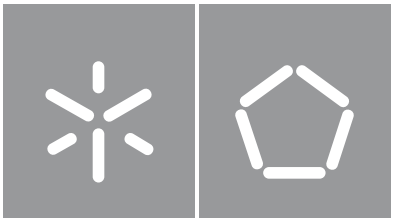
Universidade do Minho
Escola de Engenharia

Bruno Vilaça de Sá

RISC-V Lightweight Virtualization Extensions

março de 2021

RISC-V Lightweight Virtualization Extensions
Bruno Sá
UMinho | 2021



Universidade do Minho
Escola de Engenharia

Bruno Vilaça de Sá

RISC-V Lightweight Virtualization Extensions

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

Trabalho efetuado sob a orientação do
Professor Doutor Sandro Pinto

março de 2021

Acknowledgements

I would like to start by expressing my utmost gratitude to my advisor, Dr. Sandro Pinto, for allowing me to be part of this fantastic work and for the confidence placed on me. Thank you for believing me.

I would like to also thank José Martins for all the contributions and recommendations that made all of this work possible. Thank you for all your guidance and continuous support.

My sincere thank you to every friend and family that was supportive and encouraging with me throughout this journey. I want to leave a huge and special thank you to my mother, father, brother, and grandmother, to whom I owe everything, for all the love and constant support throughout my life.

To my girlfriend Catarina, thanks for all the unconditional love and support you have given me. Thank you for always being there for me.

Last but not least, I want to leave a big thanks to my parents-in-law and sister-in-law for all the support and care you have been giving me throughout my journey.

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

RISC-V Lightweight Virtualization Extensions

Na última década, a virtualização tornou-se uma tecnologia essencial para os servidores, mas também para várias indústrias, tais como a indústria automóvel e controlo industrial. Em sistemas embebidos, o número de requisitos tem vindo a aumentar de forma constante nos últimos anos. Ao mesmo tempo, a pressão do mercado para minimizar o tamanho, peso, consumo e custo (SWaP-C) tem impulsionado a consolidação de vários subsistemas, tipicamente com níveis de criticidade distintos, na mesma plataforma de hardware. Em resposta, o meio académico e a indústria têm-se focado no desenvolvimento de suporte de hardware para apoiar a virtualização, e no suporte para estas tecnologias em *hypervisors* de referência.

Os recentes avanços na área de aquitetura de computadores estão ligados ao desenvolvimento de uma arquitetura inovadora designada de RISC-V. RISC-V distingue-se das plataformas convencionais ao oferecer uma arquitectura de conjunto de instruções (ISA) livre e aberta, com um esquema de extensão modular e altamente personalizável. Todo o ecosistema RISC-V tem crescido a um ritmo alucinante, motivado pela promessa de transformar a indústria de hardware, da mesma forma que o sistema operativo Linux transformou a indústria de software. A especificação da arquitetura RISC-V define que todo o suporte de hardware para a virtualização é especificado através extensão de *hypervisor* (*H-extension*).

Esta dissertação descreve a primeira implementação e avaliação pública da última versão da especificação da extensão *hypervisor* RISC-V (*H-extension* v0.6.1) num processador RISC-V (Rocket Chip). A avaliação foi feita em um *hypervisor* de partição estática *open-source* com suporte para RISC-V, denominado Bao. O controlador de interrupções, designado de PLIC na arquitetura RISC-V, foi também extendido para permitir a injecção de interrupção directa em máquinas virtuais com latência baixa e determinística. A infra-estrutura de gestão de temporização do sistema também foi modificada para evitar suporte via emulação (e toda a perda de desempenho associada). Todos os testes foram realizados no FireSim, um simulador acelerado por FPGA (precisão ao ciclo), e o sistema foi também testado com sucesso numa Zynq UltraScale+. Disponibilizamos a nossa implementação de hardware para a comunidade RISC-V, que está atualmente a ser utilizada como referência para ratificar a especificação da *H-extension*.

Palavras-chave: Virtualização, RISC-V, *H-extension*, *Hypervisor*, Sistemas Embebidos

Abstract

RISC-V Lightweight Virtualization Extensions

In the last decade, virtualization has become a key enabling technology for servers, but also for several embedded industries such as the automotive and industrial control. In the embedded space, the number of requirements has been steadily increasing for the past few years. At the same time, the market pressure to minimize size, weight, power, and cost (SWaP-C) has pushed for the consolidation of several subsystems, typically with distinct criticality levels, onto the same hardware platform. In response, academia and industry have focused on developing hardware support to assist virtualization, and adding upstream support for these technologies in mainstream hypervisor solutions.

Recent advances in computing systems have brought to light an innovative computer architecture named RISC-V. RISC-V distinguishes itself from traditional platforms by offering a free and open instruction set architecture (ISA) featuring a modular and highly customizable extension scheme that allows it to scale from tiny embedded microcontrollers up to supercomputers. RISC-V is going towards mainstream adoption under the premise of disrupting the hardware industry, such as Linux has disrupted the software industry. As part of the RISC-V privileged architecture specification, hardware virtualization support is specified through the hypervisor extension (H-extension).

This dissertation describes the first public implementation and evaluation of the latest version of the RISC-V hypervisor extension (H-extension v0.6.1) specification in a Rocket Chip core. To perform a meaningful evaluation for modern multi-core embedded and mixed-criticality systems, we used an open-source static partitioning hypervisor with support for RISC-V, named Bao. We have also extended the RISC-V platform-level interrupt controller (PLIC) to enable direct guest interrupt injection with low and deterministic latency, and we have enhanced the timer infrastructure to avoid trap and emulation overheads. Experiments were carried out in FireSim, a cycle-accurate, FPGA-accelerated simulator, and the system was also successfully deployed and tested in a Zynq UltraScale+ MPSoC ZCU104. Our hardware implementation was open-sourced and is currently in use by the RISC-V community towards the ratification of the H-extension specification.

Keywords: Virtualization, RISC-V, H-extension, Hypervisor, Embedded Systems

Table of Contents

List of Figures

List of Tables

List of Listings

Acronyms

ABI Application Binary Interface

API Application Programming Interface

ASIC Application Specific Integrated Circuit

CLINT Core-Level Interrupt Controller

COTS Commercial Off-The-Shelf

CPU Central Processing Unit

DMA Direct Memory Access

DRAM Dynamic Random Access Memory

FIQ Fast Interrupt Request

FPGA Field-Programmable Gate Array

GIC Generic Interrupt Controller

GPOS General Purpose Operating System

HDL Hardware Description Language

I/O-MMU Input Output Memory Managment Unit

I/O Input Output

IPI Inter-Processor Interrupt

IRQ Interrupt Request

ISA Instruction Set Architecture

ISR Interrupt Service Routine

KVM Kernel-based Virtual Machine

LUT Look Up Table

MCU Microcontroller Unit

MMU Memory Management Unit

OS Operating System

PC Program Counter

PLIC Platform-Level Interrupt Controller

PTE Page Table Entry

RAM Random Access Memory

RISC Reduced Instruction Set Computer

RTOS Real Time Operating System

SBI Supervisor Binary Interface

SMP Symmetric Multi Processing

SoC System on Chip

TLB Translation Look-aside Buffer

VE Virtualization Extensions

VMCB Virtual Machine Control Block

VMM Virtual Machine Monitor

VM Virtual Machine

Chapter 1

Introduction

Embedded and cyber-physical systems have been evolving exponentially over the years. Traditionally, embedded systems were single-purpose systems with limited functionalities, minimal communication, and simple interfaces. Nowadays, modern embedded systems are evolving and assuming characteristics of general-purpose systems while maintaining real-time constraints. Embedded systems that were typically low on complexity are now a sophisticated collection of subsystems with different critical levels interacting with each other. This growth in complexity, coupled with the desire to join together all of these subsystems into one platform, forced researchers and industries to search for new methodologies and technologies.

Embedded virtualization technology arises as a natural response to consolidate several systems with different criticality levels into a single platform while guaranteeing isolation and security. This technology introduces an extra layer of software, denoted hypervisor, that virtualizes resources for one or more OSes running atop of it, giving the OSes the illusion of complete control over the system hardware [Masood et al., 2015]. Each guest OS runs on its partition, which guarantees spatial and temporal isolation. This allows a general-purpose system (GPOS) to run alongside a real-time operating system (RTOS) while leveraging from the best of the two systems, i.e., the rich set of API offered by GPOSs for graphical user interfaces and the real-time execution offered by the RTOS. Additionally, the possibility of consolidating several functionalities in separate partitions on the same hardware platform also enables re-using legacy software stacks and maximizes overall performance and hardware usage. Consequently, this means less engineering effort and time-to-market, which eventually increases revenue. In response, efforts have been made in the academia and industry on adding hardware support virtualization (e.g., Arm Virtualization Extensions) and adding upstream support for these technologies in mainstream hypervisor solutions [Kloda et al., 2019, Dall and Nieh, 2014].

Embedded software stacks are progressively targeting powerful multi-core platforms, endowed with complex memory hierarchies [Burgio et al., 2017, Xu et al., 2019]. Despite the logical CPU and memory

isolation provided by existing hypervisor layers, there are several challenges and difficulties in proving strong isolation due to the reciprocal interference caused by micro-architectural resources (e.g., last-level caches, interconnects, and memory controllers) shared among virtual machines (VM) [Kloda et al., 2019, Martins et al., 2020]. This issue is particularly relevant for mixed-criticality applications, where security- and safety-critical applications need to coexist along with non-critical ones. In this context, a malicious VM can either implement denial-of-service (DoS) attacks by increasing their consumption of a shared resource [Bechtel and Yun, 2019, Bechtel and Yun, 2020] or indirectly access other VM's data leveraging existing timing side-channels [Ge et al., 2018]. To tackle this issue, the industry has been developing specific hardware technology (e.g., Intel Cache Allocation Technology), and the research community has been very active in proposing techniques based on cache locking, cache/page coloring, or memory bandwidth reservations [Yun et al., 2013, Mancuso et al., 2013, Kloda et al., 2019, Xu et al., 2019, Farshchi et al., 2020].

Recent advances in computing systems have brought to light an innovative computer architecture named RISC-V. RISC-V distinguishes itself from traditional platforms by offering a free and open instruction set architecture (ISA) featuring a modular and highly customizable extension scheme that allows it to scale from tiny embedded microcontrollers up to supercomputers. RISC-V is going towards mainstream adoption under the premise of disrupting the hardware industry, such as Linux has disrupted the software industry. As part of the RISC-V privileged architecture specification, hardware virtualization support is specified through the hypervisor extension (H-extension). The H-extension specification is currently in version 0.6.1, and no ratification has been achieved so far.

Although virtualization is well established in desktops and servers, shifting it to the embedded world can be demanding. While on servers, each OS runs in its independent partition, the same does not apply to embedded systems where the functionalities are spread across multiple cooperating subsystems. Hardware support for virtualization is seen as a significant improvement, but few studies address the low/high tier gap, i.e., while some of these architectural features could be essential in the high tier systems, in low tier systems, it could be overkill and even introduce overhead. This dissertation intends to contribute to the maturity of current hardware virtualization support in the embedded world, mainly focusing on RISC-V. This will be accomplished by implementing the RISC-V hypervisor extension in a RISC-V Rocket Chip and re-examining existing virtualization support, and, if necessary, proposed some modifications that could be beneficial in the context of the embedded systems.

1.1 Goals

This dissertation aims at contributing to the embedded virtualization space, in particular for RISC-V, with the following goals:

- Implement the RISC-V H-extension in a RISC-V Soc generator (i.e., Rocket Chip). This means to study the inner structure of the Rocket Chip generator and extend all necessary modules accordingly to the H-extension specification.
- Evaluate RISC-V H-extension's effects in a multi-core environment by using an embedded hypervisor named Bao, with focus on the following metrics: performance and inter-VM interference, hardware-resources, interrupt latency.
- Analyze the RISC-V virtualization support and propose new extensions that could improve virtualization efficiency.

1.2 Document Structure

This remaining document is organized as follows:

- Chapter 2 provides background knowledge and related work regarding virtualization, hypervisors, and RISC-V.
- Chapter 3 discusses the RISC-V hypervisor extension and explains some of the main features.
- Chapter 4 identifies some issues with current timer virtualization support in RISC-V platforms and proposes some virtualization extensions to the current RISC-V specification.
- Chapter 5 identifies some issues with current external interrupt controller (PLIC) virtualization support in RISC-V platforms and proposes some virtualization extensions to the PLIC.
- Chapter 6 describes how we extended a RISC-V SoC generator and a RISC-V core, i.e., Rocket Chip and Rocket core, with hypervisor extensions and our interrupt virtualization enchantments on the PLIC and timer interrupt controller, i.e., CLINT.

- Chapter 7 presents all tests and evaluations conducted on an FPGA and RISC-V simulator, i.e., Firesim. Our focus is hardware-resources, performance and inter-VM interference, and interrupt latency.

- Chapter 8 concludes this dissertation. We identify some existing gaps in the RISC-V privilege specification and suggest future work to address such existing limitations.

Chapter 2

Background Concepts and Related Work

In the following chapter, firstly, we introduce some basic theoretical concepts regarding virtualization and hypervisors architectures. Next, we expose how other mainstream instruction set architectures (ISAs) add hardware support for virtualization. Then, we make a brief description of RISC-V ISA and present a well-known RISC-V system on chip (SoC) generator and a popular RISC-V core. Lastly, we present some hypervisors and microkernels with support for RISC-V.

2.1 Virtualization

In the last decade, virtualization has been gaining popularity among researchers and enterprises as it enables multiple operating systems (OSes) to run concurrently on the same physical machine by introducing a thin software abstraction layer (hypervisor), thus providing security and isolation [Garcia, 2015, Masood et al., 2015]. Virtualization has been used mainly on desktop and server environments for load balancing across clusters, power management, and firewalling by isolating high-risk services [Heiser, 2008]. In the last few years, developments have been made in embedded virtualization as a solution to integrate real-time control functionalities with rich environments, providing real-time and non-real-time characteristics [Heiser, 2011, Reinhardt and Morgan, 2014, Lee et al., 2016]. Various applications can be identified ranging from mobile devices [Liao et al., 2017] to medical devices, automotive [Thiebaut et al., 2016, Masmano et al., 2009] and aerospace [Pinto et al., 2016], where the urge to consolidate several subsystems of different critical levels, also called mixed-criticality systems (MCSs), is present [Heiser, 2011].

2.1.1 Hypervisor Architectures

To properly classify a hypervisor architecture, three aspects must be taken into account: the hierarchy (its position on the system stack), the internal hypervisor architecture, and the virtualization technique. When defining the hypervisor in terms of hierarchy, two different types can be identified:

- Type 1 hypervisors (bare-metal Hypervisors): located at the first level of the software stack, as so, controlling the hardware and managing the VMs.
- Type 2 hypervisors: located at the second level of the software stack, it runs on top of an OS (located at the first level). The OS is responsible for managing the hardware, and the hypervisor is seen as just another application, enabling several Virtual Machines (VMs) to run atop the hypervisor.

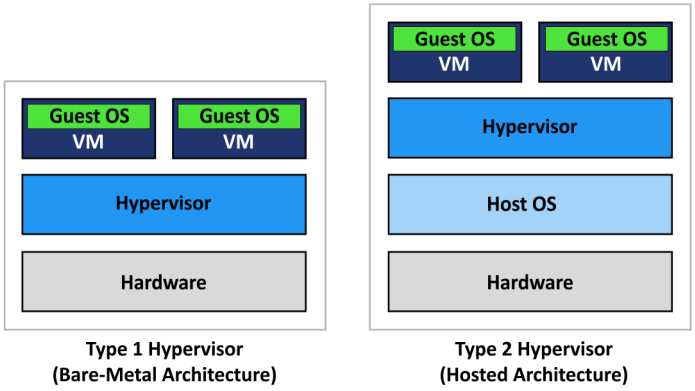


Figure 2.1: Type 1 and Type 2 Hypervisors, in [Nakivo, 2018].

In terms of the internal architecture of the hypervisor, two main architectures can be identified [Shropshire, 2014]:

- Monolithic Hypervisors: condenses a set of subsystems, i.e., CPU scheduling, memory management, file management, device drivers, and other operating system functions within a single static binary file. This means that all the subsystems share a common address space and privileged mode. Although this induces a larger memory footprint, it allows subsystems to interact more quickly and efficiently. One main disadvantage is that any service crash would mean an overall system failure.
- Microkernel Hypervisors: designed to minimize memory footprint, microkernel architectures retain basic tasks, i.e., CPU scheduling, memory management, and inter-process communication and use a separate partition to handle storage, hypercalls, and other support functions.

In terms of virtualization technology, there are three major approaches:

- Full-virtualization enables guest OSes to run in isolation without any modifications [Menascé, 2005]. Two main approaches can be identified [Lingeswaran, 2017], software-assisted which relies on either binary translation or trap-and-emulate of privileged instructions and hardware-assisted full virtualization uses CPU hardware virtualization extensions [Dall, 2018, Varanasi and Heiser, 2011].

- Paravirtualization [Lingeswaran, 2017] requires changes to the guest OSes source code, as privileged code needs to be replaced by the corresponding hypercall, i.e., set of APIs provided by the hypervisor to the VMs to deal with privileged operations. Although it provides some performance advantages, as each OS needs to be changed to fit the hypervisor, it also leads to high design cost [Garcia, 2015].

- Previrtualization [Lingeswaran, 2017] is similar to paravirtualization, except that it attempts to eliminate the additional engineering effort with it. Instead of manually changing the guest OS source code to be compatible with the hypervisor, the porting is done automatically by changing sensitive instructions and replacing them with calls to the hypervisor, also called hypercalls.

2.2 Hardware Virtualization Technology

Modern computing architectures such as x86 and Arm have been adding added hardware extensions to assist virtualization to their CPUs for more than a decade. Intel has developed the Intel Virtualization Technology (Intel VT-x) [Uhlig et al., 2005], the Advanced Programmable Interrupt Controller (APIC) virtualization extension (APICv [Nguyen, 2016]), and Intel Virtualization Technology for Directed I/O (Intel VT-d [Uhlig et al., 2005]). Intel has also included nested virtualization hardware-based capabilities with Virtual Machine Control Structure (VMCS) shadowing. Arm included the virtualization extensions (Arm VE) since Armv7-A and developed additional hardware to the Generic Interrupt Controller (vGIC) for efficient virtual interrupt management [Limited, 2016]. Recently, Arm has announced a set of extensions in the Armv8.4-A that includes the addition of secure virtualization support [Arm Ltd., 2018b] and the Memory System Resource Partition and Monitoring (MPAM) [Arm Ltd., 2018a]. There are additional COTS hardware technologies that have been leveraged to assist virtualization, namely the MIPS virtualization module [Moratelli et al., 2016], AMD Virtualization (AMD-V), and Arm TrustZone [Pinto and Santos, 2019, Pinto

et al., 2019]. The academia has also been focused on devising and proposing custom hardware virtualization support, and mechanisms [Garcia et al., 2014, Xu et al., 2017, Lim et al., 2017]. Garcia et al. proposed a hardware-based hypervisor implementation [Garcia et al., 2014]. Xu et al. proposed vCAT [Xu et al., 2017], i.e., dynamic shared cache management for multicore virtualization platforms based on Intel's Cache Allocation Technology (CAT). With NEVE [Lim et al., 2017], Lim et al. developed a set of hardware enhancements to the Armv8.3-A architecture to improve nested virtualization performance, which was then included in Armv8.4-A.

2.2.1 Arm Virtualization Extensions

Originally, Arm architecture was not developed to be a virtualizable architecture, as it contains sensitive and unprivileged instructions [Dall, 2018]. This violates Popek and Goldberg's [Popek and Goldberg, 1974] virtualization theorem, which states that architecture is classically virtualizable if all the ISA sensitive instructions (instructions that are critical for correct virtualization) are also privileged (instructions that trap in user mode but not in kernel mode). Arm introduced Virtualization Extensions (VE) to the Armv7-A and Armv8-A architectures.

Running VMs implies the existence of privileged CPU virtualization mode as the hypervisor needs to control the physical hardware. To accomplish that, Arm changed its sensitive operations to act over a virtual state instead of a physical CPU, thus providing isolation and reduce traping to the hypervisor. Arm implemented this by introducing a new and more privileged CPU mode, the hypervisor mode, designed to run hypervisors as simply as possible. Software running in hypervisor mode can configure hardware to trap from kernel or user into hypervisor mode in specific events, e.g., the hypervisor can choose if reading CPU ID registers should trap or not to the hypervisor.

In terms of virtual memory support, Arm virtualization extensions added a 2-stage address translation to allow the hypervisor to have full control over all the guest physical memory accesses. The first stage translates Virtual Addresses (VAs) into Guest Physical Addresses (GPAs), and the second stage translates GPAs into Physical Addresses (PAs). Typically, in a non-virtualized memory-management unit (MMU), traps to the hypervisor were required every time the VMs accessed virtual memory for keeping track over guests page tables (PTs) by resorting to the shadow tables technique. With the 2-stage translation, the hypervisor no longer needs to keep shadow tables in memory, leaving the MMU responsible for doing the address translation. Consequently, this leads to a reduction in the VM exits and eventually overall performance improvement [Garcia, 2015].

It is worth mentioning Arm optional Security Extension, called Trustzone. The Trustzone splits execution into two separate worlds, secure and non-secure and provides a special mode (monitor mode) to switch between the two worlds. When this extension is implemented, the system boots up at the secure world and then switches to the non-secure. With the rising of the Internet of Things (IoT) era, embedded OSes have been including features to ease the implementation of IoT applications [Silva et al., 2019, Costa et al., 2020]. Trustzone has also been used in IoT applications for security purposes by providing a trusted execution environment (TEE) and therefore separate critical software components from non-critical [Oliveira et al., 2018b, Cerdeira et al., 2020]. Originally, Trustzone technology was not targetted for virtualization, but efforts have been made to leverage this technology for embedded systems applications [Martins et al., 2017]. In Ref.[Pinto et al., 2014], a Trustzone assisted hypervisor, called LTZVisor [Pinto et al., 2017b], was presented, where a general-purpose operating system (GPOS) in the non-secure world runs side-by-side with a real-time operating system (RTOS) in the secure world. The LTZVisor runs in a secure mode (monitor mode) to manage the whole platform. The hypervisor is responsible for configuring memory, interrupts, devices assigned to each VM, and inter VM-communication [Oliveira et al., 2018a]. To ensure that real-time environment constraints are met, an asymmetric design was followed, dictating that the secure VM has greater scheduling priority than the non-secure one. This prevents the OS from preempting the RTOS but can cause starvation in the non-secure world. To tackle this problem, an extension to the LTZVisor was presented in [Pinto et al., 2017a], where a multicore asymmetric multiprocessing configuration was used. The GPOS runs in the non-secure world in one core, and the RTOS runs in the secure world in another core. One of the main limitations founded in LTZVisor is the number of supported virtual machines, as it only supports two VMs. In Ref.[Cicero et al., 2018], a dual-hypervisor design for Arm platforms is presented. It proposes one hypervisor for the non-secure world that leverages the Arm-VE and another one that relies on para-virtualization. There are benefits in terms of performance, reliability, and security in such approach. First, in case of failure in the non-secure world, the secure world is not affected. Second, no switch from a secure world to a non-secure is required to handle virtualization in the non-secure world.

The Generic Interrupt Controller(GIC) [Limited, 2016] v2 and v3 is the system interrupt controller used in Arm architectures. The GIC architecture splits into two parts, a Distributor (only one in all systems) and a CPU interface (one interface per CPU), which are accessed over a Memory-Mapped interface (MMIO).

GIC Virtualization Extensions introduced another CPU interface to each processor, specially dedicated to interacting with VMs. A VM running in a processor with interrupt virtualization extensions communicates with the virtual CPU interface. Also, a VM that receives virtual interrupts from this interface cannot

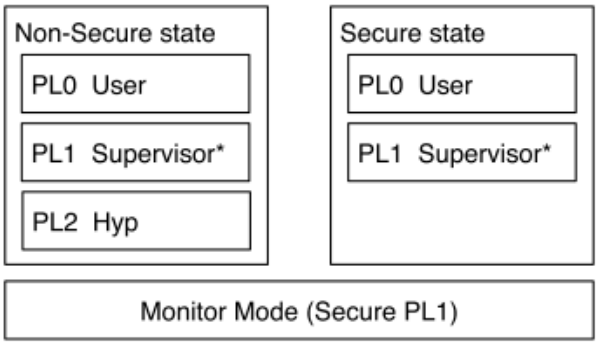


Figure 2.2: Armv7 modes, in [Dall, 2018].

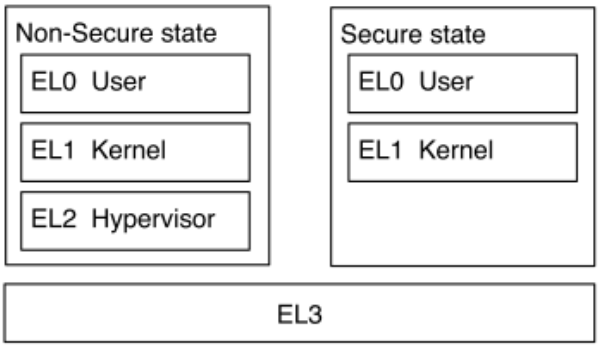


Figure 2.3: Armv8 modes, in [Dall, 2018].

distinguish them from physical interrupts, which eases the programmer's model, as virtual and physical interrupt handling is identical.

The virtual CPU interface is divided into two blocks, each separated into 4KiB address regions:

- Virtual Interface Control - The GIC virtual interface control registers comprise all the active and pending virtual interrupts; this block is typically programmed by the hypervisor. A special subset of registers was added to the control registers, the ListRegisters (LRs), to generate virtual interrupts.

- Virtual CPU interface - Each virtual CPU interface block provides physical signaling of virtual interrupts to the connected processor. The hypervisor signal these interrupts to the current VM running on that processor. The GIC virtual CPU interface registers, accessed by the VM, provide an interrupt control and status information for the virtual interrupts. The format of these registers is similar to the format of the physical CPU interface registers.

The GIC virtual interface control registers and the virtual interface registers are held at a non-secure memory map. Therefore, the hypervisor uses the 2-stage translation to prevent access from the VM to the virtual interface control registers. Moreover, given that there is only one distributor, the hypervisor needs to trap and emulate any access performed by the VM's.

The hypervisor is responsible for handling all IRQs and selecting which ones are forwarded to which VM by injecting virtual interrupts through programming the LRs that state the currently visible interrupts to that machine. There is a limit to the total number of pending, active, or active and pending interrupts inherent to the maximum number of registers in the LR. This means that when an interrupt arrives and there is no space left in the LRs, the hypervisors needs save the interrupt details and insert them into the LRs when available. Additionally, to handle de complexity of injecting interrupt using LR, Arm provides a

special type of interrupt, denoted maintenance interrupts, that signaled the hypervisor when certain events occur in the LRs, e.g., no pending interrupt or the guest interrupt acknowledge.

Looking at the Arm GIC interrupt virtualization support, we can identify some advantages and disadvantages:

Advantages:

- Provides a guest interface called virtual CPU interface, which he can interact with no need to trap and emulate accesses;
- Direct injection of virtual interrupts;
- Supports an unlimited number of interrupts;
- The VM can acknowledge physical interrupts;
- Provides interrupt number translation, meaning that a physical interrupt ID can be different from the virtual interrupt ID viewed by the guest.

Disadvantages:

- In case of the interrupts are intended for the current VM, and there is no space in the LR, interrupt details need to be stored in memory, e.g., interrupt ID and priority. Then, the hypervisor will wait for the guest to acknowledge at least one interrupt from LR and insert the previous stored one;
- Needs to trap all guest OS accesses to the GIC Distributor, as there is only one instance;
- Each context switch, the hypervisor needs to re-configure the LRs.

2.2.2 Intel Virtualization Extensions

The Intel x86 architecture was not developed to be classically virtualizable. Robin et al. [Robin and Irvine, 2000] proved that x86 contained several instructions that violated the Popek and Goldberg theorem, e.g. *pushf*, *popf*. To tackle this problem, Intel introduced Intel Virtualization Technology (VT) as a hardware virtualization support technology. Instead of having a separated mode for the hypervisor like Arm, Intel has two different modes (root and non-root mode) independent of the CPU protection mode. The hypervisor runs in root mode while the guest in non-root mode. Moreover, to support virtualization, sensitive

operations in non-root cause traps to the hypervisor, allowing the hypervisor to fully control the guest's execution. Additionally, Intel added hardware support for a VM control structure in memory (VMCS), used to automatically save a restore de CPU's state when switching to or from root mode using the VMX transitions, defined as a single atomic operation. Each VM has its own VMCS, which is divided into six logical groups: guest-state area, host-state area, VM-execution control fields, VM-exit control fields, VM-entry control fields, and VM-exit information fields.

Regarding memory virtualization, Intel has a similar mechanism as the Arm 2-stage address translation, called Extended Page Table (EPT) [Bhargava et al., 2008], which enables the guest to have full control over page tables reducing the VM exits to VMM and memory footprint, as no shadow page-tables are required [Ke, 2009].

Intel uses the Advanced Programmable Interrupt Controller (APIC) [Intel, 2011] for interrupt handling. Each processor core as one local APIC, which deals with both internal and external I/O APIC interrupts. In a multiprocessor system, the local APIC also sends and receives messages from other processors, also called inter-processor interrupt (IPI). For handing external interrupts, the external I/O APIC gathers interrupt events caused by I/O devices and routes them to the local APIC as interrupt messages. Each local APIC consists of memory-mapped registers used to control interrupt delivery and generation of IPI messages. When an interrupt is signaled to the CPU core, the Interrupt Descriptor Table (IDT) is used to decide which interrupt handler to take.

Typically, in virtualized environments without hardware support, the hypervisor needs to emulate local APIC registers accesses and adequately deliver them to the guest IDT. This incurs in a high VM-exit as the hypervisor needs to intercept interrupts and inject them into the guest and emulate the APIC behavior when the guest handles the interrupt. This impacts performance as each VM exit involves context save and restore and running the hypervisor code, which also pollutes the CPU cache. LAPIC registers emulation is one of the major causes of virtualization overhead due to the high rate of VM exits [Tu et al., 2015]. Intel address this issue by introducing the APICv, which virtualizes the LAPIC registers in the processor. Control fields were added to VMCS to allow APIC virtualization and virtual interrupt delivery. Each VM was presented with a 4KiByte virtual-APIC page to virtualize the APIC register's access and manage virtual interrupts. APIC-reads are emulated and no longer cause VM-exit, and APIC-write not longer causes a fault VM-exit but a trap-like VM-exit.

Posted Interrupts

Intel's developed a mechanism for posting virtual interrupts into a currently running vCPU without any VM exit. Posted Interrupts adds a new hardware data structure that the hypervisor uses to post interrupts. The process is quite complex and involves several steps. First, an interrupt arrives, and the hypervisor decides which vCPU is going to receive it. Then, it posts this virtual interrupt into the post-interrupt descriptor of the target vCPU. Next, the hypervisor sends an IPI with the posted-interrupt notification vector to the core in which vCPU is running. Finally, the hardware detects the IPI and synchronizes the posted interrupt with its virtual APIC page, delivering the pending virtual interrupt directly. The interrupts are then decoded into a specific handler based on the guest IDT. The interrupt completion is done without any VM-exit as it can directly interact with the EOI registers in the virtual APIC page. This mechanism allows the hypervisor to directly update the state of the guest interrupts while the VM is still running. Additionally, hypervisors can direct-assign devices to guests by configuring the interrupt remapping table to inject external interrupts directly to the guest [Zhang et al., 2018].

Advantages:

- Provides a virtual APIC interface, which the guest can interact with without causing a VM-exit;
- Direct injection of virtual interrupts through posted interrupts mechanism;
- Use trap-like APIC-write without the need to trap and emulate;
- Guest device direct-assignment.

Disadvantages:

- Interrupt mechanism is complex;
- Guest cannot access the I/O APIC;
- APIC virtual page is stored in memory.

2.3 RISC-V

RISC-V is a popular instruction set architecture (ISA), originally targeted for educational purposes [Patterson and Waterman, 2017]. The RISC-V distinguishes itself from its competitors by offering a free and open

standard ISA with a highly customizable and modular extension scheme that allows it to scale from small embedded systems with restricted resources up to high-performance computers.

It offers a base integer ISA with a minimal set of instructions (RV32I, RV64I, and RV128I) that every core must implement and a set of optional extensions to the base ISA [Waterman et al., 2014]. The RV128I is still being developed and has not yet achieved a frozen state, while the RV32I and RV64I are already frozen by the RISC-V foundation [Waterman et al., 2014]. The RISC-V specification for each new extension follows a well-defined extension development lifecycle. First, it starts as a draft, and when it has been stable for quite some time, it approaches a frozen state, after which it will enter a period of public review before finally being ratified. Table 2.1 presents all optional extensions available and their current state. Additionally, the ISA is very flexible to custom implementations, so opcode space is also reserved for non-standard extensions so that designers can easily add new features to their processors that will not conflict with existing software compiled to the standard.

Name	**State**	**Description**
M	Frozen	Integer Multiplication and Division Extension
A	Frozen	Atomic Instructions Extension
F	Frozen	Single-Precision Floating-Point Extension
D	Frozen	Double-Precision Floating-Point Extension
Q	Frozen	Quad-Precision Floating-Point Extension
L	Open	Decimal Floating-Point Extension
C	Frozen	Compressed Instructions Extension
B	Open	Bit Manipulation Extension
J	Open	Dynamically Translated Languages Extension
T	Open	Transactional Memory Extensions
P	Open	Packed-SIMD Instructions Extension
V	Open	Vector Operations Extensions
N	Open	User-Level Interrupts Extension
H	**Open**	**Hypervisor Extension**
S	Open	Supervisor-level Instructions Extension

Table 2.1: RISC-V standard extensions.

The RISC-V ISA defines three privilege levels of protection for hardware threads or harts (RISC-V terminology for CPUs) (see table 2.2). These privileged levels provide protection between different software stack components, and any attempts to perform operations not allowed by the current privilege mode will cause an exception to be raised. The machine mode (M-mode) is the highest privileged mode and the only mandatory level for a RISC-V platform. The code running on this level has low-level access and is usually trusted. User-mode (U-mode) and supervisor-mode (S-mode) are intended for conventional applications and OSes usage, respectively. Recently, the RISC-V privileged ISA was endowed with hardware virtualization support, which is the main scope of this dissertation, and as such, we will address it separately (see Chapter 3). Each privilege is endowed in a set of control and status registers (CSRs) to control the execution mode behavior and trap handling (see Table 2.3), e.g., M-mode and S-mode have *mcause and scause* to store the exception/interrupt code when a trap occurs. By default, all interrupts and exceptions are handled by the execution environment running in M-mode, which can delegate it to S-mode by writing to *mideleg* and *medeleg*. Additionally, the execution environment is also responsible for defining the number

Encoding	**Abbreviation**	**Name**
0	U-mode	User mode
1	S-mode	Supervisor mode
2	Reserved	Reserved for future use
3	M-mode	Machine mode

Table 2.2: RISC-V privileged levels.

Mode	**Register**	**Description**
M-mode	mstatus	machine status register
	mtvec	machine trap-vector base-address register
	medeleg and mideleg	machine trap delegation registers
	mip and mie	machine pending and enable interrupt registers
	mtime and mtimecmp	machine timer registers
	mepc	machine exception program counter
	mcause	machine cause register
	mtval	machine trap value register
S-mode	status	supervisor status register
	stvec	supervisor trap-vector base-address register
	sepc	supervisor exception program counter
	scause	supervisor cause register
	stval	supervisor trap value register

Table 2.3: M-mode and S-mode available CSRs.

of harts and privileges available, protect memory region accesses and control virtual memory translations, and trapping and emulating CSRs or sensitive instructions not allowed to execute in lower privilege levels. When running in M/S/U systems, the M-mode provides the supervisor execution environment (SEE) for the OS running in S-mode by implementing the Supervisor Binary Interface (SBI). The SBI acts as an abstraction layer between the supervisor and the actual hardware platform, which increases portability and allows OSes to run seamlessly in every platform. The SBI is implemented via *ecall* from S-mode to M-mode and temporarily registers (*a0-a7*) to pass arguments. There are several implementations of the SBI interface, e.g., OpenSBI, and Berkely Boot Loader (BBL), which implement several functionalities defined by the SBI interface, e.g., timer, IPIs, serial and *fence* instructions.

2.3.1 Rocket Chip Overview

Rocket Chip [Asanovi et al., 2016] is an open-source SoC generator based on RISC-V Instruction Set Architecture (ISA) developed by UC Berkeley, suitable for research and industrial purposes. Rocket Chip is a design generator with extensive parametrization which makes it flexible and highly customizable for any application [Asanovi et al., 2016, Berkeley, 2019b].

Rocket Chip generator is implemented in Chisel [Berkeley, 2019a], an open-source hardware construction language (HDLs) embedded in Scala. Chisel has some features that are not found in conventional hardware descriptions languages, e.g., rich type system with support for structured data, width inference for wires, high-level descriptions of state machines and bulk wiring operations [Asanovi et al., 2016]. Chisel can easily be converted to synthesizable Verilog suitable for FPGA and ASIC design tools. Additionally, it also allows designers to generate fast, cycle-accurate RTL simulators implemented in C++ that can be used to simulate a Rocket Chip instance. An example of a Rocket Chip instance featuring a dual Rocket core is presented in figure 2.4.

The Rocket core is an in-order core generator with a 5-stage pipeline (see Figure 2.5) that implements the RV32IMAFD and RV64IMAFD ISAs and supports M-mode, S-mode and U-mode privilege levels. Each Rocket core has its own page-table walker (PTW) unit, translation look-aside buffer (TLB), and L1 instruction and data caches on what is known as Rocket Tile. Each Tile is connected to the rest of the system through the central *SystemBus* using the TileLink protocol [SiFive, 2017]. The Tilelink is the free and open standard bus protocol used in RISC-V systems to interconnect multiple peripherals, caches, memory processors, and coprocessors. Due to the extensive parameterization supported in Rocket Chip, the TileLink is built on top of a framework for negotiating parameters (e.g., bus width), denoted Diplomacy [SiFive, 2017].

The Rocket Chip provides multiple buses with different purposes, defined as follow:

- *MemoryBus* - Connects to DRAM controller using a TileLink to AXI converter.
- *Control Bus* - Connects all standard peripherals such as the BootROM, the PLIC, the CLINT, and the Debug Unit to the *SystemBus*, so that these MMIO devices may be accessible in memory.
- *FrontBus* - Serves as an interface so that DMA devices may access the memory system directly.
- *PeripheryBus* - Connects all additional peripherals and also offers the possibility to attach a vendor-supplied AXI4IP through the Tilelink to AXI converter.
- *InterruptBus* - Connects both external and local interrupts (software and timer) to each hart coming from the PLIC and CLINT, respectively.

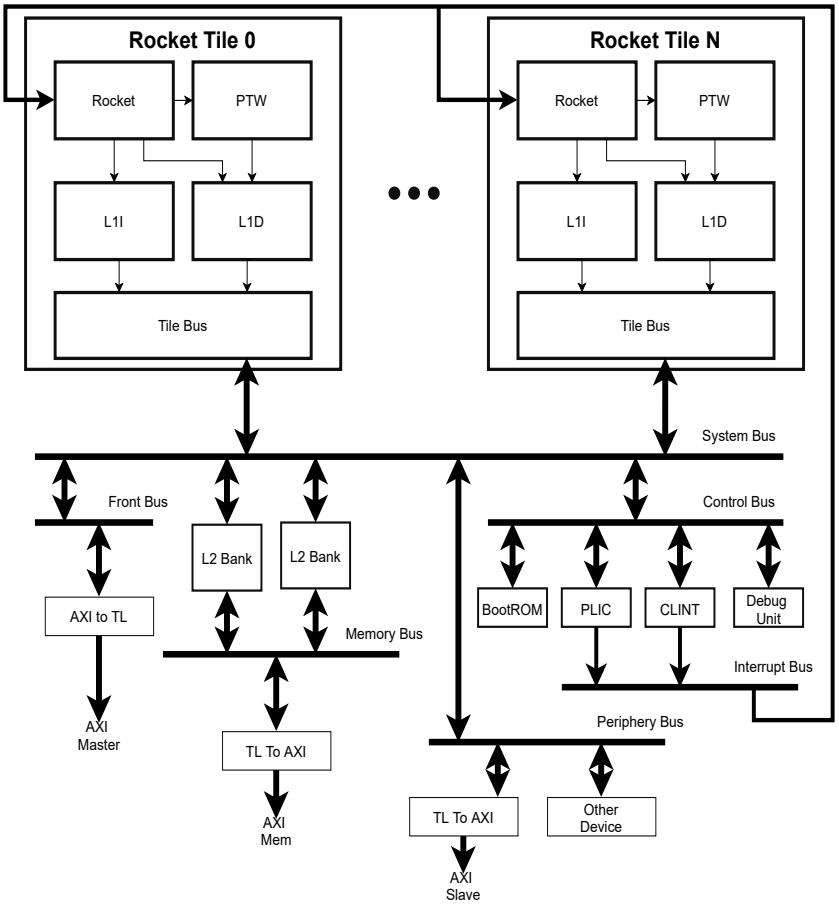


Figure 2.4: Rocket Chip SoC block diagram (adapted from [Chipyard, 2020])

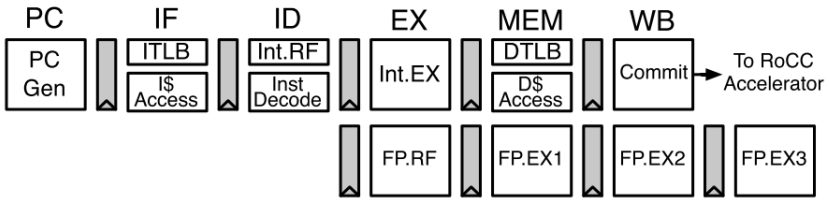


Figure 2.5: Rocket core pipeline, in [Asanovi et al., 2016].

2.4 Hypervisors and Microkernels for RISC-V

KVM [Lublin et al., 2007] and Xvisor [Patel et al., 2015] were the first hypervisors adding support for the RISC-V H-extension in QEMU. KVM [Lublin et al., 2007] is a type-2 hosted hypervisor integrated into Linux's mainline as of 2.6.20. KVM targets mainly enterprise virtualization setups for data centers and private clouds. Xvisor [Patel et al., 2015] is a type-1 monolithic hypervisor targeting embedded systems with soft real-time requirements. Both hypervisors are officially part of the RISC-V Software Ecosystem and naturally have been used by technical groups as reference implementations to validate and evolve the H-extension. RVirt is an S-mode trap-and-emulate hypervisor for RISC-V, written in Rust. Contrarily to KVM and XVisor, RVirt [Behrens et al., 2020] can run in RISC-V processors without hardware virtualization support. Diosix [Williams, 2020] is another lightweight bare-metal hypervisor written in Rust for RISC-V. Similar to RVirt, Diosix can run in RISC-V cores that lack the H-extension, leveraging the physical memory protection (PMP) to achieve isolation. Xtratum [Crespo et al., 2010], a hypervisor primarily developed for safety-critical aerospace applications, has also recently been ported to support RISC-V ISA [Gómez et al., 2020], following the same PMP-based concept for isolation as Diosix. Xen [Hwang et al., 2008] and Jailhouse [Ramsauer et al., 2017], two widely used open-source hypervisor solutions, have already given preliminary steps towards RISC-V support. However, as of this writing, upstream support for RISC-V is not yet available, but it is expected to be included in the foreseeable future. seL4 [Klein et al., 2009], a formally verified microkernel, is also verified on RISC-V [Heiser, 2020]. Other commercial microkernels already support RISC-V. Preeminent examples include the SYSGO PikeOS and the Wind River VxWorks.

2.4.1 Bao

Bao [Martins et al., 2020] is an open-source static partitioning hypervisor developed with the main goal of facilitating the straightforward consolidation of mixed-criticality systems, thus focusing on providing strong

safety and security guarantees. It comprises only a minimal, thin-layer of privileged software leveraging ISA virtualization support to partition the hardware, including 1-to-1 virtual to physical CPU pinning, static memory allocation, and device/interrupt direct assignment. Bao implements a clean-slate, standalone component featuring about 8 KSLoC (source lines of code), which depends only on standard firmware to initialize the system and perform platform-specific tasks such as power management. It provides minimal inter-VM communication facilities through statically configured shared memory and notifications in the form of interrupts. It also implements from the get-go simple hardware partitioning mechanisms such as cache coloring to avoid interference in caches shared among the multiple VMs. Bao has already been ported to RISC-V using the QEMU implementation of the H-extension.

Chapter 3

RISC-V Hypervisor Extension

The RISC-V privilege architecture splits the execution modes into three privileges: (i) the M-mode is the most privileged and where the firmware implementing the standard SBI (supervisor binary interface) lives, (ii) the S-mode whose primary targets are rich operating systems (e.g., Linux) a offers virtual memory capabilities through a dedicated memory management unit (MMU) and (iii) U-mode for running regular applications. The ISA modularity allows implementations from simple embedded systems (featuring M-mode) to high-end systems (featuring M/S/U) targetting personal or server computing.

The ISA was designed from the ground-up to be classically virtualizable [Popek and Goldberg, 1974] by allowing to selectively trap accesses to virtual memory management control and status registers (CSRs) as well as the timeout and mode change instructions from supervisor/user to machine mode (e.g., *mstatus*'s *TVM* bit enables trapping of *satp*, the root page table pointer, while setting the *TSR* bit will cause the trap of the *sret* instruction used by the supervisor to return to user mode). Furthermore, RISC-V provides fully precise exception handling, guaranteeing the capturing of the exact state of execution at the time of an exception. The ISA simplicity coupled with its virtualization-friendly design allows the easy implementation of a hypervisor recurring to traditional techniques (e.g., full trap-and-emulate, shadow page tables) and the emulation of the hypervisor extension from machine mode.

The RISC-V specification was recently endowed with hardware virtualization support, aiming to aid hypervisors and ease virtualization overhead while increasing efficiency with optional hypervisor extensions ("H"). The hypervisor extension is still on version 0.6.1 in draft state, soon to be ratified, and in the scope of this dissertation, only this version will be addressed.

3.1 Execution Modes

Like most other mainstream ISAs, the latest draft of the RISC-V privilege architecture specification offers hardware virtualization support, i.e., the optional hypervisor extension ("H"), to increase virtualization efficiency. As illustrated by Figure 3.1, the H-extension modifies the supervisor mode to an *hypervisor-extended supervisor mode* (HS-mode), which similarly to Intel's VT-x [Intel, 2011] root mode, is orthogonal to the new *virtual supervisor mode* (VS-mode) and *virtual user mode* (VU-mode), and therefore can easily accommodate both bare-metal and hosted (a.k.a. type-1 and -2) as well as hybrid hypervisor architectures.

Unavoidably, the extension also introduces a 2-stage address translation where the hypervisor has control over the page-tables mapping from Guest Physical Address (GPA) to Host Physical Address (HPA)). The *virtualization mode*, which encodes if the hart is running in a virtual machine, is controlled by the V bit. When V=1, the hardware thread is either in VS-mode or in VU-mode, and the 2-stage address translation is active. When V=0, the hardware thread is either in M-mode, in HS-mode or, in U-mode, and the 2-stage address translation is disabled. With the newly added execution modes, novel CSRs and hypervisor instructions appeared, and the old ones were extended (e.g., M-mode *mip, mie*), so that the hypervisor could easily have control over virtual execution and the 2-stage address translation.

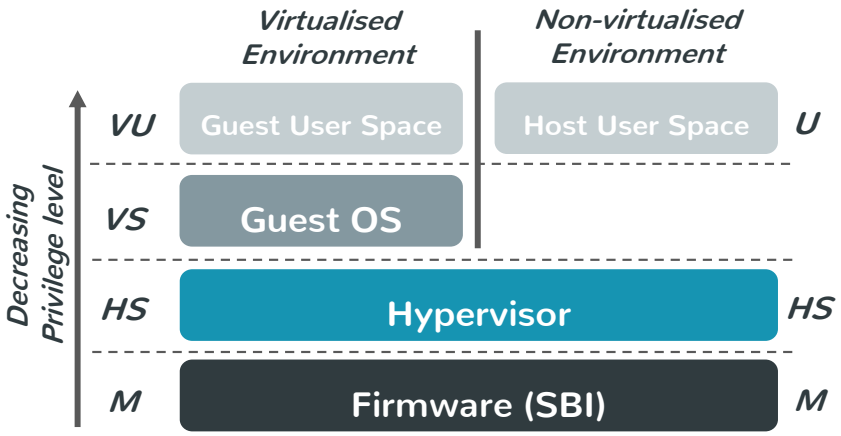


Figure 3.1: RISC-V privileged levels: machine (M), hypervisor-extended supervisor (HS), virtual supervisor (VS), and virtual user (VU).

3.2 Hypervisor and Virtual Supervisor CSRs

The RISC-V privilege specification offers hardware support for virtualization (H-extension) by virtualizing the S-mode execution into a virtualized supervisor (VS) and extending the S-mode into an HS-mode. The

HS-mode execution is the same as S-mode, with additional hypervisor CSRs to control the guest virtual memory and behavior, defined as follow: *hstatus, hedeleg, hideleg, hvip, hip, hie, hgeip, hgeie, hcounteren, htimedelta, htimedeltah, htval, htinst, and hgatp*. Furthermore, it defines a set of background virtual supervisors CSRs, which are copies of the original S-mode registers for when running in VS-mode, therefore allows regular S-mode OS to run unmodified in VS-mode. Table 3.1 summarizes all HS- and VS-mode registers and their respective functionalities. The *virtualization mode* dictate which set of registers are effectively active. In VS-mode (V = 1), normal S CSRs are swapped by the VS CSRs and vice-versa in HS-mode.

Mode	**Register**	**Description**
HS-mode	hstatus	hypervisor status register
	hedeleg and hideleg	hypervisor trap delegation registers
	hvip, hip and hie	hypervisor interrupts registers
	hgeip	hypervisor guest external interrupts register
	htimedelta	hypervisor time delta register
	htval	hypervisor trap value register
	htinst	hypervisor trap instruction register
	hgatp	hypervisor guest address translation and protection register
VS-mode	vsstatus	virtual supervisor status register
	vsip and vsie	virtual supervisor interrupt pending and enable registers
	vstvec	virtual supervisor trap-vector base address register
	vsscratch	virtual supervisor scratch register
	vsepc	virtual supervisor exception program counter register
	vscause	virtual supervisor cause register
	vstval	virtual supervisor trap value
	vstap	virtual supervisor address translation and protection register

Table 3.1: HS-mode and VS-mode CSRs summary.

Given the ISA simplicity and virtualization-aware design, the hypervisor extensions are also easily emulatable on M/S/U systems by traping to M-mode. In that case, the guest in S-mode and the hypervisor in M-mode encapsulating the former by intercepting page table accesses (using the *mstatus.TVM* feature

allied with shadow PTs technique) and swapping the background S CSRs upon privilege switching (using the *mstatus.TVM* feature).

The VS-execution exception behavior is controlled by the hypervisor status register (*hstatus*). Similar to the M-mode register *mstatus* in M/S/U systems, the *hstatus* CSR enables the hypervisor to trap virtual memory management control and status registers (CSRs) as well as the timeout and mode change instructions from virtual supervisor/user to hypervisor mode (e.g., *hstatus.VTVM* bit enables trapping of *vsatp*, the root page table pointer, while setting the *VTSR*, *VTW* bit will cause the trap of the *sret* instruction used by the virtual supervisor to return to user mode and the trap of the *wfi* instruction). These particular features are most convenient in nested virtualization. The guest hypervisor and guest OS run at VS-level, and the host hypervisor performs all background tasks of swapping CSRs state upon privileged level transfers and by controlling the page-tables (PTs) by resorting to shadow tables. Furthermore, *hstatus* is used to store VS-mode to HS-mode trap-related information, e.g., the virtualization mode (*SPV*) and the privileged mode (*SPVP*) at the time of the trap.

Regarding interrupts, the RISC-V ISA defines three types per hart: (i) external interrupts (EI), (ii) timer interrupts (TI), and (iii) software interrupts (SI) (normally used as IPIs). Each interrupt can be re-directed to one of the privileged modes by setting the bit for target interrupt/mode in a per-hart interrupt pending bitmap, which might be directly driven by some hardware device or set by software. This bitmap is fully visible to M-mode through the *mip* CSR, while the S-mode has a filtered view of its interrupt status through *sip*. These concept was extended to the new virtual modes through the *hvip*, *hip*, and *vsip* CSRs. As further detailed in Chapter 4, in current RISC-V implementations, a hardware module called the CLINT (core-local interrupter) drives the timer and software interrupts, but only for machine mode. Supervisor software must configure timer interrupts and issue IPIs via SBI, invoked through *ecall* (environment calls, i.e., system call) instructions. The firmware in M-mode is then expected to inject these interrupts through the interrupt bitmap in the supervisor. The same is true regarding VS interrupts as the hypervisor must itself service guest SBI requests and inject these interrupts through *hvip* while in the process of invoking the machine-mode layer SBI. When the interrupt is triggered for the current hart, the process is inversed: (i) the interrupt traps to machine software, which must then (ii) inject the interrupt in HS-mode through the interrupt pending bitmap, and then (iii) inject it in VS mode by setting the corresponding *hvip* bit.

Regarding the exceptions, the H-extension augments the trap encoding with multiple exceptions to support VS execution. For instance, it adds guest page fault exceptions for when guest translations at the 2-stage MMU unit fails and VS *ecall* exception to support system calls performed at VS-mode.

Additionally, to ease the hypervisor extension emulation when running nested hypervisors, the H-extension adds the virtual instruction exception for when invalid operations are taken in VS-mode (e.g., HS- and VS- mode CSRs access in VS-mode, *sfence* instruction or to access *satp*, when *hstatus.VTVM*=1 and others). The virtual instruction exception is basically the same as an illegal instruction exception but for VS/VU-mode access. By distinguishing between virtual and regular illegal instructions, virtual traps are expected to be handled much quicker as the M-mode can directly delegate them to HS-mode. Whereas with only illegal instructions, it would require to first trap to M-mode, which would then delegate it by software to HS-mode.

3.2.1 Guest External Interrupts

The H-extension *hgeip* register introduces a novel mechanism that allows direct assignment of interrupts to individual virtual machines running in VS-mode without hypervisor intervention. The *hgeip* is a bitmap register where each bit holds all interrupt status intended for a given vhart. The number of implemented bits in *hgeip*, denoted *GEILEN*, limits the number of virtual harts that may directly receive guest external interrupts. Note that bit 0 of the *hgeip* is hardwired to zero, so the maximum number that *GLEIN* can be for any hart is 31 (RV32 implementations) or 63 (RV64 implementations). The hypervisor selects the current running vhart through the *hstatus.VGEIN* field. So, when a vhart is running, the assigned *hgeip* bit is converted to a *VSEIP* interrupt. Moreover, the hypervisor can still receive non-active guest context interrupts by simply enabling the corresponding guest context *hgeie* bit and enable guest external interrupts by setting *hie.SGEI* bit. However, this feature needs to be supported by the external interrupt controller. Unfortunately, the PLIC is not yet virtualization-aware. A hypervisor must fully trap-and-emulate PLIC accesses by the guest and manually drive the VS external interrupt pending bit in *hvip*. The sheer number of traps involved in these processes is bound to impact interrupt latency, jitter, and, depending on an OS tick frequency, overall performance. For these reasons, interrupt virtualization support is one of the most pressing open-issues in RISC-V virtualization. In Chapter 5, we describe our approach for addressing this issue.

3.3 Traps

As stated earlier, the RISC-V ISA supports the delegation of interrupts/exceptions to less privileged execution modes. For instance, when a trap occurs in VS-mode, it only goes to VS-mode if M-mode delegates

it to HS-mode by writing to *mideleg/medeleg* and if HS-mode further delegates it to VS-mode by writing to *hedeleg/hideleg*. Furthermore, at each trap, the machine performs a simple context save, specifies the cause, and provides, when necessary, some exception specific information (e.g., when page faults occur in VS-level, *htval* is set with the guest physical address) before traping to the appropriate mode for handling. Additionally, there a two optional registers, the *htinst* and *mtinst*, which expose a trapping instruction in an easily digestible, pre-decoded form so that the hypervisor can more quickly handle the trap while avoiding reading the actual guest instruction and polluting the data cache. After completing handling, a return must be performed by using *mret* or *sret* instruction to restores the machine state. Note that both VS-mode and HS-mode use the *sret* to end trap handling.

3.3.1 Trap Cause Encoding

The hypervisor extension augments the trap cause encoding with virtual supervisor, and hypervisor-specific exceptions/interrupts as listed on Table 3.2 highlighted in bold. It defines VS-level related interrupts (Code 2,6 and 10) and hypervisor level guest external interrupts (code 12). As for the exceptions, it adds codes for guest page faults (exceptions 20, 21, 23) and virtual instruction trap (Code 22). Finally, it assigns different codes for environment calls coming from VS-level (cause 10) and HS-level (same as usual S-mode, i.e., cause 9). This allows them to be separately delegated so that the firmware can directly re-direct VS *ecall*s to be handle by the hypervisor.

3.4 Two-Stage Address Translation

As stated earlier, the RISC-V H-extension includes hardware support for memory virtualization by introducing two-stage address translations to the MMU, where the hypervisor has control over the page tables mapping guest-physical to host-physical addresses. Figure 3.2 presents a diagram overview of RISC-V MMU with a 2-stage translation MMU.

When the guest is active, memory accesses are subject to two stages of translation. The first stage (VS-stage) converts guest virtual address (GVA) to guest physical address (GPA) controlled by the VS-mode *vstap* registers, and the second stage (G-Stage) translates GPA to supervisor physical memory (SPA) controlled by the HS-mode *hgatp* register. This also includes translating page table entries (PTEs) accesses performed during the VS-stage of translation. Additionally, all guest memory accesses are subject to both R/W/X (read/write/execute) permissions from the VS and G stage translation.

Interrupt	Exception Code	Description
1	0	*Reserved*
1	1	Supervisor software interrupt
1	**2**	**Virtual supervisor software interrupt**
1	3	Machine software interrupt
1	4	*Reserved*
1	5	Supervisor timer interrupt
1	**6**	**Virtual supervisor timer interrupt**
1	7	Machine timer interrupt
1	8	*Reserved*
1	9	Supervisor external interrupt
1	**10**	**Virtual supervisor timer interrupt**
1	11	Machine external interrupt
1	$\geq$12	*Reserved*
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 or VU-mode
0	9	Environment call from S/HS-mode
0	**10**	**Environment call from VS-mode**
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	16-19	*Reserved*
0	**20**	**Instruction guest-page fault**
0	**21**	**Load guest-page fault**
0	**22**	**Virtual instruction**
0	**23**	**Store/AMO guest-page fault**

Table 3.2: Trap cause codes, in [Waterman et al., 2020].

Each stage involves several levels of page table walks depending on the selected scheme mode in *hgatp* and *vstap*. Currently, the ISA defines four modes targetting different virtual address space sizes: (i) *bare* metal mode - guest physical addresses are not subject translation therefore equal to supervisor physical memory, (ii) *Sv32* (RV32 only) - support 32-bit address space (RV32 only) 20-bit VPN is translated to a 22-bit using a two-level page table, (iii) *Sv39* (RV64 only) - a 27-bit VPN is translated into a 44-bit

PPN using a three-level page table, while the 12-bit page offset remains untranslated and (iv) *Sv48* (RV64 only) - a 36-bit VPN is translated into a 44-bit PPN using a three-level page table, while the 12-bit page offset remains untranslated. The same schemes were extended to the G-stage (*bare*, *Sv32x4*, *Sv39x4*, or *Sv48x4*), selected through the *hgatp*, but with a slight variation where the GPAs are widened by two bits, thus supporting larger guest physical address spaces. Moreover, the *hgatp* supports virtual machine identifiers VMIDs to tag each TLB entry with a unique number that identifies the virtual machine, thus optimizing the context switch as the hypervisor no longer needs to flush the TLB at each context switch. Consequently, this allows fences to be performed on a per-virtual-machine basis [Waterman et al., 2020].

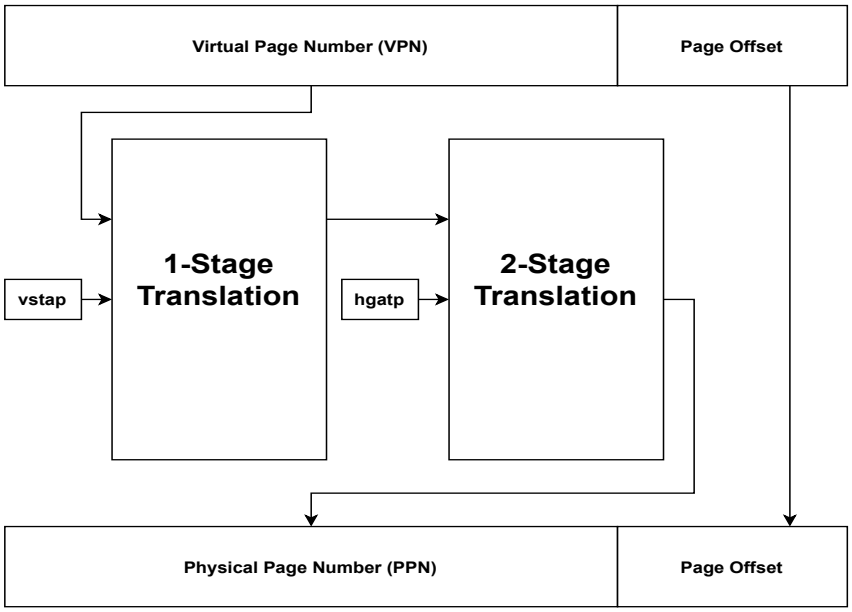


Figure 3.2: Diagram overview of MMU featuring a two stages of address translation.

The RISC-V ISA defines three types of exceptions that may arise during translation: (i) instruction access faults, (ii) load page fault, and (iii) store page fault. The H-extension extended these exceptions so that errors during VS-stage execution are reported as guest page fault exceptions. Additionally, to assist software when handling guest page fault exceptions, registers *htval/mtval2* are written, if possible, with the guest physical address, and *vstval* with the guest virtual address.

3.5 Hypervisor Instructions

The RISC-V H-extension adds two sets of hypervisor instructions to ease virtualization : (i) hypervisor virtual machine load/store instructions and (ii) hypervisor fence instructions. The hypervisor virtual machine

load/store instructions introduce a rather ingenious mechanism that allows the hypervisor to directly peek into the guest virtual address space with the same address translation and protection as a normal VS access without actually mapping it into its own address space. Furthermore, because all accesses are under the influence of the same protections as a normal VS access, it protects against confused deputy attacks when, for example, accessing indirect hypercall arguments. This capability is also available at the U-mode (by setting *hstatus.HU* bit), which further simplifies the implementation of type-2 hypervisors such as KVM [Dall and Nieh, 2014], which hosts device back-ends in userland QEMU, or microkernel-based hypervisors such as seL4 [Klein et al., 2009], which implement virtual machine monitors (VMMs) as user-space applications.

The H-extension defines for each existing normal load/store instruction there is a virtual-machine version, defined as follows: (i) *hlv.b*, *hlv.bu*- load 8-bit values from memory,(ii) *hlv.h* and *hlv.hu*- load 16-bit values, (iii) *hlv.w* and *hlv.wu* - load 32-bit values and (iv) *hlv.d* and *hlv.du* - load 64-bit values (RV64I only),(v) *hsv.b* - store 8-bit values,(vi) *hsv.h* - store 32-bit values and (vii) *hsv.h* - store 64-bit values (RV64I only). Additionally, it provides two extra load instructions (*hlvx.hu/wu*) to peek into guest memory, which is executable but possibly not readable, i.e., the memory being read must be executable in both pages independently of the read permission.

The H-extension also adds two hypervisor fence instructions, which allow the hypervisor to selectively invalidate only first (by executing *hfence.vvma*) or both (by executing *hfence.gvma*) stages of memory-management structures used during a guest translation, e.g., guest TLB entries and guest walk-cache entries.

Chapter 4

Timer Virtualization

Timekeeping is one fundamental task required by almost every modern OS. However, managing correctly the time in virtualized systems is one of the most challenging tasks. Thus time is shared between the host and possibly multiple virtual machines.

RISC-V timer specifications are restricted to the M-mode, through the *mtime* and *mtimecmp* memory-mapped registers. These are typically implemented as a part of the CLINT. *mtime* is a free-running counter and a machine timer interrupt is triggered when its value is greater than the one programmed in *mtimecmp*. There is also a read-only *time* CSR accessible to all privilege modes, which is not supposed to be implemented but converted to a memory access of *mtime* or emulated by firmware. Thus, M-mode software implementing the SBI interface (e.g., OpenSBI) must facilitate timer services to lower privileges via ecalls, by multiplexing logical timers onto the M-mode physical timer. Naturally, this mechanism introduces additional burdens and impacts the overall system performance for HS-mode and VS-Mode execution, especially in tick-driven OSes. As explained in Chapter 3, a single S-mode timer event involves several M-mode traps, i.e., first to set up the timer and then to inject the interrupt in S-mode. This issue is further aggravated in virtualized environments as it adds extra HS-mode traps. The simplest solution to mitigate this problem encompasses providing multiple independent hardware timers directly available to the lower privilege levels, HS and VS, through new registers analogous to the M-mode timer registers. This approach is followed in other well-established computing architectures. For instance, the Armv8-A architecture has separate timer registers across all privilege levels and security states.

We can conclude that RISC-V systems running without timer virtualization have two major sources of performance degradation:

- The timer interrupt and management is restricted to the M-mode. Software running on M-mode needs to virtualize timer to lower levels, following the interface specifications stated at the SBI.

- Guest OS timer management implies two traps, one to the hypervisor and another to the M-mode.

4.1 RISCV Timer Virtualization Specification

To address the aforementioned problems, we proposed a set of modifications to the current specification that could benefit the system's overall performance. Concurrently to our work, there have been some proposals discussed among the community to include dedicated timer CSRs for HS and VS modes. The latest one which is officially under consideration, at a high-level, is very similar to our implementation. However, there are differences with regard to: firstly, it does not include *stime* and *vstime* registers but only the respective timer compares; secondly, and more important, we add the new timer registers as memory-mapped IO (MMIO) in the CLINT, and not as CSRs. The rationale behind our decision is based on the fact that the RISC-V specification states that the original M-mode timer registers are memory-mapped, due to the need to share them between all harts as well as due to power and clock domain crossing concerns [Waterman et al., 2020]. As the new timers still directly depend on the original *mtime* source value, we believe its simpler to implement them as MMIO, centralizing all the timer logic. Otherwise, every hart would have to continuously be aware of the global *mtime* value, possibly through a dedicated bus. Alternatively, it would be possible to provide the new registers, as well as *htimedelta*, through the CSR interface following the same approach as the one used for *time*, i.e., by converting the CSR accesses to memory accesses. This approach would, however, in our view, add unnecessary complexity to the pipeline as supervisor software can always be informed of the platform's CLINT physical address through standard methods (e.g., device tree).

Next, we describe our timer registers specification extensions to S/HS-mode and VS-mode. We start by presenting the S-mode timer registers and by providing a full description of their purpose and how they are supposed to work. After that, we describe the VS-mode timer registers and their interaction with the hypervisor. Also, we introduce some necessary changes to an existing guest timer offset register called *htimedelta*, required by our newly introduced registers. Finally, we describe all modifications to the *VSTIP* and *STIP* bits specification on *mip*, *sip* and *hip* registers.

4.1.1 Supervisor Timer Registers (*stime* and *stimecmp*)

The *stime* is a 64-bit read/write register, responsible for holding the HS/S-mode *time* for hypervisor or OSes. The *stime* register has the same time base as the *mtime* and must increment at a constant

frequency. Also, we include a 64-bit width memory-mapped supervisor mode timer compare register, i.e., *stimecmp*. Whenever the *stime* value is greater than or equal to the *stimecmp* value, the supervisor timer interrupt becomes pending. Writing a value greater than stime to the *stimecmp* clears the interrupts. If not, the interrupt remains pending. Supervisor timer interrupts will only be taken if the *STIE* bit of the *sie* is set, and the *STIP* bit of *mideleg* is set. The *STIP* bit state is influenced by writing to the *stime* and *stimecmp* registers.

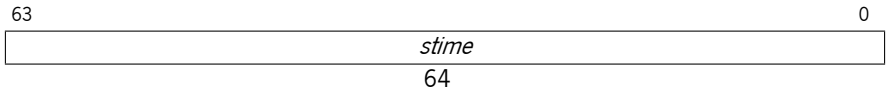


Figure 4.1: Supervisor time register (memory-mapped control register).

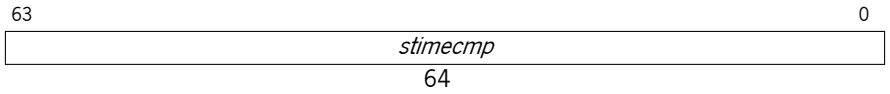


Figure 4.2: Supervisor time compare register (memory-mapped control register).

4.1.2 Virtual Supervisor Timer Registers (*vstime* and *vstimecmp*)

The *vstime* is a 64-bit memory-mapped read/write register, responsible for holding the VS-mode time for guest OSes. The *vstime* value is, as stated by the latest specification, the sum of the *mtime* with the *htimedelta* value. Also, we include a 64-bit width memory-mapped virtual supervisor mode timer compare register *vstimecmp*. Whenever the *vstime* value is greater than or equal to the *vstimecmp* value, the supervisor timer interrupt becomes pending. The interrupt remains posted until a value greater than *vstime* is written to the *vstimecmp*. Supervisor timer interrupts will only be taken if the *VSTIE* bit of the sie is set and the same *VSTIP* of *mideleg* and *hideleg* is set. As expected, writes to the *vstime* and *vstimecmp* influence the *VSTIP* state.

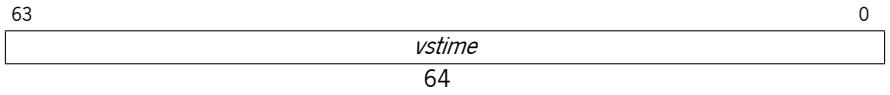


Figure 4.3: Virtual supervisor time register (memory-mapped control register).

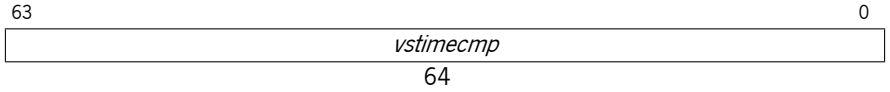


Figure 4.4: Virtual supervisor time compare register (memory-mapped control register).

4.1.3 Machine Interrupt Pending Register (*mip*)

The *mip* register holds information on pending interrupts for all privilege levels, including timer, external, and software interrupt, as stated in Chapter 3. Currently, the *STIP* bit is writable in *mip*, and the M-mode uses it to deliver timer interrupts to S-mode. By including S-mode timer registers, it allows OSes running in S-mode to access the timer directly. We propose some modifications to the description *STIP* bit in the *mip* registers to be in accordance with our newly added registers.

If the supervisor mode and supervisor timer registers (*stimecmp* and *stime*) are implemented, *STIP* is a read-only bit in *mip*, and it is set and cleared by the platform-interrupt timer controller, which must implement the supervisor timer registers. *STIP* is set and cleared by writing *stimecmp* a value less or equal, or greater than the current *stime* value.

4.1.4 Supervisor Interrupt Pending Register (*sip*)

The *sip* register holds information on pending interrupts S-mode privilege, as stated in Chapter 3. Currently, the *STIP* bit is read-only in *sip* and is set and cleared by the M-mode software implementing the SBI. By modifying the *STIP* behavior in *mip*, we also need to perform changes in the *sip*, as it represents a restricted view of *mip* register. We propose some modifications to the description *STIP* bit in the *sip* registers to be in accordance with our newly added registers.

Bits *sip.STIP* and *sie.STIE* are the interrupt-pending and interrupt-enable bits for supervisor level timer interrupts. If implemented, *STIP* is read-only in *sip*, and it can be set and cleared in two different approaches: (i) by the execution environment (when *stimecmp* and stime are not implemented), and (ii) by a platform-specific timer interrupt controller, which implements the memory-mapped S-mode timer registers that affect the timer interrupts signal (when *stimecmp* and *stime* are implemented). The *sip.STIP* bit, in response to timer interrupts generated by *stimecmp*, is set and cleared by writing *stimecmp* with a value that respectively is less than or equal to, or greater than, the current *stime* value.

4.1.5 Hypervisor Interrupt Pending Register (*hip*)

The *hip* register holds information on pending interrupts for VS-level and hypervisor-specific interrupts, as stated in chapter 3. We extend the *hip* to support the virtual supervisor timer by extending the *VSTIP*, i.e., bit *hvip.VSTIP* is the interrupt pending bit for VS-level timer interrupts. If *vstimecmp* is implemented, *VSTIP* is read-only in *hip*, and is the logical-OR of *hvip.VSTIP* and the virtual supervisor timer interrupt generated by the timer interrupt controller. Bit *hvip.VSTIP* is set and cleared by writing to the memory-mapped register *vstimecmp* with a value that respectively is less than or equal to, or greater than, the current (*mtime + htimedelta*) value.

Finally, the *mip.VSTIP* bit is an alias of *hip.VSTIP*, so it is expected these modifications also be reflected on the *mip* register.

4.1.6 Hypervisor Time Delta Register (*htimedelta*)

The *htimedelta* is a CSR that contains the difference between the actual time and time value read in VS-mode or VU-mode. The hypervisor could use this register to show both virtual time (the time the vhart was running) and wall-clock time (i.e, the physical time) by keeping the delta 0. Since *vstime* depends on *htimedelta*, we purpose some minor changes to the specification to change the *htimedelta* to a memory-mapped control and status register.

The *htimedelta* is a read/write memory-mapped control and status register containing the delta between the value of the time CSR and the value returned in VS or VU-mode. Reading the *vstime* memory-mapped CSR in VS or VU mode returns the sum of the contents of *htimedelta* and the actual value of *mtime*.

Chapter 5

PLIC Virtualization

Interrupt virtualization is one of the major sources of performance loss in virtualized systems, mainly due to its asynchronous and frequent nature. The platform-level interrupt controller (PLIC) is the RISC-V global system interrupt controller responsible for managing and delivering external interrupts to all RISC-V harts. Currently, its specification does not offer any virtualization support. Therefore, the hypervisor needs to virtualize it to the guest by trapping and emulating all guest accesses to the RISC-V interrupt controller registers. As it is commonly known, VM-exits tend to be costly, especially with high rate interrupts that cause multiple transitions to the hypervisor. In this chapter, we will discuss the problems of having a non virtualized interrupt controller and present our proposal to extend the current RISC-V PLIC architecture with virtualization support. First, we will start by overviewing the current PLIC architecture. Next, we will discuss what problems arise from not virtualizing the PLIC and, based on that, discuss the requirements to add hardware virtualization in the PLIC. Finally, we will provide a full detailed view of our PLIC virtualization extensions proposal.

5.1 PLIC Architecture Overview

The Platform-Level Interrupt Controller (PLIC) is the external interrupt controller responsible for managing and delivering all devices interrupts to one or more harts in most RISC-V systems. The PLIC is able to multiplex up to 1023 devices interrupt to the different contexts on the same hart. The PLIC contexts represent a set of control registers and associated external interrupts lines, aiming at each available external interrupt pending bit privilege level on each hart. Currently, only M-mode and S-mode are supported, as shown in figure 5.1.

The PLIC general control registers are defined as follow:

- **Interrupt priorities registers** - The interrupt priority for each interrupt source;
- **Interrupt pending registers** - The interrupt pending state for each interrupt source;
- **Interrupt enables registers** - The interrupt source enable for each context;
- **Priority thresholds registers** - The interrupt threshold of each context;
- **Interrupt claim registers** - The interrupt claim register to acquire the source interrupt ID of each context;
- **Interrupt complete registers** - The register to send interrupt completion message to the associated gateway.

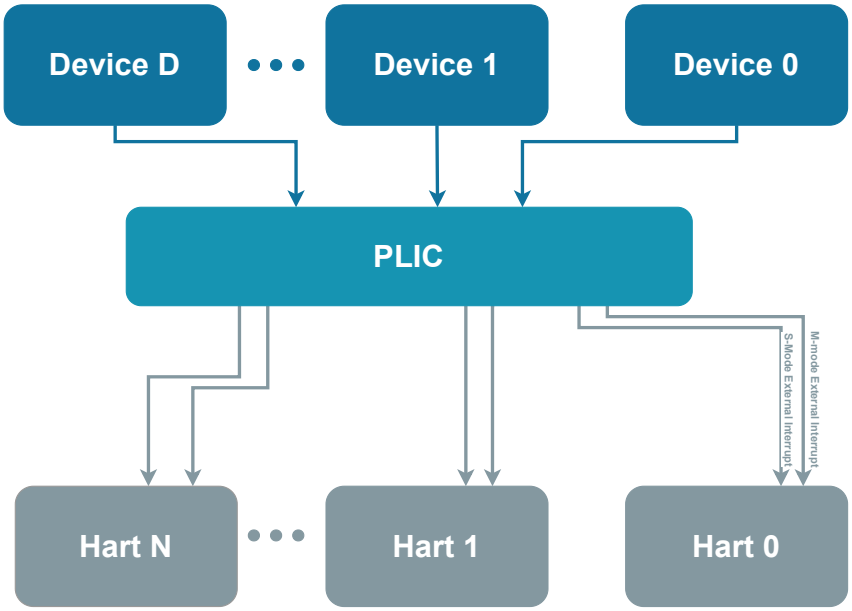


Figure 5.1: PLIC interrupt architecture block diagram.

The PLIC can be slipt among two sets of registers: (i) the interrupt *source registers*, which control the interrupt source priority and provide their current status, and (ii) the interrupt *context registers*, which controls and handles context interrupts. Figure 6.3 despites the PLIC internal operation block diagram with all aforementioned registers and their interactions, as described by the specification. Firstly, we identify two main high-level components that compose the PLIC internals structure:

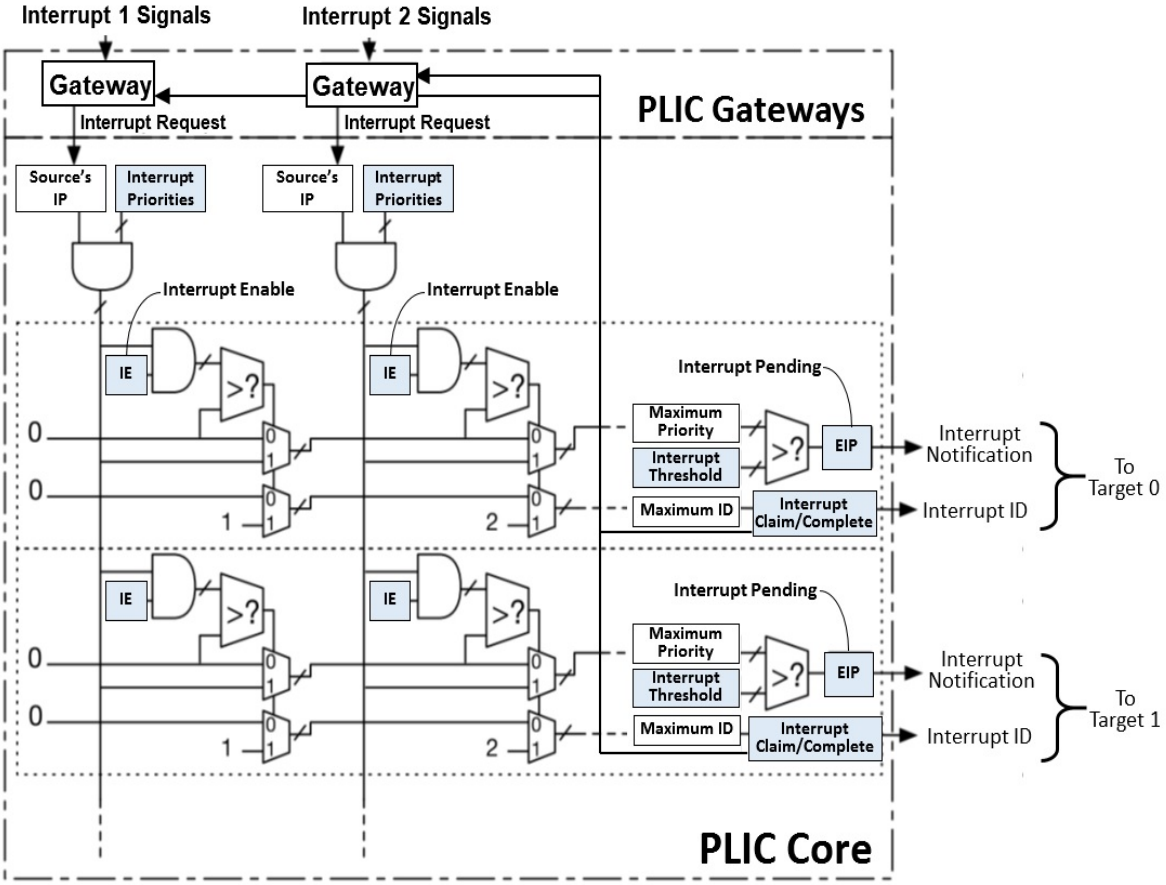


Figure 5.2: PLIC general architecture, in [Drew Barbier, 2020]

- **PLIC Gateway** - module responsible for receiving and delivering devices interrupts signals to the PLIC Core. Also, it prevents the interrupt from being asserted while the PLIC Core is servicing the interrupt;

- **PLIC Core** - the main component where all operation registers and logic are implemented. Its primary concern is to route and manage interrupts to each context external line.

Next, we discuss the internal function of each high-level module. Firstly, the Gateway makes requests to the Core whenever the device asserts an interrupt. Secondly, the Core receives the request and checks if the interrupt is enabled. If so, the Core assesses which interrupt is to deliver (1 or 2 in this case) based on their priority. If the priority is greater than the threshold, the external interrupt line is asserted, and the claim register is written with the interrupt source ID. Below, we describe each PLIC operation registers in detail, as we think it is a necessary background to propose and implement virtualization extensions adequately.

5.1.1 PLIC Interrupt Source Registers

PLIC supports a maximum of 1024 interrupts where each interrupts source is associated with a set of two control registers: the interrupt *priorities registers* and the interrupts *pending registers*. The interrupt *priorities registers* are a 32-bit memory-mapped register to store each interrupt source priority. Note that interrupt ID 0 is reserved and does not exist effectively. The priority value ranges from 0, meaning disabled, to a maximum level, which is implementation-defined. The interrupt *pending register* is read-only registers that hold the current interrupt status and are organized as a pending array of 32-bit registers. Like the *priorities registers*, interrupt ID 0 (bit 0 of word 0) is reserved as a non-existent interrupt and is hardwire 0. Each pending bit is associated with an interrupt source. So, interrupt ID N pending bit is stored in a bit (N mod 32) of the word (N/32). The pending bit directly reflects the interrupt source signals coming from the PLIC gateway.

PLIC Context Registers and Claim/Complete Process

PLIC supports up to 15872 contexts where each has a set o three control registers: interrupt *enables registers*, context *threshold registers*, and *claim/complete registers*. Each context may define the interrupt priority threshold by writing to a 32-bit memory-mapped register called the *threshold registers*. PLIC will only deliver interrupts with a priority value greater than the threshold value. By setting the value zero means allowing all interrupts to be delivered to the context. The interrupt *enables registers* allows each context to enable and disabled interrupt sources selectively. The registers are organized as a continuous array of 32-bit registers, following the same structure as the pending bits. The *claim/complete registers* act as a means of communication between the contexts and the PLIC, known as the *claim/complete process*. Upon entry on an interrupt handler, the hart needs to know which interrupt ID was asserted by using a process called a *claim*. Reading the *claim/complete register* returns the ID of the highest priority pending interrupt or zero if there is no pending interrupt. A successful *claim* will also atomically clear the corresponding pending bit on the interrupt source by blocking the gateway. The *claim* operation can be performed at any time, and the claim operation is not affected by the setting of the priority *threshold register*. When the interrupt is serviced, the hart writes back to the *claim/complete register* the same interrupt ID to *complete* the interrupt handling process.

5.1.2 Memory Map

The PLIC memory map base address is a platform implementation-specific parameter. So only the offset within the memory map is taken as reference. Table 5.1 shows how each register is organized in memory. On the first page, we have 1024 32-bit width *priority registers*, one for each interrupt. Next page, we have the pending bits organized as arrays of 32-bits, as stated before. On the third page (offset 0x2000), we have the *enable* bits for each context organized as arrays of 32-bit registers. PLIC exposes the *claim/-complete* and *threshold registers* into separate pages (4K alignment + 4) for each context, starting from offset 0x200000 and a maximum of 0x4000000, encompassing the 15872 contexts.

Field	Offset (hex)	Size (Bits)	Reset (hex)	Access Type	Description
priority *i*	0x0000000 + (*i* * 4)	1	0x0	R/W	Interrupt ID *i* priority register
pending	0x0001000	32	0x0	RO	Interrupt source pending bits. Up to 32 sources per register.
enable *c*	0x0002000 + (*c* * 0x80)	32	0x0	R/W	Interrupt source enable registers bits for context *c*. Upto 32 sources per register.
threshold *c*	0x0200000 + (*c* * 0x1000)	32	0x0	R/W	Priority threshold register for context *c*
claim/complete *c*	0x0200004 + (*c* * 0x1000)	32	0x0	R/W	Claim/Complete register for context *c*

Table 5.1: PLIC memory map.

5.2 PLIC and Virtualization problem

Currently, PLIC specification does not offer any external interrupt virtualization support, leaving the hypervisor responsible for emulating PLIC control registers accesses and managing injections into VS-mode. One major drawback of this approach is that all device interrupts must be redirected to the HS-mode context, including interrupts from guest-assigned devices. Moreover, each time an interrupt is triggered, the

hypervisor must receive the interrupt and perform calculations based on the interrupt priority to assess which interrupt to deliver and inject it by writing to the *SEIP* bit of *hvip*. Although emulating PLIC interrupt configuration registers, such as enable and priority registers, may not be a critical task as it is often a one-time-only operation performed during OS initialization, the same does not apply to *claim/complete registers*, which must be accessed before and after every interrupt handler, i.e., to read the interrupt vector number and signal interrupt completion to the controller. It is foreseeable that all these traps cause a drastic increase in interrupt latency and seriously impact overall system performance, especially in systems with real-time constraints (determinism and predictability). Based on previous statements, we can identify two major issues with a non-virtualized PLIC that could impact performance:

- Physical interrupts intended for guest context still need to be received at the hypervisor context and injected into guest context;
- *Claim/complete registers* need to be emulated. This means that at least three exceptions will occur for every interrupt: the actual interrupt directed to HS-mode and then the *claim/complete* accesses.

Once PLIC maps each context registers (*claim/complete* and *threshold*) to separate physical pages, it simplifies the process of including a new context for the VS-mode. The hypervisor can then map the context directly to the VS address space. On the other hand, we identify one problem with the PLIC memory map specification. The *enable registers* are mapped to the same page for different contexts, which breaks encapsulation, i.e., there is no way to individually block access to these registers, which allows the hypervisor to freely access at any time, even M-mode context *enable registers*.

Based on the premises above, we propose a virtualization extension for the PLIC specification that could greatly improve the system's performance. We developed the PLIC virtualization extension under the following main requirements:

- Physical interrupts direct assignment - PLIC must allow guest to interact with assigned devices directly without hypervisor intervention;
- Injection of pure virtual interrupts - PLIC must provide a mechanism to inject pure virtual interrupts into contexts;
- Minimal traps to the hypervisor, namely by removing trap-and-emulate of the *claim/complete registers*;

- Limited amount of additional registers and low complexity;

5.3 Extending PLIC with virtualization support

This section describes the PLIC virtualization extensions specification proposal. Before diving into details, one must first refer to the guest external interrupts hypervisor extension Chapter 3. The hypervisor extensions specify a guest external interrupt mechanism that allows an external interrupt controller to directly drive the VS external interrupt pending bit. This allows an interrupt to be directly forward to a virtual machine without hypervisor intervention (albeit in a hypervisor-controlled manner). To properly leverage this functionality, the PLIC must be capable of collecting and delivering virtual machine-directed interrupts separated from other interrupts. Thereby, we propose a modification to the PLIC specification to include an array of VS-Mode context to each hart. Each VS-mode context external interrupt line is connected to an associated bit in the *hgeip* registers, corresponding to a VM context running at VS-level. The maximum number of VS-context per hart is limited by the *GLEIN* (*hgeip* and *hgeie* register maximum size) minus the 0 bit, which is reserved. Therefore, we can have a maximum of 63 (for the RV64 implementation) or 31 (for the RV32 implementation) active vharts for each physical hart. Figure 5.3 illustrates a PLIC block diagram with virtualization awareness (VS- contexts external lines highlighted in blue). N is limited to the number of implemented bits in the *hgeip* register.

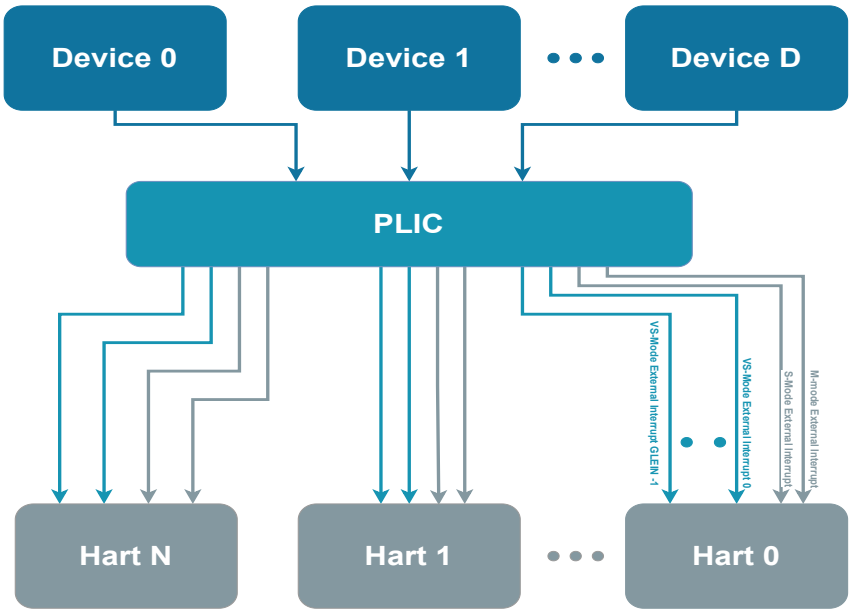


Figure 5.3: High-level virtualization-aware PLIC logic.

5.3.1 Pure Virtual Interrupt Support

By giving the guest direct control over the claim/complete registers, injection of purely virtual interrupts must also be done through the PLIC, so there are unified and consistent forwarding and handling for all the vhart's interrupts. To this end, and inspired by Arm's GIC list registers, we added two new memory-mapped 32-bit wide register sets to the PLIC to support this operation:

- *Virtual Interrupt Injection Registers (VIIR)* - register used to inject virtual interrupts.
- *Virtual Context Injection Block ID Registers (VCIBIR)* - register that holds the injection block number assigned to the context. So, there is one for each context.
- *Injection Block Management and Status Registers* (*IBMSR*) - register used to manage each *VIIR* block events, e.g., no *VIIR* pending.

When pure virtual interrupts support is implemented, the hypervisor can use the *VIIR* to inject pure virtual interrupts into a vhart. The *VIIR*s 32-bit registers are organized into separate indexed blocks attachable to each context by writing the corresponding block number to VCIBIR. When a block is attached to a context, it behaves as an additional source of interrupts, which means physical interrupts are still available to the context. In this way, virtual interrupts for multiple harts belonging to a specific VM can be injected through a single injection block, precluding the need for complex synchronization across hypervisor harts. Also, this allows a hart to directly inject an interrupt in a foreign vhart without forcing an extra HS trap. Additionaly, each virtual context block is associated with a *block management interrupt* (akin to GIC's maintenance interrupt) fed back through the PLIC itself with implementation-defined IDs. It serves to signal events related to the block's *VIIRs* lifecycle. Currently, there are two well-defined events: (i) no *VIIR* pending, and (ii) claim write of an non-present interruptID. The enabling of each type of event, signaling of currently pending events and complementary information are done through a corresponding *IBMSR*. We must point out that all of these registers are optional and may not be implemented. In that case, the hypervisor has two available options: (i) continues to inject pure virtual interrupts into the VM using a trap-and-emulate model, or (ii) a VM with only physical interrupts, which the VM controls the *claim/complete registers*.

5.3.2 Virtual Interrupt Injection Registers (*VIIR*)

The *VIIR* are 32-bit width registers used to inject virtual interrupts into any context. We grouped the *VIIR* into 240 memory-mapped blocks with 4KB (page size). So, each block can have a maximum o 1024

injection registers, illustrated in Table 5.2. Each *VIIR* register has three different fields required to inject a virtual interrupts: (i) a 10-bits *intId* field to specify the interrupt source, (ii) 1-bit *inFlight* to state the interrupt state (active or not), and (iii) 10-bits *prio* field to specify the interrupt priority. It is expected these registers are under the control of the hypervisor.

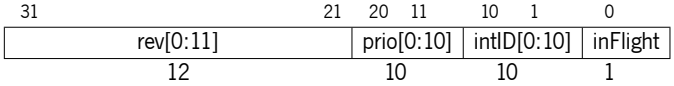


Figure 5.4: *VIIR* register layout.

A virtual interrupt is only pending when the field *intId* > 0 and the *inFlight* is not set. The bit *inFlight* is set when the virtual interrupt is pending, and a *claim* is performed, indicating that the interrupt is active and preventing the virtual interrupt from being pending. Moreover, every PLIC implementation with virtualization support must implement at least one injection register for each block. In reality, we believe that only a small amount of *VIIR*s will be required, similar to the ARM LRs.

5.3.3 Virtual Context Block ID Register (*VCIBIR*)

Each context has one register to specify the block source for virtual interrupt injection. For instance, if context 1 sets this field to 1 means block 1 of the 240 memory-mapped blocks is attached to context 1. Note that setting this field zero indicates that no injection block is attached to the context. In this way, virtual interrupts for multiple harts belonging to a specific VM can be injected through a single injection block, precluding the need for complex synchronization across hypervisor harts. Also, this allows a hart to directly inject an interrupt in a foreign vhart without forcing an extra HS trap.

5.3.4 Virtual Interrupt Claim/Complete Process

The process of claiming and completing a virtual interrupt is quite different from a physical, as now only the virtual interrupt structures, i.e., *VIIR*, will be affected. Therefore, we extended the PLIC *claim/complete process* for virtual interrupts as follow: When a *claim* is performed, the *inFlight* bit is set, indicating that the interrupt is active. When a *complete* is performed, the interrupt *intid* field is set to zero, and the *inFlight* bit is cleared.

5.3.5 Virtual Extended Mapping

To accommodate the aforementioned registers, we extend the PLIC memory map starting at offset 0x4000000. All PLIC implementations which implement the virtualization extensions must follow the offsets describe in the table 5.2.

Field	Offset (hex)	Size (Bits)	Reset (hex)	Access Type	Description
priority *i*	0x0000000 + (*i* * 4)	1	0x0	R/W	Interrupt ID *i* priority register
pending	0x0001000	32	0x0	RO	Interrupt source pending bits. Up to 32 sources per register.
enable *c*	0x0002000 + (*c* * 0x80)	32	0x0	R/W	Interrupt source enable registers bits for context *c*. Upto 32 sources per register.
threshold *c*	0x0200000 + (*c* * 0x1000)	32	0x0	R/W	Priority threshold register for context *c*
claim/complete *c*	0x0200004 + (*c* * 0x1000)	32	0x0	R/W	Claim/Complete register for context *c*
vcibir *c*	**0x4000000 + (*c* * 0x4)**	**32**	**0x0**	**R/W**	**Virtual context *c* injection block ID register**
viir *j* block *n*	**0x4010000 + (*n* * 0x1000) + (*j* * 0x4)**	**32**	**0x0**	**R/W**	**Virtual interrupt injection register *j* of block *n***
ibmsr block *n*	**0x4110000 + (*n* * 4)**	**32**	**0x0**	**R/W**	**Injection block management and status register for block *n***

Table 5.2: Extended memory map for the virtualization-aware PLIC.

Chapter 6

Extending Rocket With Virtualization Support

This chapter delves into details the implementation of the hypervisor extensions and interrupts virtualization in a RISC-V system. The H-extensions were implemented in the open-source Rocket core, a modern 5-stage, in-order, highly configurable core, part of the Rocket Chip SoC generator and written in the novel Chisel HCL. Additionally, we also extended other Rocket Chip components, namely the PLIC and the CLINT, to tackle some of the previously identified drawbacks in Chapter 3 regarding interrupt virtualization. This chapter is divided into three main topics. Firstly, we present our H-extension implementation on the Rocket core. Secondly and thirdly, we describe how we extended other Rocket Chip components with virtualization support, namely the PLIC and the CLINT.

6.1 H-extension

The bulk of our H-extension implementation in the Rocket core revolves around the CSR module, which implements most of the privilege architecture logic: exceptions triggering and delegation, mode changes, privilege instruction and CSRs, their accesses, and respective permission checks. These mechanisms were straightforward to implement, as very similar ones already exist for other privilege modes. Although most of the new CSRs and respective functionality mandated by the specification were implemented, we have left out some optional features. For example, *htinst* and *mtinst* are hardwired to zero. Nevertheless, all the mandatory H-extension features are implemented, and, therefore, our implementation is fully compliant with the RISC-V H-extension specification. Table 6.1 summarizes all the included and missing features. Despite being possible to configure this core according to the 32- or 64-bit variants of the ISA (RV32 or RV64, respectively), our implementation currently only supports the latter. The extension can be easily

enabled by adding a *WithHyp* configuration fragment in a typical Rocket Chip configuration. Listing 6.1 shows a typical Rocket Chip configuration featuring two cores with hypervisor extensions.

```
class RocketHypZCU extends Config(
  new freechips.rocketchip.subsystem.WithNExtTopInterrupts(2) ++
  new freechips.rocketchip.subsystem.WithGuestInterrupts(1) ++
  new freechips.rocketchip.subsystem.WithHyp ++
  new freechips.rocketchip.subsystem.WithNBigCores(2)  ++
  new RocketFPGAConfig
)
```

Listing 6.1: Rocket Chip configuration with hypervisor extensions

CSRs	hstatus/mstatus	●
	hideleg/hedeleg/mideleg	●
	hvip/hip/hie/mip/mie	●
	hgeip/hgeie	●
	hcounteren	◐
	htimedelta	◐
	mtval2/htval	●
	mtinst/htinst	○
	hgapt	◐
	vsstatus/vsip/vsie/vstvec/vsscratch vsepc/vscause/vstval/vsatp	●
Intructions	hlv/hlvx/hsv	●
	hfence.vvma/gvma	◐
Exceptions & Interrupts	Environment call from VS-mode	●
	Instruction/Load/Store guest-page fault	●
	Virtual instruction	●
	Virtual Supervisor sw/timer/external interrupts	●
	Supervisor guest external interrupt	●

Table 6.1: Current state of Hypervisor Extension features implemented in the Rocket core: ●fully-implemented; ◐partially implemented; ○not implemented.

6.1.1 Traps

Regarding the trap handling, some modifications to the current implementation were required to support the VS-mode execution. Listing 1 presents how we extended the trap entry at M-, HS-, and VS-mode to support virtualization. Upon a trap entry, the machine performs a context save by storing the virtualization

mode state, the program counter, and privilege encoding before traping to the target mode. Additionally, some registers are written with exception information to ease the software handlers implementation (e.g., cause code, virtual address, or guest physical address in case of page fault or guest page fault).

Algorithm 1: Trap entry implementation for M-mode, HS-mode and VS-mode.

```
if Delegated to M-mode then
    Stores the current program counter into mepc;
    Sets mcause with trap code;
    Stores the virtualization mode (mstatus.v) into mstatus.mpv and clears mstatus.v;
    Stores instruction virtual address into mtval;
    if is a guest fault exception then
        Stores instruction guest virtual address into mtval;
    else
        Sets mtval2 to 0;
    end
    Stores M-mode global interrupt enable into mstatus.mpie and disable all M-mode interrupts;
    Stores the current privilege encoding in mstatus.mpp;
    Disable all M-mode interrupts;
    Changes the privilege to M-mode;
end
if Delegated to HS-mode then
    Stores the current program counter into sepc;
    Sets scause with trap code;
    Stores the virtualization mode (mstatus.v) into mstatus.spvp;
    Stores the virtualization mode (mstatus.v) into hstatus.spv and clears mstatus.v;
    Stores instruction virtual address into stval;
    if is a guest fault exception then
        Stores instruction guest virtual address into htval;
    else
        Sets htval to 0;
    end
    Stores HS-mode global interrupt enable into mstatus.spie;
    Stores the current privilege encoding in mstatus.spp;
    Disable all HS-mode interrupts;
    Changes the privilege to HS-mode;
end
if Delegated to VS-mode then
    Stores the current program counter into vsepc;
    Sets vscause with trap code;
    Stores HS-mode global interrupt enable into mstatus.spie;
    Stores instruction virtual address into vstval;
    Stores the current privilege encoding in vsstatus.spp;
    Disable all VS-mode interrupts;
    Changes the privilege to VS-mode;
end
```

Listing 2 shows how we implemented the trap return behavior following the H-extension specification. Returns in VS-mode are performed using *sret* as in a normal S-mode execution. Our implementation simply checks if the *sret* instruction was performed by VS-mode or HS-mode and restores the machine state accordingly.

Algorithm 2: Trap return implementation.

```
if virtualization mode active and sret then
    Sets vsstats.sie with vsstatus.spie;
    Sets vsstatus.spie to true;
    Changes the privilege to vsstatus.spp;
    Sets vsstatus.spp to U-mode;
    Sets program counter to vsepc
else
    Sets mstatus.sie with mstatus.spie;
    Sets mstatus.spie to true;
    Changes the privilege to mstatus.spp;
    Sets virtualization mode to hstatus.spv;
    Sets hstatus.spv to false ;
    Sets vsstatus.spp to U-mode;
    Sets program counter to sepc;
end
if mret then
    Sets mstatus.mie with mstatus.mpie;
    Sets mstatus.mpie to true;
    Changes the privilege to mstatus.spp;
    Sets virtualization mode to mstatus.mpv;
    Sets mstatus.mpv to false;
    Sets mstatus.spp to U-mode;
    Sets program counter to mepc;
end
```

6.1.2 2nd-stage Translation

The next largest effort focused on the MMU structures, specifically the page table walker (PTW) and translation-lookaside buffer (TLB), in particular, to add to support for the 2nd-stage translation. The implementation only supports the *bare* translation mode (i.e., no translation) and the *Sv39x4*, which defines a specific page table size and topology which results in guest-physical addresses with a maximum width of 41-bits (see figure 6.1).

The modification to the PTW extends the module's state-machine so that it switches to perform 2nd-stage translation at each level of the 1st translation stage by adding a new state called *s_switch*. At each

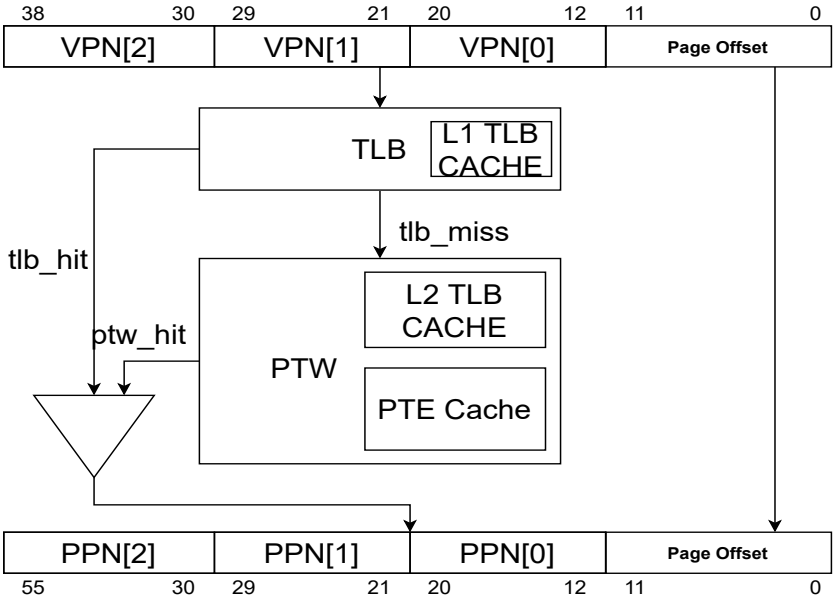


Figure 6.1: Example of a translation using mode SV39x4.

step, it merges the results of both stages. When a guest leaf PTE (page table entry) is reached, it performs a final translation of the targeted guest-physical address. This proved to be one of the trickiest mechanisms to implement, given the large number of corner cases that arise when combining different page sizes at each level and of exceptions that might occur at each step of the process. TLB entries were also extended to store both the direct guest-virtual to host-physical address as well as the resulting guest-physical address of the translation. This is needed because even for a valid cached 2-stage translation, later accesses might violate one of the RWX permissions, and the specification mandates that the guest-physical address must be reported in *htval* when the resulting exception is triggered.

Note that the implementation does not support VMID TLB entry tagging. We have decided to neglect this optional feature for two mains reasons. Firstly, at the time of this writing, the Rocket core did not even support ASIDs. Secondly, static partitioning hypervisors (our main use case) do not use it at all. A different hypervisor must invalidate these structures at each context-switch. As such, the implemented support for *hfence* instructions ignores the VMID argument. Furthermore, they invalidate all cached TLB or walk-cache entries used in guest translation, despite it specifying a virtual address argument or being targeted at only the first stage (*hfence.hvma*) or both stages (*hfence.gvma*). To this end, an extra bit was added to TLB entries to differentiate between the hypervisor and virtual-supervisor translations. Finally, we have not implemented any optimizations such as dedicated 2nd-stage TLBs as many modern comparable processors do, which still leaves room for important optimizations.

6.2 Timer Virtualization

This subsection describes the CLINT timer virtualization implementation on the Rocket Chip. First, we will describe the CLINT overall micro-architecture with virtualization support. Next, we will present the CLINT extended map with all timer virtualization control registers. Finally, we will present all the implementation details accompanied by code written in Chisel.

6.2.1 CLINT Virtualization Micro-Architecture

On the Rocket Core, as illustrated by figure 2.4, timer support is implemented as part of the core local interrupter (CLINT) responsible for maintaining memory-mapped control and status registers handling the timer and inter-processor communications interrupts.

Originally, the CLINT only supported the machine mode timer interrupts. We have extended it support for both HS-mode and VS-mode following the specification defined in Chapter 4. Figure 6.2 shows the CLINT micro-architecture block diagram for one hart. Our architecture simply defines all S/HS- and VS mode timer registers (*stime, stimecmp, vstime, vstimecmp and htimedelta*) as memory-mapped registers into the CLINT device. Moreover, we add extra logic with two extra comparators for both HS-mode and VS-mode, each associated to a corresponding timer interrupt, i.e, supervisor timer interrupt (*STIP*) and virtual supervisor timer interrupt (*VSTIP*). When the value of *stime* or *vstime* is greater than *stimecmp* or *vstimecmp*, respectively, an interrupt is triggered. The value of *vstime* is, as defined by the specification, the sum of *htimedelta* with *mtime*.

To propagate the extended timer interrupts from the CLINT to each hart, we augmented the CLINT IO with two extra signals, as depicted on table 6.2 highlighted in bold, one for each extra privileged mode.

Signal Name	Size	Direction	Description
clock	1	input	Input clock tick
msip	1	output	Software interrupt pending bit. Drive the MSIP bit in mip CSR of each core.
mtip	1	output	Machine mode timer interrupt pending bit. Drive the MTIP bit in mip CSR of each core.
stip	**1**	**output**	**Supervisor mode timer interrupt pending bit. Drive the STIP bit in mip CSR of each core.**
vstip	**1**	**output**	**Virtual supervisor mode timer interrupt pending bit. Drive the VSTIP bit in hip CSR of each core.**

Table 6.2: Extended CLINT IO signals.

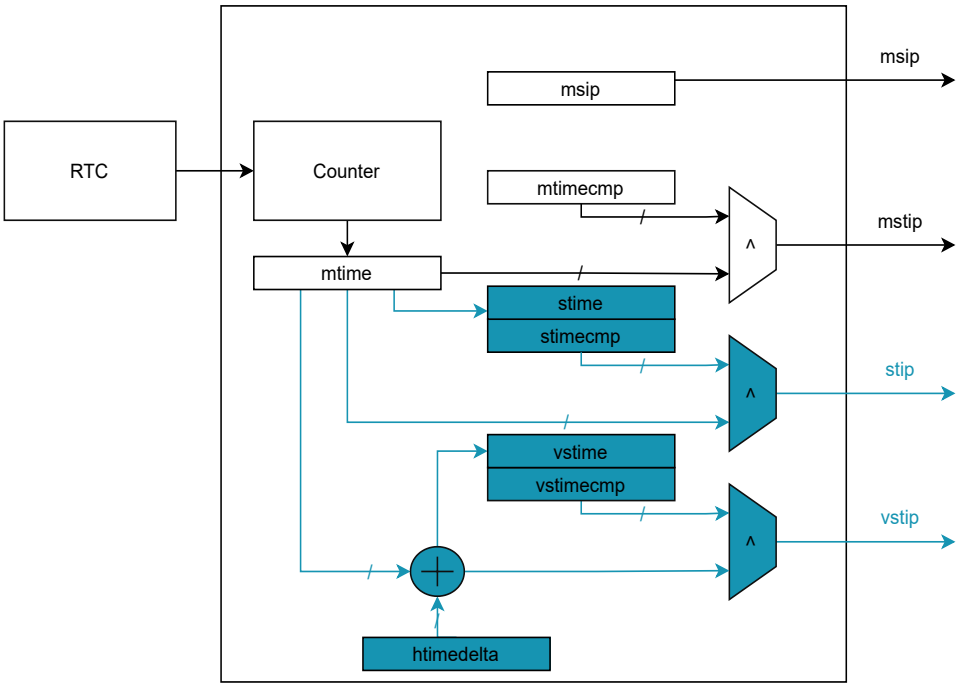


Figure 6.2: CLINT micro-architecture with virtualization support.

6.2.2 CLINT Extended Memory-Mapped

The CLINT holds three memory-mapped registers (shown in Table 6.3 not in bold). One *msip* register for each hart located at 0x0 offset with maximum size 0x4000, one *mtime* register to hold the current time value with 0xBFF8 offset, and finally one *mtimecmp* for each hart at located at 0x4000 with maximum 0x4000 size. We have extended the CLINT register map to include the previously defined five extra registers: *stime, vstime, stimecmp, vstimecmp, and htimedelta*. Table 6.3 shows the CLINT extended memory layout, with the new registers highlighted in bold. The *stime* register is located immediately after the last *mtimecmp* register at 0x1bff8 offset. The *stimecmp* follows the same layout as the *mtimecmp* but located at an offset of 0xc000. As for VS-level timer registers, both *vstime* and *vstimecmp* have the same maximum number of registers as we need a pair for each hart and are located at 0x14000 and 0x1c000, respectively. Once, *vstime* is equal to $mtime + htimedelta$, we included *htimedelta* immediately after *vstimecmp* at 0x24000 offset. Note that each type of timer registers is mapped onto separate pages. However, hart replicas of the same register are packed contiguously on the same page. As such, with this approach, the hypervisor still needs to mediate VS- register access as it cannot isolate VS registers of each individual virtual hart (or vhart) using virtual memory. Nevertheless, traps from HS- to M-mode are no longer required, and when the HS and VS timer expires, the interrupt pending bit of the respective privilege level is directly set.

Field	Offset (hex)	Size (Bits)	Reset (hex)	Access Type	Description
msip *n*	0x00000 + (*n* * 4)	1	0x0	R/W	M-mode hart *n* software interrupt.
mtimecmp *n*	0x04000 + (*n* * 8)	64	0x0	R/W	Compare value for the M-mode hart *n* timer.
mtime	0x0BFF8	64	0x0	R/W	Current time. value
stimecmp *n*	**0x0c000 + (*n* * 8)**	**64**	**0x0**	**R/W**	**Compare value for the S or HS-mode hart *n* timer.**
stime	**0x1BFF8**	**64**	**0x0**	**RO**	**Current time value for S-mode (RO replica of mtime).**
vstimecmp *n*	**0x1c000 + (*n* * 8)**	**64**	**0x0**	**R/W**	**Compare value for the VS-mode hart *n* timer.**
vstime *n*	**0x14000 + (*n* * 8)**	**64**	**0x0**	**RO**	**Current time value for the VS-mode hart *n* (mtime + htimedelta *n*).**
htimedelta *n*	**0x24000 + (*n* * 8)**	**64**	**0x0**	**R/W**	**Holds the current time delta between the value of the time CSR and the value returned in VS-mode hart *n*.**

Table 6.3: CLINT extended memory map.

6.2.3 Implementation

Regarding the implementation, all modifications were performed mostly to CLINT module and some minor changes to the *InterruptBus*, and CSR module to connect *stip* and *vstip* interrupts from the CLINT to each hart. We started by adding all memory-mapped *stime, stimecmp, vstime, vstimecmp* and *htimedelta* as described on the table 6.3. The CLINT is connected to the *ControlBus* using the standard Tilelink interface. To add new memory registers to a device, Rocket Chip provides a Chisel *Regmap* interface, where we can easily associate memory-mapped registers to a device by merely supplying a mapped list of registers with two objects: (i) base address and (ii) structure called *RegField* where registers information

are hold (width, description, and attributes). Furthermore, we also include *stip* and *vstip* logic to generate interrupts described earlier, shown in Listing 6.2.

```
1     val (intnode_out, _) = intnode.out.unzip
2     intnode_out.zipWithIndex.foreach { case (int, i) =>
3       int(0) := ShiftRegister(ipi(i)(0), params.intStages) // msip
4       int(1) := ShiftRegister(time.asUInt >= timecmp(i).asUInt, params.
      intStages) // mtip
5       int(2) := ShiftRegister(stime.asUInt >= stimecmp(i).asUInt, params.
      intStages) // stip
6       int(3) := ShiftRegister(vstime(i).asUInt >= vstimecmp(i).asUInt,
      params.intStages) // vstip
7     }
```

Listing 6.2: CLINT timer logic implementation.

Listing 6.3 shows modifications to the CSR module. All interrupts are implemented in the *mip* register. Both sip and hip are only restricted views of *mip*. So, we assign *mip.stip* and *mip.vstip* bits as a logical or between the writable *stip* and *vstip* bits and the lines comming from the CLINT module.

```
1   io.interrupts.stip.foreach { mip.stip := reg_mip.stip || _ }
2   io.interrupts.vstip.foreach { mip.vstip := reg_mip.vstip || _ }
```

Listing 6.3: CSR module S/HS-mode and VS-mode timer interrupts implementation.

6.3 PLIC Virtualization

This subsection presents the implementation of the virtualization extension in the Rocket Chip PLIC. First, we start by giving a little background on the current PLIC implementation in the Rocket Chip. Next, we provide a micro-architecture diagram with our PLIC with virtualization support, describe in Chapter 5. Thirdly, we describe some of the components that comprise the architecture and explain how we have implemented them.

6.3.1 Rocket Chip PLIC Module Background

The Rocket Chip is already packed with PLIC implementation compliant with the current specification. When diving into the PLIC module, we were able to distinguish three major submodules (see Figure 6.3): (i) Gateway module, (ii) Claim/Complete logic block, and (iii) PLICFanIn module.

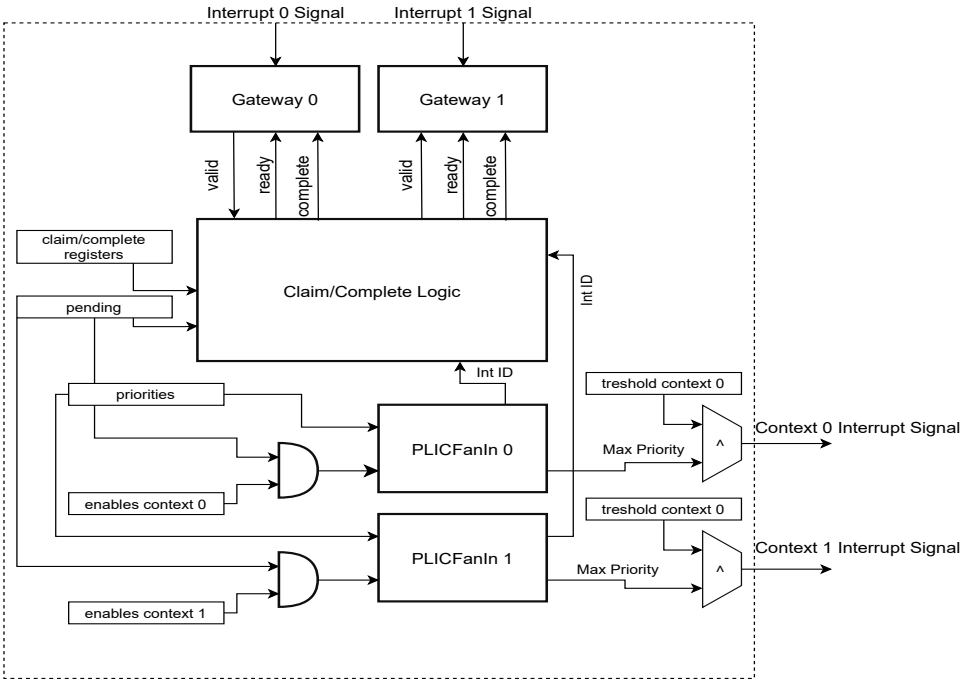


Figure 6.3: PLIC micro-architecture.

The Gateway serves as the interface between the device interrupts and the PLIC main logic. When a device asserts an interrupt, the Gateway issues a request for handling to the Claim/Complete logic block using a *ready/valid* interface and an extra *complete* signal. The interrupt only becomes pending when the Claim/Complete logic block is ready to receive another interrupt, and the gateway interrupt line is valid. When these conditions are met, the Claim/Complete logic block asserts the corresponding interrupt pending bit and Gateway blocks until a complete is signaled. Finally, there is one PLICFanIn module for each attached context. This module acts as the center of decision and defines which interrupts to deliver to each context, based on the priority, pending state, and context interrupts enables. Additionally, interrupts are only delivered to the context if not masked by the context *threshold register*.

6.3.2 Micro-Architecture

The majority of modifications were performed at the PLICFanIn and Claim/Complete logic block. Figure 6.4 illustrates the PLIC general architecture with virtualization support. We refactored the PLICFanIn module to handle both virtual and physical interrupts (highlighted in blue in figure 6.4), and additionally, we modified the Claim/Complete logic block to handle virtual interrupts (see section 6.3.7).

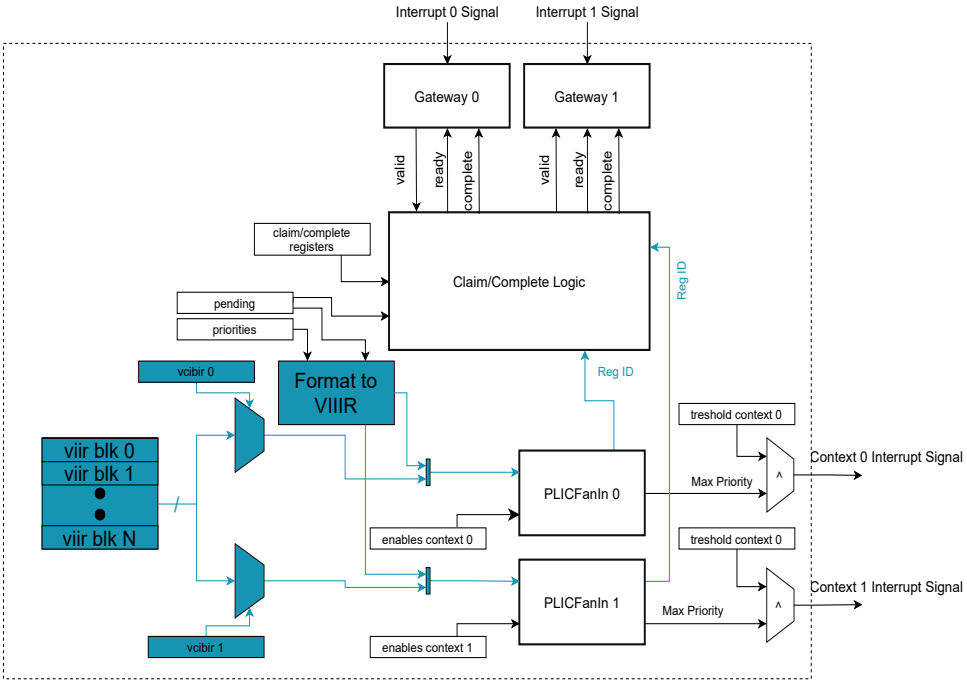


Figure 6.4: PLIC micro-architecture with virtualization support.

6.3.3 Configurations Options

The Rocket Chip PLIC generator is packed with multiple configuration options that allow us to customize it fully. For example, the PLIC base address in memory, the maximum number of contexts, and device interrupt lines. We extended the PLIC parameters so that we could easily customize our virtualized PLIC with, for example, the number of injection blocks available in our design or the number of injection registers implemented per block. Table 6.4 depicts all PLIC configurations parameters with our custom ones highlighted in bold.

Configuration	**Description**
Base address	The base address where the memory map registers begin.
maxVirtualInjectionRegsPerBlk	**Defines the number of implemented VIIR per implemented ranging from 0 to 1024.**
nVirtBlks	**Defines the number of injection blocks implemented ranging from 0 to 240.**

Table 6.4: PLIC available configuration options extended with virtual interrupts.

Furthermore, we also added a new parameter to the Rocket core so that we could easily select the size of the VS-context array per block. Associated with this parameter, we created a configuration, denoted

nGuestInterrupts, so that we could effortlessly enable/disable it in the Rocket design configuration. The *nGuestInterrupts* is the implementation of the GLEIN stated in the specification. By default, in Rocket Chip, the PLIC starts at address 0xc000000. However, we can easily change it by modifying the base address configuration option.

6.3.4 VS-Mode Contexts

The PLIC uses the Diplomacy framework for negotiating configuration parameters, e.g., the number of interrupts lines. So, by adding a vector of guest interrupts to each Rocket Tile (see Listing 6.4) and add a new interrupt connection from Rocket Tile to the PLIC, it automatically assesses the number of contexts and creates the registers accordingly. To increase versatility, the number of guest interrupts per core (*GLEIN*) is defined by the previously stated *nGuestInterrupts* parameter.

```
1  class TileInterrupts(implicit p: Parameters) extends CoreBundle()(p) {
2    val debug = Bool()
3    val mtip = Bool()
4    val stip = usingVM.option(Bool())
5    val vstip = usingHype.option(Bool())
6    val msip = Bool()
7    val meip = Bool()
8    val seip = usingSupervisor.option(Bool())
9    val geip = Vec(coreParams.nGuestInterrupts, Bool())
10   val lip = Vec(coreParams.nLocalInterrupts, Bool())
11 }
```

Listing 6.4: Tile Interrupt definition endowed with VS-context external interrupts.

On the CSR module, we drive each hart VS-context interrupt line to *hgeip* bit, starting at bit 1 (bit 0 is hardwired to 0).

6.3.5 Virtual Interrupt Registers

To ease implementation and provide a more readable code, we divided the *VIIR* registers definitions into two chisel bundles (see Listing 6.5). One named *InjRegBlock* defining a block of injection registers and two methods to obtain all pending and *inFlight*, and another defining an injection register, denoted *InjReg*.

We also define a method to assess if a virtual interrupt is pending (*is_pend*) by checking if the interrupt ID different from 0 and the interrupt is not active.

```
1  class InjRegBlock extends Bundle {
2    val injection_regs = Vec(PLICConsts.maxVirtualInjectionRegsPerBlk,new
       InjReg)
3
4    def pend_bits() : UInt = {
5        Reverse(injection_regs.zipWithIndex.map{case (inj,i) => inj.pend_bit
       ()}.reduce(_| _).asUInt)
6    }
7    def inFLight_bits() : UInt = {
8        Reverse(injection_regs.zipWithIndex.map{case (inj,i) => inj.inFlight
       }.reduce(_| _).asUInt)
9    }
10 }
11
12 class InjReg extends Bundle {
13   val rev = UInt(width = 10)
14   val prio = UInt(width = 10)
15   val int_id = UInt(width = 10)
16   val inFlight = Bool()
17   def pend_bit () : UInt = (is_pend() << int_id).asUInt
18   def is_pend() : Bool = (int_id > 0 && !inFlight).asBool
19 }
```

Listing 6.5: Injection Blocks and VIIR definition in Chisel.

6.3.6 PLICFanIn

As stated earlier, the PLIC attaches a PLICFanIn module to handle physical interrupt delivery to each context. By including support for virtual interrupts injection in each context, a few modifications were required at PLICFanIn so that both virtual interrupts and physical could coexist (see Figure 6.4). Initially, physical and virtual interrupts were packed into two very distinct formats, one in injection registers and the other as arrays of bits (*pending registers*) and 32-bits (*priority registers*). To ease implementation, we decided that first, we needed to process all interrupts in a unique format. In that case, we opted to convert physical interrupts into block injection registers and concatenate them with virtual injection blocks before driving

into the PLICFanIn module. Moreover, we modified the PLICFanIn internal logic to perform a search on the input injection registers block (as shown in Listing 6.6) and return the register index that causes the interrupt, a field required to handle virtual interrupts in the Claim/Complete logic block.

```
class PLICFanInVirtualContxt(nDevices: Int, nInjRegs: Int, prioBits: Int)
    extends Module {
  val io = new Bundle {
    val inj_blk = Vec(nInjRegs,new InjReg).flip
    val enbl    = UInt(width = nInjRegs).flip
    val dev     = UInt(width = log2Ceil(nDevices+1))
    val reg_id  = UInt(width = log2Ceil(nInjRegs+1))
    val max     = UInt(width = prioBits)
  }

  def findMax(x: Seq[UInt]): (UInt, UInt) = {
    if (x.length > 1) {
      val half = 1 << (log2Ceil(x.length) - 1)
      val left = findMax(x take half)
      val right = findMax(x drop half)
      MuxT(left._1 >= right._1, left, (right._1, UInt(half) | right._2))
    } else (x.head, UInt(0))
  }

  val effectivePriority = (UInt(1) << 9) +: (io.enbl.asBools zip io.inj_blk
    ).map { case (enbl, inj)=> Cat(inj.is_pend() & enbl, inj.prio) }
  val (maxPri, maxDev) = findMax(effectivePriority)
  io.max := maxPri // strips the always-constant high '1' bit
  io.dev := Mux(maxDev === 0.U, 0.U, io.inj_blk(maxDev - 1.U).int_id)
  io.reg_id :=  Mux(maxDev === 0.U, 0.U, maxDev - 1.U)
}
```

Listing 6.6: PLICFanIn implementation in Chisel.

6.3.7 Claim/Complete Process

PLIC specification has a well-defined protocol to handle device interrupts at each context, also denoted as the *claim/complete process*. Although the original PLIC implementation already provided such a mechanism, it is only ready to handle physical interrupts. Therefore, with the addition of virtual interrupts, the *claim/complete process* needed to be reformulated to handle both physical interrupts and virtual interrupts as described by the specification. Given that, we implemented *claim/complete process* as depicted on Listing 3 and 4.

Algorithm 3: Claim process implementation.

```
if context claimed interrupt and is virtual interrupt then
    sets injection register field int_id = 0;
    sets injection register field inFlight = false;
else
    if context claimed interrupt and is a physical interrupt then
        if physical interrupt is pending then
            sets the pending bit to true;
            blocks Gateway interrupt input line;
        end
        if device claimed then
            clears pending bit;
            releases Gateway interrupt input line;
        end
    end
end
```

According to our specification, when claiming virtual interrupts, we must clear the interrupt id field and set the interrupt as active by writing to *inFlight*. We extended the PLIC Claim/Complete logic block so that when the interrupt handler acquires the interrupt ID, an internal flag is set, indicating that claimed was performed. First, it checks whether or not the interrupt is virtual. If so, it clears the *int_id* field and sets the interrupts to active. Otherwise, it is a physical interrupt, and the pending bit is cleared, and Gateway is locked until a complete is performed. The complete algorithm also needs to handle virtual interrupts. Originally, the complete process only unlocked the Gateway allowing new interrupts requests to be handled by the PLIC. With support for virtual interrupts, when a write is performed to the complete register, an internal flag is set, and the complete process begins. First, it must check if the complete is targeted to a virtual interrupt or not. If so, the virtual interrupt handler is finished, and the *inFLight* bit is cleared, freeing the injection register to inject a new virtual interrupt. If not, it behaves like a normal

physical interrupt *complete* and frees the Gateway to trigger new interrupts.

Algorithm 4: Complete process implementation.

```
if context completed and is virtual interrupt then
    sets injection register field int_id = 0;
    sets injection register field inFlight = false;
else
    sends a complete signal to Gateway;
end
```

Chapter 7

Evaluation

In this chapter, we describe and discuss the tests and experiments performed on the Rocket Chip extended with virtualization support. All experiments were mainly conducted under six-core Rocket Chip Soc with per-core 16 KiB L1 data and instruction caches and a shared unified 512 KiB L2 LLC (last-level cache). The software stack encompasses the OpenSBI (version 0.9), Bao (version 0.1), and Linux (version 5.9), and bare metal VMs. OpenSBI, Bao, and bare metal VMs were compiled using the GNU RISC-V Toolchain (version 8.3.0 2020.04.0), with -O2 optimizations. Linux was compiled using the GNU RISC-V Linux Toolchain (version 9.3.0 2020.02-2). Our evaluation focused on: functional verification (Section 7.1), hardware resource (Section 7.2), performance and inter-VM interference (Section 7.3), and interrupt latency (Section 7.2).

7.1 Functional Verification

The functional verification of our hardware was performed on a Verilator-generated simulator and on a Zynq UltraScale+ MPSoC ZCU104 FPGA.

7.1.1 Zynq UltraScale+ MPSoC ZCU104 FPGA Setup

To run tests on the ZCU104, we develop a custom Rocket Chip top Verilog module with all required connections: serial interrupts, memory AXI connections, clock, and system reset. Our design uses 2 AXI Interconnects for MMIO operations (UART0 and UART1 peripherals) and memory accesses (BRAM). Additionally, we connected both UART0 and UART1 interrupts directly to the Rocket Chip. Figure 7.1 illustrates the final FPGA design used during our tests.

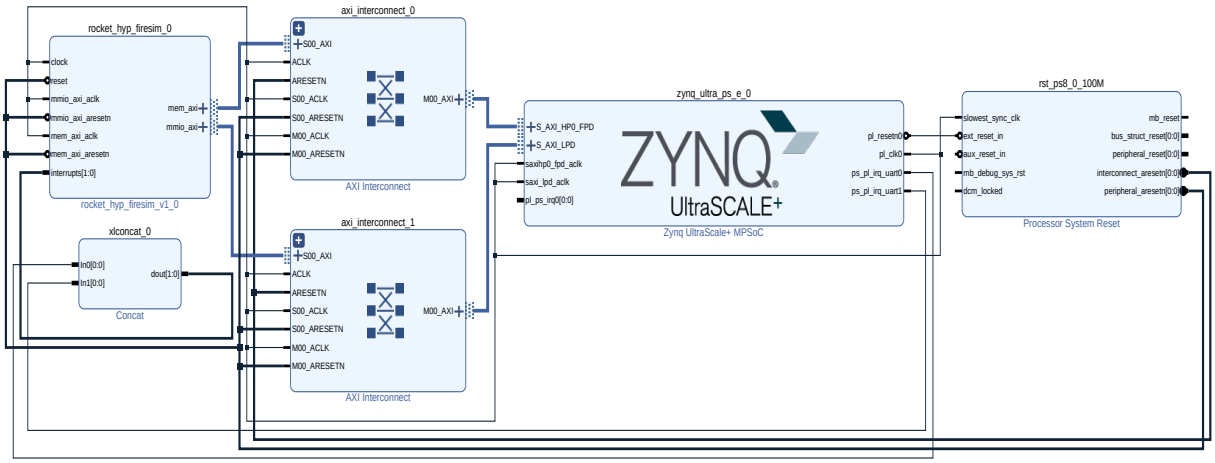


Figure 7.1: Zynq UltraScale+ MPSoCZCU104 FPGA Rocket Chip design.

7.1.2 Testing Framework

We develop an ad-hoc testing framework as a bare-metal application. Our goal was to test individual features of the hypervisor specification without any additional software complexity and following a test-driven development (TDD). The testing framework offers a simple API with four useful features for debugging and testing:

- Possibility to fully resetting processor state at the beginning of each test unit;
- Fluidly and transparently changing across privilege modes;
- Easy access a virtual guest address with any combination of 1st and 2nd stage permissions;
- Easy detection and recovery of exceptions with a full log of its state and causes.

At the time of development, we have written a comprehensive set of tests suites targetting various features such as interrupts delegation, exception delegation, virtual exceptions, two-stage translation, hypervisor load-store instructions, VS and HS-mode CSRs read/write access, and PLIC and CLINT virtualization extensions, all despited in table 7.1. These tests also serve as regression tests when a new feature was added and something went sideways. Nevertheless, the framework still has some limitations, such as not allowing user-mode execution or experimenting with superpages. This hypervisor extension testing framework and accompanying tests suite are openly available and can be easily adapted to other platforms.

Test Name Unit	**Description**
interrupt_tests	Tests *VSSI* injection with and without delegation
check_xip_regs	Tests VS-contexts interrupts correct behaviour by setting/clearing *mip* and *hvip* registers
hfence_test	Tests *hfence* intructions (*hfence.gvma* and *hfence.vvma*) and *sfence* at VS and HS-level, by checking if the TLB was correctly flushed (only hypervisor entries or only guest entries).
two_stage_translation	Tests to the two stage translation unit.
second_stage_only_translation	Tests to the two stage translation unit without first stage.
m_and_hs_using_vs_access	Tests to the hypervisor load/store instructions to the guest space.
virtual_instruction	Tests all possibly scenarios that trigger a virtual instruction when running in VS-mode (*sret* when *VTSR* is set, *wfi* when *VTW* is set, and others)
plic_registers	Tests all PLIC registers acess (read/write)
plic_virtual_injection	Tests external interrupt virtual injection at the VS-context and to a non-active guest to HS-mode.
plic_direct_inj	Tests external interrupt direct-passthrough at the VS-context and to a non-active guest to HS-mode.

Table 7.1: List of all available test units.

7.1.3 Hypervisor Validation

As a second step to reinforce our functional validation, we have successfully run two open-source hypervisors that support H-extensions: our Bao and the XVisor. Additionally, XVisor provides a "Nested MMU Test-suite", which mainly exercises the two-stage translation and allows us to test our 2nd stage translation unit extensively. At the time of this writing, our implementation fully passed this test suite. Some bugs uncovered while running these hypervisors were translated into tests and incorporated into our test suite. In particular, Bao was mapping incorrectly the PLIC guest context into PLIC physical context when the guest was running on a multi-core configuration.

7.2 Hardware Overhead

To assess the hardware overhead, we synthesized multiple SoC configurations with an increasing number of harts (2, 4, and 6). We used Vivado 2018.3 targetting the Zynq UltraScale+ MPSoC ZCU104 FPGA. Table 7.2 presents the post-synthesis results, depicting the number of look-up tables (LUTs) and registers for the three SoC configurations. We focus our evaluation on three main components: Rocket Cores, CLINT, and PLIC, as illustrated in table 7.2. Each is reflecting the impact of the H-extensions, timer virtualization, and

PLIC virtualization, respectively. For each cell, there is the absolute value for the target configuration and, in bold, the relative increment (percentage) compared to the same configuration without the hypervisor extensions. We withhold data on other resource usages (e.g., BRAMs or DSPs) as they were irrelevant and almost non-existing.

		Dual-Core	**Quad-Core**	**Six-Core**
Rocket Cores	LUTs	$^{50922}/_{\textbf{11\%}}$	$^{101744}/_{\textbf{12\%}}$	$^{152957}/_{\textbf{12\%}}$
	Regs	$^{25086}/_{\textbf{30\%}}$	$^{50172}/_{\textbf{30\%}}$	$^{75258}/_{\textbf{30\%}}$
CLINT	LUTs	$^{68}/_{\textbf{375\%}}$	$^{196}/_{\textbf{296\%}}$	$^{269}/_{\textbf{373\%}}$
	Regs	$^{194}/_{\textbf{297\%}}$	$^{324}/_{\textbf{336\%}}$	$^{454}/_{\textbf{277\%}}$
PLIC	LUTs	$^{90}/_{\textbf{140\%}}$	$^{144}/_{\textbf{236\%}}$	$^{220}/_{\textbf{263\%}}$
	Regs	$^{83}/_{\textbf{325\%}}$	$^{116}/_{\textbf{412\%}}$	$^{149}/_{\textbf{460\%}}$
Others	LUTs	$^{11207}/_{\textbf{2\%}}$	$^{13242}/_{\textbf{3\%}}$	$^{91821}/_{\textbf{0,5\%}}$
	Regs	$^{4257}/_{\textbf{0,1\%}}$	$^{4628}/_{\textbf{0,2\%}}$	$^{4728}/_{\textbf{2\%}}$
Total	LUTs	$^{62287}/_{\textbf{11\%}}$	$^{115356}/_{\textbf{11\%}}$	$^{167753}/_{\textbf{11\%}}$
	Regs	$^{29620}/_{\textbf{27\%}}$	$^{55250}/_{\textbf{28\%}}$	$^{80589}/_{\textbf{29\%}}$

Table 7.2: Rocket Chip hardware resource overhead with virtualization extensions.

According to Table 7.2, we can draw five key conclusions. Firstly, there is an overall non-negligible cost to implement the hypervisor extensions support: an extra 11% LUTs, and 27-29% registers. A deeper analysis shows that this overhead comes almost exclusively from two sources: the CSR and TLB modules. The CSR increase is explained by the amount of HS and VS registers stated by the H-extension specification. The increase in the TLB is mainly due to the widening of the data store to hold guest-physical addresses (see Chapter 6) and the extra privilege-level and permission match and check complexity. Secondly, there increase of hardware usage in the CLINT module is explained by the number of registers required by the HS-mode (*stime* and *stimecmp* per hart) and by VS-level (one *vstime*, *vstimecmp* and *htimedelta* per hart). Thirdly, the PLIC virtualization increase is explained by the additional context and injection block registers on each hart and the hardware complexity added to support virtual interrupt injection on each context. Fourthly, although the enhancements to the CLINT and PLIC reflect a large relative overhead, as these components are simple and small compared to the overall SoC infrastructure, there is no significant impact on the total hardware resources cost. Lastly, we can also conclude that increasing the number of available harts in the SoC does not impact the relative hardware costs.

7.3 Performance and Inter-VM Interference

To assess performance overhead and inter-hart / inter-VM interference, we select the MiBench Embedded Benchmark Suite. MiBench comprises a set of 35 benchmarks divided into six suites, each one targeting a specific area of the embedded market (automotive, consumer devices, office automation, networking, security, and telecommunications). We focus our evaluation on the automotive as it is one of our Bao's primary targets subset. The automotive suite includes three high memory-intensive benchmarks, i.e., more susceptible to interference due to LLC and memory contention (qsort, susan corners, and susan edges).

The experiments were conducted in Firesim, an FPGA-accelerated cycle-accurate simulator, deployed on an AWS EC2 F1 instance, running with a 3.2 GHz simulation clock. Each benchmark was ran for seven different system configurations targeting a six-core design:

- **bare** - guest native execution;
- **solo** - hosted execution running in one core;
- **solo-col** - hosted execution with cache coloring for VMs;
- **solo-hypcol**- hosted execution with cache coloring for VMs and the hypervisor;
- **interf** - hosted execution under interference from multiple colocated VMs;
- **interf-hypcol** - hosted execution under interference with cache coloring for VMs and the hypervisor.

Hosted scenarios with cache partitioning aim at evaluating the effects of partitioning micro-architectural resources at the VM and hypervisor level and to what extent it can mitigate interference. We execute the target benchmark in a Linux-based VM running in one core, and we add interference by running one VM pinned to five harts, each running a bare-metal application. Each hart runs an ad-hoc bare-metal application that continuously writes and reads a 1 MiB array with a stride equal to the cache line size (64 bytes). The platform's cache topology allows for 16 colors, each color consisting of 32 KiB. When enabling coloring, we assign seven colors (224 KiB) to each VM. The remaining two colors were reserved for the hypervisor coloring case scenario.

Fig. 7.2 presents the results as performance normalized to bare execution, meaning that higher values translate to worse results. Each bar provides the average value of 100 samples and the standard deviation.

For each benchmark, we added the execution time (i.e., absolute performance) at the top of the bare-metal execution bar.

According to Fig. 7.2, we can draw six main conclusions. Firstly, hosted execution (solo) causes a marginal decrease of performance (i.e., average 1% overhead increase) due to the virtualization overheads of 2-stage address translation. Secondly, when coloring (solo-col and solo-hypcol) is enabled, the performance overhead is further increased. This extra overhead is explained by the fact that only about half of the L2 cache is available for the target VM, and that coloring precludes the use of superpages, significantly increasing TLB pressure. Thirdly, when the system is under significant interference (inter), 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 62%. Fourthly, we can observe that cache coloring can reduce the interference (inter-col and inter-hypcol) by almost 50%, with a slight advantage when the hypervisor is also colored. Fifthly, we can observe that the cache coloring, per se, is not a magic bullet for interference. Although the interference is reduced, it is not completely mitigated, because the performance overhead for the colored configurations under interference (inter-col and inter-hypcol) is different from the ones without interference (solo-col and solo-hypcol). Finally, we observe that the less memory-intensive benchmarks (i.e., basicmath and bitcount) are less vulnerable to cache interference and that benchmarks handling smaller datasets are more susceptible to interference.

Curiously, the achieved results for RISC-V on this dissertation share a similar pattern to the ones assessed for ARM [Martins et al., 2020]. In our previous work [Martins et al., 2020], we have deeply investigated the micro-architectural events using a performance monitoring unit (PMU), which proves that shared cache and memory is one major bottleneck of virtualized systems.

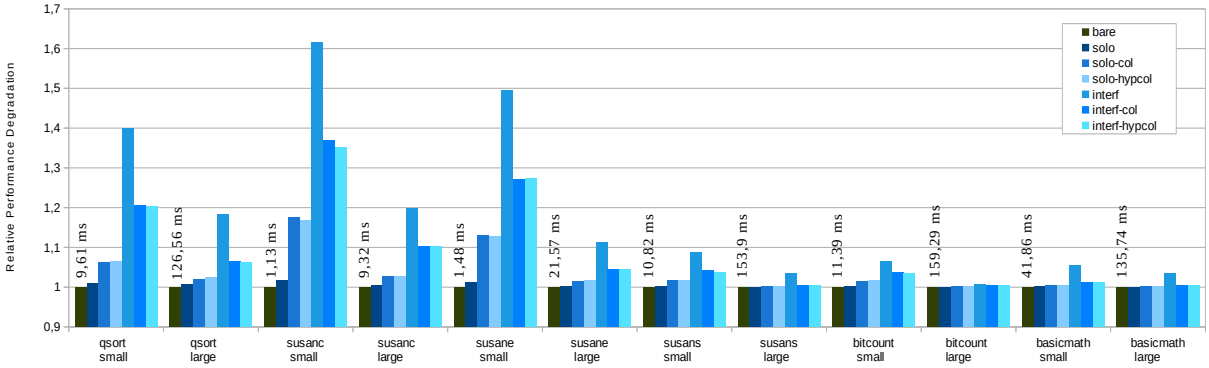


Figure 7.2: Relative performance overhead of MiBench automotive suite relative to bare-metal execution.

7.4 Interrupt Latency

To measure interrupt latency and respective interference, we develop a custom MMIO timer (0x22000000) with auto-restart feature, and an interrupt line connected to the PLIC. This timer offers a simple interface with 3 control registers: the *time* (counter timer value), the *timecmp* (counter compare value, i.e., when *time* > *timecmp* the interrupt is trigger) and *enable* register (enable and disables the interrupt). We developed a custom bare-metal benchmark application to measure the interrupt latency using our custom timer. The application programs the timer to trigger an interrupt at 100Hz (each 10 ms). The latency is given by the value read from the time registers at the interrupt handler minus the value programmed. We invalidate the L1 instruction cache at each measurement using the *fence.i* instruction as we believe it is more realistic to assume this cache is not hot with regard to the interrupt handling or the hypervisor's interrupt injection code.

For the hosted executions, we performed measurements for both trap-and-emulate and PLIC interrupts direct injection. Figure 7.3 shows the average of the results obtained from 100 samples (the first two discarded).

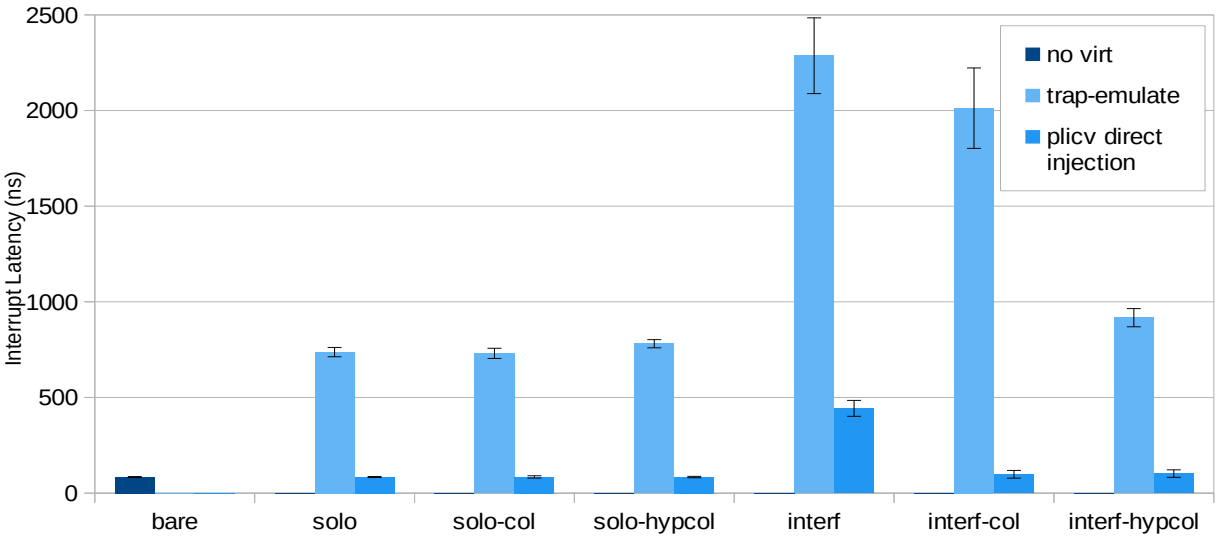


Figure 7.3: Interrupt latency for bare-metal execution and hosted execution.

The interrupt latency for the bare (in Figure 7.3, no virt) execution is quite low (approx. 80 ns) and steady. The trap-and-emulate approach introduces a penalty of an order of magnitude (740 ns) that is even more significant under interference (up to 2280 ns, about 300%) both in average and standard deviation. Applying cache partitioning via coloring helps to mitigate this, which shows that most of the interference happens in the shared L2 LLC. The difference between inter-col and interf-hypcol shows that it is of utmost

importance to assign dedicated cache partitions to the hypervisor: the interfering VM also interferes with the hypervisor while injecting the interrupt and not only with the benchmark code execution itself.

Figure 7.3 also shows that the effect of the direct injection achieved with guest external interrupt and PLIC virtualization support can bring guest interrupt latency to near-native values. Furthermore, it shows only a fractional increase under interference (when compared to the trap-and-emulate approach) which can also be attenuated with cache coloring. As the hypervisor no longer intervenes in interrupt injection, for this case, it suffices to color guest memory. A small note is that the use of cache coloring does not affect the benchmark for solo execution configurations, given that the benchmark code is very small. Thus, the L1 and L2 caches can easily fit both the benchmark and hypervisor's injection code. Finally, we can conclude that with PLIC virtualization support, it is possible to significantly improve external interrupt latencies for VMs.

7.5 Discussion

The RISC-V H-extension is currently in its 0.6.1 version and is being developed within the privileged specification working group of RISC-V International, following a well-defined extension development lifecycle. The specification draft has been stable for quite some time and therefore is approaching a frozen state, after which it will enter a period of public review before finally being ratified. However, to enter a frozen state, it will need both (i) open RTL core implementations suitable for deployment as soft-cores on FPGA platforms and (ii) hypervisor ports that exercise its mechanisms and provide feedback. Until the extensions are ratified, we do not expect any commercial IP or ASIC implementations to be available. With this work, we have contributed with one open RTL implementation, but more are needed. Xvisor and KVM have been the open-source reference hypervisors used in the extension development process. We have further contributed with the Bao port, but the more hypervisor ports are available to evaluate the suitability of the H-extension for different hypervisor architectures, the better.

As discussed in this dissertation, there are still some gaps in RISC-V, particularly with respect to virtualization. At the ISA level, features like cache management operations are needed. Fortunately, there is already a working group defining these mechanisms. At a platform level, timer and external interrupt virtualization support is needed. Our results show the importance of these mechanisms to achieve low and deterministic interrupt latency in virtualized real-time systems. There are already efforts within the RISC-V community to provide this support: a new extension proposal is on the fast track to include dedicated timers

for HS- VS-modes; and a new interrupt controller architecture featuring support for message-signaled interrupts (MSI) and virtualization support is under development within the privileged specification working group. Another missing component critical for virtualization is the IOMMU. An IOMMU is needed to implement efficient virtualization by allowing the direct assignment of DMA-capable devices to VMs while guaranteeing strong isolation between VMs and the hypervisor itself. Static partitioning hypervisors such as Bao ultimately depend on IOMMU, as they do not provide any kind of device emulation and only pass-through access. At the moment, in a RISC-V platform, a Bao guest that wishes to use a DMA device must have all its memory configured with identity mapping. Unfortunately, this still completely breaks encapsulation, serving only for experimentation and demonstration purposes, not being suitable for production.

In the conducted evaluation presented in this chapter, we have demonstrated something well-understood and documented in the literature [Yun et al., 2013, Mancuso et al., 2013, Kloda et al., 2019, Xu et al., 2019, Martins et al., 2020, Farshchi et al., 2020], i.e., that (i) in multi-core platforms, there is significant inter-core interference due to shared micro-architectural resources (e.g., caches, buses, memory controllers), (ii) which can be minimized by mechanisms such as page coloring used to partition shared caches. Other techniques such as memory bandwidth reservations [Yun et al., 2013] and DRAM bank partitioning [Yun et al., 2014] can minimize interference further ahead in the memory hierarchy. These partitioning mechanisms are important in embedded mixed-criticality systems both from the security and safety perspectives by protecting against side-channel attacks and guaranteeing determinism and freedom-from-interference required by certification standards (e.g., ISO26262). They are also useful for server systems by helping to guarantee quality-of-service (QoS) and increase overall utilization [Lo et al., 2015]. However, software-based approaches typically have significant overheads and increase the trusted computing base (TCB) complexity. Academic works such as Hybcache [Dessouky et al., 2020] or the bandwidth regulation unit (BRU) [Farshchi et al., 2020] propose the implementation of this kind of mechanism in RISC-V cores (Ariane [Zaruba and Benini, 2019] and Rocket, respectively). SiFive has provided cache partitioning mechanisms in hardware via way-locking. During this work, we found it would be useful to have a standard set of mechanisms and interfaces to rely on. We argue that RISC-V is also missing a standard extension to provide such facilities. Other ISAs have already introduced these ideas, e.g., Intel's CAT and Arm's MPAM [Arm Ltd., 2018a]. MPAM functionality is also extended to other virtualization-critical system-bus masters, including the GIC and the SMMU (Arm's interrupt controller and IOMMU, respectively), something that should also be taken into account when developing similar RISC-V specifications.

Even without virtualization support, it is possible to implement static partitioning in RISC-V leveraging

the trap-and-emulate features described in Chapter 3 and using the PMP for memory isolation instead of two-stage translation. The PMP is a RISC-V standard component that allows M-mode software to white-list (or black-list) physical address space regions on a per-core basis. This results in a kind of para-virtual approach, as the guest must be aware of the full physical address space and possibly recompiled for different system configurations. To provide direct assignment of DMA devices, the host platform would also need to provide IOPMPs (akin to IOMMU, without translation), which is a specification already on course. Furthermore, the hypervisor would be forced to flush micro-architectural state such as TLBs or virtual caches at each context switch resulting in significant performance overheads. The use of VMIDs, part of the H-extension, tackles this issue. Notwithstanding, this is not a real problem for statically partitioned systems. Thus, once there is no commercial hardware featuring the H-extension available in the market, this is the approach of some of the hypervisors mentioned in Chapter 2. We are currently developing a customized version of Bao to run in RISC-V platforms without H-extension support (e.g., Microchip PolarFire SoC Icicle or the upcoming PicoRio). Nevertheless, we believe the hypervisor extension is still a better primitive for implementing these systems, given the higher flexibility and scalability it provides.

Chapter 8

Conclusion and Future Work

In this dissertation, we have implemented the first public implementation of the RISC-V H-extension in a real RISC-V core, i.e., Rocket core. Moreover, during our research, we identify some critical components not covered by the RISC-V virtualization extensions, particularly the interrupt subsystem and the timer infrastructure. Without hardware virtualization support in the interrupt controller, the hypervisor has to fully emulate the PLIC, which involves many traps. Furthermore, the RISC-V specification defines only one physical timer infrastructure at the most privileged level (M-Mode), which is multiplexed into logical timers to the lower privileged levels. To tackle these issues, we propose a set of hardware enhancements to the interrupt controller and the timer infrastructure in the spirit of reduce hypervisor intervention and improve overall virtualization efficiency. To validate and evaluate our hardware implementation, we used the open-source Bao hypervisor, which already had support for the H-extension specification.

As of this writing, we achieved functional verification of our implementation on a Verilator-generated simulator and on a Zynq UltraScale+ MPSoC ZCU104 FPGA using a dedicated testing framework for that purpose and later the Bao hypervisor. During the tests and evaluation phase, we have carried out an extensive set of experiments in FireSim, a cycle-accurate simulator with performance results nearly to what is found in a taped-out chip. Our evaluation focuses on four main metrics: performance, inter-VM interference, interrupt latency and hardware overhead. Results show that the H-extension, per se, introduces a slight performance penalty compared to what is observed in bare-metal execution. Moreover, we demonstrated that without additional hardware to minimize interference and interrupt latency can impose a prohibitive cost for MCSs. Our proposed architectural enhancements considerably minimize these effects by reducing hypervisor intervention interrupt latency and interference by order of magnitude. We have also shown that most interference origin from shared L2 cache level, which we mitigate by applying cache coloring to both VM's and hypervisor. Finally, we concluded that although our architecture virtualization enchantments assume a cost in hardware resources, it is somehow bearable considering the

benefits.

Lastly, we discussed identified gaps existing in RISC-V regarding virtualization and outlined ongoing internal efforts within RISC-V virtualization. Our hardware design was made freely available for the RISC-V community and is currently being used by the KVM hypervisor as the single reference implementation to ratify the H-extension.

8.1 Future Work

With this dissertation, we have successfully augmented Rocket Core with all mandatory H-extension specifications. Nevertheless, some optional features of the hypervisor extensions were left out of implementation for future work. For example, the *htinst* and *mtinst* which were defined in the core but left as hardwire to zero. These registers would provide a quicker and less intrusive manner to read the trapping instruction in a pre-decoded format. This architectural feature would be a great advantage as the hypervisor could handle all traps more efficiently and with less code without actually reading guest instruction from memory and decoding it, which consequently means less data cache pollution. Furthermore, the RV32 CSRs version of the H-extension is yet to be implemented, as we only support the RV64.

However, we believe there are more pressing issues to be resolved which could greatly benefit virtualization overall's performance. Firstly, at the shared micro-architectural level, we identify four major improvements that could be performed as future steps:

- Although we have successfully implemented a 2-stage address, there are still some features missing and much room for optimizations. For instance, our implementation only supports *bare* and *Sv39x4* address-translation schemes modes for guest physical addresses, leaving still missing the extended version for 44-bit virtual addresses. Moreover, we have not included VMID at the TLB entry tagging. Although this is a useless feature in Bao as it is a statically partitioning hypervisor, the same does not stand for other types of hypervisors. Without it, each context-switch would require invalidation of such structures.

- Results shown that inter-VM interference at cache level is one major source of performance degradation. So, as future work, we believe a deeper evaluation of micro-architectural interference effects

would be required. Moreover, in turn, develop additional mechanisms to help reduce inter-VM interference on shared hardware subsystems, such as Hybcache [Dessouky et al., 2020] or the bandwidth regulation unit (BRU) [Farshchi et al., 2020], or SiFive cache partition, already mentioned in the discussion.

- At the time being, RISC-V still has no IOMMU support for DMA capable devices. Without this feature, it would be impossible to direct assign devices to VM's as it would mean a breach in isolation between VMs and the hypervisor itself. For instance, static partitioning hypervisors such as Bao ultimately depend on IOMMU, as they do not provide any kind of device emulation and only pass-through access. Given that, we believe that developing an IOMMU specification is one necessary step to achieve a fully capable virtualized RISC-V based system.

Finally, there is still one crucial topic that needs to be addressed soon: IPIs support virtualized RISC-V systems. At the time being, communication across harts is solely restricted to M-mode execution, normally accessed to memory-mapped registers as part of the CLINT specification. Like the timer feature, IPIs for lower privilege levels are implemented by the firmware and available through the SBI interface. Thus guests running on VS-mode that which to communicate with other vhart, need to trap to HS-mode and then to M-mode. More traping, in turn, translates to less performance, which will surely take more impact on embedded systems with real-time constraints. Therefore, as future work, it would be interesting to propose changes to the current specification to provide architectural support for IPIs directly without firmware intervention at HS-level and VS-level.

References

[Arm Ltd., 2018a] Arm Ltd. (2018a). Arm Architecture Reference Manual Supplement - Memory System Resource Partitioning and Monitoring (MPAM), for Armv8-A.

[Arm Ltd., 2018b] Arm Ltd. (2018b). Isolation using virtualization in the Secure world Secure world software architecture on Armv8.4.

[Asanovi et al., 2016] Asanovi, K., Avižienis, R., Bachrach, J., Beamer, S., Biancolin, D., Celio, C., Cook, H., Dabbelt, P., Hauser, J., Izraelevitz, A., Karandikar, S., Keller, B., Kim, D., Koenig, J., Lee, Y., Love, E., Maas, M., Magyar, A., Mao, H., Moreto, M., Ou, A., Patterson, D., Richards, B., Schmidt, C., Twigg, S., Vo, H., and Waterman, A. (2016). The Rocket Chip Generator. Technical report.

[Bechtel and Yun, 2019] Bechtel, M. and Yun, H. (2019). Denial-of-Service Attacks on Shared Cache in Multicore: Analysis and Prevention. In *IEEE RTAS*, pages 357–367.

[Bechtel and Yun, 2020] Bechtel, M. and Yun, H. (2020). Exploiting DRAM Bank Mapping and HugePages for Effective Denial-of-Service Attacks on Shared Cache in Multicore. In *Symposium on Hot Topics in the Science of Security*, New York, NY, USA.

[Behrens et al., 2020] Behrens, J., Skeggs, C., Ortiz, S., and Kaashoek, F. (2020). Rvirt. `https://github.com/mit-pdos/RVirt`.

[Berkeley, 2019a] Berkeley (2019a). Chisel/FIRRTL: Home. https://www.chisel-lang.org/. Accessed on 2019-11-14.

[Berkeley, 2019b] Berkeley (2019b). GitHub - chipsalliance/rocket-chip: Rocket Chip Generator. https://github.com/chipsalliance/rocket-chip. Accessed on 2019-11-18.

[Bhargava et al., 2008] Bhargava, R., Serebrin, B., Spadini, F., and Manne, S. (2008). Accelerating two-dimensional page walks for virtualized systems. *SIGARCH Comput. Archit. News*, 36(1):26–35.

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

[Cerdeira et al., 2020] Cerdeira, D., Santos, N., Fonseca, P., and Pinto, S. (2020). Sok: Understanding the prevailing security vulnerabilities in trustzone-assisted tee systems. In *2020 IEEE Symposium on Security and Privacy (SP)*, pages 1416–1432.

[Chipyard, 2020] Chipyard (2020). Rocket Chip. https://chipyard.readthedocs.io/en/latest/Generators/Rocket-Chip.html. Accessed on 2020-02-20.

[Cicero et al., 2018] Cicero, G., Biondi, A., Buttazzo, G., and Patel, A. (2018). Reconciling security with virtualization: A dual-hypervisor design for ARM TrustZone. *Proceedings of the IEEE International Conference on Industrial Technology*, 2018-Febru:1628–1633.

[Costa et al., 2020] Costa, M., Moreira, R., Cabral, J., Dias, J., and Pinto, S. (2020). Wall screen: An ultra-high definition video-card for the internet of things. *IEEE MultiMedia*, 27(3):76–87.

[Crespo et al., 2010] Crespo, A., Ripoll, I., and Masmano, M. (2010). Partitioned Embedded Architecture Based on Hypervisor: The XtratuM Approach. In *2010 European Dependable Computing Conference*, pages 67–72.

[Dall, 2018] Dall, C. (2018). The Design , Implementation , and Evaluation of Software and Architectural Support for ARM Virtualization.

[Dall and Nieh, 2014] Dall, C. and Nieh, J. (2014). KVM/ARM: The Design and Implementation of the Linux ARM Hypervisor. In *Proceedings of the 19th International Conference on ASPLOS*, page 333–348, New York, NY, USA.

[Dessouky et al., 2020] Dessouky, G., Frassetto, T., and Sadeghi, A.-R. (2020). Hybcache: Hybrid side-channel-resilient caches for trusted execution environments. In *USENIX Security Symposium*, pages 451–468.

[Drew Barbier, 2020] Drew Barbier, Palmer Dabbelt, A. C. (2020). Risc-v platform-level interrupt controller specification. `https://https://github.com/riscv/riscv-plic-spec/blob/master/riscv-plic.adoc`.

[Farshchi et al., 2020] Farshchi, F., Huang, Q., and Yun, H. (2020). BRU: Bandwidth Regulation Unit for Real-Time Multicore Processors. In *IEEE RTAS*, pages 364–375.

[Garcia, 2015] Garcia, P. (2015). Hybrid Hypervisor Partially Deployed on FPGA.

[Garcia et al., 2014] Garcia, P., Gomes, T., Salgado, F., Monteiro, J., and Tavares, A. (2014). Towards Hardware Embedded Virtualization Technology: Architectural Enhancements to an ARM SoC. *SIGBED Rev.*, 11(2):45–47.

[Ge et al., 2018] Ge, Q., Yarom, Y., Cock, D., and Heiser, G. (2018). A survey of microarchitectural timing attacks and countermeasures on contemporary hardware. *Journal of Cryptographic Engineering*, 8(1):1–27.

[Gómez et al., 2020] Gómez, F., Masmano, M., Nicolau, V., Andersson, J., Le Rhun, J., Trilla, D., Gallego, F., Cabo, G., and Abella, J. (2020). De-RISC–Dependable Real-Time Infrastructure for Safety-Critical Computer Systems. *ADA USER*, 41(2):107.

[Heiser, 2008] Heiser, G. (2008). The role of virtualization in embedded systems. In *IIES 2008 - Proceedings of the 1st Workshop on Isolation and Integration in Embedded Systems*, pages 11–16.

[Heiser, 2011] Heiser, G. (2011). *Virtualizing Embedded Systems-Why Bother?*

[Heiser, 2020] Heiser, G. (2020). seL4 is verified on RISC-V! In *RISC-V International*.

[Hwang et al., 2008] Hwang, J., Suh, S., Heo, S., Park, C., Ryu, J., Park, S., and Kim, C. (2008). Xen on ARM: System Virtualization Using Xen Hypervisor for ARM-Based Secure Mobile Phones. In *2008 5th IEEE Consumer Communications and Networking Conference*, pages 257–261.

[Intel, 2011] Intel (2011). *Intel 64 and IA-32 Architectures Software Developer's Manual Combined Volumes*, volume 3.

[Ke, 2009] Ke, Y. (2009). Intel Virtualization Technology.

[Klein et al., 2009] Klein, G., Elphinstone, K., Heiser, G., Andronick, J., Cock, D., Derrin, P., Elkaduwe, D., Engelhardt, K., Kolanski, R., Norrish, M., Sewell, T., Tuch, H., and Winwood, S. (2009). SeL4: Formal Verification of an OS Kernel. In *ACM SOSP*, page 207–220, New York, NY, USA.

[Kloda et al., 2019] Kloda, T., Solieri, M., Mancuso, R., Capodieci, N., Valente, P., and Bertogna, M. (2019). Deterministic Memory Hierarchy and Virtualization for Modern Multi-Core Embedded Systems. In *IEEE RTAS*.

[Lee et al., 2016] Lee, C., Kim, S. W., and Yoo, C. (2016). VADI: GPU Virtualization for an Automotive Platform. *IEEE Transactions on Industrial Informatics*, 12(1):277–290.

[Liao et al., 2017] Liao, S., Yang, X., Guo, W., Sun, H., Jiang, Z., and Zhao, X. (2017). Gemini: A lightweight virtualization architecture for protecting privacy and security of smartphone. In *Proceedings - 2016 13th International Conference on Embedded Software and System, ICESS 2016*, pages 186–191. Institute of Electrical and Electronics Engineers Inc.

[Lim et al., 2017] Lim, J. T., Dall, C., Li, S.-W., Nieh, J., and Zyngier, M. (2017). NEVE: Nested Virtualization Extensions for ARM. In *Proceedings of SOSP*, page 201–217, New York, NY, USA.

[Limited, 2016] Limited, A. (2016). ARM ® Generic Interrupt Controller Architecture Specification GIC architecture version 3.0 and version 4.0 ARM Generic Interrupt Controller Architecture Specification GIC architecture version 3.0 and version 4.0.

[Lingeswaran, 2017] Lingeswaran, R. (2017). Para virtualization vs Full virtualization vs Hardware assisted Virtualization. https://www.unixarena.com/2017/12/para-virtualization-full-virtualization-hardware-assisted-virtualization.html/. Accessed on 2019-10-22.

[Lo et al., 2015] Lo, D., Cheng, L., Govindaraju, R., Ranganathan, P., and Kozyrakis, C. (2015). Heracles: Improving resource efficiency at scale. In *ACM/IEEE ISCA*, pages 450–462.

[Lublin et al., 2007] Lublin, U., Kamay, Y., Laor, D., and Liguori, A. (2007). KVM: the Linux virtual machine monitor. In *Proceedings of the Linux Symposium*.

[Mancuso et al., 2013] Mancuso, R., Dudko, R., Betti, E., Cesati, M., Caccamo, M., and Pellizzoni, R. (2013). Real-time cache management framework for multi-core architectures. In *IEEE RTAS*, pages 45–54.

[Martins et al., 2017] Martins, J., Alves, J., Cabral, J., Tavares, A., and Pinto, S. (2017). μrtzvisor: A secure and safe real-time hypervisor. *Electronics*, 6:93.

[Martins et al., 2020] Martins, J., Tavares, A., Solieri, M., Bertogna, M., and Pinto, S. (2020). Bao: A Lightweight Static Partitioning Hypervisor for Modern Multi-Core Embedded Systems. In *Workshop on NG-RES*, volume 77, pages 3:1–3:14. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik.

[Masmano et al., 2009] Masmano, M., Ripoll, I., and Crespo, A. (2009). XtratuM: a Hypervisor for Safety Critical Embedded Systems. *11th Real-Time Linux Workshop*, (September 2009):263–272.

[Masood et al., 2015] Masood, A., Sharif, M., Yasmin, M., and Raza, M. (2015). Virtualization Tools and Techniques: Survey. *Nepal Journal of Science and Technology*, 15(2):141–150.

[Menascé, 2005] Menascé, D. A. (2005). Virtualization: Concepts, applications, and performance modeling. In *31st International Conference Computer Measurement Group*.

[Moratelli et al., 2016] Moratelli, C., Filho, S., and Hessel, F. (2016). Hardware-assisted interrupt delivery optimization for virtualized embedded platforms. *Proceedings of the IEEE International Conference on Electronics, Circuits, and Systems*, 2016-March:304–307.

[Nakivo, 2018] Nakivo (2018). Hyper-V vs. VirtualBox Comparison. https://www.nakivo.com/blog/hyper-v-virtualbox-one-choose-infrastructure/. Accessed on 2019-11-18.

[Nguyen, 2016] Nguyen, K. (2016). Introduction to Cache Allocation Technology in the Intel® Xeon® Processor E5 v4 Family.

[Oliveira et al., 2018a] Oliveira, A., Martins, J., Cabral, J., Tavares, A., and Pinto, S. (2018a). Tz- virtio: Enabling standardized inter-partition communication in a trustzone-assisted hypervisor. In *2018 IEEE 27th International Symposium on Industrial Electronics (ISIE)*, pages 708–713.

[Oliveira et al., 2018b] Oliveira, D., Gomes, T., and Pinto, S. (2018b). Towards a green and secure architecture for reconfigurable iot end-devices. In *2018 ACM/IEEE 9th International Conference on Cyber-Physical Systems (ICCPS)*, pages 335–336.

[Patel et al., 2015] Patel, A., Daftedar, M., Shalan, M., and El-Kharashi, M. W. (2015). Embedded Hypervisor Xvisor: A Comparative Analysis. In *Euromicro International Conference on Parallel, Distributed, and Network-Based Processing*, pages 682–691.

[Patterson and Waterman, 2017] Patterson, D. and Waterman, A. (2017). *The RISC-V Reader: An Open Architecture Atlas*. Beta editi edition.

[Pinto et al., 2019] Pinto, S., Araujo, H., Oliveira, D., Martins, J., and Tavares, A. (2019). Virtualization on TrustZone-Enabled Microcontrollers? Voilà! In *IEEE RTAS*, pages 293–304.

[Pinto et al., 2017a] Pinto, S., Oliveira, A., Pereira, J., Cabral, J., Monteiro, J., and Tavares, A. (2017a). Lightweight multicore virtualization architecture exploiting ARM TrustZone. *Proceedings IECON 2017 - 43rd Annual Conference of the IEEE Industrial Electronics Society*, 2017-Janua:3562–3567.

[Pinto et al., 2014] Pinto, S., Oliveira, D., Pereira, J., Cardoso, N., Ekpanyapong, M., Cabral, J., and Tavares, A. (2014). Towards a lightweight embedded virtualization architecture exploiting ARM TrustZone. *19th IEEE International Conference on Emerging Technologies and Factory Automation, ETFA 2014*, pages 1–4.

[Pinto et al., 2017b] Pinto, S., Pereira, J., Gomes, T., Tavares, A., and Cabral, J. (2017b). Ltzvisor: Trustzone is the key. pages 4:1–4:22.

[Pinto and Santos, 2019] Pinto, S. and Santos, N. (2019). Demystifying Arm TrustZone: A Comprehensive Survey. *ACM Comput. Surv.*, 51(6).

[Pinto et al., 2016] Pinto, S., Tavares, A., and Montenegro, S. (2016). Space and Time Partitioning with Hardware Support for Space Applications. In Ouwehand, L., editor, *DASIA 2016 - Data Systems In Aerospace*, volume 736 of *ESA Special Publication*, page 19.

[Popek and Goldberg, 1974] Popek, G. J. and Goldberg, R. P. (1974). Formal Requirements for Virtualizable Third Generation Architectures. *Communications of the ACM*, 17(7):412–421.

[Ramsauer et al., 2017] Ramsauer, R., Kiszka, J., Lohmann, D., and Mauerer, W. (2017). Look Mum, no VM Exits!(Almost). In *Workshop on OSPERT*.

[Reinhardt and Morgan, 2014] Reinhardt, D. and Morgan, G. (2014). An embedded hypervisor for safety-relevant automotive E/E-systems. In *Proceedings of the 9th IEEE International Symposium on Industrial Embedded Systems, SIES 2014*, pages 189–198. IEEE Computer Society.

[Robin and Irvine, 2000] Robin, J. S. and Irvine, C. E. (2000). Analysis of the intel pentium's ability to support a secure virtual machine monitor. In *9th USENIX Security Symposium (USENIX Security 00)*, Denver, CO. USENIX Association.

[Shropshire, 2014] Shropshire, J. (2014). Analysis of monolithic and microkernel architectures: Towards secure hypervisor design. *Proceedings of the Annual Hawaii International Conference on System Sciences*, pages 5008–5017.

[SiFive, 2017] SiFive, H. C. (2017). Diplomatic design patterns : A tilelink case study.

[Silva et al., 2019] Silva, M., Cerdeira, D., Pinto, S., and Gomes, T. (2019). Operating systems for internet of things low-end devices: Analysis and benchmarking. *IEEE Internet of Things Journal*, 6(6):10375–10383.

[Thiebaut et al., 2016] Thiebaut, S. S., De Rosa, A. D., and Sasse, R. (2016). Secure Embedded Hypervisor Based Systems for Automotive. *Proceedings - 46th Annual IEEE/IFIP International Conference on Dependable Systems and Networks, DSN-W 2016*, pages 211–212.

[Tu et al., 2015] Tu, C. C., Ferdman, M., Lee, C. T., and Chiueh, T. C. (2015). A comprehensive implementation and evaluation of direct interrupt delivery. *VEE 2015 - Proceedings of the 11th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*, pages 1–15.

[Uhlig et al., 2005] Uhlig, R., Neiger, G., Rodgers, D., Santoni, A. L., Martins, F. C., Anderson, A. V., Bennett, S. M., Kägi, A., Leung, F. H., and Smith, L. (2005). Intel virtualization technology. *Computer*, 38(5):48–56.

[Varanasi and Heiser, 2011] Varanasi, P. and Heiser, G. (2011). Hardware-supported virtualization on ARM. *Proceedings of the 2nd Asia-Pacific Workshop on Systems, APSys'11*.

[Waterman et al., 2020] Waterman, A., Asanović, K., and Hauser, J. (2020). The RISC-V Instruction Set Manual Volume II: Privileged Architecture, Document Version 1.12-draft. *RISC-V Foundation, December*.

[Waterman et al., 2014] Waterman, A., Lee, Y., and Patterson, D. (2014). The RISC-V Instruction Set Manual Volume I:Unprivileged ISA Version 20190621. I.

[Williams, 2020] Williams, C. (2020). Diosix. `https://diosix.org/`. Last checked on Fev 19, 2020.

[Xu et al., 2019] Xu, M., Phan, L. T. X., Choi, H., Lin, Y., Li, H., Lu, C., and Lee, I. (2019). Holistic Resource Allocation for Multicore Real-Time Systems. In *IEEE RTAS*, pages 345–356.

[Xu et al., 2017] Xu, M., Thi, L., Phan, X., Choi, H., and Lee, I. (2017). vCAT: Dynamic Cache Management Using CAT Virtualization. In *IEEE RTAS*.

[Yun et al., 2014] Yun, H., Mancuso, R., Wu, Z., and Pellizzoni, R. (2014). PALLOC: DRAM bank-aware memory allocator for performance isolation on multicore platforms. In *IEEE RTAS*, pages 155–166.

[Yun et al., 2013] Yun, H., Yao, G., Pellizzoni, R., Caccamo, M., and Sha, L. (2013). MemGuard: Memory bandwidth reservation system for efficient performance isolation in multi-core platforms. In *IEEE RTAS*, pages 55–64.

[Zaruba and Benini, 2019] Zaruba, F. and Benini, L. (2019). The Cost of Application-Class Processing: Energy and Performance Analysis of a Linux-Ready 1.7-GHz 64-Bit RISC-V Core in 22-nm FDSOI Technology. *IEEE Trans. on Very Large Scale Integration Systems*, 27(11):2629–2640.

[Zhang et al., 2018] Zhang, W., Li, Y., and Liu, Y. (2018). A survey of direct interrupt delivery techniques in I/O virtualization. *Proceedings of the IEEE International Conference on Software Engineering and Service Sciences, ICSESS*, 2017-Novem:169–172.