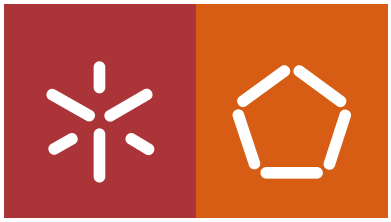
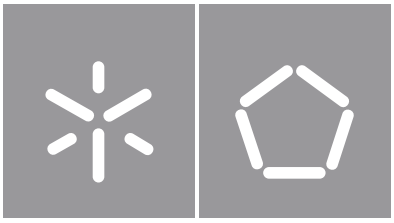


Universidade do Minho
Escola de Engenharia

João Miguel Costa Sousa

HSP-V: Holistic Static Partitioning on RISC-V Platforms

julho de 2022



Universidade do Minho
Escola de Engenharia

João Miguel Costa Sousa

HSP-V: Holistic Static Partitioning on RISC-V Platforms

Dissertação de Mestrado
Engenharia Eletrónica Industrial e Computadores
Sistemas Embebidos e Computadores

Trabalho efetuado sob a orientação de
Professor Doutor Sandro Emanuel Salgado Pinto
Professor Doutor João Luís Marques Pereira Monteiro

julho de 2022

Agradecimentos

É nesta altura que oficialmente finalizo uma etapa nobre. Uma etapa cheia de luta, de altos e baixos, de medos que se transformaram em conhecimento, de insegurança que virou confiança e de uma completa transformação. Certamente que doravante na minha carreira me irei lembrar de muitos momentos vividos nestes anos: da ansiedade pre testes, dos ditotes de cada professor, dos projetos realizados, do bar da Bertina (mãezinha), dos momentos de laboratório, dos lanches no bar, dos jogos de xadrez, do caderno mítico do lab. Por isso um agradecimento especial ao Antrapunas (Samuel Pereira), ao Elon Felps (André Campos), DJ'Review (Luís Cunha), O'Mestre (Diogo Costa) e ao irmão, mas não de sangue (Pedro Sousa). Estas pessoas fizeram o meu percurso, e merecem um especial agradecimento porque se não existissem, iria ter que as inventar. Foram aqueles que estiveram sempre lá para me tirar dúvidas; para me dar vícios; para me distrair; para me dizerem que estavam a sofrer (fazendo-me acreditar que não estava na luta sozinho); para me anestesiar com todas as conversas de criptomoedas, empreendedorismo, política, futebol, etc. Foram mais que colegas de trabalho, foram amigos!

Numa vertente mais técnica, recordo-me do dia em que entrei naquele laboratório de prestígio pela primeira vez, o dia em que me expus a escolher a orientação. Entrei a tremer, com noção da decisão que tomava. Hoje, entro naquele laboratório com confiança, "segurança", com vontade e orgulho. Por isso deixo também um agradecimento especial aos meus orientadores: Professor Doutor Sandro Pinto, pelas vezes que me disse para bater com a cabeça, pela exigência, pela paciência, por todo o suporte e claro pela excelente orientação; Mestre José Martins, agradeço pela paciência, pela partilha de conhecimento, pela disponibilidade, pela exigência e claro também pela excelente orientação. Ainda numa parte técnica quero agradecer a uma pessoa que apareceu em alguns momentos difíceis, ao Mestre Bruno Sã que esteve lá para suportar algumas dúvidas, mesmo cheio de trabalho. Uma orientação que me fez acreditar que "é sempre" possível, mesmo que demore tempo.

Por último, mas não menos importante, queria agradecer a uma parte de suporte mais pessoal: Mãe, por fazer o meu tempo mais organizado e pela preocupação incansável; Pai, pelo suporte mental e por me mostrar que sou capaz daquilo que ainda nem sequer acredito; avós, madrinha, tios e primos, pelo apoio e pressão, que me fez não sair do foco; e Diana, que esteve lá nos momentos que ninguém viu, sempre pronta a sorrir (fosse em momentos de sol ou momentos nublados), pela sua paciência e pelos seus conselhos.

STATEMENT OF INTEGRITY

I hereby declare having conducted this academic work with integrity. I confirm that I have not used plagiarism or any form of undue use of information or falsification of results along the process leading to its elaboration. I further declare that I have fully acknowledged the Code of Ethical Conduct of the University of Minho.

Resumo

Particionamento Estático Holístico em Plataformas RISC-V

No mundo dos sistemas embebidos, a virtualização apareceu como uma das soluções mais seguras, eficientes e económicas para consolidar os Mixed-Critical Systems (MCSs) na mesma plataforma de hardware. Neste contexto, os hipervisores têm sido utilizados para enfrentar vários desafios de segurança, monitorizando os recursos de hardware e virtualizando as suas partições através de extensões de virtualização. Tipicamente, o RISC-V, uma inovadora, livre, priveligiada e com Instruction Set Architecture (ISA) aberto, implementa estas tecnologias com uma extenção do modo supervisor (modo HS), fornecendo novos Control and Status Registers (CSRs) e realizando a tradução de endereços em duas fases. Contudo, para evitar esta camada extra de tradução e empregar um sistema de partição estática nas plataformas RISC-V, esta dissertação utiliza um método para-virtualizado com o apoio da unidade Physical Memory Protection (PMP) e a técnica de trap-and-emulate.

Para estabelecer o Holistic Static Partitioning on RISC-V platforms (HSP-V), esta dissertação explora uma camada fina de software, an Open-source implementation of the RISC-V Supervisor Binary Interface (OpenSBI). Este é responsável por estabelecer de comunicação entre camadas de nível superior com camadas de nível inferior de privilégio, além disso, trata as partições como instâncias de domínio e sofre da exposição componentes críticos da memória sobre os mesmos. Entre os componentes críticos está o controlador de interrupção externa RISC-V, o Platform-Level Interrupt Controller (PLIC), que segue uma estrutura de Memory Mapped Input Output (MMIO). Como resultado, o HSP-V revela o reforço do isolamento entre domínios ao particionar a estrutura do PLIC e conectando diretamente as regiões mapeamento de MMIO, através da técnica de trap-and-emulate e da unidade de PMP. Para avaliar as modificações foi utilizada a placa MPFS-Icicle-Kit, analisando: o desempenho de sobrecarga e interferência entre domínios (explorando métodos de bloqueio da cache), tempos de inicialização do sistema, latência de interrupção, e recursos de hardware em termos de maior complexidade do código OpenSBI, antes e depois de todo o melhoramento.

Palavras-chave: Sistemas Embebidos, RISC-V, Sistemas de criticidade mista, Virtualização, OpenSBI, PLIC, Particionamento Estático

Abstract

Holistic Static Partitioning on RISC-V Platforms.

In the world of embedded systems, virtualization has appeared as one of the most secure, efficient, and cost-effective solution to consolidate Mixed-Critical Systems (MCSs) onto the same hardware platform. Hereupon, hypervisors have been used to face several security challenges by monitoring hardware resources and virtualizing their partitions through virtualization extensions. Typically, the RISC-V, an innovative, free, open Instruction Set Architecture (ISA) and privileged computer architecture, implements these technologies with the hypervisor-extended supervisor mode (HS-mode), by providing new Control Status Registers (CSRs) and performing the two-stage-address translation. However, to avoid this extra translation layer and employ a static partitioning system in RISC-V platforms, this dissertation uses a para-virtualized method with the support of the Physical Memory Protection (PMP) unit and the trap-and-emulate technique.

To establish a Holistic Static Partitioning system over RISC-V platforms (HSP-V), this dissertation enhances a thin layer of software, the Open-source Supervisor Binary Interface (OpenSBI). This is responsible for communication establishment between high and low-privileged layers. In addition, it treats partitions as domain instances and suffers from exposing critical memory components between domains. Between them is the RISC-V external interrupt controller, the Platform-Level Interrupt Controller (PLIC), following a Memory Mapped Input/Output (MMIO) structure. As a result, the HSP-V reveals the reinforcement of the inter-domain isolation by partitioning the PLIC structure and direct assigning the device MMIO regions through domain instances by leveraging the PMP unit and the trap-and-emulate technique. To evaluate HSP-V implementations was used the MPFS-Icicle-Kit board to analyze: performance in terms of overhead and inter-domain interference (leveraging the cache locking methods), boot overhead, interrupt latency, and hardware resources in terms of enhanced OpenSBI code complexity, before and afterward all enhancement.

Keywords: Embedded Systems, RISC-V, MCSs, Virtualization, OpenSBI, PLIC, Static Partitioning.

Contents

List of Figures

List of Tables

Glossary

ABI Aplication Binary Interface

AEE Aplication Execution Environment

APP Computer Application

ASID Address Space Identifier

CLINT Core Local Interrupt

COTS Commercial Off-The-Shelf

CPU Central Processing Unit

CSR Control and Status Register

CSRRC Atomic Read and Clear Bits in CSR

CSRRS Atomic Read and Set Bits in CSR

CSRRW Atomic Read/Write CSR

DMA Direct Memory Access

DoS Denial-of-Service

DT Device Tree

DTC Device Tree Compiler

ECall Environment Call

EID SBI Extension ID

eMMC embedded Multi-Media Card

FDT Flattened Device Tree

FID SBI Funtion ID

FSBL First-Stage Boot Loader

GDB Gnu DeBugger

GPA Guest Physical Address

GPOS General Purpose Operating System

GVA Guest Virtual Address

HSP-V Holistic Static Partitioning on RISC-V platforms

HSS Hart Software Services

IPI Inter Processor Interrupt

ISA Instruction Set Architecture

ISR Instruction Service Routine

LLC Last-Level Cache

MCS Mixed-Critical System

MMC Multi-Media Card

MMIO Memory Mapped Input Output

MMU Memory Management Unit

NA4 Naturally Aligned Four-byte region

NAPOT Naturally Aligned Power-Of-Two region

OpenOCD Open On-Chip Debugger

OpenSBI Open-source implementation of the RISC-V Supervisor Binary Interface

OS Operating System

PA Physical Address

PC Program Counter

PLIC Platform-Level Interrupt Controller

PMA Physical Memory Attributes

PMP Physical Memory Protection

PMU Physical Memory Unit

PPN Physical Page Number

RISC Reduced Instruction Set Computer

ROM Read-Only Memory

RTOS Real-Time Operating System

SBI Supervisor Binary Interface

SEE Supervisor Execution Environment

SLoC Source Lines of Code

SP Stack Pointer

SRAM Static Random Access Memory

SSBL Second-Stage Boot Loader

TCB Trusted Computing Base

TEE Trusted execution environment

TLB Translation Lookaside Buffer

TOR Top Of Range region)

VA Virtual Address

VM Virtual Machine

VMM Virtual Machine Monitor

ZSBL Zero-Stage Boot Loader

1. Introduction

The world of embedded and cyber-physical systems has evolved exponentially, being explored both in industry and academia [1, 2]. The evolution has been changing from single-purpose systems, with reduced communications and simple interfaces, to general-purpose systems with multiple functionalities and a higher level of complexity [3]. As a result, different challenges arise from this paradigm, such as integrating several subsystems with distinct criticality levels, the so-called Mixed-Critical System (MCS)s, into the same hardware platform while preserving real-time requirements [4].

After significant academic and industry research on different methodologies, virtualization emerges as one of the most secure, efficient, and cost-effective solution to consolidate different critical level subsystems onto the same hardware platform [5, 6]. Moreover, to provide spatial and temporal isolation across several subsystems, this technology employs an extra software layer, the Virtual Machine Monitor (VMM), also known as a hypervisor. The hypervisor is responsible for monitoring hardware resources and idealizes its partitions to be operating on real hardware while, in reality, they are running on virtual machines [7]. A good example of a system leveraging virtualization capabilities is a General Purpose Operating System (GPOS) running alongside a Real-Time Operating System (RTOS) as if each were running in separated hardware, both leveraging the best performance. However, despite the hypervisor providing a logical Central Processing Unit (CPU) and memory isolation, its partitions still share some resources. Due to the complex memory hierarchies of powerful multi-core market platforms, resources such as Last-Level Cache (LLC), buses, and memory controllers are shared [8, 9]. Thus, a malicious partition may conduct aDenial-of-Service (DoS) attack, by stressing these shared resources through repeated accesses, or by indirectly accessing the data of adjacent partitions by reusing existing time side channels [10, 11]. Thus, research communities propose techniques such as cache locking [12] or cache coloring [13].

Along with the proposed solutions to solve mixed-criticality issues, the industry and academia researchers developed an open Instruction Set Architecture (ISA), the RISC-V [14]. This architecture can provide advantages comparing to other well-established architectures (e.g., Intel [15] and Arm [16]), i.e., in its free-to-use, high modularity, and customizable extension scheme. These features allow implementations to scale from tiny embedded micro-controllers to supercomputers. Furthermore, to protect different components of the software stack, it provides three main privileged modes [17]: (i) the machine mode (M-mode), the only mandatory and the highest privileged level; (ii) the supervisor mode (S-mode),

typically used to run an Operating System (OS) and provides virtual memory capabilities through a dedicated Memory Management Unit (MMU); and (iii) the user mode (U-mode), for conventional applications. For hypervisor support and to contribute to the virtualization efficiency, the S-mode can be extended for hypervisor-extended supervisor mode (HS-mode) [18, 19]. Inevitably, HS-mode introduces memory translation support through a two-stage address translation, from a Virtual Address (VA) to a Physical Address (PA). This process splits into two stages: (i) the Guest Virtual Address (GVA) to Guest Physical Address (GPA), through the OS; and (ii) the GPA to PA, through the hypervisor. On the other hand, due to other architectural support, i.e., the Physical Memory Protection (PMP) unit, it is possible to avoid this two-stage address translation [20]. The PMP unit grants/suppress access permissions to low privileged layers (S or U-mode), and each hart (essentially a core in RISC-V terminology) can manage different accesses without passing through a hypervisor layer while providing isolation between partitions [21, 22]. When running a GPOS over the S-Mode privileged layer, the address translation is done directly from VA to PA through the MMU.

To establish static partitioning systems in RISC-V without hypervisor extensions, Open-source implementation of the RISC-V Supervisor Binary Interface (OpenSBI) provides a thin layer of software running in M-mode. It integrates a partitioning system, the OpenSBI domains systems, which assign memory regions to one or several harts. Each domain establishes its memory regions during Second-Stage Boot Loader (SSBL) through to the PMP unit. Additionally, the OpenSBI offers run-time services through a Supervisor Binary Interface (SBI) between low and higher privileged levels (following RISC-V SBI specifications).

Despite the well-provided inter-domain isolation and due to the PMP constraints, OpenSBI does not protect all domain resources, as is the case of the RISC-V external interrupt controller, the Platform-Level Interrupt Controller (PLIC). Because the PLIC is an Memory Mapped Input Output (MMIO) structure and needs to be shared among all domains, when a malicious domain tries to access it, it can compromise other domains and break the spacial isolation. Aiming to tackle this issue, this dissertation contributes to the protection of this memory component by enhancing the OpenSBI with additional features such as, the partitioning PLIC, as the direct assignment of device MMIO regions to domains and integrating a virtualization technique, the trap-and-emulate, to establish inter-domain isolation.

1.1 Main Goal

This dissertation aims to develop an Holistic Static Partitioning on RISC-V platforms (HSP-V), which enhances the OpenSBI, either (i) in the configuration of OpenSBI domain instances, (ii) in automatic direct device MMIO assignment, and (iii) the partitioning of PLIC structure. The main goals of this dissertation are outlined below:

- Implement a RISC-V static partitioning system without hypervisor extension by enhancing OpenSBI configuration and adding virtualization techniques (e.g., the trap-and-emulate) to improve the external interrupt controller isolation between partitions/domains;

- Evaluate the enhanced OpenSBI over a multi-core environment, in a real Commercial Off-The-Shelf (COTS) RISC-V platform, while supporting different combinations of subsystems (e.g., a GPOS with an RTOS; a RTOS with a bare-metal application; a bare-metal application with a GPOS);
- Evaluate the following metrics: performance in terms of inter-domain interference (with and without cache locking); boot overhead; interrupt latency; and hardware resources in terms of complexity of the code of OpenSBI, previously and afterward all changes;

1.2 Dissertation Structure

This dissertation is structured as follows: Chapter 2 starts by giving a background of the virtualization, going through several concepts, presenting its impact in embedded system and different approaches of its use. Next, it goes through the used architecture, the RISC-V, which reveals all of its specifications, privileged modes, stack terminology and its interrupt controllers. Lastly, describes the main target software, i.e., the OpenSBI, going through its contributions, its functionalities and its configuration details (e.g., how it statically instantiates different partitions in same hardware platform). Last but not least, this chapter ends by specifying the related work and comparing their contributions and their protection methods with the used by this dissertation. After the first contextualization, Chapter 3 begins by examining a multi-domain scenario formed by the original OpenSBI. It analyzes the isolation of OpenSBI domains and reveals some of their critical memory regions. Next, it proposes a design solution adopted by this dissertation, i.e., the HSP-V. This solution essentially consists in enhancement of the OpenSBI in different aspects: (i) by adding a OpenSBI configuration feature and (ii) by enhancing the isolation through architecture capabilities (the PMP) or though a virtualization technique (the trap-and-emulate). To close the chapter, it analyzed a multiple domains' scenario with the proposed design. Chapter 4, shows the implementation details of the proposed design, leveraging all topics mention in previous chapter. Chapter 5, evaluate the results over a specific RISC-V platform by leveraging several topics like: functional validation tests; code complexity; boot overhead; interrupt latency; performance overhead and interference. At the end of this chapter, this dissertation discusses the obtained results going through each of them. Last but not least, Chapter 6 leverages all work, arguing about all changes and mention all obtained contributions. At the end, this dissertation finishes by mention some work that could possibly be added to reinforce the HSP-V.

2. Background and Related Work

2.1 Virtualization

The concept of virtualization emerged between the years 1960-1970 when IBM launched time-sharing solutions [23]. Time-sharing solutions refer to the pooling of computer resources between several users, enabling better efficiency and cost savings. This process resulted in a breakthrough in computer technology, affecting sectors such as the avionics and automotive industry [24, 25], which drove virtualization to the point where it has emerged as the greatest solution for increasing resource usage. With virtualization, industries can achieve competitive advantages [26] by increasing efficiency and reducing production costs. However, the required consolidation of several subsystems onto a single hardware platform brings the form of MCSs [27]. When combining components with different levels of criticality, embedded systems face challenges in security, safety, and resource-sharing efficiency.

2.1.1 Virtualization for embedded systems

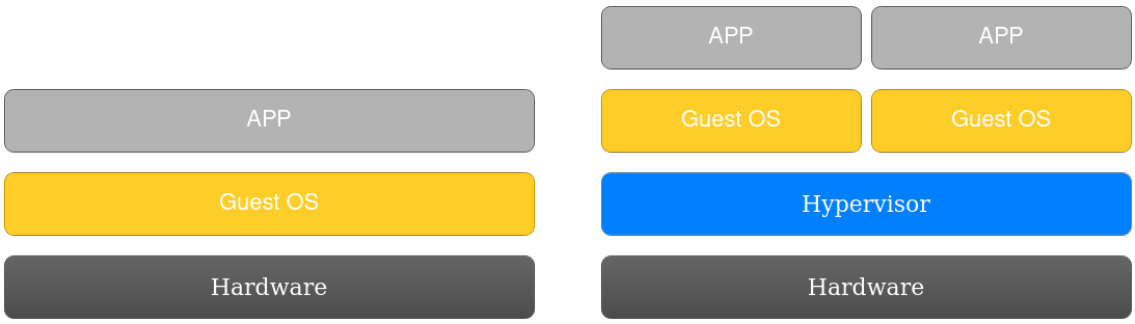
As the complexity and real-time needs have increased, virtualization has become more prevalent in embedded systems. These have changed from single functionality and single-purpose to sophisticated and diverse/complex purposes systems [28, 29]. For example, network-connected infotainment and air conditioning are typically integrated into automotive or avionics alongside safety-critical control systems [30, 31]. Therefore, system virtualization designers must pay special attention while combining these subsystems to ensure truly spatial and temporal isolation between partitions. Spatial isolation exists when partitions are unable to modify one another's code or private data [32], and it is typically disrupted when shared resources are exposed (memory buses, cache, and others). On the other hand, if the execution time of a partition is not affected by other partitions, the system is termed temporally isolated [33].

Nowadays, the struggle to realize the integration of MCSs on the same hardware platform has created immense challenges in computing [34, 35]. Challenges in the trustworthiness of partitions with the absence of catastrophic failures based on non-deliberate acts or circumstances, safety [36], or even when exposed to malicious attacks on deliberated actions, security [36]. However, virtualization appears as a natural solution with hypervisors (also known as VMM) like Xen [37], KVM [38], and Bao [39] to achieve consolidation and integration or even focus on safety and security functionalities.

2.1.2 Virtual Machine Monitor

Hypervisors appear as an additional layer of software responsible for monitoring hardware resources of its partitions and idealize them that they are running in a separated hardware, as similar as possible compared to the original [40]. These underlying hardware subsystems that use a virtualized interface, and that are monitored by hypervisors, are called Virtual Machine (VM)s.

In MCSs, the hypervisor typically have complete control over the underlying hardware platform [40, 41]. As seen in Figure 2.1, the hypervisor is placed closest to the hardware (type-1 hypervisors), while the OSs are confined to lower privilege levels. The virtualized computing stack highlights how several concurrent guests may be consolidated while being isolated from the underlying global hardware, contrarily to a regular computing stack, which only supports a single OS. Alternatively, a hypervisor can be installed on top of an OS (type-2 hypervisors). In this scenario, the operating system is in charge of controlling the hardware, and the hypervisor is viewed as just another application, allowing many Virtual Machines (VMs) to operate on top of it.



(a) Regular Computing Stack **(b)** Virtualized Computing Stack

Figure 2.1: System stack with (b) and without (a) virtualization.

A type-1 hypervisor follows three certain characteristics: (i) *essentially identical*, (ii) *efficient*, and (iii) *resource control*. [40]:

- **Essentially identical** - A hypervisor complies with the *essential identical* property when all applications running under it, behave exactly as if it were operating on the original machine;
- **Efficient** - A hypervisor complies with the efficient property when the current processor executes a significant portion of the virtual instructions directly into VMs, without software interference from the VMM;
- **Resource control** - A hypervisor complies with the resource control property when it controls the VMs and their entire resources;

In virtualized architectures, there is the requirement for a processor that supports multiple privilege levels, where typically OSs run at a high privilege levels and applications run at a low privilege levels. Vertical transitions between the privilege levels occur through synchronous events called traps [18], in which they

manage the control of the processor mode. Depending on the current machine context, the execution of architecture instructions has different classifications concerning classical virtualization [40, 41, 42]:

- **Privileged instructions:** instructions that are restricted for higher privileged levels, occurring trap when executed under low privileged levels then the delegated;
- **Sensitive instructions:** critical instructions capable of controling resources at a machine privileged level;
- **Innocuous instructions:** instructions that are not classified privileged or sensitive. Typically those whose not depend on a specific context processor;

Due to the resource control property, a VMM takes control of critical instructions, i.e., it control of all sensitive instructions (instructions required for proper virtualization) that are mutually considered privileged instructions (instructions susceptible to trap). As a result, VMM must have a higher privilege level than their guests. In virtualization, the execution context can switch to a higher privileged mode following two main virtualization techniques:

- **Fullvirtualization**: In this approach, VMs have no notion that they are running in a virtualized environment. The fundamental advantage of this method is its portability, as it allows the execution of an unmodified OS. The hypervisor completely simulates the underlying hardware environments by employing two-stage address translation over sensitive instructions and allowing direct execution of non-sensitive instructions. Multiple hypervisors such as KVM [38], XEN [43], HYPER-V [44], between others explore these virtualization techniques;
- **Paravirtualization**: In this approach, VMs are aware that they are running in a virtualized environment. This technique is less portable than fullvirtualization since the guest OS requires modifications. However, since OSs rely on hypercalls to execute sensitive instructions, they show better performance. Specific versions of Xen use this approach [45];

2.1.3 Trap-and-emulate virtualization technique

As already mentioned, using virtualization, the VMs operates at lower privilege levels with a restricted behavior controlled by VMM (over critical resources), i.e., VMM controls all sensitive or privileged instructions' execution by VMs. Executing these instructions at a lower privilege level will make the system trap, and VMM will perform the trap-and-emulate technique [42]. This technique consists of taking the necessary actions at the VMM level, when sensitive or privileged instructions are executed on the guests located at low privileges levels. Figure 2.2 depicts the system behavior when a trap occurs. First, the execution the Guest-OS tries to execute a sensitive/privileged instructions. Then, the VMM traps the event and emulates the expected behavior by protecting the system integrity. To complete, the VMM returns the control to the Guest-OS, switching the context to a lower privileged level.

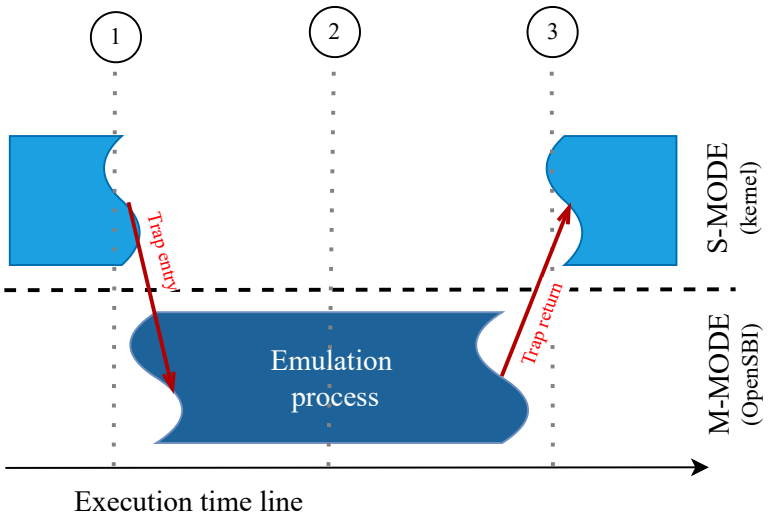


Figure 2.2: Trap-and-emulate steps flow.

It's important to highlight the high overhead that these transitions bring to the performance evaluation of the system [46] [47]. These transitions between the hypervisor and its guests bring latency problems, forcing hypervisor designers to implement strategies to reduce the frequency and cost of these transitions as much as possible [19] [48].

Latency of transactions

When a trap occurs, multiple steps are performed: (i) the CPU collects and saves its prior state; (ii) jumps to a hypervisor privileged layer; and then (iii) restores the state before trap. All process impose direct costs on the system, requiring CPUs to switch consequently between multiple modes and privilege levels. Furthermore, the cache gets filled with instructions and data belonging to the different execution layers, resulting in a phenomenon known as cache pollution [49]. One way of reducing the frequency of traps is to direct mapping the device memories to guests. This method is implemented by the Xen hypervisor when producing a privileged guest to whom the device drivers are allocated [48]. Additionally, this process mitigates the risk of device driver failure/misbehavior and tackles issues with I/O device dependability, maintainability, and management [49].

2.1.4 Memory Virtualization

Nowadays, Arm satisfies memory virtualization requirements. Its design includes a hypervisor mode to manage the physical hardware while ensuring the security of guests and adds a virtualization extension (Arm VE) [50, 51] for VM support. The hypervisor mode enables the simplest feasible design of hypervisors and chooses the level layer handling of each event. For example, when software executes in the hypervisor mode, it can configure whether to perform memory access at the hypervisor privilege level or the guest privilege level.

Arm enables memory translation via virtualization extensions, either through software support, using shadow tables, or through hardware support, by employing a two-stage address translation and an MMU. This latter consists of two stages, wherein the first stage, the OS performs translations from VAs to GPAs, and in the second stage, the hypervisor performs translation of GPAs into a PAs.

The shadow tables technique [47] does not use the MMU and the hypervisor, which is responsible for handling the Translation Lookaside Buffer (TLB), directly mapping the VA to a PA (eliminating one level of indirection). With two-stage translation, the hypervisor no longer has to maintain shadow tables in memory, leaving address translation to MMU. This results in considerable performance advantages since there is a significant reduction in the VM exit frequency and performance overall [46].

2.2 RISC-V

The RISC-V is an open standard ISA based on the principles of the Reduced Instruction Set Computer (RISC) [14]. It was created in 2010 in response to the need for a readily expandable architecture, facing market commercial designs, extremely complicated and fraught with legal issues [14]. Being classified as an open ISA, it can be freely used, enabling companies and individuals to implement their own RISC-V core without the need of paying any royalties. Due to its widespread expansion, it has been used to solve embedded systems issues in the academic and industry [20, 52, 53].

2.2.1 RISC-V specifications

The RISC-V ISA is made of two specifications: unprivileged [54] and privileged specifications [18]. The unprivileged specification covers the implementations with non-privileged base instructions. In RISC-V, unprivileged instructions are typically executed in the user mode (U-mode), involving standard extensions described as I (Integer), M (Integer Multiplication and Division), A (Atomic), F (Single-Precision Floating-Point), D (Double-Precision Floating Point), G (General Purpose, I+M+A+F+D), and C (Compressed Instructions), where each, defines the instruction set on a given implementation.

On the other hand, the privileged specification includes privileged instructions dedicated for privileged modes, including the machine mode (M-mode), which executes the most trusted code, and the supervisor mode (S-mode), which supports OSs such as FreeBSD, Linux, and Windows.

2.2.2 Privileged Modes

The privilege level dictates the number of features that the current hart can access. The RISC-V architecture supports three distinct privilege levels [18]: (i) the M-mode, (ii) the S-mode, and (iii) the U-mode. The highest privilege level and only mandatory is the M-mode, commonly intended for firmware devices while acting directly in the hardware. Typically, firmware executes in M-mode, operating only with PAs, and not using virtual address translation. Other privileged levels are the S and U-mode, where both

protect Supervisor Execution Environment (SEE) from OS, and OS from their applications, respectively [17, 18]. Typically, the S-mode privileged level is used to run OSs and provides VM capabilities through a dedicated MMU, and the U-mode for conventional applications. In summary, privilege levels exist to protect the different components of the software stack. Each privileged level enables the execution of specific extension-related instructions, and in an attempt to perform operations not allowed by the current privilege mode, the system will cause an exception.

To enable efficient hosting of guest OS on top of a type-1 or type-2 hypervisors, the S-mode level extends to the hypervisor-extended supervisor mode (HS-mode) [18]. The HS-mode behaves similarly to S-mode but with supplementary instructions and Control and Status Register (CSR)s that regulate the two-stage address translation, allowing the guest OS to host in virtual S-mode (VS-mode) [21]. As mentioned in Chapter 2.1.4, the two-stage address translation process splits into two stages: (i) GVA to GPA through OS; and (ii) the GPA to PA through the hypervisor. One of the main goals of this dissertation is to avoid the HS-mode extension by only use M, S and U-mode levels, and consequently perform the address translation in only one stage. As a result of this para-virtualization approach, the partitions are aware that they are running in a physical memory and its interaction with the host is optimized to avoid excessive trapping.

2.2.3 Software Stack Terminology

Due to this dissertation's privilege levels RISC-V architecture handles a stack terminology, with each layer presenting a system running either in U, S or M-mode. A system operating at a higher privilege level must have more permissions than a system operating at a lower privilege level. As a result, they must be able to access their functionality without the need for a communication interface. However, the contrary is not applicable; for a lower privilege level to interact with a higher privilege level, it requires a communication binary interface. Figure 2.3 illustrates a stack model that boils down to an Computer Application (APP) running on top of an execution environment. To communicate with the Aplication Execution Environment (AEE), APP uses an Aplication Binary Interface (ABI), which provides a set of user-level ISA and where is normally used to abstract the details of the AEE from the APP itself.

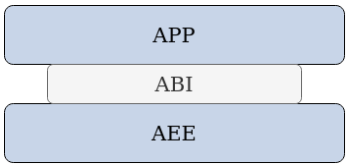


Figure 2.3: Stack model of an Application Execution Environment.

In comparison to Figure 2.3, Figure 2.4 depicts a traditional system stack capable of operating numerous APPs reporting a more complex stack model. It incorporates the previously described paradigm, in which the APPs communicate with AEE via ABI and add an OS together with an SEE, which composes the AEE. As a result, the OS manages its APPs and services for secure system execution; the APPs interact with the OS via ABI, and the OS communicates with its SEE via SBI. The SEE can be assumed as a

hypervisor provider on a high-end server or even a thin layer of software capable of providing to the OSs, all run-time services after boot.

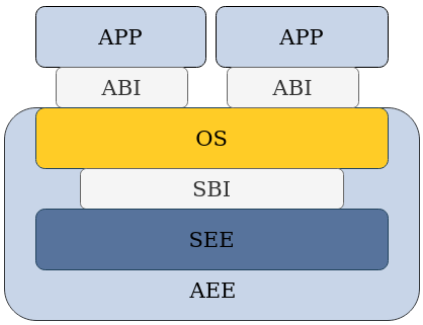


Figure 2.4: Stack model of a Supervisor Execution Environment.

2.2.4 RISC-V CSRs

CSRs in the RISC-V architecture take control of execution mode behavior and trap handling, composing the majority of the privileged specification [18]. To modify the state of CSRs, it is required to leverage pseudo-instructions like Atomic Read and Set Bits in CSR (CSRRS), Atomic Read and Clear Bits in CSR (CSRRC), and Atomic Read/Write CSR (CSRRW) [54]. Harts observe their own CSRs, producing illegal instructions when trying to execute non-associated privilege mode instructions. As this dissertation only touches on M, S, or U-mode privileged levels, below, there is a description of some RISC-V architecture CSRs crucial for its development. The letter *X* represents the privileged mode, either M or S.

- **Xstatus** (Status Register): The *Xstatus* register contains the most important information about the execution context. It enables/disables global interrupts; It stores context execution prior to the trap; It enables virtual-memory management operations;

- **Xie/Xip** (Interrupt Enable/Interrupt Pending Register): The *Xie* and *Xip* control software, timer, and external interrupts independently. Bits in the *Xie* register enable/disable the interrupts in the associated privileged layer. In the interrupt trigger event, it is natural that one of the present bits in the *Xip* register change to set, holding the interrupt until being claimed by an interrupt routine;

- **Xcause** (Xcause Register): Each trap sent by the system, whether synchronous or asynchronous, may be decoded as an interrupt or exception respectively, as seen in Table 2.1 (Adapted from [18]). The CSR *xcause* register is a register that indicates the cause of the trap event, with each bit allocated to a different cause;

- **mideleg/medeleg** (machine trap delegation registers): By default, M-mode is used to handle all traps regardless of the privilege level. The *medeleg* and *mideleg* CSRs enable the delegation of exceptions and interrupts to lower privilege levels, respectively. To force an interrupt/exception to

Table 2.1: Machine cause register (*mcause*) values after trap.

Interrupt	Exception Code	Description
1	0	Reserved
1	1	Supervisor software interrupt
1	2	Reserved
1	3	Machine software interrupt
1	4	Reserved
1	5	Supervisor timer interrupt
1	6	Reserved
1	7	Machine timer interrupt
1	8	Reserved
1	9	Supervisor external interrupt
1	10	Reserved
1	11	Machine external interrupt
1	12–15	Reserved
1	16	Designated for platform use
0	0	Instruction address misaligned
0	1	Instruction access fault
0	2	Illegal instruction
0	3	Breakpoint
0	4	Load address misaligned
0	5	Load access fault
0	6	Store/AMO address misaligned
0	7	Store/AMO access fault
0	8	Environment call from U-mode
0	9	Environment call from S-mode
0	10	Reserved
0	11	Environment call from M-mode
0	12	Instruction page fault
0	13	Load page fault
0	14	Reserved
0	15	Store/AMO page fault
0	16–23	Reserved
0	24–31	Designated for custom use
0	32–47	Reserved
0	48–63	Designated for custom use
0	64	Reserved

be treated at a lower privilege level, the associated bit cause in the *mideleg/medeleg* register must be set. In the context of this dissertation, this CSR only operates in M-mode and all interrupts, i.e., timer, software and external interrupts, are delegated to S-mode.

- **Xtvec** (trap-vector base-address register): The CSR *Xtvec* is the register responsible for the configuration of the traps vector. It can be implemented in S and M-mode and is composed of a *base* and a *mode* fields. The *mode* field specifies the destination address of the Program Counter (PC) when a trap occurs and can be classified as *direct* or *vectored*. When the *mode* field is *direct*, the destination address assumes the *base* field value, often associated with a handler function for both exceptions or interrupts concurrently. Meanwhile, in the Vectored *mode*, the PC value will assume $(base + 4 \times xcause)$ in an interrupt scenario and the *base* field value in an exception case.

- **Xtval** (trap value register): The *Xtval* contains exception-specific data that aids software in managing the trap. Moreover, the implementation never writes into *Xtval*, though it may be deliberately written by software. The hardware platform will define which exceptions must informatively set the *Xtval* and which may set it to zero unconditionally.

- **Xatp** (address-translation and protection Register): The *Xatp* CSR's role is to control the S or U-mode memory translation and protection mechanism. This register encompasses (i) the operation mode to define an address-translation scheme, (ii) an Address Space Identifier (ASID) to facilitate address translation, and (iii) a Physical Page Number (PPN) to define the root page table. The translation mechanism used in this dissertation follows the Sv39 operation mode, which involves page-based 39-bit virtual addressing with a three-level page table.

CSRs Trap behaviour

When only M, S, and U-mode are involved, every time a trap occurs over S-mode, the system follows a set of steps that involves multiple CSRs: (i) the *scause* contains the trap's cause; (ii) the *sepc* contains the last PC position; (iii) the *stval* contains an exception-specific data value; (iv) and the *sstatus* disables interrupts and collects the privileged layer before the trap. At the end of the trap handler, the *mret* instruction is performed, restoring the execution position before the trap.

2.2.5 RISC-V Memory Protection Support

A complete memory mapping system has various sizes of addresses, classified as (i) memory regions and (ii) control register memory mappings in the address space. These addresses have attributes classified as Physical Memory Attributes (PMA)s with a variety of metrics, such as the support for access lengths, atomic operations, and even the classification of various types of access [18]. The system operation rarely changes these metrics, and some of them are even fixed at chip design time, which is the case for on-chip Read-Only Memory (ROM).

Having control of physical memory accesses could contribute to the establishment of isolation and simultaneously support the safety of multi-core systems by limiting eventual faults and enforcing confidential/trusted/secure computing [55, 56, 57]. Since then, another check is analyzed simultaneously with PMA, i.e., the PMP. This standard RISC-V feature allows controlling hart accesses (read, write and execute operations) from protected memory regions according to its current privilege level. It only operates in M-mode, and unlike the PMA attribute variation, the PMP approach allows modularity in changing protected regions, i.e., different memory regions can be created or modified at runtime.

In summary, due to the containerization of the PMP unit [58], each partition does not need to run under virtual machines (no HS-mode) or use virtualization extensions to have protection. Each hart can manage different memory accesses without passing through a hypervisor layer while providing isolation between partitions. When running a GPOS over the S-Mode privileged layer, the address translation is done directly from VA to VA through the MMU, avoiding the two-stage address translation imposed by HS-mode. Thanks to the PMP security technology, this dissertation does not require HS-mode to separate distinct memories across many harts.

Numerous systems have used PMP to protect memory regions. For example, Keystone [22] leverages numerous PMP entries, to protect each enclave from the rest of the system, including the privileged OS, and to manage shared memory regions. Keystone employs multiple PMP policies, including whitelist-based or prioritized address matching, to build a customizable memory isolation strategy. Consequently, it is reasonable to assert that Keystone's whole security promise is contingent on PMP's functional correctness and not leveraging virtualization extensions.

Access Violation Attempts

PMP checks run on all physical address access attempts, whether in U or S-mode. Hence, when a PMP entry violation occurs, a trap is always sent directly to the processor, prohibiting malicious accesses on protected memory areas. Despite the security provided for low-privileged layers, enforced PMP permissions also have the potential to restrict the M-mode privilege level when setting a locked bit at each PMP entry. In short, the functionality of the PMP mechanism is to grant permissions to S-mode and U-mode, which by default have none, and suppress permissions from M-mode, which by default have all [18].

Physical Memory Protection CSR'S

For the PMP mechanism configuration, the RISC-V architecture provides two registers capable of satisfying a memory region, a configuration register, *pmpcfgx*, and an address register, *pmpaddrx*, only executed at M-mode. As illustrated in Figure 2.5, the *pmpcfgx* holds a set of parameters to control whether access is revoked or granted (W, X, and R bits), parameters to control the PMP entry range (bits A), and others to lock against the M-mode (bit L). The *pmpaddrx* register encodes the address matching, defining the length of a memory region.

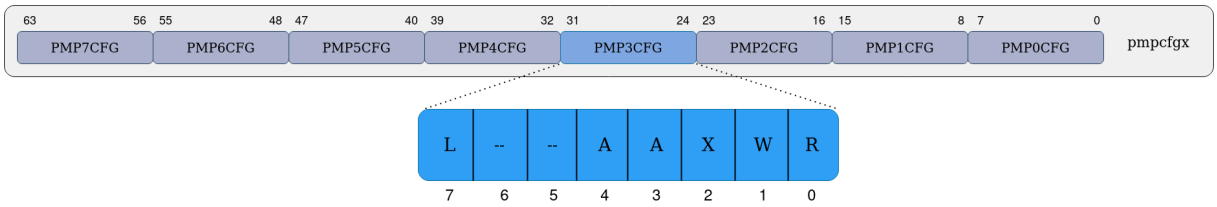


Figure 2.5: RV64 PMP configuration register layout.

According to Figure 2.5, a *pmpcfgx* in a 64-bit RISC-V architecture has eight distinct PMP entry configurations, where the least significant registers take precedence over the most significant registers. Additionally, the smaller the region's size, the higher the priority of the PMP entry. When a processor in U or S-mode attempts to fetch an instruction or perform a load/store, the address is iteratively compared to each correspondent PMP region, starting with the first PMP entry (the lowest significant register of *pmpcfgx*). In the absence of a correspondence match, the address is verified with the following PMP region. This procedure is repeated until the last PMP entry, and if it is still unmatched, the PMP check revokes the execution of the address access attempt.

Address matching

The *pmpcfgx* register's bits four and five, in combination with the *pmpaddrx*, specify the width of a PMP region. The RISC-V architecture supports several forms of address matching, including (i) the Top Of Range region) (TOR), (ii) Naturally Aligned Four-byte region (NA4), and (iii) Naturally Aligned Power-Of-Two region (NAPOT) region. The TOR is a type of address matching in which, as the name implies, it is an arbitrary range definition, that the top and bottom addresses are arbitrarily defined. For example, an attempt to access a given address j is granted when $pmpaddr_{i-1} \leq j < pmpaddr_i$. In case of the PMP entry is zero, the match will be in range at any address smaller than $pmpaddr_0$, in other words, $j < pmpaddr_0$. As alternatives, NA4 and NAPOT style address matching defines the width of the PMP region with just one *pmpaddrx* register. To calculate the value of *pmpaddrx* for each region:

$$pmpaddr = (base \gg 2) + ((size - 1) \gg 3) \tag{2.1}$$

$$pmpaddr = (base \gg 2) + (size \gg 3) - 1 \tag{2.2}$$

To explain each parameter in equations 2.1 or 2.2, it will take as an example of a PMP entry creation. Since the correct operation of the system depends on the bootloader phase, its region is classified as a trusted zone memory, and hence, it is necessary to protect it and maintain its integrity with a PMP region. In this PMP entry creation, the *pmpaddrx* value will contain a result of both expressions. The *base* value will carry the system booting address, and the *size* value will assume the width of the bootloader region, assuming a maximum number of 2^{64}, covering the entire memory system. Maintaining an environment

with isolated regions has always been crucial for the integrity and protection of systems with mixed-criticality requirements.

The PMP unit in RISC-V architecture emerges as a solution, forming regions and hence facilitating the process of system partitioning. However, such a scheme has several drawbacks that limit its use in general-purpose computing. Because it supports a limited number of PMP entries [18, 59] and consequently a limited number of regions, it may not be scalable for complex applications.

2.2.6 RISC-V Interrupts

As discussed previously, RISC-V interrupts are composed of software, timer, and external interrupts. Each time an interrupt is triggered, the corresponding bit in the *mcause* is set, helping the interrupt handler process. However, when *mideleg* delegates interrupts to S-mode, also *scause* bits are changed when an interrupt is triggered. Two hardware modules handle these interrupts: the Core Local Interrupt (CLINT) for software and timer interrupts and the PLIC for external interrupts.

2.2.6.1 CLINT

CLINT [1] is the core-level interrupt controller in the majority of RISC-V systems, only operating in M-mode. S-mode implementations must configure timer interrupts and issue Inter Processor Interrupt (IPI)s via an SBI, invoked through Environment Call (ECall) instructions. Moreover, in S-mode implementations is expected that M-mode firmware injects these interrupts into S-mode when interrupts are triggered. First (i) the interrupt traps to machine software, which must then (ii) inject the interrupt into S-mode location through the interrupt pending bitmap, and then (iii) inject it in S-mode by setting the corresponding sip bit.

2.2.6.2 PLIC

PLIC [2] is a system level interrupt controller capable of assisting harts in handling the external interrupts in the majority of RISC-V systems. The PLIC memory map has the layout depicted in Figure 2.6, where each colored section distinguishes different regions, the *priority*, the *pending*, the *enable*, the *claim/complete*, and even the *threshold* regions. The PLIC multiplexes different external interrupts up to 1023 devices on the same hart context, where each source device connects either to the 32-bit PLIC register or only to the corresponding source bits, depending on the PLIC region. A PLIC region is composed of registers classified either as (i) *source registers*, which control the source priority and provide their status (*Priority* and *Pending* regions) or classified as (ii) *context registers*, which control and handle the context interrupts (*Enable*, *Claim/Complete*, and *Threshold* regions). In *source registers*, each device source connects to a single bit, in *pending* regions, or to a 32-bit register, in *priority* regions.

[1] CLINT: https://github.com/pulp-platform/clint
[2] PLIC: https://github.com/riscv/riscv-plic-spec/blob/master/riscv-plic.adoc

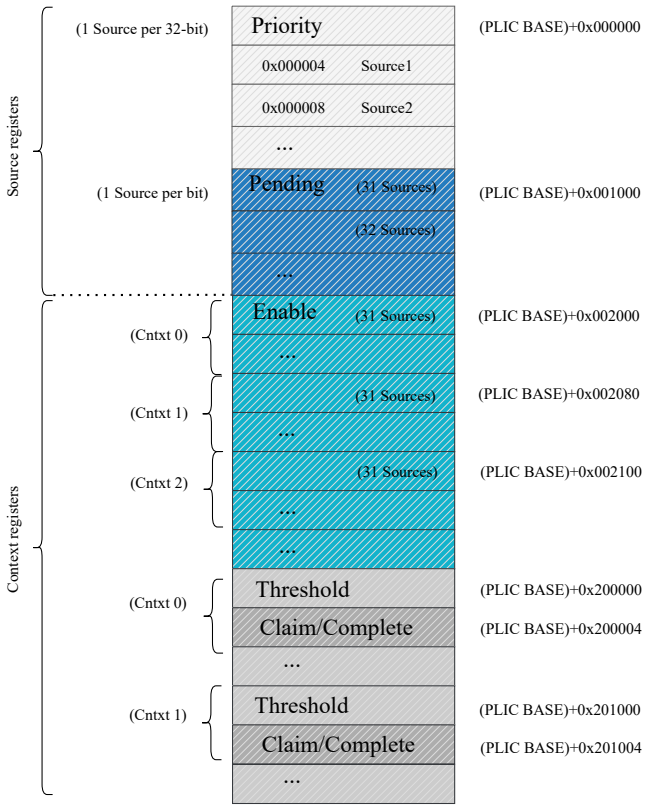


Figure 2.6: PLIC memory-map layout.

In a partitioned system, due to the arrangement of source devices on the PLIC and the context-dependency of its registers, PLIC memory space tends to be shared across partitions, breaking its spatial isolation. For example, a malicious partition could alter the execution of others by writing into PLIC enable registers of a neighbor partition (in a different context). This dissertation focuses on this memory component by establishing a spacial isolation system between partitions and leveraging virtualization techniques such as trap-and-emulate (mentioned in Section 2.1.3). Each PLIC register is described below, including its location inside the PLIC structure, its purpose for external interrupt management, and whether it is a context-dependent register.

- **Priority**: It is at the top of the PLIC structure. Each 32-bit register is aligned by four and linked to a single device. If the content of one of these registers is zero, the interrupt will never be triggered. Otherwise, the minimum value it can assume is one, and the maximum value is explicit either at the implementation or defined in the Device Tree (DT) description. In concurrent interrupt cases with the same priority, the interrupt ID breaks ties amongst global interrupts. Interrupts with the lowest ID have a higher priority and lowest priority register location. Each *Priority* interrupt source position is calculated through $(PlicBase + 0x4 \times Id)$;

- **Pending**: One level down in the PLIC structure is the *Pending* registers, in which each 32-bit register defines 32 source devices, where each bit is assigned to a device according to its associated

interrupt ID, except bit zero of the first *Pending* register, which must have no associated interrupt. If the bit one of the first register is set, the device with ID one triggered an external interrupt. A *Pending* bit is cleared as soon as the associated interrupt is claimed. Each interrupt *Pending* bit is located at an address, which follow the position $(PlicBase + PendPlicBase + (0x4 \times (Id/0x20))$;

- **Enable**: As a *Pending* register, each 32-bit *Enable* register defines 32 source devices, where each bit is assigned to a device according to its associated interrupt ID, except bit zero of the first enable register, which must have no associated interrupt. If the bit one of the first register is set, the device ID one will be able to trigger an interrupt. Otherwise, the interrupts will be disabled. Moreover, despite the bit location *Enable* registers also depends on context. Higher the hart ID or lower the privileged mode, the higher the enable PLIC register address. The offset between different contexts is $0x80$. Each interrupt *Enable* bit is located at an address, which follow the position $(PlicBase + EnbPlicBase + (0x80 \times Cntxt) + (0x4 \times (Id/0x20)))$;

- **Threshold**: Each *Threshold* register follows a 32-bit register, where the PLIC will mask all interrupts with a priority less than or equal to the threshold value. When a threshold value of zero, PLIC permits all interrupts with non-zero priority. It is a context-dependent register since the offset context between different *Threshold* address registers is $0x1000$. Each threshold address position in PLIC structure should be follow the $(PlicBase + ThreshPlicBase + (0x1000 \times Cntxt)$;

- **Claim/Complete**: It follows the *Threshold* registers. This register contains information about two processes related to interrupt handling: (i) the claim and (ii) the complete process. Once an interrupt ID is pending and the conditions are appropriate for handling, (i) the interrupt ID with the highest priority is claimed, automatically clearing the associated interrupt *pending* bit. The (ii) complete process is summarized by the completion of interrupt handling, referring to the end of the interrupt time handling. The position of *Claim/Complete* registers follow the $(PlicBase + ThreshPlicBase + (0x1000 \times Cntxt) + 0x4$;

2.3 OpenSBI

The PMP modularity plays a critical role in spacial isolation protection against malicious attacks [20, 56]. To establish static partitioning systems in RISC-V without hypervisor extensions and leveraging PMP unit, OpenSBI presents as a thin layer of software running in M-mode. This chapter contextualizes OpenSBI within the software stack and explains its significance for embedded system security. This software has been growing over time [3], adding more and more features to it and facilitating the use of different RISC-V platforms. Its main contributions [4] are:

[3] RISC-V Boot Process: https://www.slideshare.net/atishpatra/riscv-boot-process-one-step-at-a-time
[4] OpenSBI Deep Dive: https://www.youtube.com/watch?v=jstwB-o9ll0

- Establish communication between different privilege layers through the runtime services in M-mode. Ensuring security and protection against sensitive instructions;
- Open-Source;
- Support the platform adaption for several multiple RISC-V platforms. Providing reusable common code of different RISC-V platforms, aiming to assist a secure boot;

2.3.1 OpenSBI as a Runtime stage

Before any operating system can start, a boot flow with numerous boot stages must commit a set of system requirements. Any embedded system will always have a multistage boot flow [60], with each stage devoted to a specific set of tasks. A typical boot sequence is represented in Figure 2.7, where the first stage is referred to as the (i) Zero-Stage Boot Loader (ZSBL) or the ROM since it operates on the ROM. It initializes clocks, manages system power, and resets the system. After the voltages have steadied and the hardware is ready to begin booting, the processor initializes the hardware buses and peripherals. It is the initial code run by the CPU, and it embeds all of the logic required for the following boot stage through external peripherals such as an embedded Multi-Media Card (eMMC), microSD card, or even via specialized protocols on a bus for data transfer (like USB, UART, and others). Nextly has presented the (ii) *LOADER*, commonly known as the First-Stage Boot Loader (FSBL). It initializes the DDRs and loads the subsequent stages, including the *RUNTIME* stage. Though the FSBL does not need regular updates, any change may result in unexpected behavior and could jeopardize the board. The (iii) *RUNTIME* stage executes all safe boot flow components on top of the on-chip Static Random Access Memory (SRAM), and it is assumed to be the second stage boot loader. Software layers such as U-Boot [5], and OpenSBI may commit to this, as well as being responsible to provides runtime services to the OSs following system boot.

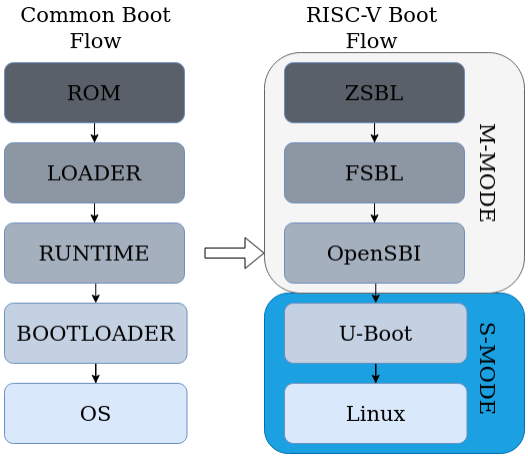


Figure 2.7: Common Boot Flow to RISC-V Boot Flow.

[5]The universal boot loader: http://www.denx.de/wiki/U-Boot

Runtime services: It translates as runtime services [6] to all the communication services that lower privilege layers do to higher privilege layers. Typically, they are carried out in S-mode using the previously stated ECall instructions through an SBI layer.

The penultimate boot stage contains the (iv) *BOOTLOADER*, which loads kernel images from media like SD cards or networks. Grub [61] is an example of a software bootloader capable of loading Linux kernel images from Multi-Media Card (MMC) devices. The last step introduces the (v) OS, which typically operates in S-mode and handles all non-privileged applications. Regarding all boot stages, this dissertation will focus on the *RUNTIME* boot stage, where OpenSBI operates in M-mode and provide services to lower privilege levels, as seen in Figure 2.7, where it supports both U-Boot and Linux OS running in S-mode.

2.3.2 SBI Specifications

In RISC-V, (i) to establish inter privileged layers communication, (ii) make the S-mode privilege level transferable across many implementations, and (iii) to provide run-time services, M and S-mode must comprehend SBI specifications [7] [62]. The OpenSBI contains additional code to answer the ECall instructions from S-mode, providing run-time services and complying with the SBI specifications. These are composed of extensions and functions, which are binary encoded by registers $a6$ and $a7$. Each function value, the SBI Funtion ID (FID), is represented by register $a6$ for a particular extension value in $a7$, the SBI Extension ID (EID), indicating that each extension has several functions.

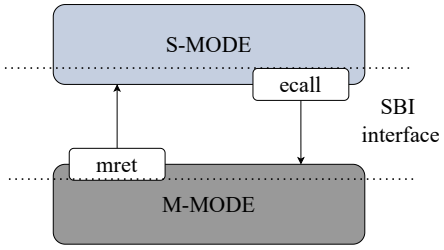


Figure 2.8: SBI interface between privileged layers.

When S-mode software executes an ECall, a transition from a lower privilege level to a higher privilege level occurs and encoded information by registers EIDs and FIDs reaches the implementation of these SBI services. As seen in Figure 2.8, each time the processor executes an ECall instruction in S-mode, the machine's state transits from S to M-mode. On the other hand, after performing a run-time service, the machine's state is transferred from M-mode to S-mode through *mret* instruction. In the case of an unsuccessful return, the system follows the calling convention return type, where the return error goes to register $a1$, and the associated value goes to $a0$ register, accordingly to Table 2.2. As an example case, the executing of an ECall instruction that does not correctly encode the SBI extension through register $a7$, the value -2 is returned, corresponding to the error *SBI_ERR_NOT_SUPPORTED*.

[6] OpenSBI Bootflow: https://www.youtube.com/watch?v=sPjtvqfGjnY&t=2364s
[7] RISC-V SBI Specifications: https://github.com/riscv-non-isa/riscv-sbi-do

Table 2.2: SBI specifications: associated returned values.

Error Type	Value
SBI_SUCCESS	0
SBI_ERR_FAILED	-1
SBI_ERR_NOT_SUPPORTED	-2
SBI_ERR_INVALID_PARAM	-3
SBI_ERR_DENIED	-4
SBI_ERR_INVALID_ADDRESS	-5
SBI_ERR_ALREADY_AVAILABLE	-6
SBI_ERR_ALREADY_STARTED	-7
SBI_ERR_ALREADY_STOPPED	-8

OpenSBI adheres to several functionalities presented in several extensions as:

- **Base extension:** Since base extensions do not report errors, any RISC-V implementation can perform base extensions. They examine information about the implemented extensions, and it can identify by the EID $0\text{x}10$;

- **Legacy extension:** Legacy extensions are represented by EIDs between $0\text{x}0$ to $0\text{x}f$. This extension provides significant support for S-mode implementations like (i) updating the timer values through the *stime_value* parameter (since timer register handling is only possible in M-mode [19]), providing (ii) access to the serial port connected to the OpenSBI, allowing the reception and sending of characters through the SBI ECall instructions, providing (iii) IPI's functionalities, used for example in FreeRTOS environments when tasks yield and need signaling current hart for task scheduling [63], providing (iv) synchronization in the execution of instructions by the different harts through the *FENCE* instructions, and finally, allowing (v) the system shut down through a simple system call;

- **Hart State Management Extension:** A current platform hart assumes a state that reveals its activity. A hart state can be characterized as *started*, which initiates normal execution, *stopped*, which suspends execution in S and U-mode, or *suspended*, which consumes less power [62]. When hart is transitioning from the *stopped* to the *started* state, it may enter the *start pending* state. In contrast, the hart can enter the *stop pending* state when it is transitioning from the *started* to the *stopped* state. When its transition from *started* to the *suspended* state, it assumes the *suspended pending* state, and when it transitions from *suspended* to *started*, it assumes the *resume pending* intermediate state. This module adds a collection of functions to the S-mode environment for controlling the state of the harts and their transitions. This extension provides functions for initializing, suspending, and terminating the execution of harts, as well as their state. Their return value may indicate success, an invalid argument, an invalid address, that the resource is already available, that the resource has failed, or that the resource is unsupported;

- **System Reset Extension:** In some specific cases like system failure, SBI implementation-specific reset reason, or vendor or platform-specific reset reason the system needs to execute the reset functionality to restore normal execution. The system reset extension allows the reset of both SEE and SBI implementation in M-mode. In other words, it is possible to reset the system both in OpenSBI and in S-mode;

2.3.3 Multi-platform support

Due to the manufacturers describe their hardware platforms in different specifications, it forces the development of modular and adaptable software systems for any platform. One of the OpenSBI advantages is the multi RISC-V platform support through distinct configurations. Typically, these configurations are done at the boot stage and follow two methodologies, either by inserting a pre-compiled DT, which depends on a default generic platform, or selecting one of the available platform implementations through compiler arguments. Due to the higher configuration flexibility, the platform portability, and the OpenSBI domain support, this dissertation will focus only on configuration via the DT [8] with a generic platform. The generic, because contrary to the other provided platforms, the OpenSBI implements the domain system via Flattened Device Tree (FDT).

2.3.4 OpenSBI domains

OpenSBI includes an integrated system capable of aggregating and associating a collection of memory regions to one or more harts, denominated as OpenSBI domains [9]. Therefore, the more harts available on the hardware platform, the more domains may be produced. To establish each domain memory region OpenSBI leverage the PMP unit following a non-conflicting memory region rule, i.e., do not allow domain memory regions overleaping. Over a domain, multiple domain memory regions might assign the same domain, and consequently, the more domain memory regions associated, the higher the probability of memory overlapping. The overlap of two or more memories happens when part of the memories coincide between them. In case of memory overlapping, one of the regions should be a sub-region of another, not having the same size and access flags.

The domain memory regions establishment occurs thanks to the PMP unit capabilities, which establishes various non-conflicting memory regions to individual harts. The domain support is done in M-mode, during boot time, and through DT configuration.

[8] DT: https://www.devicetree.org/specifications/

[9] OpenSBI domain system: https://github.com/riscv-software-src/opensbi/blob/master/docs/domain_support.md

2.3.5 DT Configuration

The DT is a tree-like data structure that describes machine hardware, following a child-parent hierarchy with numerous configurable properties inside each child (i.e. node), as illustrated in Figure 2.9. The top node of the DT hierarchy, also termed the root node, is denoted by the character "/" and indicates the

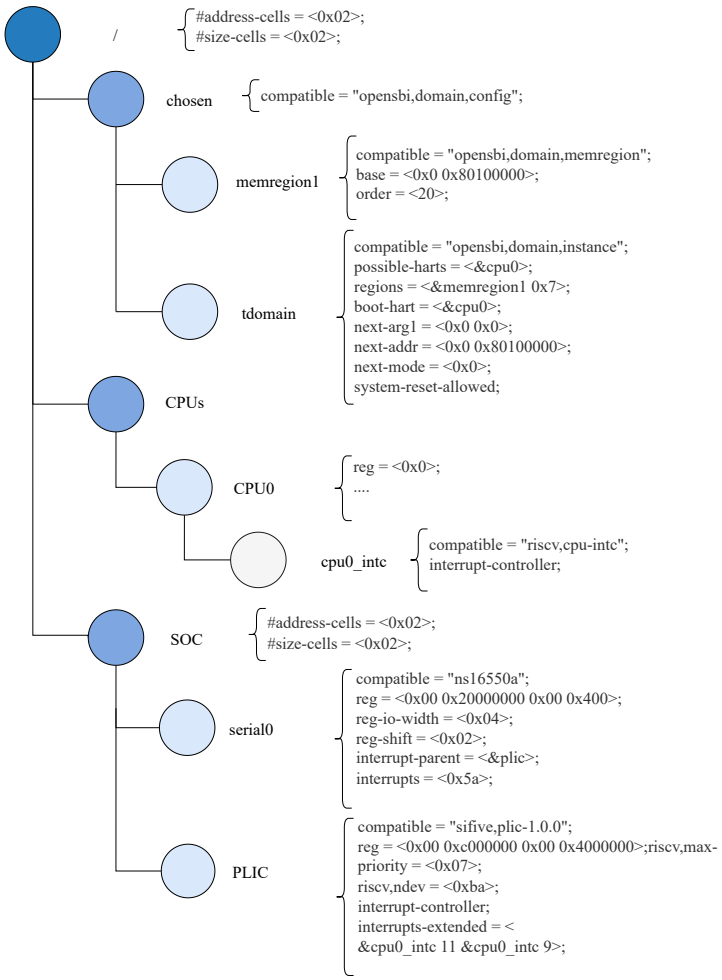


Figure 2.9: Example of OpenSBI domain system configuration over a DT.

start of the DT description. Inside this node are described their children, represented as CPUs, memories, devices, interrupt controllers, and others. As depicted in Figure 2.9, between these nodes is also the chosen node. The chosen node is not a physical device, but it is the DT location for transmitting data between firmware and the OS, such as boot parameters. Furthermore, this node is also leverage by OpenSBI to collect and support domains configurations.

DT Domain Configuration

Two distinct components define the OpenSBI domain configuration: the *domain instances* and the *domain memory regions*. *Domain memory regions* present the base of the static partitioning system

used in this dissertation. Typically, it encompasses device regions and MMIO regions. To establish them, OpenSBI (i) agglomerate all *domain memory regions*, (ii) sort them, and (iii) create a PMP entry for each. According to this approach, the lowest the regions, the lowest the number of the PMP entry and consequently, the higher the priority [18]. Regarding its configuration, each domain memory region must compose a base address value, a size value defined by 2^{order}, and the *compatible* syntax value: "$opensbi, domain, memregion$".

Table 2.3: Description of OpenSBI domain instances properties.

Property	Description
compatible	It should assume value: "Opensbi,domain,instance";
possible-harts	It defines which harts are allowed to manage this instance; All the harts described in this property are exclusively linked to this domain;
regions	It describes all *domain memory regions* assigned to this domain and their permissions; All *domain memory regions* described in this property follow the accesses encoded by the last four bits. The write/execute/read accesses are defined by the three least significant bits and bit fourth defines if the access is locked in M-Mode or not. This association is only possible thanks to the Locked bit mentioned in Section.
boot-hart	It defines which hart is responsible for booting the *domain instance*; If the primary hart is connected to this *domain instance*, the value set in this property will be ignored, and the primary hart will take over the startup of the next booting stage.
next-arg1	It sets the arg1 of the next booting stage. When DT configuration is available, its memory address is set in this property. If no DT is detected, this property is set to 0x0 by default;
next-addr	It sets the address of the next booting stage. After the boot stage is complete, OpenSBI firmware will pass the execution context to the address located at *next-addr*.
next-mode	It defines the next boot stage privilege mode. As privileged modes for the next boot stage, S and U-Mode can be selected. If this property is not given and primary hart is not assigned to this domain, the default value 0x01 (S-Mode) is assumed; if this property is not given and primary hart is defined in this domain, the next booting stage of the primary hart is taken as the default value.
system-reset-allowed	It sets this *domain instance* can reset the system.

Regarding the *domain instances*, they are responsible to provide each partition, associated with one or more exclusive harts and with one or more associated and non-conflicting *domain memory regions*, followed by their permissions. Figure 2.9 illustrates some configuration instance node properties, where *tdomain* has an associated hart (i.e. CPU0) with all permissions above an associated *domain memory region*, the *memregion1*. Regarding *domain instances* configuration, the following Table 2.3 contains all conceivable parametrization and their meanings.

OpenSBI Root Domain

Since the OpenSBI is present directly over the hardware memory, and since each *domain instance* should be aware of its incubated *domain memories*, OpenSBI should also have a *domain instance* reserved for its firmware storage. For this purpose, by default, at a very early boot process, OpenSBI establishes the root domain, a *domain instance* that does not depend on DT configuration; if there are no *domain instances* in DT, this domain will support OpenSBI by associating all platform harts and memories to it. OpenSBI always produces a root domain instance, even when configuration deliberates *domain instance* establishments. When *domain instances* are defined, the customization can range from none to several *domain instances*, as similarly occurs with the assignment of harts. OpenSBI assigns the harts to a *domain instance* from none to several harts. In a *domain instance* with no harts assigned, the root domain will support them and let them remain in it until the system is modified.

DT specifications

Besides this this OpenSBI domain parametrization, the DT example depicted in Figure 2.9 it describes one CPU node, one serial device node, and an external interrupt controller node, the PLIC. All of them are composed by different properties that describe the hardware in different ways. Starting by their memory position, the *reg* property, where its syntax is established by *#address-cell* and *#size-cell* properties, specifing how many cells are in each field of any *reg* child node. The *reg* property follows the syntax:

$reg =< [address1\ \ length1]\ [address2\ \ length2]\ [address3\ \ length3]\ \ ...\ \ >.$

In Figure 2.9, both the *#address-cells* and *#size-cells* assume the value of two, meaning that each *reg* property of any root child node needs two fields to represent the base location and two fields to represent its length. Taking the PLIC node as an example, its base address assumes a value of $0\text{x}c000000$ and a length value of $0\text{x}40000000$. Along with these properties, other device-specific interpretation properties contribute to development of this dissertation, such as *compatible*, *interrupts*, *interrupt-parent*, *interrupt-controller*, and *interrupt-extended* properties, where all are described over Table 2.4.

2.4 Related work

There are several solutions available today that aim to provide partitioning [22, 39, 43, 64] by using the strong primitives of the RISC-V Privileged architecture. There are RISC-V solutions that make use of the hypervisor extension, while others may be implemented on RISC-V systems without hardware virtualization support. This section leverages some of these types by classifying and comparing them with this dissertation work.

Table 2.4: Description of common DT properties.

Property	Description
compatible	Used to bind devices; It is mandatory; In some cases also informs the device version; Any DT node must contain this property to be identified by the OS or the boot entity;
interrupts	Used to identify the interrupt value triggered by the device, ranging from one to multiple values.
interrupt-parent	Used to identify the interrupt controller to which the devices' interrupts are wired; In Figure 2.9, the serial device *interrupt-parent* property connects with the PLIC;
interrupt-controller controller.	Every node that includes this property must be able to behave as an interrupt
interrupt-extended	Determines interrupt values triggered by the interrupt-controller and whether many or single interrupt controllers will handle them. In Figure, the PLIC will be able to trigger interrupts with the values $0x11$ and $0x09$. As its interrupt controller, the DT presents the *cpu0_intc* in *CPU0* node.

XVisor

XVisor [64] is a type-1 RISC-V hypervisor targeting embedded systems with soft real-time requirements. It has a complete monolithic design that supports both full virtualization and para-virtualization. This hypervisor was tested in QEMU [10] with H-Extension and provides features like CPU virtualization, guest I/O emulation, background threads, para-virtualization services, and device drivers run as a single software layer with no prerequisite tool or binary file.

Xen

The Xen [43] is an open-source type-1 hypervisor that already gives its preliminary steps in RISC-V. It boots from a bootloader like U-boot and produces VMs denominated as domains. Between domains is the Dom0, which controls the guest OS and Hypervisor at a privileged level, and the non-privileged VMs (DomUs). Recently, the Xilin has been working at dom0-lessx [11], which enables static partitioning configuration, reduces the boot time, and implies the device assignment, to each domain. Similarly to this dissertation, domain setup is performed through DT and incorporates the domain's boot and device assignment, which reduces hypercalls in paravirtualization processes.

Siemens' Jailhouse

Siemens' Jailhouse [65] pioneered the static partitioning hypervisor design. This design employs a minimum software layer that statically separates all platform resources and allocates them to a single VM instance using hardware-assisted virtualization technologies. Because each virtual core is statically pinned

[10] QEMU: https://www.qemu.org/docs/master/

[11] Dom0-less: https://xenproject.org/2019/12/16/true-static-partitioning-with-xen-dom0-less/

to a single physical CPU, no scheduler is required, and no expensive semantic services are supplied, reducing size and complexity. Although it may compromise the necessity for optimal resource use, static partitioning provides higher assurances in terms of isolation and real-time. Despite its design philosophy, Jailhouse falls short by relying on Linux to boot the system and operate, which suffers from the same other hypervisors' limitations. This hypervisor already ports to RISC-V by leveraging hypervisor extensions. Recently, SELENE [66] made use of Linux in cooperation with the SIEMENS Jailhouse to develop an open-source software layer on RISC-V platforms while allowing functional and temporal isolation properties between different critical applications.

Bao

Bao [39] is a security static partitioning type-1 hypervisor that aims to facilitate the consolidation of mixed-critical systems, thus focusing on providing strong safety and security guarantees. It is composed of a thin layer of privileged software that exploits ISA virtualization extensions to partition the hardware. It includes hardware-based methods such as cache coloring that help prevent interference between numerous guests. Bao is mainly developed for Armv8 architecture. However, it has been ported to RISC-V on QEMU and to RISC-V cores with hardware virtualization support (e.g., Rocket, CVA6). [19].

RVirt

RVirt [12] is an S-mode trap-and-emulate RISC-V hypervisor designed on QEMU and partially supports HiFive Unleashed. It is developed in Rust because it is a comfortable language and enables for direct targeting of bare-metal platforms. It has the problem of including a large amount of dangerous code during VM startup and entry/exit. This dissertation, as RVirt, is not using virtualization extensions and also use the virtualization trap-and-emulate approach.

VOSySmonitoRV

VOSySmonitoRV [67] presents as a RISC-V solution for mixed-critical systems allowing the co-execution of two or more OSs isolated from each other without implementing hypervisor extensions. It operates in M-mode meaning that it is responsible for managing the PMP entries to establish a security boundary between partitions. Similar to this dissertation, VOSySmonitoRV applies a similar static partitioning system, with an equal focus on security and fault-containment without H-extensions. The work proposed in this dissertation distinguishes VOSySmonitorRV by automatically partitioning the interrupt controller component among multiple harts during the device assignment process and leveraging the trap-and-emulation technique. Moreover, since this dissertation was developed over OpenSBI it carries its RISC-V platform modularity advantages and a static partitioning configuration through the DT.

[12] RISC-V hypervisor written in Rust: <https://github.com/mit-pdos/RVirt

Keystone

Keystone [22] is presented as a RISC-V open framework for Trusted execution environment (TEE)s. It utilizes a security monitor to enforce TEE guarantees and uses the PMP feature of RISC-V to isolate individual enclaves. Although the Keystone seems to be a trusted hypervisor, it is just considered a reference monitor. It does not implement or execute resource management, virtualization, or scheduling [22], and it simply verifies if its untrusted OS and enclaves are processing shared resources correctly. Keystone additionally includes a shared memory method for communication between the supervisor mode runtime and the security monitor. Similarly, this dissertation utilizes the PMP approach to create isolated static partitioning systems on RISC-V. However, instead of keystone, OpenSBI is used as a secure monitor.

Diosix

Diosix [13] is presented as a work in progress of a lightweight, fast, and secure multiprocessor bare-metal hypervisor written in Rust for 64-bit RISC-V computers. It leverages the PMP unit to achieve isolation between capsules (domains). Just as in OpenSBI, it parses DT for capsules configuration and to provide information about the available hardware, it provides an SBI syscall interface to communicate between system services.

seL4

The seL4 [68] is an open-source, high-performance, microkernel and hypervisor with the most comprehensive mathematical proofs of correctness and security. It is a microkernel and not an OS because of its minimal amount of privileged code. It provides control communication between different system components by following its principle of isolating applications from each other and executing them on a safe and trusted platform. As a hypervisor, it supports virtual machines by leveraging hypervisor extensions. However, recent implementations [69, 70] show that seL4 can avoid virtualization extensions while operating as a secure monitor at the M-mode level and establish boundaries between trusted and untrusted worlds through the PMP unit.

[13] Diosix: https://github.com/diodesign/diosix

3. HSP-V: Design

This section introduces the design of a RISC-V static partitioning solution while precluding the RISC-V HS-extension. It leverages an OpenSBI mechanism called OpenSBI domains to establish isolated static partitions and communication interfaces between higher and lower RISC-V privilege layers, without addressing virtualization extensions. This chapter starts by scrutinizing OpenSBI domains and exposing their isolation problems. Next, it proposes a solution that augments the OpenSBI with additional capabilities, aimed to cover all issues revealed in Section 3.1. To conclude, this dissertation analyze a static partitioning system with all enhanced features in OpenSBI.

3.1 MCSs with OpenSBI Domain System

As detailed in Chapter 2.3, OpenSBI is a thin layer of open-source software that provide runtime services through the implementation of SBI specifications. Moreover, the support of multiple domains by OpenSBI allows the establishment of MCSs on the same hardware platform without using virtualization extensions. Hence, each domain runs either in S or U-mode (not leveraging HS-mode) by only using PAs or managing the memory translation from VAs directly to PAs.

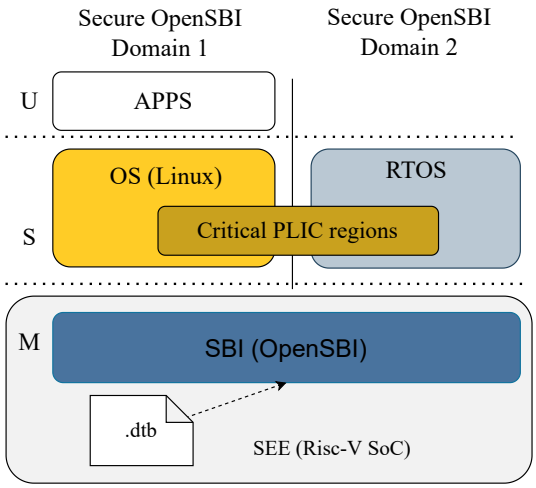


Figure 3.1: OpenSBI domains system support with PLIC critical issues.

Figure 3.1 depicts an OpenSBI domain system comprising two *domain instances* running in S-mode: (i) one containing an RTOS and (ii) the other with a GPOS with its applications running in U-mode. To support these two *domain instances* is placed the OpenSBI operating in M-mode, where use a DT to parse either the platform or the *domain instances*' features. During the boot, the DT configuration is evaluated through sanity checks, ensuring the non-existence of conflicting memory regions. Additionally, OpenSBI performs the isolation between domains by creating a PMP unit per domain memory region defined in DT configuration. With this approach, OpenSBI limit the access of each *domain instance* by ensuring the system to trap in each access attempt to an adjacent *domain memory regions*.

In this scenario, although each domain is assigned exclusive *domain memory regions*, some may be shared between them, as is the case of the external interrupt controller, the PLIC (mentioned in Section 2.2.6.2). Because the PLIC is an MMIO structure and needs to be shared among all domains, when *Domain1* or *Domain2* try to access PLIC memory, they can compromise each other behaviors, breaking the isolation between them. However, with the current domain system configuration provided by OpenSBI is impossible to isolate PLIC regions across different contexts. Typically, some PLIC registers have a device bit-map representation across a 32-bit register (e.g. *Enable* and *Pending* PLIC registers), with each bit encoding an interrupt device information. Furthermore, since each *domain memory region* is often created into the DT with the shortest range of four bytes (minimum size of a PMP entry), is not possible to partition these registers across domains. A more detailed explanation of all the critical PLIC regions is covered in Section 3.3.

This dissertation proposes a solution based on device assignment to different domains by leveraging a PMP unit and a virtualization technique to emulate the accesses from all domains that try to access each critical PLIC region. This chapter shows a design solution that leverages: (i) OpenSBI domains configuration enhancement; (ii) the application of the automatic device memory regions assignment; (iii) the partitioning approach over critical PLIC regions; and (iv) the trap-and-emulate virtualization technique to emulate each PLIC region access.

3.2 OpenSBI Domain Support

As mentioned before in Chapter 2.3.4, OpenSBI will be responsible for establishing the static partitioning system. Over it, will be used the OpenSBI domains feature, based on a set of memories associated with one or multiple harts. As also was mentioned in Chapter 2.3.5, to control these domains and statically configure the partitions, this dissertation will adopt a DT as a basis for OpenSBI configuration. Due to its flexibility, its platform portability, and the OpenSBI domain support, *domain instances* can be easily configured by defining a set of properties over a chosen DT node, as described in table 2.3. However, given the issue presented over Section 3.1, i.e., a minimum size of a PMP entry, the current OpenSBI configuration does not cover an approach to partitioning PLIC memory space in a multiple domain scenario. To bypass this problem and protect the PLIC memory space, an user grant the device MMIO regions and suppress

the PLIC memory space through *domain memory regions*. However, due to this configuration, the system will always trap every time the *domain instance* try to control the external device interrupts in PLIC space. Since the goal is to isolate the PLIC structure between different domains and not suppress the accesses to all PLIC structure, this is not the best approach. This section proposes the inclusion of a *domain instance* configuration property to support the PLIC partitioning and the direct device MMIO regions assignment by giving to user an easy-to-use DT configuration property for this purpose.

OpenSBI Configuration Enhancement

This dissertation proposes the insertion of property *possible-devices* over the DT *domain instance* node. The main goal is to assign one or several devices to a *domain instance* without the user having to worry about the inter-domain isolation (including the isolation of PLIC memory space). The *possible-devices* follow the same semantics as the *possible-harts* property, but instead of assigning harts, it assigns device MMIO regions, which indirectly includes the memory space dedicated to control its external interrupts (in PLIC). Figure 3.2 depicts a scenario where it assigns two devices, a *uart* and an *i2c* device, to a *domain instance* nominated *tdomain* through the property *possible-devices*. Due to this configuration scenario, the *tdomain* has permissions to access these devices' MMIO regions and simultaneously the PLIC information about them.

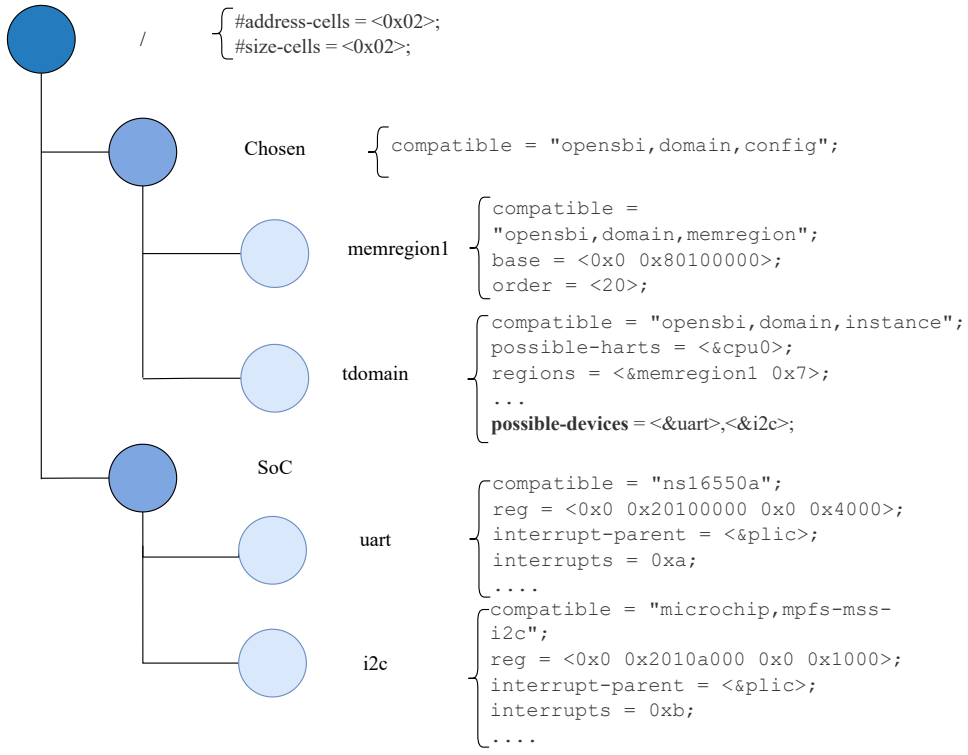


Figure 3.2: Device tree example including *possible-devices* property.

After the configuration, at boot time, the OpenSBI performs the DT binding, collecting the DT domain properties. Between them, this dissertation adds the *possible-device* property binding, where OpenSBI

parses each device on it. Resulting in a collection of their device interrupt ID values and their device memory locations. With this binding process, the *domain instance* contains all necessary information to (i) automatically create a *domain memory region* per device and (ii) reserve different IDs per *domain instance*. Reserving device IDs to an exclusive *domain instance* allows each domain to access only the correspondent PLIC regions where these IDs are present. Regarding the automatic device MMIO regions assignation, this process not only facilitates the user creation process but also prevents the possibility of user OpenSBI misconfiguration. By mistake, a user can configure a *domain memory region* that corresponds to another *domain instance* (added by the *possible-devices* property). However, (i) by the insertion of the device in *possible-devices* property, (ii) by the automatic device memory region assignation, and (iii) by the memory overlapping semantics of OpenSBI (mentioned in Chapter 2.3.5), in a misconfiguration scenario like this, OpenSBI will report a configuration error, stopping the execution before launching the domain execution.

3.3 PLIC Partitioning

After enhancing OpenSBI *domain instance* configurations by connecting MMIO regions to each domain, this dissertation also enhances OpenSBI to protect PLIC memory regions. This chapter begins by (i) evaluating the PLIC components and as their criticality, and ends by (ii) proposing a method for isolating them.

3.3.1 PLIC Registers Protection

In MMIOs, I/O devices are placed in memory space, allowing the processor to access devices identically as accessing memory. For processors to interact with external interrupts of devices, the PLIC needs to map the external interrupt device information in a reserved memory region. In a multiple *domain instances* scenario with multiple devices assigned for each instance, the PLIC poses as an MMIO structure and is to be shared between domains. Therefore, over PLIC memory space, there are critical PLIC regions that must be isolated between the *domain instances*; however, due to the current *domain instance* configurations, these regions are unable to be protected, as mentioned in Section 3.1. As mentioned in Chapter 2.3.4, *domain memory regions* are performed through PMP entries, and due to the RISC-V specifications, each PMP entry has a minimum size of four bytes (32-bits). As some PLIC registers have a device bit-map representation across a 32-bit register with each bit encoding an interrupt device information, as depicted in Figure 3.3, it is not possible to partition these registers through the *domain memory regions* creation. Typically, the PLIC regions represent two types of critical regions: (i) the regions which deal with different device interrupts per bit, and (ii) the regions that must be global between different *domain instances*. Any shared-required PLIC region between *domain instances* is a critical PLIC region. Below are the considered critical PLIC regions as their protection methodology.

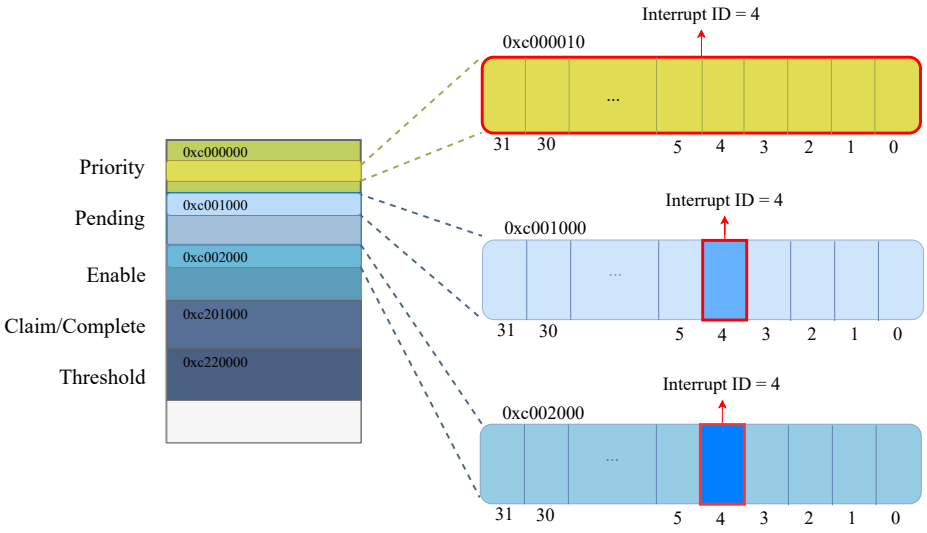


Figure 3.3: Encoding representation of critical PLIC registers.

Priority PLIC Register

The *Priority* region is considered a critical PLIC region. It provides several interrupt priorities across the structure. Despite each interrupt being connected to each 32-bit register and not a device per bit register, this PLIC region needs to be global across the entire system to inform the priority of each interrupt ID, as illustrated in Figure 3.3. To illustrate the priority of interrupt ID four, the PLIC needs a 32-bit register in address $0xc000010$ ($plic\_base + (4 \times interrupt\_id)$). To isolate the *Priority* region across *domain instances*, instead of creating *domain memory regions*, and to avoid wasting PMP entries, this dissertation uses the property *possible-devices* together with a virtualization technique, the trap-and-emulate, to emulate accesses by the *domain instances*.

Pending PLIC Register

The *Pending* region is considered a critical PLIC region. It plays an informative role concerning the associated device sources. In this register, each device interrupt connects to each bit of its 32-bit registers, as illustrated in Figure 3.3. To illustrate the pending interrupt with ID four, the PLIC just needs the bit four of the provided 32-bit register (in address $0xc001000$). To isolate the *Pending* registers across *domain instances*, this dissertation uses the property *possible-devices* and a virtualization technique, the trap-and-emulate.

Enable PLIC Register

The *Enable* region is considered a critical PLIC region. Similar to the *Pending* sources registers, the *Enable* registers also follow a bit-map organization. Where each bit can control a device interrupt ID, either enabling it, or disabling it. However, each *Enable* register position depends on the machine

context. The machine context is obtained through $cntxt = priv\_mode \times 2 + hart\_id$. To illustrate the *Enable* status of the interrupt with ID four, the PLIC just needs the bit four of the provided 32-bit register ($plic\_base + 0\text{x}2000 + (cntxt \times 0\text{x}80)$), as depicted in Figure 3.3. Due to the already mentioned PMP entry limitation, its bit-map representation will imply shared memory between *domain instances*. This dissertation protects the *Enable* registers through a virtualization technique, the trap-and-emulate.

3.3.2 PLIC Memory Regions Automatic Assignment

The critical PLIC regions face limitations of the PMP unit; however, not all PLIC registers are considered critical regions. The *Threshold* and the *Claim/Complete* are considered non-critical PLIC regions. To isolate them between several *domain instances* HSP-V establish boundaries through *domain memory regions*, i.e., this dissertation enforces the inter-domain isolation by direct assigning these PLIC regions to domains.

In spite of non-critical PLIC memories can be deliberately configured through DT configuration, this dissertation adopts an automatic memory creation mechanism that adds a PMP entry per non-critical regions. Since both of these PLIC registers (*Threshold* and *Claim/Complete*) are in the same memory strip with a width of $0\text{x}1000$, a single PMP entry can protect them. To create a PMP entry to these context-dependence registers, OpenSBI starts reading the execution context, supporting the base address calculation of the PMP entry. Next, for each allocated hart in *domain instances*, the OpenSBI establishes a PMP entry with a range of $0\text{x}1000$, i.e, with an order value of 12.

After automatically defining the non-critical PLIC regions, the HSP-V enhance the OpenSBI to create *domain memory regions* for MMIO devices specified in *possible-devices* property. With the information collected by the binding process mentioned in Section 3.2, OpenSBI automatically creates a PMP entry with the range specified in the MMIO device node. In other words, when a user assigns an MMIO device to the *possible-devices* property, its associated memory is directly assigned to the *domain instance*. This process enforces inter-domain isolation and prevents OpenSBI misconfigurations. When a user configures a *domain memory region* similar to any device region in *possible-devices* of an adjacent *domain instances*, the OpenSBI will report an error and not launch any domain. Essentially, as the created region is already assigned to another domain, do not need to be reassigned; otherwise, it will violate the overlap policies mentioned in Section 2.3.4.

3.4 Trap-and-emulate Solution

At this point, a user has the requirements to create inter-domain isolated systems in OpenSBI without worrying about misconfigurations. The (i) *possible-devices* property, the (ii) automatic non-critical PLIC memories assignment, and the (iii) automatic MMIO device assignment, compose the RISC-V static partitioning system. Moreover, it provides isolation of each interrupt controller component in multiple domains environments. Since any *domain instance* operates either in S or U-mode, every time the execution tries

to access the PLIC structure, the system will behave in one of two unique ways. It will directly access the interrupt controller space when PMP entry conditions are favorable (when accessing correspondent non-critical PLIC regions), or it will trap and request access to the higher privileged layer (when accessing critical PLIC regions).

As already presented in 2.1.3, the trap-and-emulate mechanism emerges as a virtualization technique capable of solving isolation problems. In this dissertation, this technique will be adopted to isolate critical PLIC regions. When domains try to access the suppressed regions, i.e., the critical PLIC regions, the system will trap through synchronous events, and then, the execution emulates or ignores the accesses attempts by the *domain instances*. To emulate all critical PLIC accesses, this dissertation will enhance the OpenSBI trap handler space, at M-mode, by integrating the decoding and emulation of any access to the critical PLIC registers.

Emulation of Critical PLIC Acesses

Figure 3.4 depicts the trap-and-emulate stages. First, (1) the system executes a load/store instruction access over a critical PLIC region (suppressed by the PMP unit). As a result of a PMP entry violation, (2) a synchronous event occurs (*trap_entry*), and the execution state change from S-Mode to M-mode. Due to the *mtvec* establishment, the execution jumps to the OpenSBI trap handler, which preserves the generic registers, the stack pointer, the *mepc*, and the *mstatus*. Next, (3) the exception is decoded and verified if it treats as a load-access fault (i.e. *mcause* with a value of five) or as a store-access fault (*mcause* with a value of seven). Following, OpenSBI performs the emulation of the critical PLIC accesses. This approach can either emulate the access, if validated emulation conditions (detailed in Section 4.4), or can ignore the access, in case of a malicious domain access attempts. For example, when a malicious domain tries to access a critical PLIC register assigned to an adjacent *domain instance*, the OpenSBI just ignores the access and maintains the system integrity. To conclude this phase, the OpenSBI restore the preserved register in point (2) and, (4) the system switch from M to S-mode using the *mret* and *mstatus* CSRs (*trap_exit*). In the last trap-and-emulate stage (5), the machine resumes the execution to the point that was left.

The trap-and-emulate technique enforces the inter-domain isolation in OpenSBI domain systems by supporting the control of PLIC accesses. Depending on the critical PLIC registers, the OpenSBI emulates the accesses differently. The emulation of write/read operations over *Priority* registers depends on the *mtval*/value address, which specify the address that the domain is attempting to access, and the *possible-devices* property values, which specify the possible source *Priority* registers that the domain is allowed to access. In case of *mtval*/value match with one of the possible source *Priority* registers, OpenSBI performs the emulation, in the opposite scenario, the OpenSBI ignores the access. Regarding the *Pending* and *Enable* registers, the process is similar. The difference is that the *possible-devices* property indicates the possible bits of these registers that the domain is allowed to access. This approach allows the domain to

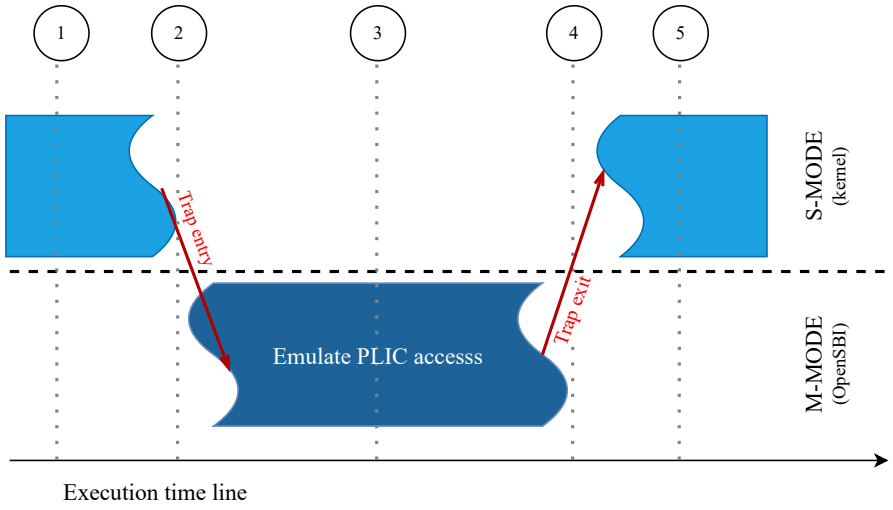


Figure 3.4: Trap-and-emulate technique behavior to emulate critical PLIC accesses.

write/read the masked bits of *possible-devices* in *Enable/ Pending* registers. The information details about this kind of filter are explained in Section 4.4.

3.5 MCSs with enhanced OpenSBI Domain System

This section leverages analysis of the proposed static partitioning system through the enhancement of the OpenSBI. The design in Figure 3.5 configures two domains (e.g. *Domain1* and *Domain2*) using a DT configuration. It starts by deliberately allocating one hart and one *domain memory region* (e.g. *Application1* and *Application2*) for each *domain instance*. Furthermore, OpenSBI automatically assigns *domain memory regions*, i.e., the MMIO device regions of devices defined in the *possible-devices* property and the respective non-critical PLIC regions. With assignment of devices in property possible-devices, OpenSBI assigns the device with interrupt ID 10 to *Domain1* and the device with interrupt ID 35 to *Domain2*.

Regarding inter-domain visibility, the *Domain1* and *Domain2* do not have access to critical PLIC registers and to *domain memory regions* of each other. The *Domain1* [1] have visibility over (i) their configured *domain memory regions* (i.e. the Application1); (ii) over regions of the devices presented in *possible-devices* configuration property (i.e. the memory of the device with ID 10); and (iii) over their assigned non-critical PLIC regions (i.e. the *Claim/Complete* and *Threshold* memories). The isolation is established when the *Domain1* has not access to *Domain2* memories, as well the opposite. In this scenario, the system will trap every time that one of the *domain instances* tries to access adjacent *domain memory regions*, i.e., every time the execution provokes a Load/Store access fault by an attentive to execute an instruction

[1] The visibility case repeats for Domain2. Domain2 has visibility over (i) Application2, (ii) over regions of the devices presented in *possible-devices* configuration property, i.e., the memory of the device with ID 35, and (iii) over their assigned non-critical PLIC regions.

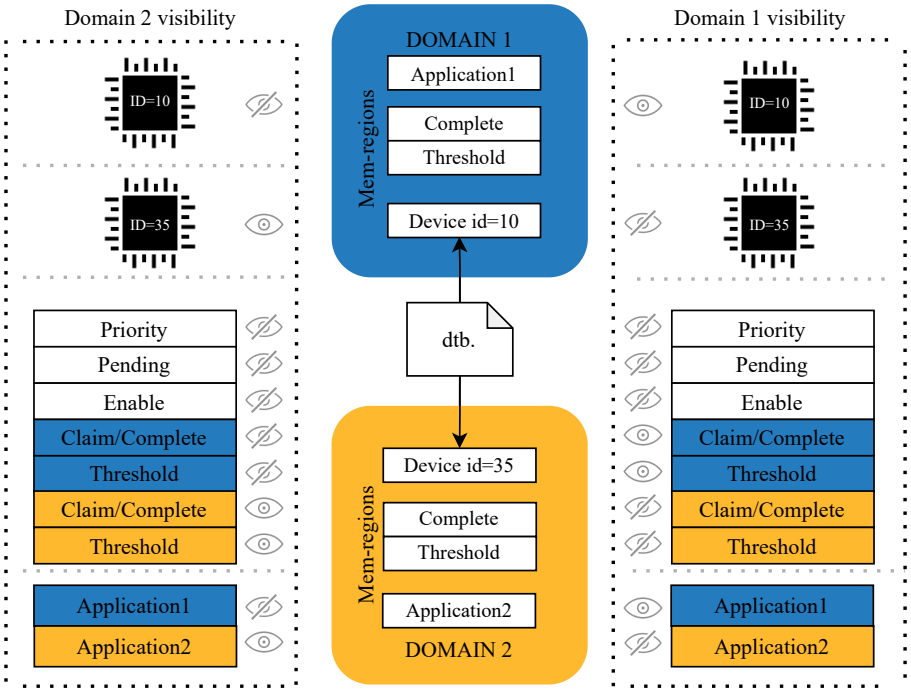


Figure 3.5: Automatic domain configuration by integrating all proposed solutions.

that access a suppressed PMP entry. Furthermore, besides these automated memory assign techniques improve the inter-domain isolation, it also offers an improvement in OpenSBI correct configuration.

At this point, OpenSBI will safeguard the critical PLIC register accesses by just do not assign them to domains. With this approach, when S or U-mode environments try to access any critical PLIC registers, the machine will trigger synchronous events, i.e., the exceptions load/store access faults. Any attempt to execute a load/store operation over the critical PLIC register will force the execution to change the machine state from the S or U-mode to M-mode (where OpenSBI trap handler operates). At this privileged level is performed the trap-and-emulate process to protect each critical PLIC register. As mentioned in Section 3.4, the OpenSBI trap handler will emulate the accesses by performing them or ignoring them. Several CSRs and the information collected in the *possible-devices* property will support the validation of this access attempt. After the emulation process, the execution return to the point where it was before the trap.

According to this analysis, the proposed OpenSBI enhancement is enough to improve the inter-domain isolation. Including a flexible and straightforward configuration through the DT, an automatic MMIO device assignment mechanism, an automatic non-critical PLIC partitioning, and a trap-and-emulate technique to emulate critical PLIC registers.

4. HSP-V: Implementation

In Chapter 3, we have proposed a solution through the OpenSBI enhancement to achieve a RISC-V static partitioning system across different platforms. This chapter aims to describe its implementation: (i) the DT binding enhancement; (ii) automatic device MMIO regions assignment; (iii) automatic non-critical PLIC partitioning; and the trap-and-emulate technique.

4.1 Device Tree Binding

Two approaches establish the OpenSBI configuration (Chapter 2.3.5); however, the previous Chapter mention that not all configuration approaches generate robust and isolated systems. Due to its flexibility, its easy platform adaptation, and its OpenSBI domain system feature, this Chapter starts by exploring the DT configuration. The goal of this implementation stage is to add one more configuration property to the DT domain system configuration by providing to OpenSBI an association mechanism between domains and devices, not precluding the external interrupt controller space of devices.

OpenSBI Domains Parameterization

According to the parameterization analysis outlined in Chapter 2.3.5, there is no configuration property to assign devices to *domain instances*. This Chapter starts by developing a method to link devices with *domain instances* by adding a property to the OpenSBI domain configuration, i.e., the *possible-devices*. The goal is to ensure that the assigned devices are inaccessible to adjacent *domain instances*, enforcing security and isolation to the system. To develop this process, this implementation employ the OpenSBI functionalities following the OpenSBI code structure, as well as its nomenclature.

This implementation start locating the code where the OpenSBI binds *domain instances* properties, in other words, the place where OpenSBI will be enhanced with a new binding process. Due to possible-devices being applied over each *domain instance*, make sense add its binding process during the the DT bind of *domain instances* properties.

Binding of "possible-devices" Property

The pseudocode described in Algorithm 1 analyzes how the additional attribute was bound. Initially, a vector of domains was established with each entry defining an independent *domain instance*, the *implemented_domains*. Each entry of this vector contains a *domain instance* property, which includes the new DT configuration feature, the *possible-devices* property. Next, was established the vector *implemented_devices* which will support this algorithm to prevent multiple device assignation into the same *domain instance*. Last, but not least, was established a vector of *Devices*, which will contain the DT location nodes of devices defined in *possible-devices* property.

Algorithm 1 Interpretation of DT's properties

Require:

$implemented\_domains$: Vector of already implemented domains
$implemented\_devices$: Vector of already implemented devices
$Devices$: Vector of device locations in DT

for $dom \in implemented\_domains$ **do**
  $implemented\_devices \leftarrow get\_implemented\_devices(dom)$
end for
$Devices \leftarrow get\_properties("possible\text{-}devices")$
for $dev\_num \in Devices$ **do**
  while $\neg last\_dev\_id$ **do**
    $dev\_id, dev\_id\_size \leftarrow fdt\_parse\_device\_id(dev\_num)$
    if $device\_is\_implemented(dev\_id, implemented\_devices)$ **then**
      $id\_already\_assigned \leftarrow true$
    end if
    if $\neg(id\_already\_assigned)$ **then**
      $set\_device\_to\_domain(dev\_id, dev\_mask)$
    else
      $report\ \ error$
    end if
    if $dev\_id = dev\_id\_size$ **then**
      $last\_dev\_id \leftarrow true$
    end if
  end while
end for

To start, the Algorithm 1 collects already *implemented_devices* defined in adjacent *domain instances*

through *get_implemented_devices* function. Essentially, the execution goes through all already implemented *domain instances* and collect the *possible-devices* property of each of them. Posteriorly, the function returns all device IDs to *implemented_devices* vector, where each bit represents an interrupt ID or IDs of each already implemented device.

Next, to bind the *possible-devices* property of the *domain instance* under analysis, this implementation use the function *get_properties*. This function binds the new property, i.e., the *possible-devices*, by applying a parser in DT, and returns DT location nodes of each defined device to the variable *Devices*. In other words, each *Devices* position contains a location of a device in the DT.

After collecting the device nodes' locations, OpenSBI goes through each *Devices* node through the variable *dev_num* and collects their IDs. As HSP-V enhances pretend to protect the PLIC regions associated with external device interrupts, it is necessary to collect each interrupt ID triggered by each *dev_num* device, and associate it exclusively with a *domain instance*. The property *interrupts* inside each device node contains its device interrupt ID or IDs. To collect either each Device ID and the number of IDs in a device node, this algorithm executes the function *fdt_parse_device_id*, returning the variables *dev_id* and *dev_id_size*, respectively. While not last ID in device *dev_num*, this algorithm connects each of them to the *domain instances* if not assigned in *implemented_devices*. Otherwise, to guarantee inter-domain isolation and avoid misconfigurations, the devices in *possible-devices* must pass through a mechanism that avoids multiple *domain instance* assignation. The goal is to ensure that each device is assigned to one exclusive domain. If the *dev_num* is already defined in other *domain instances*, an error will be reported by OpenSBI, informing that the user is configuring the same device in different domains. In a valid scenario, where the *dev_id* is not between the possible-devices of adjacent *domain instances*, each device is linked to the *domain instance* under analysis through the function *set_device_to_domain*, where each bit of a variable *dev_mask* will correspond to the interrupt ID or IDs of devices in *possible-devices*. For example, if the *dev_mask* variable specifies the value two, it means that the domain has permission to access the interrupt with ID one (bit zero corresponds to interrupt ID zero). The *dev_mask* of each *domain instance* will support either in the access validation by the trap-and-emulate technique or in the automatic device MMIO regions assignment.

Implemented IDs From Adjacent Domain Instances

As already mentioned, to ensure that devices already linked and not repeatedly assigned in different *domain instances*, the function *get_implemented_devices* runs through all the domains to collect each of *implemented_devices*, as shown in Algorithm 2. For each domain, the value of the variable *possible-devices* is analyzed and collected bit by bit, filling the variable *implemented_devices.* At the end of this process, the variable *implemented_devices* will contain all device IDs previously linked to other domains.

After implementing the illustrated functions, it is possible to extend the DT configuration with the *possible-devices* property to assign device associated interrupts to *domain instances*. As already mentioned in the previous Section 3.2, Figure 3.2 depicts a configuration example, in which *uart* and *i2c*

Algorithm 2 get_implemented_devices function

for $dom \in implemented\_domains$ **do**
  for $bit \in max\_device\_bits$ **do**
    $bit\_value \leftarrow test\_bit(bit, dom.possible\_devices)$
    $implemented\_devices \leftarrow (bit\_value \ll bit)$
  end for
end for

device nodes are connected to the domain instance *tdomain*. In this scenario, *tdomain* has access to the PLIC information about interrupts of *uart* and *i2c* devices, i.e., interrupt IDs $0xa$ and $0xb$ respectively. During *get_implemented_devices* function execution, during boot time, the *implemented_devices* will contain the value $0xC00$ (with bits 11 and 12 active).

Device ID Parsing

The following subsections illustrate in detail each function referenced in the previous Algorithm 1. The function *fdt_parse_device_id* presented in Algorithm 3 returns the device interrupt ID or IDs, as well the number of IDs in the device *dev_num*. To begin, it determines if the evaluated device contains the interrupt controller PLIC as an *interrupt-parent* property. To check this condition, the value of this property is

Algorithm 3 fdt_parse_device_id function

Require:
$str1$: String to compare with value "sifive,plic1.0.0"
$str2$: String to compare with value "sifive,fu540c000plic"
$str3$: String to compare with value "sifive,plic0"
$implemented\_domains$: Vector of domains already implemented
$device\_offset$: Offset to localize node in DT
id_offset : Offset value of multiple IDs into a device

$interrupt\_parent\_phandle \leftarrow get\_phandle(device\_offset, "interrupt\text{-}parent")$
$interrupt\_parent\_offset \leftarrow get\_offset\_by\_phandle(interrupt\_parent\_phandle)$
$property\_value \leftarrow get\_property(interrupt\_parent\_offset, "compatible")$
if $property\_value = str1 \vee property\_value = str2 \vee property\_value = str3$ **then**
  $val, number\_of\_ids \leftarrow get\_property(device\_offset, "interrupts")$
  $dev\_id \leftarrow val[id\_offset]$
  $return\ \ val, number\_of\_ids$
else
  $return\ \ error$
end if

collected, and then followed by a check to verify if it contains one of the available PLIC compatible property (sifive,plic1.0.0", "sifive,fu540c000plic" and "riscv,plic0"). In a negative case, an error is reported by the OpenSBI informing that the interrupt parent is not the PLIC and consequently, the device is not assigned to the *domain instance*. In the positive case, it returns the associated interrupt ID in the device node and the number of IDs.

4.2 Automatic Integration of Device MMIO Regions

With *possible-devices* configuration property, each domain can attach multiple external interrupt device IDs to it, but it must also include the device MMIO regions associated with these IDs. Otherwise, the execution tends to trap each time the domain tries to access the MMIO region of these devices. A mechanism for linking these regions to *domain instances* was developed, streamlining the partitioning process and decreasing the possibility of user configuration errors. The purpose is to automatically add *domain memory regions* associated with devices in *possible-devices* property. The implementation is done by reusing a few lines of code from the OpenSBI source code at *domain instance* creation time.

After binding the *possible-devices* property from DT, OpenSBI executes the function *create_device_memregion* outlined in Algorithm 4, which creates *domain memory regions* associated with the values of this property. This procedure starts by iterating and sorting all device offsets of *Devices*, a vector previously collected in Algorithm 1. Next, to avoid the creation of multiple identical MMIO regions, the OpenSBI proceeds to a comparison of each vector entry to the succeeding vector position, skipping the equal *Devices* values. The OpenSBI determines the MMIO base address region, the size, and the necessary flags for the PMP entry creation. Finally, to sanitize *domain instances* with these new regions,

Algorithm 4 create_device_memregion function

Require:

$Devices$: Vector of device locations in DT

$Devices \leftarrow sort\_vector(Devices)$
for $dev \in Devices$ **do**
  if $Devices[dev] \neq Devices[dev+1]$ **then**
    $dev\_base\_addr, size \leftarrow parse\_addr\_and\_size\_property(dev)$
    $order \leftarrow log\_base\_two\_of(size)$
    $flags \leftarrow (Read\_access\ \ and\ \ Write\_access)$
    $add\_region\_to\_domain\_regions(dev\_base\_addr, order, flags)$
    $increment\_count\_regions$
  end if
end for

OpenSBI adds the MMIO region to the *domain memory regions* and increases the *count_regions* variable. When OpenSBI collects all *domain instances* with all *domain memory regions*, it sanitizes each of them, checking if there is anything wrong or misconfigured. It checks if any of the *domain memory regions* has a null range; if any of them has an invalid base address/order/flags; and even the presence of the firmware region.

4.3 Automatic Integration of PLIC Regions Over Partitions

To partition PLIC and rectify the problem in Chapter 3.1, this implementation separate critical and non-critical PLIC regions. Due to the reasons mentioned in chapter 3.3.1, the non-critical PLIC regions are the *Claim/ Complete* and *Threshold* registers; and the critical PLIC regions are the *Priority*, *Pending*, and *Enable* registers. This section demonstrates how we enhance OpenSBI to automatically isolate the non-critical PLIC regions between different domains.

To protect them, this implementation adopt the direct memory assignment of the non-critical PLIC regions to domains by establishing a PMP entry for each region. To follow the PMP entry creation requirements, i.e., the base address, size, and flags definition, was implemented the *get_non_critical_plic_regions* function, represented by Algorithm 5. This function starts to collect the PLIC base address from the DT by parsing and searching its *compatible* property. In case of not finding any *compatible* property with

Algorithm 5 Get non-critical PLIC regions

Require:

$str1$: String to identify PLIC node in DT with value "sifive,plic1.0.0"
$str2$: String to identify PLIC node in DT with value "sifive,fu540c000plic"
$str3$: String to identify PLIC node in DT with value "sifive,plic0"

$plic \leftarrow found\_PLIC\_node\_in\_DT(str1)$
if $\neg plic.addr$ **then**
  $plic \leftarrow found\_PLIC\_node\_in\_DT(str2)$
  if $\neg plic.addr$ **then**
    $plic \leftarrow found\_PLIC\_node\_in\_DT(str3)$
  else
    $report\ \ error$
  end if
end if
$reg.base \leftarrow calculate\_noncriticalregion\_base\_addr(plic.addr, cntxt)$
$reg.order \leftarrow define\_memory\_size$
$reg.flags \leftarrow define\_memory\_acesses$

provided strings, the system returns and reports an error. Otherwise, (i) it calculates the PMP entry base address through $base\_addr = plic\_base\_addr + 0x200000 + (context \times 0x1000)$; (ii) it set the write and read flags; and (iii) it defines the size of the region with an order value of 12.

After creating these regions, OpenSBI combines them with other *domain memory regions*. As a result of entrusting only non-critical PLIC regions to *domain instances*, the domains make the system trap when accessing the critical PLIC regions. Moreover, since the user does not have to worry about creating any *domain memory regions* to isolate non-critical PLIC regions, this automatic approach assists the OpenSBI configuration by preventing possible user misconfiguration.

4.4 Trap-and-emulate Mechanism Implementation

Now that the non-critical PLIC regions are protected, this section will focus on the critical PLIC regions by implementing the trap-and-emulate technique. The trap-and-emulate approach is executed at the M-Mode level during the OpenSBI trap handler and helps to preventing domains to control external interrupts that are not assigned to them. The OpenSBI trap handler's flow is depicted in Figure 4.1, where it starts by collecting and saving the contents of several registers, including the Stack Pointer (SP), PC, and some CSRs. Then, OpenSBI executes a trap routine named *TRAP_CALL_C_ROUTINE*, which covers various critical services such as, run-time services, SBI ECalls handling, virtual-to-physical memory translations, and the trap-and-emulate mechanism.

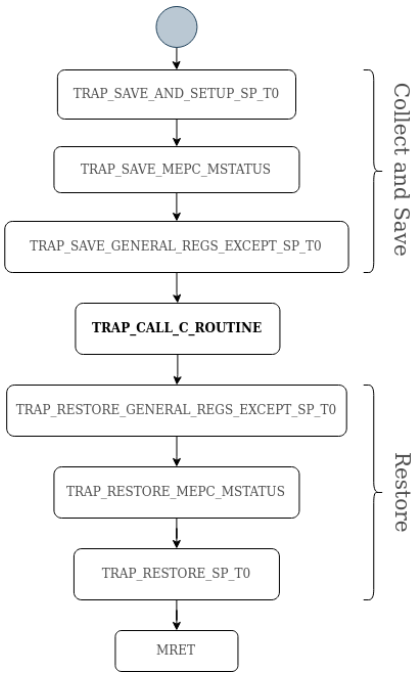


Figure 4.1: Flowchart of the OpenSBI trap handler flow.

When the system executes the *TRAP_CALL_C_ROUTINE*, OpenSBI executes the trap handler process to handle either interrupts or exceptions, respectively described in Algorithms 6 and 7. Firstly, the OpenSBI

executes Algorithm 6, which starts to gather the CSRs *mcause* and *mtval* values. The CSR *mcause* to inform the system about the trap cause, and the CSR *mtval* to get the memory address that the domain is trying to access. Once the acquisition of preceding data, OpenSBI determines whether the cause of the trap arises from an exception or an interrupt. If it comes from an interrupt, i.e., the most significant bit of the CSR *mcause* is set, OpenSBI performs an Instruction Service Routine (ISR). When (i) a software interrupt occurs, the system executes the function *sbi_ipi_process*; when (ii) a timer interrupt occurs, the system executes the function *sbi_timer_process*; and when (iii) an external interrupt occurs, the OpenSBI trap handler reports the message *"Unhandled external interrupt"*. However, since in this dissertation all interrupts are delegated to the lower privileged levels by *mideleg* CSR, external interrupts will never reach the M-mode privilege level, benefiting interrupt latency mitigation. In a different case, that interrupts are delegated to a higher privileged level (in OpenSBI trap handler) the context-switch would insert unwanted latencies into the system each time an interrupt occurs.

Algorithm 6 OpenSBI trap handler - Interrupts

```
Get_value_of_mcause_and_mtval
if is_interrupt then
    if mcause = irq_m_timer then
        executes_timer_handler
    end if
    if mcause = irq_m_soft then
        executes_software_handler
    end if
    if mcause = irq_m_external then
        report message "unhandled external interrupt"
    end if
    exits_trap_and_restore_execution
end if
```

On the other hand, in the exception scenario trap cause, i.e., the most significant bit of the CSR *mcause* is clear, OpenSBI executes different handlers for different exception causes, described in Algorithm 7. It presents a pre-built handler implementation to face causes such as illegal instruction, misaligned loads/stores, supervisor/machine ECalls (handler to treat the run-time services mentioned in Chapter 2.3.2), and loads/stores access faults. The trap-and-emulate technique will be implemented in this last case, i.e., the load/store access faults, emulating the access to the address value in *mtval*. However, because trap-and-emulate only operates with PAs, this algorithm must verify if the MMU is enabled prior to performing emulation, i.e., check if the address-translation scheme (mode of *sapt* CSR) is different from

"Bare" (mode zero of *mtval* CSR) [18]. The *mtval* CSR needs to be translated in case of the *sapt* mode is not zero. Otherwise, the MMU is disabled and the S-mode layer only operate over PAs.

Algorithm 7 OpenSBI trap handler – Exceptions

```
if is_exception then
    if mcause = illegal_instruction then
        executes_illegal_insn_handler
    end if
    if mcause = misaligned_load then
        executes_illegal_load_handler
    end if
    if mcause = misaligned_store then
        executes_illegal_store_handler
    end if
    if mcause = supervisor_ecall ∨ mcause = machine_ecall then
        executes_ecall_handler
    end if
    if mcause = load_access_fault ∨ mcause = store_access_fault then
        if is_MMU_activated then
            translate_va_to_pa(mtval, mepc);
        end if
        trap_and_emulate algorithm
    end if
    exits_trap_and_restore_execution
end if
```

The translation is performed as a software page-walk following the algorithm described in the RISC-V privileged specification [18], and executed in the function *translate_va_to_pa*. It consists of a set of steps using CSRs *satp* and the *mstatus*. As mentioned in Section 2.2.4, *satp* CSR's role is to control the S-Mode memory translation and protection mechanism. In case of *mstatus*, this CSR contains useful control bits to perform translation, such as MXR and SUM bits [18], that allows loads from pages marked executable and permit supervisor user memory accesses, respectively. At the end of this function, the *mtval* and *mepc* values will contain the PA of the correspondent VAs, making the trap-and-emulate operates only with PAs.

Trap-and-emulate process

The body of the trap-and-emulate, in Algorithm 8, starts by verifying whether *mtval* belongs to PLIC space or not. In a negative case, the access is ignored, and the emulation is ignored. Otherwise, if the access target address is in PLIC space, the trap-and-emulate starts to collect relevant data for further verifications to support the emulation: (i) the *interrupt_id* by function *translate_interrupt* and (ii) the *opcode_leaf* by function *get_opcode_leaf*. The *opcode_leaf* gets part of the trap instruction containing the

access data, e.g., the value of a *Priority* register in a write access; and the *interrupt_id* gets the external interrupt that the *domain instance* tries to access, e.g., an interrupt ID value triggered by a device.

Algorithm 8 Trap-and-emulate Algorithm

Require:

$prev\_mode$: Privileged mode prior the trap

if $mtval \in plic\_space$ **then**
  $interrupt\_id \leftarrow translate\_interrupt(trap.tval, opcode\_leaf)$
  $opcode\_leaf \leftarrow get\_opcode\_leaf$
  $mhartid \leftarrow get\_value\_of\_mhartid$
  $prev\_mode \leftarrow get\_prev\_mode(mstatus)$
  $cntxt \leftarrow calculate\_execution\_context(prev\_mode, mhartid)$
  $cntxt\_valid \leftarrow is\_cntxt\_valid(trap.tval, cntxt)$
  $interrupt\_assign \leftarrow is\_interrupt\_assigned\_to\_domain(hartid, interrupt\_id)$
  if $cntxt\_valid \wedge interrupt\_assign$ **then**
    $acc.add \leftarrow trap.tval$
    $acc.reg \leftarrow opcode\_leaf$
    $acc.write \leftarrow (mcause \wedge 2)?true : false$
    $acc.cntxt \leftarrow cntxt$
    $plic\_emul\_hanlder(acc)$
    if $!acc.write$ **then**
      $update\_rd(regs, acc.reg)$
    end if
  end if
  $calculate\_next\_addr(regs)$
end if
$exits\_trap\_and\_restore\_execution$

The *opcode_leaf* variable has two purposes: (i) supporting Read-Modify-Write operations of *Pending* and *Enable* register emulations; and (ii) supporting all critical PLIC registers' store operations. Typically, Read-Modify-Write operations are used in bitwise operations, where each bit controls different system functionalities, just like *Pending/Enable* registers and their bitwise representations (mention in Chapter 3.3.1). Figure 4.2 illustrates the execution of two different instructions, executed at different times. The figure depicts (i) an instruction to commonly store the value 0x7 in a PLIC *Priority* register $0xc000010$, which corresponds to an interrupt ID four ($interrupt\_id = mtval/4$); and a load operation resulted from a (ii) Read-Modify-Write operation, reading only the desired bits of an *Enable* register $0xc002400$. Due to the PLIC structure, the value $0x10$ masks the interrupt four in this *Enable* register.

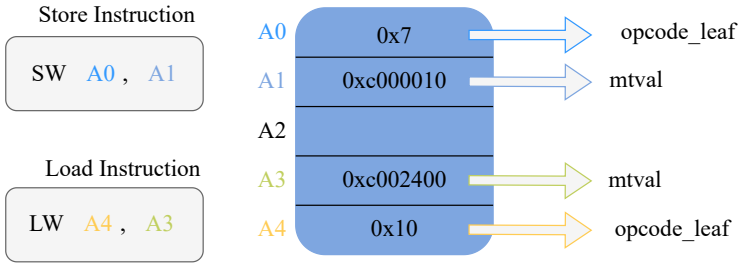


Figure 4.2: *Opcode_leaf* in store and load operations.

After *opcode_leaf* and *interrupt_id* collection, as there are context-dependent PLIC registers, i.e., the *Enable* registers, Algorithm 8 also perform a context validation over critical PLIC accesses to provide only valid emulations. To get the context at the trap time, the trap-and-emulate algorithm implement the *calculate_execution_context* function, which goes through expression $cntxt = prevmode + mhartid \times 2$. Next, with the result of this expression, the dependent PLIC registers are evaluated with the *is_cntxt_valid* function, returning a value that indicates if the execution should ignore the emulation or process it. Beyond context validation, the *is_interrupt_assigned_to_domain* function evaluates if the calculated *interrupt_id* is assigned to the domain or not. In other words, this validation process validates if the device which triggers this *interrupt_id* is assigned between the *possible-devices* property of the *domain instance* or not. In a positive case, the emulation is performed. Otherwise, the OpenSBI trap handler discards the access to the critical PLIC region.

Once validated all verifications, the OpenSBI emulates the execution of the access trap instruction (load or store access faults). The emulation in Algorithm 8 starts to agglomerate all data related to the access by creating a structure, the *acc*. The structure elements are (i) the *tval* as the PA of the access, the (ii) *reg* as a leaf of the opcode, the (iii) *write* as an indicator that specifies if it treats load or store access, and the (iv) *cntxt* as the value of the execution context. To emulate critical PLIC register accesses, i.e., *Priority*, *Pending*, or *Enable* registers, the OpenSBI executes the *plic_emul_handler* function, described in Algorithm 9.

Algorithm 9 PLIC access emulation handler

$plic\_region \leftarrow Get\_PLIC\_addr\_target$ ▷ $(acc.addr \gg 12) \wedge 0x3$
if $\neg plic\_region$ **then**
  $plic\_emul\_prio\_access(acc)$ ▷ $plic\_region = 0 = Priority$
else
  if $plic\_region = 1$ **then**
    $plic\_emul\_pend\_access(acc)$ ▷ $plic\_region = 1 = Pending$
  else
    $plic\_emul\_enable\_access(acc)$ ▷ $plic\_region = 2 = Enable$
  end if
end if

This algorithm starts by identifying the target register of the PLIC. Since each critical PLIC registers are separated from each other with an offset of $0x1000$, the algorithm only needs to divide *acc.addr* by 2^{12}, followed by a logic *And* operation to filter the two less significant bits. As a result, these two last significant bits identify the target PLIC register: if *plic_region* is zero, the handler emulates a *Priority* register access through function *plic_emul_prio_access*; if *plic_region* is one, the handler emulates a *Pending* register access through function *plic_emul_pend_access*; and if *plic_region* is two, the handler emulates a *Enable* register access through function *plic_emul_enable_access*. To emulate *Priority* accesses, this implementation first determines whether the access is a store or load operation via *acc.write*; if the access is a store, it writes the value of *acc.reg* to address *acc.addr* $((acc.id \times 4) + plic\_base)$; if the access is a read, it reads the value in *acc.addr* to the *acc.reg*. To emulate *Pending* accesses, if it is a load operation, the emulation process reads the value in *acc.addr* to the *acc.reg*, only with the content of the assigned devices by masking ID bits of each *possible-devices* property, collected by the Algorithm 1, as depicted in Figure 4.3; on other hand, if it is a store operation, since *Pending* registers are read-only registers, the emulation process ignores the access. With this approach in Figure 4.3, the emulation filter all devices which should not be visible to the adjacent *domain instances*, letting the domain be aware only device IDs in *possible-devices* property. Regarding the emulation of *Enable* accesses, is applied a similar methodology as *Pending* accesses, filtering bits which domain cannot access, as depicted in Figure 4.3; however, since *Enable* registers are not read-only registers, besides the load operation, the OpenSBI also applies operations. It stores the value *acc.reg* into *acc.addr* (only storing the allowed bits in *possible-devices*).

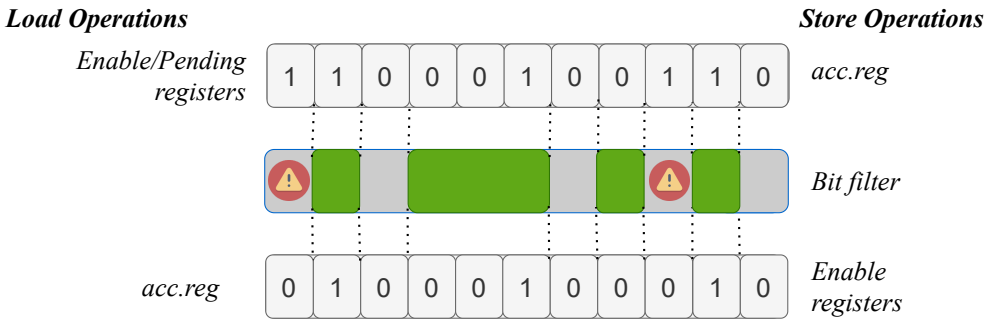


Figure 4.3: Emulation example after a load/store operation.

The trap-and-emulate method concludes by recalculating the return address of the next PC position when returning to the S or U-mode. This method is implemented via the *calculate_next_addr* function and concerns the RISC-V compressed instructions. A compressed instruction is often a 16-bit instruction that encodes regular integer operations. It is possible to introduce a compressed instruction when adding the C extension over ISA. The adoption of this improvement reduces RISC-V instructions by fifty percent and code size by twenty-five percent [71]. After recalculating the next PC address, OpenSBI recovers the execution context and proceeds with or without the critical PLIC registers altered.

5. HSP-V: Evaluation and Results

This section presents the evaluation of developed RISC-V static partitioning system, the HSP-V. Initially, it details its (i) functional validation to prove the correctness of the binding process, the automatic device MMIO regions assignment, and PLIC partitioning protection. Then it evaluates (ii) the Trusted Computing Base (TCB) using code size and memory footprint, by comparing the original OpenSBI (without changes) with the enhanced OpenSBI for HSP-V. Furthermore, this section also leverages (iii) the boot overhead by measuring the time with and without the HSP-V system (forward and previous changes); (iv) the performance, leveraging cache locking technique in a system with and without interference; and (v) the interrupt latency, evaluating the impact of HSP-V over interrupts.

Evaluation Setup

Microchip PolarFire SoC Icicle Kit [1] conducts all experiments running three of its five harts at 625 MHz, one e51 monitor hart, and two u54 application harts, with per-core 32 KiB L1 data and instruction caches, and a shared unified 2 MiB L2 (LLC) cache. For unknown reasons, the FSBL entity, i.e., the Hart Software Services (HSS), does not provide the remaining two application harts. This limitation has to be considered, especially in performance evaluation when the system only has one hart to inject interference instead of three. To manage FSBL and the system monitoring of this board, the HSS provides a set of services running in the e51 monitor hart: (i) masking and locking the L2 cache ways (cache locking); (ii) establishing the first PMP entries; (iii) copying the OpenSBI with its payload(s) from eMMC to the LIM or DDR; (iv) creating a payload for boot OpenSBI.

Besides the limited number of harts, the HSP-V evaluation also faces a limited number of cache ways. As previously mentioned, this board provides 2 MiB of LLC composed of 16 ways (each way with 128 MiB). However, due to the FSBL does not provide the total number of ways to OpenSBI, the HSP-V only leads with 12 ways (1.512 MiB) of LLC. In this board, the L2 configuration follows three cache typologies [2]: L2-LIM, Scratchpad, and L2-Cache. The L2-LIM space is filled with the content of disabled ways, setting aside space in the main memory to replicate every cacheable address. Moreover, with this approach, the L2-LIM provides determinist behavior equivalent to a cache hit, with no probability of a cache miss. On the other hand, when ways are enabled, the HSS manages a cache locking mechanism defining what is

[1] MPFS-ICICLE-KIT-ES: https://www.microsemi.com/existing-parts/parts/152514
[2] PolarFire® SoC MSS Technical Reference Manual: shorturl.at/drzBC

Scratchpad or L2-Cache space. When a way is enabled and locked, it works as L2-Cache. When a way is enabled but not locked, it is addressable in Scratchpad space, i.e., a space of main memory, as L2-LIM, reserved to be always in cache. The difference between Scratchpad and L2-LIM is that the Scratchpad is also cacheable to L1 data cache, resulting in faster access. In conclusion, in this dissertation, four ways (512 MiB) are reserved for Scratchpad and L2-LIM, and the remaining 12 ways (1.512 MiB) to manage the way locking method.

Evaluation tools

The evaluation tools used are as follows: (i) Device Tree Compiler (DTC) [3], to compile the DT scripts, which have a crucial impact in system configuration; (ii) GCC [4] (with first level of compile optimization), to compile all HSP-V files; (iii) Buildroot [5], to build and compile Linux images; (iv) SoftConsole [6], to compile the HSS [7], configure primary PMP entries and define the cache locking mechanism; (v) QEMU [8], to simulate hardware platforms; (vi) Gnu DeBugger (GDB) [9], to enable debugging capabilities; (vii) Open On-Chip Debugger (OpenOCD) [10], to extend GDB functionalities by enabling remote debug access through OpenOCD server support; and (viii) PUTTY [11], to interact with platform used in this dissertation and display serial interfaces. Table 5.1 depicts each tool version.

Table 5.1: Description of the tool versions.

Tool	Version
DTC	1.5.0
GCC	8.3.0
Buildroot	2021.08.01
SoftConsole	2021.1
QEMU	6.0.90
GDB	8.3.0
OpenOCD	0.10.0
Putty	0.73

[3] DTC: https://elinux.org/Device_Tree_Reference
[4] GCC: https://gcc.gnu.org/
[5] Buildroot: https://buildroot.org/
[6] SoftConsole: https://www.microsemi.com/document-portal/doc_download/130766-softconsole-faq
[7] HSS: https://github.com/polarfire-soc/hart-software-services
[8] QEMU: https://www.qemu.org/docs/master/
[9] GDB: https://lists.gnu.org/archive/html/info-gnu/2019-05/msg00007.html
[10] OpenOCD: https://openocd.org
[11] Putty: https://www.putty.org/

5.1 Functional Validation

This dissertation was implemented iteratively and tested through a debug process by finding and solving implementation bugs. Therefore, the validation tests were performed over a simulated platform in QEMU and then passed to the real hardware platform. To prove its effectiveness, this section presents the functional validation of the system. It composes a set of intermediate tests: (i) tests to validate *domain instance* configuration; (ii) tests to validate the automatic device memory allocation to *domain instances*; and (iii) tests to validate the trap-and-emulate technique.

5.1.1 Functional Validation: Domain Instance Configuration

As mentioned in previous chapters, this dissertation adds to OpenSBI DT configuration a new property to bind, the *possible-devices*. This section intends to prove the correct binding process through the Figure 5.1. This figure depicts a simple domain system configuration in the DT at left side, which assign the *uart* device to the *domain instance tdomain*, and presents a debugging analysis during the execution of the binding function at right side, i.e., the inspection of *fdt_parse_device_id* behavior. Since this function is responsible to bind each ID over devices in *possible-devices* property, if it collects an ID number in accordance to the DT configuration, it proves the correct behavior of the binding function. As a result of this debugging process, the OpenSBI identifies the interrupt ID in accordance to the interrupt ID of the *uart* device (*dev_id = 10*), and simultaneously assign it to the *domain instance* (*tdomain*). In this case,

Figure 5.1: Result of Functional Validation: Domain Instance Configuration.

the device ID belongs to a serial device and has the value $0x\alpha$ as the interrupt ID value, validating the *possible-devices* property binding.

5.1.2 Functional Validation: Automatic Memory Assignment

Another provided feature to OpenSBI is the automatic memory assignment mechanism. According to implementation and design phase, the memory regions to automatic assign are (i) the non-critical PLIC regions and (ii) the device MMIO regions in property *possible-devices*. The following Figure 5.1, depicts a DT configuration on left side, and an OpenSBI configuration result in right side. This configuration provides a partitioned environment composed by two *domain instances*, each running a from the scratch application operating in S-mode.

On the configuration side, the *domain instance* nodes identify their *domain memory regions*. The *bare1-domain* identifies one *domain memory region* at $0x80200000$, with the size of 2^{20} bytes, and the *bare2-domain* identifies one *domain memory region* at $0x80100000$, with the size of 2^{20} bytes. Regarding *possible-devices* values of each *domain instance*, both domains have only one dedicated and assigned device, i.e, *serial1* and *serial2* devices. To validate the automatic memory assignment mechanism, the OpenSBI logger on the right side identifies not one but three *domain memory regions* to each *domain instance*. The red color identifies the *bare1-domain* memories and the green color identifies the *bare2-domain* domains. This provided mechanism automatically creates and assigns (i) the associated

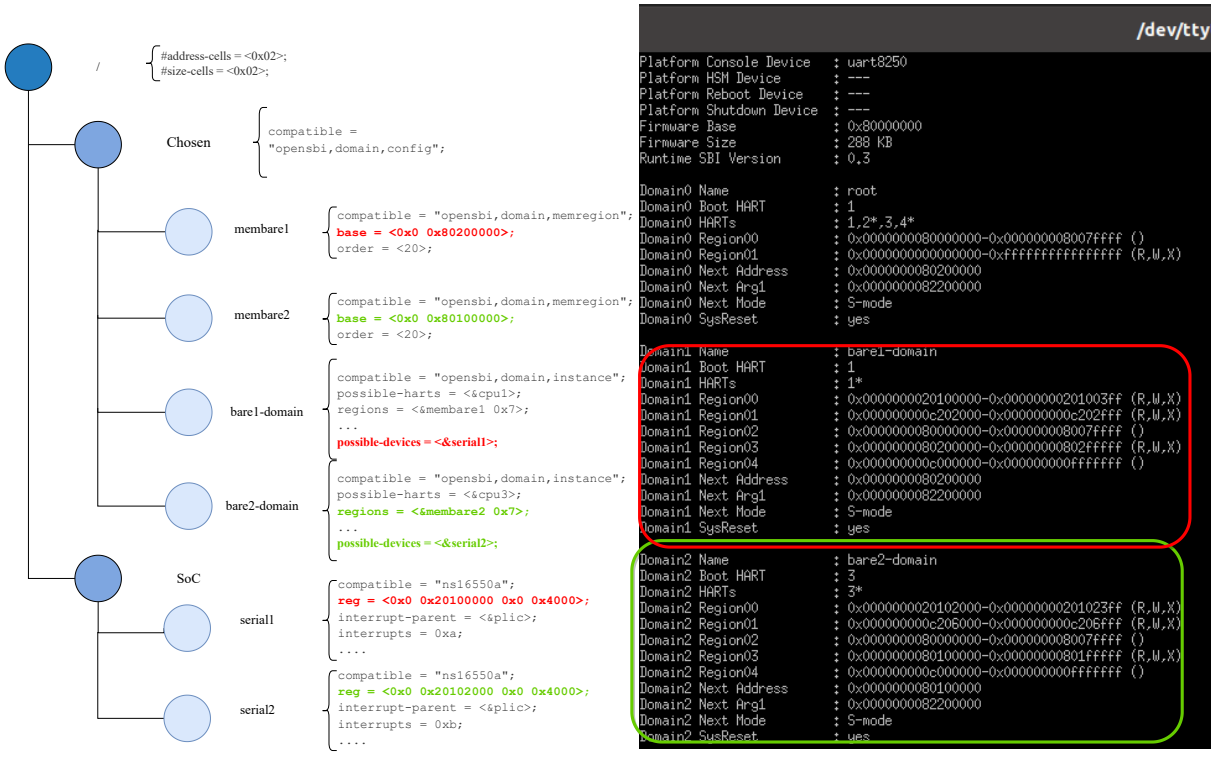


Figure 5.2: Result of Functional Validation: Automatic Memory Assignment.

non-critical PLIC memories according to their contexts, (ii) the serial device MMIO regions defined in *possible-devices*, and (iii) the application memory at DT configuration. This screenshot proves the proper automatic memory assignment according to the DT configuration.

5.1.3 Functional Validation: Emulation of Critical PLIC Accesses

In the example scenario of Figure 5.3, the OpenSBI configures two *domain instances*: (i) the *bare1-domain*, a bare-metal application executed by CPU1, containing a serial device with interrupt ID 91; and (ii) the *bare2-domain*, a bare-metal application executed by CPU3, containing a serial device with interrupt ID 92. Each *domain instance* is meant to execute a periodic loop that waits for an external interrupt. When

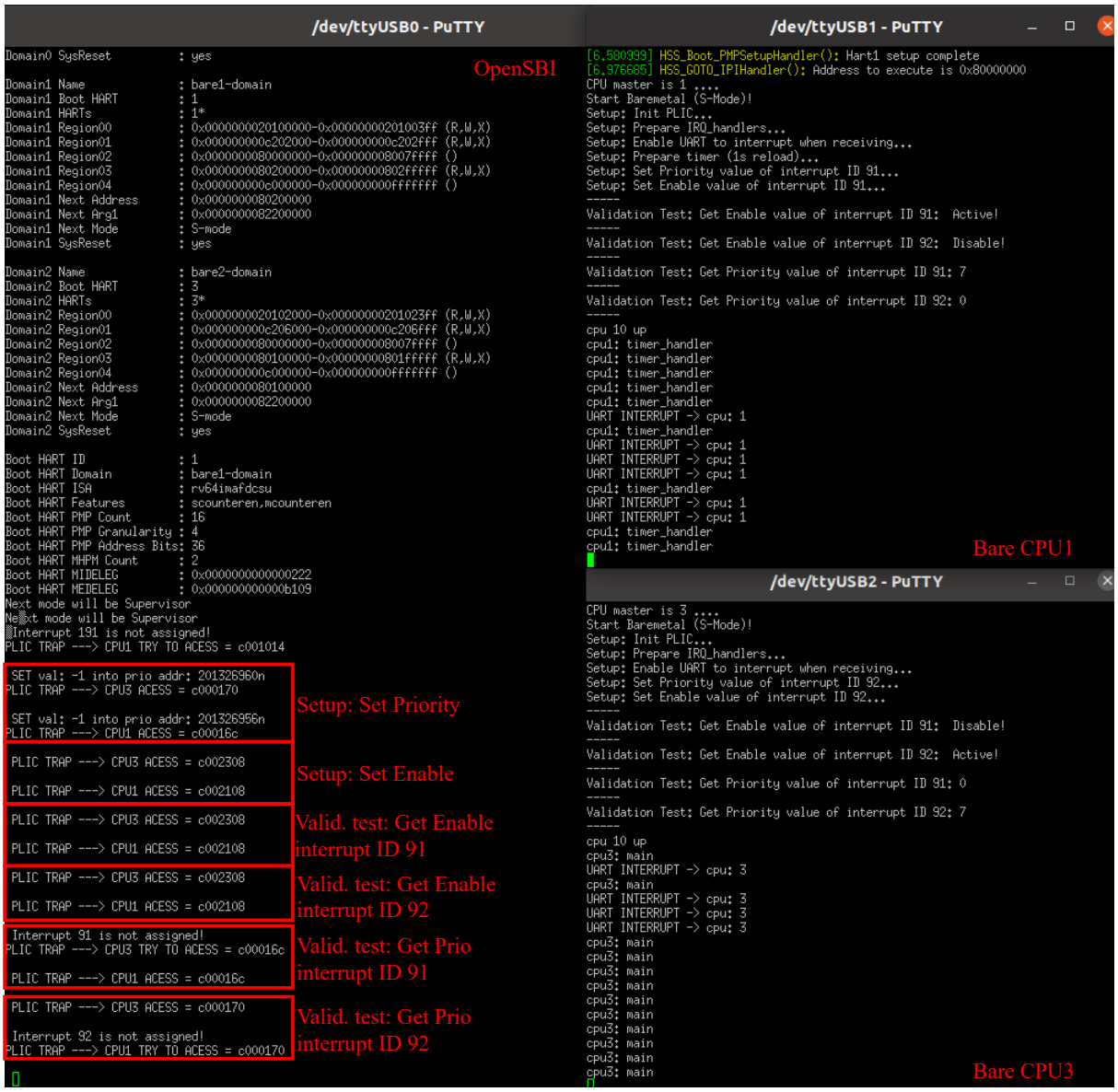


Figure 5.3: Result of Functional Validation: Emulation of Critical PLIC Accesses.

injecting characters to the respective serial device, the respective *domain instance* interrupts the infinite loop and executes the interrupt handler. Both *domain instances* start the execution with a set of functionalities to initialize the system: (i) initializing the PLIC, by defining the *Threshold* register value; (ii) preparing the IRQ handlers (the UART and timer handlers); (iii) configuring the UART device; (iv) configuring the frequency of timer; and (v) establishing the *Priority* and *Enable* registers of the associated serial device. Due to the latter covers critical PLIC regions, OpenSBI performs trap-and-emulate technique when *domain instances* execute them.

Following the initialization, the execution runs several validation tests over the trap-and-emulate technique. Both *domain instances* execute read accesses over *Enable* and *Priority* registers related to interrupt IDs 91 and 92. Looking at the OpenSBI logger, the trap-and-emulate mechanism has completed the access of interrupt 91 when executed by *bare1-domain*, and completed the access of interrupt 92 when executed by *bare2-domain*.

Besides this system example, this dissertation proves the PLIC partitioning over several combinations of MCSs: (i) a bare-metal application alongside an RTOS system, both running in S-mode; (ii) a bare-metal application alongside a GPOS (Linux 5.9), both running in S-mode; and (iii) a RTOS alongside a GPOS, both running in S-mode. Due to the limited number of harts provided by HSS to OpenSBI (two harts), the partitioning system is limited to a number of two *domain instances*.

5.2 Code Size

This dissertation extended the open-source code of OpenSBI, following its code structure and enhancing it with additional features. This section analyzes the contribution of this dissertation by evaluating the code complexity using Source Lines of Code (SLoC).

The OpenSBI code is divided into four main sections: the *utils* and the *platform* directories contain target-specific functionality (e.g. DT parser functionalities, drivers for platforms), while the *sbi* and *firmware* directories feature the main secure monitor logic and utilities (e.g., perform run-time services, managing *domain instances*, perform trap-and-emulate technique). Table 5.2 presents the SLoC and final binary sizes for each directory of the original and the HSP-V OpenSBI.

Table 5.2 shows that, for the target platform, i.e., the generic platform, the original implementation comprises a total of 16.265 KSLoC. This small code base reflects the overall low degree of complexity of the system. Most of the code is written in C, although functionalities such as low-level initialization and context save/restore (exception entry and exit) are written in assembly language. The *sbi* folder contains the most of the total SLoC provided by the original OpenSBI (The majority of additional code is SLoC).

Regarding all changes of HSP-V, the OpenSBI increases about 10% of its source code (1.656 KSLoC). The majority of additional code is in the *utils* folder, i.e., in the parsing management over DT and in the automatic device MMIO and non-critical PLIC regions assignment. Moreover, in terms of the trap-and-emulate technique insertion, this also has a remarkable impact with its 623 SLoC. These results prove

the small amount of additional code, contributing to the system security, with a small attack surface over code that operates in M-mode level.

Table 5.2: Source lines of code (SLoC) and binary size (bytes).

		SLoC			Size (bytes)			
		c	asm	Total	.text	.data	.bss	Total
OpenSBI (original)	utils	6236	27	6263	41597	1240	13224	56061
	platform/generic	373	0	373	2247	192	544	2983
	sbi	8507	229	8736	52010	816	128376	181202
	firmware	33	860	893	49695	120	0	49815
	Total	15149	1116	16265	145549	2368	142144	290061
OpenSBI (HSP-V)	utils	7159	27	7186	44532	1240	15624	61396
	platform/generic	480	0	480	2273	320	40	2633
	sbi	9130	229	9359	54680	824	128376	183880
	firmware	33	863	896	49691	120	0	49811
	Total	16802	1119	17921	151176	2504	144040	297720

The resulting binary size is detailed in the rightmost section of Table 5.2. To get each table value, the compiler operates with moderate optimizations, i.e., optimizes quite effectively but does not greatly impact compilation time. The total size of statically allocated memory is about 290 KiB and 298 KiB in the original and the HSP-V OpenSBI, respectively. Note that the large *.bss* section size (before and after OpenSBI changes) is mainly due to the static allocation of the hardware and firmware events data to support the Physical Memory Unit (PMU) (125 KiB). Ignoring it, this brings the total size of the final binary to be loaded to about 165 KiB on original OpenSBI and 173 KiB in enhanced one. Therefore, these results reveal a lightweight increase of binary size with just 8 KiB.

5.3 Boot Overhead

This section evaluates the enhanced OpenSBI's overhead on boot time (not the system's overall boot time). The complete boot flow includes several platform-specific boot stages mentioned in Chapter 2.2.4: (i) a ZSBL, which performs low-level initializations and loads the FSBL (the HSS) to on-chip memory from an eMMC; (ii) the FSBL, which prepare the DDRs, manage the cache locking, and loads the SSBL (the OpenSBI); (iii) the SSBL, which runs the OpenSBI and prepare all domain system; and then (iv) the last boot stage, which runs each *domain instance*. The evaluated boot overhead includes the SSBL, i.e., the OpenSBI boot time (coldboot and warmboot time), and the last boot stage, i.e., the boot overhead imposed by each *domain instance* (e.g., the Linux boot time). The coldboot and warmboot terms describe the boot of primary and secondary harts, respectively. Typically, the OpenSBI completes the coldboot first, while

secondary harts wait for the coldboot to finish. After coldboot time, the secondary harts perform the warm-boot, which boots the rest of the platform-defined harts. Regarding these boot overhead measurements, these are only approximate values to the platform's total boot time, as they do not take into account the ZSBL and the FSBL time. Furthermore, results include measurements on both the original and enhanced OpenSBI, the HSP-V, allowing the awareness of imposed boot overhead in a system with or without PLIC partition.

For the measurements performed in this evaluation, it was used the processor frequency and *rdcycle* pseudo-instruction, which reads the clock cycles executed by the processor core. The system environment consider two cases: (i) a small domain (310 KiB image size and 16 MiB of memory) running FreeRTOS, and (ii) a large domain (20 MiB image size and 52 MiB of memory) running Linux. For each *domain instance*, a considered a single-core S-mode execution scenarios with cache locking disabled and enabled (*solo* and *solo-lock*, respectively). When cache locking is enabled, 8 ways are locked/assigned to the large domain and four ways to the small domain. The remaining ways are assigned to the remaining L2 cache configurations, i.e, to the scratchpad when target ways are enabled, and to L2-LIM when target ways are disabled.

Figure 5.4 depicts an overview diagram representing the boot flow of OpenSBI, FreeRTOS, and Linux. During SSBL, OpenSBI initializes the primary hart through the coldboot time, followed by the initialization of the secondary harts through the warmboot time, while simultaneously launching the small domain scenario. At the end of warmboot, OpenSBI launches the large domain scenario. Moreover, during each S-mode boot initialization, the OpenSBI performs the trap-and-emulate technique over critical PLIC accesses, represented in the figure with the label *"T.A.E"*. The measured time are: (i) the OpenSBI initialization (*OpenSBI init. time* of each scenario), the time taken from the first instruction executed by the OpenSBI until to the moment it passes the control to the *domain instance*, i.e., the time taken from coldboot and

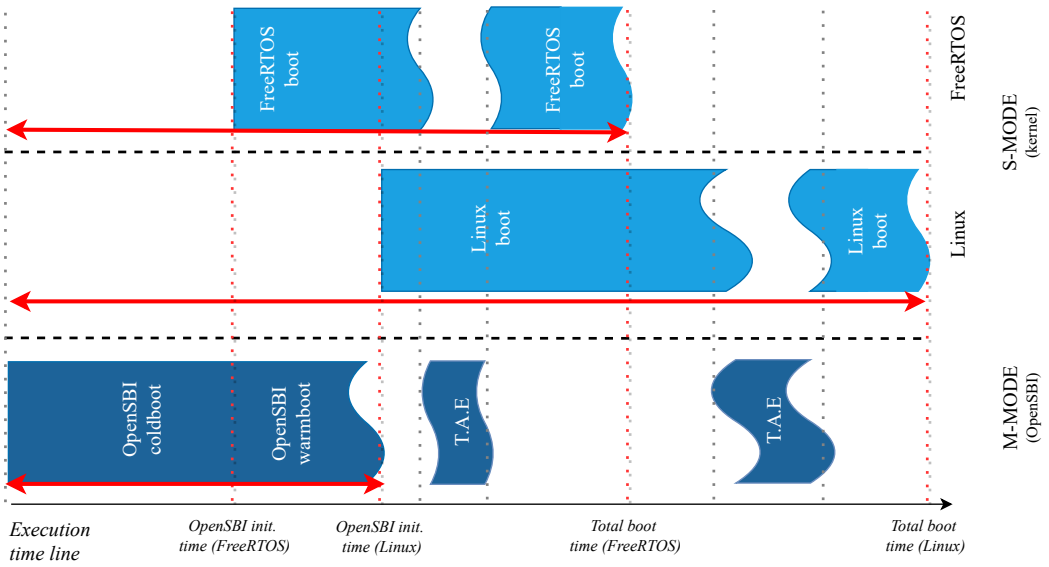


Figure 5.4: Execution time representation of the boot and initialization flow.

warmboot, and (ii) the *Total boot time*, the time taken from the first instruction executed by the OpenSBI until to the first application inside the *domain instance*.

Table 5.3 shows the average results of 20 samples for each measurement, considering each scenario running in the original and HSP-V OpenSBI. The boot flow of the original OpenSBI, i.e., the *OpenSBI init. time*, have significant values in terms of coldboot and warmboot ($74ms$ and $195ms$, respectively). The warmboot takes $121ms$ more than coldboot. In the small domain case, the *Total boot time* prevailed approximately the same, with 89ms, in *solo* and *solo-lock* scenarios. The difference between the *Total boot time* and coldboot time revel $15ms$ to boot the small domain, i.e., the time taken to boot FreeRTOS. On the other hand, in the large domain scenarios, the *Total boot time* increases by approximately 50 ms when comparing *solo* with *solo-lock* scenarios. The difference between the *Total boot time* and warmboot time revel $6,842s$ to boot the large domain scenario, i.e., the time taken to boot Linux.

Regarding measurements in enhanced OpenSBI (the HSP-V scenarios), the first point to highlight is the difference in coldboot overhead. HSP-V *OpenSBI init. time* spends $108ms$ more than the original OpenSBI. This is because all DT parser process is done during the coldboot time and due to the addition of new features to DT configuration, the coldboot need more time to perform the new parser functionalities. Furthermore, coldboot also needs to perform the non-critical PLIC partition and the creation of device MMIO regions. A second point to highlight is the coldboot time being always smaller than warmboot, either in original and in HSP-V scenarios. This is because the secondary harts only resume from suspended execution after the completion of primary hart initialization by the OpenSBI coldboot. Third, is the slight increase between *solo-lock* face *solo* cases in *Total boot time* of both original and HSP-V OpenSBI values. This is mainly because the shared cache space available is decreased due to partitioning using cache locking. Last but not least, the values of *Total boot time* in HSP-V being higher than the original OpenSBI. The increase of these values is due to the virtualization overhead imposed by trap-and-emulate technique, which have notary impact of approximately $136ms$ and $2.8s$ in small and large cases, respectively.

Table 5.3: OpenSBI initialization time (ms) and total boot time (ms).

	Scenario	OpenSBI init. time (ms)		Total boot time (ms)	
		avg	std-dev	avg	std-dev
OpenSBI (original)	freertos solo	74.22	0.009	89.11	0.05
	freertos solo-lock	74.31	0.021	89.21	0.08
	linux solo	195.11	0.108	6793.80	7.35
	linux solo-lock	195.11	0.183	6842.62	4.52
OpenSBI (HSP-V)	freertos solo	182.44	0.030	225.72	0.03
	freertos solo-lock	182.44	0.074	225.76	0.04
	linux solo	195.21	0.094	9590.27	5.67
	linux solo-lock	195.22	0.288	9642.01	7.77

5.4 Performance Overhead and Interference

To assess performance overhead and inter-domain interference, this dissertation adopts the use of MiBench Embedded Benchmark Suite. MiBench is a collection of 35 benchmarks divided into six sub-groups, with each subset focusing on a particular embedded industry segment: automotive (including industrial control), consumer devices, office automation, networking, security, and telecommunications. This dissertation focus the evaluation on the automotive subset, as this is one of the main application domains targeted by related work (e.g. Bao). The automotive suite includes three high memory-intensive benchmarks, that are more susceptible to interference due to LLC and memory contention (*qsort*, *susan corners*, and *susan edges*).

Each benchmark was ran for four different system configurations targeting a dual-core design: (i) a single hart running a *domain instance* (*solo*); (ii) a single hart running a *domain instance* with cache locking enable(*solo-lock*); (iii) two *domain instances*, each with a single hart assigned, one being the target and the other injecting interference (*interf*); (iv) two *domain instances*, each with a single hart assigned, one being the target and the other injecting interference, with cache locking enable (*interf-lock*).

Hosted scenarios with cache locking aim at evaluating the effects of partitioning micro-architectural resources at the domain level and to what extent it can mitigate interference. The target benchmark executes in a Linux-based domain running in one hart, and adding interference with other domain also running in one hart, running a bare-metal application. To add interference, the ad-hoc bare-metal application continuously writes and reads a 3 MiB array with a stride equal to the cache line size (64 bytes). The platform's cache topology allows for 16 ways, each way consisting of 128 KiB. When enabling locking, this evaluation follow the same cache lock approach in Section 5.3, assigning four ways (512 KiB) to bare-metal and eight ways (1 MiB) to Linux-based domain. The remaining four ways were reserved for the HSS (L2-LIM) and for OpenSBI scratchpad space (as mentioned in begin of Section 5.3).

Figure 5.5 presents the results as performance normalized to *solo* scenario, meaning that higher values report worse results. Each bar represents the average value of 100 samples on top of *solo* bar. For each benchmark, is mentioned the execution time (i.e., absolute performance) at the top of the *solo* execution bar. This evaluation take five main conclusions. Firstly, when cache locking (*solo-lock*) is enabled, the performance overhead is further increased. This extra overhead is explained by the fact that only part of the LLC cache is available for the target domain. Secondly, when the system is under significant interference (*interf*), there is a considerable decrease of performance, in particular, for the memory-intensive benchmarks, i.e., qsort (small), susan corners (small), and susan edges (small). For instance, for the susan corners (small) benchmark, the performance overhead increases by 22%. Thirdly, cache locking can reduce the interference (*interf-lock*) by almost 2%. Fourthly, the less memory-intensive benchmarks, i.e., basicmath and bitcount, are less vulnerable to cache interference. Lastly, benchmarks which handle smaller datasets are more susceptible to interference.

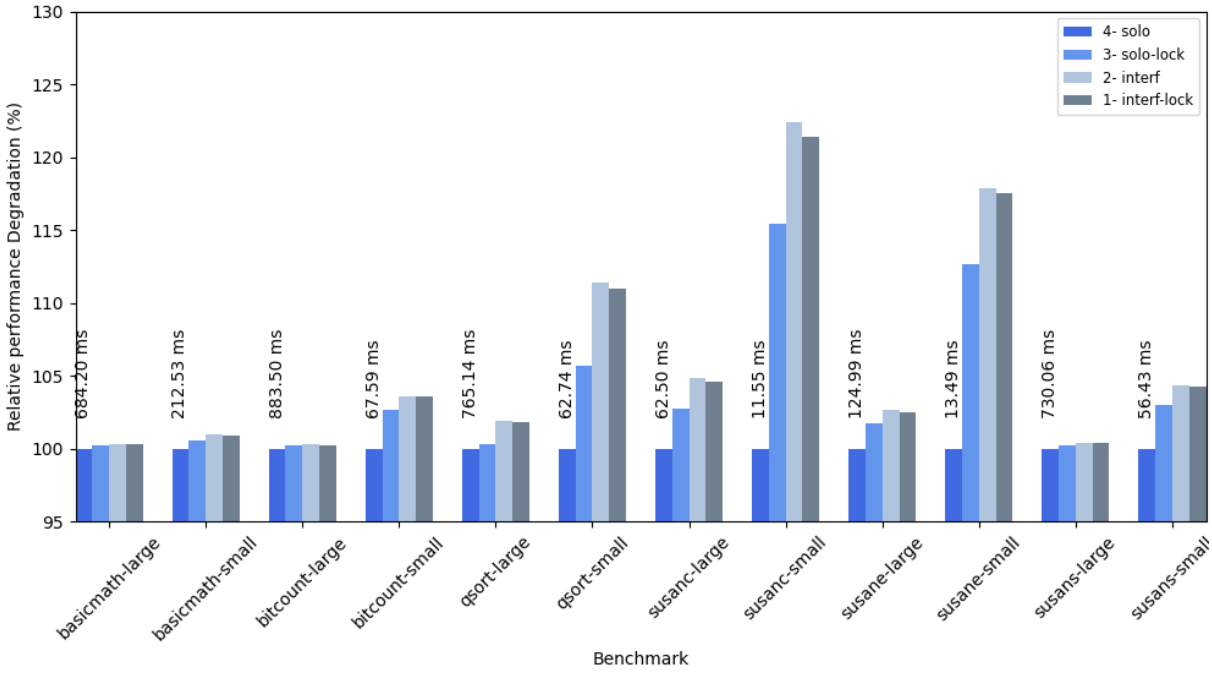


Figure 5.5: Relative performance overhead of MiBench automotive suite.

5.5 Interrupt Latency

To measure interrupt latency, this evaluation crafted a minimal bare-metal benchmark application. This bare-metal application continuously read the value of *rdcycle* (reference cycle instant) while waiting for an external interrupt (triggered when a serial device receives a character). When the external interrupt is triggered, the latency equals the difference between the reference cycle instant and the first cycle instant inside the interrupt handler. The latter preserves the number of executed cycles to start handling the interrupt. Then, the latency time is obtained by dividing the number of cycles by the processor's frequency (625MHz). Figure 5.6 summarizes the 500 sample results for both scenarios, i.e., (i) Interrupt latency in

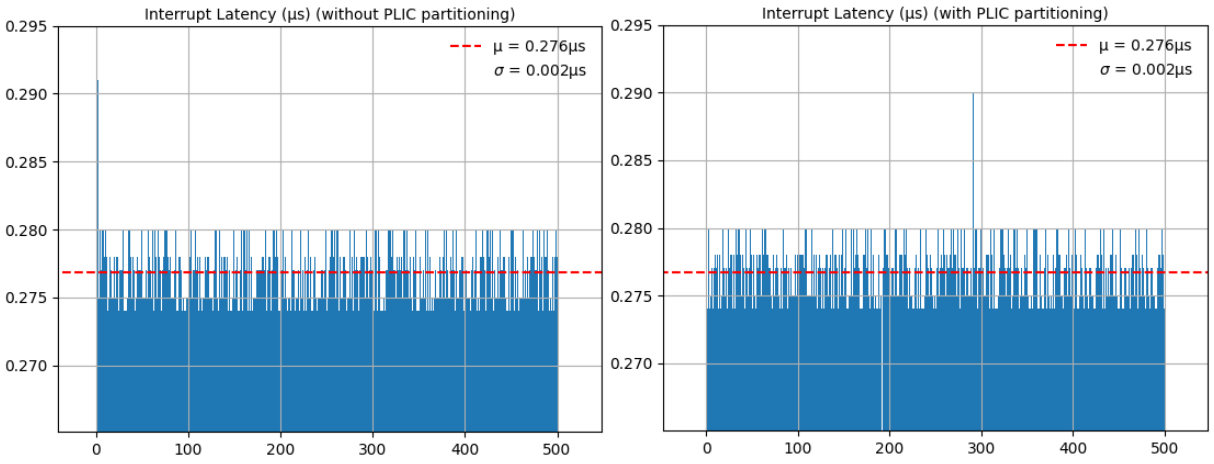


Figure 5.6: Interrupt latency measurements before, and after OpenSBI modifications.

unmodified OpenSBI (without PLIC partitioning) and (ii) interrupt latency in modified OpenSBI (with PLIC partitioning). Since the interrupts are delegated to the S-mode by the *mideleg* CSR (mention in Chapter 2.2.4), the system does not require context switch when an interrupt occurs, consequently modifications over OpenSBI do not impact into interrupt latency. Therefore, when comparing the interrupt latency before and after OpenSBI modifications, the system perform identically. Figure 5.6 depicts interrupt latency similar in both scenarios, with an average latency time of $276n$s with a standard deviation of $2n$s.

5.6 Discussion

The obtained results show that the enhanced OpenSBI in HSP-V achieves a considerable contribution in terms of inter-domain isolation. Starting with the enhancement of OpenSBI configuration, it shows a new easy-to-use property, that effectively supports either the automatic device MMIO regions assignment or the PLIC emulation process. Regarding the trap-and-emulate behavior, the results validate its security since any access to the critical PLIC regions reveals isolation between adjacent *domain instances*.

After proving the proper HSP-V configuration matching and protection of PLIC regions, this section evaluates the TCB with SLoC and the binary size, showing that with HSP-V, the OpenSBI increase about 10% of its code, i.e., 1.656 KSLoC. Moreover, the outstanding OpenSBI enhancement is in the trap-and-emulate process with its 623 SLoC. On the other hand, the binary size results after the compile process (with the first level of optimization) reveal a reduced increase of 8 KiB, when compared the original to the enhanced OpenSBI.

In terms of boot overhead, this dissertation results do not access the ZSBL and FSBL time; however, when comparing the original with the enhanced OpenSBI, the initialization time reveals a notable increase, especially in coldboot and in applications boot time. Since coldboot is responsible for DT parsing and the automatic *domain memory regions* assignment, its overhead increases about $108ms$ more than the original OpenSBI ($74ms$). On another hand, since the initialization of applications requires access to critical PLIC registers, the HSP-V changes have a significant boot overhead when compared to the original, e.g., approximately $3s$ more in Linux and $28ms$ more in FreeRTOS.

Empirical results of performance show a similar pattern to the ones assessed for Bao in an Arm-based platform and later to a RISC-V implementation with virtualization support. Martins et al. [39] employed cache coloring instead of locking to partition the shared cache. Furthermore, instead of three harts injecting interference (case of BAO), HSP-V only has one hart. Due to this limitation, the interference is limited, and hence, the performance results show a residual improvement of 2%. As part of future work, this evaluation can be extended and later repeated with all platform harts under the worst scenarios.

Last but not least, this dissertation shows that interrupt latency results act as expected, i.e., behave similarly in a system with and without PLIC partitioning with $0.276\mu s$. Since the interrupts were previously delegated to be handled in the S-mode layer, the execution does not need to change the context and jump to the OpenSBI trap handler when triggered an interrupt.

In general, the results act as expected, except for the performance results, in which the cache locking approach does not show a considerable improvement of performance over a stressed environment. However, the limited number of provided platform harts justifies this low improvement of performance results. Moreover, the trap-and-emulate technique, used to protect critical PLIC registers, has a significant impact in terms of boot overhead. Since the domain instances require several accesses to critical PLIC regions at boot time, the system inserts a considerable boot overhead, as is the case of the Linux boot, which takes $3s$ more than the original with approximately $7s$. However, despite the imposed boot overhead, the HSP-V implementation does not affect the application execution regarding the interrupt latency, giving the right to the domain instances to handle external interrupts and executing without switching the context mode.

6. Conclusion

In the world of embedded systems, the need to consolidate MCSs into the same hardware platform has increased in the last decades. Over the last few years, academia and industry have proposed different methodologies to prove that virtualization shows as a reliable solution in terms of security, efficiency, and cost-effectiveness. Typically, to provide spacial and temporal isolation, hypervisors technologies in RISC-V architecture, uses an extended supervisor layer to hypervisors, the HS-mode. The imposed virtualization extensions by HS-mode, introduces new CSRs and memory translation through a two-stage address translation. However, the HSP-V consists in a different approach which not leverage virtualization extensions (the HS-mode). Its essence is to eliminate one address translation level by leveraging RISC-V capabilities. Even without virtualization support, static partitioning may be implemented in RISC-V by exploiting the PMP for memory protection and trap-and-emulate technique rather than two-stage translation. As a result, the partitions must be aware of all PA space, employing a para-virtualization approach.

To establish a static partitioning system in RISC-V platforms, this dissertation leverages a thin layer of software, the OpenSBI, running in a higher privileged layer (the M-mode) to operate as a secure monitor and to provide multiple *domain instances* under the same hardware platform. After analyzing an established multiple-domain scenario provided by the original OpenSBI, this dissertation concludes that the inter-domain isolation of the system is broken due to the share of some memory components. Between them, is the RISC-V external interrupt controller, the PLIC, by following a device MMIO structure and leading with all external interrupt platform devices. Therefore, a malicious domain instance can compromise the adjacent *domain instance* behaviors by accessing the adjacent domain PLIC data. This dissertation proposes a inter-domain PLIC partitioning system, either by direct assigning PLIC registers to each of them or by just not assigning PLIC memories to none of them (making the system to trap and switch the context to the OpenSBI privileged level in each access attempt execution). Since the PLIC structure presents different registers' purposes and encode representations, they are classified differently in terms of criticality, being protected either by PMP entry establishment or by trap-and-emulate technique.

In general, due to the enhanced OpenSBI domains' system and following the obtained results, the HSP-V implementation reinforces the inter-domain isolation, limiting the *domain instances* visibility in terms of device MMIO regions and PLIC memory space. Furthermore, when comparing the HSP-V to the original OpenSBI, the following conclusions may be drawn: (i) the enforcement of OpenSBI configuration gives an easy-to-use setup interface; (ii) the automatic memories' assignment to *domain instances*, prevents

domain misconfiguration, provides faster access (when compared to the trap-and-emulate approach) and limit the *domain instances* visibility; (iii) the trap-and-emulate approach has a significant impact in terms of boot overhead, due to the context switch, emulation process and, in some cases, the virtual to physical memory translation; (iv) the cache locking approach is not so efficient in terms of interference mitigation when compared with Bao, but this can be justified by the limitations of evaluation setup (number of provided harts by the FSBL); and (v) the interrupt latency values remain unchanged since the interrupts are delegated to the *domain instance* privileged level.

Due to the OpenSBI contributions: this dissertation is open-source, stimulating further research and development for any other improvement; does not need virtualization extensions to support MCSs; establishes communication between different privilege layers through runtime services; allows the evaluation under several provided multicore RISC-V platforms. With the enhancement of OpenSBI by HSP-V, the OpenSBI can control *domain instance* accesses to critical PLIC memory space, and the OpenSBI can automatically assign device MMIO regions to *domain instances*. Finally, the HSP-V shows the reliability and suitability to real-world embedded applications by augmenting isolation on RISC-V multicore platforms.

6.1 Future Work

This dissertation has successfully developed HSP-V committing all the proposed contributions. Nevertheless, due to time limitations, some optional possible system improvements were left out of implementation for future work. The following points outline the next steps of this research field:

- **Evaluate performance under worst conditions:** Due to the evaluation setup the results face two limitations: (i) the limited number of provided harts and (ii) the limited number of cache ways. In the future, when FSBL provide all application harts to SSBL, it would be of the utmost interest to study the performance behavior in the worst scenarios, i.e., with more harts injecting interference the performance of cache locking mechanism could be more notable;

- **Provide DT configuration properties:** This dissertation already enhance the OpenSBI DT configuration, but after a work analysis, was concluded that another property can be inserted. Looking at cache ways configuration used in setup evaluation, 75% (eight cache ways) of the available L2-Cache (12 cache ways) was locked to the large *domain instances*, while only 25% (four cache ways) was locked to the small *domain instance*. A future step consists of inserting a configurable DT property to define cache locking percentage dedicated to a *domain instance*;

- **Protect Direct Memory Access (DMA) accesses to device MMIO regions:** Typically, to regulate transactions issued by RISC-V harts the PMP unit is used. However, transitions from DMA can reach any memory or any MMIO device without any protection. Therefore, malicious software can compromise all system by using DMA. Since there is not any verification to limit its transactions, the establishment of IOPMP unit would reinforce the inter-domain isolation by preventing accesses

of DMA to unwanted space. A future step consists of integrating the protection of devices with IOPMP;

References

[1] N. Jazdi, "Cyber physical systems in the context of Industry 4.0," in *2014 IEEE international conference on automation, quality and testing, robotics*, pp. 1–4, 2014.

[2] "Theoretical foundations for cyber-physical systems: a literature review", author=Chen, Hong," *Journal of Industrial Integration and Management*, vol. 2, no. 03, p. 1750013, 2017.

[3] J. Martins, A. Tavares, M. Solieri, M. Bertogna, and S. Pinto, "Bao: A lightweight static partitioning hypervisor for modern multi-core embedded systems," in *Workshop on Next Generation Real-Time Embedded Systems (NG-RES 2020)*, 2020.

[4] M. García-Valls, T. Cucinotta, and C. Lu, "Challenges in real-time virtualization and predictable cloud computing," *Journal of Systems Architecture*, vol. 60, no. 9, pp. 726–740, 2014.

[5] S. Pinto, H. Araujo, D. Oliveira, J. Martins, and A. Tavares, "Virtualization on trustzone-enabled microcontrollers? voilà!," in *2019 IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, pp. 293–304, 2019.

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

[7] G. Heiser, "Virtualization for embedded systems," *Open Kernel Labs Technology White Paper*, 2007.

[8] P. Burgio, M. Bertogna, N. Capodieci, R. Cavicchioli, M. Sojka, P. Houdek, A. Marongiu, P. Gai, C. Scordino, and B. Morelli, "A software stack for next-generation automotive systems on many-core heterogeneous platforms," *Microprocessors and Microsystems*, vol. 52, pp. 299–311, 2017.

[9] M. Xu, L. T. X. Phan, H.-Y. Choi, Y. Lin, H. Li, C. Lu, and I. Lee, "Holistic resource allocation for multicore real-time systems," in *2019 IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, pp. 345–356, 2019.

[10] M. Bechtel and H. Yun, "Denial-of-service attacks on shared cache in multicore: Analysis and prevention," in *2019 IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, pp. 357–367, 2019.

[11] M. Bechtel and H. Yun, "Exploiting DRAM bank mapping and HugePages for effective denial-of-service attacks on shared cache in multicore," in *Proceedings of the 7th Symposium on Hot Topics in the*

Science of Security, pp. 1–2, 2020.

[12] L. Gwennap,"Deterministic Processing for Mission-Critical Applications", *The Linley Group*, 2020, [online] Available at: <https://www.linleygroup.com/uploads/sifive-deterministic-processing-wp.pdf>.

[13] N. Suzuki, H. Kim, D. De Niz, B. Andersson, L. Wrage, M. Klein, and R. Rajkumar, "Coordinated bank and cache coloring for temporal protection of memory accesses," in *2013 IEEE 16th International Conference on Computational Science and Engineering*, pp. 685–692, 2013.

[14] K. Asanović and D. A. Patterson, "Instruction sets should be free: The case for risc-v," *EECS Department, University of California, Berkeley, Tech. Rep. UCB/EECS-2014-146*, 2014.

[15] R. Uhlig, G. Neiger, D. Rodgers, A. Santoni, F. Martins, A. Anderson, S. Bennett, A. Kagi, F. Leung, and L. Smith, "Intel virtualization technology," *Computer*, vol. 38, no. 5, pp. 48–56, 2005.

[16] D. Seal, *ARM architecture reference manual*. 2001.

[17] D. Kanter, "RISC-V offers simple, modular ISA," *Microprocessor Report*, 2016.

[18] A. Waterman, Y. Lee, R. Avizienis, D. A. Patterson, and K. Asanović, "The RISC-V Instruction Set Manual Volume II: Privileged Architecture Version 1.9," Tech. Rep. UCB/EECS-2016-129, EECS Department, University of California, Berkeley, 2016.

[19] B. Sá, J. Martins, and S. E. S. Pinto, "A First Look at RISC-V Virtualization from an Embedded Systems Perspective," *IEEE Transactions on Computers*, 2021.

[20] C. Garlati and S. Pinto, "Secure IoT Firmware For RISC-V Processors," *Embbedded world 2021*, 2021.

[21] B. Sá, *RISC-V lightweight virtualization extensions*. PhD thesis, Universidade do Minho, 2021.

[22] D. Lee, D. Kohlbrenner, S. Shinde, K. Asanović, and D. Song, "Keystone: An open framework for architecting trusted execution environments," in *Proceedings of the Fifteenth European Conference on Computer Systems*, pp. 1–16, 2020.

[23] F. R. Yu, J. Liu, Y. He, P. Si, and Y. Zhang, "Virtualization for Distributed Ledger Technology (vDLT)," *IEEE Access*, vol. 6, pp. 25019–25028, 2018.

[24] D. Muench, M. Paulitsch, and A. Herkersdorf, "Iompu: Spatial separation for hardware-based i/o virtualization for mixed-criticality embedded real-time systems using non-transparent bridges," in *2015 IEEE 17th International Conference on High Performance Computing and Communications, 2015 IEEE 7th International Symposium on Cyberspace Safety and Security, and 2015 IEEE 12th International Conference on Embedded Software and Systems*, pp. 1037–1044, 2015.

[25] M. Strobl, M. Kucera, A. Foeldi, T. Waas, N. Balbierer, and C. Hilbert, "Towards automotive virtualization," in *2013 International Conference on Applied Electronics*, pp. 1–6, 2013.

[26] A. Galluccio and P. J. Agrell, "Industry 4.0 in focus: The Adidas Speedfactory," *Université catholique de Louvain*, pp. 21–32, 2022.

[27] C. Garlati and S. Pinto, "A clean slate approach to Linux security RISC-V enclaves," in *Embedded World Conference*, p. 5, 2020.

[28] Vinay T R and A. A. Chikkamannur, "A methodology for migration of software from single-core to multi-core machine," in *2016 International Conference on Computation System and Information Technology for Sustainable Solutions (CSITSS)*, pp. 367–369, 2016.

[29] E. Qaralleh, D. Lima, T. Gomes, A. Tavares, and S. Pinto, "HcM-FreeRTOS: hardware-centric FreeRTOS for ARM multicore," in *2015 IEEE 20th Conference on Emerging Technologies & Factory Automation (ETFA)*, pp. 1–4, 2015.

[30] P. Burgio, M. Bertogna, I. S. Olmedo, P. Gai, A. Marongiu, and M. Sojka, "A Software Stack for Next-Generation Automotive Systems on Many-Core Heterogeneous Platforms," in *2016 Euromicro Conference on Digital System Design (DSD)*, pp. 55–59, 2016.

[31] R. Ernst and M. Di Natale, "Mixed Criticality Systems—A History of Misconceptions?," *IEEE Design Test*, vol. 33, no. 5, pp. 65–74, 2016.

[32] E. Armbrust, J. Song, G. Bloom, and G. Parmer, "On spatial isolation for mixed criticality, embedded systems," in *Proc. 2nd Workshop on Mixed Criticality Systems (WMC), RTSS*, pp. 15–20, 2014.

[33] B. Cilku and P. Puschner, "Towards temporal and spatial isolation in memory hierarchies for mixed-criticality systems with hypervisors," *Proc. ReTiMiCS, RTCSA*, pp. 25–28, 2013.

[34] S. Pinto, A. Oliveira, J. Pereira, J. Cabral, J. Monteiro, and A. Tavares, "Lightweight multicore virtualization architecture exploiting ARM TrustZone," in *IECON 2017-43rd Annual Conference of the IEEE Industrial Electronics Society*, pp. 3562–3567, 2017.

[35] J. Martins, J. Alves, J. Cabral, A. Tavares, and S. Pinto, "μRTZVisor: a secure and safe real-time hypervisor," *Electronics*, vol. 6, no. 4, p. 93, 2017.

[36] E. Schoitsch, "Design for Safety and Security of Complex Embedded Systems: A Unified Approach," *In: Proceedings of the NATO advanced research workshop on cyberspace security and defense*, 2004.

[37] J.-Y. Hwang, S.-B. Suh, S.-K. Heo, C.-J. Park, J.-M. Ryu, S.-Y. Park, and C.-R. Kim, "Xen on ARM: System virtualization using Xen hypervisor for ARM-based secure mobile phones," in *2008 5th IEEE Consumer Communications and Networking Conference*, pp. 257–261, 2008.

[38] C. Dall and J. Nieh, "KVM/ARM: the design and implementation of the linux ARM hypervisor," *Acm Sigplan Notices*, vol. 49, no. 4, pp. 333–348, 2014.

[39] J. Martins, A. Tavares, M. Solieri, M. Bertogna, and S. Pinto, "Bao: A Lightweight Static Partitioning Hypervisor for Modern Multi-Core Embedded Systems," in *Workshop on Next Generation Real-Time Embedded Systems (NG-RES 2020)* (M. Bertogna and F. Terraneo, eds.), vol. 77 of *OpenAccess Series in Informatics (OASIcs)*, (Dagstuhl, Germany), pp. 3:1–3:14, 2020.

[40] G. J. Popek and R. P. Goldberg, "Formal Requirements for Virtualizable Third Generation Architectures," *Commun. ACM*, vol. 17, no. 7, p. 412–421, 1974.

[41] G. Heiser, "The role of virtualization in embedded systems," *ACM*, pp. 11–16, 2008.

[42] T. Xia, J.-C. Prevotet, and F. Nouvel, "Mini-NOVA: A Lightweight ARM-based Virtualization Microkernel Supporting Dynamic Partial Reconfiguration," in *2015 IEEE International Parallel and Distributed Processing Symposium Workshop*, pp. 71–80, 2015.

[43] P. Barham, B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, R. Neugebauer, I. Pratt, and A. Warfield, "Xen and the art of virtualization," *ACM SIGOPS operating systems review*, vol. 37, no. 5, pp. 164–177, 2003.

[44] Z. H. Shah, "Windows Server 2012 Hyper-V: Deploying the Hyper-V Enterprise Server Virtualization Platform," *Packt Publishing*, 2013.

[45] H. Fayyad-Kazan, L. Perneel, and M. Timmerman, "Full and para-virtualization with Xen: a performance comparison," *Journal of Emerging Trends in Computing and Information Sciences*, vol. 4, no. 9, pp. 719–727, 2013.

[46] P. F. R. Garcia, "Hybrid hypervisor partially deployed on FPGA," *repositorium*, 2015.

[47] S. M. Jain, "Virtualization basics," in *Linux containers and virtualization*, pp. 1–14, 2020.

[48] L. Abeni and D. Faggioli, "Using Xen and KVM as real-time hypervisors," *Journal of Systems Architecture*, vol. 106, p. 101709, 2020.

[49] A. Landau, M. Ben-Yehuda, and A. Gordon, "SplitX: Split Guest/Hypervisor Execution on Multi-Core," in *3rd Workshop on I/O Virtualization (WIOV 11)*, 2011.

[50] J. T. Lim, C. Dall, S.-W. Li, J. Nieh, and M. Zyngier, "NEVE: Nested virtualization extensions for ARM," in *Proceedings of the 26th Symposium on Operating Systems Principles*, pp. 201–217, 2017.

[51] L. Jünger, J. L. M. Bölke, S. Tobies, R. Leupers, and A. Hoffmann, "ARM-on-ARM: leveraging virtualization extensions for fast virtual platforms," in *2020 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, pp. 1508–1513, 2020.

[52] E. De Mulder, S. Gummalla, and M. Hutter, "Protecting RISC-V against side-channel attacks," in *2019 56th ACM/IEEE Design Automation Conference (DAC)*, pp. 1–4, 2019.

[53] M. Silva, T. Gomes, and S. Pinto, "Leveraging RISC-V to build an open-source (hardware) OS framework for reconfigurable IoT devices," *CARRV2021*, vol. 6, 2021.

[54] A. Waterman, Y. Lee, D. A. Patterson, and K. Asanovi, "The risc-v instruction set manual. volume 1: User-level isa, version 2.0," tech. rep., California Univ Berkeley Dept of Electrical Engineering and Computer Sciences, 2014.

[55] S. Chadwick, S. Graham, and J. Dean, "Performance Implications for Multi-Core RISC-V Systems with Dedicated Security Hardware," in *International Conference on Cyber Warfare and Security*, vol. 17,

pp. 440–448, 2022.

[56] K. Cheang, C. Rasmussen, D. Lee, D. W. Kohlbrenner, K. Asanovic, and S. A. Seshia, "Verifying RISC-V physical memory protection," in *IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS) Workshop on Secure RISC-V Architecture Design*, 2020.

[57] S. Pinto and C. Garlati, "User mode interrupts: A must for securing embedded systems," in *Embedded World Conference*, 2019.

[58] S. Pinto and J. Martins, "The industry-first secure IoT stack for RISC-V: a research project," in *RISC-V Workshop,(Zurich)*, 2019.

[59] F. Arakawa, A. Tsukamoto, K. Suzaki, and M. Ikeda, "Examination of applicability of RISC-V security specifications to low-end processors," 2020.

[60] P. Lucas, K. Chappuis, M. Paolino, N. Dagieu, and D. Raho, "Vosysmonitor, a low latency monitor layer for mixed-criticality systems on armv8-a," in *29th Euromicro Conference on Real-Time Systems (ECRTS 2017)*, 2017.

[61] Y. Babar, "GRUB Bootloader," in *Hands-on Booting*, pp. 133–181, 2020.

[62] T. Lu, "A survey on risc-v security: Hardware and architecture," *arXiv preprint arXiv:2107.04175*, 2021.

[63] P. Chandrasekaran, K. B. Shibu Kumar, R. L. Minz, D. D'Souza, and L. Meshram, "A multi-core version of FreeRTOS verified for datarace and deadlock freedom," in *2014 Twelfth ACM/IEEE Conference on Formal Methods and Models for Codesign (MEMOCODE)*, pp. 62–71, 2014.

[64] A. Patel, M. Daftedar, M. Shalan, and M. W. El-Kharashi, "Embedded hypervisor xvisor: A comparative analysis," in *2015 23rd Euromicro International Conference on Parallel, Distributed, and Network-Based Processing*, pp. 682–691, 2015.

[65] R. Ramsauer, J. Kiszka, D. Lohmann, and W. Mauerer, "Look mum, no VM exits!(almost)," *arXiv preprint arXiv:1705.06932*, 2017.

[66] C. Hernandez, J. Flieh, R. Paredes, C.-A. Lefebvre, I. Allende, J. Abella, D. Trillin, M. Matschnig, B. Fischer, K. Schwarz, *et al.*, "SELENE: Self-monitored dependable platform for high-performance safety-critical systems," in *2020 23rd Euromicro Conference on Digital System Design (DSD)*, pp. 370–377, 2020.

[67] F. Caforio, P. Iannicelli, M. Paolino, and D. Raho, "Vosysmonitorv: a mixed-criticality solution on linux-capable risc-v platforms," in *2021 10th Mediterranean Conference on Embedded Computing (MECO)*, pp. 1–4, 2021.

[68] G. Heiser, G. Klein, and J. Andronick, "seL4 in Australia: from research to real-world trustworthy systems," *Communications of the ACM*, vol. 63, no. 4, pp. 72–75, 2020.

[69] M. Boubakri, F. Chiatante, and B. Zouari, "Open Portable Trusted Execution Environment framework

for RISC-V," in *2021 IEEE 19th International Conference on Embedded and Ubiquitous Computing (EUC)*, pp. 1–8, 2021.

[70] M. Boubakri, F. Chiatante, and B. Zouari, "Towards a firmware TPM on RISC-V," in *2021 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, pp. 647–650, 2021.

[71] A. Waterman, Y. Lee, D. A. Patterson, and K. Asanović, "The RISC-V compressed instruction set manual, version 1.7," *EECS Department, University of California, Berkeley, UCB/EECS-2015-157*, 2015.