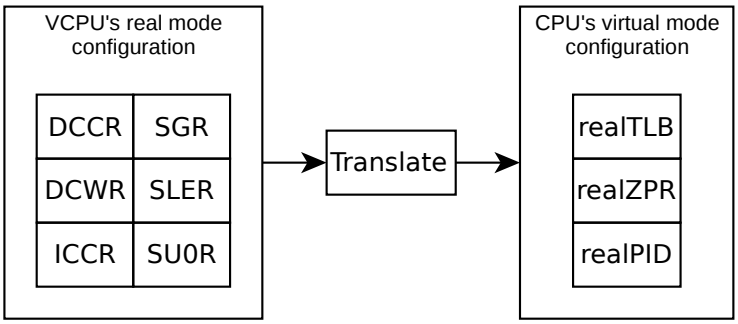


Figure 2.19. Translation between the VCPU's real mode configuration and the CPU's virtual mode configuration.

Table 2.8. Translation between the VCPU's real mode configuration and the CPU's TLB (i.e., realTLB).

realTLB entry bit field	**Setting**
EPN	EPN is set during initialization and never changes; it is set to the guest real/physical start address of the "i^{th}" page of the guest-physical address space. EPN, RPN, and SIZE, together, define the guest-physical address space and map it to the hypervisor-physical address space.
SIZE	The SIZE field, similarly to EPN and RPN, is set during initialization and never changes; it is set to the size of the "i^{th}" page of the guest-physical address space.
V	All of the entries required to map the guest-physical address space are valid (i.e., V = 1); all other entries are invalid (i.e., V = 0). The "realTLB" can have up to 64 entries; however, in most cases, only a few entries are need.
E	E = bit[a](EPN / 2^{27}, SLER). Copy the bit in SLER which has an effect on the guest-physical address space defined by EPN, SIZE and RPN.
U0	U0 = bit[a](EPN / 2^{27}, SU0R). Copy the bit in SU0R which has an effect on the guest-physical address space defined by EPN, SIZE and RPN.
TID	TID = 0. Always disable PID-TID comparison; it is not required.
RPN	RPN is set during initialization and never changes; it is set to the hypervisor-physical start address of the "i^{th}" page of the guest-physical address space.
ZSEL	ZSEL is always set to select the Zone 0 in ZPR, which as shown in Table 2.9, is set to override the write-enabled (WR) and execute-enable (EX) fields and always grant write and execute permission.
EX	Overridden by ZSEL and ZPR.
WR	Overridden by ZSEL and ZPR.
W	W = bit[a](EPN / 2^{27}, DCWR). Copy the bit in DCWR which has an effect on the guest-physical address space defined by EPN, SIZE and RPN.
I	Refer to Table 2.10.
G	G = 1. Always disable speculative memory accesses.

[a]bit: a function which extracts the "n^{th}" bit, as specified by the first argument, from the second one (big-endian notation).

Table 2.9. Translation between the VCPU's real mode configuration and the CPU's PID and ZPR (i.e., realPID and realZPR).

SPR	Setting
realPID	Ignored; PID-TID comparison is always disabled.
realZPR	Zone 0 = b'11': execute and write permissions granted. Zones 1 through 15: not used, ignored.

Table 2.10. Translation between the VCPU's ICCR and DCCR, and the I bit field in the CPU's TLB.

ICCR	DCCR	I	Remarks
0	0	1	Data and instruction cacheability disabled as expected.
0	1	0	Instruction cacheability unexpectedly enabled.
1	0	1	Instruction cacheability unexpectedly disabled.
1	1	0	Data and instruction cacheability enabled as expected.

As explained in *Section 2.4.3.1. PowerPC 405 Memory Management Unit*, real mode storage attributes are controlled by 32-bit special-purpose register (SPR) where, each bit controls a 128 MB section of the real mode address space. When translating the VCPU's real mode configuration to the CPU's TLB, EPN is used to calculate the corresponding 128 MB section and extract the corresponding bit from the SPR. For SLER/E, SU0R/U0, and DCWR/W, the mapping is direct, as shown in Table 2.8.

However, for the real mode storage attribute controlled by SGR no identical virtual mode storage attribute can be found. The closest virtual mode storage attribute is G (guarded); however, in virtual mode, an instruction fetch to guarded storage generates an exceptions, while in real mode, nothing happens. To overcome this difference, storage is never marked as guarded in the “realTLB.” The consequence is that, when guests execute in the VCPU's real mode, the CPU will speculatively fetch instructions (the PowerPC 405 does not perform speculative data accesses). If this is undesired, guests should use other methods to prevent speculative instruction fetches, as described in [22], such as using virtual mode. As future work, we propose the evaluation of alternatives to this translation, which may be acceptable for some guests.

Similarly to SGR, there is no direct mapping between DCCR and ICCR, and the “realTLB.” While DCCR and ICCR provide independent control over data and instruction cacheability, respectively, the I bit field, on the other end, either inhibits data and instruction cacheability altogether, or not. To overcome this problem the mapping between DCCR and ICCR, and the I field in the “realTLB” is performed as shown in Table 2.10. It can be summarized as follows: if DCCR is set to enable cacheability, then enable cacheability; otherwise, disable it; ICCR is basically ignored. This particular mapping has been chosen because after a careful analysis of the instruction set, it has been verified that the “dcba” and “dcbz” user instructions behave differently when data cacheability is enabled and when data cacheability is disabled. More specifically, when cacheability is enabled, “dcba” is able to indirectly modify the contents of main memory, and when cacheability is disabled, the execution of “dcbz” always causes an exception. Therefore, if the CPU's data cacheability configuration did not reflect the VCPU's configuration, main memory could be inadvertently modified or an exception could be inadvertently generated. The same cannot be said for the CPU's instruction cacheability configuration, which does not affect the behavior of any non-privileged instructions. This mapping is such that:

- When the VCPU configuration enables instruction cacheability but disables data cacheability, the I bit field will be set to disable both data and instruction cacheability, resulting in lower instruction access performance than what would be normally expected.
- When the VCPU configuration disables instruction cacheability but enables data cacheability, the I bit field will be set so that that data and instruction cacheability are both enabled. This mapping has the following two issues. First, if some memory address is modified and then flushed by the data cache, that change will not be immediately visible to the instruction cache as it would if instruction cacheability was disabled. To overcome this, guests need only to append an “icbi” and an “isync” instruction to the code which may modify memory containing executable code. Second, the instruction cache will become "polluted" and the processor may access parts of the memory it would not normally access (e.g., to fill an instruction cache line the PowerPC 405 may read up to 8 instruction). To overcome this: (1) guests should safeguard their code to prevent unexpected accesses to memory, as described in [22]; and (2) guests should not rely on the data that is returned by the “icread” instruction after setting this particular configuration.

Despite the issues with the mapping between DCCR and ICCR, and the I bit field, the problematic conditions occur rarely and can be easily overcome, as described above. As future work, we propose the evaluation of alternatives to this translation, which may be acceptable for some guests.

During the execution of the guest, in real mode, the following interrupts, related to the MMU, can be generated: storage interrupts and TLB-miss interrupts. Data and instruction storage interrupts are always forwarded to the VCPU and subsequently to the guest. A careful analysis of the conditions for data and instruction storage interrupts revealed that, whenever they occur, they need only to be redirected to the VCPU. When a data or instruction TLB-miss interrupt is generated, however, it indicates that the guest tried to access an address outside its dedicated address space; in this case, the VCPU exception handler is called, which may choose to ignore the error, generate a machine check interrupt, among other things. In *Section 2.5. I/O virtualization*, it is described how the VCPU exception handler is used for I/O virtualization.

2.4.3.4. Virtual Mode Translation

The translation between the VCPU's virtual mode configuration and the CPU's virtual mode configuration is illustrated in Figure 2.20. In Table 2.11, Table 2.12, and Table 2.13 the translation between the VCPU's virtual mode configuration and the CPU's virtual mode configuration is described. As shown in Table 2.11, most of the fields of a VCPU's TLB entry do not require translation. The only exceptions are the V and RPN bit fields because of the translation required between the guest-physical and the hypervisor-physical address spaces.

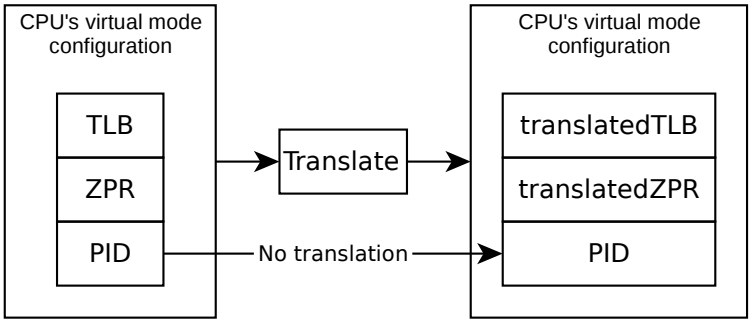


Figure 2.20. Translation between the VCPU's virtual mode configuration and the CPU's virtual mode configuration.

Table 2.11. Translation of the VCPU's TLB to the CPU's TLB (i.e., the translatedTLB).

translatedTLB entry bit field	**Translation**
EPN	No translation.
SIZE	No translation.
V	Marked as valid (i.e., V is set to 1), only if RPN, together with SIZE, are legal, and the original V is set to valid; otherwise, it is marked as invalid (i.e., V is set to 0).
TID	No translation.
RPN	Translated to: (RPN – G) + H, where G is the start address of the guest-physical address space containing RPN, and H is the start address of the corresponding hypervisor-physical address space. The RPN specified by a guest is always guest-physical; therefore, RPN must be translated from the guest-physical address space to the hypervisor-physical address space. The VCPU must find the corresponding hypervisor-physical address space which has been mapped to the guest-physical address space containing RPN. If any part of the address space defined by RPN and SIZE cannot be found in the guest-physical address space, the TLB entry is considered illegal and is marked as invalid.
ZSEL	No translation.
EX	No translation.
WR	No translation.
W	No translation.
I	No translation.
M	No translation.
G	No translation.
U0	No translation.

In Table 2.12 only ZPR requires translation. The ZPR is interpreted differently when the CPU is in

privileged mode and when the CPU is in user mode. Guests always run in user mode and, to emulate the effect of the ZPR in privileged mode, the mapping shown in Table 2.13, is performed when the VCPU is in privileged mode: the CPU's privileged mode behavior is mapped to the CPU's user mode behavior.

Table 2.12. Translation of the VCPU's PID and ZPR to the CPU's PID and ZPR.

SPR	**Translation**
PID	No translation.
ZPR	When the VCPU is in privileged mode, translation is according to Table 2.13. When the VCPU is in user mode, there is no translation.

Table 2.13. Translation of the VCPU's ZN (ZPR).

VCPU's ZN	**Translation**
b'00'	b'01' or b'10'
b'01'	b'01' or b'10'
b'10'	b'11'
b'11'	b'11'
ZN	*ZN | b'01'*

The VCPU's TLB and the translatedTLB as well as the VCPU's ZPR and translatedZPR are all saved in the VCPU's register file in order to: (1) retain the original VCPU's virtual mode configuration, and (2) so that translation is performed only once. An alternative would be to maintain only one of them and to perform translation between the two whenever necessary; even though this would lead to lower memory footprint, it would also lead to higher virtualization overhead.

As in real mode, during the execution of the guest, in virtual mode, the following interrupts related to the MMU, can be generated: storage interrupts and TLB-miss interrupts. Data and instruction storage interrupts are always forwarded to the VCPU and subsequently to the guest. A careful analysis of the conditions for data and instruction storage interrupts revealed that, whenever they

occur, they need only to be redirected to the VCPU. Data and instruction TLB-miss interrupts, on the other end, can occur under two conditions. One: the VCPU's TLB and the VCPU's translatedTLB do not contain a matching TLB entry. When this is true, the data or instruction TLB-miss interrupt is forwarded to the VCPU. Two: the VCPU's TLB contains a matching TLB entry, but the translatedTLB does not contain a matching TLB entry. This happens when a entry in the VCPU's TLB maps an address space outside the VCPU's address space, and thus, the VCPU "refuses" to translate such an entry. In bare metal, this is equivalent to a memory access to an address that is not mapped to memory or I/O. In bare metal, however, such an access, can have the following outcomes: (a) a data or instruction storage interrupt, if the TLB entry is configured to deny data or instruction memory access; (b) a machine check interrupt, if none of the conditions for a storage interrupt are met, and if there was an external bus error while performing the memory access; or (c) nothing, if the conditions for (a) and (b) are not met. Therefore, whenever a data or instruction TLB-miss is generated, and it is true that the guest tried to perform an access outside the VCPU's dedicated address, then, if the conditions for a storage interrupt are met, a corresponding storage interrupt is requested; otherwise, the VCPU exception handler is called, which may choose to ignore the error, generate a machine check interrupt, among other things. In *Section 2.5. I/O virtualization*, it is described how the VCPU exception handler is used for I/O virtualization.

2.4.4. Interrupts

Whenever the VCPU detects a (virtual) interrupt condition, it performs an interrupt request. An interrupt request can occur while a VCPU is being loaded or in any of the VCPU interrupt-handlers, as illustrated in Figure 2.21. An interrupt request must not occur while the VCPU is being resumed, paused, or saved. A higher priority interrupt request replaces a lower priority interrupt request. An interrupt request by itself does not mean that that interrupt will be actually generated. Only when the VCPU is being resumed does an interrupt request (i.e., the highest priority interrupt request) actually gets generated, during interrupt dispatch, as illustrated in Figure 2.21. This particular approach is necessary because several interrupt conditions can be reached simultaneously, but only the highest priority interrupt among them can actually be generated.

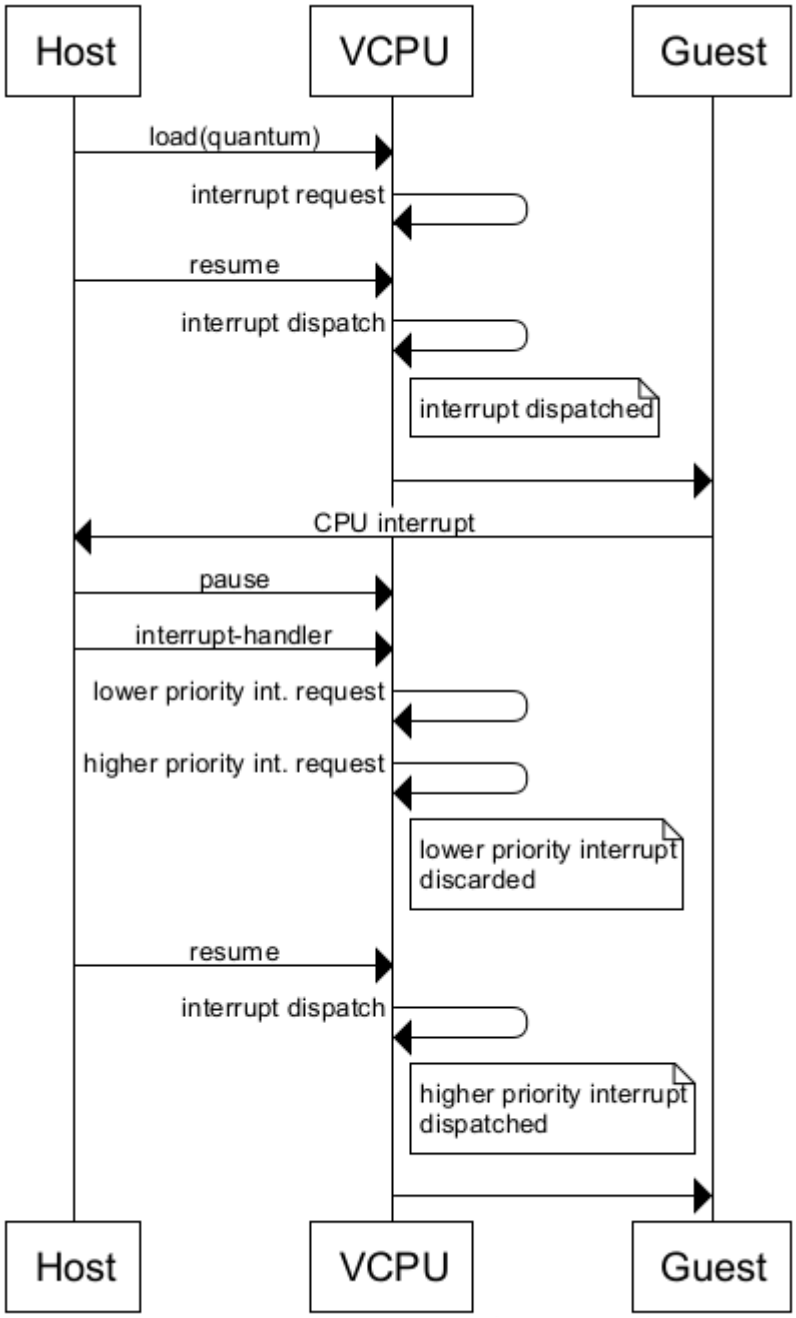


Figure 2.21. VCPU interrupt requests and dispatching.

2.5. I/O virtualization

In the PowerPC 405, access to I/O devices can be performed on two different buses: the memory bus and the Device Control Register (DCR) bus. Access on the memory bus is performed through data load and data store instruction for reading or writing data, respectively. A data load/store instruction specifies a 32-bit address and data (byte-wise, 2-byte-wise, or 4-byte-wise). Similarly,

the DCR bus is accessed through the Move To/From Device Control Register privileged instructions (MTDCR and MFDCR), which specify a 10-bit address and data (only 32-bit data is supported, but the receiving end can ignore unneeded bits).

In terms of memory-mapped devices the VCPU provides two access mechanisms: bypass and supervised. In bypass, the address space of the memory-mapped device is part of the VCPU address space, as described in *Section 2.4.3. Memory Management Unit*, and thus, the VCPU has no control over the access to the device. It is possible to map the same device in different VCPU; however, the VCPUs are responsible for coordinating access using, for example, shared memory or paravirtualization, explained in *Section 2.6. Paravirtualization*. By mapping the same memory device, or part thereof, in different VCPUs, it is possible for VCPUs to share memory.

The second mechanism available for memory-mapped devices is supervised. With supervised memory-mapped I/O, the address space of the device is not part of the VCPU address-space, and thus, as explained in *Section 2.4.3. Memory Management Unit*, whenever the guests tries to access the device, a data TLB-miss interrupt will always be generated. The VCPU, in the data TLB-miss interrupt handler upon discovering that the guest tried to access an address outside the VCPU address space, calls the exception handler (EH) to handle the “error,” as illustrated in Figure 2.22. The EH, in turn, may choose to emulate the access on a real or virtual device. If the EH is unable to handle the access, it may choose to ignore it or to request an interrupt. With supervised memory-mapped I/O it is possible to share a single device among more than one VCPU in a controlled way. Each VCPU can have its own dedicated EH; therefore, different VCPU can be assigned with different memory-mapped devices.

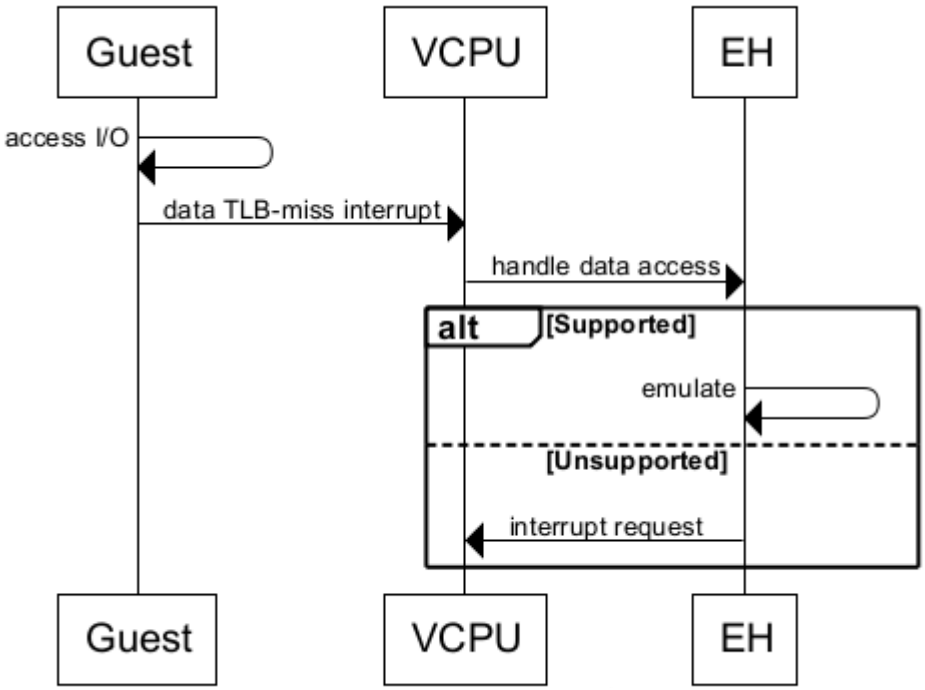


Figure 2.22. Supervised memory-mapped I/O.

In terms of DCR-mapped devices, there is only one mechanism available: supervised DCR-mapped I/O. DCR-mapped devices are only accessible through privileged instruction which, as explained in *Section 2.4.1. Instruction Set*, will always lead to the generation of a program interrupt. Similarly to supervised memory-mapped I/O, in the MTDCR and MFDCR emulation routines, the VCPU calls the EH which, in turn, may choose to emulate the access on a real or virtual device, as illustrated in Figure 2.23. If the EH is unable to handle the access, it may ignore the error, generate an interrupt, etc. Each VCPU can have its own dedicated EH, and thus, each VCPU can be assigned with a distinct set of DCR-mapped devices.

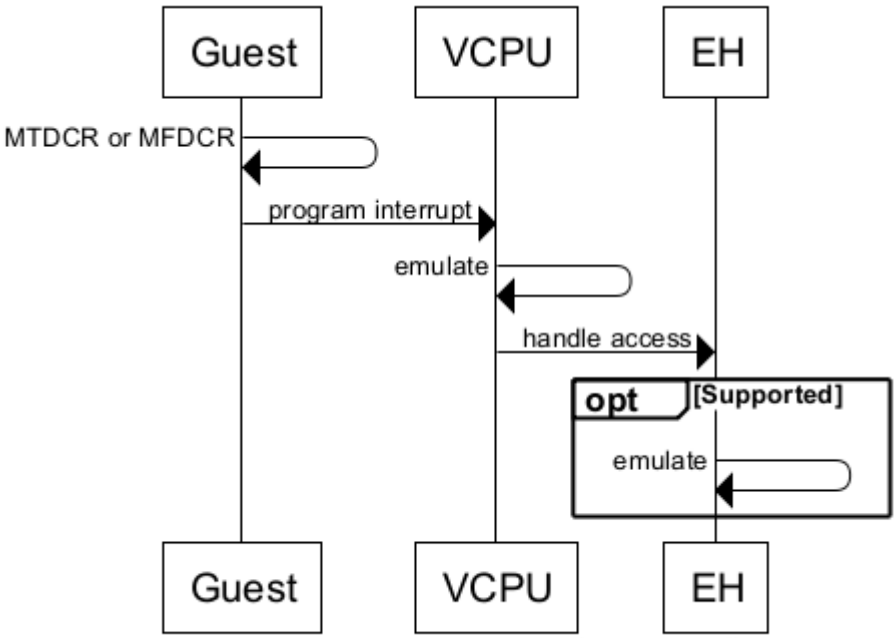


Figure 2.23. Supervised DCR-mapped I/O.

2.6. Paravirtualization

The VCPU provides three mechanisms dedicated to paravirtualization: (1) hypercall detection based on the system call interrupt, (2) hypercall detection based on access to supervised memory; and (3) hypercall detection based on access to supervised DCR. These three mechanism enable the detection of a “hypercall condition” and, subsequently, the execution of a corresponding handler.

When “hypercall detection based on the system call interrupt” is enabled, whenever the guest executes a System Call instruction (SC), generating, in turn, a system call interrupt, and the VCPU is in privileged mode, the exception handler (EH) is called. The EH, then, can decode the hypercall and call the respective handler. This is illustrated in Figure 2.24.

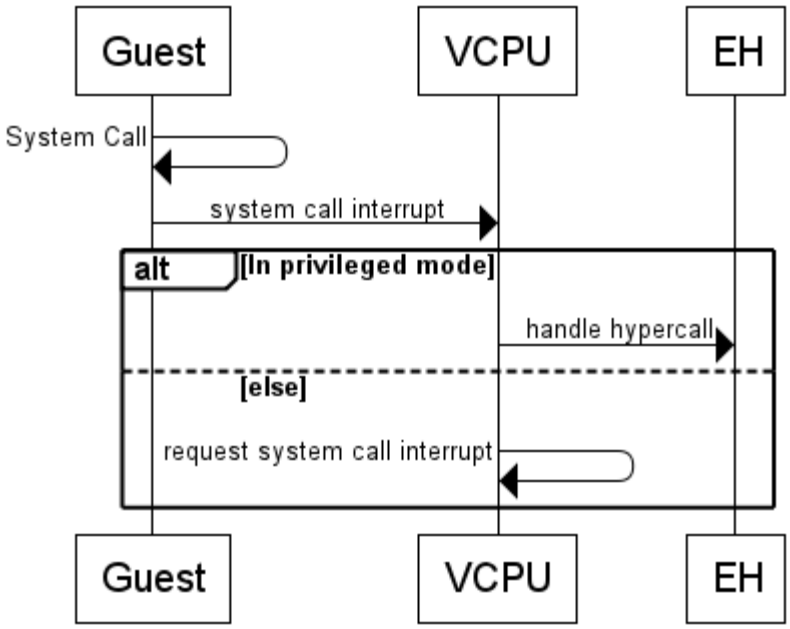


Figure 2.24. Sequence diagram illustrating hypercall detection based on the system call interrupt.

The two other mechanisms are realized as explained in *Section 2.5. I/O virtualization*. Whenever there is an access to supervised memory or to supervised DCR, the EH, depending on a predefined memory address or on a predefined DCR signaling a hypercall, can decode the hypercall and call the respective handler.

The “hypercall detection based on the system call interrupt” may not be suitable if the guest relies on the CPU's original behavior (i.e., the generation of a system call interrupt). The two other mechanisms, on the other end, resemble I/O devices and are always “safe” to use. The overhead of “hypercall detection based on the system call interrupt,” however, is lower than the other two, as will be shown in *Section 3.5.3. Hypervisor's Performance Profile*. Finally, the three mechanisms are not mutually exclusive and, if required, can be simultaneously enabled.

2.7. Future Work

In the context of this section, we propose the following for future work:

- Currently, the minimum size of a VCPU address space is 16 MB. This limitation however, is only a limitation of the implementation and not of the hardware, and therefore, we propose the removal of this limitation.
- Virtualization of some parts of the CPU are still unsupported (e.g., Watchdog), and thus, we

propose the development of the necessary methods.

- Evaluate different translations between the VCPU's MMU configurations and the CPU's virtual mode configuration, which may lead to improved performance when acceptable for some guests which do not use all of the features provided by the CPU.
- Find new method to fix or work around the limitations of the PowerPC 405 for full virtualization (e.g., limitations of the MMU).
- Move the virtualization of the Programmable-Interval Timer outside the VCPU and onto the host; this would remove the need to stop the host's scheduler while a VCPU is scheduled.
- Further decomposition of the implementation and improvement of its configurability, enabling, for example, the VCPU to include only the functionality actually required by the guest (i.e., partial virtualization), leading to lower virtualization overhead and smaller footprint.
- Apply the same methodologies to other processor architectures in an attempt to find out the most common and most efficient design patterns, so that these can be optimized and easily reused in other processor architectures.
- Study the feasibility and the impact of the certification of Rodosvisor using, for example, formal methods.
- Based on the PowerPC 405 limitations concerning full virtualization, extend Popek's et al. requirements for virtualization [20] to accommodate the PowerPC 405 and similar processor architectures. This would facilitate the development of future processor architectures with support for low-end hardware full virtualization.

2.8. Summary

In this chapter, background has been given on hypervisors and virtualization and, it has been demonstrated that, nowadays, most hypervisors either do not provide complete compatibility with legacy software or lead to large system size, weight, power consumption, and cost. It has been shown how to accomplish low-end hardware full virtualization in general, and of the PowerPC 405 in particular. It has been explained how to virtualize the instruction set using a trap-and-emulate mechanism, and how to virtualize the time and the timer units. It has been described how the memory management unit is virtualized: by mapping the VCPU's real and virtual modes onto the CPU's virtual mode. It has also been explained how interrupts are virtualized. It has been shown that

full virtualization of the PowerPC 405, nevertheless, is not completely possible; this can be used to extend Popek's et al. requirements for virtualization [20] and to improve future processor architectures in supporting low-end hardware full virtualization. Finally, in this chapter, the various mechanisms available for I/O virtualization and for paravirtualization have been described.

In this chapter, it has been shown that low-end hardware full virtualization is heavily dependent on the target architecture. We believe that porting this implementation to a target architecture in the same family should not be difficult. However, porting this implementation to a different architecture is not feasible nor recommended. We believe it is better to start from scratch. Still, some of the design patterns used in this implementation can be reused or taken as a reference.

3. POK/rodosvisor

3.1. Introduction

In this chapter, first, background on ARINC 653 [14] and POK [23] is provided. POK is an ARINC 653 separation kernel (and real-time operating system) for safety-critical embedded system. POK is configurable and enables, among other things, the realization of ARINC-653-compliant systems. POK relies on a preprocessor-based compile-time variability management system and on static allocation of resources to configure and customize the kernel according to application-specific requirements. In the final kernel image it is included only the functionality that is actually necessary (i.e., unused functionality is trimmed out).

POK, however, did not support the PowerPC 405; it did, nevertheless, support the PowerPC 440. POK, therefore, has been extended to support the PowerPC 405 using the PowerPC 440 hardware abstraction layer as a starting point.

POK has been chosen because, to best of our knowledge, it is the only ARINC-653-compliant solution that is freely available, and because, together with Ocarina [34]–[37] and AADL [38], it enables a model-driven engineering environment. Model-driven engineering is the subject of *Chapter 4. Model-Driven Engineering using Ocarina*.

In this chapter, it is also described how support for privileged partitions has been added to POK. Privileged partitions are a requirement of functionality farming for enforcing time-only partitioning while retaining compatibility with legacy software. Functionality farming is the subject of *Chapter 5. Functionality Farming*. In summary, adding support for privileged partitions required a new partition type, a different compilation and initialization process, and a new set of configuration options. The impact on the size of the trusted computing base and memory footprint by privileged partitions is also presented in this chapter.

In this chapter, it is also explained the integration of Rodosvisor's virtual processors (VCPU), and thus, virtual machines, described in the previous chapter, with POK. As explained in the previous chapter, Rodosvisor provides “cycle-accurate” virtual machines, which are compatible with ARINC 653 system partitions. The reason for the integration of Rodosvisor with POK is, as explained in the previous chapter, that Rodosvisor by itself is unable to support a running system and thus, needs to be integrated with a host, which, in this case is POK. In summary, POK/rodosvisor's integration required the following:

- a new partition type, and more specifically, a new thread type which takes care of the

interface with host, described in *Section 2.3. Interface with the Host*;

- modified interrupt handling which either redirects interrupts to the current VCPU or to POK's default interrupt handlers;
- a new set of configuration options which provide control over the features of Rodosvisor's VCPUs.

The impact of virtual machines on the trusted computing base and memory footprint, the cumulative virtualization overhead and POK/rodosvisor's performance profile are presented in this chapter as well. Virtual machines are another requirement of functionality farming: they enforce time and space partitioning while retaining compatibility with legacy software.

3.1.1. Chapter Organization

This chapter is organized as follows.

In the next section, *Section 3.2. ARINC 653*, the ARINC 653 standard and how it is realized in POK is described.

In *Section 3.3. Privileged Partitions*, it is described how support for privileged partitions was added to POK.

Next, in *Section 3.4. POK and Rodosvisor Integration*, the integration of Rodosvisor with POK is explained.

Then, in *Section 3.5. Evaluation*, the impact on the trusted computing base and memory footprint by privileged partitions as well as by virtual machines, the cumulative virtualization overhead and POK/rodosvisor's performance profile are presented and discussed.

In *Section 3.6. Future Work*, future work is proposed.

Finally, in *Section 3.7. Summary*, a summary of the major findings and contributions in this chapter is given.

3.2. ARINC 653

ARINC 653 [14] belongs to a group of standards (e.g., ARINC 650, ARINC 651) which attempt to standardize and regulate the use of the integrated modular avionics architecture [11], thus improving interoperability between different vendors and suppliers, and facilitating the integration of components of-the-shelf.

ARINC 653 defines an application programming model by defining the services that must be

available to partitions, namely, a hierarchical, two-tier scheduling policy and the APplication EXecutive (APEX). The software architecture specified by ARINC 653 is illustrated in Figure 3.1. The separation kernel sits on top of the hardware alongside system/application-specific functions and has full control over the hardware. The separation kernel is responsible for enforcing time and space partitioning and for managing the execution of the partitions. ARINC 653 distinguishes two types of partitions: application partitions and system partitions. Application partitions interact only with the separation kernel through the APEX; application partitions may interact with other partitions indirectly, through the services provided by the APEX. System partitions, on the other end, are application-specific and the standard does not specify upon them.

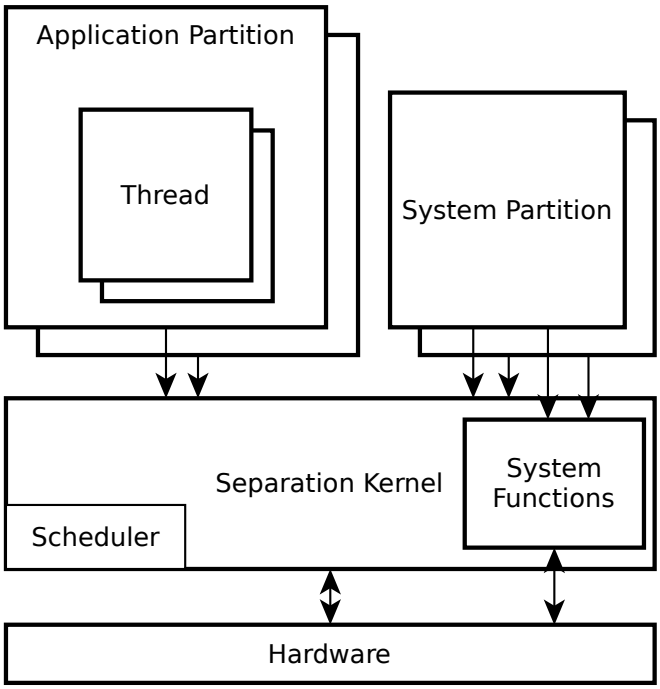


Figure 3.1. ARINC 653 software architecture.

ARINC 653 defines an hierarchical scheduling scheme with two schedulable entities: partitions at the top, and threads at the bottom-level. (In the standard “threads” are actually referred to as “processes;” POK, however, uses “threads” and, in this work, it has been chosen to use POK's nomenclature as it accurately represents POK's implementation.) Partitions by themselves are not executable, their threads are. Partitions are scheduled according to a statically defined fixed-cycle scheduling scheme, thus enforcing the necessary time partitioning. Partition scheduling is accomplished by dividing the time into major frames, which are subsequently divided into minor frames or partition windows, as illustrated in Figure 3.2. Each partition is then assigned to one or

more partition windows in the major frame. The duration of a major frame and the duration of all partition windows are defined at compile-time. If, at a given time, a partition is not doing anything (i.e., it is idle), its partition window is still not reused by other partitions.

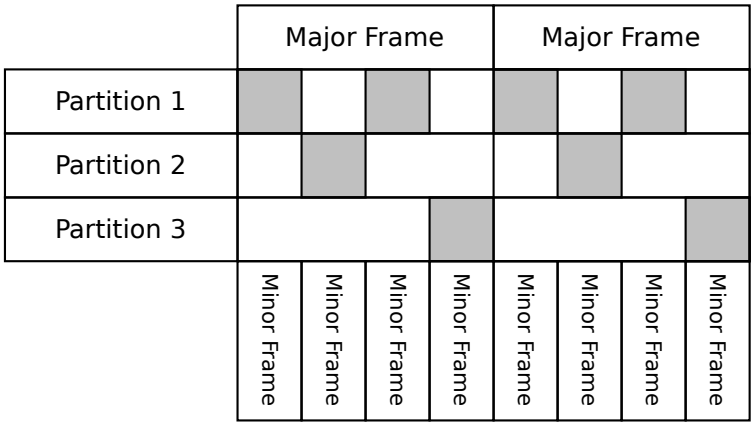


Figure 3.2. ARINC 653 major and minor frames.

After partition scheduling, the second level of scheduling follows, that is, thread scheduling. Thread scheduling is performed only among the threads of the currently scheduled partition, and it depends on the partition's current operating mode. The first time a partition is scheduled its operating mode is COLD_START, illustrated in Figure 3.3. In COLD_START only the main thread, or the error thread in case of error, can execute and the thread scheduler is actually inhibited. The main thread is responsible for, and is the only thread allowed to: initialize the partition, create other threads, create communication ports, etc. When the main thread successfully completes its task it requests the operating mode of the partition to be changed to NORMAL, also illustrated in Figure 3.3, and consequently, activating the thread scheduler; the thread scheduler, however, is not activated immediately, but only at the next partition window, and until then, the partition will be idle. In the NORMAL operating mode, ARINC 653 specifies a priority preemptive scheduling policy between threads. In the IDLE operating mode, thread execution is completely inhibited, including the main and error threads, and the partition is considered stopped.

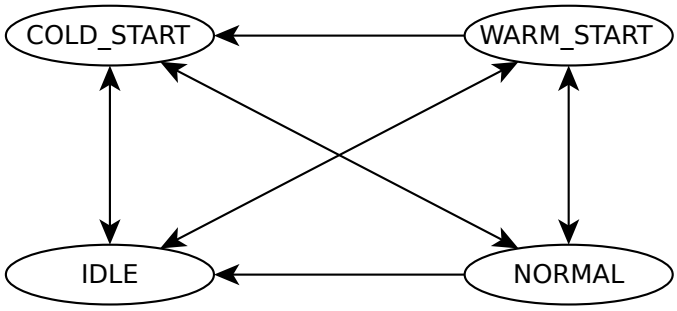


Figure 3.3. ARINC 653 partition's operating modes.

When an error is detected, thread scheduling is inhibited, if not already, and the partition's error thread is scheduled to handle the error. Error types include: configuration errors, invalid requests, illegal memory access, illegal instructions, hardware errors, errors explicitly raised by the partition, etc. If the error thread has not been created, or if the error thread is unable to handle the error, or if an error occurs in the error thread itself, a dedicated partition error handler in the separation kernel is called to handle the error. An error handler (i.e., an error thread or a partition error handler) may request the operating mode of a partition to change to WARM_START, COLD_START, or IDLE, as illustrated in Figure 3.3. The COLD_START and WARM_START operating modes are very similar; nevertheless, in COLD_START the state of the partition is completely reset, while in WARM_START, part of the previous state of the partition may have been preserved; therefore, a COLD_START must always precede a WARM_START.

Apart from an hierarchical scheduling scheme, ARINC 653 also defines the APEX (APplication EXecutive), a programming interface for:

- partition managements: get partition status, change the partition's operating mode;
- thread management: create a thread, get thread status, start, suspend, resume, stop a thread;
- time management: get current time, suspend a thread for a specified amount of time;
- inter-partition communication: sampling (unbuffered) and queuing (buffered) ports, create port, send and receive data, get port status;
- intra-partition communication: blackboard (unbuffered) and buffer'ed ports, semaphores and events for synchronization and mutual exclusion;
- health monitoring: get error status, create error handler, raise an error.

There are two ways in which the APEX is commonly implemented. As illustrated in Figure 3.4(a), the APEX can be directly provided by the separation kernel, or, as illustrated in Figure 3.4(b), the APEX can be implemented on top of the separation kernel's middleware. POK uses the latter approach, and provides *libpok* which based on POK's middleware implements the APEX.

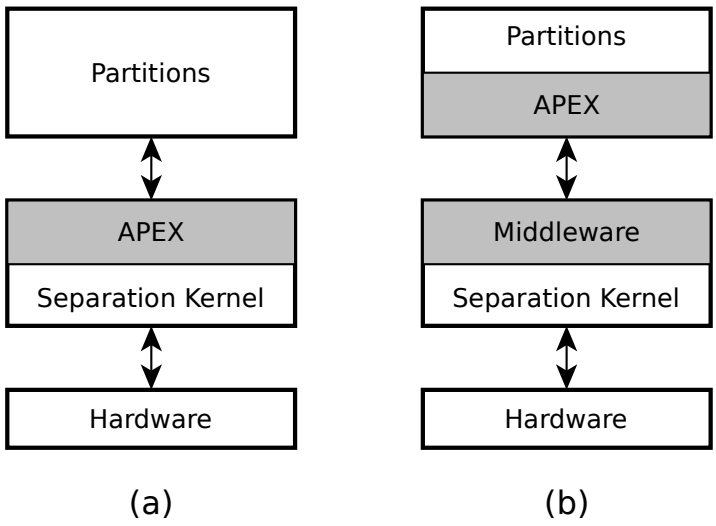


Figure 3.4. Implementation of the APEX: (a) implemented in the separation kernel, (b) implemented by partitions based on the separation kernel's middleware.

ARINC 653 also specifies that all resources (e.g., partitions, threads, communication ports) must be statically allocated during compile-time and cannot change at run-time; `if` a partition requests a resource that has not been specified during compile-time, that request must fail. This improves predictability, prevents the system from running out of resources unexpectedly, and facilitates certification.

There are only a few known implementations of the ARINC 653 standard. Most of them are commercial or access to the source code and its underlying design is not possible (e.g., [17], [67]–[71]).

Simulated Integrated Modular Avionics (SIMA) [72]–[74] is a particular deployment of ARINC 653 dedicated to simulation. It is based on Linux 2.6 in which partitions correspond to POSIX processes and threads correspond to POSIX threads. The APEX is provided as a static library, named Partition Operating System (POS), which is statically linked with every partition. The ARINC 653 thread scheduler is implemented on top of the POSIX FIFO scheduler by POS. The Module Operating System (MOS), also a POSIX process, controls all the other partitions: the ARINC 653 partition

scheduler and the health monitoring services are implemented by the MOS. The MOS communicates with the POS layer through signals and shared memory.

Distributed Integrated Modular Avionics (DIMA, a.k.a., IMA 2G) [75]–[79] is currently being studied has a natural evolution of IMA. Unlike IMA which limits itself to a single computing unit, DIMA encompasses the computing units and the communication between them. DIMA, however, is still a work in progress and many issues need to be solved before it can be deployed in a real-world scenario. One of the most interesting outcomes of DIMA might be the ability of enabling dynamic reconfiguration so as to limit the impact of hardware and software failures on the overall reliability.

Alternatives to ARINC 653, include AUTOSAR [12] and OSEK/VDX [80]; these however, have a different scope and do not provide the same features as ARINC 653, most notably, time and space partitioning.

3.3. Privileged Partitions

Figure 3.5 illustrates the difference between an ARINC 653 partition and a privileged partition. An ARINC 653 partition, Figure 3.5(a), is space partitioned (denoted by a solid line between the ARINC 653 partition and the APEX layers), and thus, only has access to a dedicated address space and to the APEX. A privileged partition, Figure 3.5(b), on the other end, is not space partitioned and has full access to the hardware, including all memory and I/O devices as well as access to the APEX and to POK. A privileged partition executes in the same address space as the kernel and with the same level of privilege. By not enforcing space partitioning, a privileged partition has the following benefits over an ARINC 653 partition:

- With full access to the hardware, and by executing in the same address space as the kernel and with the same level of privilege, a privileged partition is compatible with legacy kernel and user-level software.
- With an ARINC 653 partition, I/O can only be performed directly by the kernel. With a privileged partition, however, those operations can be offloaded from the kernel to a privileged partition, enabling a reduction of the complexity and size of the core kernel (i.e., the kernel, not including privileged partitions), and thus, an improvement of its predictability and lower certification effort. Similarly, other operations requiring kernel-level privileges, such as inter-partition communication and health monitoring, can also be offloaded to a privileged partition, further reducing the complexity and the size of the core kernel.

- A privileged partition facilitates control over kernel-level software, such as establishing application-specific run-time environments in an efficient manner (e.g., OSEK/VDX, virtual machines). A privileged partition therefore, also improves variability management.

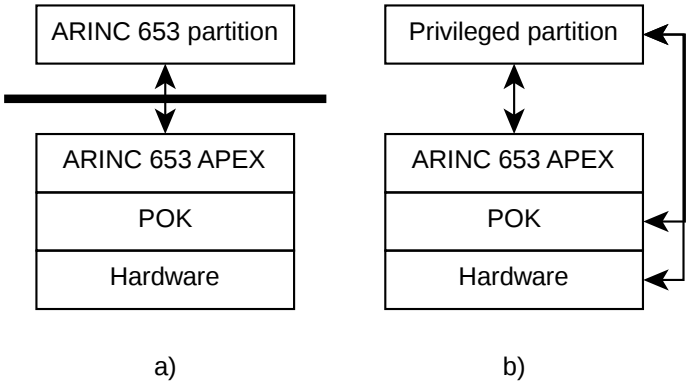


Figure 3.5. Comparison between (a) ARINC 653 partitions and (b) privileged partitions.

Without space partitioning, however, a privileged partition is part of the trusted computing base (TCB) and the system is especially sensitive to errors that may occur within a privileged partition. Nonetheless, since a privileged partition does not enforce a protected address space, for any given system only one privileged partition is required, and thus, the impact on the TCB is low and scalable. The modification applied to POK to support privileged partitions are described next.

For every thread (i.e., the basic execution unit of a partition), POK allocates a stack in the kernel address space. This kernel-level stack is used whenever the intervention of the kernel is required (e.g., a system call, a scheduling point). This kernel-level stack is, among other things, initialized with:

- The configuration of the CPU, including the configuration of the memory management unit (MMU), which will be set on the CPU during the execution of the associated thread.
- The thread's entry point (within the partition's address space).
- A pointer to the first stack frame of the thread (within the partition's address space).

For ARINC 653 partitions the configuration of the MMU is set to enforce space partitioning by mapping only the partition's dedicated address space. Conversely, as illustrated in Figure 3.6, for a privileged partition, the MMU configuration is set to not enforce space partitioning. Similarly, the CPU configuration for ARINC 653 partitions is set to deny access to the privileged instruction set,

which can be used, accidentally or maliciously, to violate time and space partitioning. For a privileged partition, however, the CPU configuration is set to grant access to the privileged instruction set, enabling a privileged partition to have full access to the CPU's instruction set and to support software which cannot be supported by ARINC 653 partitions (e.g., hardware-dependent software); at the cost, nevertheless, of a larger TCB.

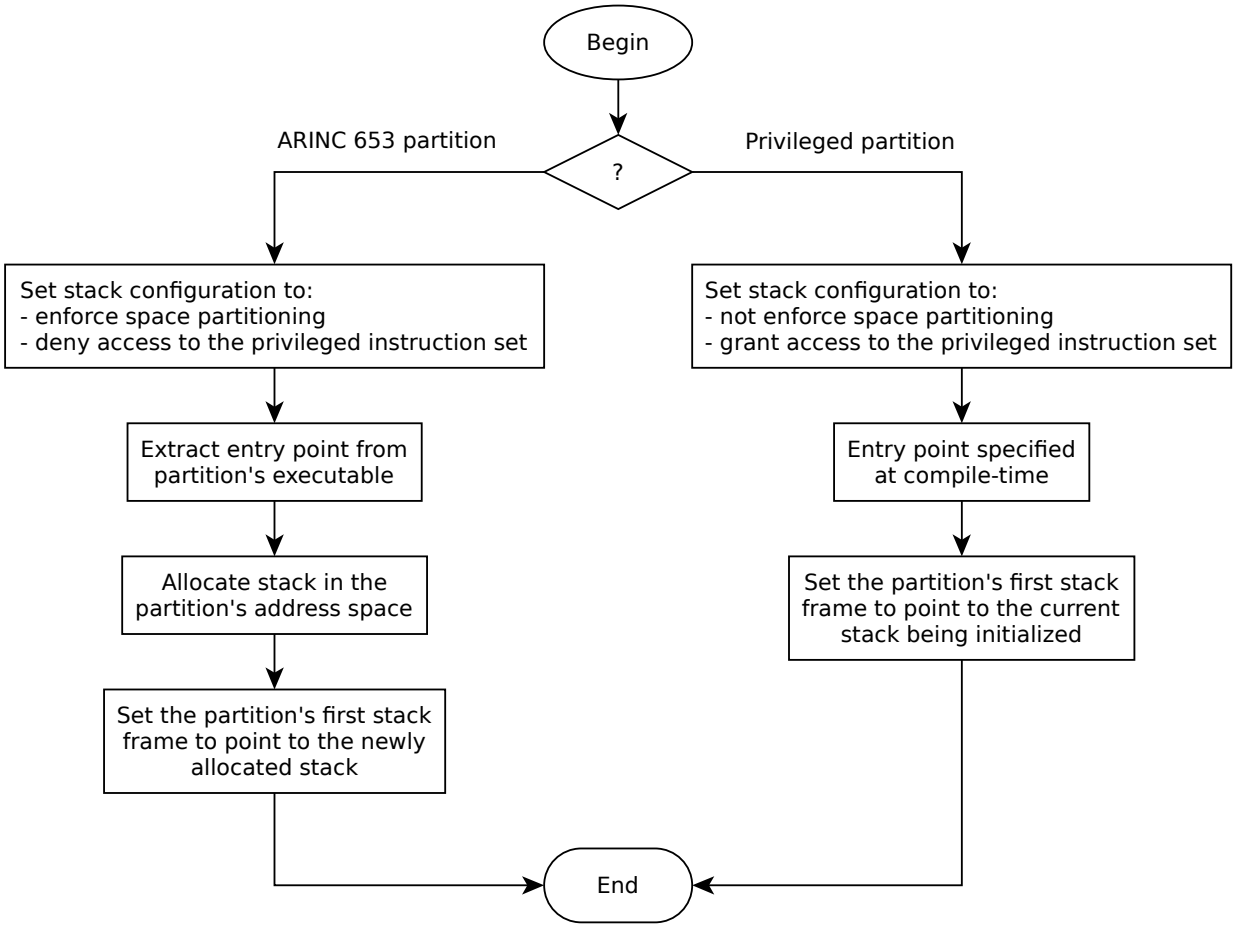


Figure 3.6. Initialization process of a thread's stack for an ARINC 653 or privileged partition.

While the entry point and the first stack frame of an ARINC 653 partition's thread (an ARINC 653 thread for short) are within the partition's dedicated address space, those of a privileged partition are within the kernel address space. The entry point of an ARINC 653 partition is extracted at run-time from the respective executable (partitions' executables are bundled together with the kernel in a dedicated section of the kernel executable), while the entry point of a privileged partition must be specified at compile-time through the associated global function name, as shown in Figure 3.6. Moreover, while an ARINC 653 thread requires two different stacks, one in the kernel address space

and another at the partition's dedicated address space, a privileged partition requires only one stack in the kernel address space. As shown in Figure 3.6, when an ARINC 653 thread is created, another stack is allocated in the partition's dedicated address space and the first stack frame pointer is set to point to this stack, as illustrated in Figure 3.7(a). Conversely, for a privileged thread, the stack in the kernel address space is reused and the first stack frame pointer is set to point to this stack, as illustrated in Figure 3.7(b). A privileged partition, therefore, also benefits from lower memory footprint than ARINC 653 partitions as they do not require a dedicated address space nor a second stack.

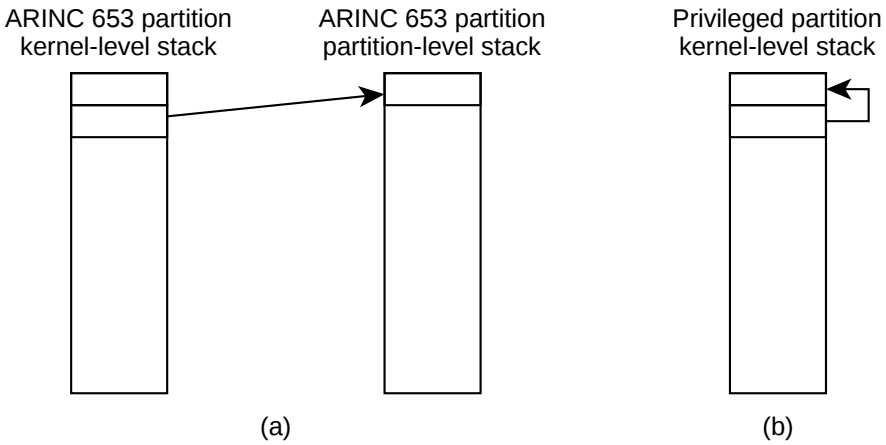


Figure 3.7. Stacks required (a) by an ARINC 653 partition and (b) by a privileged partition.

When an ARINC 653 partition issues a system call (e.g., an APEX service call) through the system call (SC) instruction, as illustrated in Figure 3.8, a CPU interrupt is generated, partially switching the context to the kernel. Then, in POK's system call interrupt-handler:

1. The context switch to the kernel is completed (e.g., switch from the current thread's partition-level stack to the corresponding kernel-level stack).
2. The system call arguments are decoded from partition to kernel-level.
3. The specified system function is called.
4. And, when the system function returns, a context switch back to partition-level is performed, resuming the execution of the partition.

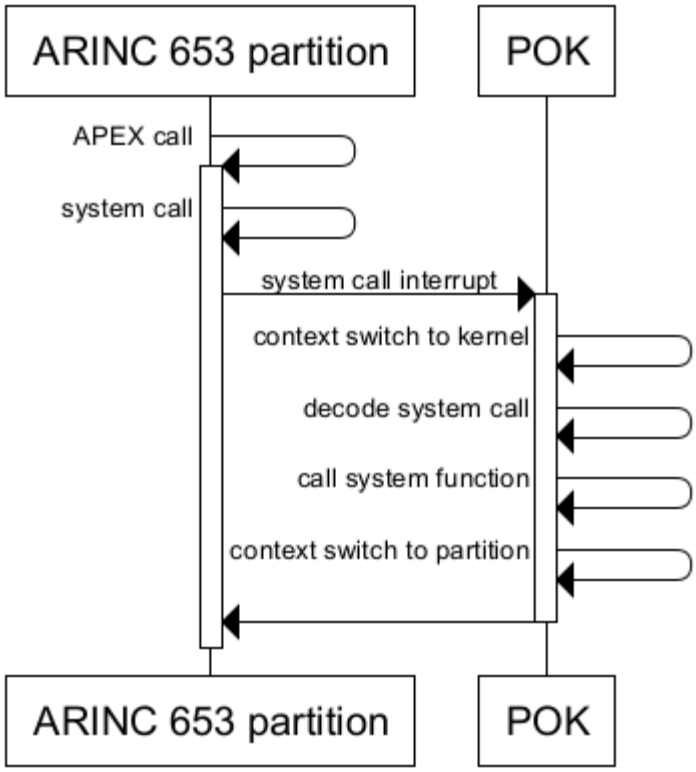


Figure 3.8. An APEX call by an ARINC 653 partition.

A privileged partition, on the other end, runs at kernel-level and has direct access to the system functions. Therefore, a privileged partition does not need to issue a system call and can instead call the system function directly. So, it is not necessary to trigger an expensive context switch and decode the system call. Nevertheless, because system functions do not support preemption, preemption must be disabled prior to calling the system function (delaying, for example, the generation of a timer interrupt), and re-enabled when the system function returns, as illustrated in Figure 3.9. A privileged partition, thus, also benefits from lower system call overhead than ARINC 653 partitions due to a reduction in address space crossings. In other words, where ARINC 653 partitions require an expensive “upcall” and a corresponding, expensive “downcall,” a privileged partition requires none. To make this difference between ARINC 653 and privileged partitions transparent to partition developers, two system call libraries are available: (1) the ARINC 653 partition library which issues system calls by generating a CPU interrupt, as described above, and (2) the privileged partition library which disables preemption, calls the desired system function directly, and re-enables interrupts when the system function returns.

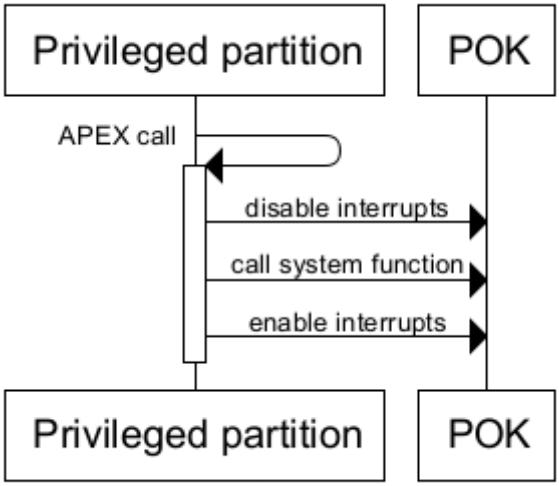


Figure 3.9. An APEX call by a privileged partition.

As illustrated in Figure 3.10, ARINC 653 partitions are compiled and linked independently of the kernel; then, in the link stage of the kernel, ARINC 653 partitions executables are all wrapped together in a special section of the kernel executable; at run-time, during initialization, the kernel sets up each partition's dedicated address space with the respective executable. A privileged partition, on the other end, is compiled, but not linked, independently of the kernel; in the link stage of the kernel, privileged partitions are linked and integrated with the kernel. A privileged partition does not require a dedicated address space, and common code and data can be shared with the kernel and other privileged partitions, such as standard run-time environments and libraries (e.g., C/C++ run-time). As a result, a privileged partition also benefits from faster boot times and lower memory footprint than ARINC 653 partitions.

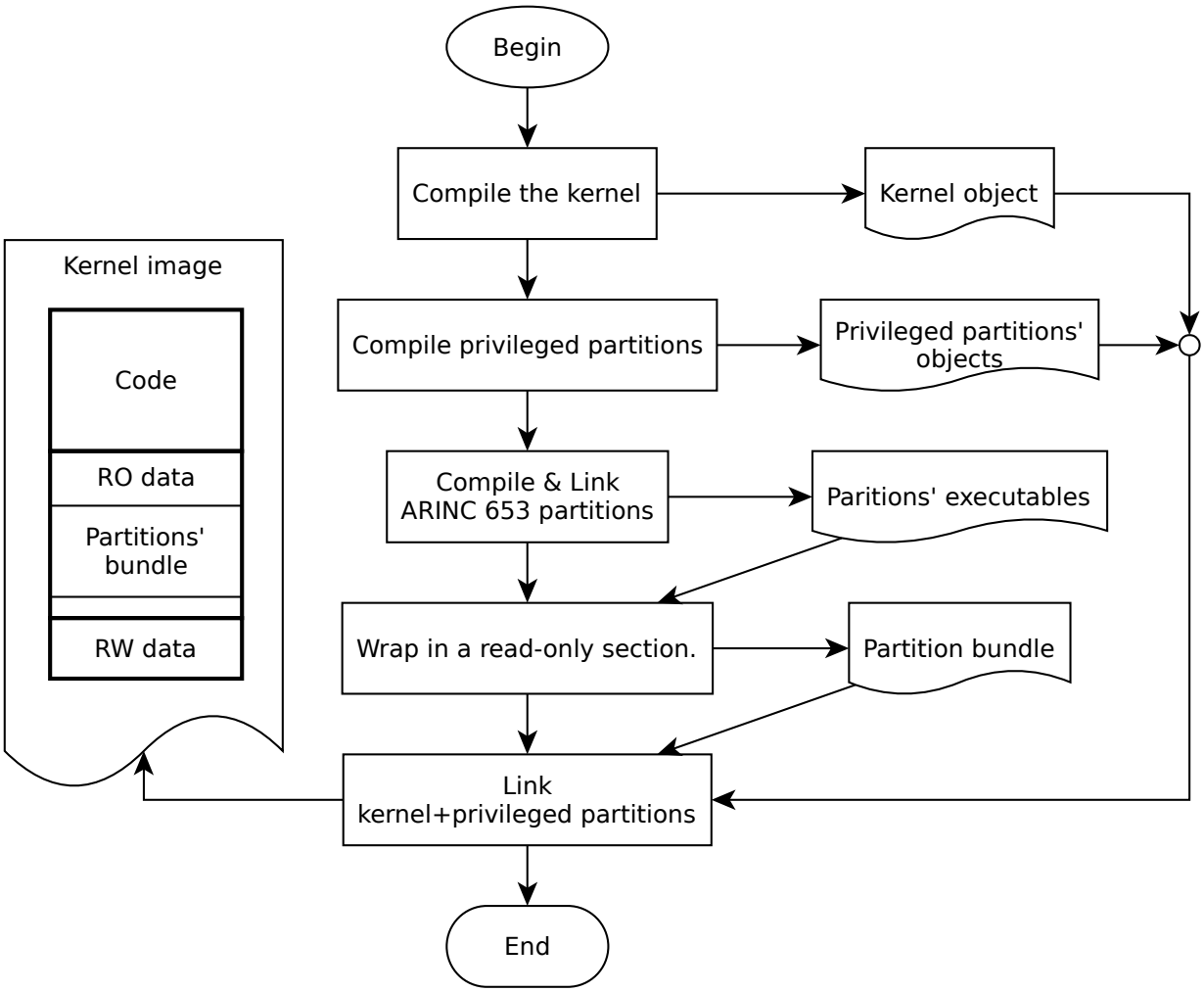


Figure 3.10. POK/rodosvisor compilation process.

As explained earlier, POK implements a preprocessor-based compile-time variability management system which, according to the desired system configuration, compiles a kernel containing only the functionality that is actually needed. Privileged partitions have been included in that system. A new configuration option (i.e., POK_CONFIG_PARTITION_TYPE) enables, if necessary, the specification of which partitions are ARINC 653 partitions and which are privileged partitions. When this option is specified, support for privileged partitions is included in the kernel; conversely, if this option is not specified, support for privileged partitions is not included in the kernel, leading to a smaller footprint.

The number of new and modified source lines of code (SLOC) required in order to add support for privileged partitions to POK is shown in Table 3.1. Table 3.1 also shows an estimation of the

development effort, schedule, number of developers and cost, based on the basic Constructive Cost Model (COCOMO) [81]. Table 3.1 shows that only 52 new/modified SLOC of C were required, with an estimated cost of about $1200.

Table 3.1. The number of new and modified source lines of code (SLOC), and an estimation of the development effort, schedule, number of developers and cost, required in order to add support for privileged partitions to POK.

Metric	Value
Architecture-independent (SLOC), C programming language	26
Architecture-dependent (SLOC), C programming language	26
Total (SLOC)	52
Development Effort Estimate (Person-Months)	0.11
Schedule Estimate (Months)	1.07
Estimated Average Number of Developers (Effort/Schedule)	0.10
Total Estimated Cost to Develop (average salary = $56,286/year, overhead = 2.40)	$1211

3.4. POK and Rodosvisor Integration

For each thread of a “real” partition (i.e., an ARINC 653 or privileged partition), a kernel-level stack is allocated and initialized so that its first stack frame points to the “pok_arch_rfi” function, as illustrated in Figure 3.11(a). When there is a context switch to this stack for the first time, the CPU program counter will be set to point to “pok_arch_rfi” which, in turn, will set up the execution environment of the partition based on how the stack has been initialized, and then resume the execution of the partition.

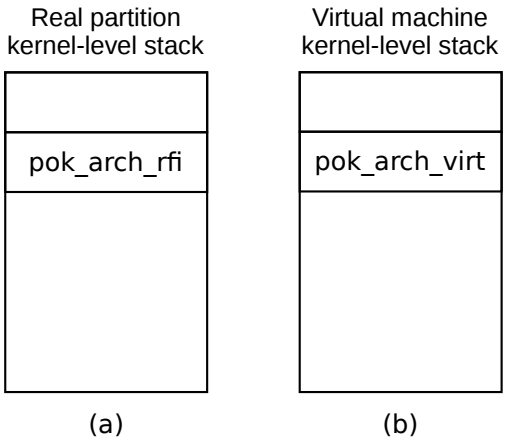


Figure 3.11. Kernel-level stack (a) for a real partition and (b) for a virtual machine.

Virtual machines, on the other end, first, have only one thread. Second, and similarly to “real” partitions, a stack is allocated and initialized for its only thread; this stack, however, is initialized so that its first stack frame points to the “pok_arch_virt” function, instead of “pok_arch_rfi”, as illustrated in Figure 3.11(b). Therefore, when there is a context switch to this stack for the first time, the CPU program counter will be set to point to “pok_arch_virt” which, in turn and as illustrated in Figure 3.12:

1. Creates a VCPU.
2. Loads the VCPU, passing the duration of the corresponding partition window (a.k.a., quantum) as an argument.
3. Sets “__VM” (a global variable) to point to the VCPU context.
4. Finally, resumes the VCPU, thus resuming the execution of the guest.

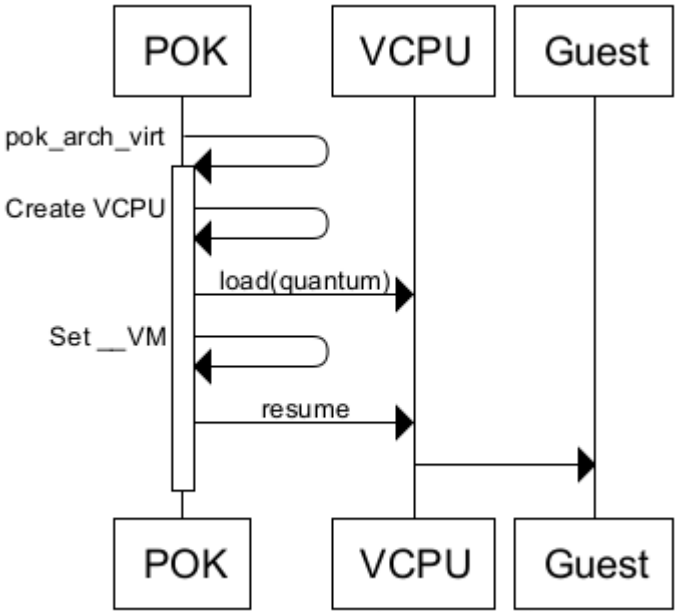


Figure 3.12. Sequence diagram for "pok_arch_virt" in POK/rodosvisor.

After resuming the execution of the guest, whenever an interrupt occurs, it is always handled first by POK (the host) which, as illustrated in Figure 3.13:

- If "__VM" is not null then, it will redirect interrupts to the corresponding VCPU interrupt handlers;
- Else, if "__VM" is null, POK's default interrupt handlers are called.

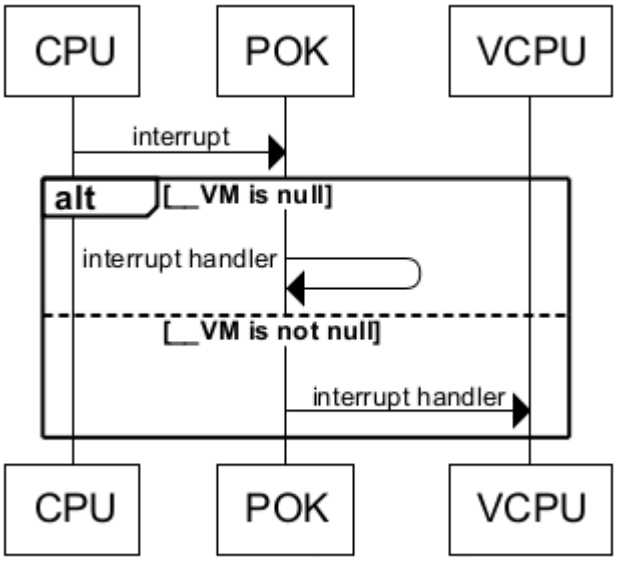


Figure 3.13. Interrupt handling in POK/rodosvisor.

As explained in *Section 2.3. Interface with the Host*, when the end of the partition window is reached, the VCPU calls “systick” which, as illustrated in Figure 3.14, performs the following operations:

1. Save the VCPU.
2. Set “__VM” to zero.
3. Update POK's scheduler tick counter to reflect the execution time of the VCPU (when a VCPU is scheduled, POK's scheduler is suspended).
4. Call the scheduler.
5. At last, when the scheduler returns (i.e., when the context is switched again to the VCPU), perform a “longjmp” to “pok_arch_virt”, step #2 above, and the cycle is restarted.

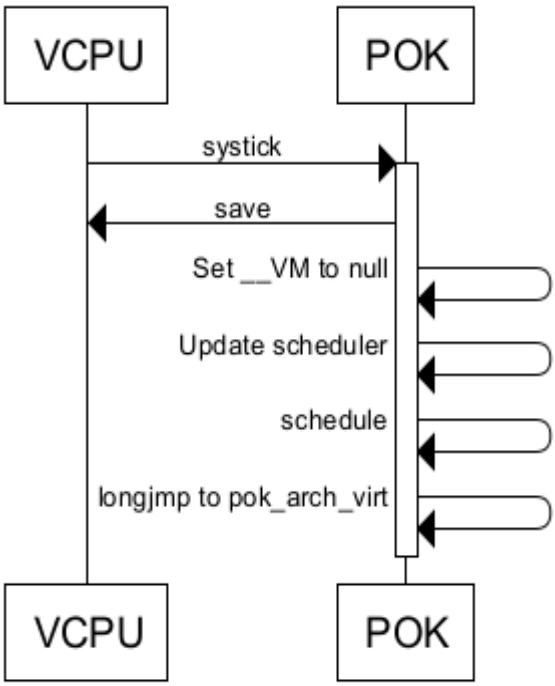


Figure 3.14. Sequence diagram for POK/rodosvisor's systick.

Like ARINC 653 partitions, guests of a virtual machine must be compiled and linked independently of the kernel, and during the link stage of the kernel, guests' executables are bundled together with the kernel in a dedicated section of the kernel executable, as explained in the previous section.

Similarly to what has been done for privileged partitions, new configuration options have been added to POK's variability management system. These new configuration options enable:

- the selection of which partitions are virtual machines and which are “real” partitions;
- the configuration of each VCPU address space (includes memory and memory-mapped I/O devices);
- and, the configuration of whether or not a VCPU requires a virtual interrupt controller, and which devices are connected to the virtual interrupt controller.

Table 3.2 presents the number of new and modified source lines of code (SLOC) required for the integration of Rodosvisor with POK as well as an estimation of the development effort, schedule, number of developers and cost, based on the basic Constructive Cost Model (COCOMO) [81]. It shows that almost 2000 SLOC were added/modified, with an estimated cost of $32660.

Table 3.2. The number of new and modified source lines of code (SLOC), and an estimation of the development effort, schedule, number of developers and cost, needed for the integration of Rodosvisor with POK.

Metric	Value
Architecture-dependent (SLOC), C++ programming language	470
Architecture-dependent (SLOC), C programming language	98
Architecture-dependent (SLOC), Assembly (PowerPC)	630
Total (SLOC)	1198
Development Effort Estimate (Person-Months)	2.90
Schedule Estimate (Months)	3.75
Estimated Average Number of Developers (Effort/Schedule)	0.77
Total Estimated Cost to Develop (average salary = $56,286/year, overhead = 2.40)	$32660

3.5. Evaluation

3.5.1. Evaluation Platform

The results shown in the following sections have been collected on a hardware platform with the architecture depicted in Figure 3.15. It consists of:

- a PowerPC 405 at 300 MHz;
- a memory controller (i.e., Xilinx Multi-Port Memory Controller, version 4.03a) connected to 256 MB of DDR SDRAM;
- a serial port (i.e., Xilinx XPS UART Lite, version 1.00a) with one start bit, one stop bit, and a baudrate of 115200;
- an ethernet media access controller (MAC) (i.e., Xilinx XPS Ethernet Lite Media Access Controller, version 2.00b), which supports the IEEE 802.3 media independent interface to industry standard physical layer devices, and provides 10/100 Mbps interfaces;
- an interrupt controller (i.e., Xilinx XPS Interrupt Controller, version 1.00a) connected to the serial port's and the ethernet MAC's interrupt request signals;

- three processor local buses (PLB), version 4.6: two of them connect the PowerPC 405 exclusively to the memory controller for improved performance (one for fetching instructions and another for data load/store operations); the third, connects the PowerPC 405 to memory-mapped I/O devices, namely, the ethernet MAC, the serial port, and the interrupt controller.

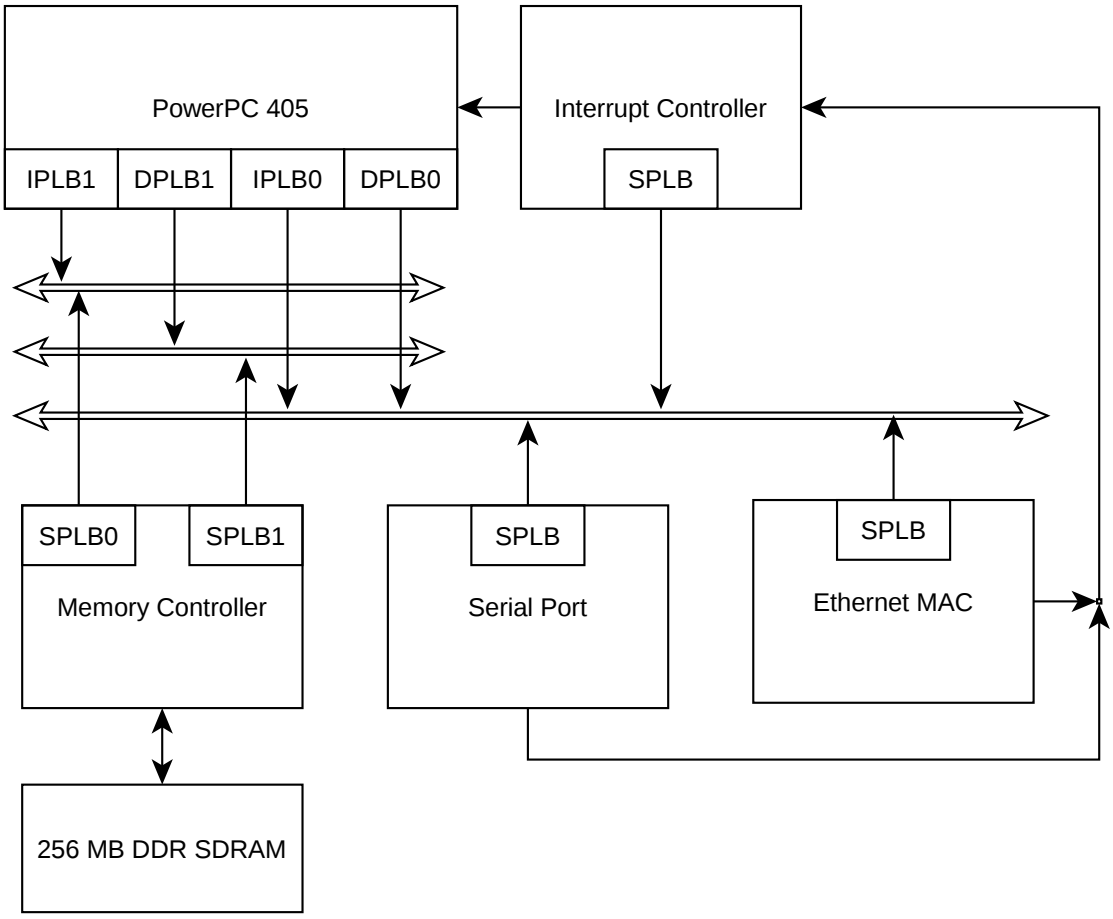


Figure 3.15. Hardware architecture of the evaluation platform.

The PowerPC 405 [22] is a 32-bits Reduced Instruction Set Computer (RISC) processor with a five-stage pipeline; it is a low power processor (0.9 mW/MHz) designed for high performance embedded systems. The PowerPC 405 features: a hardware multiply/divide unit; 16 KB two-way set-associative instruction cache and 16 KB two-way set-associative data cache; a memory management unit with a software-managed 64-entry translation look-aside buffer; several timer units; debug facilities; etc.

The hardware platform just described has been realized on a Xilinx University Program Virtex-II

Pro Development System (XUPV2P) [82]. The XUPV2P is equipped with a Virtex-II Pro (XC2VP30) field-programmable gate array (FPGA) [83], [84] which is surrounded by a comprehensive collection of input and output devices which can be used to create a complex computing platform. Some of the features of the Virtex-II Pro FPGA are shown in Table 3.3. The XUPV2P is equipped with the following devices and ports:

- Power supplies
- Power-on reset circuitry
- 100 MHz system clock
- 75 MHz SATA clock
- User clocks (2)
- RS-232 DB9 serial port
- PS/2 serial ports for mouse and keyboard (2)
- 10/100 Fast Ethernet transceiver and port
- Video DAC and 15-pin high-density D-sub connector for XSGA output, up to 1200 x 1600 at 70 Hz
- AC-97 audio CODEC with audio amplifier and speaker/headphone output and line level output
- Microphone and line level audio input
- LED (4)
- Switches (4)
- Push Buttons (5)
- 184-pin dual in-line memory module (DIMM) for Double Data Rate (DDR) Synchronous Dynamic RAM (SDRAM) with up to 2 GB
- CPU trace and debug port
- Serial Advanced Technology Attachment (SATA) ports (two Host ports and one Target port)
- Sub-Miniature A port (SMA)
- High-speed expansion connector
- Low-speed expansion connectors (6)
- etc...

Table 3.3. Features of the Virtex-II Pro (XC2VP30).

Feature	**Virtex-II Pro (XC2VP30)**
Logic Cells	30816
Slices	13969
Array Size	80 X 46
Distributed RAM	428 Kb
Multiplier Blocks (18 x 18 bit)	136
Block RAM	2448 Kb
Digital Clock Managers	8
PowerPC 405	2
Multi-Gigabit Transceivers	8

Xilinx Embedded Development Kit, version 10.1 [48], has been used to create, configure and generate the hardware platform, on a Fedora 19 host.

All of the software has been compiled on a Fedora 19 host using freely available tools such as GNU make [85], a GCC cross-compiler for the PowerPC 405 [86], version 4.8.1, among others. All of the software has been compiled with optimizations enabled ("-O2" compiler option), unless otherwise noted. The following compiler options were also used when compiling C++ code:

- -fno-rtti: "Disable generation of information about every class with virtual functions for use by the C++ run-time type identification features (dynamic_cast and typeid). If you do not use those parts of the language, you can save some space by using this flag. Note that exception handling uses the same information, but G++ generates it as needed. The dynamic_cast operator can still be used for casts that do not require run-time type information, i.e. casts to void * or to unambiguous base classes" [86].
- -fno-threadsafe-statics: "Do not emit the extra code to use the routines specified in the C++ ABI for thread-safe initialization of local statics. You can use this option to reduce code size slightly in code that does not need to be thread-safe" [86].

- -fno-exceptions: “[Disable] exception handling. [When enabled,] generates extra code needed to propagate exceptions. For some targets, this implies GCC generates frame unwind information for all functions, which can produce significant data size overhead, although it does not affect execution“ [86].

3.5.2. Cumulative Virtualization Overhead

In this section, the cumulative virtualization overhead in a Linux-based operating system, as a guest on top of POK/rodosvisor, is presented, for three different types of workloads, namely: compute-, I/O- and CPU-management-intensive workloads. For that two configurations have been developed and tested:

(L1) A system based on Linux 2.6.39, with a small RAM-based file system (i.e., initramfs), running on bare metal, as illustrated in Figure 3.16(L1).

(L2) A system based on POK/rodosvisor, with one virtual machine whose guest is the same Linux-based operating system described in L1, as illustrated in Figure 3.16(L2). The virtual machine is configured: (1) to have direct access to the serial port and to the ethernet MAC; and (2) with a virtual interrupt controller which emulates a physical interrupt controller. Direct access to a physical interrupt controller is not given because it can be configured to bypass the hypervisor (i.e., POK/rodosvisor) and violate time and space partitioning.

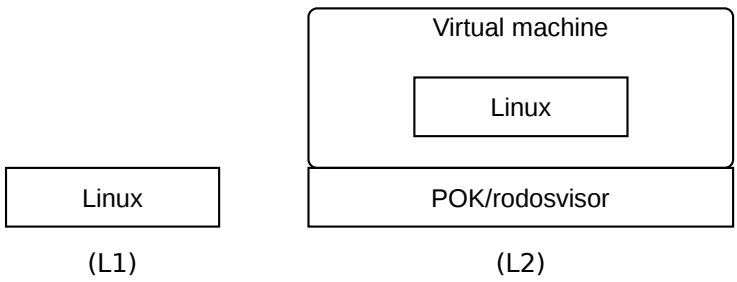


Figure 3.16. Configurations (L1) and (L2).

As explained in *Section 2.4.3. Memory Management Unit*, independent control over instruction and data address translation, a feature of the PowerPC 405, is currently not supported by POK/rodosvisor. Linux, however, uses this feature when flushing the cache during initialization. Therefore, the Linux kernel's source code has been modified (for both L1 and L2) to work around

this limitation, requiring 34 new/modified source lines of code. Other than that, no modifications were necessary.

On top of the two configurations described above, several benchmarks have been executed. Table 3.4 presents the results of those benchmarks, the associated virtualization overhead, and their respective type.

Table 3.4. The results of several benchmarks, performed on top of a Linux-based operating system, running on bare metal (i.e., L1), or as a guest on POK/rodosvisor (i.e., L2). For each benchmark, the associated virtualization overhead, in percentage, and its type are also shown.

	L1	L2	Overhead (%)	Type
Boot (seconds)				
Decompression	36	36	0	Compute
Linux	2.9	4.2	44.83	CPU mgmt.
Shell	1	4	300	CPU mgmt.
Total	39.9	43.2	8.27	
Dhrystone (Dhrystones per Second)				
	577800.9	566989.9	1.91	Compute
Whetstone (C Converted Double Precision Whetstones (MIPS))				
	3.8	3.7	2.7	Compute
Netperf (Mbit/s)				
TCP_STREAM	17.4	13.9	25.18	I/O

The following benchmarks can be found in Table 3.4.

Boot, Decompression: The boot process of the Linux-based operating system has been divided into 3 stages, and this result corresponds to the first stage. It corresponds to the time required to decompress the Linux kernel image (including the initramfs). In this stage, the use of CPU management operations (e.g., privileged instructions, interrupts) is negligible, and thus, this benchmark has been classified as compute-intensive (i.e., “Compute” in Table 3.4). Results show

that the virtualization overhead is non-existent. This is expected since there is very little use of CPU management operations, and therefore, the hypervisor is rarely activated.

Boot, Linux: This is the second stage of the boot process of the Linux-based operating system, and it corresponds to the initialization of the Linux kernel. In this stage the use of CPU management operations is high, therefore, this benchmark has been classified as CPU-management-intensive (i.e., “CPU mgmt.” in Table 3.4). Results show that the virtualization overhead is high (i.e., 44.83%), which is expected, since the use of CPU management operations is high and, therefore, the hypervisor is activated often. This overhead, however, includes not only the hypervisor's overhead, but also the overhead caused by cache trashing during the interaction between the hypervisor and the Linux kernel.

Boot, Shell: This is the third and last stage of the boot process, which corresponds to the initialization of a Buildroot-based shell [87]. Similarly to the previous stage, the use of CPU management operations is significant, thus, this benchmark has been classified as CPU-management-intensive. Compared to the previous stage, however, the use of CPU management operations is even higher as this stage begins with the execution of the “init” process, on top of Linux, and with the activation of process scheduling, context switching, system calls, etc. Therefore, as the results show, the virtualization overhead is also higher than in the previous stage.

Dhrystone and Whetstone: Dhrystone [88] is a benchmark which measures the performance of integer and string operations, and Whetstone [89] is a benchmark which measures, mainly, the performance of floating-point arithmetic. These two benchmarks have been classified as compute-intensive. Results show that the virtualization overhead is low, but noticeable. It was expected that, similarly to the first stage of the boot process, the virtualization overhead would be negligible, as the operations being benchmarked do not rely on CPU management operations, and thus, do not lead to the activation of the hypervisor. What these results show is the virtualization overhead during the execution of the benchmark, caused by Linux's normal operation (e.g., scheduling) as well as the overhead caused by cache trashing during the interaction among the hypervisor, the Linux kernel, the benchmark process, and other processes running concurrently on top of the Linux kernel.

Netperf: Lastly, Netperf [90] is a benchmark which measures the throughput and latency for various kinds of network connections (e.g., TCP, UDP). To perform this benchmark, *netserver* executed on the target while *netperf* executed on a Fedora 19 host. This benchmark has been classified as I/O-intensive (i.e., “I/O” in Table 3.4). The results for the TCP_STREAM test profile,

which measures throughput over a TCP connection, are shown. It can be seen that L1's maximum throughput is 25% higher than that of L2's. In this benchmark, the virtualization overhead includes not only the overhead caused by CPU management operations (in the Linux kernel, in the device drivers, in the protocol stack, etc.), but also the virtualization overhead of the virtual interrupt controller which is connected to the ethernet MAC. Considering the complexity of this benchmark, which includes two device drivers (i.e., one for the ethernet MAC and another for the interrupt controller) and a protocol stack, we expect the virtualization overhead to be much lower for most I/O-intensive workloads, which are not as complex.

To sum up, these results show that:

- for compute-intensive workloads, the virtualization overhead is low and, in some cases, negligible;
- for I/O-intensive workloads, the virtualization overhead is significant, but for most workloads, we expect it to be low;
- for CPU-management-intensive workloads, however, the virtualization overhead can be quite significant.

This indicates that low-end hardware full virtualization is more adequate for compute-intensive workloads, with moderate use of I/O, and with low use of CPU management operations. Some of the reasons for this behavior will become much more clear in the next section, which presents POK/rodosvisor's performance profile.

By showing the results of some benchmarks executed on top of a Linux-based operating system as a guest on POK/rodosvisor, it demonstrates, indirectly, compatibility with legacy software (i.e., a complete Linux-based operating system), despite some minor modifications that had to be applied to the Linux kernel's source code. To the best of our knowledge, this is the first work ever to present performance measurements about a Linux-based operating system as a guest on top of a hypervisor based on low-end hardware full virtualization, and therefore, demonstrating compatibility with a common real-world scenario.

3.5.3. Hypervisor's Performance Profile

To obtain the hypervisor's performance profile, configuration L2, described in the previous section, was reused. POK/rodosvisor's built-in, custom profiler was enabled and profiling data were collected during 90 seconds since boot (it includes almost 45 seconds required to boot the Linux-based operating system, and another 45 seconds of idle time).

The profiling data is composed of several probes, each associated with a specific code path. For each probe the following data were collected: the number of samples (or activations), the minimum, the maximum and the total execution time; the average execution time was obtained by dividing the total execution time by the number of samples.

In Table 3.5, the number of samples, the total and the average execution times per probe are shown (only probes with a number of samples greater than zero are presented). This table includes all the interrupt handlers that have been activated (e.g., program, instruction and data TLB-miss), as well as the time to context switch in and out of the virtual machine. It can be seen that:

- The program interrupt handler, which is responsible for the emulation of privileged instructions, makes up for 91.7% of the total execution time. However, it is also, by far, the most activated interrupt handler (i.e., 96.9%), and it actually has a low average execution time. This partially explains why CPU-management-intensive benchmarks performed so poorly in the previous section.
- Next to the program interrupt handler, the instruction and data TLB-miss interrupt handlers make up for 8% of the total execution time even though their contribution to the total number of samples is less than 3%. This happens because of the high (actually, the highest) average execution time for these two interrupt handlers.
- The remaining interrupt handlers and context switching make up for less than 0.4%, which compared with the ones discussed above is negligible.
- Finally, the average execution time of the data TLB-miss interrupt handler is higher than the system call interrupt handler, indicating that, as mentioned in *Section 2.6. Paravirtualization*, the overhead of “hypercall detection based on the system call interrupt” is lower than “hypercall detection based on access to supervised memory.”

To sum up, these results indicate, similarly to the previous section, that low-end hardware full virtualization is a serious alternative for workloads which do not rely heavily on CPU management operations.

Table 3.5. The number of samples, the average and the total execution times per probe. Execution times are given in CPU clock cycles. In parenthesis, the ratio to the summation of all the values in the same column is given, in percentage.

	No. samples	Average	Total
Program	1841211 (96.857%)	511	868824642 (91.639%)
Data TLB-Miss	29310 (1.542%)	1356	39366270 (4.152%)
Instruction TLB-Miss	22824 (1.201%)	1594	36074969 (3.805%)
System Call	4800 (0.253%)	419	1945588 (0.205%)
Data Storage	1875 (0.099%)	364	657625 (0.069%)
PIT	475 (0.025%)	1312	616728 (0.065%)
Context Switch In	90 (0.005%)	3247	292259 (0.031%)
Context Switch Out	89 (0.005%)	1815	161549 (0.017%)
Instruction Storage	253 (0.013%)	522	128658 (0.014%)
External	28 (0.001%)	988	27300 (0.003%)

3.5.4. Footprint

3.5.4.1. Trusted Computing Base

In Table 3.6 the number of source lines of code (SLOC) in Rodosvisor alone is presented, as measured by SLOCCount [91]. Alongside, an estimation of the development effort, schedule, number of developers and total cost, based on the basic Constructive Cost Model (COCOMO) [81], is also shown. It can be seen that Rodosvisor is made of around 8.5 KSLOC, with a schedule estimate of around 8 months (using 2.77 developers), and a total estimated cost of $254509.

Table 3.6. Rodosvisor's trusted computing base in terms of source lines of code (SLOC), and an estimate of the respective development effort, schedule, number of developers and total cost.

Metric	Value
C++ (SLOC)	8294
Assembly (PowerPC 405) (SLOC)	172
Total (SLOC)	8466
Development Effort Estimate (Person-Months)	22.61
Schedule Estimate (Months)	8.18
Estimated Average Number of Developers (Effort/Schedule)	2.77
Total Estimated Cost to Develop (average salary = $56,286/year, overhead = 2.40)	$254509

Similarly, Table 3.7 shows the number of POK/rodosvisor's SLOC, as measured by SLOCCount, and an estimation of the development effort, schedule, number of developers and total cost. Table 3.7 shows that POK/rodosvisor is composed by 15.2 KSLOC, with a schedule estimate of around 10 months (using 4.04 developers), and a total estimated cost of $469219.

Table 3.7. POK/rodosvisor's trusted computing base in terms of source lines of code (SLOC), and an estimate of the respective development effort, schedule, number of developers and total cost.

Metric	Value
C (SLOC)	5640
C++ (SLOC)	8792
Assembly (PowerPC 405) (SLOC)	728
Total (SLOC)	15160
Development Effort Estimate (Person-Months)	41.68
Schedule Estimate (Months)	10.32
Estimated Average Number of Developers (Effort/Schedule)	4.04
Total Estimated Cost to Develop (average salary = $56,286/year, overhead = 2.40)	$469219

3.5.4.2. Memory Footprint

In order to evaluate the impact of the various partition types in terms of memory footprint, the following three sets of configurations have been developed:

(A) Configurations with one up to ten ARINC 653 partitions as illustrated in Figure 3.17(a); each partition has two threads (a mandatory main thread, and a worker thread which performs no operation).

(P) Configurations with a single privileged partition composed of a mandatory main thread and one up to ten worker threads which perform no operation as illustrated in Figure 3.17(b). While ARINC 653 partitions and virtual machines enforce a protected address space, a privileged partition does not; therefore, a system with more than one privileged partition, even though it is possible, brings no advantages, and leads to higher memory footprint.

(V) Configurations with one up to ten virtual machines, as illustrated in Figure 3.17(c); the guest is the same for all virtual machines and performs no operation.

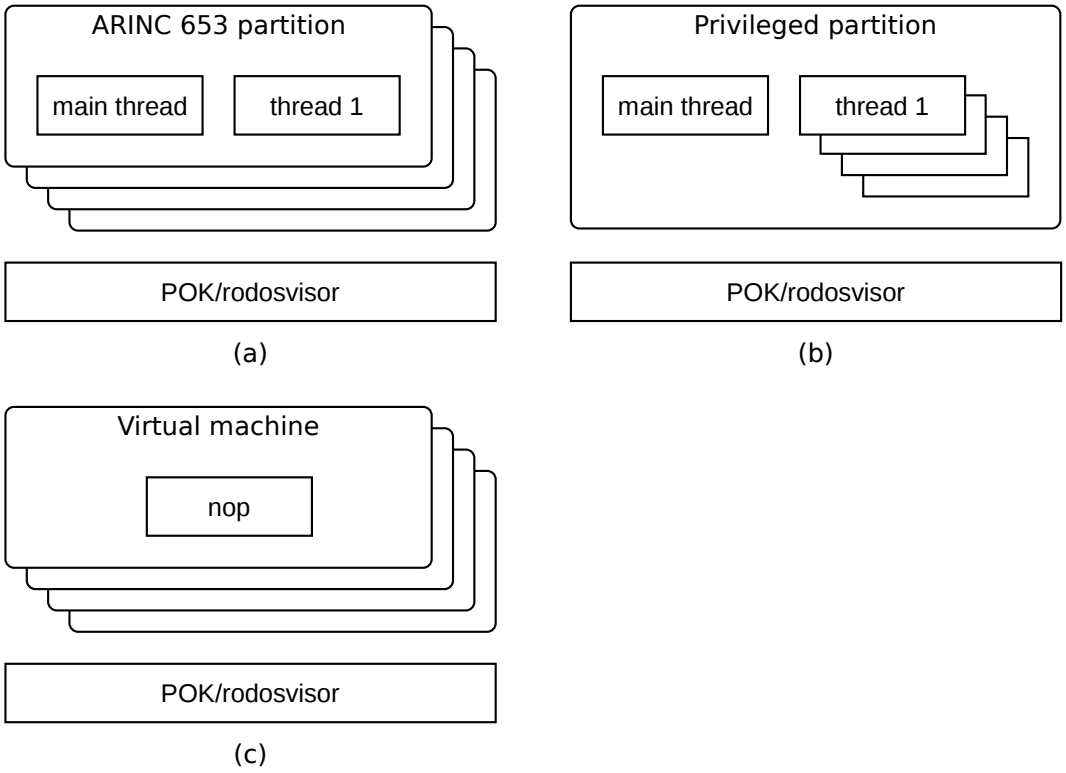


Figure 3.17. Configurations based on (a) ARINC 653 partitions, (b) a privileged partition, and (c) virtual machines.

All of the configuration described above have been compiled, and from the compilation results, the following has been measured: (1) the size of the code; (2) the size of read-only data; (3) the size of read/write data; and (4) the combined size of all stacks.

In Figure 3.18 the size of the code measured from all of the configurations is shown. It can be seen that:

- ARINC 653 partitions lead to the smallest code size and, as the number of partitions increase, the code size remains approximately constant. Differences in code size are the result from the optimizations performed by the compiler.
- A privileged partition leads to slightly larger code size than ARINC 653 partitions and, as the number of worker threads increase, the code size remains approximately constant. The configurations used for these results, however, are composed of worker threads which perform no operations, their size is very small, and therefore, their impact on memory footprint goes almost unnoticed. If the worker threads were larger their impact on memory footprint would also be larger. As explained in *Section 3.3. Privileged Partitions*, a

privileged partition is always part of the kernel.

- Virtual machines lead to the largest code size: around 25 KB larger. From this, the Rodosvisor's code size can be derived, i.e., 25 KB. As the number of virtual machines increase, the code size also increases due to optimizations for speed performed by the compiler (e.g., loop unrolling).

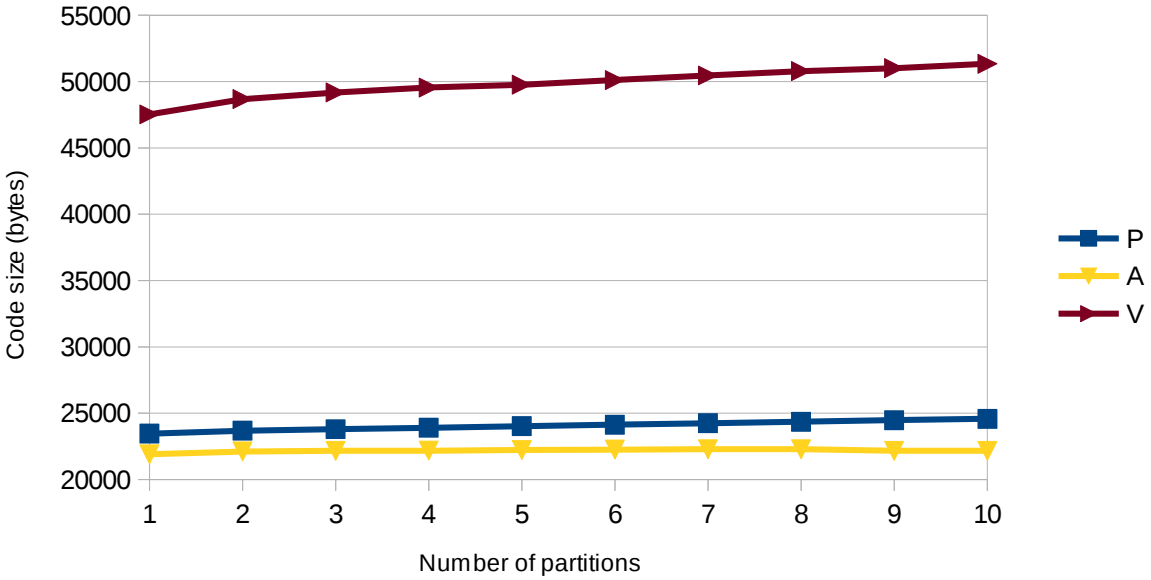


Figure 3.18. Code size for all configurations.

In Figure 3.19 the size of read-only data for all of the configuration is shown. It can be seen that:

- An ARINC 653 partition has a higher impact on the size of read-only data than a privileged partition's thread. This has to do with the fact that the size of many configuration structures is directly proportional to the number of partitions, and independent of the number of threads.
- Compared with ARINC 653 partitions, a privileged partition's thread leads to a slightly smaller size of read-only data. However, as explained previously, if the size of the worker thread's read-only data was larger, the overall read-only data would also be larger.
- Configurations based on virtual machines lead to a size of read-only data that is up to two orders of magnitude higher than configurations based on other partition types. The main reason for this is that three large jump tables are required for emulation of privileged instruction.

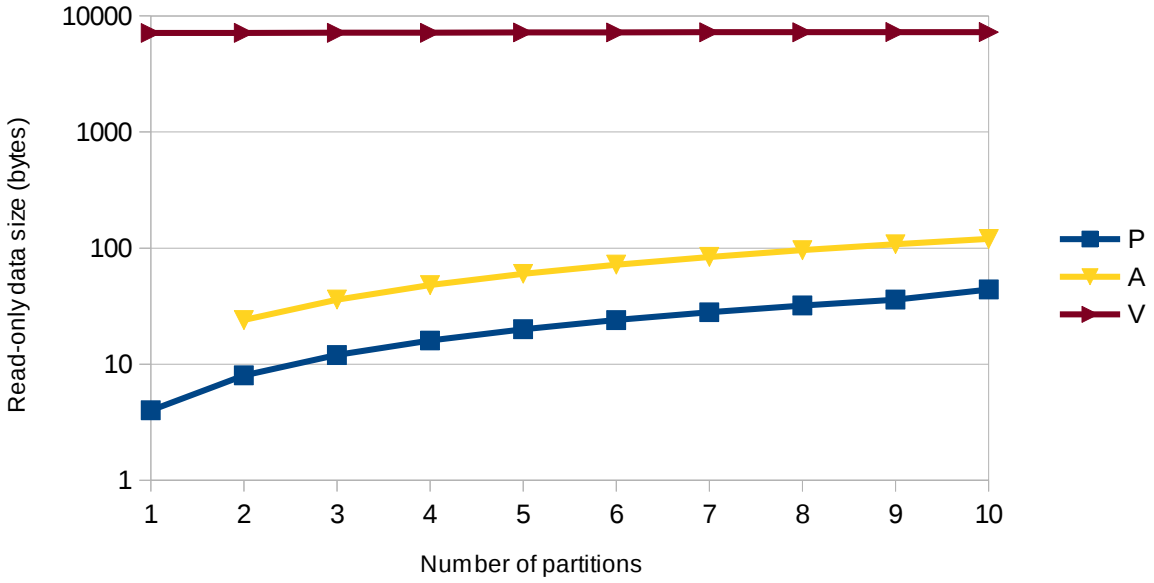


Figure 3.19. Read-only data size for all configurations. When there is only one ARINC 653 partition, the size of read-only data is zero (data point not shown).

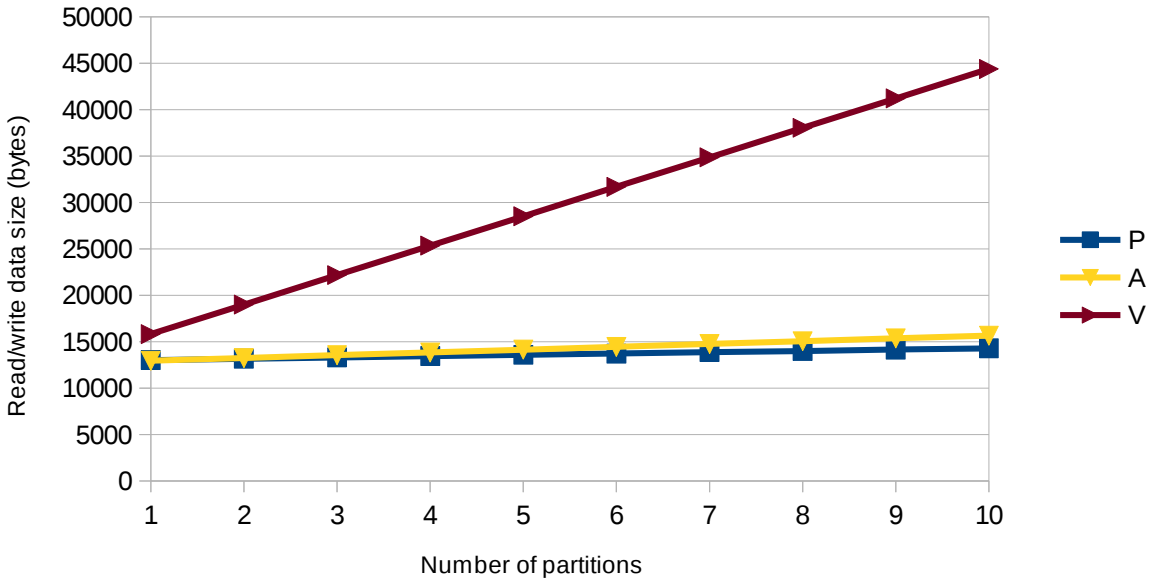


Figure 3.20. Size of read/write data for all configurations.

In Figure 3.20 the size of read/write (RW) data from all of the configuration is shown. It can be seen that:

- Compared with ARINC 653 partitions, privileged partitions, in these particular cases, lead to

a smaller size of RW data. However, if the size of the worker thread's RW data was larger, the overall RW data would also be larger. Still, the size of many configuration structures is directly proportional to the number of partitions, and independent of the number of threads.

- Configurations based on virtual machines, lead to a large size of RW data. The main reason for this is the large register file (around 3 KB) required per virtual machine.

In Figure 3.21, the combined size of all stacks for all configurations is shown (only kernel-level stacks are considered). Stacks may be considered RW data; they have been distinguished from other RW data in order to reveal other properties. It can be seen that:

- ARINC 653 partitions lead to the largest combined size of the stacks. Each ARINC 653 partition requires one stack for the main thread, and an additional stack for each worker thread. Each main/worker thread stack has a size of 8 KB; an additional idle thread with a size of 1 KB is also required for all configurations.
- Compared with ARINC 653 partitions, a privileged partition leads to a smaller combined size of the stacks. The stack requirements for privileged partitions are the same as for ARINC 653 partitions; however, since for any given system, only one privileged partition is required, then, only one main thread's stack is required, leading to a smaller combined size of the stacks.
- Configurations based on virtual machines lead to the smallest combined size of the stacks since each virtual machine requires only one thread, and thus, only one stack.

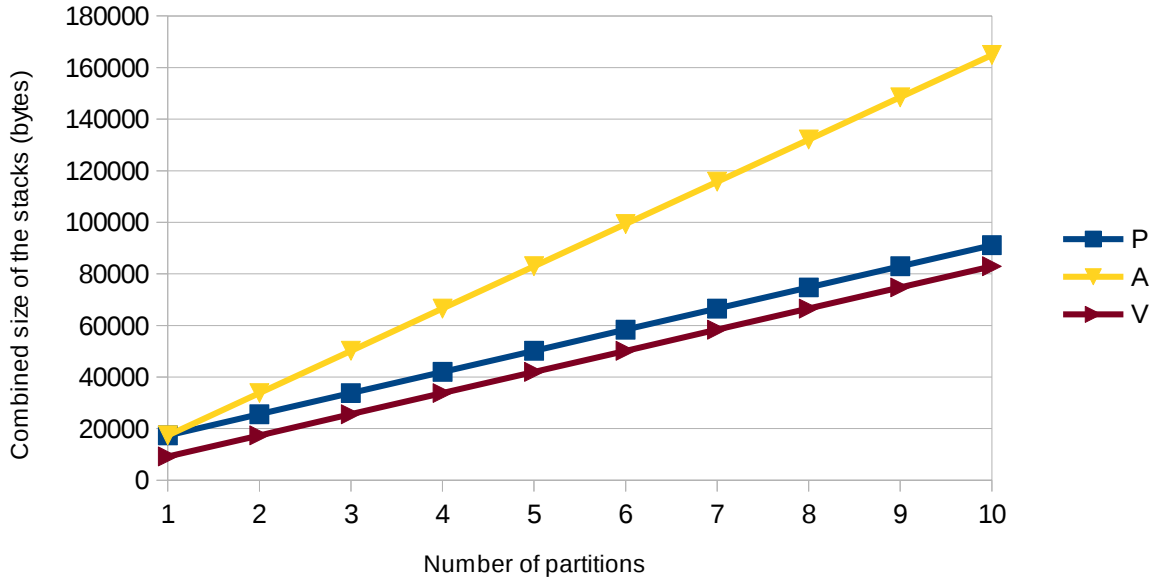


Figure 3.21. Combined size of all stacks for all configurations.

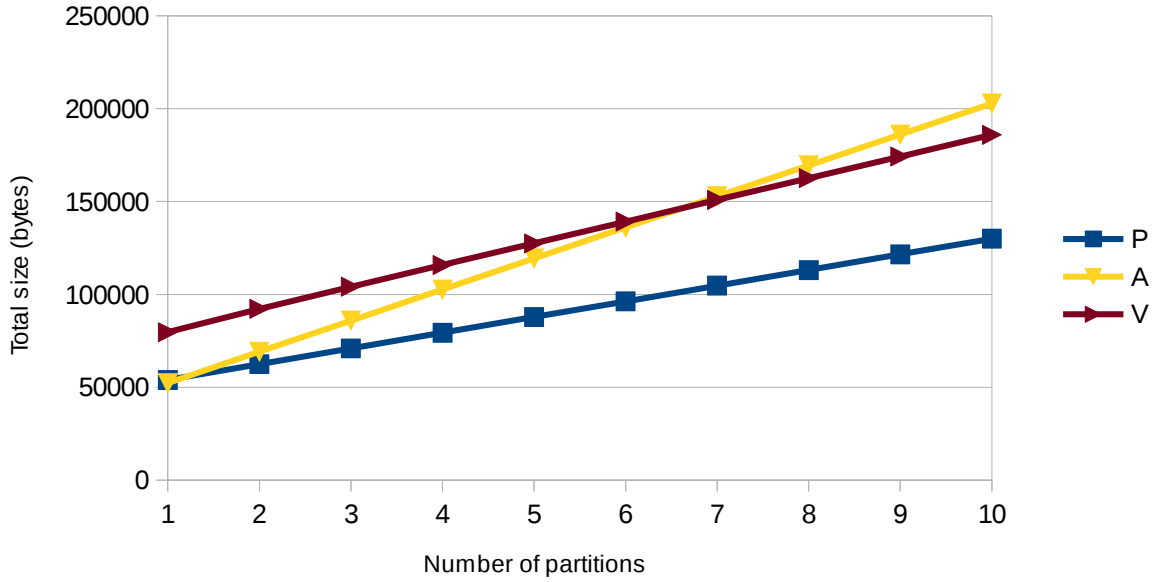


Figure 3.22. Combined size of (1) the code, (2) read-only and (3) read/write data, and (4) the combined size of all stacks, for all configurations.

At last, in Figure 3.22, the combined size of the code, read-only data, read/write data, as well as the combined size of the stacks for all configurations is presented. It can be seen that:

- Configurations based on a privileged partition lead to the smallest size; the size increases

with the number of partitions, mainly because of the combined size of the stacks.

- Configurations based on ARINC 653 partitions lead to a larger size than configurations based on a privileged partition, mostly because of the additional stack requirements.
- For a configuration with a small number of partitions (less or equal than six), virtual machines leads to larger size than ARINC 653 partitions. However, for a configuration with a large number of partitions (more than six), virtual machines lead to a smaller size.

3.6. Future Work

As future work we propose:

- Taking into account the bottlenecks found in Rodosvisor's VCPUs, find new methods to improve its performance.
- Integrate Rodosvisor with another kernel or operating system (e.g., FreeRTOS, Linux) in order to further evaluate and improve its design and reusability.
- Integrate POK's interrupt-handlers with Rodosvisor's VCPU interrupt-handlers and reduce the size of the TCB and improve performance.
- Implement all partition types in terms of privileged partitions. This will enable a reduction of the complexity and coupling in the core kernel; it will also foster decoupling between the various functionalities provided by the kernel.

3.7. Summary

In this chapter, it has been described how support for privileged partitions has been added to POK as well as how Rodosvisor has been integrated with POK. It has been shown that the engineering effort required for adding support for privileged partitions was very low and, similarly, that the engineering effort required by POK/rodosvisor's integration was also low.

Performance and footprint measurements from POK/rodosvisor have also been presented.

The evaluation of the virtualization overhead, and POK/rodosvisor's performance profile showed that, low-end hardware full virtualization is more adequate for compute-intensive workloads, with moderate use of I/O, and with low use of CPU management operations. Compatibility with legacy software has been demonstrated by showing the results of some benchmarks executed on top of a Linux-based operating system as a guest on POK/rodosvisor.

In terms of footprint, it has been shown that the kernel's footprint for a virtual-machine-based

system can be lower than the kernel's footprint for an ARINC-653-compliant system, which indicates that the requirements of low-end hardware full virtualization regarding the kernel's footprint are not high.

Altogether, we believe that these results demonstrate that, for many applications, low-end hardware full virtualization can be a serious alternative to high-end hardware full virtualization and paravirtualization, enabling:

- a reduction of system size, weight, power consumption, cost, etc., when compared with high-end hardware full virtualization;
- a reduction (in many cases, elimination) of the effort required to port legacy software to a hypervisor-specific interface, when compared with paravirtualization;
- as well as, in some cases, a reduction of the kernel's footprint, when compared with ARINC-653-based systems.

4. Model-Driven Engineering using Ocarina

4.1. Introduction

As explained in *Section 1.2. Model-Driven Engineering*, conventional development methods are unable to keep up with the requirements of modern safety-critical systems. Many methodologies are proposed in the literature to address this problem, such as computer-aided software engineering, component-based software engineering, software product line engineering, and model-driven engineering, all of which are described in *Section 4.2. Development Methods: State of the Art*.

POK, mentioned throughout this thesis, through Ocarina [34]–[37], supports a model-driven engineering approach which advocates that engineering/development should be driven by high-level models; more specifically, code generation based on high-level models.

Ocarina is a compiler for the Analysis & Architecture Description Language (AADL) [38]. AADL enables the specification of the software and hardware architecture, and thus, the specification of the desired system configuration using a high-level abstraction language featuring: processors, memory, partitions, virtual machines, threads, inter and intra-partition communication, etc. Ocarina is able to transform an AADL model into a POK configuration composed by C source code and makefiles; Ocarina is also capable of generating partitions, if necessary. Ocarina, however, did not support some of the features added to POK during this work, namely: virtual machines, and privileged partitions.

In this chapter, it is explained how AADL is used to represent some of the features developed during this work, namely: privileged partitions and virtual machines. At the same time, it is explained how Ocarina has been modified to support those new representation, and generate a POK configuration accordingly. In this way, it is demonstrated the ability of AADL and Ocarina to support privileged partitions and virtual machines, and thus, their ability to replace conventional development methods. In this chapter, the source lines of code (SLOC) of all AADL models developed for this thesis are also compared with the corresponding SLOC of generated code, to demonstrate the ability of AADL to reduce the development effort.

4.1.1. Chapter Organization

This chapter is organized as follows.

In *Section 4.2. Development Methods: State of the Art*, it is reviewed some novel software engineering methods, proposed in the literature, which attempt to address the limitations of

conventional development methods.

In *Section 4.3. Architecture Analysis & Design Language* and *Section 4.4. Ocarina*, AADL and Ocarina are described, respectively.

The way in which we have chosen to represent privileged partitions and virtual machines in AADL, and how Ocarina has been modified to support those representations, are described in *Section 4.5. Privileged Partitions* and *Section 4.6. Virtual Machines*, respectively.

In *Section 4.7. Evaluation*, a comparison between the SLOC of all AADL models developed for this thesis and the corresponding SLOC of generated code is presented, and the ability of AADL-based model-driven engineering to reduce the development effort demonstrated.

Proposals for future work can be found in *Section 4.8. Future Work*.

Finally, in *Section 4.9. Summary*, a summary of the chapter is given.

4.2. Development Methods: State of the Art

4.2.1. Computer-Aided Software Engineering

The first major effort to evolve past 3rd Generation Programming Languages (3GPL) was computer-aided software engineering (CASE) [27], [28]. At the time 3GPL started to reveal their limitations to cope with system complexity and the semantic gap between the problem and the solution spaces was very high. CASE focused on general-purpose graphical programming (e.g., state machines, structure diagrams, data flow diagrams) to express design intent. Major goals of CASE included:

- Analysis of graphical programs (less complex than conventional 3GPL);
- Synthesization of implementation artifacts from graphical programs.

CASE, however, failed to become widely adopted. The technology was not mature enough to have wide applicability. At the same time, programming languages and platforms evolved (e.g., abstract data type, objects, and object-oriented programming), raising the abstraction level, and thus alleviating the need for CASE.

Nowadays, 3GPL are again no longer able to control the complexity of new software. System complexity has grown fast and 3GPL are unable to express domain concepts effectively. 3GPL are abstractions of the underlying computing environment (the solution space) and do not enable the actual design intent to be expressed effectively (in the problem space). There is a semantic gap between the design intent and the expression of this intent in source code.

Furthermore, nowadays software (e.g., firmware) is often designed using informal, resistant to change, error prone and tedious methods (e.g., designs on paper or white boards) [92]. Design decisions, important for the success of long-lived products, are easily lost, as well as opportunities for automation. After design, software is implemented “manually” using languages such as C/C++, Ada, Java, etc. The current size of the software however, makes these long and difficult processes, with consequences on the resulting quality and time-to-market.

4.2.2. Component-Based Software Engineering

Component-based software engineering (CBSE) defends that systems should be designed by assembling software components together [29], [30]. Components' configuration may evolve (even during run-time) as a response to changing requirements or to the environment [93], [94]. CBSE has the following advantages:

- Components clearly separate implementation from interface, and thus reduce coupling, improve separation of concerns, and improve reuse [93], [95].
- CBSE facilitates variation in structure (the architecture) and content (the implementation) [30].
- CBSE fosters independent development of components, and thus, contributes to lower development time [30], [96].
- CBSE enables that which is absolutely necessary to be included in the final system [93].
- High reuse, and thus, lower fault rate, lower development time and cost [93].

CBSE however, is not without limitations, namely:

- A trade-off between performance (e.g., throughput, latency) and flexibility has often to be made [94], [95]. However, the overhead introduced by CBSE, if any, is “negligible” according to [95].
- Component-based architectures for the embedded systems' domain are not mature enough.

There has been a wide adoption of CBSE in the enterprise computing domain (e.g., .Net, J2EE, CORBA's CM). Results in that domain however, cannot be applied directly to embedded systems because of the different constrains, such as tight resource budget, performance constrains, and so on.

In the past, the concept of CBSE has been introduced into the mainstream through Object Oriented

Programming (OOP) where classes separate interface from implementation. OOP, however, has limitations and is too fine-grained [30].

THINK [93] is a CBSE framework which provides flexible, efficient binding of components (establishing communication, send and receive data). There are many forms of bindings (from simple pointers to complex distributed channels), and a lot of backstage support is necessary, such that a framework has been created to manage all that. Other known examples of the application of CBSE for embedded systems include: CAmkES [96], TinyOS [97], Hartex [98], and KOALA [30].

4.2.3. Software Product Line Engineering

Software Product Line Engineering (SPLE) [31], [32] enables organizations to built an array of similar software products (applications) from a common (domain specific) and a variable (application specific) pool of resources, and thus, reduce development effort and cost, when compared with conventional single system development. SPLE takes advantage of the commonalities between the software in a particular domain while enabling control over product-specific variability. SPLE emerged as a way to address a demand for highly personalized software systems [32].

At the core of SPLE are the software product lines which are usually split into two major processes, as illustrated in the Figure 4.1: domain engineering and application engineering. Domain engineering focuses on the commonalities of the domain and where product-specific variability may be necessary. The result from domain engineering is a framework enabling developers to reuse a common set of artifacts, obeying to a specific architecture/paradigm. Application engineering focuses on reusing the artifacts developed during domain engineering and on the development of application-specific artifacts to build a particular product. These two processes (domain and application engineering) can be further split into the conventional software engineering sub-processes, as illustrated in Figure 4.1, namely: requirements engineering, design, realization/implementation, and quality assurance/testing. The major advantage of SPLE is high reuse, and thus, improved quality, improved variability management, and lower time-to-market [31], [32]. On the other end, the major limitation of SPL is that development of a SPL requires an upfront investment; however, according to [31], the return of investment is commonly achieved after 3 products developed from the SPL.

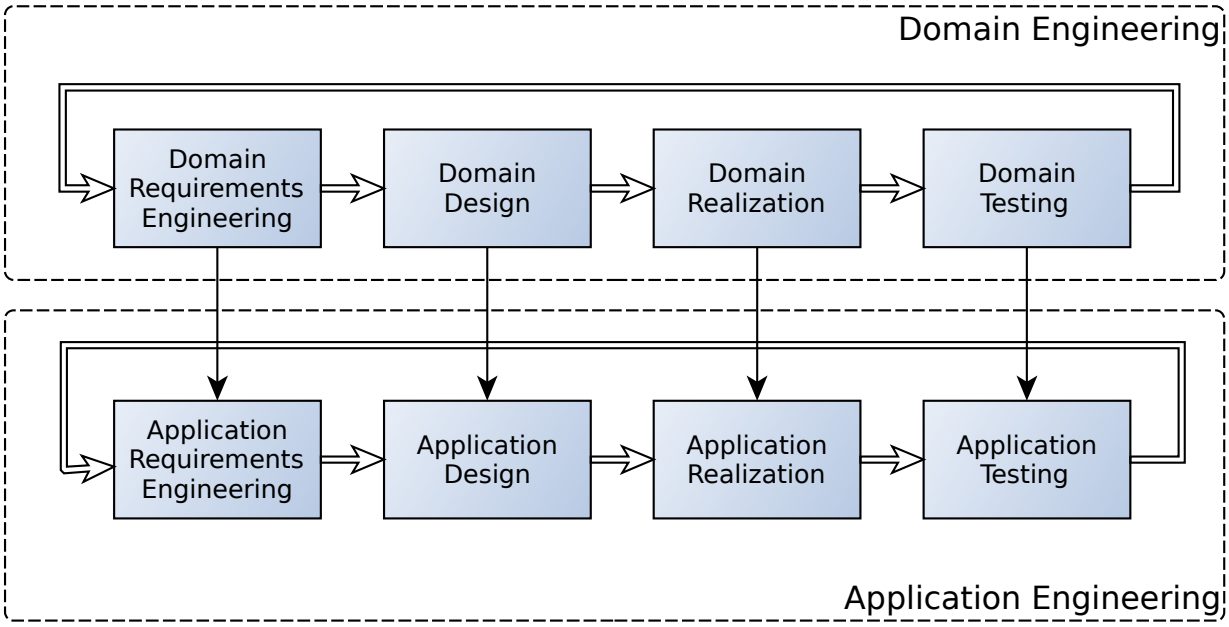


Figure 4.1. Domain and application engineering in SPLE (adapted from [124]).

4.2.4. Model-Driven Engineering

Model-driven engineering (MDE) [28], [33] advocates that engineering/development should be driven by high-level models. High-level models raise the abstraction level (when compared with 3GPL) and reduce the distance between the problem space and the solution space, thus facilitating development. Models have formal semantics and, therefore, implementation artifacts can be automatically generated from those models. These models may represent the software architecture, its behavior, structure and content, as well as domain-specific concepts (e.g., real-time systems, databases). Although models are clearly distinguished from 3GPL, the two are essentially the same: abstractions of the solution space. Models, however, raise the abstraction level much higher than 3GPL, enabling them to express complex concepts effectively and closer to the problem space [99]. MDE has the following advantages over conventional development methods [34], [99], [100]:

- Lower accidental complexity and lower learning curve, and thus, improved communication between stakeholders with different backgrounds.
- In the design phase, it enables early design analysis and error detection (e.g., violation of domain-specific constrains), and thus, it facilitates the job of finding a design that fulfills the requirements.

- It enables automated synthesization of implementation artifacts (e.g., boilerplate code generation); furthermore, compilers can perform a better job at optimizing their output, and thus, produce highly efficient systems.

MDE, however, also has the following limitations [25], [100]:

- A trade-off must be made between raising the abstraction level and oversimplification.
- Redundancy: To properly model a system there has to be a set of models, all of them modeling the same systems but from different perspectives.
- Round-trip: With multiple levels of abstraction, a change in a lower abstraction level must be propagated to the higher abstraction levels. That however, is not an easy thing to do.
- One can be moving complexity instead of reducing it.
- Expertise is required: In order to fully understand a MDE development environment, one has to have an understanding of all the abstraction layers, and, for example, understand the effect that a change in some abstraction level will have on the other levels.
- Existing modeling languages (e.g., UML) are known for its complexity, and, in some cases, imprecise semantics or none at all.
- Limited tool support: That however, is starting to change together with the development in MDE and other tightly related areas (e.g., Eclipse Modeling Framework).

A particular realization of MDE is the model-driven architecture (MDA) from OMG [101]. It defines that business logic should be implemented in a platform independent model (PIM) which is subsequently translated into one or more platform specific models (PSM). A PSM describes the implementation of the PIM in terms of the target platform. Other realization of MDE include: Agile MDE, domain-oriented programming, software factories, etc.

Hutchinson et. al, 2011 [102], evaluated how three commercial organizations adopted MDE, through in-depth semi-structured interviews, in order to contribute to the very limited number of empirical studies on MDE and its application in the “real world.” In this study, they focus on the “organizational, managerial and social factors” which contribute to the success or failure of introducing MDE; technical factors are not considered. For the study three companies were considered: “the printer company,” “the car company,” and “the telecom company.” With this study the authors identified that:

- The success of the deployment of MDE should be done in a “progressive and iterative”

manner, instead of an “all or nothing” approach.

- The users, developers and maintainers of the MDE framework must be committed and motivated to do so.
- MDE's stakeholder must be committed to respond to any issues that may come up and to adopt new methods of working.
- MDE must have a clear business goal.

One interesting outcome of this study is the realization that MDE, besides the technicalities involved, requires a strong cultural change and commitment.

4.3. Architecture Analysis & Design Language

The Architecture Analysis & Design Language (AADL) [38] is an architecture description language which can be represented as text, graphically, or using XML Metadata Interchange (XMI). AADL has been developed by academic and industrial partners and standardized by the Society of Automotive Engineers. AADL has been developed for modeling distributed real-time embedded systems, and to support the complete engineering life-cycle, including: specification, analysis, tuning, integration, and upgrade.

AADL is based on a component model which distinguishes interface from implementation and where an implementation is a particular realization of a specific interface. As illustrated in Listing 4.1 and Listing 4.2, an interface is specified by a type declaration (e.g., lines 7-11 and 24-28), and an implementation is specified by an implementation declaration (e.g., lines 15-20 and 31-40). In a type declaration the externally visible interface of a component is specified in terms of “features” (e.g., lines 8-10 and 25-27). On the other end, in an implementation declaration the internal structure of a component is defined; the internal structure of a component may include sub-components (i.e., other components) (e.g., lines 32-33 and 53-54), connections between sub-components, and connections between the component interface and the sub-components interfaces (e.g., lines 34-36 and 55-57).

```
1  -- an AADL comment is preceded by "--"
2  package example -- beginning of the "example" package
3  public -- public contents of the "example" package
4
5  -- a subprogram type declaration
6
7  subprogram compute
8  features -- interface
9     i : in parameter integer;
10    o : out parameter integer;
11 end compute;
12
13 -- a subprogram implementation declaration
14
15 subprogram implementation compute.impl
16 properties
17    source_language => C;
18    source_text => ("compute.o"); -- object file name
19    source_name => "do_compute"; -- function name
20 end compute.impl;
21
22 -- a thread type declaration
23
24 thread thread_compute
25 features
26    inp : in event data port integer;
27    outp : out event data port integer;
28 end thread_compute;
```

Listing 4.1. An example of an AADL model (continued in Listing 4.2).

```
29 -- a thread implementation declaration
30
31 thread implementation thread_compute.impl
32 calls
33     c1 : { comp : subprogram compute.impl; };
34 connections
35     port inp -> comp.i;
36     port comp.o -> outp;
37 properties
38     period => 500 ms;
39     deadline => 100 ms;
40 end thread_compute.impl;
41
42 -- a process type declaration
43
44 process process_type
45 features
46    data_in : in event data port integer;
47    data_out : out event data port integer;
48 end process_type;
49
50 -- a process implementation declaration
51
52 process implementation process_type.impl
53 subcomponents
54     th : thread thread_compute.impl;
55 connections
56     port data_in -> thread_compute.inp;
57     port thread_compute.outp -> data_out;
58 end process_type.impl;
59
60 end; -- end of the “example” package
```

Listing 4.2. An example of an AADL model (continuation of Listing 4.1).

In AADL, all type and implementation declarations are associated with a specific component category. The category of component defines what is expected of the component during run-time,

restricting the interface and implementation that is possible. AADL specifies four sets of component categories: (1) software, (2) execution platform, (3) composite, and (4) abstract.

First, the set of software component categories, enable modeling of source text (e.g., C/C++, Ada, Java), concurrent tasks, protected and virtual address space, among others. More specifically, software component categories include:

- subprogram: models source text that is executed sequentially (e.g., lines 7-11 and 15-20); graphically, a subprogram should be represented as a solid line ellipse, as illustrated in Figure 4.2, based on Listing 4.1 and Listing 4.2.
- thread: models concurrent tasks, the basic unit of execution and schedulability (e.g., lines 24-28 and 31-40); graphically, a thread should be represented as a dashed line parallelogram, as illustrated in Figure 4.2, based on Listing 4.1 and Listing 4.2.
- process: models a protected or virtual address space, and must contain at least one thread (e.g., lines 44-48 and 52-58); graphically, a process should be represented as a solid line parallelogram, as illustrated in Figure 4.2, based on Listing 4.1 and Listing 4.2.
- data: models static data in source text; graphically, a data components should be represented by a solid line rectangle, as illustrated in Figure 4.3.

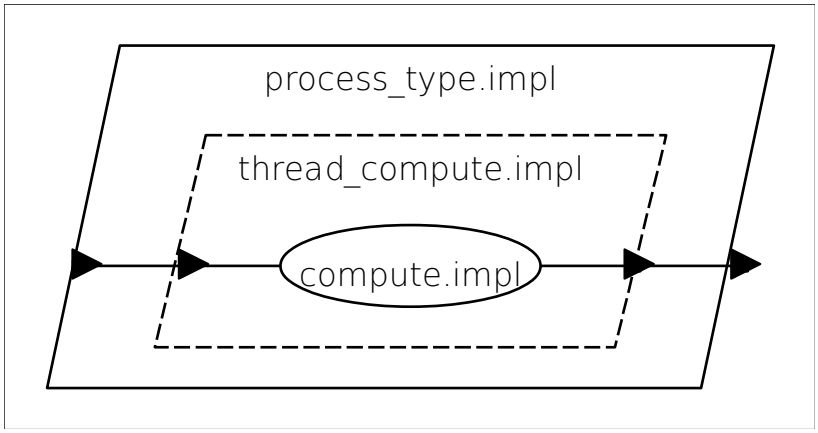


Figure 4.2. Standard AADL graphical representation of the model in Listing 4.1 and Listing 4.2.

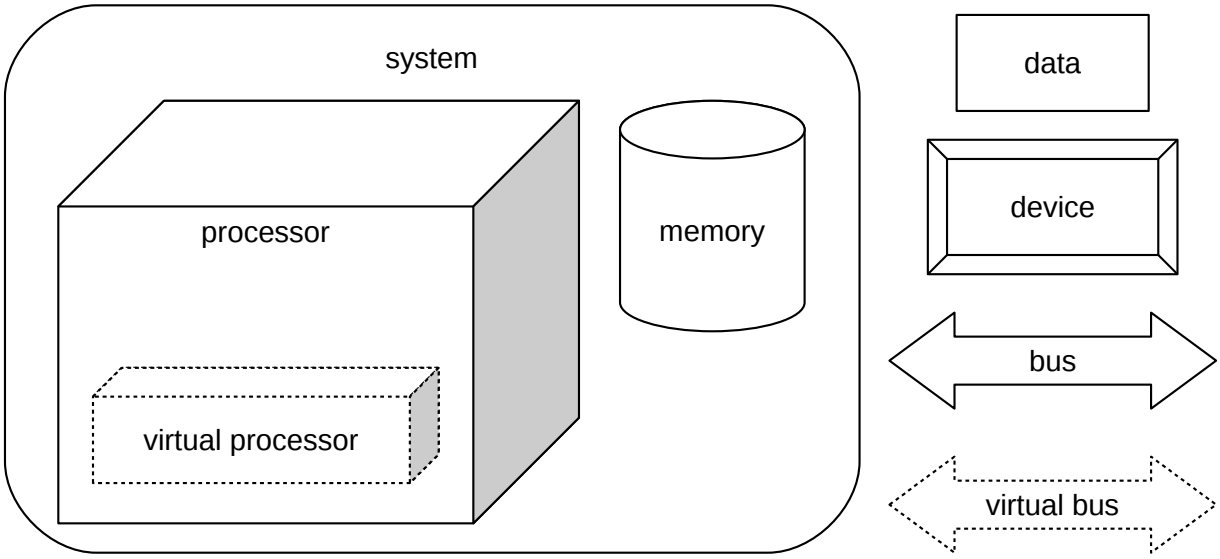


Figure 4.3. Standard AADL graphical representation for some AADL components.

Second, the set of execution platform component categories represent a combination of hardware and software capable of supporting the execution of software, storage, and communications. Execution platform components include:

- processor: a combination of hardware and software responsible for protected address spaces and for scheduling threads; graphically, a processor should be represented by a solid line rectangular parallelepiped, as illustrated in Figure 4.3.
- virtual processor: a logical resource which provides an additional level of schedulability, and which enables the specification of virtual machines; graphically, a virtual processor should be represented by a dashed line rectangular parallelipiped, as illustrated in Figure 4.3.
- memory: represents randomly accessible physical storage such as ROM and RAM; graphically, memory should be represented by a cylinder, as illustrated in Figure 4.3.
- device: represents a component capable of interacting with the external environment (e.g., sensors, actuators); graphically, a device should be represented as illustrated in Figure 4.3.
- bus: a physical or logical communication channel between execution platform components; graphically, a bus should be represented by a solid line double edged block arrow, as illustrated in Figure 4.3.
- virtual bus: a virtual communication channel or a communication protocol built on top of a

bus; graphically, a virtual bus should be represented by a dashed line double edged block arrow, as illustrated in Figure 4.3.

Third, the set of composite component categories, is only composed by the system component. Software components cannot be sub-components of hardware components, and vice versa. The system component enables the integration of the two, and enables software components to be mapped onto execution platform components; for example: mapping a process to memory, mapping a process to a processor or virtual processor, etc. Graphically, as illustrated in Figure 4.3, a system should be represented by solid line rounded rectangle.

Forth and last, the set of abstract component categories, is only composed by the abstract component which can be refined into any other component category.

The interface of a component is specified in terms of “features,” as explained above, and the connections between components are defined by connections between “features.” Each component category supports a different set of features, such as:

- data port: models transfer of state data (e.g., sensor data);
- event port: models transfer of control, through events (e.g., set-point reached, start thread);
- event data port: models transfer of events with data (e.g., logging) (e.g., lines 26, 27, 46 and 47);
- subprogram parameters: models the inputs and outputs of subprograms (e.g., lines 9 and 10).

Properties are name/value pairs which can be used to characterize a component, such as the period and deadline of threads (e.g., lines 38 and 39), or the source text associated with a subprogram (e.g., lines 17-19), among many others. Properties can be declared within type and implementation declarations; properties declared in a type declaration are inherited by an implementation declaration. There are a set of standard properties and standard property types; additional properties can be added through custom property sets. Standard properties sets include: real-time, communication, data representation, error handling, code generation, deployment, etc.

AADL has been chosen for this work because it provides the necessary abstraction for the representation of ARINC 653 partitions, privileged partitions, and virtual machines. Furthermore, it is supported by Ocarina, described in *Section 4.4. Ocarina*, a free and open source AADL compiler with one back-end for POK, used in this work.

Many architecture description languages can be found in the literature. Most of then, however, are

too broad and generic [103]–[105]. AADL, on the other end, is well established and standardized, and it is not overly specific yet defines precise semantics.

4.4. Ocarina

Ocarina is an open source compiler with several front-ends and several back-ends. The front-ends include: AADL versions 1 and 2, and Requirement Enforcement Analysis Language (REAL) [106], [107]. REAL is a language which enables the specification of constrains associated with AADL models, and thus, the verification of AADL models against those constraints; in this work, REAL has not been used. On the other end, it includes back-ends such as: POK, Linux, ORK+, PolyORB and PolyORB-HI, XtratuM, among many others.

As illustrated in Figure 4.4, Ocarina, after initialization and after processing all command line arguments, operates according to following process. First, it performs syntactic and semantic analysis on all input AADL models and on all standard AADL property sets required by all input models; at the end of semantic analysis an abstract syntax tree (AST) is obtained which mirrors the structure of the input models. Second, the AST obtained after semantic analysis is instantiated. Instantiation consists in reducing the AST by eliminating unused nodes, by binding related nodes with each other, and, depending on the back-end, by associating extra information with nodes in the AST. At the end of this stage, the “instantiated AST” is obtained. Third, it runs through the instantiated AST several times, transforming it into a language-specific AST (e.g., C, Ada). This process is also known as model transformation, that is, the input AADL model is transformed into another language model. At the end of this stage, the “final AST” is obtained. POK's final AST is composed by:

- Kernel configuration code.
- Partitions' implementation and configuration code, if specified in the input model.
- Common implementation and configuration code which is responsible for integrating the partitions and the kernel.

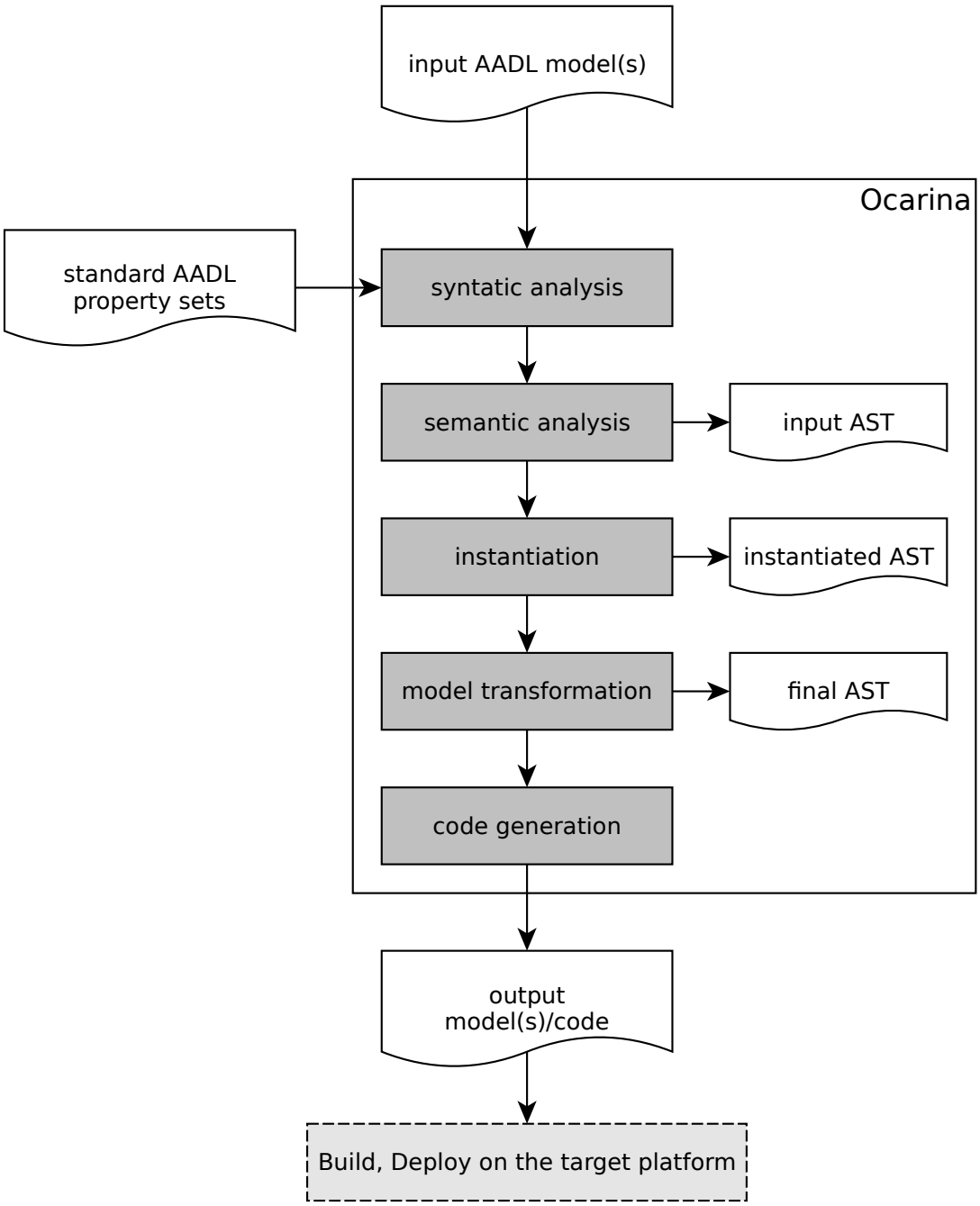


Figure 4.4. Ocarina's process after initialization and after processing all command line arguments.

Forth and last, the final AST is saved on the file system as (1) regular source code and library files, containing the implementation, and (2) Makefiles, which contain the instructions on how to build the partitions, the kernel, and the integration of the two (i.e., the final kernel image) ready to be downloaded to the target. This is illustrated by Figure 4.5. With the AST saved on the file system, it can be built/compiled and then, deployed on the desired target platform.

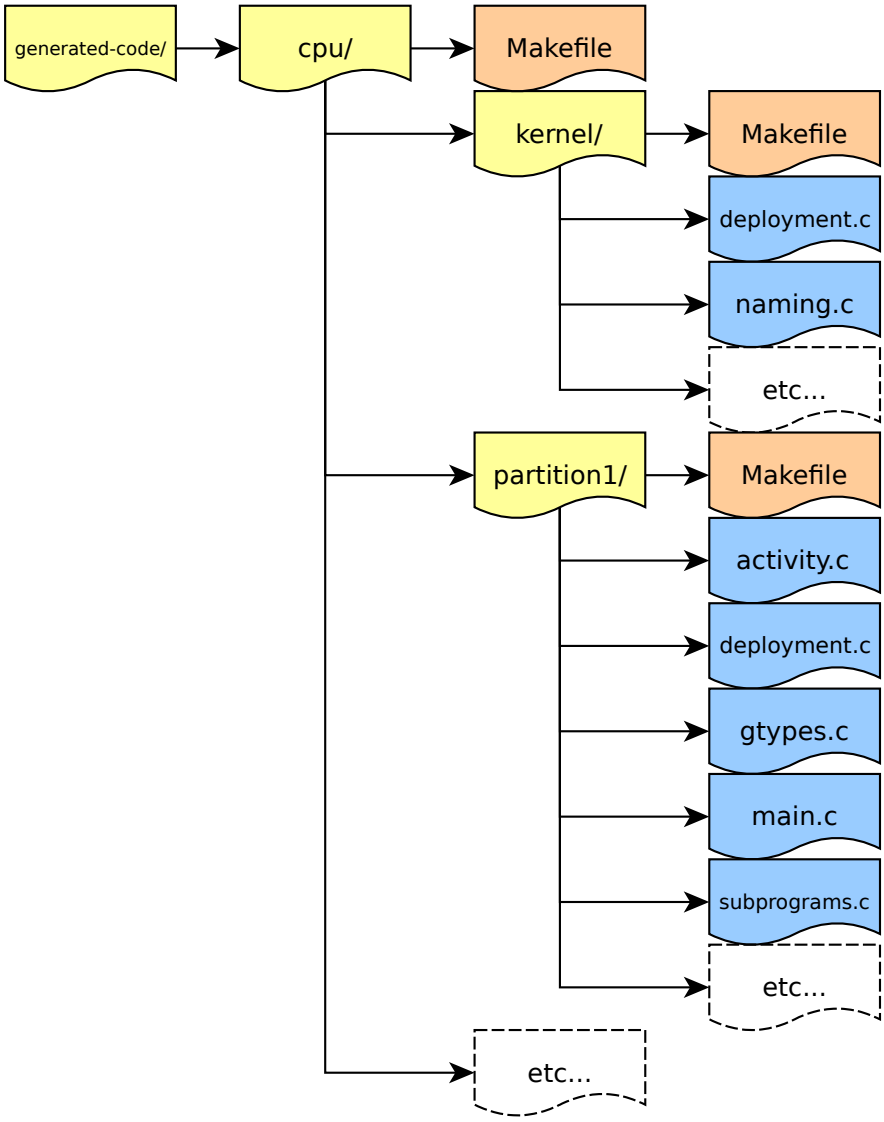


Figure 4.5. An example on how the final AST can be stored in the file system. Example based on Ocarina's POK back-end.

4.5. Privileged Partitions

In this section it is explained how we have chosen to represent a privileged partition in AADL, and how Ocarina's POK back-end has been modified to support privileged partitions and to generate a POK configuration accordingly.

In Ocarina's POK back-end a partition is represented as an AADL virtual processor, as illustrated in Listing 4.3; by default an AADL virtual processor represents an ARINC 653 partition. Support for privileged partitions has been added by extending Ocarina's POK AADL property set with a new property: “Virtual_Processor_Type”, applicable only to the virtual processor component category.

This is illustrated in Listing 4.4. First, the available virtual processor types are declared (lines 5-6): “arinc_653”, for ARINC 653 partitions, and “privileged”, for privileged partitions. Second, the “Virtual_Processor_Type” property is declared (lines 8-10), which can be assigned with any of the available virtual processor types declared earlier. With this property the developer can declare an ARINC 653 partition or a privileged partition as illustrated in Listing 4.5: the AADL virtual processor named “partition_1” is an ARINC 653 partition (lines 8-12) while “partition_2” is a privileged partition (lines 14-18).

```
1  package example -- AADL comment
1  public -- public namespace
2
3  virtual processor partition_generic
4  end partition_generic;
5
6  end example;
```

Listing 4.3. Representation of a generic partition in AADL.

```
1  property set POK is -- AADL comment
2
3  -- existing properties...
4
5  Available_Virtual_Processor_Types:
6      type enumeration ( arinc_653, privileged );
7
8  Virtual_Processor_Type:
9      POK::Available_Virtual_Processor_Types
10     applies to virtual processor;
11
12 end;
```

Listing 4.4. Code added to POK’s AADL property set in order to support privileged partitions through the new Virtual_Processor_Type property.

```
 1 package example -- AADL comment
 2 public -- public namespace
 3
 4 with POK; -- include the POK library
 5
 6 -- other components...
 7
 8 virtual processor partition_1
 9 properties
10    -- other properties
11    POK::Virtual_Processor_Type => arinc_653;
12 end partition_1;
13
14 virtual processor partition_2
15 properties
16    -- other properties
17    POK::Virtual_Processor_Type => privileged;
18 end partition_2;
19
20 -- other components...
21
22 end example;
```

Listing 4.5. Sample usage of the Virtual_Processor_Type property: the virtual processor named "partition_1" is an ARINC 653 partition and "partition_2" is a privileged partition.

So far, it has been described how Ocarina's POK AADL property set has been modified to support privileged partitions. This alone, however, only enables the AADL model in Listing 4.5 to be a correct model; it does not enable Ocarina to generate a POK configuration accordingly. For that, Ocarina has been modified to recognize the new property (i.e., Virtual_Processor_Type) and to generate code for privileged partitions and the kernel accordingly. Ocarina has been modified such that, as illustrated in Figure 4.6, during privileged partitions' code generation:

- a flag to enable code common to ARINC 653 and privileged partitions to take action depending on the type of the partition that is being compiled (e.g., select the proper system call interface as explained in *Section 3.3. Privileged Partitions*);

- a flag to inhibit the link stage during compilation of the partition (privileged partitions are linked together with the kernel during compilation of the final kernel image as explained in *Section 3.3. Privileged Partitions*).

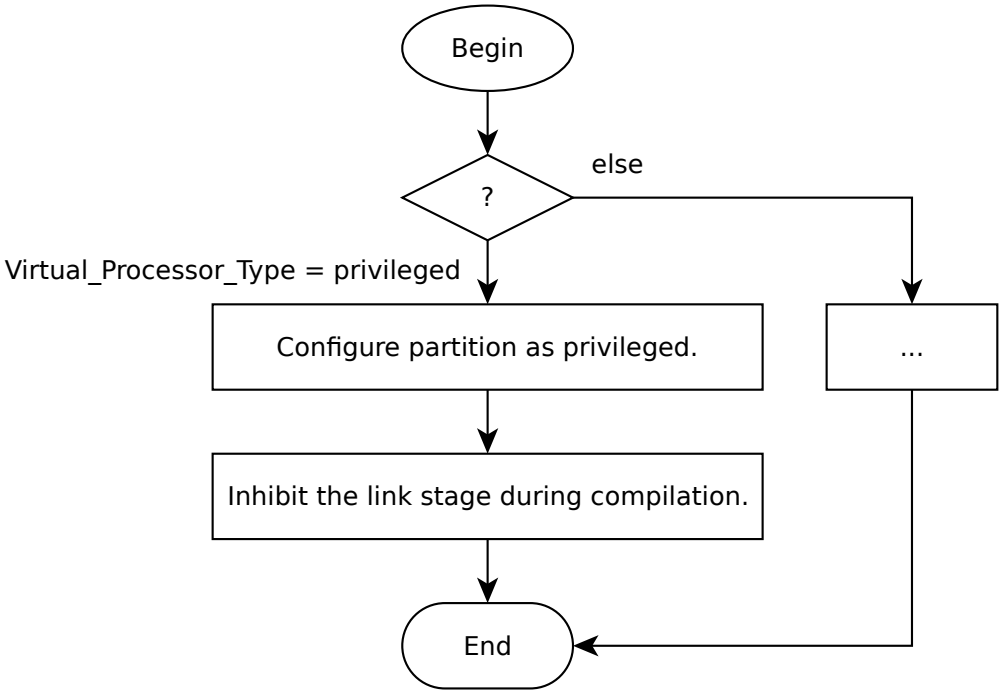


Figure 4.6. Ocarina's modified operation during partition's code generation.

During kernel's code generation, on the other end, as shown in Figure 4.7, the following is generated:

- an array describing the type of all partitions (i.e., ARINC 653 or privileged);
- an array with the entry points of privileged partitions which have to be specified manually, as explained in *Section 3.3. Privileged Partitions*;
- compilation flags extended with references to compiled-but-not-linked privileged partitions so that these can be subsequently linked and integrated with the kernel.

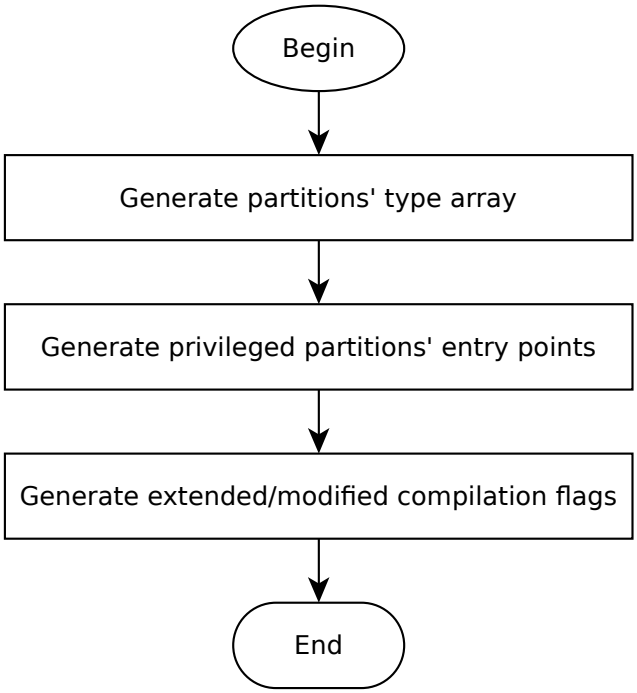


Figure 4.7. Ocarina's modified operation, required by privileged partitions, when generating code for the kernel.

In summary, adding support for privileged partition to Ocarina required: (1) the addition of new AADL property, enabling the specification of ARINC 653 or privileged partitions, and (2) the modification of Ocarina to recognize the new property and to generate code for privileged partitions and for the kernel accordingly. Table 4.1 presents the required number of new/modified Ocarina's source lines of code (SLOC) and an estimation of the respective development effort, schedule, number of developers, and cost. It is shown that 149 new/modified SLOC were required, with an estimated cost of $3660.

Table 4.1. The number of new and modified source lines of code (SLOC), and an estimation of the development effort, schedule, number of developers and cost [81], required in order to add support for privileged partition to Ocarina.

Metric	Value
Architecture-independent (SLOC), Ada programming language	149
Total (SLOC)	149
Development Effort Estimate (Person-Months)	0.33
Schedule Estimate (Months)	1.63
Estimated Average Number of Developers (Effort/Schedule)	0.20
Total Estimated Cost to Develop (average salary = $56,286/year, overhead = 2.40)	$3660

4.6. Virtual Machines

Similarly to the previous section, in this section it is explained how we have chosen to represent virtual machines in AADL, and how Ocarina's POK back-end has been modified to support virtual machines and to generate a POK configuration accordingly.

In AADL a partition is represented by a virtual processor and, by default, all virtual processors correspond to ARINC 653 partitions. Similarly to privileged partitions, support for virtual machines has been added by extending Ocarina's POK AADL “Virtual_Processor_Type” with a new possible assignment: “virtual_machine”. In Listing 4.6, the combined modifications to Ocarina's POK AADL property set required for privileged partitions and virtual machines is shown. First, the available virtual processor types are declared, “arinc_653”, “privileged”, and “virtual_machine” (lines 5-6). Second, the Virtual_Processor_Type property is declared, which can be assigned with any of the available virtual processor types declared earlier (lines 8-10). With this property the developer can declare a virtual machine as illustrated in Listing 4.7: the AADL virtual processor named “partition_1” is an ARINC 653 partition (lines 8-12), and “partition_3” is a virtual machine (lines 14-18).

```
property set POK is -- AADL comment

-- existing properties...

Available_Virtual_Processor_Types:
    type enumeration ( arinc_653, privileged, virtual_machine );

Virtual_Processor_Type:
    POK::Available_Virtual_Processor_Types
    applies to virtual processor;

end;
```

Listing 4.6. Code added to POK's AADL property set in order to support virtual machines through the new Virtual_Processor_Type property.

```
package example -- AADL comment
public -- public namespace

with POK; -- include the POK library

-- other components...

virtual processor partition_1
properties
   -- other properties
   POK::Virtual_Processor_Type => arinc_653;
end partition_1;

virtual processor partition_3
properties
   -- other properties
   POK::Virtual_Processor_Type => virtual_machine;
end partition_3;

-- other components...

end example;
```

Listing 4.7. Sample usage of the Virtual_Processor_Type property: the virtual processor named "partition_1" is an ARINC 653 partition, and "partition_3" is a virtual machine.

Up to this point, it has been described how POK's AADL property set has been modified to support

virtual machines. This alone, however, only enables the AADL model in Listing 4.7 to be a correct model; it does not enable Ocarina to generate a configuration accordingly. For that, Ocarina has been modified to recognize the new property (i.e., Virtual_Processor_Type) and to generate code for virtual machines and for the kernel accordingly. Currently, code generation for virtual machines is not supported; instead, the guest's executable must be provided directly. Therefore, Ocarina has been modified to inhibit code generation for virtual machines. On the other end, Ocarina has been modified so that, as shown in Figure 4.8, during kernel's code generation the following additional code is generated:

- an array specifying which partition are virtual machines and which are not;
- several arrays describing the characteristics/attributes of all the virtual machines (e.g., address space, error-handling, etc.).

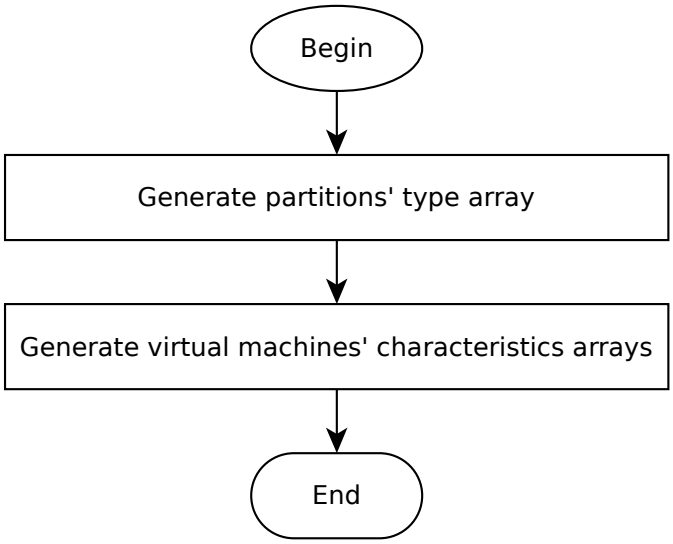


Figure 4.8. Ocarina's modified operation, required by virtual machines, when generating code for the kernel.

In summary, adding support for virtual machines to Ocarina required: (1) the addition of a new AADL property, enabling the specification of ARINC 653, privileged partitions, or virtual machines, and (2) the modification of Ocarina to recognize this new property and to generate code for the kernel accordingly, but to not generate code for virtual machines. Table 4.2 presents the required number of new/modified Ocarina's source lines of code (SLOC) and an estimation of the associated development effort, schedule, number of developers, and cost. It shows that 269

new/modified SLOC were required, with an estimated cost of $6805.

Table 4.2. The number of new and modified source lines of code (SLOC), and an estimation of the development effort, schedule, number of developers and cost [81], required in order to add support for virtual machines to Ocarina.

Metric	Value
Architecture-independent (SLOC), Ada programming language	269
Total (SLOC)	269
Development Effort Estimate (Person-Months)	0.6
Schedule Estimate (Months)	2.06
Estimated Average Number of Developers (Effort/Schedule)	0.29
Total Estimated Cost to Develop (average salary = $56,286/year, overhead = 2.40)	$6805

4.7. Evaluation

In this section the ability of AADL to reduce the development effort is demonstrated. For that, the source lines of code (SLOC) from all the AADL models developed for this thesis are compared with the corresponding generated SLOC.

Table 4.3 shows, for all configuration developed for this thesis: (1) the number of AADL SLOC, (2) the number of generated SLOC, and (3) the ratio of the number of AADL SLOC to the number of generated SLOC. Similarly, the graph in Figure 4.9 presents the ratio of the number of AADL SLOC to the number of generated SLOC in relation to the number of AADL SLOC.

It can be seen that, the no. AADL SLOC is always lower than the no. generated SLOC. In the worst case, the no. AADL SLOC corresponds to 56% of generated SLOC, while in the best case the no. AADL SLOC corresponds to less than 15% of generated SLOC. It can also be seen that as the no. AADL SLOC increase, the reduction of the engineering effort increases, meaning that AADL-based engineering/development is the most adequate for large systems. Considering that the higher the no. SLOC, the higher the development effort, then, using AADL always leads to lower development effort than a manual implementation; lower development effort, in turn, should lead to lower time-to-market and costs.

Table 4.3. For all configuration developed for this thesis: (1) the number of AADL SLOC, (2) the number of generated SLOC, and (3) the ratio of the number of AADL SLOC to the number of generated SLOC. The first and second columns represent, respectively: (1) the section where the configuration has been mentioned, and (2) the reference name of the configuration.

Section	Configuration	No. AADL SLOC	No. generated SLOC	Ratio (%)
3.5.2. Cumulative Virtualization Overhead	L2	81	190	42.63
3.5.4. Footprint	A1	116	379	30.61
	A2	121	605	20.00
	A3	126	831	15.16
	A4	131	1057	12.39
	P1	116	388	29.90
	P2	121	619	19.55
	P3	126	850	14.82
	P4	131	1081	12.12
	V1	104	186	55.91
	V2	109	197	55.33
	V3	114	208	54.81
	V4	119	219	54.34
5.4. Use Case: Serial Port Device Driver	Reference	89	393	23.23
5.5. Use Case: Inter-Partition Communication Subsystem	Reference	150	761	19.71

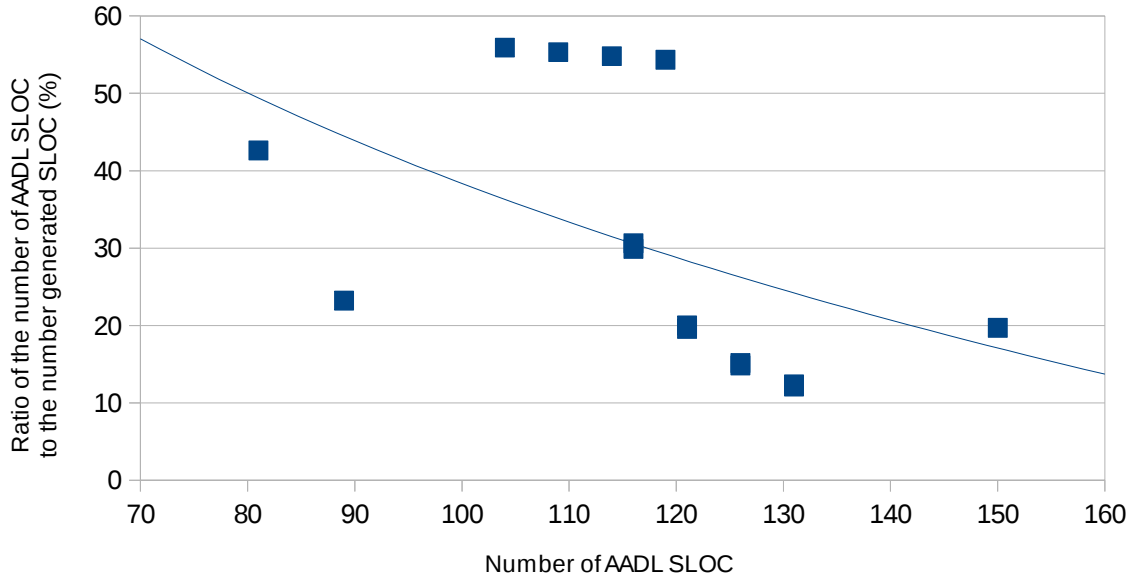


Figure 4.9. The ratio of the number of AADL SLOC to the number of generated SLOC in relation to the number of AADL SLOC.

4.8. Future Work

In the context of this chapter, we propose the following as future work:

- Employ MDE not only for generating the implementation but also for generating test cases, profilers, etc.
- Use REAL [106], [107] to enforce the constrains associated with virtual machines, such as: only one thread, the minimum size of the address space, etc.
- Ocarina is currently implemented in Ada, and it is not easy to modify; we propose a new implementation based on, for example, Model to Text Transformation Language within OMG's Meta-Object Facility [108].

4.9. Summary

In this chapter, it has been shown that AADL is able to support the representation of privileged partitions and virtual machines, and thus, it is able to replace conventional development methods. Moreover, it has been shown that the engineering effort required to modify Ocarina in order for it to support privileged partitions and virtual machines is low.

Also in this chapter, it has been shown that, at least, for all the configuration developed for this

thesis, AADL always leads to lower development effort, when compared with manual implementation, which should lead to lower time-to-market and costs.

5. Functionality Farming

5.1. Introduction

As explained in *Section 1.3. Functionality Farming*, nowadays, most operating systems follow a monolithic architecture; a monolithic architecture, however, leads to a large kernel, which in turn, leads to low reliability, weak security, high certification effort, as well as poor predictability and scalability. To reduce the size of the kernel, some authors propose the use of architectures based on: (1) a virtual machine monitor (or hypervisor), or (2) a microkernel, or (3) a combination of the two (i.e., a microkernel with virtualization support). These architectures, however, depend on the development of a new kernel, and thus, of a significant upfront investment; if the development of the new kernel fails, the cost is huge. And, in the end, these architectures do not tackle the source of the problem, i.e., the large size of the kernel in commodity operating systems, and just work around it.

In this thesis, functionality farming is presented, which, instead of a new architecture, consists in partitioning existing kernels by (1) moving functionality out of the kernel and onto the application (or partition) level, to reduce the size of the kernel, and by (2) replacing the functionality being moved with remote procedure calls to the partition level, to bridge the gap between the kernel and the partition level. Unlike other works, functionality farming tackles the source of the problem (i.e., the large size of the kernel in commodity operating systems). It requires a lower upfront investment, as it enables a progressive reduction of the size of the kernel, instead of an all-or-nothing approach, and thus, it is a more agile approach since it enables some decisions to be postponed closer to delivery time when information about the system's requirements is more precise.

Time and space partitioning an existing kernel, nevertheless, is not an easy task. In some cases, because of a functionality's level of coupling with kernel or its functional requirements (e.g., compatibility with hardware-dependent, kernel-level software), ensuring that time and space partitioning is possible, and at the same time, fulfilling the functionality's functional requirements may depend on a significant engineering effort. To address this issue, functionality farming relies on various partition types. The different partition types each provide distinct levels of partitioning (e.g., from time-only partitioning to both time and space partitioning), as well as they fulfill different functional requirements (e.g., from compatibility with hardware-dependent, kernel-level software to compatibility with only hardware-independent software). These different partition types, not only increase the extent to which functionality farming is more easily accomplished, but also enable an even more progressive reduction of the size of the kernel. Thus, these different partition types

enable fast design space exploration, and reduce the associated risk.

In *Chapter 3. POK/rodosvisor*, it has been described how POK has been extended to support three partition types, becoming POK/rodosvisor, namely: privileged partitions, virtual machines and ARINC 653 partitions.

Privileged partitions enforce time-only partitioning and provide compatibility with legacy software. As explained in *Section 3.3. Privileged Partitions*, a privileged partition runs at the kernel-level and thus, also provides compatibility with kernel-level functionality. Being compatible with kernel-level functionality, the effort of porting legacy kernel-level functionality onto a privileged partition should be low. Privileged partitions, however, by running at the kernel-level are still part of the kernel. Furthermore, as part of the kernel, only a select few failures can be contained; nevertheless, in those cases, only the faulty partition needs to be restarted, instead of the entire kernel.

Virtual machines enforce time and space partitioning and still provide compatibility with legacy software (as explained throughout *Chapter 2. Rodosvisor*, and in *Section 3.4. POK and Rodosvisor Integration*). Virtual machines, through virtualization, provide compatibility with kernel-level functionality and thus, similarly to privileged partitions, the effort of porting legacy functionality onto a virtual machine should be low. Unlike privileged partitions, however, virtual machines are space partitioned and thus, are not part of the kernel. Additionally, because virtual machines are space partitioned, any failure can be contained and, if detected, only the faulty virtual machine needs to be restarted. The downside to virtual machines is the virtualization overhead and, consequently, lower performance than privileged partitions.

Finally, ARINC 653 partitions enforce not only time and space partitioning, but also hardware-independence. The end goal of functionality farming is to move most of the kernel functionality onto ARINC 653 partitions; however, in some cases, because of their complexity or size, it is hard to move functionality directly from the kernel onto ARINC 653 partitions; moving functionality from the kernel onto privileged partitions or virtual machines, nevertheless, is easier.

Functionality farming, despite the benefits, still depends on a significant engineering effort, as will be shown later, and its effects are often very hard to predict, meaning that the associated risk is still high.

This chapter presents FF-AUTO, a tool which performs functionality farming semi-automatically in POK/rodosvisor. With FF-AUTO, the engineering effort, and thus, the risk associated with functionality farming is significantly reduced, making it also an ideal tool for design space exploration.

This chapter also demonstrates how functionality farming is able to improve the design and the performance of POK/rodosvisor, as well as how FF-AUTO enables a significant reduction of the required engineering effort. Currently, we are unable to demonstrate a reduction of the size of the kernel since POK/rodosvisor is already a very small kernel (very close to microkernel). Even though functionality farming and FF-AUTO, described here, are currently targeted towards only POK/rodosvisor, its underlying methodology can be applied to any other operating system.

5.1.1. Chapter Organization

This chapter is organized as follows.

In the following section, *Section 5.2. Related Work*, an overview of related work is given.

In *Section 5.3. Functionality Farming Automated: ff-auto*, FF-AUTO and its underlying methodology are presented.

In *Section 5.4. Use Case: Serial Port Device Driver* and *Section 5.5. Use Case: Inter-Partition Communication Subsystem*, two use cases are presented, demonstrating that functionality farming is able to improve the design and the performance of POK/rodosvisor, and that FF-AUTO contributes to a significant reduction of the required engineering effort, and thus, of the associated risk.

Finally, in *Section 5.6. Future Work* and *Section 5.7. Summary*, future work is proposed, and a summary of this chapter is given, respectively.

5.2. Related Work

Functionality farming is loosely based on the concept of task farming. Task farming consists on the decomposition of computations into identical and independent serial tasks, which are then executed by different processor cores in a multicore processor or in multi-processor systems such as computational grids [109]. Task farming is suited for computations such as a Montecarlo simulations in which the same model is run many times but with different start points (or inputs). A task farm is generically composed by:

1. The farmer, responsible for distributing the input to a pool of tasks and for retrieving and merging the output;
2. A pool of identical tasks, which perform the actual computation in parallel.

Similarly, functionality farming consists on the decomposition/partitioning of the kernel into partitions which perform the actual computation (not necessarily in parallel). Continuing with this analogy, then, the kernel is the farmer, and partitions are the pool of identical tasks.

To reduce the size of the trusted computing base (TCB), and thus, to improve security and reduce the certification effort, some authors propose the use of architectures based on: (1) a virtual machine monitor (or hypervisor), or (2) a microkernel, or (3) a combination of the two (i.e., a microkernel with virtualization support), in which the size of the kernel is much smaller than the kernel found in most commodity operating systems. More specifically, Hohmuth et al. [42] and Garfinkel et al. [44] propose an architecture based on a hypervisor. In particular, Hohmuth et al. use paravirtualization to achieve an even smaller size of the kernel (and thus of the TCB); to accomplish communication between virtual machines, in the approach used by Garfinkel et al., the hypervisor fully virtualizes one or more network devices, while Hohmuth et al. used paravirtualization to achieve the same result but with a smaller kernel (at the cost, however, of requiring modification to legacy software). Similarly, Liu et al. [46] propose an architecture based on a hypervisor to better partition the system across the cores of a multicore processor system. On the other end, Heiser et al. [41] propose a microkernel architecture, and Heiser et al. [47] propose an architecture based on a microkernel with virtualization support. In these architectures, commodity operating systems are pushed onto a virtual machine (on a hypervisor-based architecture), or onto one or more user-level servers (on a microkernel-based architecture), reducing the effects that a compromised commodity operating system can have on the system as a whole. Alongside the operating system, critical services are deployed on other virtual machines (or user-level servers), which depend on a much smaller TCB than that in a commodity operating system. These architectures, however, depend on an additional level of indirection which leads to poor performance. Furthermore, on a hypervisor-based architecture in particular, virtual machines are often coarse-grained and heavyweight leading to high resource usage. On a microkernel architecture, on the other end, there is no compatibility with legacy software, and the porting effort can be very significant. An architecture based on a "microkernel with virtualization support" solves the issues with the other two architectures, at the cost, however, of a larger size of the kernel. In the end, these architectures depend on the development of a new kernel, and thus, on a significant upfront investment; if the development of the new kernel fails, the cost is huge. These architectures do not tackle the source of the problem, i.e., the large size of the kernel in commodity operating systems, and just work around it. Conversely, functionality farming tackles the source of the problem by enabling a progressive reduction of the size of the kernel, and requires a lower upfront investment. Moreover, through FF-AUTO, the associated engineering effort is significantly reduced, facilitating even further the application of functionality farming.

Similarly to FF-AUTO, other works propose the improvement of existing software through

refactoring. For example: [110]–[113] focus on the parallelization of existing sequential programs, [114]–[117] on improving security, [118] on improving modularity, and [119] on enabling reentrancy, among others. Similarly to our work, these works enable a reduction of the engineering effort required to implement and evaluate different design alternatives, and thus, also reduce the associated risk. These works, however, imply modifications to the original source code, and require knowledge of the implementation's details. Moreover, these works are limited to hardware-independent applications. Our work, on the other end, instead of transformations at the level of the source code, relies on link time transformations. The original source code is not modified, and only the knowledge of external control and data dependencies is required. Lastly, our work addresses the requirements of hardware-dependent, kernel-level software, and thus, it is not limited to hardware-independent application.

5.3. Functionality Farming Automated: FF-AUTO

As shown in Figure 5.1, FF-AUTO requires as input: (1) an AADL model, specifying the reference configuration, and (2) a functionality farming configuration (FFC) file, which specifies the functions that should be farmed and how, based on the results from a profiler, by manual inspection, etc. As output, FF-AUTO generates a POK/rodosvisor configuration (i.e., a modified configuration) which is the result of applying functionality farming to the reference POK/rodosvisor configuration specified by the input model.

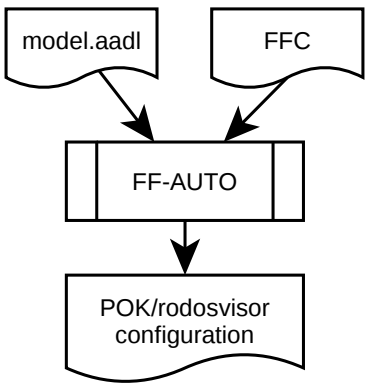


Figure 5.1. Inputs and outputs of FF-AUTO.

The modified POK/rodosvisor configuration needs, in some cases, to be manually modified as FF-AUTO is unable to automatically derive all of the necessary parameters, as will be explained later. In most cases, however, only the scheduler's configuration needs to be updated. In the future, we

expect to enable the specification of the modified scheduler's configuration in the FFC file. Similarly, for optimization purposes, the output from FF-AUTO may be manually modified.

In the following subsections, the FFC file format and the transformations applied to the reference POK/rodosvisor configuration are described.

5.3.1. Functionality Farming Configuration File

The FFC file format consists of a list of statements preceded by "%" followed by a list of function prototypes, as shown in Listing 5.1. The following statements are required:

system: this statement requires three parameters separated by a semicolon. The first parameter specifies the name of the AADL system implementation containing the AADL processor which is being farmed. As explained in *Section 4.3. Architecture Analysis & Design Language*, in AADL, a processor represents a combination of software and hardware responsible for scheduling threads, enforcing protected address spaces, among other things. In other words, an AADL processor corresponds to the operating system, including the underlying hardware platform. In Ocarina, it corresponds to a POK/rodosvisor kernel. The second parameter specifies the implementation type name of the AADL processor to be farmed. The third parameter specifies the instance name of the AADL processor to be farmed in the AADL system specified as the first parameter. This statement can appear only once. For example, the system statement for the model in Listing 5.2 is: "%system example::node.impl; example::pok.impl; cpu". In this example, the instance of "example::pok.impl", named "cpu" in "example::node.impl", is the processor to be farmed.

```
1 %system example::node.impl; example::pok.impl; cpu
2 %worker worker_1; privileged
3 %worker worker_2; vm
4 %worker_thread worker_thread_1; worker_1
5 %worker_thread worker_thread_2; worker_2
6 void pok_port_flushall(); worker_thread_1; call-only
7 void pok_cons_write(char*, int); worker_thread_2
```

Listing 5.1. A functionality farming configuration file.

```
 1 package example
 2 public
 3
 4 processor pok
 5 end pok;
 6
 7 processor implementation pok.impl
 8 end pok.impl;
 9
10 system node
11 end node;
12
13 system implementation node.impl;
14 subcomponents
15     cpu : processor pok.impl;
16 end node.impl;
17
18 end example;
```

Listing 5.2. An incomplete AADL model.

worker: two parameters separated by a semicolon. The first parameter specifies a worker's identifier, and the second parameter specifies its type. Worker is a synonym of partition; we use “worker” to distinguish the partitions in the reference configuration (partitions) and the partitions involved in functionality farming (workers). Worker threads, described below, that should be assigned to this particular worker must use the identifier specified as the first parameter. Currently, there are three types of workers supported, namely: “arinc653” for a worker based on an ARINC 653 partition; “privileged” for a worker based on a privileged partition; and “vm” for a worker based on a virtual machine. This statement can be repeated more than once, as necessary. For example, “%worker worker_1; privileged”, specifies a worker based on a privileged partition, identified as “worker_1”.

worker_thread: this statement requires two parameters separated by a semicolon. The first parameter specifies an identifier for a worker thread. The second parameter specifies the worker's identifier with which the worker thread is assigned to. Functions that should be assigned to this particular worker thread must use the identifier specified as the first parameter. This statement can

be repeated more than once, as necessary. For example, “%worker_thread worker_thread_1; worker_1”, specifies a worker thread identified as “worker_thread_1” and assigned to the worker identified as “worker_1”.

The list of statements just described is followed by a list of function prototypes. A function prototype is specified using the syntax of the C programming language, followed by a semicolon and the identifier of the worker thread which the function is assigned to. Optionally, a second semicolon followed by the keyword “call-only” can be specified, as shown in Listing 5.1, line 6. The “call-only” keyword indicates that only the function call, and not its implementation should be farmed. Function call farming and its motivation are explained in the following section.

5.3.2. Function Call Farming and Complete Farming

Some partition (or worker) types are unable to support specific kinds of functionality, and therefore, the implementation of some functions cannot be moved to them. To address this limitation, function call farming can be performed instead. With function call farming, the function's implementation remains in the kernel, and only the “function call” is moved. With function call farming, it cannot be expected that the size of the kernel will ever be reduced, because the function's implementation is not moved; however, as will be shown later, function call farming alone can still reveal good design alternatives. And, after finding a good design alternative, the function's implementation can be modified so as to enable complete farming (i.e., where the function's implementation is actually moved to the partition).

More specifically, ARINC 653 partitions cannot support hardware-dependent functionality; therefore, when farming hardware-dependent functionality onto an ARINC 653 partition, function call farming must be specified. A privileged partition, on the other end, because it runs at the same level as, and with the same level of privilege as the kernel, farming any functionality onto a privileged partition means that the function's implementation will remain in the same address space, and therefore, only function call farming is ever performed. A virtual machine supports both hardware-independent and hardware-dependent functionality, which means it is able to support complete farming of all kinds of functionality; however, with some exceptions, as explained next.

Independently of the partition type, currently, our methodology does not support moving functionality with kernel dependencies (i.e., which requires access to kernel data), and thus, for those kinds of functionality, function call farming needs to be specified. In the future, we expect to use existing code rewriting or binary translation techniques to automatically replace direct accesses to kernel data with other mechanism such as, for example, system calls.

5.3.3. Workers and Worker Threads

For each worker specified in the FFC file a worker is added to the reference configuration. The type of the worker is as specified in the FFC file. Similarly, worker threads are added to the corresponding workers, as specified in the FFC file. This is illustrated by the transformation between (a) and (b) in Figure 5.2.

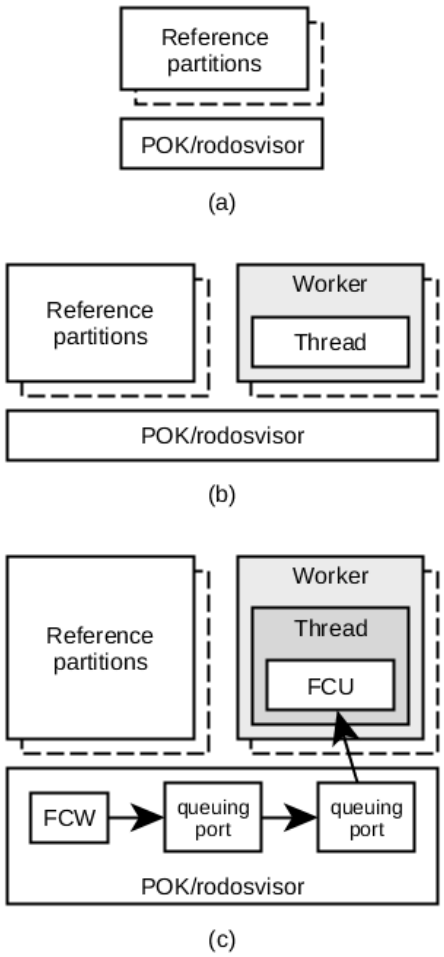


Figure 5.2. Illustration of the transformations applied to the reference configuration by FF-AUTO: (a) the reference configuration; (b) workers and worker threads added to the reference configuration; (c) FCW, FCU, and associated communication channels added to the reference configuration.

For functions which require access to I/O and which have been assigned to a virtual-machine-based worker, the output from FF-AUTO needs to be manually modified to update the virtual machine's configuration such that it can have access to the required I/O. In the future, it is expected that a function's I/O requirements could be derived automatically using methods such as those described

in [120]. The output from FF-AUTO also needs to be manually modified in order to update the scheduler's configuration to accommodate the new workers and worker threads, as explained earlier. In any case, as will be shown in *Section 5.4. Use Case: Serial Port Device Driver* and *Section 5.5. Use Case: Inter-Partition Communication Subsystem*, the required modifications are very small.

5.3.4. Function Call Wrappers and Unwrappers

For each function prototype specified in the FFC file, a function call wrapper (FCW) and a corresponding function call unwrapper (FCU) is generated. Additionally, in the kernel, function calls to the functions specified in the FFC file are replaced with function calls to the corresponding FCW. Finally, for those functions specified for complete farming, the functions' implementation is moved from the kernel onto the specified worker. This is illustrated by the transformation between (b) and (c) in Figure 5.2 (shown in the previous section). For those functions specified for function call farming, on the other end, the functions' implementation remains in the kernel.

A FCW, as shown in Figure 5.3, serializes function calls through a communication channel, and returns immediately after. A FCU, conversely, as shown in Figure 5.4: (1) deserializes function calls from a communication channel, and (2) jumps to the function's implementation. Return values are currently not supported. Serialization and deserialization of function calls relies on a data structure declaration that is generated based on the function's prototype. In POK/rodosvisor, communication channels are queuing communication channels, with one sending port (used by the FCW), and one receiving port (used by the corresponding FCU), as illustrated in Figure 5.2(c). For each function prototype specified in the FFC file, currently, a dedicated communication channel is established. In the future, it is expected that some functions will be able to share the same communication channel, forming a function group, and thus, lowering resource usage.

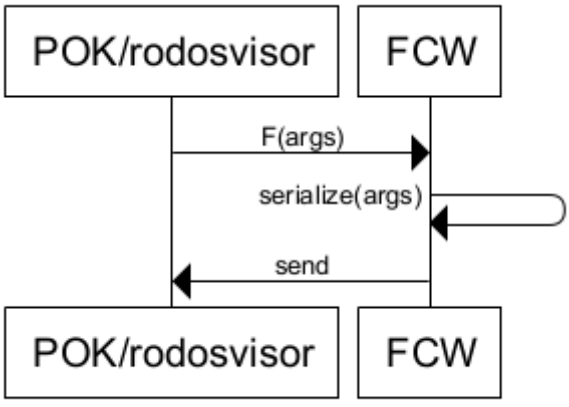


Figure 5.3. Sequence diagram of a generic function call wrapper (FCW).

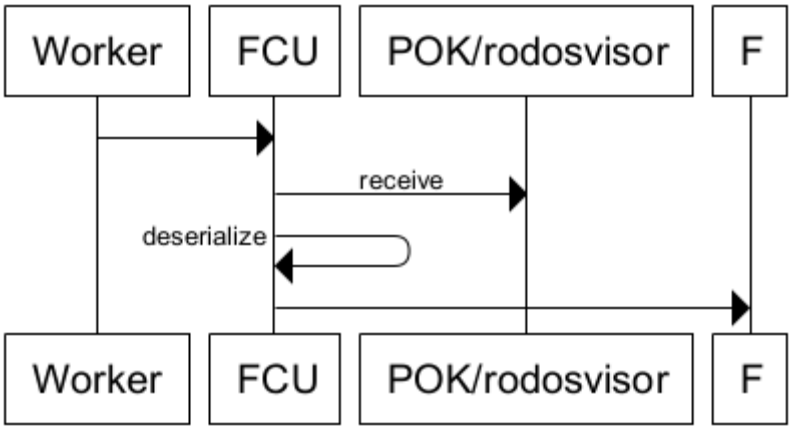


Figure 5.4. Sequence diagram of a generic function call unwrapper (FCU).

The process of checking for and receiving new function calls depends on the type of the worker. For workers based on ARINC 653 or privileged partitions, function calls are received using the communication services provided by the ARINC 653 APEX to which they have access by default. For workers based on virtual machines, which by default do not have access to the APEX, a dedicated hypercall controller is generated which, indirectly, enables a virtual machine to access the communication services provided by the APEX, and thus, receive function calls.

Similarly, the process of jumping to a function's implementation depends on whether or not its implementation has been moved from the kernel onto the specified worker, and it depends on the specified worker's type. For those functions whose implementation is moved from the kernel onto the specified worker (i.e., complete farming), jumping to its implementation is performed directly. On the other end, for those functions whose implementation is not moved from the kernel to the specified worker (i.e., function call farming), jumping to its implementation, which remains in the kernel, depends on the specified worker's type. If the worker is an ARINC 653 partition, then, support for a dedicated system call is added to the kernel so that the worker can request the kernel to jump to the function's implementation. If, however, the worker is a privileged partition, then: (1) preemption is disabled, (2) a direct jump to the function's implementation is performed, and (3) when the function's implementation returns, preemption is enabled again. Preemption is disabled in order to prevent the kernel from becoming in a corrupt state. Furthermore, a privileged partition is part of the kernel and, therefore, it can jump to the function's implementation directly. Lastly, if the worker is a virtual machine, then, an hypercall is used to request the virtual machine's hypercall controller to jump to the function's implementation. A hypercall controller is part of the hypervisor,

which is a part of the kernel, and thus, has access to the function's implementation. A dedicated hypercall controller is automatically generated whenever necessary.

5.4. Use Case: Serial Port Device Driver

In this section, first, a POK/rodosvisor configuration (i.e., the reference configuration) is described and it is demonstrated that it reveals a limitation in the design of POK/rodosvisor, more specifically, in the design of the serial port device driver. Second, it is described how functionality farming has been applied to the reference configuration and how it is expected to address the limitation identified earlier. Third and last, it is demonstrated that functionality farming addresses that limitation by comparing the performance before and after functionality farming. Even though functionality farming is used to address a limitation in the design of POK/rodosvisor, we do not claim it is the only or the best way to do it; our goal is only to demonstrate a possible use case for functionality farming.

The reference architecture is illustrated in Figure 5.5. It consists of a POK/rodosvisor-based system with one ARINC 653 partition composed by one thread (the writer). As illustrated in Figure 5.6, for each partition window, the writer sends data to the serial port, through an ARINC 653 virtual queuing port which, in turn, forwards all data to the serial port. After sending the data, the writer reports the time (as number of CPU clock cycles) required to so, and then goes idle until the next partition window.

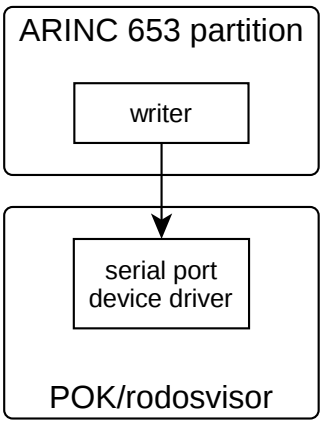


Figure 5.5. The reference architecture used for the serial port device driver's use case: an ARINC 653 partition which sends data through the serial port.

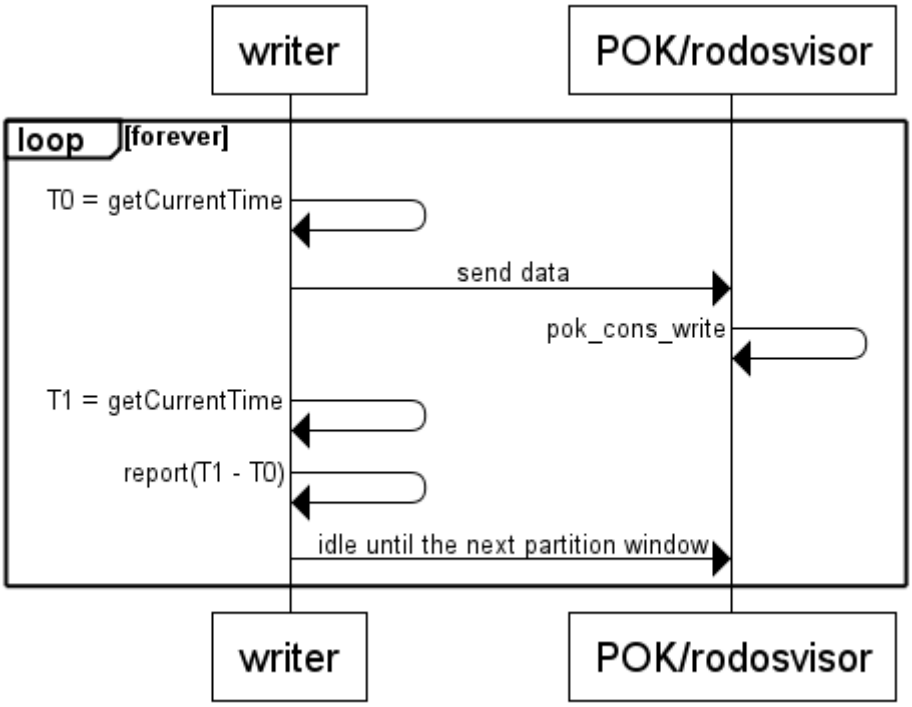


Figure 5.6. A sequence diagram illustrating the process performed by the writer in the serial port device driver's use case.

From the reference configuration, the average number of CPU clock cycles per byte required to send data to the serial port for various sizes of data have been measured. For each data size, the configuration ran until 100 samples have been collected; the average number of CPU clock cycles was obtained by averaging all the samples.

Figure 5.7 shows, for the reference configuration, the average number of CPU clock cycles per byte required to send data to the serial port for different sizes of data. It can be seen that, the larger the data, the higher the number of CPU clock cycles per byte, indicating that sending data to the serial port does not scale well.

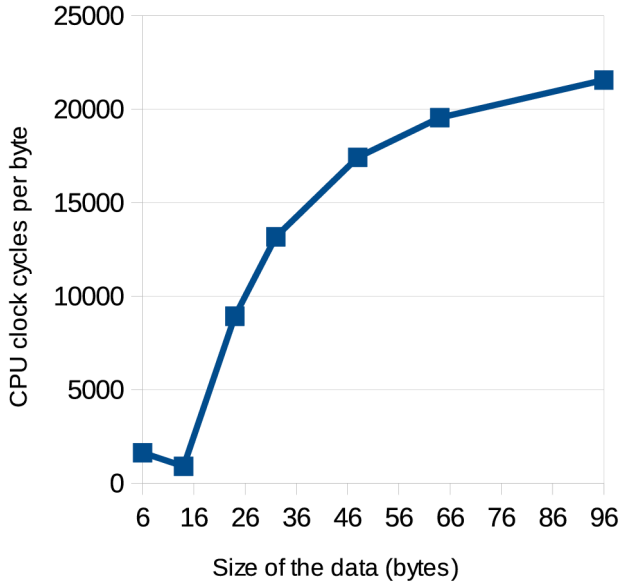


Figure 5.7. The average number of CPU clock cycles per byte required to send data to the serial port for different sizes of data for the reference configuration.

Further investigation revealed that the lack of scalability identified above could be traced back to a function in POK/rodosvisor named “pok_cons_write”, illustrated in Figure 5.6. In POK/rodosvisor, “pok_cons_write” is the function which interacts directly with, and sends data to the serial port. Once the serial port's transmit buffer (i.e., a 16-byte buffer) becomes full, “pok_cons_write” busy-waits until the buffer becomes available before sending more data. This means that, when the size of the data is larger than the serial port's transmit buffer, “pok_cons_write” needs to constantly busy-wait until all data is sent. Taking into account that the rate at which the serial port dispatches data into the transmission line is very slow compared to the CPU, then, as illustrated in Figure 5.7, when the size of the data is larger than the serial port's transmit buffer, the impact on the number of CPU clock cycles required to send the data is significant.

Hence, the application of functionality farming has been considered. More specifically, to farm “pok_cons_write” into a dedicated worker. In this way, “pok_cons_write”, instead of interacting directly with, and sending data to the serial port, sends data to a sending queuing port, which can feature a much larger buffer and is also much faster than the serial port. The worker, on the other end, reads data from a receiving queuing port and, only then, sends it to the serial port. Because the worker is assigned with a dedicated execution time in the major frame, the lack of scalability when sending data to the serial port does not affect the rest of system.

Functionality farming has been applied using FF-AUTO and three FFC files, which specify that “pok_cons_write” shall be farmed into one worker with one worker thread. One of the FFC files is shown in Listing 5.3 and it specifies a virtual-machine-based worker (configuration VM); the resulting architecture is shown in Figure 5.8. The other two FFC files specified an ARINC-653-partition-based worker (configuration AP) and a privileged-partition-based worker (configuration PP). For configuration VM, complete farming has been specified. For configuration AP, on the other end, function call farming has been specified; “pok_cons_write” is hardware-dependent and, as explained in *Section 5.3. Functionality Farming Automated: ff-auto*, only function call farming is supported. For a privileged partition, configuration PP, only function call farming is supported. The output from FF-AUTO (i.e., a modified POK/rodosvisor configuration), for all configurations has been manually modified in order to accommodate the worker and its worker thread in the scheduler's configuration. For configuration VM, additionally, the virtual-machine-based worker's configuration has been manually modified in order to enable access to the serial port hardware, as required by “pok_cons_write”.

```
1 %system test::node.impl; test::ppc.impl; cpu
2 %worker worker_1; vm
3 %worker_thread worker_thread_1; worker_1
4 void pok_cons_write(char*, int); worker_thread_1
```

Listing 5.3. FFC file for farming “pok_cons_write” on a worker based on a virtual machine.

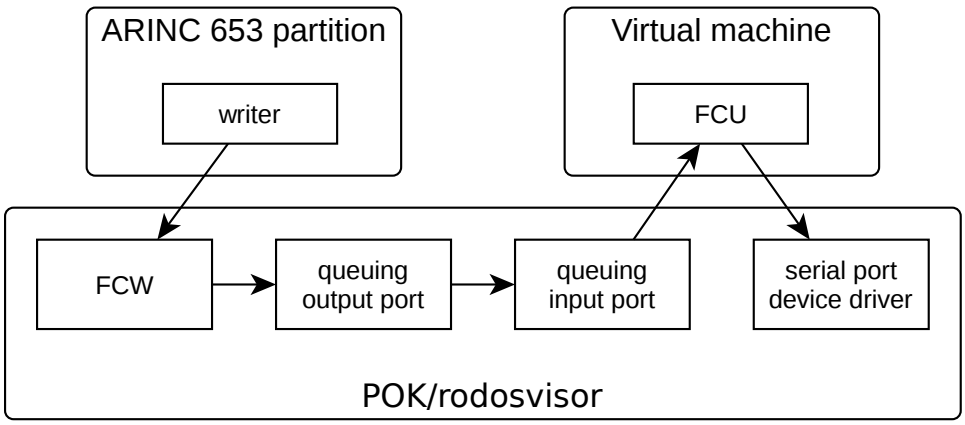


Figure 5.8. The resulting architecture for configuration VM on the serial port device driver's use case.

Similarly to the reference configuration, from all the modified configurations described above, the average number of CPU clock cycles per byte required to send data to the serial port for various sizes of data has been measured. The results are presented in Figure 5.9. It can be seen that, after functionality farming, for data sizes larger than the serial port's transmit buffer, as the size of the data increases, the cost per byte decreases, indicating that scalability is good. Furthermore, it can be seen that the different configurations display very approximate results.

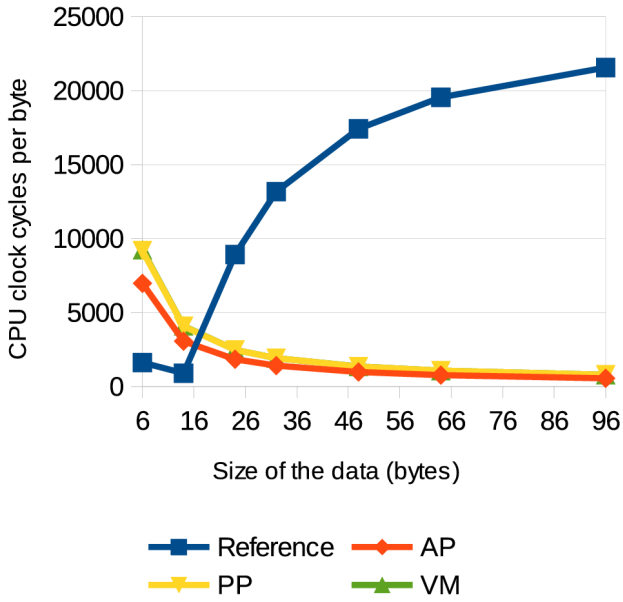


Figure 5.9. Comparison of the average number of CPU clock cycles per byte required to send data to the serial port for different sizes of data between the reference configuration and after functionality farming. "VM" is barely seen because it is overlapped by "PP."

In Figure 5.10, the kernel's footprint for the reference and all the modified configurations is presented. It can be seen that the footprint of all the modified configurations is higher than the reference configuration's. This was expected for configurations AP and PP, since the function's implementation is not moved out of the kernel; the added footprint is due to a larger size of the code, and a larger size of the stacks due to the additional partition/worker. For configuration VM, where the function's implementation is moved out of the kernel, the added footprint is much higher than the size of the function's implementation, and thus, overall, the footprint is higher than the reference configuration's; the added footprint is mostly due to the hypervisor (code, read-only, and read/write data), and due to a larger size of the stacks.

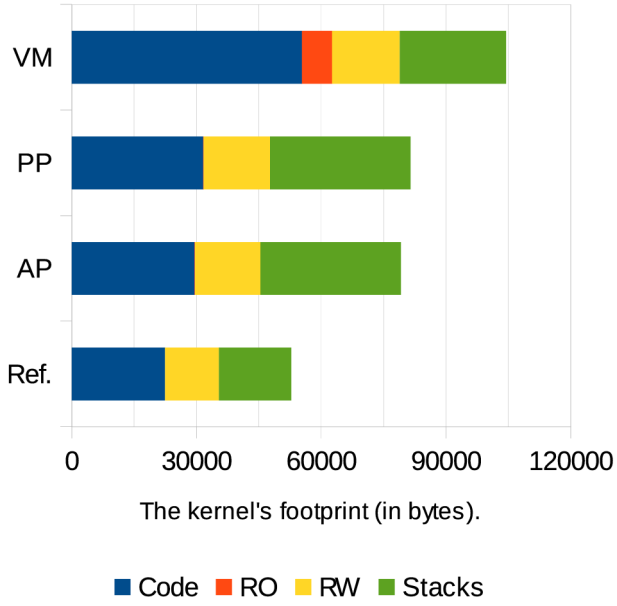


Figure 5.10. The kernel's footprint for the reference and all the modified configurations in the serial port device driver's use case, in terms of the size of the code, read-only (RO) and read-write (RW) data, as well as the size of the stacks.

Lastly, in terms of the engineering effort, the reference configuration (i.e., the output from Ocarina) consists of 393 source lines of code (SLOC). As explained in *Section 4.4. Ocarina*, a POK/rodosvisor configuration is composed by the implementation and configuration of a POK/rodosvisor kernel, the implementation and configuration of the partitions/workers, as well as the implementation of the configuration's build system. The three modified configurations (i.e., the output from FF-AUTO) consist of at least 369 new/modified SLOC when compared with the reference configuration. Using FF-AUTO, 4 SLOC were required for the FFC file and, in the worst case, an additional 14 new/modified SLOC were also required. Knowing that, if functionality farming was performed manually, 369 new/modified SLOC would be required, and that, using FF-AUTO, only 18 SLOC were required, then, FF-AUTO enabled a reduction of engineering effort by more than 20 times.

5.5. Use Case: Inter-Partition Communication Subsystem

In this section, similarly to the previous section, first, a POK/rodosvisor configuration (i.e., the reference configuration) is described and it is demonstrated that it reveals a limitation in the design of POK/rodosvisor's inter-partition communication subsystem. Second, it is described how functionality farming has been applied to the reference configuration, and how it is expected to

address the limitation identified earlier. Third and last, it is demonstrated that functionality farming addresses that limitation by comparing the performance before and after functionality farming. Even though functionality farming is used to address a limitation in the design of POK/rodosvisor, we do not intend to claim that it is the only or the best way to do it. Our goal is only to demonstrate another use case for functionality farming.

The reference configuration is illustrated in Figure 5.11. It consists of a POK/rodosvisor-based system with two ARINC 653 partitions which communicate with each other through a single queuing communication channel. For each major frame, the “sender” sends data by writing to a sending queuing port, while the “receiver” receives data by reading a corresponding receiving queuing port. The size of the data is never larger than the size of the sending and receiving ports' buffer size.

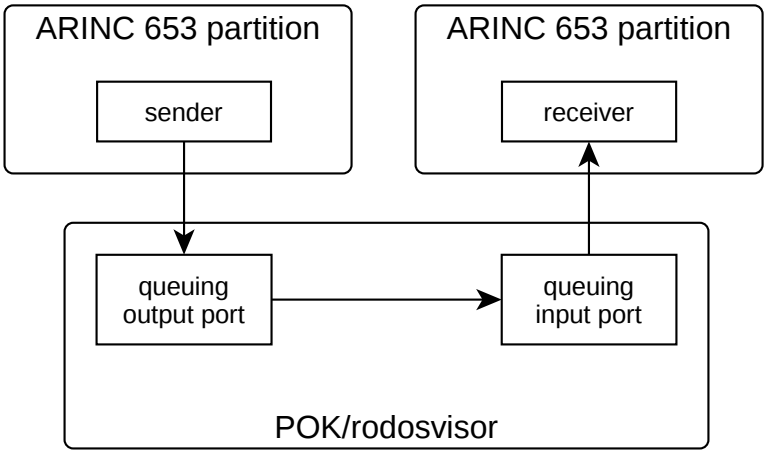


Figure 5.11. The reference architecture used in the inter-partition communication subsystem's use case: two ARINC 653 partitions communicating through a queuing channel.

From the configuration just described, the scheduling jitter has been measured for different sizes of the data that are transmitted between “sender” and “receiver.” To measure the scheduling jitter, the output of the partitions' context switch times was enabled and the configuration ran until 300 samples were collected. To obtain the scheduling jitter, the expected context switch times were subtracted from the measured context switch times. The average scheduling jitter was obtained by averaging the results from all the samples.

Figure 5.12 shows the scheduling jitter for the “receiver” and the “sender,” for different sizes of transmitted data. It can be seen that, the larger the size of the data, the larger the scheduling jitter of

the “receiver.” The scheduling jitter of the “sender,” on the other end, is independent of the size of the data.

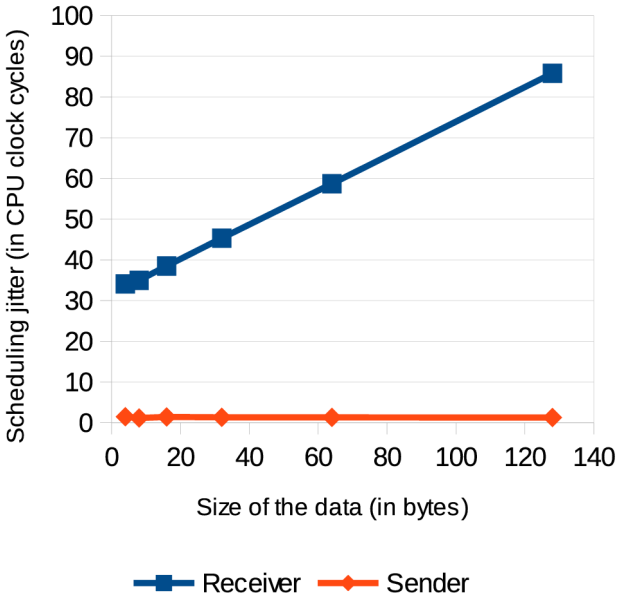


Figure 5.12. The average scheduling jitter for the “receiver” and the “sender.”

Further investigation revealed that the issue identified earlier could be traced back to a function in POK/rodosvisor named “pok_port_flushall”. This function moves (i.e., flushes) the data in all sending ports into the corresponding receiving ports; the higher the size of transmitted data, the higher its execution time. It is called at the beginning of a major frame, and thus, at the beginning of the first partition window in the major frame (i.e., the partition window of the “receiver”). This approach provides a predictable execution time when sending or receiving data, especially when an output port is connected to multiple input ports, as it is only necessary to copy data to or from a buffer in the kernel. The execution time of “pok_port_flushall”, however, overwrites the first partition window in the major frame, which as demonstrated leads to high scheduling jitter for the “receiver.”

To solve this problem, the application of functionality farming has been considered. More specifically, to farm “pok_port_flushall” into a dedicated worker, and allocate it into a predefined slot in the major frame, such that it does not overwrite other partitions' execution time.

Functionality farming has been applied using FF-AUTO and three FFC files, which specify that “pok_port_flushall” shall be farmed into one worker with a single worker thread. One FFC file,

shown in Listing 5.4, specifies a worker based on an ARINC 653 partition (configuration AP); the resulting architecture is shown in Figure 5.13. The other two FFC files specified a privileged-partition-based worker (configuration PP) and a virtual-machine-based worker (configuration VM). Since “pok_port_flushall” has shared dependencies with the kernel, as explained in *Section 5.3. Functionality Farming Automated: ff-auto*, function call farming as been specified for all configurations. For all configurations, the modified POK/rodosvisor configuration (i.e., the output from FF-AUTO) has been manually modified in order to accommodate the worker and its worker thread in the scheduler's configuration. Furthermore, because “pok_port_flushall” is part of the implementation of the inter-partition communication subsystem, on which functionality farming depends on, the output from FF-AUTO has also been manually modified such that, instead of a queuing communication channel, counting semaphores are used to serialize calls to “pok_port_flushall”, requiring 15 new/modified SLOC. Using a counting semaphore to serialize calls is possible because “pok_port_flushall” accepts no parameters.

```
1 %system test::node.impl; test::ppc.impl; cpu
2 %worker worker_1; arinc653
3 %worker_thread worker_thread_1; worker_1
4 void pok_port_flushall(); worker_thread_1; call-only
```

Listing 5.4. FFC file for farming “pok_port_flushall” on a worker based on an ARINC 653 partition.

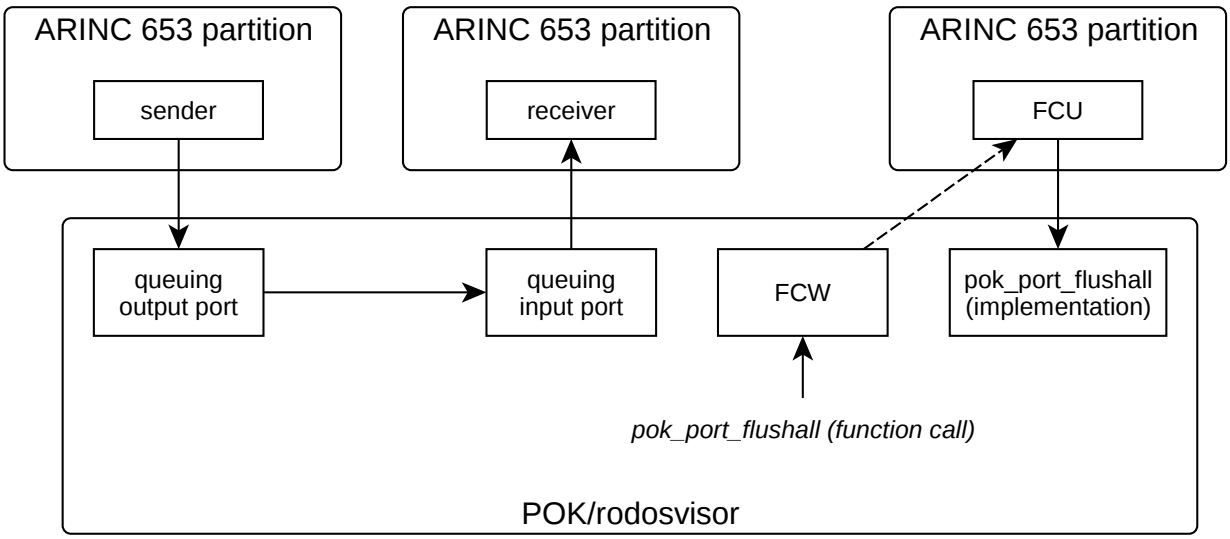


Figure 5.13. The resulting architecture for configuration AP on the inter-partition communication subsystem's use case.

Similarly to the reference configuration, for all the modified configurations described above, the scheduling jitter has been measured for different sizes of transmitted data. The results are shown in Figure 5.14. It can be seen that, after functionality farming, the scheduling jitter of the first partition in the major frame is equivalent to the scheduling jitter of other partitions in the major frame, and it is independent of the size of the data that is transmitted. These results demonstrate that functionality farming addressed the limitation identified earlier.

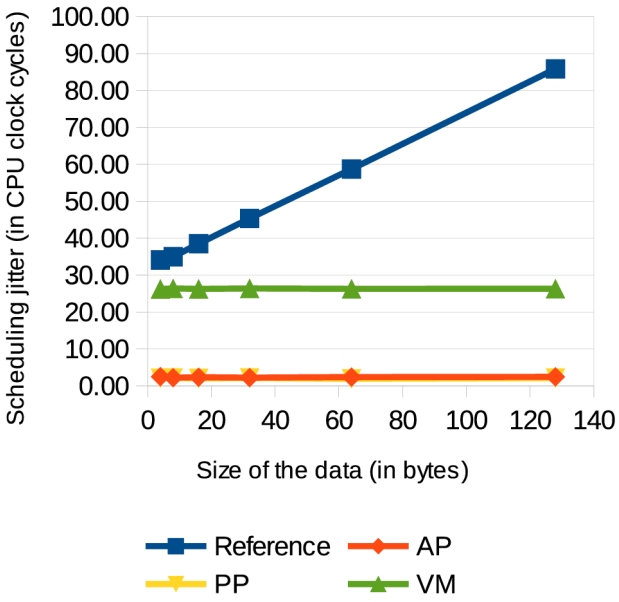


Figure 5.14. The average scheduling jitter for the first partition in the major frame on the reference and all the modified configurations. "PP" is barely seen as it is overlapped by "AP."

In Figure 5.15, the kernel's footprint for the reference and modified configurations is presented, in terms of the size of the code, read-only and read-write data, as well as the size of the stacks. It can be seen that, as expected, the footprint for all the modified configurations is higher than the reference configuration's footprint, since only function call farming has been performed. AP's and PP's larger footprint is mostly due to a larger size of the stacks because of an additional partition/worker. VM's large footprint, on the other end, is due to the size of hypervisor (code, read-only, and read-write data) as well as due to a slightly larger size of the stacks.

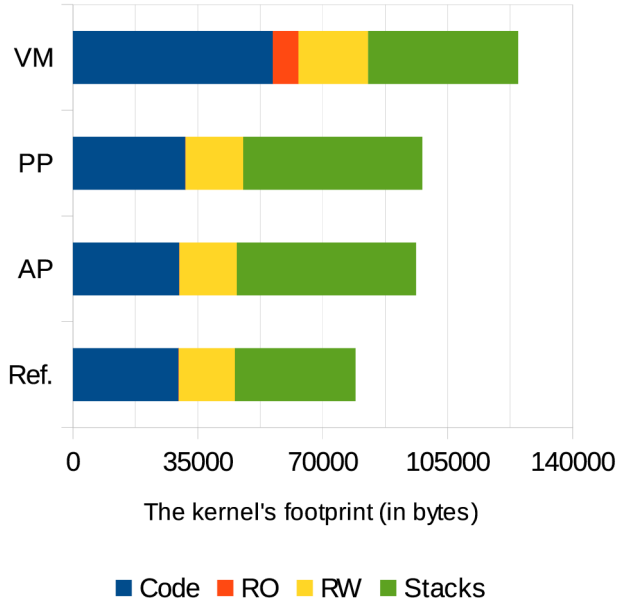


Figure 5.15. The kernel's footprint for the reference and all the modified configurations in the inter-partition communication subsystem's use case, in terms of the size of the code, read-only (RO) and read-write (RW) data, as well as the size of the stacks.

Finally, in terms of the engineering effort, the reference configuration consists of 761 source lines of code (SLOC). The three modified configurations, on the other end, consist of at least 279 new/modified SLOC when compared with the reference configuration. Using FF-AUTO, 4 SLOC were required for the FFC file and, in the worst case, an additional 19 new/modified SLOC were also required. Knowing that, if functionality farming was performed manually, 279 new/modified SLOC would be required, and that, using FF-AUTO, only 23 SLOC were required, then, FF-AUTO enabled a reduction of engineering effort by more than 10 times.

5.6. Future Work

As future work, we propose the following.

- Currently, the granularity that is supported is at the level of complete functions, including all callees. We expect to decrease the granularity to exclude some callees which otherwise limit the extent to which farming is possible using, for example, code rewriting techniques.
- Similarly, functions with shared dependencies with the kernel are not well supported; we expect to address this limitation using code rewriting techniques as well.
- Currently, the developer is responsible for understanding to what extend some functionality

can be farmed; we propose the development of an analysis tool which would enable such information to be obtained automatically.

- As explained earlier, the POK/rodosvisor configuration output by FF-AUTO needs, in some cases, to be manually modified as FF-AUTO is unable to derive all the necessary parameters. In the future, we propose to improve derivation of parameters and, wherever necessary, to extend the format of the FFC file, enabling it to specify some parameters which cannot be derived otherwise.
- Currently, the I/O requirements of a functionality need to be manually inspected and specified; we propose the use of methods such as those described in [120] to automatically derive a functionality's I/O requirements.
- We also propose the improvement of resource usage through function groups, which enable a single communication channel to be shared by several function call wrappers and unwrappers.
- At last, we propose to try the same approach on other operating systems, such as Linux-based operating systems, where the advantages can be even greater when compared with small-size POK/rodosvisor.

5.7. Summary

In this chapter, FF-AUTO, a tool which performs functionality farming semi-automatically in POK/rodosvisor, has been presented. Also in this chapter, two use cases which demonstrate how functionality farming is able to improve the design of POK/rodosvisor, and how FF-AUTO enables a significant reduction of the engineering effort required, have been presented.

6. Conclusion

6.1. Full Virtualization on Low-End Hardware

In this thesis it has been explained that, in the domain of safety-critical embedded systems, there is an ongoing transition from federated architectures to integrated architectures, and that the use of a hypervisor as the separation kernel is being considered. A hypervisor, similarly to a separation kernel, is capable of enforcing time and space partitioning; however, through virtualization, a hypervisor also provides compatibility with legacy software and thus, the porting and re-certification effort of legacy software from a federated architecture is lower. It has also been demonstrated that, however, most hypervisors nowadays either (1) rely on paravirtualization, or (2) depend on high-end hardware. Paravirtualization requires legacy software to be ported to a hypervisor-specific interface, leading to high porting and re-certification effort, and thus, longer time-to-market. High-end hardware, on the other end, does not satisfy the constrains associated with safety-critical embedded systems. Full virtualization on low-end hardware has none of these limitations. Therefore, the development of a hypervisor based on software-only full virtualization has been proposed, in order to:

- Evaluate the feasibility of full virtualization on low-end hardware to address the limitations of existing hypervisors. Low-end hardware full virtualization is able to provide compatibility with legacy software on low-end hardware; at the cost, however, of higher virtualization overhead.
- Understand the limitations of existing processor architectures for the realization of low-end hardware full virtualization so that these limitations can be addressed in the future. We believe that virtualization is going to be a recurring theme as long as the software complexity continues to increase, or as long as new processor architectures are released, or even as long as there is the need to consolidate legacy alongside new software.

In this thesis, performance and footprint measurements from POK/rodosvisor, featuring low-end hardware full virtualization, have been presented.

Compatibility with legacy software has been demonstrated by showing the results of some benchmarks executed on top of a Linux-based operating system as a guest on POK/rodosvisor.

The evaluation of the virtualization overhead, and POK/rodosvisor's performance profile showed that: low-end hardware full virtualization is more adequate for compute-intensive workloads, with moderate use of I/O, and with low use of CPU management operations.

In terms of footprint, it has been shown that the kernel's footprint for a virtual-machine-based system can actually be lower than the kernel's footprint for an ARINC-653-compliant system, which indicates that the requirements of low-end hardware full virtualization regarding the kernel's footprint are not high.

Altogether, we believe that these results demonstrate that, for many applications, low-end hardware full virtualization can be a serious alternative to high-end hardware full virtualization and paravirtualization, enabling:

- a reduction of system size, weight, power consumption, cost, etc., when compared with high-end hardware full virtualization;
- a reduction (in many cases, elimination) of the effort required to port legacy software to a hypervisor-specific interface, when compared with paravirtualization;
- as well as, in some cases, a reduction of the kernel's footprint, when compared with ARINC-653-based systems.

6.2. Model-Driven Engineering

In this thesis it has been explained that conventional development methods are unable to keep up with the requirements of nowadays and future safety-critical embedded systems. To address this problem, the approach taken by Ocarina (i.e., model-driven code generation) has been used to support the features developed for this thesis, namely: privileged partitions and virtual machines.

In this thesis it has been shown how AADL can be used to represent privileged partitions and virtual machines, and thus, that it can replace conventional development methods. At the same time, it has been explained how Ocarina has been modified to support those representation and to generate a POK/rodosvisor configuration accordingly. Finally, it also been demonstrated that, at least, for all the configuration developed for this thesis, in the worst case, AADL contributes to 56% lower development effort, and, in the best case, AADL contributes to less than 15% lower development effort.

6.3. Functionality Farming

In this thesis, it has also been explained that, nowadays, most operating systems follow a monolithic architecture which, however, leads to a large kernel, and is associated with the following drawbacks: low reliability, weak security, high certification effort, as well as poor predictability and scalability. To reduce the size of the kernel and address those drawbacks, some authors propose new

architectures. Those architectures, however, depend on the development of a new kernel, and thus, on a significant upfront investment; if the development of the new kernel fails, the cost is huge. Moreover, in the end, those architectures do not tackle the source of the problem, i.e., the large size of the kernel in commodity operating systems, and just work around it.

In this thesis, functionality farming has been presented, which, unlike other works, tackles the source of the problem (i.e., the size of the kernel in commodity operating systems). It requires a lower upfront investment, as it enables a progressive reduction of the size of the kernel, instead of an all-or-nothing approach, and thus, it is a more agile approach as it enables some decisions to be postponed closer to delivery time when information about the system's requirements is more precise.

Functionality farming alone, however, is not an easy task. In this thesis, FF-AUTO has also been presented, which performs functionality farming semi-automatically in POK/rodosvisor. With FF-AUTO, the engineering effort, and thus, the risk associated with functionality farming is significantly reduced, making it an ideal tool for design space exploration.

Finally, it has been demonstrated that functionality farming is able to improve the design and the performance of POK/rodosvisor, and that FF-AUTO enables a significant reduction of the required engineering effort.

Bibliography

[1] N. Storey, *Safety Critical Computer Systems*, 1 edition. Harlow, England ; Reading, Mass: Addison-Wesley, 1996.

[2] International Electrotechnical Commission, *IEC 61508:2010 Commented version, Functional safety of electrical/electronic/programmable electronic safety-related systems*. 2010.

[3] Radio Technical Commission for Aeronautics, *DO-178C Software Considerations in Airborne Systems and Equipment Certification*. 2011.

[4] International Organization for Standardization, *Road vehicles: Functional safety (ISO 26262)*. 2011.

[5] International Organization for Standardization, *Medical device software: Software life cycle processes (ISO 62304)*. 2006.

[6] “ISO 9000 quality management - ISO.” [Online]. Available: http://www.iso.org/iso/iso_9000. [Accessed: 23-Dec-2015].

[7] National Aeronautics and Space Administration (NASA), “Research Opportunities in Aeronautics,” Aug. 2011.

[8] C. B. Watkins and R. Walter, “Transitioning from federated avionics architectures to Integrated Modular Avionics,” in *Digital Avionics Systems Conference, 2007. DASC ’07. IEEE/AIAA 26th*, 2007, pp. 2.A.1–1–2.A.1–10.

[9] P. Parkinson, “Safety, Security and Multicore,” in *Advances in Systems Safety*, C. Dale and T. Anderson, Eds. Springer London, 2011, pp. 215–232.

[10] P. J. Prisaznuk, “Integrated modular avionics,” in *Aerospace and Electronics Conference, 1992. NAECON 1992., Proceedings of the IEEE 1992 National*, 1992, pp. 39–45.

[11] Radio Technical Commission for Aeronautics, “DO-297 Integrated Modular Avionics (IMA) Development Guidance and Certification Considerations.” [Online]. Available: http://www.rtca.org/store_product.asp?prodid=617. [Accessed: 22-May-2015].

[12] “Home : AUTOSAR.” [Online]. Available: http://www.autosar.org/. [Accessed: 07-Dec-2015].

[13] R. Obermaisser, P. Peti, B. Huber, and C. El Salloum, “DECOS: an integrated time-triggered architecture,” *e & i Elektrotechnik und Informationstechnik*, vol. 123, no. 3, pp. 83–95, 2006.

[14] Airlines Electronic Engineering Committee, *Draft 3 of Supplement 1 to ARINC Specification*

653: Avionics Application Software Standard Interface. 2003.

[15] M. Masmano, I. Ripoll, A. Crespo, and J.-J. Metge, “Xtratum: a hypervisor for safety critical embedded systems,” in *11th Real-Time Linux Workshop*, 2009.

[16] Lynx Software Technologies, Inc., “LynxSecure Separation Kernel Hypervisor | Real-Time Operating Systems and Virtualization Security: Lynx Software TechnologiesReal-Time Operating Systems and Virtualization Security: Lynx Software Technologies,” 2015. [Online]. Available: http://www.lynx.com/products/hypervisors/lynxsecure-separation-kernel-hypervisor/. [Accessed: 08-Jul-2014].

[17] SYSGO AG, “PikeOS Hypervisor,” 2015. [Online]. Available: http://www.sysgo.com/products/pikeos-rtos-and-virtualization-concept/. [Accessed: 07-Jul-2014].

[18] Green Hills Software, “INTEGRITY Multivisor,” 2015. [Online]. Available: http://www.ghs.com/products/rtos/integrity_virtualization.html. [Accessed: 10-Jan-2014].

[19] S. H. VanderLeest, “ARINC 653 hypervisor,” in *Digital Avionics Systems Conference (DASC), 2010 IEEE/AIAA 29th*, 2010, pp. 5.E.2–1–5.E.2–20.

[20] G. J. Popek and R. P. Goldberg, “Formal requirements for virtualizable third generation architectures,” *Commun. ACM*, vol. 17, no. 7, pp. 412–421, Jul. 1974.

[21] S. Trujillo, A. Crespo, and A. Alonso, “MultiPARTES: Multicore Virtualization for Mixed-Criticality Systems,” in *2013 Euromicro Conference on Digital System Design (DSD)*, 2013, pp. 260–265.

[22] IBM, *PowerPC 405-S Embedded Processor Core User’s Manual*, 1.2 ed. 2010.

[23] “POK homepage: home.” [Online]. Available: http://pok.safety-critical.net/. [Accessed: 10-Jan-2014].

[24] S. Campagna and M. Violante, “On the Evaluation of the Performance Overhead of a Commercial Embedded Hypervisor,” in *The First Workshop on Manufacturable and Dependable Multicore Architectures at Nanoscale (MEDIAN’12)*, 2012, pp. 59–63.

[25] B. Hailpern and P. Tarr, “Model-driven development: The good, the bad, and the ugly,” *IBM Systems Journal*, vol. 45, no. 3, pp. 451–461, 2006.

[26] J. Delange and L. Lec, “POK, an ARINC653-compliant operating system released under the BSD license,” in *13th Real-Time Linux Workshop*, 2011, vol. 10.

[27] H. A. Muller, R. J. Norman, and J. Slonim, *Computer Aided Software Engineering*. Springer Science & Business Media, 2012.

[28] D. C. Schmidt, “Guest Editor’s Introduction: Model-Driven Engineering,” *Computer*, vol. 39, no. 2, pp. 25–31, Feb. 2006.

[29] W. Kozaczynski and G. Booch, “Guest Editors’ Introduction: Component-Based Software Engineering,” *IEEE software*, no. 5, pp. 34–36, 1998.

[30] R. Van Ommering, F. Van Der Linden, J. Kramer, and J. Magee, “The Koala component model for consumer electronics software,” *Computer*, vol. 33, no. 3, pp. 78–85, 2000.

[31] F. van der Linden, K. Schmid, and E. Rommes, *Software Product Lines in Action: The Best Industrial Practice in Product Line Engineering*. Springer, 2007.

[32] A. Metzger and K. Pohl, “Software Product Line Engineering and Variability Management: Achievements and Challenges,” in *Proceedings of the on Future of Software Engineering*, New York, NY, USA, 2014, pp. 70–84.

[33] S. Kent, “Model driven engineering,” in *Integrated formal methods*, 2002, pp. 286–298.

[34] G. Lasnier, B. Zalila, L. Pautet, and J. Hugues, “Ocarina : An Environment for AADL Models Analysis and Automatic Code Generation for High Integrity Applications,” in *Reliable Software Technologies – Ada-Europe 2009*, F. Kordon and Y. Kermarrec, Eds. Springer Berlin Heidelberg, 2009, pp. 237–250.

[35] J. HUGUES, B. ZALILA, L. PAUTET, and F. KORDON, “Rapid Prototyping of Distributed Real-Time Embedded Systems Using the AADL and Ocarina,” in *Rapid System Prototyping, IEEE International Workshop on*, Los Alamitos, CA, USA, 2007, vol. 0, pp. 106–112.

[36] J. Hugues, B. Zalila, L. Pautet, and F. Kordon, “From the Prototype to the Final Embedded System Using the Ocarina AADL Tool Suite,” *ACM Trans. Embed. Comput. Syst.*, vol. 7, no. 4, pp. 42:1–42:25, Aug. 2008.

[37] B. Zalila, L. Pautet, and J. Hugues, “Towards Automatic Middleware Generation,” in *2008 11th IEEE International Symposium on Object Oriented Real-Time Distributed Computing (ISORC)*, 2008, pp. 221–228.

[38] SAE International, *AS5506B: Architecture Analysis & Design Language (AADL)*. 2012.

[39] A. S. Tanenbaum, J. N. Herder, and H. Bos, “Can we make operating systems reliable and secure?,” *Computer*, vol. 39, no. 5, pp. 44–51, 2006.

[40] F. Armand and M. Gien, “A practical look at micro-kernels and virtual machine monitors,” in *Consumer Communications and Networking Conference, 2009. CCNC 2009. 6th IEEE*, 2009, pp. 1–7.

[41] G. Heiser, K. Elphinstone, I. Kuz, G. Klein, and S. M. Petters, “Towards trustworthy computing systems: taking microkernels to the next level,” *SIGOPS Oper. Syst. Rev.*, vol. 41, no. 4, pp. 3–11, Jul. 2007.

[42] M. Hohmuth, M. Peter, H. Härtig, and J. S. Shapiro, “Reducing TCB size by using untrusted components: small kernels versus virtual-machine monitors,” in *Proceedings of the 11th workshop on ACM SIGOPS European workshop*, 2004, p. 22.

[43] A. Chou, J. Yang, B. Chelf, S. Hallem, and D. Engler, “An Empirical Study of Operating Systems Errors,” in *Proceedings of the Eighteenth ACM Symposium on Operating Systems Principles*, New York, NY, USA, 2001, pp. 73–88.

[44] T. Garfinkel, B. Pfaff, J. Chow, M. Rosenblum, and D. Boneh, “Terra: A virtual machine-based platform for trusted computing,” in *ACM SIGOPS Operating Systems Review*, 2003, vol. 37, pp. 193–206.

[45] G. Klein, K. Elphinstone, G. Heiser, J. Andronick, D. Cock, P. Derrin, D. Elkaduwe, K. Engelhardt, R. Kolanski, M. Norrish, T. Sewell, H. Tuch, and S. Winwood, “seL4: Formal Verification of an OS Kernel,” in *Proceedings of the ACM SIGOPS 22Nd Symposium on Operating Systems Principles*, New York, NY, USA, 2009, pp. 207–220.

[46] R. Liu, K. Klues, S. Bird, S. Hofmeyr, K. Asanovic, and J. Kubiatowicz, “Tessellation: Space-time partitioning in a manycore client OS,” in *Proceedings of the First USENIX conference on Hot topics in parallelism*, 2009, pp. 10–10.

[47] G. Heiser and B. Leslie, “The OKL4 microvisor: convergence point of microkernels and hypervisors,” in *Proceedings of the first ACM asia-pacific workshop on Workshop on systems*, New York, NY, USA, 2010, pp. 19–24.

[48] “Xilinx.” [Online]. Available: http://www.xilinx.com/. [Accessed: 07-Dec-2015].

[49] “Xilinx University Program Virtex-II Pro Development System.” [Online]. Available: http://www.xilinx.com/products/boards-and-kits/XUPV2P.htm. [Accessed: 11-Oct-2012].

[50] A. Nelson, A. B. Nejad, A. Molnos, M. Koedam, and K. Goossens, “CoMik: A Predictable and Cycle-accurately Composable Real-time Microkernel,” in *Proceedings of the Conference on Design, Automation & Test in Europe*, 3001 Leuven, Belgium, Belgium, 2014, pp. 222:1–

222:4.

[51] A. Tavares, A. Carvalho, P. Rodrigues, P. Garcia, T. Gomes, J. Cabral, P. Cardoso, S. Montenegro, and M. Ekpanyapong, “A customizable and ARINC 653 quasi-compliant hypervisor,” in *2012 IEEE International Conference on Industrial Technology (ICIT)*, 2012, pp. 140–147.

[52] A. Carvalho, F. Afonso, P. Cardoso, J. Cabral, M. Ekpanyapong, S. Montenegro, and A. Tavares, “Cache full-virtualization for the PowerPC 405-S,” presented at the 11th IEEE International Conference on Industrial Informatics, Bochum, Germany, 2013.

[53] A. Tavares, A. Didimo, S. Montenegro, T. Gomes, J. Cabral, P. Cardoso, and M. Ekpanyapong, “RodosVisor - an Object-Oriented and Customizable Hypervisor: The CPU Virtualization,” presented at the Conference on Embedded Systems, Computational Intelligence and Telematics in Control (CESCIT), University of Würzburg, Germany, 2012, pp. 200–205.

[54] G. Heiser, “The role of virtualization in embedded systems,” in *Proceedings of the 1st workshop on Isolation and integration in embedded systems*, 2008, pp. 11–16.

[55] “AMD Virtualization.” [Online]. Available: http://www.amd.com/en-us/solutions/servers/virtualization. [Accessed: 07-Dec-2015].

[56] “Intel® Virtualization Technology for Directed I/O (VT-d): Enhancing Intel platforms for efficient virtualization of I/O devices | Intel® Developer Zone.” [Online]. Available: https://software.intel.com/en-us/articles/intel-virtualization-technology-for-directed-io-vt-d-enhancing-intel-platforms-for-efficient-virtualization-of-io-devices. [Accessed: 07-Dec-2015].

[57] “Intel® Virtualization Technology List.” [Online]. Available: http://ark.intel.com/Products/VirtualizationTechnology. [Accessed: 07-Dec-2015].

[58] “MIPS Virtualization - Imagination Technologies.” [Online]. Available: https://imgtec.com/mips/architectures/virtualization/. [Accessed: 07-Dec-2015].

[59] “Virtualization Extensions - ARM.” [Online]. Available: https://www.arm.com/products/processors/technologies/virtualization-extensions.php. [Accessed: 07-Dec-2015].

[60] D. Baldin and T. Kerstan, “Proteus, a Hybrid Virtualization Platform for Embedded Systems,” in *Analysis, Architectures and Modelling of Embedded Systems*, vol. 310, A. Rettberg, M. Zanella, M. Amann, M. Keckeisen, and F. Rammig, Eds. Springer Boston, 2009, pp. 185–194.

[61] K. Gilles, S. Groesbrink, D. Baldin, and T. Kerstan, “Proteus Hypervisor: Full Virtualization

and Paravirtualization for Multi-core Embedded Systems," in *Embedded Systems: Design, Analysis and Verification*, G. Schirner, M. Götz, A. Rettberg, M. C. Zanella, and F. J. Rammig, Eds. Springer Berlin Heidelberg, 2013, pp. 293–305.

[62] "Main Page - KVM." [Online]. Available: http://www.linux-kvm.org/page/Main_Page. [Accessed: 09-Feb-2013].

[63] A. Kivity, Y. Kamay, D. Laor, U. Lublin, and A. Liguori, "kvm: the Linux virtual machine monitor," in *Proceedings of the Linux Symposium*, 2007, vol. 1, pp. 225–230.

[64] H. Joe, H. Jeong, Y. Yoon, H. Kim, S. Han, and H.-W. Jin, "Full virtualizing micro hypervisor for spacecraft flight computer," in *Digital Avionics Systems Conference (DASC), 2012 IEEE/AIAA 31st*, 2012, pp. 6C5–1–6C5–9.

[65] R. J. Creasy, "The Origin of the VM/370 Time-Sharing System," *IBM Journal of Research and Development*, vol. 25, no. 5, pp. 483 –490, Sep. 1981.

[66] M. Rosenblum and T. Garfinkel, "Virtual machine monitors: Current technology and future trends," *Computer*, vol. 38, no. 5, pp. 39–47, 2005.

[67] "DO-178B Level A Certified RTOS, Certified POSIX IEEE 1003.13, MILS, EAL 6+ Safety Critical software - Green Hills Software." [Online]. Available: http://www.ghs.com/products/safety_critical/integrity-do-178b.html. [Accessed: 10-Jan-2014].

[68] "Time & Space Partitioned DO-178 Level A Certifiable RTOS." [Online]. Available: http://www.ddci.com/products_deos.php. [Accessed: 10-Jan-2014].

[69] "LynxOS-178 RTOS for DO-178B Software Certification | Real-Time Operating Systems and Virtualization Security: Lynx Software TechnologiesReal-Time Operating Systems and Virtualization Security: Lynx Software Technologies." [Online]. Available: http://www.lynx.com/products/real-time-operating-systems/lynxos-178-rtos-for-do-178b-software-certification/. [Accessed: 08-Jul-2014].

[70] "Wind River VxWorks: Certification Profiles." [Online]. Available: http://www.windriver.com/products/vxworks/certification-profiles/#vxworks_653. [Accessed: 10-Dec-2015].

[71] M. Masmano, Y. Valiente, P. Balbastre, I. Ripoll, A. Crespo, and J. J. Metge, "LithOS: a ARINC-653 guest operating for XtratuM," in *Proc. of the 12th Real-Time Linux Workshop, Nairobi (Kenya)*, 2010.

[72] GMV, "SIMA Overview." 2010.

[73] A. Horváth, D. Varró, and T. Schoofs, “Model-driven development of ARINC 653 configuration tables,” in *Digital Avionics Systems Conference (DASC), 2010 IEEE/AIAA 29th*, 2010, pp. 5.A.5–1–5.A.5–115.

[74] T. Schoofs, S. Santos, C. Tatibana, and J. Anjos, “An integrated modular avionics development environment,” in *Digital Avionics Systems Conference, 2009. DASC ’09. IEEE/AIAA 28th*, 2009, pp. 1.A.2–1 –1.A.2–9.

[75] “SCARLETT PROJECT.” [Online]. Available: http://www.scarlettproject.eu/. [Accessed: 02-Mar-2016].

[76] Q. Zhou, Z. Xiong, Z. Zhan, T. You, and N. Jiang, “The mapping mechanism between Distributed Integrated Modular Avionics and data distribution service,” in *2015 12th International Conference on Fuzzy Systems and Knowledge Discovery (FSKD)*, 2015, pp. 2502–2507.

[77] G. Wang and Q. Gu, “Research on Distributed Integrated Modular Avionics system architecture design and implementation,” in *Digital Avionics Systems Conference (DASC), 2013 IEEE/AIAA 32nd*, 2013, pp. 7D6–1–7D6–10.

[78] R. Wolfig and M. Jakovljevic, “Distributed IMA and DO-297: Architectural, communication and certification attributes,” in *Digital Avionics Systems Conference, 2008. DASC 2008. IEEE/AIAA 27th*, 2008, pp. 1.E.4–1–1.E.4–10.

[79] Q. Zhou, T. Gu, R. Hong, and S. Wang, “An AADL-based design for dynamic reconfiguration of DIMA,” in *Digital Avionics Systems Conference (DASC), 2013 IEEE/AIAA 32nd*, 2013, pp. 4C1–1–4C1–8.

[80] “OSEK VDX Portal - Home.” [Online]. Available: http://www.osek-vdx.org/. [Accessed: 10-Dec-2015].

[81] Barry Boehm, *Software Engineering Economics*. 1981.

[82] Xilinx, *Xilinx University Program Virtex-II Pro Development System: Hardware Reference Manual*, 1.2 ed. 2009.

[83] Xilinx, *Virtex-II Pro and Virtex-II Pro X FPGA User Guide*, 4.2 ed. 2007.

[84] Xilinx, *Virtex-II Pro and Virtex-II Pro X Platform FPGAs: Complete Data Sheet*, 4.7 ed. 2007.

[85] “Make - GNU Project - Free Software Foundation.” [Online]. Available: https://www.gnu.org/software/make/. [Accessed: 11-Dec-2015].

[86] “GCC, the GNU Compiler Collection - GNU Project - Free Software Foundation (FSF).” [Online]. Available: https://gcc.gnu.org/. [Accessed: 11-Dec-2015].

[87] “Buildroot - Making Embedded Linux Easy.” [Online]. Available: https://buildroot.org/. [Accessed: 22-Apr-2016].

[88] “http://www.netlib.org/benchmark/dhry-c.” [Online]. Available: http://www.netlib.org/benchmark/dhry-c. [Accessed: 11-Mar-2016].

[89] “http://www.netlib.org/benchmark/whetstone.c.” [Online]. Available: http://www.netlib.org/benchmark/whetstone.c. [Accessed: 11-Mar-2016].

[90] “The Netperf Homepage.” [Online]. Available: http://www.netperf.org/netperf/. [Accessed: 11-Mar-2015].

[91] “SLOCCount,” *SourceForge*. [Online]. Available: https://sourceforge.net/projects/sloccount/. [Accessed: 11-Mar-2016].

[92] R. France and B. Rumpe, “Model-driven Development of Complex Software: A Research Roadmap,” in *2007 Future of Software Engineering*, Washington, DC, USA, 2007, pp. 37–54.

[93] J.-P. Fassino, J.-B. Stefani, J. L. Lawall, and G. Muller, “Think: A Software Framework for Component-based Operating System Kernels.,” in *USENIX Annual Technical Conference, General Track*, 2002, pp. 73–86.

[94] O. Lobry and J. Polakovic, “Controlling the Performance Overhead of Component-Based Systems,” in *Software Composition*, C. Pautasso and É. Tanter, Eds. Springer Berlin Heidelberg, 2008, pp. 149–156.

[95] F. Loiret, J. Navas, J.-P. Babau, and O. Lobry, “Component-Based Real-Time Operating System for Embedded Applications,” in *Component-Based Software Engineering*, G. A. Lewis, I. Poernomo, and C. Hofmeister, Eds. Springer Berlin Heidelberg, 2009, pp. 209–226.

[96] I. Kuz, Y. Liu, I. Gorton, and G. Heiser, “CAmkES: A component model for secure microkernel-based embedded systems,” *Journal of Systems and Software*, vol. 80, no. 5, pp. 687–699, May 2007.

[97] P. Levis, S. Madden, J. Polastre, R. Szewczyk, K. Whitehouse, A. Woo, D. Gay, J. Hill, M. Welsh, E. Brewer, and D. Culler, “TinyOS: An Operating System for Sensor Networks,” in *Ambient Intelligence*, W. Weber, J. M. Rabaey, and E. Aarts, Eds. Springer Berlin Heidelberg, 2005, pp. 115–148.

[98] C. K. Angelov, I. E. Ivanov, and A. Burns, “HARTEX—a safe real-time kernel for distributed computer control systems,” *Softw: Pract. Exper.*, vol. 32, no. 3, pp. 209–232, 2002.

[99] T. Weigert and F. Weil, “Practical experiences in using model-driven engineering to develop trustworthy computing systems,” in *IEEE International Conference on Sensor Networks, Ubiquitous, and Trustworthy Computing, 2006*, 2006, vol. 1, p. 8 pp.–.

[100] P. Mohagheghi and V. Dehlen, “Where Is the Proof? - A Review of Experiences from Applying MDE in Industry,” in *Model Driven Architecture – Foundations and Applications*, I. Schieferdecker and A. Hartman, Eds. Springer Berlin Heidelberg, 2008, pp. 432–443.

[101] “MDA.” [Online]. Available: http://www.omg.org/mda/. [Accessed: 22-Dec-2015].

[102] J. Hutchinson, M. Rouncefield, and J. Whittle, “Model-driven engineering practices in industry,” in *2011 33rd International Conference on Software Engineering (ICSE)*, 2011, pp. 633–642.

[103] M. Mohammad and V. Alagar, “TADL - An Architecture Description Language for Trustworthy Component-Based Systems,” in *Software Architecture*, R. Morrison, D. Balasubramaniam, and K. Falkner, Eds. Springer Berlin Heidelberg, 2008, pp. 290–297.

[104] D. Garlan, R. Monroe, and D. Wile, “Acme: An Architecture Description Interchange Language,” in *CASCON First Decade High Impact Papers*, Riverton, NJ, USA, 2010, pp. 159–173.

[105] J. Li, N. T. Pilkington, F. Xie, and Q. Liu, “Embedded architecture description language,” *Journal of Systems and Software*, vol. 83, no. 2, pp. 235–252, Feb. 2010.

[106] O. Gilles and J. Hugues, “Validating requirements at model-level,” in *Proc. of the 4th workshop on Model-Oriented Engineering*, 2008.

[107] O. Gilles and J. Hugues, “Expressing and enforcing user-defined constraints of AADL models,” in *Engineering of Complex Computer Systems (ICECCS), 2010 15th IEEE International Conference on*, 2010, pp. 337–342.

[108] “MOFM2T.” [Online]. Available: http://www.omg.org/spec/MOFM2T/. [Accessed: 22-Dec-2015].

[109] H. Casanova, M. Kim, J. S. Plank, and J. J. Dongarra, “Adaptive scheduling for task farming with grid middleware,” *International Journal of High Performance Computing Applications*, vol. 13, no. 3, pp. 231–240, 1999.

[110] C. Brown, V. Janjic, K. Hammond, H. Schoner, K. Idrees, and C. W. Glass, “Agricultural reform: more efficient farming using advanced parallel refactoring tools,” in *Parallel, Distributed and Network-Based Processing (PDP), 2014 22nd Euromicro International Conference on*, 2014, pp. 36–43.

[111]K. Molitorisz, “Pattern-based refactoring process of sequential source code,” in *Software Maintenance and Reengineering (CSMR), 2013 17th European Conference on*, 2013, pp. 357–360.

[112] K. Hammond, M. Aldinucci, C. Brown, F. Cesarini, M. Danelutto, H. González-Vélez, P. Kilpatrick, R. Keller, M. Rossbory, and G. Shainer, “The paraphrase project: Parallel patterns for adaptive heterogeneous multicore systems,” in *Formal Methods for Components and Objects*, 2013, pp. 218–236.

[113] D. Dig, “A refactoring approach to parallelism,” *Software, IEEE*, vol. 28, no. 1, pp. 17–22, 2011.

[114] A. Bittau, P. Marchenko, M. Handley, and B. Karp, “Wedge: Splitting Applications into Reduced-Privilege Compartments.,” in *NSDI*, 2008, vol. 8, pp. 309–322.

[115] S. F. Smith and M. Thober, “Refactoring programs to secure information flows,” in *Proceedings of the 2006 workshop on Programming languages and analysis for security*, 2006, pp. 75–84.

[116] V. Ganapathy, T. Jaeger, and S. Jha, “Retrofitting legacy code for authorization policy enforcement,” in *Security and Privacy, 2006 IEEE Symposium on*, 2006, p. 15–pp.

[117] D. Brumley and D. Song, “Privtrans: Automatically partitioning programs for privilege separation,” in *USENIX Security Symposium*, 2004, pp. 57–72.

[118] L. Yu and S. Ramaswamy, “Improving Modularity by Refactoring Code Clones: A Feasibility Study on Linux,” *SIGSOFT Softw. Eng. Notes*, vol. 33, no. 2, pp. 9:1–9:5, Mar. 2008.

[119] J. Wloka, M. Sridharan, and F. Tip, “Refactoring for reentrancy,” in *Proceedings of the the 7th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering*, 2009, pp. 173–182.

[120] V. Chipounov and G. Candea, “Reverse Engineering of Binary Device Drivers with RevNIC,” in *Proceedings of the 5th European Conference on Computer Systems*, New York, NY, USA, 2010, pp. 167–180.

[121] A. Whitaker, M. Shaw, S. D. Gribble, and others, “Denali: Lightweight virtual machines for distributed and networked applications,” Citeseer, 2002.

[122] A. Whitaker, M. Shaw, and S. D. Gribble, “Scale and performance in the Denali isolation kernel,” *SIGOPS Oper. Syst. Rev.*, vol. 36, no. SI, pp. 195–209, Dec. 2002.

[123] A. Whitaker, R. S. Cox, M. Shaw, and S. D. Gribble, “Rethinking the design of virtual machine monitors,” *Computer*, vol. 38, no. 5, pp. 57 – 62, May 2005.

[124] K. Pohl, G. Bockle, and F. Van Der Linden, *Software product line engineering*, vol. 10. Springer, 2005.

[125] Real-Time Systems GmbH, “Real-Time Systems - Embedded Hypervisor for Intel x86 Multicore Architecture,” 2015. [Online]. Available: http://www.real-time-systems.com/real-time_hypervisor/index.php. [Accessed: 24-Oct-2014].

[126] W. Kanda, Y. Yumura, Y. Kinebuchi, K. Makijima, and T. Nakajima, “Spumone: Lightweight cpu virtualization layer for embedded systems,” in *Embedded and Ubiquitous Computing, 2008. EUC’08. IEEE/IFIP International Conference on*, 2008, vol. 1, pp. 144–151.

[127] “VxWorks.” [Online]. Available: http://windriver.com/products/vxworks/. [Accessed: 28-Aug-2015].

[128] “X-HYP Realtime Hypervisor.” [Online]. Available: http://x-hyp.org/. [Accessed: 27-Aug-2015].

[129] “The Xen Project, the powerful open source industry standard for virtualization.” [Online]. Available: http://www.xenproject.org/. [Accessed: 22-Jul-2013].

[130] E. Carrascosa, J. Coronel, M. Masmano, P. Balbastre, and A. Crespo, “XtratuM Hypervisor Redesign for LEON4 Multicore Processor,” *SIGBED Rev.*, vol. 11, no. 2, pp. 27–31, Sep. 2014.

[131] A. Crespo, I. Ripoll, and M. Masmano, “Partitioned Embedded Architecture Based on Hypervisor: The XtratuM Approach,” in *Dependable Computing Conference (EDCC), 2010 European*, 2010, pp. 67–72.

[132] A. Crespo, I. Ripoll, M. Masmano, P. Arberet, and J. J. Metge, “XtratuM an Open Source Hypervisor for TSP Embedded Systems in Aerospace,” *Data Systems In Aerospace DASIA, Istanbul, Turkey*, 2009.

[133] S. Zampiva, C. Moratelli, and F. Hessel, “A hypervisor approach with real-time support to the MIPS M5150 processor,” in *2015 16th International Symposium on Quality Electronic Design (ISQED)*, 2015, pp. 495–501.

[134] “SierraVisor Virtualization Hypervisor for ARM.” [Online]. Available: http://www.sierraware.com/arm_hypervisor.html. [Accessed: 14-Jan-2014].

[135] S. H. VanderLeest, D. Greve, and P. Skentzos, “A safe & secure arinc 653 hypervisor,” in *Digital Avionics Systems Conference (DASC), 2013 IEEE/AIAA 32nd*, 2013, pp. 7B4–1.

[136] J. Rufino, J. Craveiro, T. Schoofs, C. Tatibana, and J. Windsor, “AIR Technology: a step towards ARINC 653 in space,” in *Proc. DASIA*, 2009.

[137] J. Craveiro, J. Rufino, and F. Singhoff, “Architecture, Mechanisms and Scheduling Analysis Tool for Multicore Time- and Space-partitioned Systems,” *SIGBED Rev.*, vol. 8, no. 3, pp. 23–27, Sep. 2011.

[138] F. Bruns, D. Kuschnerus, and A. Bilgic, “Virtualization for safety-critical, deeply-embedded devices,” in *Proceedings of the 28th Annual ACM Symposium on Applied Computing*, 2013, pp. 1485–1492.

[139] U. Steinberg and B. Kauer, “NOVA: a microhypervisor-based secure virtualization architecture,” in *Proceedings of the 5th European conference on Computer systems*, New York, NY, USA, 2010, pp. 209–222.

[140] “Codezero Embedded Hypervisor™ - B Labs | ARM Connected Community.” [Online]. Available: http://community.arm.com/docs/DOC-7123. [Accessed: 15-Oct-2014].

[141] “QNX Hypervisor.” [Online]. Available: http://www.qnx.com/products/hypervisor/index.html. [Accessed: 18-Jun-2015].