

Software Infrastructure for Isolation and Performance Monitoring in Virtualized Systems

by

Abdur Rahman

A thesis
presented to the University of Waterloo
in fulfillment of the
thesis requirement for the degree of
Master of Applied Science
in
Electrical and Computer Engineering

Waterloo, Ontario, Canada, 2025

© Abdur Rahman 2025

Author's Declaration

I hereby declare that I am the sole author of this thesis. This is a true copy of the thesis, including any required final revisions, as accepted by my examiners.

I understand that my thesis may be made electronically available to the public.

Abstract

Modern multiprocessor System-on-Chip (SoC) architectures host a rich tapestry of heterogeneous components, enabling multiple workloads with differing requirements to run simultaneously on the same hardware platform. However, managing and isolating these concurrently running applications presents significant challenges. Traditional virtualization techniques, even with static partitioning hypervisors, could struggle to ensure robust isolation due to contention in shared system resources such as caches and memory bandwidth. To address this issue, this thesis investigates memory bandwidth contention among cores and explores isolation strategies by implementing MemGuard [56] in the Bao Hypervisor on ARMv8-based systems. This implementation is complemented by cache coloring and DRAM bank partitioning techniques. The results, evaluated using the San Diego Vision Benchmark Suite[50], quantify the effectiveness of these mechanisms in reducing interference and provide insights into program behavior under varying isolation parameters.

Beyond improving isolation, performance monitoring must extend beyond core-level observation to encompass system-wide interactions. To this end, this thesis develops a comprehensive software infrastructure for an Advanced Performance Monitoring Unit (APMU), designed for event-driven monitoring and dynamic runtime reconfiguration. By leveraging an LLVM-based toolchain to support custom instructions and integrating seamlessly with the hypervisor and guest OS layers, the APMU framework enables diverse applications while optimizing memory utilization and execution time.

Collectively, the results and infrastructure presented in this work contribute to more predictable, secure, and efficient computing systems, advancing the state of the art in virtualization, performance isolation, and heterogeneous system analysis.

Acknowledgements

First and foremost, I thank God for granting me the strength and perseverance to complete this journey.

I am deeply grateful to my parents and siblings for their unwavering love, support, and encouragement, which have been a constant source of motivation.

I extend my heartfelt gratitude to my supervisor, Dr. Rodolfo Pellizzoni, for his invaluable guidance and mentorship. Throughout this journey his guidance imparted me with invaluable knowledge and experience. I am also thankful to my thesis reviewers, Dr. Sebastian Fischmeister and Dr. Seyed Majid Zahedi, for their insightful feedback and valuable suggestions.

Additionally, I would like to thank my research group mate Muhammad Sajjad Jaffri who provided the necessary platform to conduct my research and the rest of my group members including Wafic Lawand for their consistent support and motivation throughout this work.

To everyone who has contributed to my journey, I express my sincere gratitude.

Dedication

This thesis is dedicated to all those who pursue knowledge, strive for innovation, and work tirelessly to overcome challenges in their journey of growth and discovery.

Table of Contents

Author's Declaration	ii
Abstract	iii
Acknowledgements	iv
Dedication	v
List of Figures	x
List of Tables	xi
1 Introduction	1
1.1 Motivation	1
1.2 Research Objectives	2
1.3 Contributions	3
1.4 Thesis Organization	3
1.5 Acknowledgments	4
2 Background	5
2.1 Virtualization	5
2.1.1 Two-Stage Address Translation	6

2.2	Hypervisors	7
2.2.1	Static Partitioning	8
2.2.2	Bao Hypervisor	9
2.3	Performance Monitoring	10
2.3.1	Software Frameworks for Performance Monitoring	12
2.4	Temporal Isolation	12
2.4.1	Cache Partitioning	12
2.4.2	DRAM Bank Partitioning	14
2.4.3	Memory Bandwidth Partitioning	14
2.5	Advanced Performance Monitoring Unit	16
2.5.1	Event Units (EVUs)	16
2.5.2	APMU Counters and Event Processing	17
2.5.3	APMU Core	17
2.5.4	Implementation	18
2.6	LLVM Overview	20
2.6.1	TableGen Reference	21
2.6.2	LLVM Backend	22
3	Memory Isolation in ARM Cortex-A53	23
3.1	Memory Geometry and Cache Coloring	24
3.2	Evaluation of Shared Memory Contention	25
3.2.1	IsolBench	26
3.2.2	Experimentation	28
3.3	Memguard Implementation on Bao Hypervisor	30
3.3.1	Performance Monitoring Unit Hardware	31
3.3.2	Hardware Configuration for PMU and Timer	32
3.3.3	Software Configuration for Interrupts	33
3.3.4	Interrupt Service Routines	34

3.3.5	Timer Interrupt Overhead	35
3.3.6	Minimum Sustainable Bandwidth	36
3.4	Evaluation of Bandwidth Regulation	37
3.4.1	Evaluation of SDVB Benchmarks	38
3.5	Discussion	40
3.6	Conclusion	43
4	APMU Software Support	44
4.1	APMU Software Framework Overview	45
4.2	Toolchain	46
4.2.1	Inline Assembly for Custom Instructions	47
4.2.2	Inline Assembly in LLVM	48
4.2.3	Adding New Instructions in RISC-V Backend	49
4.2.4	Linker and LibC	50
4.2.5	Testing	51
4.3	APMU Core Software Runtime	51
4.3.1	Queues	53
4.3.2	Components	56
4.3.3	Event Triggered Scheduler	58
4.3.4	Base Component	60
4.3.5	Miscellaneous Drivers	61
4.3.6	Configuration	61
4.4	Application Core Runtime Software	61
4.4.1	APMU Bootup	62
4.4.2	APMU Bidirectional Communication	62
4.5	Dynamic Component Deployment	63
4.5.1	Memory Allocator	64
4.5.2	Dynamic Linker	66

4.6	Evaluation	68
4.6.1	Platform	68
4.6.2	Software Setup	69
4.6.3	Execution Time	70
4.6.4	Code Size	72
4.7	Conclusion	72
5	Conclusion	74
	References	76
	APPENDICES	82
A	APMU Software Stack	83
A.1	Linker and Sample Memory Map for APMU	83
A.2	Configuration Files	85
A.2.1	APMU Configuration	86
A.2.2	VM Configuration in Bao	87
A.3	Supported Relocation Types	89

List of Figures

2.1	Two stage address translation	6
2.2	Virtualized system with hypervisor	8
2.3	APMU Block Diagram	16
2.4	Common RISC-V Instruction Formats from Integer Extension	19
2.5	Counter Instruction Formats	20
2.6	APMU Block Diagram	21
2.7	Stages of LLVM Based Compilation	21
3.1	Memory geometry of ZCU102	25
3.2	Contention results for interference tests.	29
3.3	PMU Architecture in Cortex A53	32
3.4	Timer interrupt overhead in terms of slowdown of synthetic read benchmark	36
3.5	Observed bandwidth for different interference tests	37
3.6	Results for San Diego Vision Benchmark without interference	41
3.7	Results for San Diego Vision Benchmark with interference	42
4.1	Software Infrastructure for APMU	45
4.2	Software Block Diagram	52
4.3	Queue Block Diagram	54
4.4	Block diagram representing example queue communication	63
4.5	Installation of dynamic component	65
4.6	Effect of component ID on event handler trigger latency	72

List of Tables

2.1	Hypervisors and Supported Architectures	10
3.1	Cache Geometry Calculations for ZCU102	24
4.1	Matching Compiled Machine Code with Instruction Encodings	51
4.2	Summary of Relevant ELF Structures	68
4.3	Execution time breakdown for various operations in the software setup . .	71
A.1	Supported RISC-V Relocation Types	89

Chapter 1

Introduction

1.1 Motivation

Modern computing platforms, ranging from embedded devices to large-scale data centers, are becoming increasingly complex and heterogeneous. Multiprocessor System-on-Chip (SoC) architectures integrate multiple cores—often with differing capabilities—alongside specialized accelerators, memory controllers, and communication fabrics. This heterogeneity enables high computational throughput, fine-tuned energy efficiency, and flexible execution of diverse workloads. As a result, contemporary SoCs can simultaneously host multiple applications. This trend is growing both in the embedded systems domain, where industry is shifting towards packaging different functionalities within the same SoC, as well as the cloud domain. However, in our work, we particularly focus on the embedded domain where ARM and RISC-V architectures remain popular.

The ability to run different applications in parallel on shared hardware requires robust isolation. Virtualization technologies address this challenge by decoupling software environments, encapsulating them into Virtual Machines (VMs) that share physical resources without interfering with one another. Among these solutions, static partitioning hypervisors have emerged as a promising approach, providing strict resource partitioning to ensure predictable and isolated execution environments. However, ensuring strong isolation in such diverse and dynamic environments is non-trivial, as performance interference can still arise from shared system components—such as caches and memory bandwidth—that lie outside the strict partitions.

Understanding and mitigating these effects requires effective performance monitoring frameworks. A Performance Monitoring Unit (PMU) provides the means to gather detailed

runtime data, aiding in identifying bottlenecks, guiding isolation policies, and verifying performance targets. Out of two widely used embedded microprocessor architectures, ARM implements matured performance monitoring capabilities, while the RISC-V ecosystem lags behind in terms of comparable functionalities. Moreover, as SoCs grow increasingly heterogeneous, it becomes important to aggregate events across the entire system—not just at the core level—to achieve a holistic understanding of resource utilization. Prior work introduced the concept of an Advanced Performance Monitoring Unit (APMU) [23] to address these needs, but a comprehensive software framework, including tooling and integration support, remains absent.

We focus our efforts on isolation and performance monitoring on both ARM and RISC-V platforms. By integrating it within a static partitioning hypervisor environment, we can enable next-generation virtualization scenarios that balance isolation, flexibility, predictability, and efficiency.

1.2 Research Objectives

This thesis addresses the challenges of virtualization, resource contention, and system-wide performance monitoring in modern SoCs by pursuing the following key objectives:

1. Bandwidth Regulation for Static Partitioning Hypervisor (Bao):

Develop a bandwidth regulation mechanism within the Bao hypervisor [35]. This mechanism aims to ensure predictable resource utilization and mitigate performance interference, thereby improving isolation and determinism in multicore virtualized environments.

2. Software Infrastructure for Advanced Performance Monitoring Unit (APMU):

Establish a comprehensive software support stack for the APMU which can assist software development on the APMU and its integration within a static partitioning hypervisor, Bao. This infrastructure should facilitate event aggregation and monitoring across the entire SoC, taking actions based on events observed to enable a deeper understanding of system behavior and enhance the performance analysis capabilities.

1.3 Contributions

This thesis makes several key contributions to the field of isolation and performance monitoring in virtualized heterogeneous multicore SoCs:

1. **Memory Bandwidth Contention and Regulation:** Investigated and implemented MemGuard [56] in Bao Hypervisor on an ARMv8-based multicore SoC. Through experiments using San Diego Vision Benchmarks [50], the study quantifies MemGuard’s impact on reducing memory bandwidth interference between VMs.
2. **Advanced Performance Monitoring Unit (APMU) Software Infrastructure:** Developed a comprehensive software solution for the APMU, including a customized LLVM-based compiler, event-driven scheduling mechanisms, communication and runtime libraries.
3. **APMU-Hypervisor Integration:** Integrated the APMU framework into a hypervisor-based system, providing controlled and restricted APMU access to VMs. This includes implementing communication drivers for both the hypervisor and guest OSes, and creating a dynamic loader capable of installing functions targeted to APMU at runtime. This integration paves the way for adaptive, runtime monitoring strategies that maintain isolation and security.

1.4 Thesis Organization

The remainder of this thesis is structured as follows:

1. Chapter 2 (Background) presents foundational concepts related to virtualization, isolation, and performance monitoring. It introduces memory bandwidth isolation techniques and hardware performance monitoring solutions, focusing on the challenges and state-of-the-art approaches in the literature. This chapter also provides background on the APMU hardware architecture and LLVM-based compiler infrastructure, serving as a reference point for the subsequent chapters.
2. Chapter 3 (Memory Bandwidth Contention and Isolation) examines the implementation of MemGuard within Bao hypervisor on an ARMv8-based SoC. It measures the impact of memory bandwidth isolation on system performance using representative benchmarks, thereby connecting the theoretical background of Chapter 2 with implementation and empirical results.

3. Chapter 4 (APMU Software Infrastructure) details the development of a complete software support stack for the APMU, including compiler toolchains, runtime support in the APMU core, hypervisor integration, and guest OS drivers. This chapter extends the performance monitoring concepts introduced in Chapter 2 and provides a practical framework that can be used to gather, analyze, and act upon performance data in real time.
4. Chapter 5 (Conclusion and Future Work) summarizes the key findings, discussing their implications in virtualized multicore systems. It highlights areas for potential improvements and future research directions, building on the methods and insights developed throughout the thesis.

1.5 Acknowledgments

I would like to thank Mohammed Sajjad Jaffri for his invaluable contributions to hardware development and simulation in the RISC-V SoC project, including the development and integration of the APMU hardware and its corresponding event units [23]. His foundational work has been instrumental in enabling the research presented in this thesis. I am also deeply grateful to my research supervisor for their guidance, support, and insightful feedback throughout the course of this work.

Chapter 2

Background

In this chapter, we provide the foundational background and review related work that supports the contributions of this thesis. We begin the background on isolation in virtualized environment with an overview of virtualization fundamentals, focusing on hardware support for virtualization in ARMv8 and RISC-V architectures, in Section 2.1. This is followed by an exploration of hypervisors in Section 2.2, emphasizing static partitioning hypervisors that statically partition system resources among Virtual Machines, thus providing some degree of isolation.

Next, we review Performance Monitoring Architectures, which provide insights into system interactions central to temporal isolation, in Section 2.3. Then we delve into the domain of temporal isolation, examining popular techniques and their limitations in Section 2.4. This is complemented by a discussion of the design and implementation of a Performance Monitoring Unit from prior work in Section 2.5. Finally, we conclude with a review of key LLVM concepts, which are leveraged to develop the toolchain for the Performance Monitoring Unit, in Section 2.6.

2.1 Virtualization

Virtualization technology enables multiple virtual machines (VMs) to run simultaneously on shared physical hardware, abstracting resources such as CPUs, memory, and I/O devices into virtualized counterparts. It enhances flexibility, scalability, and resource utilization in applications ranging from cloud computing to embedded systems. In mixed-criticality systems, virtualization ensures that safety-critical real-time tasks can coexist with less

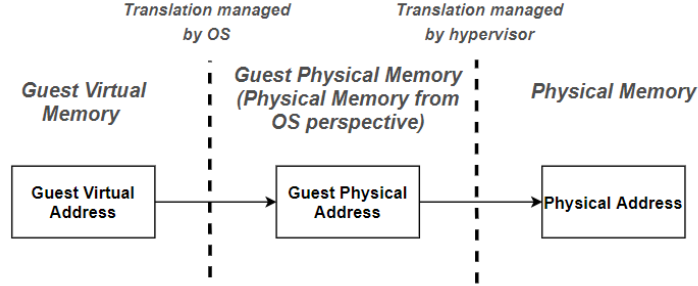


Figure 2.1: Two stage address translation

critical workloads without sacrificing determinism or reliability [11]. VMs provide spatial isolation, improving security and fault tolerance while reducing size, weight, and power (SWaP) constraints in domains like aerospace, automotive, and IoT. Modern architectures support virtualization with specialized instructions, privileged execution levels, and two-stage address translation, enabling secure and efficient isolation among VMs.

2.1.1 Two-Stage Address Translation

Two-stage address translation is essential for virtualization, introducing an additional level of address mapping to ensure memory isolation between virtual machines (VMs). In a virtualized environment, the guest operating system (OS) translates virtual addresses (VAs) to guest physical addresses (GPAs) using its own page tables (stage 1 translation). However, these GPAs are not the actual physical addresses; instead, the virtual machine monitor (also known as the hypervisor) maps GPAs to real physical addresses (PAs) using a second set of page tables (stage 2 translation). A visual representation of two stage address translation is depicted in Figure 2.1

The Memory Management Unit (MMU) performs these translations sequentially: VAs are first resolved to GPAs by the guest OS, then GPAs are mapped to PAs by the hypervisor. This layered translation ensures the guest OS can only access memory within its assigned range, while the hypervisor retains full control over physical memory. Modern architectures enhance this process with hardware-assisted two-stage translation, including virtualization-aware Translation Lookaside Buffers (TLBs) that cache intermediate address translations. These optimizations minimize performance overhead while enforcing strict memory isolation, enabling secure and efficient execution of multiple VMs on shared physical hardware.

ARMv8 and RISC-V architectures provide robust hardware support for virtualization, leveraging two-stage address translation, privilege separation, and interrupt virtualization to ensure efficient and secure execution of virtual machines (VMs). Both architectures also support interrupt virtualization, enabling efficient trapping and delivery of interrupts to guest environments, reducing overhead and improving performance. These capabilities simplify hypervisor design and enhance the isolation and scalability required for modern virtualized systems.

The ARMv8 architecture implements virtualization through a hierarchical exception level model, including EL0 for user applications, EL1 for the operating system kernel, and EL2, a dedicated hypervisor mode [32]. EL2 provides privileged control to manage guest VM transitions and enforce isolation. RISC-V introduces virtualization through its Hypervisor Extension (H-extension) [15]. This extension adds Hypervisor Supervisor Mode (HS-mode), allowing the hypervisor to manage guest execution. Guest environments operate in Virtual Supervisor Mode (VS-mode) and Virtual User Mode (VU-mode), enabling efficient virtualization of both operating systems and applications. While RISC-V’s virtualization features are still maturing, its modularity and open-source nature make it particularly suited for research, embedded systems, and specialized workloads such as FPGA-based implementations.

2.2 Hypervisors

A hypervisor, or Virtual Machine Monitor (VMM), manages Virtual Machines (VMs) by abstracting physical hardware into virtualized resources, providing each VM with an isolated execution environment. It mediates access to hardware, manages memory translation, and handles device I/O and interrupt routing to ensure secure and efficient resource utilization.

Hypervisor are broadly categorized in two types. Type-1 or baremetal hypervisors run directly on the hardware, without requiring a host operating system. By minimizing the software stack between VMs and the physical platform, Type-1 hypervisors can achieve low latency, reduced overhead, and high performance. They are therefore well-suited to real-time systems, embedded devices, and other performance-critical workloads. Examples include Xen [40], Xvisor [36], Jailhouse [1], Bao [35], and ACRN [26]. Figure 2.2 represents a typical type-1 hypervisor based system with multiple Virtual Machines including Rich OS (e.g. Linux), RTOS and Baremetal VMs. Type-2 or hosted hypervisors run on top of a general-purpose operating system. While this approach provides greater flexibility,

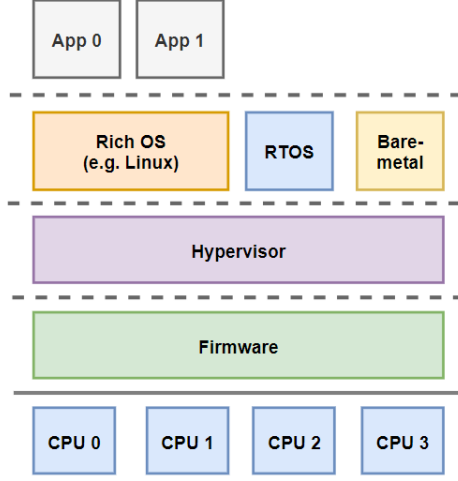


Figure 2.2: Virtualized system with hypervisor

it introduces additional overhead due to the host OS layer. Consequently, Type-2 hypervisors are more common in development, desktop, and testing environments, where peak performance or stringent real-time constraints are less critical. Examples include KVM [19], VMware Workstation [51], and VirtualBox.

2.2.1 Static Partitioning

In real-time and safety-critical domains, ensuring deterministic performance and predictability is paramount. Static partitioning hypervisors address this need by deterministically allocating system resources—such as CPU cores, devices and memory—at configuration time. This approach guarantees that each VM receives a pre-defined share of resources, mitigating the dynamic contention that may occur in scheduling-based hypervisors. The allocation of devices and memory is supported by two-stage translation, where the hypervisor is responsible for mapping the devices and memory regions from the actual physical address space to the respective VM’s guest physical address space. By eliminating the complexity of runtime allocation, static partitioning helps reduce jitter and latency, contributing to more predictable system behavior.

However, while static partitioning effectively isolates CPUs, I/O devices and memory regions, it cannot completely ensure full isolation. On modern SoCs, VMs still compete for shared hardware resources, such as caches and memory bandwidth. This contention

can degrade performance and introduce unpredictability, even in carefully designed static partitioning environments. As such, static partitioning hypervisors alone, without additional isolation mechanisms, may not be sufficient to guarantee the strong isolation and determinism required by certain high-integrity or mixed-criticality applications.

In the embedded and real-time domains, lightweight Type-1 hypervisors with static partitioning have gained prominence due to their minimal Trusted Computing Base (TCB), reduced complexity, and stringent focus on determinism. Examples include:

- *Bao Hypervisor*: Bao is a Type-1 hypervisor tailored for ARM and RISC-V platforms. It adopts a static partitioning approach and offers advanced isolation features such as cache coloring. With its small TCB and emphasis on real-time guarantees, Bao is well-suited to safety-critical embedded applications.
- *Jailhouse*: Siemens’s Jailhouse hypervisor pioneered the static partitioning approach in the embedded domain. It runs alongside Linux initially, then transitions to a bare-metal mode to statically partition CPU cores and memory among VMs. This architecture reduces complexity and ensures deterministic behavior.
- *Xvisor*: Xvisor is an open-source Type-1 hypervisor designed to offer minimal overhead and strong isolation. Its compatibility with ARM and RISC-V architectures makes it suitable for research, academia, and embedded systems.
- *ACRN*: Developed by Intel, ACRN [26] targets the IoT segment with a lightweight design and static partitioning principles, optimizing virtualization performance for embedded use cases.

The growing adoption of such hypervisors in automotive, aerospace, and other embedded sectors reflects the industry’s need for secure, high-performance, and deterministic virtualization.

2.2.2 Bao Hypervisor

The Bao hypervisor is a type-1 hypervisor, which targets better isolation features. One of its standout features is its use of two-stage page translation for implementing cache coloring. This technique partitions the last-level cache (LLC) to reduce interference between virtual machines (VMs), thereby improving determinism and real-time performance (discussed further in Section 2.4.1). Furthermore, unlike Jailhouse hypervisor, it does not rely on

Table 2.1: Hypervisors and Supported Architectures

Hypervisor	Type	ARMv8-A Support	RISC-V Support
Bao Hypervisor	Type-1	Yes	Yes
Xen Project	Type-1	Yes	Yes
ACRN	Type-1	Yes	No
Xvisor	Type-1	Yes	Yes
ARM Trusted Firmware	Type-1	Yes	No
KVM	Type-2	Yes	Yes
Jailhouse	Type-1	Yes	Yes

Linux to load itself, and is supported on variety of COTS systems, which makes Bao a suitable choice for virtualized embedded systems.

Bao’s interrupt handling mechanism is flexible, allowing it to either delegate interrupts and exceptions directly to VMs or manage them within the hypervisor. However, it does not support nested interrupts, simplifying its design and reducing overheads in time-critical scenarios. Hypercalls are available to facilitate communication between VMs and the hypervisor, enabling efficient operation of complex virtualized environments.

Bao relies on two separate configuration files: one tailored to the specific platform, describing hardware resources, and another that defines the properties and allocations for each VM. Through these configurations, Bao enables static allocation of memory regions and devices to VMs, ensuring deterministic resource usage. Details of these configurations are given in Appendix A. Additionally, it supports shared memory regions between VMs, fostering inter-VM communication while maintaining isolation.

This meticulous resource partitioning and isolation make Bao particularly suited for safety-critical and mixed-criticality embedded systems, where robust fault containment and performance predictability are paramount.

2.3 Performance Monitoring

As systems become increasingly heterogeneous and complex with various hardware components, such as accelerators, specialized communication fabrics etc., runtime performance

monitoring has become essential for understanding how applications utilize these hardware resources. This insight helps identify bottlenecks, inefficiencies, and anomalies, which is particularly critical in a virtualized environment when multiple Virtual Machines (VMs) share a platform. Performance monitoring enables the detection of interference sources and supports the enforcement or validation of resource isolation strategies.

In the real-time domain, memory bandwidth regulation techniques such as MemGuard [56], MemPol [58], and Latency Distribution Driven Regulation [43] leverage performance counters to measure and control memory bandwidth usage. Similarly, in security applications, performance counters are used for tasks such as verifying Control Flow Integrity [10] and detecting denial-of-service attacks [7].

In cloud environments, performance counters have been employed to enable fair pricing in HPC clusters [9], ensure high quality-of-service (QoS) for latency-sensitive applications [55], and estimate interference in shared servers [24]. These examples illustrate how performance monitoring extends beyond isolation enforcement to support workload management and resource optimization. As platforms integrate diverse processing elements like CPUs, GPUs, and accelerators, and operate within distributed systems, performance monitoring becomes critical for understanding complex component interactions. It can provide granular measurements that guide memory partitioning, and bandwidth enforcement, ensuring fair resource sharing, predictable execution times, and comprehensive system-level visibility. We specifically look at Performance Monitoring hardware support in ARM and RISC-V architectures.

ARM processors incorporate a Performance Monitoring Unit (PMU) [6], which provides a standardized interface for counting hardware events such as cycles, instructions, cache hits/misses, and branch mispredictions. The PMU supports interrupts triggered by counter overflows, enabling efficient performance profiling with minimal overhead. In addition to PMUs, ARM introduces Memory Partitioning and Monitoring (MPAM) [5], a framework for managing memory resources like caches and memory interfaces. MPAM offers resource models, priority controls, and usage metrics, enabling fine-grained memory isolation and monitoring. Together, PMU and MPAM provide a comprehensive mechanism for profiling and managing system performance, helping developers identify bottlenecks and optimize memory resource allocation.

RISC-V’s modular architecture supports flexible hardware performance counters, with a baseline including counters for retired instructions and clock cycles as specified in the ISA [15]. The architecture allows for up to 29 configurable event counters, enabling system designers to monitor custom hardware events, such as cache misses or accelerator-specific operations, via Control and Status Registers (CSRs). These counters facilitate targeted

performance analysis, providing system architects with tools to optimize hardware and software for enhanced isolation and predictability. However, the RISC-V performance monitoring infrastructure is less mature than that of modern processors, with limitations in both specification and implementation.

2.3.1 Software Frameworks for Performance Monitoring

The `perf` subsystem in the Linux kernel [37] provides a unified interface and user-space utilities for gathering both hardware and software event data. By communicating with vendor-specific PMU drivers, `perf` can sample metrics such as cache misses, branch mispredictions, and instruction retirements. Its command-line tools allow event configuration and data collection.

Another framework called the Performance API (PAPI) [13] offers a platform-independent layer for accessing hardware performance counters, abstracting away architecture-specific details. Through the defined API, applications can instrument performance events without manually handling low-level counter registers. In comparison to `perf`, PAPI’s design is more portable across diverse platforms, while still supporting both basic measurements (e.g., cycle counts, cache events) and complex multi-event profiling.

2.4 Temporal Isolation

Achieving stronger temporal isolation often hinges on controlling the shared resources that can cause contention. Specifically, within the memory subsystem, managing the last-level cache (LLC) and main memory bandwidth is crucial for preventing any single entity from monopolizing these resources and degrading overall system predictability, as seen in [56, 58, 16]. In virtualized systems, logical and temporal isolation is often an objective. By applying fine-grained isolation techniques, as we discuss ahead, designers can mitigate interference and ensure that critical workloads meet their performance and timing requirements.

2.4.1 Cache Partitioning

Cache partitioning is a technique used to divide a shared cache into distinct regions that can be allocated to different processes or applications within the OS. These techniques comprise of hardware and software based approaches [52, 27, 22, 20]. Cache coloring is a hardware-software based technique that aims to partition the shared cache or LLC into logically

distinct regions, or “colors,” that can be allocated to different processes. Modern processors often implement the LLC as a set-associative cache, where memory addresses are mapped to specific cache sets based on a subset of the physical address bits. If multiple processes share the same cache sets, they compete for space, causing performance interference and unpredictability. Cache coloring directly addresses this issue by ensuring that the physical memory allocated to each process maps to a disjoint subset of cache sets.

Earliest adaptation of coloring mechanisms was used at the operating system level to mitigate contention in cache ([8], [25]). This technique was later adopted to reduce contention among processes and improve fairness in multi-core systems [46]. Follow-up approaches introduced the concept of hot-page coloring, where online mechanism decides ‘hotness’ of the page and allocates colors accordingly [57]. Up until now, research mostly focused on utilizing the coloring mechanisms within the OS but similar mechanisms can be employed at the hypervisor level as well. For example, [35] implements cache coloring through two-stage address translation.

The core idea arises from the observation that the index bits used to determine a cache set are derived from a portion of the physical address. By carefully controlling the distribution of physical pages—specifically, by selecting pages whose addresses differ in the cache set index bits—system software (e.g., a hypervisor or an OS) can effectively ensure that distinct applications or VMs (for OS or hypervisor respectively) do not overlap in the same cache sets. This approach translates into more deterministic cache usage patterns, as each VM’s data occupies unique regions of the LLC.

Cache coloring involves three key steps: determining the number of colors, page allocation, and enforcement. A “color” corresponds to a group of pages mapping to the same subset of cache sets, and the number of colors depends on the LLC’s size and associativity. During page allocation, the hypervisor or OS assigns physical pages to VMs from pre-segmented colored page pools, ensuring each process or a VM maps to specific cache sets. Enforcement is typically performed in software, where the memory allocator tracks and assigns colors. While this process may introduce some overhead, the predictable performance gains often justify the complexity, particularly in embedded or safety-critical contexts. Moreover, cache coloring complements static partitioning hypervisors, which allocate CPU and memory resources but lack inherent LLC control. By adding cache coloring, system architects achieve greater isolation and predictability through more fine-grained control over resource usage.

2.4.2 DRAM Bank Partitioning

While cache coloring addresses contention in the LLC, memory interference can also arise at the DRAM level. Modern DRAM subsystems consist of multiple banks that can be accessed in parallel. However, contention occurs when multiple VMs or cores access the same DRAM bank, leading to closing each other’s rows and thus, inducing overhead and unpredictable latencies.

PALLOCC [16] (short for “Page ALLOCation”) provides a software-based approach to DRAM bank partitioning by leveraging the DRAM controller’s address mapping. Physical addresses are mapped to specific DRAM banks based on the controller’s design. By carefully limiting the pages allocated to a VM to a subset of physical addresses that map to distinct DRAM banks, interference can be mitigated, and VMs can achieve isolated DRAM Bank access.

Similar to cache coloring, this approach ensures that VMs operate on separate regions of a shared resource, enhancing predictability and isolation. While DRAM bank partitioning may reduce total available memory for each VM, this trade-off is acceptable in real-time or safety-critical applications where deterministic performance is a priority. Combined with cache coloring, DRAM bank partitioning provides a complementary mechanism for achieving fine-grained memory isolation.

2.4.3 Memory Bandwidth Partitioning

Even when CPU cores and memory regions are statically allocated, the memory interconnect remains a shared and finite resource. Multiple VMs or tasks competing for memory bandwidth can experience unpredictable latency and reduced throughput, which is particularly problematic in real-time or mixed-criticality systems. To address this issue, a range of memory bandwidth partitioning techniques have been developed. These techniques regulate each VM’s memory traffic, ensuring that no single VM dominates the bandwidth and degrades the performance of others. By enforcing bandwidth policies and reservations, system architects can improve predictability and maintain fairness across co-located workloads.

MemGuard [56] is a representative approach designed to achieve predictable memory performance by regulating memory bandwidth usage on multi-core platforms at OS level. It introduces a mechanism that assigns a predefined memory bandwidth budget to each core, and does not allow the core to exceed that budget, ensuring that each core receives a guaranteed share of the available bandwidth over a defined time window.

The operation of MemGuard is summarized here. A fixed regulation period P is established. Within each period, memory bandwidth usage is monitored and controlled. Each core i is assigned a memory bandwidth budget Q_i . This budget represents the maximum allowable memory consumption for that core per period. If a core exceeds its allocated budget Q_i during the period P , MemGuard prevents it from issuing further memory requests until the next period by halting the core through available means such as busy wait in interrupts or other hardware supported mechanisms. This enforcement ensures that no core can monopolize memory bandwidth. At the end of each period, all budgets are replenished, allowing each core to again utilize its allocated share in the subsequent interval.

Memguard also proposes a reclamation mechanism which is capable of redistributing bandwidth dynamically to efficiently utilize the memory bandwidth. However, this reclamation mechanism is prediction based and is suggested not to be utilized for hard real-time systems. It is important to note that Memguard performs regulation based on a period. Within one period, the individual requests may suffer contention leading to increased latency per request.

By combining static bandwidth reservations with dynamic monitoring and enforcement, MemGuard provides strong performance isolation in multi-core systems. It ensures that each core maintains predictable memory access times and prevents any single workload from causing undue interference. This approach thus offers a practical solution for systems that require stable and reliable memory performance under concurrency.

MemPol [58] improves on MemGuard by replacing interrupt-based throttling with a polling-based mechanism implemented outside the cores. Using on-chip debug facilities like ARM CoreSight [4] and a monitoring core, it monitors performance counters at microsecond intervals without disrupting application execution, significantly reducing overhead while enabling finer control over memory bandwidth. MemPol also integrates multiple metrics, such as cache refills and write-backs, for more accurate quota enforcement. Its sliding window mechanism smooths out transient memory bursts, and it supports both per-core and global regulation policies. These refinements—finer granularity, lower overhead, and multi-metric monitoring—make MemPol a more precise and efficient solution for managing memory bandwidth in multi-core systems. However, since MemPol relies on the presence of an external monitoring core, its implementation is limited to systems equipped with such capabilities.

While the focus for the aforementioned temporal isolation techniques has been directed towards operating systems, we employ these techniques at the hypervisor level in conjunction with each other to determine their effectiveness in a virtualized multicore system.

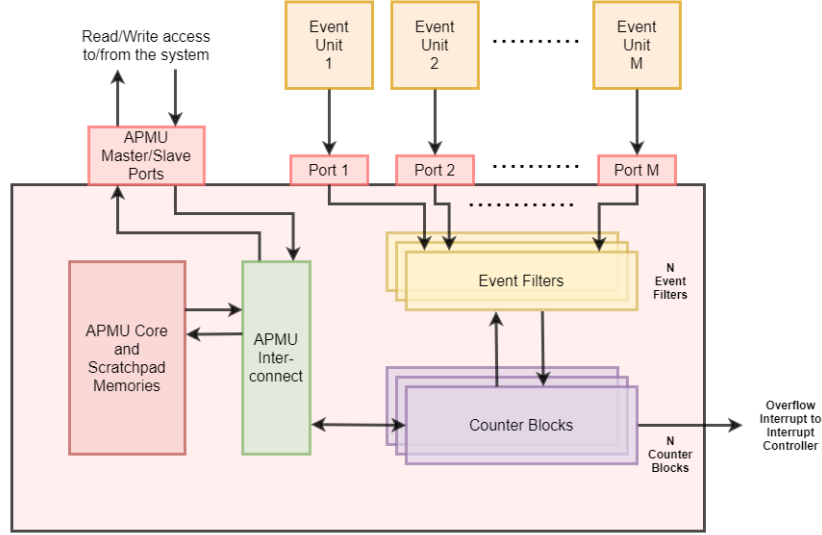


Figure 2.3: APMU Block Diagram

2.5 Advanced Performance Monitoring Unit

The Advanced Performance Monitoring Unit (APMU) provides a centralized infrastructure for advanced performance monitoring and event-driven operations. Its design and an FPGA-based prototype were developed in earlier work [23]. Figure 2.3 shows the APMU block diagram. This section focuses on the APMU’s core components and their implementation.

2.5.1 Event Units (EVUs)

Event Units (EVUs) are distributed hardware modules integrated into system components such as processors, memory controllers, and caches. They observe microarchitectural events and send event packets with associated metadata to the APMU Counters via standardized EVU-APMU ports. Since EVUs are hardware-level units, they introduce negligible overhead to the monitored components. Each EVU includes event tables that define the events and corresponding metadata.

2.5.2 APMU Counters and Event Processing

APMU Counters form the core of the monitoring infrastructure, supporting arithmetic and logical operations on event metadata through an internal ALU. They can operate in count mode (tracking event frequencies) or functional mode (performing specified operations). Event Filter blocks connect EVUs to counters, allowing selective event monitoring. Each counter maintains a pending bit and an overflow bit, which signal updates or overflows, respectively.

Configuration Registers define which events a counter monitors and how the ALU processes their metadata. The Event Selection Register specifies the target events, while the Event Info Register controls ALU operations and enables optional overflow-triggered interrupts.

2.5.3 APMU Core

The APMU Core is a dedicated processing unit that can execute event-based mechanisms. It gathers data from counters using custom instructions and can take actions based on observed events. The core has dedicated Instruction Scratchpad Memory (ISPM) for code and Data Scratchpad Memory (DSPM) for runtime data such as profiling information and communication buffers. The ISPM supports dynamic reprogramming, allowing the APMU Core to adapt its behavior as system requirements change.

The APMU Core is designed as a low-power, embedded-style processor. In addition to standard embedded features, it supports runtime reprogrammability and can access system address spaces and APMU resources (registers, counters, and SPMs) through defined ports. Two registers—Boot Address Register and Status Register—allow resetting the core and specifying a boot address.

Custom instructions enable efficient interaction with counters for fast read/write operations and event-driven synchronization. Counter read/write instructions directly access a specified counter index with minimal overhead. The wait-for-pending and wait-for-overflow instructions are designed for low-power event waiting based on hardware counter states. These instructions rely on a 32-bit bitmask to identify the counters of interest, where each bit corresponds to a specific counter index. The wait-for-pending instruction suspends execution until any of the specified counters are updated, indicated by a pending bit being set. Similarly, the wait-for-overflow instruction suspends execution until any of the specified counters overflow. In both cases, the processor enters a low-power state while waiting for the specified condition, and upon resumption, a bitmask is returned indicating which

counter(s) triggered the return event. This supports an efficient and low-power design where simultaneous monitoring of multiple counters can be performed using the bitmask.

2.5.4 Implementation

The APMU is integrated with a RISC-V based core and a dedicated Event Unit (EVU) to facilitate flexible and fine-grained performance monitoring. The core utilized in the APMU is an IBEX RISC-V32 core [31], chosen for its low-power and area-efficient design. IBEX supports the RISC-V integer (I), multiplication (M), and compressed (C) extensions, providing a balance of performance and code density suitable for event-driven monitoring tasks.

A key component of the APMU’s monitoring capabilities is the EVU, which is able to snoop system events on the AXI bus. The EVU can detect and measure various events, including read requests, write requests, read responses, and write responses, providing crucial insights into memory and interconnect behavior. Additionally, the EVU can record metadata, such as event counts or associated latencies, enabling more detailed analysis of system interactions. This data is transmitted to the APMU’s counters via standardized EVU-APMU ports, where it can be processed and aggregated to guide runtime decisions.

To enable direct interaction between software and the APMU’s counters and event-driven synchronization mechanisms, four custom RISC-V instructions are implemented. In general, RISC-V instructions typically follow a straightforward assembly format, for example:

```
<instr_mnemonic> rd, rs1, rs2 ...
```

Here, `rd` denotes the destination register, while `rs1` and `rs2` represent source registers. As illustrated in Figure 2.4, the ISA defines several common instruction formats. The implemented `cnt.rd`, `cnt.wr`, `cnt.wfp`, and `cnt.wfo` instructions follow I-type, S-type and R-type encodings respectively. Following the RISC-V extension conventions, these instructions are encoded with a common OpCode (7'b0000111) and differentiated by their `funct3` and `funct7` fields. The `cnt.rd` and `cnt.wr` instructions provide fast, low-latency access to read or write counter values. Meanwhile, `cnt.wfp` (Wait-for-Pending) and `cnt.wfo` (Wait-for-Overflow) support event-driven synchronization by suspending execution until specified counters are updated or overflow, respectively. During this wait, the processor can enter a low-power state, resuming execution once an event occurs. Upon resumption, the triggered counters are identified via a returned bitmask.

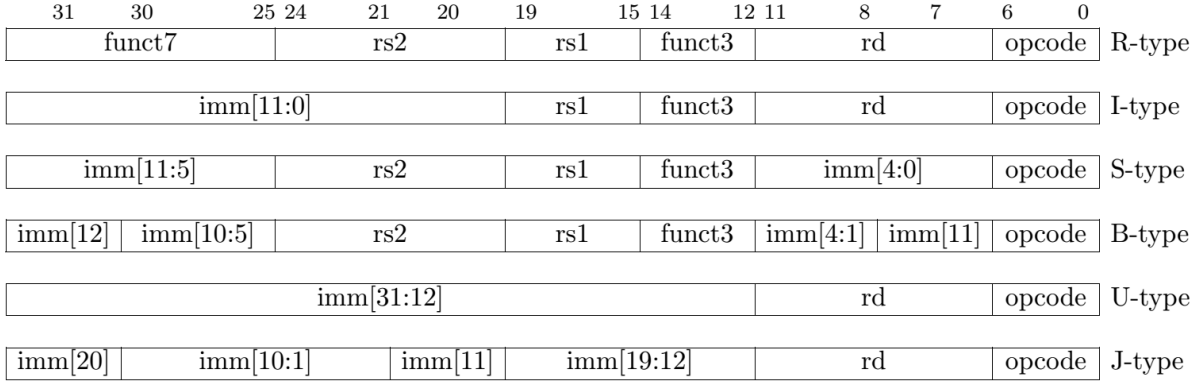


Figure 2.4: Common RISC-V Instruction Formats from Integer Extension

By combining a low-power RISC-V IBEX core, a configurable EVU capable of snooping AXI bus events, and custom instructions for efficient counter interaction, the APMU implementation provides a flexible and extensible platform for runtime performance monitoring and event-based control. This architecture enables precise observation of system behavior—spanning from memory traffic patterns to timing-sensitive event occurrences—and facilitates automated, low-overhead responses to maintain predictability and optimize resource usage. The detailed instruction encoding is illustrated in Figure 2.5.

The APMU is integrated in a RISC-V SoC Project with based on RISC-V64 CVA6 cores which supports hardware hypervisor extension ([48] [15] [30]). The system was modified to include 4 CVA6 cores and the APMU was integrated in the system. The design was implemented on VCU118 FPGA board in a prior work [23] and the overall block diagram is shown in Figure 2.6. The SoC features L1D cache of 32KB, LLC of 1MB and a DDR4 Main Memory of 4GB and various peripherals such as I2C/SPI controllers as well as UART. The AXI4 EVUs were placed between all the cores and the LLC and after the LLC.

The platform supports boot-up with a privileged software stack, which includes M-Mode firmware, OpenSBI. OpenSBI operates to provide foundational services for bootstrapping and runtime management. For hypervisor-enabled systems, OpenSBI performs critical platform initialization steps and uses `medeleg` and `mideleg` to route interrupts and traps to the hypervisor. Bao hypervisor, as HS-Mode software, is deployed to manage resources and enable strong isolation between Virtual Machines (VMs). Bao leverages the services provided by OpenSBI and integrates seamlessly with the underlying hardware setup. The bootup process of this software stack includes packaging of all the VMs into Bao Hypervisor binary, which in turn is packaged into the binary of OpenSBI as a payload. OpenSBI is

31	30	25	24	21	20	19	15	14	12	11	7	6	0
0	0	0	0	rs1	3'b000	rd	opcode						

(a)

31	30	25	24	20	19	15	14	12	11	8	7	6	0
0	0	rs2	rs1	3'b001	0	0	opcode						

(b)

31	25	24	20	19	15	14	12	11	7	6	0
7'b000_0000	0	rs1	3'b010	rd	opcode						

(c)

31	25	24	20	19	15	14	12	11	7	6	0
7'b000_0001	0	rs1	3'b010	rd	opcode						

(d)

Figure 2.5: Counter Instruction Formats. Figure (a) shows encoding of counter read instruction following I-type format. Figure (b) shows encoding of coutner write instruction following S-type format. Figure (c) and (d) respectively show encoding of wait-for-pending and wait-for-overflow instructions following R-type format.

responsible to bootup and initialize system components and yield to Bao Hypervisor, which in turn performs necessary initializations of virtual memory and other components and runs the VMs according to their specific configuration.

2.6 LLVM Overview

The integration of custom instructions for the APMU requires extending the compiler toolchain, enabling seamless generation of code that leverages the new hardware capabilities. LLVM, an open-source and modular compiler infrastructure, is well-suited for this task due to its mature RISC-V support and highly flexible structure. By incorporating the APMU's custom instructions into LLVM, developers can efficiently generate optimized machine code that fully exploits the performance monitoring features provided by the APMU.

LLVM operates by translating high-level source code into an Intermediate Represen-

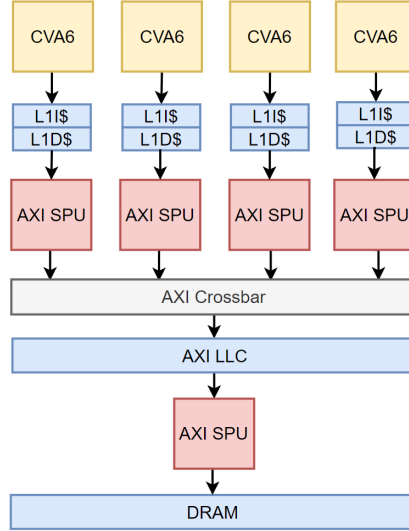


Figure 2.6: APMU Block Diagram

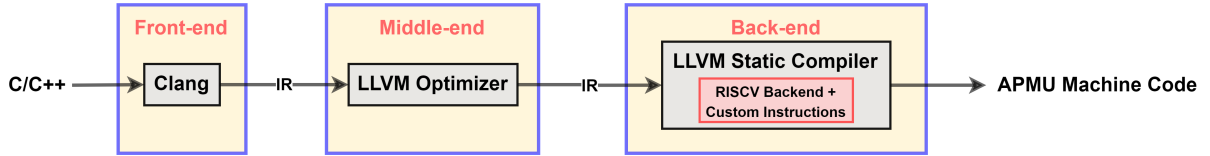


Figure 2.7: Stages of LLVM Based Compilation

tation (IR) that serves as a common platform for language-agnostic optimizations and target-specific transformations. The compilation process broadly divides into three stages: the Front-End (e.g., Clang), which converts source code into LLVM IR; the Middle-End, which applies various optimizations on the IR; and the Back-End, which generates target-specific machine code. RISC-V support in LLVM is well-established, enabling developers to easily adapt and enhance the compilation process for their custom hardware modifications, as illustrated in Figure 2.7.

2.6.1 TableGen Reference

TableGen is a key component of LLVM that streamlines the definition of target-specific instructions, registers, and other architectural properties [29]. Using a declarative syntax, developers can define large sets of related entities, which TableGen processes to produce

the necessary C++ code for LLVM’s code generator. For the APMU, this means custom instructions can be cleanly integrated into the RISC-V backend’s TableGen (.td) files, ensuring that the new operations seamlessly fit into the existing compilation flow.

2.6.2 LLVM Backend

LLVM Backend converts optimized LLVM IR into machine code using various libraries and modules. The RISC-V Backend operates primarily on a Directed Acyclic Graph (DAG) of IR or instructions. Key phases of LLVM RISC-V Backend, excluding the various optimization phases are [28]:

1. **DAG Generation:** Converts IR from the Middle-End into a SelectionDAG. This DAG may include illegal operations for the target architecture, such as 64-bit operations on a 32-bit target.
2. **DAG Legalization:** Transforms illegal operations in the SelectionDAG into legal ones, e.g., breaking a 64-bit operation into multiple 32-bit operations.
3. **Target Instruction Selection:** Replaces instructions in the SelectionDAG with target-specific ones using TableGen description files.
4. **Scheduling and Formation:** Orders instructions in the DAG according to target constraints, then converts the DAG into a list of MCInstr objects representing target machine instructions.
5. **Register Allocation:** Maps virtual registers in MCInstr objects to physical registers.
6. **Code Emission:** Converts assembly instructions into final binary machine code.

By integrating the APMU instructions into this pipeline, developers can generate efficient and event-driven code that leverages the APMU’s hardware counters, events, and synchronization primitives.

Chapter 3

Memory Isolation in ARM Cortex-A53

In this section, we take a closer look at achieving bandwidth isolation in a virtualized commercial off-the-shelf (COTS) system based on the ARMv8 architecture. Although hypervisors like Bao can statically partition system resources—such as CPUs, memory, and I/O—virtual machines (VMs) still run without strict isolation guarantees. Our primary focus is on the problem of memory bandwidth contention and the use of bandwidth regulation techniques to mitigate it.

We conduct our evaluation on the AMD Xilinx ZCU102 System-on-Chip (SoC), which incorporates four ARM Cortex-A53 cores supporting ARMv8-A virtualization extensions and 32KB of L1D cache and 1MB of L2 cache. The virtualization extension enables the execution of both complex operating systems and simple bare-metal applications as virtual machines [3]. The ZCU102 also implements the ARM PMUv3 specification, a crucial feature for enabling and assessing memguard—a memory bandwidth regulation methods we investigate [56].

Bao [35], a static partitioning hypervisor, natively supports the ZCU102 platform. Bao allows up to four VMs, in which case it would be one VM per core, and provides cache partitioning through Cache Coloring. However, despite static partitioning and cache coloring, VMs can still experience performance interference through the memory subsystem, where the limited bandwidth of the memory hierarchy introduces contention, as discussed in Section 2.4. In this chapter, we explore how that contention looks like and implement Memguard on Bao Hypervisor to provide bandwidth isolation at hypervisor level. Our goal is to minimize contention suffered by interference from other cores and to see how regulation and isolation parameters affect performance.

3.1 Memory Geometry and Cache Coloring

Before delving into the bandwidth contention and isolation, we first examine in detail how the number of available cache colors is determined to be able to allocate these cache colors to different VMs. To determine the maximum number of cache colors available on the ZCU102 platform, we begin by examining its memory subsystem geometry. The relevant calculations are summarized in Table 3.1.

Cache coloring leverages the fact that specific address patterns map to specific cache sets, as discussed in Section 2.4.1. On an ARM Cortex-A53, the page size is 4KB, and consequently, out of the index bits, bits 11:6 overlap with the page bits (11:0) and cannot be used for coloring. As a result, we have four remaining bits (15:12) available for cache coloring. However, because our goal is to partition the L2 cache and the uppermost L1 index bit overlaps with these color bits, utilizing that bit would effectively prevent half of the L1 cache from being used. Thus, we are left with bits 15:13, giving us a total of 8 cache colors.

Parameter	L1 Cache	L2 Cache
Associativity	4 Way	16 Way
Cache Lines	512	16384
Number of Sets	128	1024
Index Bits	7	10
Offset Bits	6	6

Table 3.1: Cache Geometry Calculations for ZCU102

Beyond the cache, memory bandwidth contention can also arise from DDR memory access patterns. When multiple cores share the same DDR bank, they can interfere with each other’s bandwidth as introduced in Section 2.4.2. The ZCU102 Memory Controller IP offers two DDR addressing modes: `Row_Bank_Col` and `Bank_Row_Col`. In `Row_Bank_Col` mode, the Bank and Bank Group bits appear before the Row bits, allowing nearby addresses to map to different banks and bank groups. In contrast, `Bank_Row_Col` mode keeps the Bank and Bank Group bits fixed for large contiguous memory regions. This increases the contention for addresses that are close to each other, since, they would access different closed rows within the same bank.

For our use case, we choose the `Row_Bank_Col` addressing mode. This arrangement enables the Bank and Bank Group bits to overlap with the cache coloring bits. In practical scenarios, ensuring that cache colors overlap with Banks and Bank Groups means

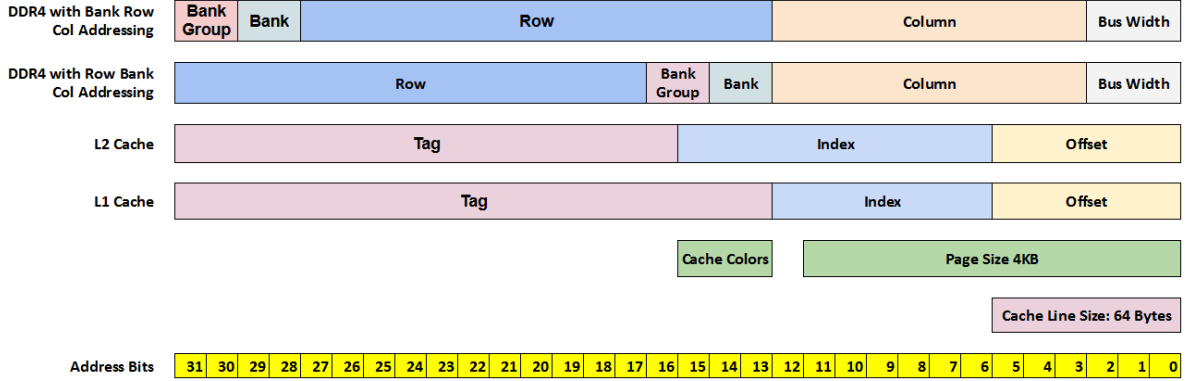


Figure 3.1: Memory geometry of ZCU102 with L1 Cache, L2 Cache, and two DDR4 Memory Controller addressing options

different colors (and thus different VMs) will not share the same bank. In contrast, the `Bank_Row_Col` option would assign the same Bank and Bank Group values to large contiguous memory regions. Figure 3.1 provides an overview of both addressing options and their relationship to the DDR4 memory controller’s address map and the cache geometry of ZCU102.

3.2 Evaluation of Shared Memory Contention

As introduced in Section 2.4.3, the memory subsystem offers limited bandwidth that must be shared among all entities that use it. Consequently, when multiple components compete for memory resources, the available bandwidth for the primary entity under analysis decreases, potentially increasing the execution time of memory-dependent operations.

In this section, we evaluate how concurrent reads and writes from interfering cores affect the memory bandwidth available to a particular core under analysis. The goal is to measure the increase in average latency per request when other cores are stressing the memory subsystem and the core under analysis is hitting or missing in the caches.

3.2.1 IsolBench

To systematically evaluate these contention in memory subsystem, researchers often rely on controlled benchmark suites. One such suite, IsolBench [49], provides a set of synthetic micro-benchmarks tailored for analyzing worst-case interference in shared memory systems. Our experimental methodology draws inspiration from IsolBench, but we have re-implemented it to run in a bare-metal environment. This adaptation was necessary due to IsolBench’s reliance on the Linux operating system. While Linux provides extensive functionality, it also introduces substantial overhead through background processes, interrupts, drivers, and system services, all of which can interfere with the tasks under execution. Such interference leads to increased variation in measured outcomes, reducing the reliability of the results in a real-time context.

Algorithm 1 illustrates the adapted bare-metal implementation. The core idea is to have one core repeatedly perform load or store operations on an array of size N , where N can be adjusted to target various memory levels (L1, L2 or Main Memory). The array is iterated by jumping over 64 bytes as this is the cache line size. By working at a cache-line granularity, we minimize the interference from cache lines that might have been brought in by other cache misses, ensuring more consistent measurements. To prevent GCC optimizations from optimizing out the loops that perform these operations, we implemented the loops using inline assembly blocks. Thus, we make sequential accesses to cache lines to stress the system memory bandwidth. It is important to note that changing the size of the array is a means to generate specific kinds of memory requests: requests that hit in L1, requests that miss in L1 cache and hit in L2 cache, and the requests that miss in L2 cache and go to main memory. As such, array size is irrelevant to our experiment except to produce specific hit and miss events in cache, which can be caused due to a variety of reasons in different kinds of applications.

To ensure a consistent initial cache state, we perform a *Cache Priming* loop before the main measurement loop. In the Main Loop itself, we measure the number of cycles required for each load or store operation and report the resulting cycles per operation. The interfering cores execute only the main loop to create maximum contention against the core under analysis.

Finally, it is important to note that both levels of cache are configured as write-back and write-allocate by default. Therefore, a write to a clean cache line triggers a read, while a write to a dirty cache line results in both a read and a write to the next level of the memory hierarchy, further influencing measured performance.

Algorithm 1 IsolBench

```
1: Set  $\text{jump\_size} \leftarrow \text{cache\_line\_size}$ 
2: Cache Priming:
3: Set  $\text{current\_address} \leftarrow \text{base\_address\_of\_array}$ 
4: while  $\text{current\_address} < \text{end\_address\_of\_array}$  do
5:   Load/Store data from/to  $\text{current\_address}$ 
6:   Update  $\text{current\_address} \leftarrow \text{current\_address} + \text{jump\_size}$ 
7: end while
8: Start Timer
9: Main Loop:
10: for  $i \leftarrow 1$  to  $\text{repeat\_count}$  do
11:   Set  $\text{current\_address} \leftarrow \text{base\_address\_of\_array}$ 
12:   while  $\text{current\_address} < \text{end\_address\_of\_array}$  do
13:     Load/Store data from/to  $\text{current\_address}$ 
14:     Update  $\text{current\_address} \leftarrow \text{current\_address} + \text{jump\_size}$ 
15:   end while
16: end for
17: End Timer
18: Calculate  $\text{total\_cycles} \leftarrow (\text{End Timer} - \text{Start Timer})$ 
19: Calculate  $\text{cycles\_per\_load\_store} \leftarrow \text{total\_cycles} / (\text{array\_size} / \text{jump\_size} * \text{repeat\_count})$ 
```

3.2.2 Experimentation

In our evaluation setup, since we are using 4 VMs with 4 cores and there is a one-to-one mapping between VMs and cores, we will use the terms VM and Core interchangeably. We conducted a series of experiments by varying both the type of operations performed by the Core Under Analysis (CUA)—either read or write—and the operations performed by the Interfering Cores (Intf Cores). We also adjusted the number of interfering cores involved to assess their cumulative impact on bandwidth contention. Importantly, each core was allocated an equal share of the cache colors, resulting in each core having access to 256 KB of L2 cache.

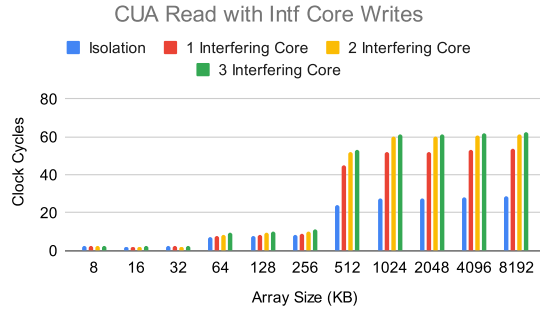
We repeated the experiments for different array sizes on the CUA. Each measurement comprised 100 repetitions of the Main Loop and to ensure stable and representative timing data, the average number of clock cycles per operation were calculated. For the interfering cores, we fixed the array size at 4 MB. This large array size guarantees that interfering cores must access main memory for most operations, causing them to continually miss in their own 256 KB L2 cache partitions and thus exert maximum interference. For each case, we measure the average number of clock cycles per request since we do not have the infrastructure to measure for individual request.

The observed trends, illustrated in Figure 3.2, highlight how different access patterns by interfering cores can affect performance, even when spatial isolation is enforced at the L2 cache and DRAM Bank level.

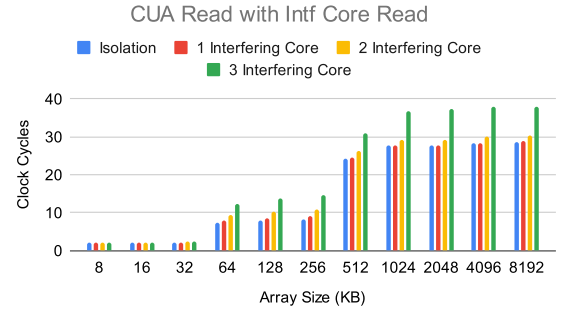
The results from Figure 3.2 clearly show that the clock cycles per request increase once the CUA’s array size grows beyond 32 KB and again beyond 256 KB. These thresholds correspond to the CUA’s L1 cache size (32 KB) and its allocated L2 partition (256 KB). Surpassing these limits forces the CUA to access the next level in the memory hierarchy, introducing additional clock cycles and producing the observed increase in clock cycles.

When the CUA operates entirely within its L1 cache, read and write requests average about 2.16 clock cycles per request. This reflects the fact that the L1 cache is private to the core under analysis, allowing it to avoid bandwidth contention at this level. However, as soon as the CUA begins accessing data in the L2 shared cache, the clock cycles per request increase due to interference from other cores. Reads and writes now compete for shared bandwidth, driving up the measured clock cycles per request.

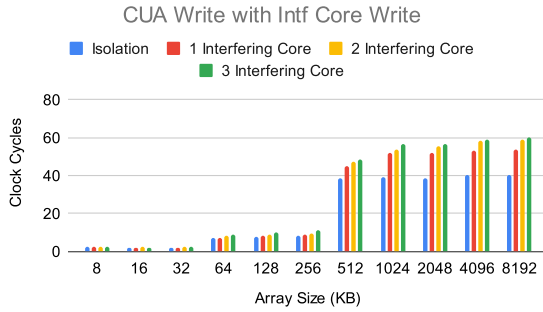
When the CUA experiences L2 cache hits, write interference from other cores increases the average clock cycles. For example, in the case where the CUA is doing reads while the interfering cores are doing writes the average clock cycles per request increase from 8.16 to 11.02. Whereas in the case where the interfering cores do reads with CUA doing reads,



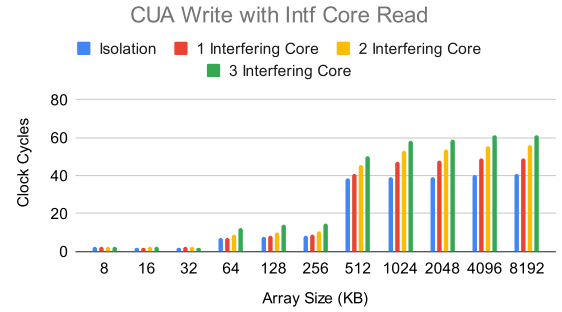
(a) Core under analysis is doing reads while interfering cores are doing writes.



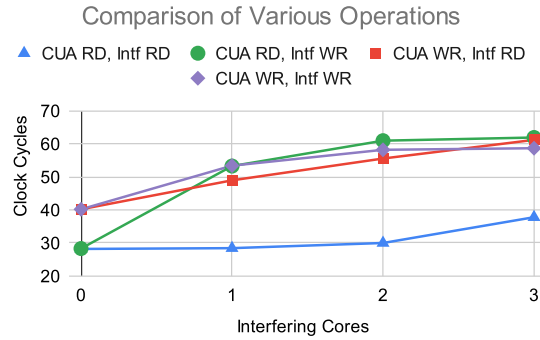
(b) Core under analysis is doing reads while interfering cores are doing reads.



(c) Core under analysis is doing writes while interfering cores are doing writes.



(d) Core under analysis is doing writes while interfering cores are doing reads.



(e) Summary of different experiments with array size of 4MB and varying number of interfering cores

Figure 3.2: Contention results for interference tests.

the average clock cycles increase from 8.16 to 14.14. This occurs because write-allocate misses from interfering cores generate both read and write traffic from the LLC to the main memory. This in turn causes them to use less bandwidth from the Core to LLC, and thus less interference is caused when the CUA is hitting in the cache. In contrast, read interference under L2 hits grows more rapidly, as interfering cores issuing only read requests and thus a write is not caused after the L2 cache, allowing the interfering cores to consume higher share of bandwidth from the core to L2 cache level.

Figure 3.2e focuses on L2 cache misses, where the CUA must access main memory. Here, interfering writes have a particularly strong impact. If the CUA is performing reads and the interfering cores issue writes, just one interfering core causes a substantial increase in clock cycles, and this effect saturates with more interfering cores. This scenario produces the most severe slowdown—up to 2.2x compared to isolated execution. When the CUA itself is issuing writes, the interference pattern (whether from reads or writes) increases clock cycles more linearly as the number of interfering cores grows. Meanwhile, when both the CUA and the interfering cores are reading, significant increases in clock cycles occur only once the third core is introduced. Regardless of the specific pattern, once the memory hierarchy beyond the L2 cache is engaged, bandwidth contention becomes a critical factor.

In summary, these results demonstrate that even with static partitioning and spatial isolation at the L2 cache level, interfering cores can still increase clock cycles per request and degrade performance. This underscores the need for robust bandwidth regulation techniques to achieve stricter isolation guarantees in virtualized systems.

3.3 Memguard Implementation on Bao Hypervisor

As discussed in Section 2.4.3, Memguard is a bandwidth partitioning technique that relies on two key interrupt sources: performance counter interrupts and periodic timer interrupts. These interrupts work together to enforce memory bandwidth constraints among multiple cores. Since we aim to regulate virtual machines, it is essential to implement these interrupts at the hypervisor level. This design ensures that the regulation mechanism is managed by a trusted, minimally complex layer. Delegating such responsibilities to larger operating systems (e.g., Linux or FreeRTOS) could introduce unnecessary complexity and potential security risks. At hypervisor level, Memguard was implemented for Jailhouse hypervisor [34], however, the methodology was not published. The supported Memguard Period was 1ms and periodic timer and performance counter overflow interrupts were used to replenish the budget and halt the core respectively. Here, we implement Memguard on Bao, by carefully analyzing each parameter of the implementation, and using a much faster

regulation period. Our contribution also determines the effect of different sizes of cache partition along with the varying budget allocated to the benchmarks.

To accomplish this, we must:

1. Configure the system timer and Performance Monitoring Unit (PMU) to enable the appropriate interrupts.
2. Integrate interrupt handling within Bao, ensuring that the hypervisor can respond to these events.
3. Implement the corresponding Interrupt Service Routines (ISRs) in Bao.

To meet these goals, we introduce a dedicated memguard library inside Bao. This library provides the necessary functionality to configure and manage Memguard’s interrupts, while additional configuration options enable users to set timer interrupt periods and define bandwidth budgets. The OS can request to reconfigure the budgets at runtime through the hypervisor. As the OS may be compromised, the hypervisor can additionally validate the parameters of this request.

3.3.1 Performance Monitoring Unit Hardware

A high-level view of the PMU in Cortex A53 is shown in Figure 3.3. The performance counters can be configured to count variety of events through event selection registers and to generate an interrupt on overflow through interrupt/overflow registers. These registers are memory mapped and can be configured from the cores. For Memguard, we focus on the L2D_CACHE_REFILL event, which indicates that a cache line was refilled into the L2 cache. Such refills occur either because cache miss caused a dirty cache line to be evicted to main memory and a read being performed, or simply because a cache miss led to a direct read from memory. We also enable the corresponding counter overflow interrupts for each core which is routed through Generic Interrupt Controller (GIC) to corresponding cores. According to [53], these interrupts are assigned IDs 175 - 178 for the four cores respectively.

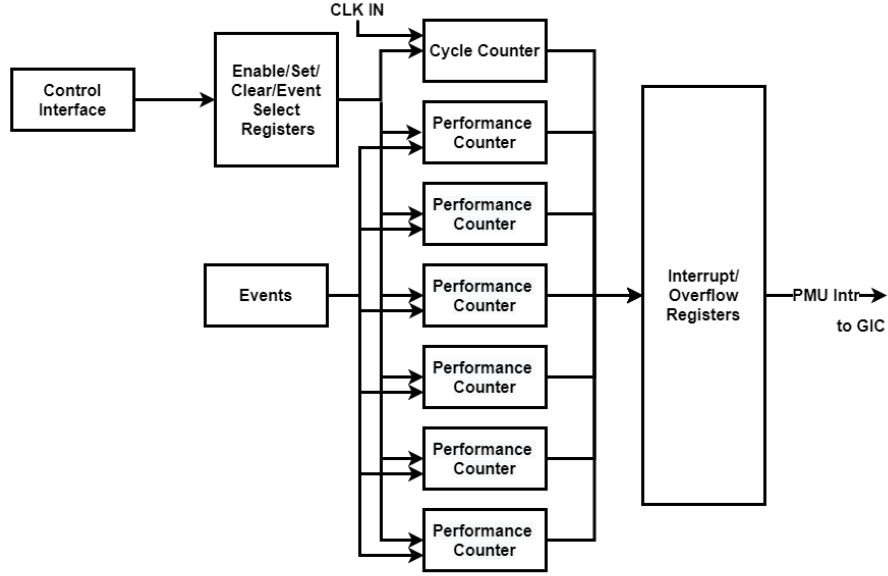


Figure 3.3: PMU Architecture in Cortex A53 with 6 performance counters and one cycle counter. Figure shows PMU Intr interrupt going from the PMU IP. This is received by GIC. The figure is taken from ARM Cortex A53 Reference Manual [2]

3.3.2 Hardware Configuration for PMU and Timer

In this section, we detail how the PMU counters are configured to track events and generate overflow interrupts for Memguard, as well as how the timer is set up to enable Memguard's periodic timer interrupt.

To start, we initialize the PMU by enabling its hardware within the core. This is done by setting the enable bit (bit 0) of the `PMCR_EL0` register. To reserve a performance counter for the hypervisor level, the `MDCR_EL2` register is used to specify the counter through a bitfield. Next, to monitor L2 cache refills, we configure Performance Counter 0 by writing the appropriate event code into the `PMXEVTYPER<0>_EL0` register. This monitors the events across all the exception levels. The counter's initial value is then set via the `PMEVCNTR<0>_EL0` register. Since the overflow occurs when the 32-bit counter wraps around, we preset it by subtracting the desired threshold from the maximum. Overflow interrupts are enabled by setting bit 0 in `PMINTENSET_EL1`, which corresponds to our chosen counter. However, the counter only begins counting after we set its corresponding bit (bit

0) in `PMCNTENSET_ELO`. The `memguard_init` function in the `memguard` library performs this configuration, preparing the PMU counters for Memguard’s operation.

For the timer interrupts, we use the ARM Cortex-A53 Generic Timer as described in the ARM A-Profile Architecture Reference Manual (Chapter D12) [32]. Although the Generic Timer provides one-shot interrupts by default, the interrupt handler can be programmed to re-enable the timer, thus achieving periodic interrupts needed by Memguard. The timer interrupt has an ID of 30 and, like the PMU interrupts, is routed through the Generic Interrupt Controller (GIC). The final steps—enabling the timer interrupt and starting the PMU counter—are performed within the `memguard_run()` function.

The Memguard period and the budget for each core can be adjusted through the `memguard` header file, however, a hypercall interface has been provided for the OS to adjust these parameters on the runtime. In case where the OS is not fully trusted to change these parameters, the hypercall interface can be disabled or further checks can be implemented within the hypercall to prevent the OS from causing unexpected changes to the regulation. The hypercall reserved for this purpose is numbered 74, and the arguments to this hypercall is the `memguard` period, followed by the budget for each core.

3.3.3 Software Configuration for Interrupts

In Bao, interrupts must be explicitly designated for handling by either the hypervisor or a VM. To support Memguard, we reserve the PMU interrupts (IDs 175–179) and the timer interrupt (ID 30) for the hypervisor. Bao’s `interrupts_reserve` function is used to reserve these interrupts for the hypervisor and register the corresponding interrupt handler. Once reserved, the `interrupts_cpu_enable` function is used to enable the interrupts.

GIC-based interrupt routing allows configuring interrupt priorities, but Bao does not support nested interrupts. For Memguard, we assign a lower priority to the timer interrupt than to the counter overflow interrupt, using Bao’s `gic_set_prio` function. This ensures the counter overflow interrupt—responsible for running until the end of the Memguard period—always takes precedence. Since the counter overflow interrupt sets the timer for the next Memguard period, the timer interrupt does not need to run once the counter overflow ISR has completed. Within the counter overflow ISR, a tight polling loop checks if the Memguard period has ended, incurring minimal overhead before configuring timer registers for the new period.

3.3.4 Interrupt Service Routines

Algorithm 2 outlines the implementation of the Memguard periodic timer Interrupt Service Routine (ISR). The routine begins by disabling the timer using the timer control register, `cntp_ctl_el0`. The compare value register, `cntp_cval_el0` is incremented with the memguard period, and re-enabled. This enables the timer to generate another interrupt after the absolute time value is equal to `cntp_cval_el0` register. Next, the budgets for the PMU counters are reset by updating the corresponding counter values with their allocated budgets. In our experimental setup, Core with ID of 0 is designated as the Core Under Analysis (CUA) and rest of the cores are designated interfering cores.

Algorithm 2 Periodic Timer Interrupt Service Routine

```
1: set_reg cntp_ctl_el0, 0x0
2: set_reg cntp_cval_el0, (cntp_cval_el0 + period) //Interrupt for next period
3: set_reg cntp_ctl_el0, 0x1
4: set_reg pmevctr0_el0, budget[cpu_id] // Budget according to the Core ID
```

Algorithm 3 depicts the implementation of the counter overflow ISR. A counter overflow interrupt occurs when a 32-bit PMU counter, tracking `L2_DCACHE_REFILL` events, exceeds the 32-bit unsigned maximum value. Consequently, the value for a counter is specified using the formula:

$$counter_value = 0 - budget$$

In the overflow ISR, the first step is to clear the PMU overflow interrupt flag. The corresponding counter is then reloaded with the budget set for the specific core. Lines 7–11 implement a busy-wait loop that ensures the routine does not proceed until the Memguard timer for the current period has expired. This is checked by reading the timer physical count register `cntpct_el0`, which holds the absolute timer value, and the `cntp_cval_el0` register which holds the timer compare value which represents the end of memguard period. Once the loop exits, and the pending timer interrupt is cleared, as shown in Lines 13–15.

It is important to note that once the overflow interrupt is triggered, entering the hypervisor into the overflow ISR may cause additional cache misses and may cause an overshoot of the budget. It can be avoided by subtracting the budget from the counter value to compensate the overshoot in the next period, however, reading a performance counter also incurs additional overhead and hence, it is more efficient to replenish the counter with the exact value of the budget.

Algorithm 3 Counter Overflow Interrupt Service Routine

```
1: set_reg pmovsclr_el0, 1 //Clear the PMU overflow interrupt flag
2: set_reg pmevcntr0_el0, budget[cpu_id]
3: get_reg cntpct_el0, init_val
4: get_reg cntp_cval_el0, final_val
5: while final_val > init_val do
6:   get_reg cntpct_el0, init_val
7: end while
8: clear_reg gic_icpendr, timer_interrupt //Clear the timer interrupt pending
9: clear_reg gic_icactiver, timer_interrupt //Clear the timer interrupt active
10: clear_reg gic_dir, timer_interrupt //Deactivate the interrupt explicitly
```

3.3.5 Timer Interrupt Overhead

The Memguard timer interrupt is a periodic event managed by Bao, and its handling introduces overhead due to the context switch it requires. During the context switch, the state of general-purpose and system registers must be saved, followed by the retrieval of the interrupt ID and invocation of the corresponding handler. This overhead occurs even when cores remain within their allocated budgets for a given period, adding to the execution time of programs running on the VMs. While reducing the timer period allows for finer regulation of memory bandwidth, it also increases the overhead from frequent context switching. Thus, determining an optimal timer period that balances regulation precision with minimal performance impact is essential.

To evaluate this trade-off, experiments were conducted using a single VM under Bao, running a synthetic read benchmark while varying the timer interrupt period (or Memguard period). For the benchmark, an array of 4 MB was sequentially traversed performing reads at cache-line granularity. The core operated in isolation, with one-fourth of the L2 cache partitioned for its use. The budget was set high enough to ensure that overflow interrupts would not be triggered.

As shown in Figure 3.4, the percentage slowdown for the synthetic read benchmark was observed to be inversely proportional to the timer interrupt period. Shorter periods caused more frequent context switches, leading to greater overhead. Based on these findings, a timer period of 20 μ s was chosen to ensure that the overhead remains below 3.9%, creating a balance between regulation precision and performance impact.

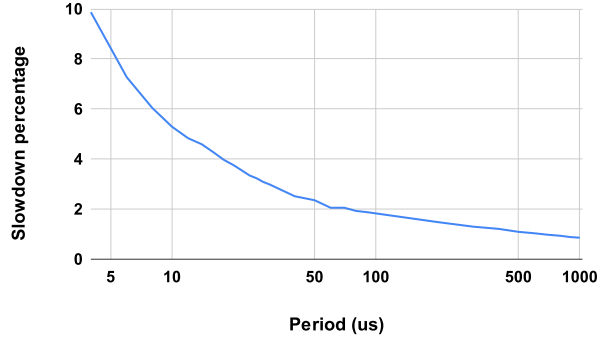


Figure 3.4: Timer interrupt overhead in terms of slowdown of synthetic read benchmark

3.3.6 Minimum Sustainable Bandwidth

The next step in the implementation involves determining the maximum worst-case sustainable memory bandwidth. This was evaluated using a series of benchmarks designed to stress the DRAM controller under adverse access patterns. DRAM controllers experience their worst-case performance when memory accesses target the same bank with closed rows. However, as long as VMs are assigned different cache colors (see Section 3.1), they will not share the same DRAM bank. Therefore, instead of focusing on access patterns within the same bank, our tests were designed to evaluate bandwidth across different banks to determine the maximum sustainable memory bandwidth under worst-case conditions.

The benchmarks were conducted in a bare-metal environment to allow full control over the physical address space. The tests involved performing memory jumps in 256 KB increments across a contiguous array. This ensured that memory accesses mapped to different rows within the same bank, creating the desired stress on the DRAM. To ensure cache misses at both levels of the memory hierarchy, the test iterated the 256 KB jumps more times than the number of ways in the cache. This approach ensured that only a single cache set was used, continuously evicting previously cached data.

To further evaluate the impact of the memory prefetcher, additional tests were conducted using a random pointer-chasing pattern instead of static 256 KB jumps. However, no significant difference was observed between the results of the static and random patterns, confirming that the prefetcher had minimal impact on the outcomes.

The results, presented in Figure 3.5, show a linear increase in bandwidth as the number of cores increases. The maximum observed bandwidth for writes with four interfering cores

was 2025 MB/s, which is lower than that for reads. This value represents the maximum sustainable memory bandwidth for our use case. When converted to a memguard budget with a period of 20 μ s, it gives the total number of cacheline refills that we can allocate among VMs for one Memguard period:

$$\begin{aligned} \text{Maximum Budget} &= 2025 \frac{\text{MB}}{\text{s}} = 33.1776 \times 10^6 \frac{\text{cache lines}}{\text{s}} \\ \text{Maximum Budget} &= 33.1776 \times 10^6 \times 20 \times 10^{-6} \approx 663.6 \frac{\text{cache lines}}{20\mu\text{s}} \end{aligned}$$

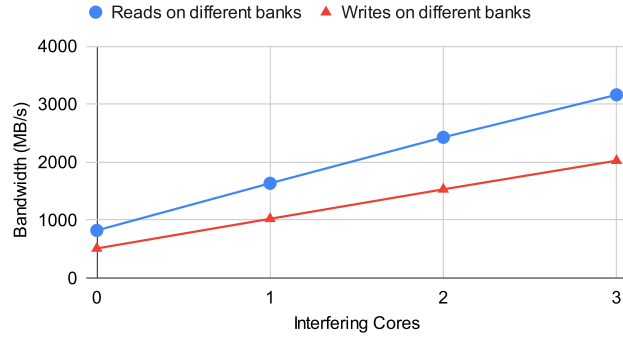


Figure 3.5: Observed bandwidth for different interference tests

3.4 Evaluation of Bandwidth Regulation

In this section, we evaluate the effect of bandwidth regulation on San Diego Vision Benchmark Suite along with the effect of variation of parameters such as budget and number of cache colors assigned. Since the benchmarks are supported on Linux, we use Linux with the Real-Time patch (PREEMPT_RT) [14] to run these benchmarks. The other three VMs, referred to as "interfering VMs," ran synthetic write benchmarks, which involved sequentially iterating over a 4 MB array. Synthetic write benchmarks were chosen because they generate the highest level of contention, as discussed in Section 3.2.2. All VMs were configured to be regulated by Memguard.

The experiments were conducted with the Linux VM assigned between 1 and 5 cache colors. Limiting the Linux VM to a maximum of 5 cache colors ensured that the 8 total

cache colors available were sufficient for each interfering VM to be assigned a unique cache color, avoiding overlap. The remaining cache colors were allocated to the interfering VMs. While the allocation of cache colors among the interfering VMs was not always equal, the variation did not impact their behavior because these benchmarks are iterating over an array that is number of times larger than the total cache size. Thus, the interfering VMs consistently produce the same number of cache misses regardless of the amount of cache allocated.

We use the San Diego Vision Benchmark (SDVB) Suite [50] for our evaluation. These benchmarks perform compute and memory intensive tasks and are representative of computer vision workloads. For the Linux VM, we tested a range of budget values from 30 to 120. Budgets above 120 were not evaluated because they did not result in significant performance improvements for the SDVB benchmarks. This behavior is attributed to the memory bandwidth utilization characteristics of SDVB; benchmarks with higher memory bandwidth demands would exhibit more pronounced performance gains at higher budget values. The remaining budget was evenly distributed among the interfering VMs. It is important to note that the value of budget for one core means that core is only allowed to have that number of cacheline refills in L2 cache for one memguard period (20us). If it goes beyond that number, Memguard counter overflow interrupt will trigger and halt the execution of VM till the next Memguard period.

3.4.1 Evaluation of SDVB Benchmarks

Figure 3.6 presents the slowdown ratios for the San Diego Vision Benchmark Suite using a heatmap representation under no interference conditions. The horizontal axis shows the number of cache colors assigned to the benchmark, while the vertical axis indicates the allocated budget. Execution times were measured over 30 repetitions under the same configurations and the values above the 85th percentile and below 15th percentile were removed and an average of the remaining readings was taken. To normalize the results, the average execution time of the benchmark configuration with the highest budget and cache colors was used.

Overall, the results demonstrate that benchmarks exhibit greater slowdown with fewer cache colors and smaller budgets. As the number of cache colors and the allocated budget increase, the slowdown decreases. For benchmarks such as `mser`, there is a strong dependence on both the number of cache colors and the budget allocated. The normalized execution time for the case with highest budget and cache colors has an average 1.0. The 15th and 85th percentile are 0.9969 and 1.0046 respectively. With the lowest budget

and cache color, the normalized execution time is 2.9963 with 15th and 85th percentiles at 2.9295 and 3.0198 respectively. This gives the maximum slowdown of approximately 3 times with respect to averages. This can be attributed to **mser**'s high memory bandwidth requirements. Increasing the budget allows the benchmark to issue more memory requests within a given period, resulting in faster execution time. This behavior highlights the importance of memory access patterns as well. Benchmarks with bursty memory access patterns tend to experience greater slowdowns compared to those with similar total memory accesses but smoother access patterns.

In contrast, benchmarks such as **svm** exhibit less significant slowdowns, even with minimal cache partitions and budgets. The **svm** benchmark, for example, shows a slowdown ratio of 1.12 with respect to the average execution times between the case with minimum budget and colors assigned and the case with maximum budget and color assigned. This suggests that the execution time of such benchmarks may be more dependent on CPU-intensive workloads or that their memory access patterns are less bursty in nature, and hence do not cause memguard to halt the core much.

The **multi_ncut** benchmark did not show much sensitivity to the budget or the number of colors assigned as the average execution time goes from 1.0 with the percentiles at 0.9996 and 1.0, to 1.0116 with percentiles at 1.011 and 1.015. This indicates that a higher portion of the benchmark may be consumed by CPU intensive tasks or the benchmark may be operating on a small set of data, thus causing less number of misses in shared cache. Benchmarks such as **localization** and **stitch** do not follow completely monotonic trend. This variability can be attributed to the presence of outliers in data and non-real-time nature of Linux, where background processes, services, and the scheduler's behavior can impact execution times.

Figure 3.7 shows the results when benchmarks were executed with write contention by interfering cores. To normalize the execution time in this case, same value is chosen to normalize with as done for the case with no interference. This helps to compare the execution times across the two experiments. As a general trend, it is observed that, the benchmarks exhibit similar execution times with or without interference. Although comparable, the benchmarks ran in contention show slightly higher execution times in some cases. For example, the disparity benchmark, when ran with lowest budget and colors, has a normalized execution time of 1.7030, with percentiles at 1.6892 and 1.7074 when it is running without interference, and it goes to 1.7382, with percentiles at 1.7314 and 1.7443 when it runs with interference. When comparing the case with highest colors and budget for disparity benchmark, the execution time goes from 1.0, with percentiles at 0.998 to 1.0011, to 1.001, with percentiles at 1.0 and 1.0014 when it runs in contention. The similarity in execution times between isolated and contention scenarios for the most cases demonstrate MemGuard's

effectiveness in mitigating interference from other cores competing for memory bandwidth.

3.5 Discussion

We see how Memguard implementation regulates bandwidth on the basis of interrupts from performance counter and timer. The evaluation shows that Memguard is able to regulate the interference from other cores such that the execution times with or without interference remains similar. However, the implementation introduces a constant overhead due to the periodic timer interrupts. Additionally, the overflow interrupts contribute further overhead. These overheads arise not only from the execution time of the interrupt service routine but also from potential cache line evictions during context switches from the VM to the hypervisor, which involve saving and restoring state registers. This overhead could be minimized if hardware support existed for periodic budget refreshes and mechanisms to prevent cores from accessing memory until the next regulation period without incurring additional overhead. Unfortunately, such hardware support is absent on our platform and many other commercial off-the-shelf systems, limiting the efficiency of this approach.

We also saw that Memguard regulates on the basis of minimum sustainable bandwidth, which is determined through the worst case access pattern to DRAM. The cumulative bandwidth of the system is enforced to stay below the minimum sustainable bandwidth which can worsen the performance of applications as the DRAM can provide bandwidth much higher than the worst case. This limitation can be improved upon by utilizing performance counters to understand how latency of requests may be affected within DRAM.

Our implementation is based on the L2 Refill events. These events may be caused by an eviction of clean cache line which will cause a read at the next level of memory, or a dirty cache line which will cause a write-back operation, flushing the data to main memory and then causing a read. However, since the tracking of events is based on overflow interrupt, more than one event cannot be tracked without specialized hardware support. Moreover, it is seen in Section 3.2.2 that there is interference observed in L2 cache as well, as the number of cores are increased. However, Memguard does not take into account the interference in L2 cache either.

Our platform does not allow the measurement of memory activity of other entities such as I/O or DMA and thus, we do not incorporate them in the regulation scheme. Section 2.4.3, also discusses a prediction based reclamation mechanism which is able to redistribute budget, however, we do not implement it as it is not recommended to be used for real-time systems.

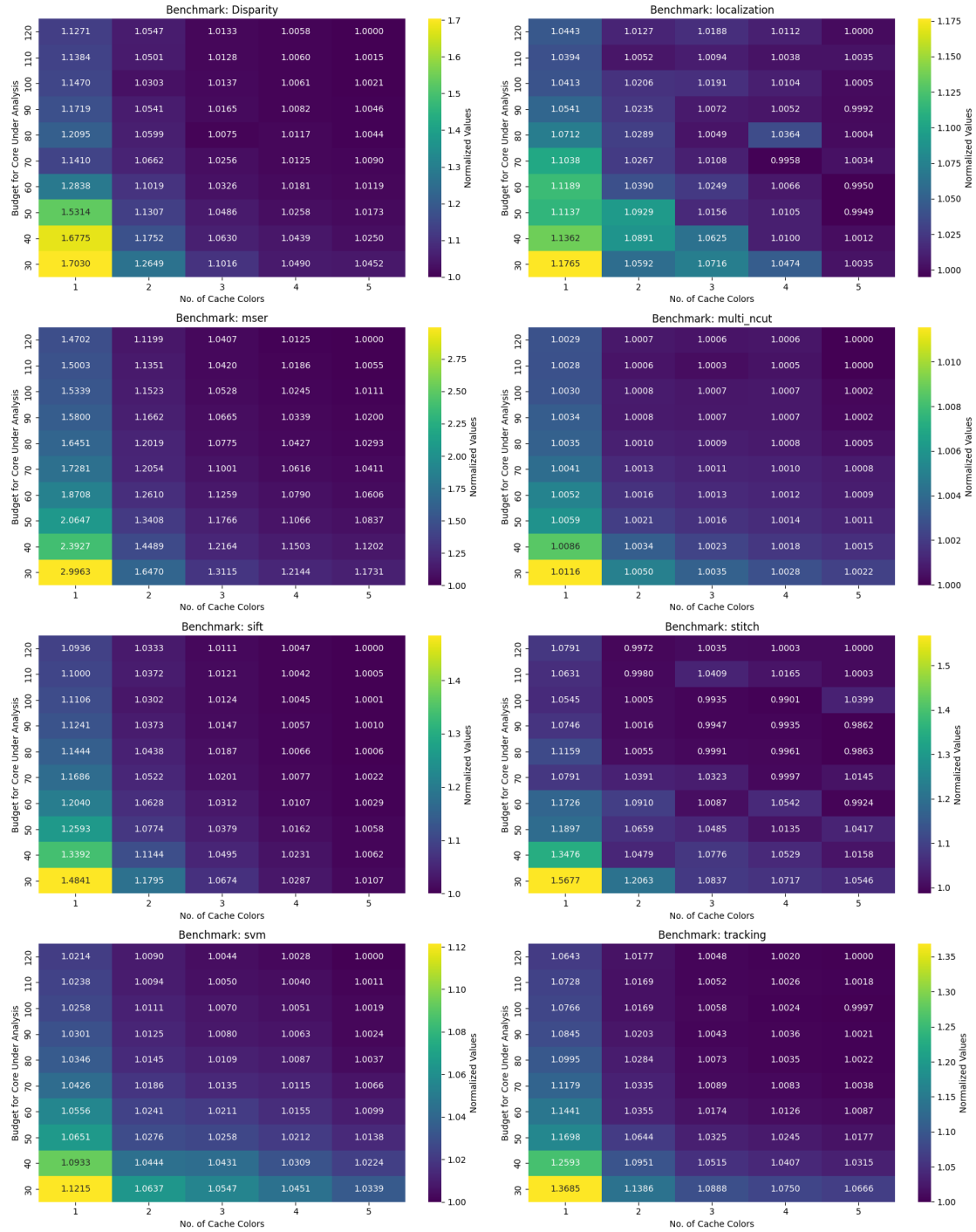


Figure 3.6: Results for San Diego Vision Benchmark without interference

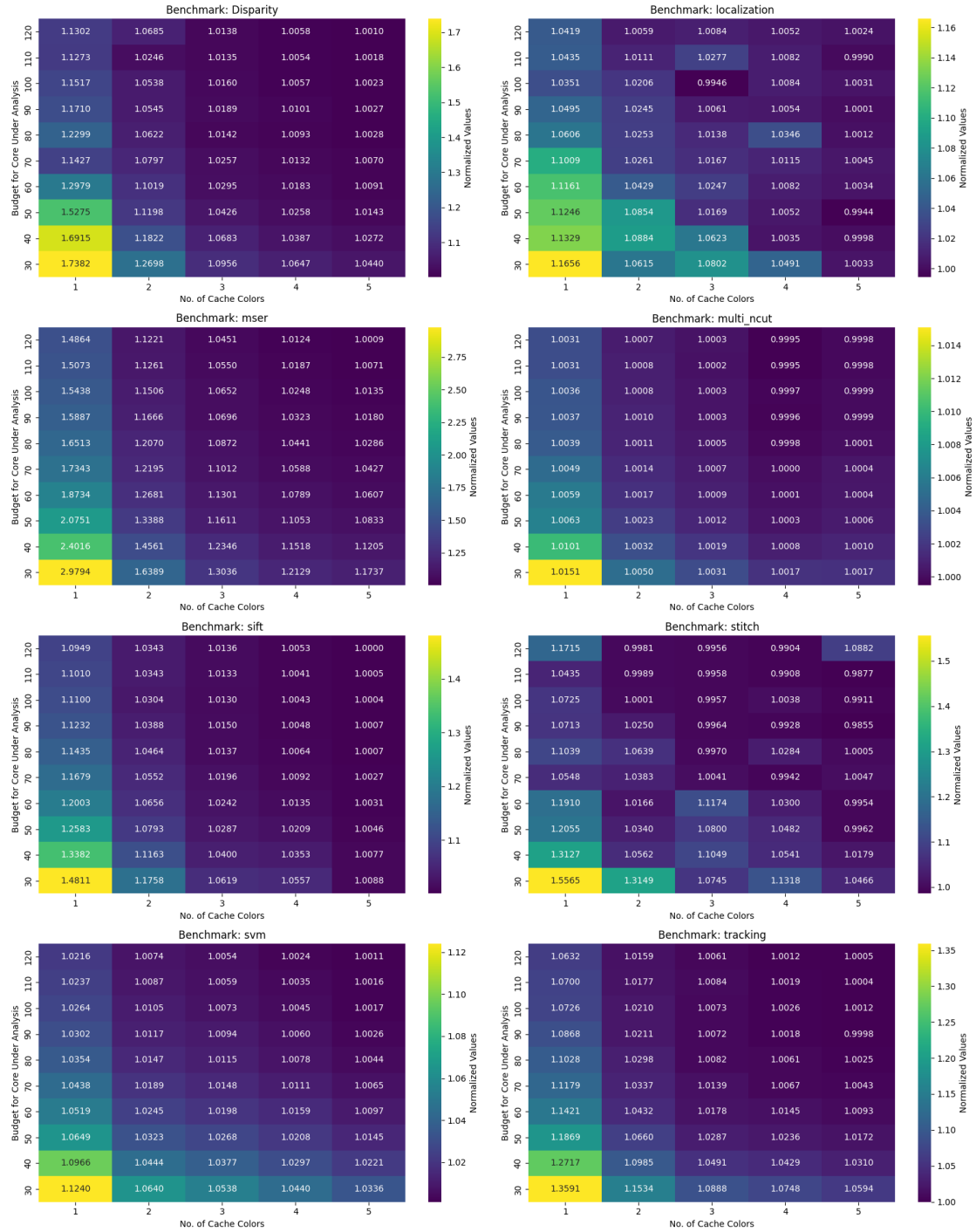


Figure 3.7: Results for San Diego Vision Benchmark with interference

The results also shows that the memguard regulation depends on the nature of workload. In case where the workload is cpu intensive and does not make much memory accesses, the addition of regulation may not result in significant differences. However, in cases where the benchmarks make a significant number of memory requests, it is important to perform regulation as this would cause significant degradation of performance.

3.6 Conclusion

Hence we implement Memguard for Bao Hypervisor and evaluate the results for regulation of VMs at hypervisor level, while also varying the cache partitions used by the VMs. Minimal difference in execution time with or without interference shows that we were able to address our design goal. Comparing this implementation with respect to [34], we support a much smaller regulation period of 20us as compared to 1ms in Jailhouse memguard implementation. Based on the results we conclude that Memguard is an effective technique in regulation bandwidth contention such that the overall execution times of tasks remains similar when executed with or without contention.

Chapter 4

APMU Software Support

In Chapter 2, we introduced the Advanced Performance Monitoring Unit (APMU) and its potential for enabling system-wide performance monitoring, with applications across various domains. Building on this foundation, this chapter focuses on the design and implementation of the software stack for the APMU which can provide a rich development environment.

The primary objective is to develop a lightweight software system, optimized for minimal code size and execution time, as the APMU incorporates a resource-constrained processor. This includes designing a robust toolchain capable of recognizing and utilizing custom instructions implemented within the APMU. The software system is further designed as an event-driven framework and efficient communication between the APMU and other components of the system is also desired. Overall, we aim our software to meet the following goals:

1. The software should be reasonably efficient with respect to execution time and the code size to minimize the latency of operations and to have a small memory footprint.
2. Any communication sent to the APMU should be mediated by the hypervisor to prevent malicious attacks directly through the OS.
3. It should be dynamically reconfigurable at runtime, allowing functionalities to be loaded or unloaded as needed, while maintaining operational efficiency.

In this chapter, we focus on discussing the design and implementation of the software stack. An evaluation of its performance is also presented in the subsequent sections.

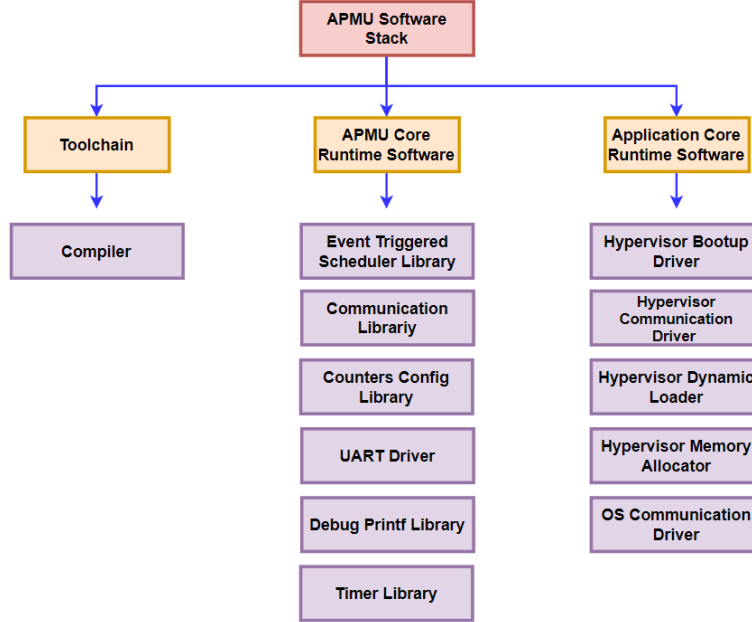


Figure 4.1: Software Infrastructure for APMU

4.1 APMU Software Framework Overview

This section provides an overview of the APMU’s supporting software infrastructure. Figure 4.1 illustrates the core contributions. Since the APMU Core relies on custom instructions, a specialized toolchain capable of compiling as well as disassembling for debugging purposes, is required. We describe modifications to LLVM that enable a comprehensive toolchain solution for the APMU in Section 4.2.

Then we start with the development of APMU Core runtime software stack in Section 4.3. Bidirectional communication between the APMU core and the application cores is also essential, prompting the introduction of low-footprint communication queues in Section 4.3.1. These queues are developed keeping the second design goal in mind. The APMU is a low-power monitoring core designed to detect events across the system and take actions based on those event. We develop an event-triggered scheduler, introduced in Section 4.3.3, that leverages the custom instructions to deliver functionality in response to detected events.

Next, we introduce the concept of components in Section 4.3.2. There can be a number of components present at bootup, each with configurable event handlers to be invoked by

the event-triggered scheduler. These are called static components which are a part of code generated for APMU and are installed at bootup through the bootup routine. To cater to the third design goal, we introduce features to allow components to be installed at runtime through the application cores. These are referred to as dynamic components. Components also support sending and receiving data through communication queues, and they include communication callbacks to handle data received from application cores to the APMU, that targets a specific component.

In addition, we present a Base Component in Section 4.3.4 that streamlines common operations, such as invoking the appropriate request handlers of individual components when a communication object arrives via the communication queues. The base component is a pre-installed component which provides the install and uninstall functionalities for both the static and dynamic components. For static components, the install function can be invoked in the bootup routine to install all of them sequentially. For dynamic components, these functions are invoked when the base component receives the appropriate request to install or uninstall the component through the communication queues.

To further facilitate APMU’s runtime software development, in Section 4.3.5, we provide additional drivers and libraries for tasks such as timer operations, debug printing, and managing the APMU counters and their associated registers.

Facilitating the interaction of APMU core and the application cores, Section 4.4 discusses the bootup and communication drivers provided for the hypervisor and the OS. Section 4.5 discusses the software support provided on the application cores to be able to install dynamic components. Evaluation of the implemented system is performed in Section 4.6 followed by the conclusion in Section 4.7.

The APMU is intended to work in a hypervisor-enabled system where hypervisor executes at a higher privilege than the Virtual Machines which run the OS or baremetal applications. Libraries to support APMU bootup, communication and dynamic installation of components onto APMU are provided for Bao Hypervisor. Furthermore, communication libraries are provided for the operating system or baremetal applications.

4.2 Toolchain

As discussed in Section 2.5.4, APMU supports four custom instructions: counter read, counter write, wait for pending, and wait for overflow. Existing RISC-V toolchains, such as GNU GCC [12] or LLVM [39], do not natively support these instructions, making a tailored toolchain solution essential for practical software development.

As noted in Section 2.6, LLVM’s modular structure and extensive documentation render it more adaptable for customization compared to GNU GCC’s monolithic design. This adaptability is crucial for incorporating the APMU’s custom instructions into a streamlined software development flow. The following sections describe how we leverage LLVM’s flexibility to integrate these instructions.

4.2.1 Inline Assembly for Custom Instructions

Before detailing the LLVM modifications, we first clarify the intended use of the APMU custom instructions. They are designed to be employed as inline assembly, since their specialized functionality cannot be represented as a subset of standard instructions translated directly from C code.

Listing 1 presents inline functions built around inline assembly, making it convenient to invoke these custom instructions. The inline assembly specifies input and output operands, enabling the compiler to optimize these functions and avoid function call overhead.

Listing 1 Macros for Custom Instructions

```
1  static inline uint32_t pmu_counter_read(uint32_t counter_num) {
2      uint32_t res;
3      asm volatile ("cnt.rd %0, %1" : "=r"(res) : "r"(counter_num));
4      return res;
5  }
6
7  static inline void pmu_counter_write(uint32_t counter_num, uint32_t val) {
8      asm volatile ("cnt.wr %0, %1" : : "r"(counter_num), "r"(val));
9      return;
10 }
11
12 static inline uint32_t pmu_wait_for_pending(uint32_t counter_bitmask) {
13     uint32_t bitmask;
14     asm volatile ("cnt.wfp %0, %1" : "=r"(bitmask) : "r"(counter_bitmask));
15     return bitmask;
16 }
17
18 static inline uint32_t pmu_wait_for_overflow(uint32_t counter_bitmask) {
19     uint32_t bitmask;
20     asm volatile ("cnt.wfo %0, %1" : "=r"(bitmask) : "r"(counter_bitmask));
21     return bitmask;
22 }
```

4.2.2 Inline Assembly in LLVM

In LLVM’s workflow (introduced in Section 2.6), inline assembly strings are carried from the frontend through to the backend with only minimal constraint checking. Since the constraints in Listing 1 are generic rather than target-specific, their handling remains unaltered in both the Clang frontend and the LLVM IR.

On the backend, however, these operands are directly lowered to the target assembly instructions, bypassing the Target Dependent Instruction Selection Phase. Registers are allocated during the Register Allocation Phase, and the instructions then proceed through the Code Emission Phase. Our modifications thus focus solely on the backend’s Code Emission Phase to ensure correct translation of machine code from the custom instructions.

4.2.3 Adding New Instructions in RISC-V Backend

To introduce the APMU custom instructions into LLVM's RISC-V backend, we first build upon the existing infrastructure defined by the `Instruction` class in LLVM. This class includes variables defining the instruction semantics, translation as well as the instruction behavior within backend optimization phases. Within the RISC-V backend, the file `RISCVInstrFormats.td` provides a hierarchy of classes tailored to different instruction types, depicted in Figure 2.4.

The first step in adding APMU instructions is to define a new `OpCode`. Since we are introducing an additional set of instructions that share the same `OpCode`, we add a corresponding entry to `RISCVInstrFormats.td`:

```
1 +def OPC_APMU_CNT : RISCVOpcode<"APMU_CNT", 0b0000111>;
```

With the `OpCode` in place, we then define each custom instruction. All instructions must inherit from the base `Instruction` class. This base class is extended by various class hierarchies in `RISCVInstrFormats.td`, which provide common patterns for different instruction formats (e.g., I-type, S-type). These extended classes, define common variables from `Instruction` base class and allow easier addition of instructions. For the counter read instruction (`cnt.rd`), which follows the I-type format, we leverage the `RVInstI` class:

```
1 class RVInstI<bits<3> funct3, RISCVOpcode opcode, dag outs, dag ins, string  
  ↪  opcodestr, string argstr>
```

We then define the counter read instruction as follows:

```
1 +def CNTR_READ : RVInstI<0b000, OPC_APMU_CNT, (outs GPR:$rd), (ins GPR:$rs1),  
  ↪  "cnt.rd", "$rd, $rs1">, Sched<[]> {  
2 +   let imm12 = 0;  
3 +   let hasSideEffects = 0;  
4 +   let mayLoad = 1;  
5 +}
```

In this definition, `OPC_APMU_CNT` specifies the newly introduced `OpCode`, and `funct3` is set to `0b000` based on the encoding shown in Figure 2.5a. The `(outs GPR:$rd)` and `(ins GPR:$rs1)` parameters define the destination (`rd`) and source (`rs1`) registers. The

mnemonic (`"cnt.rd"`) and operand string (`"$rd, $rs1"`) clarify the assembly syntax, and we set `imm12 = 0`, since no immediate is required for the counter read instruction.

The attributes `mayLoad` and `mayStore` indicate whether the instruction can load or store values in registers. For `cnt.rd`, as well as the wait-for-pending and wait-for-overflow instructions, `mayLoad` is set to 1 because they load values into registers. For the counter write instruction, we set `mayStore = 1` to signify that it writes a value. The `hasSideEffects` variable is set to 1 for all instructions except the read instruction to ensure the compiler does not optimize these operations in unexpected ways, as they have non-trivial architectural side effects.

Finally, we define an instruction alias using the `InstAlias` class to help LLVM's Assembler and Disassembler recognize and generate the assembly mnemonics:

```
1 def : InstAlias<"cnt.rd $rd, $rs1", (CNTR_READ GPR:$rd, GPR:$rs1)>;
```

The same process is then repeated for the other APMU custom instructions, ensuring that each one is properly integrated into the RISC-V backend's instruction hierarchy, `OpCode` definitions, and assembly/disassembly infrastructure.

4.2.4 Linker and LibC

While the LLVM Clang project integrates the frontend, middle end, and backend stages within its build, it does not include a linker. In principle, any linker that supports the target ELF specification can be used. To maintain consistency and simplify the toolchain, we employ LLVM's own linker, `lld`, as our default choice.

At the time of writing, LLVM's `libc` implementation for RISC-V remains underdeveloped. Consequently, it is advisable to rely on an external `libc`. The GNU GCC toolchain provides a collection of `libc` libraries that can be seamlessly integrated with LLVM Clang. For lightweight applications running on the APMU Core, we use `newlib-nano`. This library is carefully optimized to minimize code size and memory usage, making it especially suitable for resource-constrained environments such as APMU. Its lean footprint ensures reduced overhead, faster processing times, and improved efficiency, all of which are critical for the APMU.

4.2.5 Testing

To verify the functionality of the custom instructions, a small assembly snippet was written and compiled into an object file. The resulting object file was then disassembled to examine both the raw hex code and the corresponding disassembled instructions. Table 4.1 presents each instruction alongside its machine code and a detailed breakdown based on the instruction encoding from Figure 2.5. This breakdown confirms that the generated machine code contains the correct OpCode, funct3, funct7 (where applicable), and register encodings. Thus, we conclude that the compiler is successfully recognizing and encoding the APMU custom instructions.

Instruction	Machine Code	Breakdown
cnt.wr x14, x15	0x00e79007	OpCode = 0b0000111, funct3 = 0b001, rs1 = 15 (x15), rs2 = 14 (x14), imm = 0
cnt.rd x16, x14	0x00070807	OpCode = 0b0000111, funct3 = 0b000, rd = 16 (x16), rs1 = 14 (x14)
cnt.wfp x17, x14	0x00072887	OpCode = 0b0000111, funct3 = 0b010, funct7 = 0b0000000, rd = 17 (x17), rs1 = 14 (x14)
cnt.wfo x18, x14	0x02072907	OpCode = 0b0000111, funct3 = 0b010, funct7 = 0b0000001, rd = 18 (x18), rs1 = 14 (x14)

Table 4.1: Matching Compiled Machine Code with Instruction Encodings

4.3 APMU Core Software Runtime

In the previous section, we introduced the APMU toolchain. Here, we focus on the APMU Core runtime, which supports a diverse range of software applications. The APMU is controlled by a privileged hypervisor that manages multiple virtual machines (VMs). This hypervisor serves as a trusted entity, leveraging its minimal compute base and hardware-enforced isolation mechanisms, such as higher privilege levels and two-stage address translation. Given the critical nature of the APMU—which includes capabilities for reading and writing to system memory and monitoring various system-wide events—it is essential that its control remains with a reliable and secure authority. Consequently, the development of the APMU software runtime is guided by a virtualization-centric framework, where the hypervisor actively intervenes in critical interactions of VMs with APMU to mitigate potential security risks to the overall system. As illustrated in Figure 4.2, the system comprises three principal elements: (1) Communication Queues (2) Components and (3) Event-Triggered Scheduler

At the heart of the runtime are the communication queues, which form the backbone for data exchange. These queues follow a FIFO, single-producer, single-consumer (SPSC) model, ensuring memory efficiency and predictable behavior. Bidirectional communication requires at least two queues — one for requests and one for responses — and if multiple VMs interface with the APMU, there must be at least one dedicated response queue per VM to route the data back from the APMU to the VM. The request queue is unique and is managed by the hypervisor, which employs proper lock-based synchronization to maintain mutual exclusion, ensuring consistent and orderly data flow of data from multiple VMs. Keeping single request queue for multiple VMs ensures minimal usage of resources as we will discuss in Section 4.3.4. This design facilitates minimal overhead in inter-partition communication and preserves deterministic behavior under high concurrency levels.

The second key concept is the notion of a *component*, a code module associated with specific event handlers and request handlers. Components can be installed or uninstalled into the system through dedicated functions and *init* and *exit* hooks are invoked in respective operations. These hooks handle resource registration and deallocation, enabling a clean integration and removal process. The event-triggered scheduler utilizes custom instructions to detect hardware events. When a relevant event is detected, the scheduler invokes the event handler of the appropriate component. Each component has a unique Component ID which is used to identify the component. This modular and extensible structure simplifies maintenance and allows new components to be incorporated without significantly altering the core runtime framework.

Among the components, an internal *Base Component* is responsible for receiving and routing incoming requests from the request queue. This routing is done on the basis of Request ID which has one-to-one correspondence with the Component ID. Some requests may be forwarded to other components based on their Request ID, while others with Request ID matching that of the *Base Component*, may trigger dynamic component installation or removal operations within the Base Component. The Base Component can also write to all the response queues. This centralized approach not only streamlines request management but also reduces complexity in inter-component data flows, with the Base Component acting as a central arbitration mechanism.

4.3.1 Queues

A key element of the APMU runtime is its FIFO queue library driver, which uses a circular buffer to store variable-sized objects. The queue is represented by the `queue_obj_t` struct, holding payloads of arbitrary size. By employing a circular buffer and linking objects via a

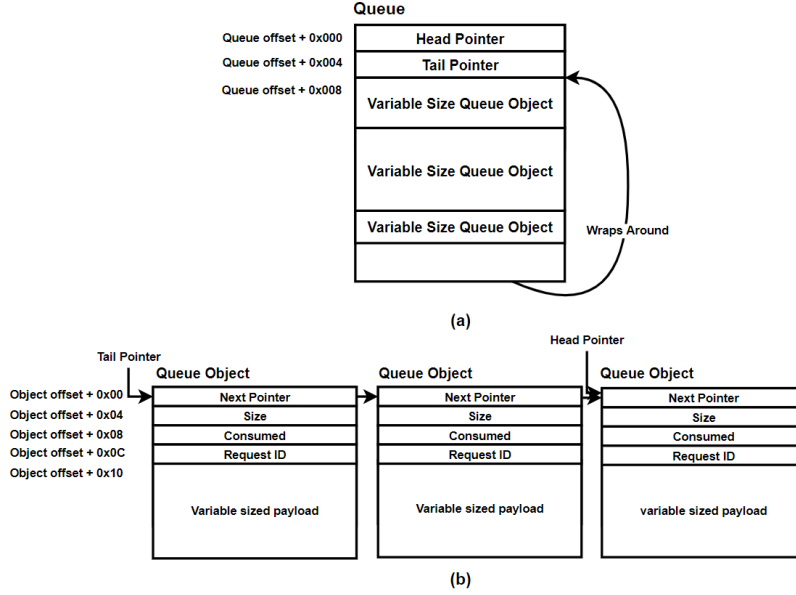


Figure 4.3: Queue Block Diagram (a) A visual representation of fields of queue instance within the memory (b) Shows the internal fields of queue objects and how they are linked with each other and with the head and tail pointer.

next pointer, the design provides efficient memory usage. Each object resides contiguously in memory, and the circular layout enables wrapping around the end when trailing space is lower than required, which allows the object to be placed at the start of queue. Although memory allocation and deallocation occur internally, the queue's start address and size are defined in the configuration file (see Section 4.3.6). These queues follow a single-producer, single-consumer model. Because the APMU does not support atomic operations, a given queue can only be accessed by one producer and one consumer at a time. Queues are identified by a unique `queue_id` parameter which is also specified in the configuration file.

To maximize flexibility and avoid unnecessary data copying, the push and pop operations are each divided into two distinct phases: one for allocating the buffer and another for committing or finalizing its use. This decoupled approach enables producers and consumers to directly work on the queued data before committing to the push or pop operation. Figure 4.3 shows the queue's internal data structures, including how the queue objects are arranged and linked.

A summary of supported queue operations is given here. Most fields of the queue data structure are described here as well; the role of the Request ID will be explained later.

Listing 2 Queue operations

```
1 void queue_init(uint32_t queue_id);
2
3 queue_obj_t* queue_push_get_buffer(uint32_t queue_id, uint32_t size);
4 void queue_push_buffer(uint32_t queue_id, queue_obj_t* obj);
5
6 queue_obj_t* queue_pop_get_buffer(uint32_t queue_id, uint32_t* size_out);
7 void queue_pop(uint32_t queue_id);
```

Following is the description of these fields:

1. **Queue Init:** The `queue_init` function takes the start address of the queue and initializes two 32-bit variables at fixed memory offsets (start address and start address + 4) to store the head and tail pointers. By fixing these addresses, different system entities such as APMU Core or the VMs are able to operate on the queue without complex synchronization mechanisms.

2. **Queue Push:** The queue push operation is split in two stages:

Queue Push Get Buffer: The `queue_push_get_buffer` function requests a contiguous region of memory of desired size. It allocates space for a new `queue_obj_t` at the next available location in the circular buffer without altering the head or tail pointers. This allows the system entities to directly write their payload into the returned buffer, performing custom formatting as needed before finalizing the operation.

Queue Push Buffer: After the producer has populated the payload, the function `queue_push_buffer` updates the necessary pointers and internal structures to commit the newly prepared object to the queue. This step adjusts the head pointer and the `next` pointer of the previous head to the new object. Queue's internal state for the push operation is not modified unless this operation is called. Hence, if this function is not called after `queue_push_get_buffer` function, the next call to `queue_push_get_buffer` will return the buffer starting same address and any changes to the buffer will be overwritten.

This push operation can wrap around the end of the queue buffer if the object exceeds the available space after the head pointer. In that case, the `queue_push_get_buffer` will return the buffer from start of queue if there is space, or null otherwise.

3. **Queue Pop:** Similarly, the pop operation is also split into two distinct stages:

Queue Pop Get Buffer: Calling `queue_pop_get_buffer` returns a pointer to the next available object in the queue, if one exists. This function does not immediately modify the queue tail pointer. Instead, it grants the consumer access to the payload, enabling the consumer to process the data in-place or copy it. This approach is conducive to zero-copy consumption patterns, as the consumer may parse or analyze the payload directly within the queue’s memory region.

Queue Pop: Once the consumer has finished processing the retrieved object, the function `queue_pop` is called to advance the tail pointer and remove the object from the queue. Consumed flag is also set to indicate that the queue object is safe to be deleted. Unless this function is invoked the `queue_pop_get_buffer` function would return the same buffer on the next call.

4.3.2 Components

Building on the concepts introduced in Section 4.3, components are modular pieces of code designed to provide specific functionality. Each component is associated with events through an event bitmask and an event handler, enabling it to perform defined actions whenever these events occur. To support event detection, components must also reserve and configure the appropriate counters; this setup is carried out during installation via the component’s initialization (init) hook. In addition, each component defines a request handler, which is invoked when a corresponding request is received. The init and exit hooks ensure that the component can be properly initialized at installation and can exit gracefully upon uninstallation, releasing any resources it previously held.

At the core of this component-based architecture lies the `component_t` structure, which provides a common abstraction for all components. Listing 3 shows the declaration of this data structure. We further discuss each field of `component_t` type:

1. **Component ID:** A unique identifier for the component, starting at 1. The ID 0 is reserved for the Base Component, presented in Section 4.3.4. This ID is also referenced as the Request ID in queue objects (see Figure 4.3), indicating which component should handle a given request. For static components, IDs should be uniquely assigned to components and for dynamic components, the hypervisor is responsible for selecting the component ID based on the configuration parameters set in a configuration file (see Section 4.3.6).
2. **Event Trigger Bitmask:** A 32-bit unsigned bitmask specifying the counters that trigger the component’s event handler. Each set bit index corresponds to a counter

Listing 3 Component data structure

```
1  typedef struct {
2      uint32_t component_id;
3      uint32_t component_bitmask;
4      event_handler_t component_event_handler;    \ \ Function pointer type
5      request_handler_t component_request_handler; \ \ Function pointer type
6      void* init_hook;
7      void* exit_hook;
8  } component_t;
```

index; when that counter is updated, the component's event handler is invoked. This bitmask is used by event-triggered scheduler which we will in the next section.

3. **Event Handler:** A function invoked when a detected event matches any set bits in the component's bitmask. By setting multiple bits, the component can handle multiple events through a single handler. The detailed mechanism for how the event handler is called by the event-triggered scheduler will be discussed in the next section.
4. **Init Hook:** A function executed at the time of component installation to perform initial setup. This function may allocate resources, configure counters or other hardware used by the component, or carry out any other one-time initialization tasks. All the counters that are specified in the event trigger bitmask should be enabled and configured here.
5. **Exit Hook:** A function executed upon component uninstallation to ensure a clean exit. This may include freeing resources, undoing hardware configuration changes, or restoring system state as needed. It may release any counters and disable any counter interrupts that were set.
6. **Request Handler:** A function called whenever a request arrives with a Request ID matching the component's ID. The request payload, provided as a contiguous memory block, is passed to this function. Different request types can be distinguished using an additional ID parameter within the payload, allowing a single request handler to manage various request types. The components are responsible to implement the additional ID parameter. When discussing Base Component in detail we will see how this request handler is invoked when the corresponding request arrives and how an additional ID parameter is used to differentiate between install and uninstall requests.

Components are integrated into the system using the `component_install` function, which takes a `component_t` reference as input. These components can either be static components which are present at the time of APMU image generation, or be dynamically installed as we will discuss ahead. During installation, the function associates the component's bitmask and event handler with the event-triggered scheduler and connects its request handler to the internal Base Component, ensuring that incoming requests with matching IDs are properly routed. Consequently, all desired system components must be installed before they can operate.

4.3.3 Event Triggered Scheduler

An event-triggered, non-preemptive scheduler is implemented using the wait-for-pending (WFP) instruction. When the WFP instruction returns, it provides a bitmask representing detected events. This bitmask is then compared against the registered components' event bitmasks using a logical AND operation. If the comparison yields a nonzero result, the corresponding component's event handler is invoked. In the following, we examine the data structures and functionalities involved in the event-triggered scheduler.

The events library defines an `event_map_t` data structure, illustrated in Listing 4. This structure holds a 32-bit unsigned bitmask and a function pointer to an event handler that accepts a 32-bit unsigned argument. From this data structure, an event list of configurable size is created. Each entry in the event list stores a component's event bitmask and its associated event handler. The indexing scheme is straightforward: the component's ID directly corresponds to the index in the `event_list`, ensuring simple and direct access. As components are installed, their respective entries in the `event_list` are populated with their bitmasks and event handlers. The `event_list` is initialized to zero and updated as new components are registered. Helper functions are provided to add or remove events from the `event_list`, maintaining a clean and manageable structure.

A global 32-bit bitmask, `global_event_bitmask`, represents the logical OR of all component event bitmasks stored in the `event_list`. Listing 5 shows the `process_events` function, which is central to the scheduler's operation. This function executes an infinite loop that relies on the WFP instruction, using `global_event_bitmask` as the input bitmask to WFP instruction, to detect and wait for events. The `global_event_bitmask` allows to WFP instruction to wait on all the counters specified in the `global_event_bitmask` and hence all the components. Once an event occurs and WFP returns with another bitmask representing the counters that triggered the return. The `process_events` function iterates over the `event_list`, performing a bitwise AND between the returned bitmask

Listing 4 Data structures associated with events

```
1  uint32_t global_event_bitmask;
2
3  // Event handler function prototype
4  typedef void (*event_handler_t)(uint32_t);
5
6  typedef struct {
7      uint32_t bitmask;
8      event_handler_t handler;
9  } event_map_t;
10
11  event_map_t event_list[MAX_EVENTS];
```

Listing 5 Main event triggered scheduling function

```
1  void process_events() {
2      while(1){
3          uint32_t bitmask = pmu_wait_for_pending(global_event_bitmask);
4
5          for (size_t i = 0; i < MAX_EVENTS; i++) {
6              if (bitmask & event_list[i].bitmask)
7                  event_list[i].handler(bitmask);
8          }
9      }
10 }
```

and each registered component's bitmask. If the result is nonzero, the corresponding event handler is called with the returned bitmask as an argument. The iteration of this loop is always limited to `MAX_EVENTS`. We do not keep a track of the number of installed events as this would make the installation and uninstall of event handlers relatively complex or may involve traversing non-contiguous data structures such as a linked list.

The `process_events` function implements a infinite loop which continuously looks for events and invokes the respective event handlers when events are detected. The event triggered scheduler can also be modified to work based on the wait for overflow instruction, where only the Line 3 in Listing 5 will be modified to invoke wait for overflow instruction instead of wait for pending instruction. This allows the system to become responsive on various types of events and this behavior is central to the functionality of the APMU Core

Listing 6 Base component install structure

```
1  typedef struct {  
2      uint32_t payload_id;  
3      component_t component;  
4  } base_component_install_t
```

software runtime.

4.3.4 Base Component

The base component is an internal component with an ID of zero, responsible for key system functions such as request delegation and dynamic component installation and uninstallation. It is initialized at system startup. By having a bitmask set to 32'b1, the base component is triggered by the `process_events` function whenever the first counter is updated. Consequently, the base component reserves the first available counter. Its request handler is not externally linked; instead, it is invoked internally by the base component's own event handler.

As the primary arbiter of incoming requests to the APMU, the base component reads and processes requests from the APMU request queue. Other components must have their request handlers registered with the base component. To facilitate this, the base component maintains an array of function pointers, all initially zeroed, where each entry corresponds to a component ID. Thus, when a request arrives, the base component can either handle it directly — if its Request ID is zero — or forward it to the appropriate component's request handler, as depicted in Figure 4.2.

When a request with an ID of zero arrives, it targets the base component and represents either a dynamic component installation or uninstallation request. We examine the request structure used for this purpose. To parse the requests and to distinguish between installation and uninstallation, the received payload is cast onto the structure shown in Listing 6. The `payload_id` field indicates whether the request is for installation or uninstallation, and the `component` field of the `base_component_install_t` structure holds the details of the component to be installed or uninstalled. Based on the action, the base component either uses `component_install` or `component_uninstall` function from the component library to install or uninstall the component.

4.3.5 Miscellaneous Drivers

Additionally, the following libraries and drivers are also developed:

1. **Debug Print:** A minimal printf implementation for debug printing is also included. Depending on system configuration, the driver can either print to memory or to the UART peripheral. Support to print in memory is provided and can be configured through configuration options. In case print to UART is desired, a putchar function implementation is required to link the printf function to the UART peripheral.
2. **Timer:** A driver for timer is included which utilizes the `mcycle` CSR register of RISC-V to compute time, providing common functionality such as getting the current time and creating delays based on clock cycles or absolute time.
3. **APMU Counter Library:** A library header which abstracts the counter interface with descriptive macros is also provided to facilitate the programming of event counters.

4.3.6 Configuration

Configuration options are supplied through a configuration file. Within this file, parameters such as the number of queues, their sizes, and their starting addresses in the DSPM can be specified. Additionally, the configuration clarifies whether each queue serves as a request queue or a response queue. The file specifies the starting component ID for the installation of components dynamically as well as the maximum number of components that can be installed dynamically. The file also offers the option to direct the debug print library to a memory region instead of a UART peripheral. In that case, it provides the start address and size of the memory buffer for printing. The full configuration file format is described in [Appendix A](#).

4.4 Application Core Runtime Software

In a virtualized environment, VMs cannot be trusted to take control of a critical component such as APMU. To still be able to interact with the APMU, the hypervisor maps certain APMU DSPM pages to individual VMs using two-stage translation, allowing communication via the queues. It may also map counters and configuration registers to specific VMs.

If heightened security is required, the hypervisor can refrain from exposing these resources directly to the VMs and instead mediate all access to these resources through hypercalls. This support, however, is not currently provided.

Bao is a type-1 static partitioning hypervisor with builtin cache coloring support. It provides a suitable environment to integrate the APMU drivers and services. Since the APMU depends on the host system for its startup process, a dedicated bootup driver is included in Bao. Additionally, because the hypervisor manages the APMU request queue and may have additional queues assigned to it, a driver for these queues is also incorporated into Bao.

4.4.1 APMU Bootup

A driver is provided as a patch to Bao to handle APMU bootup. The driver copies the APMU binaries into the APMU's ISPM and DSPM before starting the APMU core via the Status and Boot Address registers (see Section 2.5.3). The APMU binary is divided into text and data sections and linked with Bao using the same infrastructure used for linking VM image binaries. This mechanism supplies global variables that indicate the loading addresses for the text and data sections, which the driver uses to copy these sections into the APMU ISPM and DSPM respectively. Additional configuration parameters for specifying physical addresses of ISPM and DSPM are integrated into Bao's configuration file. Appendix A details these configuration parameters.

4.4.2 APMU Bidirectional Communication

The hypervisor's APMU communication driver uses the same queue library described in Section 4.3.1. For the request queue, the starting address and size are provided via configuration parameters. Because the hypervisor is responsible for mapping DSPM pages as well as any counters or configuration registers to the VMs, thereby facilitating their direct interaction with the APMU, these pages can be integrated as devices using Bao's existing device configuration infrastructure.

For bare-metal or RTOS-based applications, a similar queue library is provided, enabling direct interaction with the APMU's respective response queue without incurring the overhead of the hypervisor. However, hypervisor is responsible for exposing the relevant memory regions from the APMU SPMS to the VMs. The VMs have to poll on the response queue to determine if a response is available.

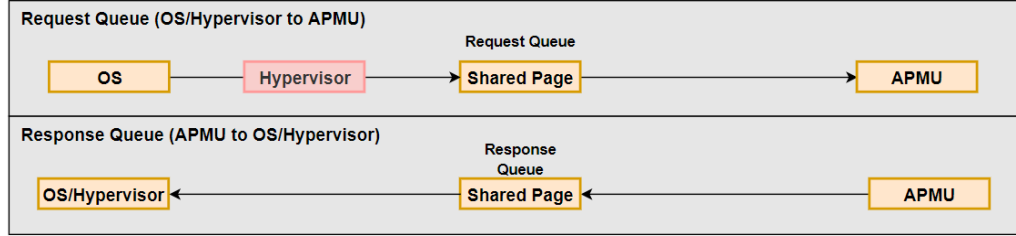


Figure 4.4: Block diagram representing example queue communication

By implementing hypercalls, instead of allowing a VM’s OS to directly interface with the APMU, the hypervisor can validate the data being written to the queue. It can also verify the data before sending a trigger to the component indicating that a request has been received. This approach is illustrated in Figure 4.4.

4.5 Dynamic Component Deployment

The APMU provides advanced system-wide monitoring capabilities, enabling its core to perform complex arithmetic and make decisions based on global performance counters. In many scenarios, there is a need to load additional monitoring functionalities at runtime. For example, the OS might need to temporarily deploy a certain component to collect some tracing data and remove the component once the data collection is completed. To support this, the APMU Core Runtime Software allows dynamic installation of components through the Base Component, with the hypervisor acting as the facilitator.

Since the APMU is a critical component which can monitor system-wide events and has read and write access to the system memory, it poses a security threat if a malicious component is installed, as the components on the APMU do not run in a protected environment. With a trusted entity, hypervisor, facilitating the installation of dynamic components, verification can be performed at the hypervisor level to ensure no malicious component is deployed on the APMU dynamically. Discussion of these verification methodologies is not in the scope of this thesis.

By leveraging a hypercall interface, virtual machines can deploy new components onto the the APMU. In the hypercall, VMs provide the address and size of an ELF-format object file representing the dynamic component. This object file can be generated locally by the VM or acquired through other channels, such as a network interface.

Memory for these dynamic components is reserved within the APMU via a linker script. The hypervisor employs a memory allocator to manage this region and places the dynamically loaded components there. A dynamic linker in the hypervisor processes the ELF object file, performs necessary relocations according to the allocated space in the APMU ISPM and DSPM, and copies the relocated binaries into the allocated memory. Once this setup is complete, the hypervisor assigns the component an ID based on the configuration parameters and sends a request to the APMU, via the request queue, to install the component. The Base Component receives and processes this request. The ID that is selected by the hypervisor is set as the return value of the hypercall for the OS.

As discussed in Section 4.3.4, when the base component receives a component installation request, it maps the payload to `base_component_install_t` structure and extracts the `component_t` structure instance included in the request. This instance contains the function pointers, the component ID and the bitmask needed to represent the new component. The Base Component then uses the `component_install` function to register the event handler and bitmask with the event list, as well as integrate the request handler into its internal registry, effectively completing the component installation process. Uninstall requests can be made to the base component with the ID of dynamic component to effectively remove the dynamic component from the APMU. For this, the OS is responsible to send the component ID to the hypervisor which it wishes to uninstall. The hypervisor first checks if the ID belongs to the OS that requested the uninstall, and then proceeds to send an uninstall request to the base component.

A dynamic component must adhere to a specific model recognized by the dynamic linker. In particular, the C code must provide the exact same function definitions and names as those specified in the `component_t` structure shown in Listing 3. Additionally, the variables `component_id` and `component_bitmask` must be declared as global variables. This approach ensures that the symbols for the functions and the global variables appear in the object file and the dynamic linker can use them to construct a `component_t` instance and issue an installation request to the Base Component. In the following sections, we further examine the design and implementation of the memory allocator and the dynamic linker.

4.5.1 Memory Allocator

This section details the memory allocator implemented for managing dynamically deployed components within the ISPM and DSPM regions. A sample linker script for the APMU is given in Appendix A showing how regions are allocated. The allocator is designed to

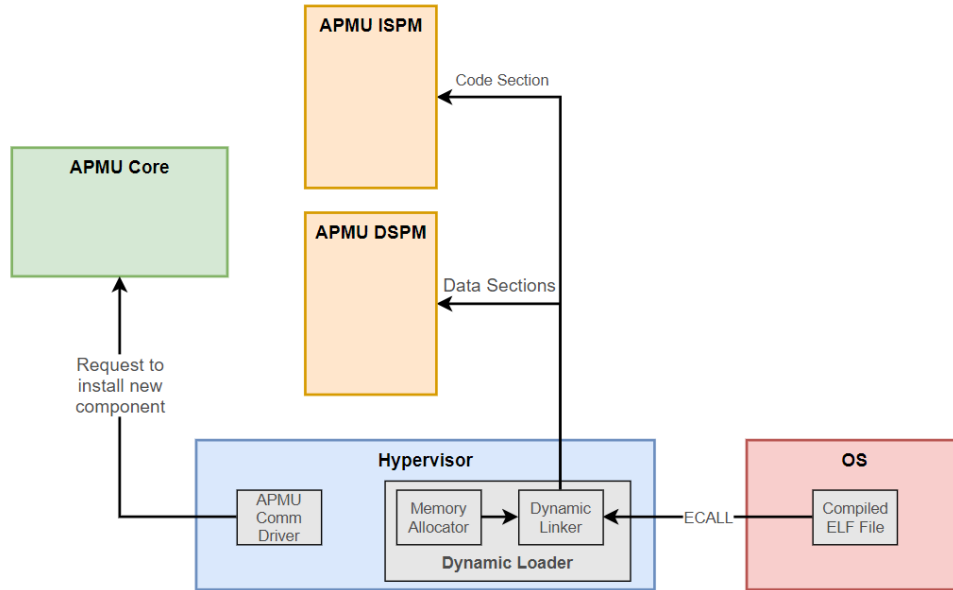


Figure 4.5: Installation of dynamic component

provide efficient, contiguous memory allocation and deallocation, supporting the dynamic installation and removal of components while minimizing fragmentation and overhead.

Functionality Overview

The allocator allows dynamic requests for contiguous memory from the ISPM and DSPM pools allocated for dynamic component deployment. When memory is freed, it is reclaimed and returned to a pool of available blocks, enabling subsequent allocations to reuse previously freed regions. By merging adjacent free blocks, the allocator also reduces external fragmentation and preserves larger contiguous regions for future allocations.

Design

The allocator operates over memory regions allocated for deployment of dynamic components by the APMU. These regions include a block of memory from the ISPM and DSPM, each managed independently. Each region is represented by a structure holding information about its base address, maximum capacity, and a linked collection of available

blocks, tracked using a free-list approach. The design principle is straightforward, focusing on maintaining a simple allocation mechanism that efficiently handles memory reuse and minimizes fragmentation.

Implementation Details

Initially, the implementation sets up a single free block spanning the entirety of each memory region, ensuring that all memory is available at the start. In initialization function, `apmu_init_allocator`, both ISPM and DSPM regions are prepared with their initial free blocks. In the allocation function, `apmu_mem_<x>spm_alloc`, a suitable free block is identified and reserved; if insufficient space is available, the function returns a null pointer. Deallocation reinserts freed memory into the free-list and checks for adjacent free blocks, merging them to maintain larger contiguous segments. Overall, this approach ensures that the allocator remains efficient and adaptable, making it suitable for deployment of dynamic components.

4.5.2 Dynamic Linker

In this section, we describe a lightweight dynamic linker designed for baremetal environments, capable of running without OS support. It dynamically relocates and loads object files into memory, enabling a type-1 baremetal hypervisor such as Bao, to deploy code on APMU from object files. The linker operates independently of system libraries, only using minimal baremetal functions. This linker is designed specifically to support the deployment of dynamic components.

Functionality Overview

The linker processes ELF object files provided by the hypercall interface and prepares them to be deployed to APMU Core as dynamic components. It takes the ELF file as input, parses its sections, performs relocations based on addresses given by memory allocator, and produces two output binaries: one for the text section (instructions) and one for the data section. These binaries are loaded into the APMU's memory regions at addresses allocated by the memory allocator.

The dynamic linker relies on the RISC-V ELF ABI specification [17] to handle relocations for loads, stores, function calls, and jumps. It supports a subset of relocation types,

sufficient for any code that is developed for a component, while maintaining a low memory footprint implementation compatible with limitations of a lightweight type-1 hypervisor.

Since the dynamic linker is also responsible for generating the request to APMU to install the dynamic component, it looks for the symbols with the exact same names as defined in component data structure definition in Listing 3 and stores them to be able to package as a request and send the request to APMU to install the component.

Design

The dynamic linker executes the following major steps. First, it parses the ELF object file to identify critical sections (.text, .data, .rodata, .bss), the symbol table, and relocation entries using ELF headers. All the data sections are combined to produce one data section to be stored in DSPM. Then it uses the memory allocator to request chunks of memories from the ISPM and DSPM with the sizes based on the sizes of text and combined data sections. Next, it relocates the text section to the address for ISPM and the combined data section to the address for DSPM, updating the symbols in the symbol table section (.symtab) accordingly. Then, it adjusts references in the .text section using a subset of RISC-V relocation types, each relocation updating the target address based on the symbol's relocated address. Finally, it produces two binaries: one for the text and combined data sections each, which are to be written to the DSPM which are further loaded using `load_pmu_sections` function.

Implementation Details

The implementation adheres to the ELF specification, using the following components:

1. **ELF Parsing:** The ELF file is parsed to extract section headers, the symbol table, and relocation entries. This is accomplished using the `Elf32_Ehdr`, `Elf32_Shdr`, `Elf32_Rela`, and `Elf32_Sym` structures. A brief description of these structures and symbols is given in Table 4.2
2. **Relocation Application:** Relocation entries are iterated over and applied based on their type. Details of the relocation operations are given in RISC-V ELF processor specific application binary interface (psABI) [17]. For example, `R_RISCV_32` relocation type involves directly writing the symbol's new address plus an addend to the target location. `R_RISCV_BRANCH` requires recalculating branch offsets, considering the relocated symbol's address and the current instruction's location.

Table 4.2: Summary of Relevant ELF Structures

Definition	Description
Elf32_Ehdr	ELF header structure, contains metadata about the ELF file, including offsets to section headers and program headers.
Elf32_Shdr	Section header structure, used to locate and describe sections in the ELF file.
Elf32_Sym	Symbol table entry structure, used to represent symbols (functions, variables) and their attributes.
Elf32_Rela	Relocation entry with addend, used to describe relocations that require an explicit addend.

The supported relocation types are given in Appendix A. These relocations are processed using switch-case logic which ensures extensibility, allowing additional relocation types to be supported as needed.

3. **Loading:** Once the text and combined data section are relocated, the binary files are copied into the respective addresses in the ISPM and DSPM. A request based on the saved symbols for component_t structure is generated and sent to the APMU Base Component to install the dynamic component.

4.6 Evaluation

This section evaluates the performance of the implemented software stack, focusing on execution time and code size through a representative software setup. The analysis demonstrates the stack’s lightweight design, efficient resource usage, and ability to handle dynamic components, making it suitable for resource-constrained environments.

4.6.1 Platform

The APMU was integrated into a RISC-V SoC platform based on four CVA6 (RV64) cores supporting the hardware hypervisor extension [48, 15, 30]. The SoC features a 32 KB L1 cache, a 1 MB LLC, and a 4 GB DDR4 main memory, along with peripherals such as I2C/SPI controllers and UART. AXI4-based Event Units (EVUs) were placed between each core and the LLC, as well as after the LLC, enabling comprehensive event monitoring. The SoC design, including the integrated APMU, was implemented on a VCU118 FPGA

board as presented in prior work [23]. Although the APMU core and application cores share the same clock frequency, the DDR4 memory operates at a higher speed leading to much lower contention from the DDR4 memory.

4.6.2 Software Setup

Next, we are going to describe the software setup used to evaluate execution times on both the APMU core and the application cores. We initialized an event list of size 10 and introduced a “simple component” (Component ID 1) that uses a bitmask to trigger its event handler whenever the counter at index 1 is updated. Upon event handler invocation, the component prints a message to APMU memory using the Debug Print library. We also instrumented the APMU software to measure various operations, recording execution times (in clock cycles) and storing these measurements in a designated memory region via the Debug Print library.

On the application core, we ran Bao and a simple bare-metal VM that sends a request to the APMU through the request queue and reads the response from the response queue, both in a cold cache state. The request directed to the simple component triggers its request handler, which writes a message to the response queue. We measured the queue operations’ timings on both the APMU core and the application core, and also triggered the simple component’s event handler by updating the associated counter once from the application core.

Next, we dynamically deployed a “latency binning” component (Component ID 2) that uses a counter at index 2, programmed to measure the read request latency to the LLC. The internal ALU of this counter records the maximum latency observed and resets the counter afterwards. Each time a new latency value is updated, the event-triggered scheduler invokes the latency binning component’s event handler, which increments a corresponding bin. In this manner, the component can produce a latency distribution of the workload. The object file of this component, generated externally using the LLVM-based cross-compiler, is packaged into the baremetal VM. The VM makes a hypercall to Bao, which employs the dynamic linker and memory allocator to load this code into the APMU’s memory. The base component then installs the latency binning component.

After this deployment, the VM runs the “disparity” benchmark from the San Diego Vision Benchmark Suite. During execution, the latency binning component increments latency bins according to observed read request latencies. Once the benchmark completes, the application core sends an uninstall request for the component, after which the component outputs its bin values in the print memory via the Debug Print library and is removed

from the APMU. Throughout this process, we measure timing for various operations on both the APMU core and application core.

4.6.3 Execution Time

Table 4.3 summarizes the timing measurements for different operations on both the APMU core and the application core. In the initialization, we measured the time required to register a static component and initialize an event list of size 10 for the event-triggered scheduler. Then we measure the latency of event triggered scheduler to invoke the event handler of the simple component, once the event for the simple component is detected. For the queue operations, we measured execution times for both the APMU core and the application core. Measurements on the application core were taken in a cold-cache state. The results indicate that queue operations on the application core require significantly more clock cycles than on the APMU core, likely due to the application core’s more complex memory hierarchy.

We also measure the timing associated with deployment of the latency binning component and its event handler. The event handler took 23 clock cycles. The complete operation from the return of wait for pending instruction to the next wait for pending instruction execution took 43 clock cycles. On the application core, the dynamic linker required 17295 clock cycles to process an object file of 5.2KB containing 114 relocation entries. The generated text and combined data sections were 0.6KB and 58 bytes respectively. The timing for memory allocation and deallocation was measured for the ISPM and DSPM individually.

We also evaluated the timing for base component operations, including the request routing mechanism. We measured the time taken for the base component to route a request to the simple component, from the return of the event-triggered scheduler to the invocation of the request handler. Additionally, we also measured the time required by the base component to perform the install and uninstall operations of dynamic components. This was done by measuring the time when the event-triggered scheduler returns till the point where the operation was completed and returned from the base component.

To evaluate the latency for triggering a component’s event handler based on its Component ID, we modified the component ID of the simple component from 1 to 9 and measured the latency. Figure 4.6 reveals a linear trend in latency as the Component ID increases. This is due to higher loop iteration counts required to locate the correct event handler. It is to be noted that no event handler was called prior to the invocation of the event handler for the simple component.

Operation	Clock Cycles
<i>Initialization:</i>	
Registration of a static component	9
<i>Event Triggered Scheduler Latency:</i>	
Latency for simple component	11
<i>Queue Operations On APMU Core:</i>	
Queue push get buffer	48
Queue push buffer	16
Queue pop get buffer	18
Queue pop buffer	18
<i>Queue Operations On Application Core:</i>	
Queue push get buffer	352
Queue push buffer	55
Queue pop get buffer	153
Queue pop buffer	84
<i>Base Component Operations:</i>	
Request routing	57
Component install	162
Component uninstall	150
<i>'Latency Binning' Component Operations on Application Core:</i>	
Memory allocation in SPM	142
Memory deallocation in SPM	52
Dynamic linker processing	17295
<i>'Latency Binning' Component Operations on APMU:</i>	
Latency binning event handler	23

Table 4.3: Execution time breakdown for various operations in the software setup

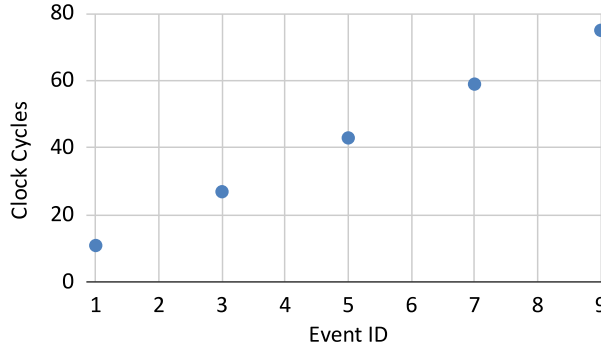


Figure 4.6: Effect of component ID on event handler trigger latency

All the timing measurements were performed repeatedly and the measurements turn out to be constants for both the application core and the APMU core, and thus, no distribution is available for these timing measurements.

4.6.4 Code Size

For a generated APMU software binary with the simple component—including queue push and pop operations, the printf driver, and the timer driver—the text section takes up to 2.1 KB of memory, while the combined data sections, including `.rodata` and `.bss`, take up to 0.6 KB. This implementation is fairly representative of a simple implementation as this does not contain any complex pieces of code or functionalities and only includes the basic libraries.

The entire APMU runtime software stack, including all drivers, consists of fewer than 1.5k lines of code, showcasing its lightweight and efficient design. Similarly, the application core runtime software’s dynamic loader, which includes the dynamic linker and memory allocator, is also implemented in under 1.5k lines of code. Additionally, the dynamic linker processes object files in place, minimizing stack space consumption.

4.7 Conclusion

In this chapter, we see the implementation of a lightweight software stack, both in terms of code size as well as execution time, for the APMU Core which is supported by its evaluation

and addresses the first design goal. The toolchain developed through LLVM is capable of recognizing the custom instructions. The APMU core runtime software framework provides a rich environment including the event-triggered scheduler and components which can be used together to implement event driven logic. The communication queues allow the APMU and the application cores to interact with each other. We further see how the software allows the integration with Bao hypervisor which also provides the ability to load code on the runtime on the APMU Core.

The ability to debug the software on the APMU core is limited due to underlying hardware limitations. As APMU is not attached to a debug module which can allow for halting and stepping through instructions, the debugging capability is limited to the use of Debug Print library and the use of techniques such as the use of infinite loops to halt the execution by modifying the program.

The security implications of the addition of overall APMU infrastructure have not been completely evaluated. The possibility of additional side-channels introduced can be further investigated. However, we consider the hypervisor a trusted entity, which takes control of the APMU. The hypervisor mediates the interface of OS with APMU where OS has to request the hypervisor to transmit a communication object or request the install of a new component. Security measures within the hypervisor can prevent malicious requests from the OS and hypervisor can choose to reject those requests. Using two-stage address translation, hypervisor can map only particular regions of APMU's memory to the OS which can be used by the OS to read the response directly from the APMU and hypervisor mediation is not needed in this case, improving on efficiency of communication.

It is clear from the evaluation that the execution times for various operations on the APMU core remains under 50 clock cycles apart from the Base Component Operations. This signifies the efficient design of various libraries. Timing for APMU core for various operations apart from the dynamic linker remain in few hundred clock cycles. The dynamic linker processes a large file in memory, and thus requires significant time. This streamlined and minimal design underscores the software stack's efficiency and its resource-conscious approach, ensuring minimal overhead while maintaining a rich functionality.

Chapter 5

Conclusion

This thesis explored the challenges and solutions associated with isolation, performance monitoring, and resource management in modern multiprocessor SoCs. Starting from the need to partition and isolate workloads in complex heterogeneous environments, it examined virtualization technologies and memory bandwidth isolation techniques, focusing on how shared resources such as caches and memory interfaces impact system-level predictability and performance. Implementing MemGuard at the hypervisor level on an ARMv8-based platform demonstrated how memory bandwidth regulation can mitigate contention, improving performance stability in scenarios where mixed-criticality tasks share the same hardware.

A key contribution of this thesis was the design and integration of software support for the Advanced Performance Monitoring Unit (APMU). By developing an LLVM-based toolchain, event-triggered runtime mechanisms, communication queues, and integration with both hypervisor and guest OS layers, the APMU framework supports fine-grained observation of hardware events. This architecture enables dynamic monitoring and analysis of system behavior, facilitating informed decision-making and automated runtime optimizations. The framework’s flexibility and extensibility lay the groundwork for a wide range of applications—both for improving system efficiency and ensuring robust isolation.

Overall, this thesis provided insight into the interplay between virtualization, performance isolation, and advanced performance monitoring. Through the evaluation we saw the limitations of MemGuard and the APMU’s design, with an integrated core, provides a platform which can be used to perform finer regulation with events from all across the system. Hence, one of the lessons learned from our evaluation is that the considerations for regulation should be taken at hardware level as well because techniques that solely

rely on software may incur additional overhead. The results and infrastructure developed here pave the way for more predictable, secure, and efficient embedded and server-class computing systems, serving as a valuable foundation for future research and industrial deployments.

There are several avenues for extending this work. Additional hardware support, such as primitives for directly halting or throttling cores, would simplify bandwidth partitioning enforcement and enable finer-grained regulation through techniques like MemGuard and MemPol. Similarly, incorporating more Event Units (EVUs) into diverse system components—from accelerators to I/O subsystems—would facilitate more comprehensive tracing and correlation of events, revealing complex interactions among various entities and guiding more nuanced isolation policies.

Beyond performance isolation, the APMU’s flexibility lends itself to a broad range of applications. Its tracing capabilities could be harnessed to detect and mitigate side-channel attacks, enforce stricter real-time guarantees in safety-critical scenarios, or ensure stable bandwidth and latency regulation under varying workloads. Evaluating and refining the APMU’s functionality in these directions will help establish it as a robust and versatile performance monitoring framework for future multiprocessor systems.

References

- [1] Siemens AG. Jailhouse hypervisor. <https://github.com/siemens/jailhouse>, 2024. Accessed: 2024-12-02.
- [2] ARM. *ARM Cortex-A53 MPCore Processor Technical Reference Manual*, 2023. Revision: r0p4.
- [3] Arm Developer. Virtualization Host Extensions. <https://developer.arm.com/documentation/102142/0100/Virtualization-host-extensions>, 2024. Accessed: 2024-11-13.
- [4] Arm Limited. *Arm CoreSight Architecture Specification*, 2024. Accessed: 2024-12-18.
- [5] Arm Limited. *Arm Memory Partitioning and Monitoring (MPAM)*, 2024. Accessed: 2024-12-18.
- [6] Arm Limited. *Performance Monitor Unit*, 2024. Accessed: 2024-12-18.
- [7] Michael Bechtel and Heechul Yun. Denial-of-service attacks on shared cache in multicore: Analysis and prevention. In *2019 IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, pages 357–367. IEEE, 2019.
- [8] Brian K Bray, William L Lynch, and Michael J Flynn. *Page allocation to reduce access time of physical caches*. Computer Systems Laboratory, Stanford University, 1990.
- [9] Alex D Breslow, Ananta Tiwari, Martin Schulz, Laura Carrington, Lingjia Tang, and Jason Mars. Enabling fair pricing on hpc systems with node sharing. In *Proceedings of the international conference on high performance computing, networking, storage and analysis*, pages 1–12, 2013.

- [10] Weifan Chen, Ivan Izhibirdeev, Denis Hoornaert, Shahin Roozkhosh, Patrick Carpanedo, Sanskriti Sharma, and Renato Mancuso. Low-overhead online assessment of timely progress as a system commodity. In *35th Euromicro Conference on Real-Time Systems (ECRTS 2023)*. Schloss Dagstuhl-Leibniz-Zentrum für Informatik, 2023.
- [11] Marcello Cinque, Domenico Cotroneo, Luigi De Simone, and Stefano Rosiello. Virtualizing mixed-criticality systems: A survey on industrial trends and issues. *Future Generation Computer Systems*, 129:315–330, 2022.
- [12] RISC-V GNU Toolchain Contributors. Risc-v gnu toolchain. <https://github.com/riscv-collab/riscv-gnu-toolchain>, 2024. Accessed: 2024-12-07.
- [13] Jack Dongarra, Kevin London, Shirley Moore, Phil Mucci, and Dan Terpstra. Using papi for hardware performance monitoring on linux systems. In *Conference on Linux Clusters: The HPC Revolution*, volume 5. Linux Clusters Institute, 2001.
- [14] Linux Foundation. Real-time linux patch. <https://wiki.linuxfoundation.org/realtime/start>, 2024. Accessed: 2024-12-02.
- [15] RISC-V Foundation. *RISC-V Instruction Set Manual*, 2024. Release ISA-449CD0C.
- [16] Santosh Gondi, Siddhartha Biswas, and Heechul Yun. Performance isolation for real-time applications on multicore platforms using pallocc and memguard. In *Proc. of Real-Time Linux Workshop (RTLWS)*, 2014.
- [17] RISC-V Non-ISA Working Group. Risc-v elf platform abi specification. <https://github.com/riscv-non-isa/riscv-elf-psabi-doc>, 2024. Accessed: 2024-12-02.
- [18] RISC-V Non-ISA Working Group. Risc-v supervisor binary interface documentation. <https://github.com/riscv-non-isa/riscv-sbi-doc>, 2024. Accessed: November 19, 2024.
- [19] Irfan Habib. Virtualization with kvm. *Linux Journal*, 2008(166):8, 2008.
- [20] Jorg Herter, Peter Backes, Florian Hauptenthal, and Jan Reineke. Cama: A predictable cache-aware memory allocator. In *2011 23rd Euromicro Conference on Real-Time Systems*, pages 23–32. IEEE, 2011.
- [21] Intel Corporation. *Intel Resource Director Technology (Intel RDT)*, 2024. Accessed: 2024-12-18.

- [22] Ravi Iyer. Cqos: a framework for enabling qos in shared caches of cmp platforms. In *Proceedings of the 18th annual international conference on Supercomputing*, pages 257–266, 2004.
- [23] Mohammed Sajjad Jafri. A centralized system performance monitoring infrastructure. Master’s thesis, University of Waterloo, 2024.
- [24] Ram Srivatsa Kannan, Michael Laurenzano, Jeongseob Ahn, Jason Mars, and Lingjia Tang. Caliper: Interference estimator for multi-tenant environments sharing architectural resources. *ACM Transactions on Architecture and Code Optimization (TACO)*, 16(3):1–25, 2019.
- [25] Richard E Kessler and Mark D Hill. Page placement algorithms for large real-indexed caches. *ACM Transactions on Computer Systems (TOCS)*, 10(4):338–359, 1992.
- [26] Hao Li, Xuefei Xu, Jinkui Ren, and Yaozu Dong. Acrn: a big little hypervisor for iot development. In *Proceedings of the 15th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*, pages 31–44, 2019.
- [27] Jiang Lin, Qingda Lu, Xiaoning Ding, Zhao Zhang, Xiaodong Zhang, and P Sadayappan. Enabling software management for multicore caches with a lightweight hardware support. In *Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis*, pages 1–12, 2009.
- [28] LLVM Project. *Code Generator*, 2024. <https://llvm.org/docs/CodeGenerator.html>.
- [29] LLVM Project. *TableGen Programmer’s Reference*, 2024. <https://llvm.org/docs/TableGen/ProgRef.html>.
- [30] lowRISC. Ariane: A risc-v processor. <https://github.com/lowRISC/ariane>, 2024. Accessed: 2024-12-02.
- [31] lowRISC contributors. Ibex: A small risc-v core. <https://github.com/lowRISC/ibex>, 2024. Accessed: 2024-12-02.
- [32] Arm Ltd. *Arm Architecture Reference Manual for A-profile Architecture*, armv8-a and armv8-r edition (ddi0487ka) edition, 2024. Accessed: 2024-12-02.
- [33] Arm Ltd. *Learn the Architecture - Generic Timer*, 2024. Accessed: 2024-12-02.
- [34] Renato Mancuso. Jailhouse-rt, 2024. Accessed: 2024-12-18.

- [35] José Martins, Adriano Tavares, Marco Solieri, Marko Bertogna, and Sandro Pinto. Bao: A lightweight static partitioning hypervisor for modern multi-core embedded systems. In *Workshop on next generation real-time embedded systems (NG-RES 2020)*. Schloss Dagstuhl-Leibniz-Zentrum für Informatik, 2020.
- [36] Anup Patel and Co. Xvisor: extensible versatile hypervisor. <https://xhypervisor.org/>, 2024. Accessed: 2024-12-02.
- [37] Perf Wiki Contributors. Perf Wiki: Performance Engineering Wiki, 2025. Accessed: 2025-01-12.
- [38] Gerald J Popek and Robert P Goldberg. Formal requirements for virtualizable third generation architectures. *Communications of the ACM*, 17(7):412–421, 1974.
- [39] LLVM Project. Risc-v backend - llvm documentation. <https://llvm.org/docs/RISCVUsage.html>, 2024. Accessed: 2024-12-07.
- [40] Xen Project. Xen project: The hypervisor powering a secure and open cloud. <https://xenproject.org/>, 2024. Accessed: 2024-12-02.
- [41] Roger Pujol, Mohamed Hassan, Hamid Tabani, Jaume Abella, and Francisco Javier Cazorla. Tracking coherence-related contention delays in real-time multicore systems. In *Proceedings of the 38th ACM/SIGAPP Symposium on Applied Computing*, pages 461–470, 2023.
- [42] Bruno Sá, Luca Valente, José Martins, Davide Rossi, Luca Benini, and Sandro Pinto. Cva6 risc-v virtualization: Architecture, microarchitecture, and design space exploration. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 2023.
- [43] Ahsan Saeed, Denis Hoornaert, Dakshina Dasari, Dirk Ziegenbein, Daniel Mueller-Gritschneider, Ulf Schlichtmann, Andreas Gerstlauer, and Renato Mancuso. Memory latency distribution-driven regulation for temporal isolation in mpsoes. In *35th Euromicro Conference on Real-Time Systems (ECRTS 2023)*. Schloss-Dagstuhl-Leibniz Zentrum für Informatik, 2023.
- [44] RISC-V Software Source. Opensbi: Open source supervisor binary interface. <https://github.com/riscv-software-src/opensbi>, 2024. Accessed: November 19, 2024.
- [45] Noriaki Suzuki, Hyoseung Kim, Dionisio De Niz, Bjorn Andersson, Lutz Wrage, Mark Klein, and Ragunathan Rajkumar. Coordinated bank and cache coloring for temporal protection of memory accesses. In *2013 IEEE 16th International Conference on Computational Science and Engineering*, pages 685–692. IEEE, 2013.

- [46] David Tam, Reza Azimi, Livio Soares, and Michael Stumm. Managing shared l2 caches on multicore systems in software. In *Workshop on the Interaction between Operating Systems and Computer Architecture*, pages 26–33, 2007.
- [47] Trusted Firmware Project. Trusted Firmware-A Documentation. <https://trustedfirmware-a.readthedocs.io/en/latest/>, 2024. Accessed: 2024-11-13.
- [48] Luca Valente, Asif Veeran, Mattia Sinigaglia, Yvan Tortorella, Alessandro Nadalini, Nils Wistoff, Bruno Sá, Angelo Garofalo, Rafail Psiakis, M Tolba, et al. Shaheen: An open, secure, and scalable rv64 soc for autonomous nano-uavs. In *HCS*, pages 1–12, 2023.
- [49] Prathap Kumar Valsan, Heechul Yun, and Farzad Farshchi. Taming non-blocking caches to improve isolation in multicore real-time systems. In *2016 IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, pages 1–12, 2016.
- [50] Sravanthi Kota Venkata, Ikkjin Ahn, Donghwan Jeon, Anshuman Gupta, Christopher Louie, Saturnino Garcia, Serge Belongie, and Michael Bedford Taylor. Sd-vbs: The san diego vision benchmark suite. In *2009 IEEE International Symposium on Workload Characterization (IISWC)*, pages 55–64. IEEE, 2009.
- [51] Inc. VMware. VMware workstation and fusion: Desktop hypervisor products. <https://www.vmware.com/products/desktop-hypervisor/workstation-and-fusion>, 2024. Accessed: 2024-12-02.
- [52] Andrew Wolfe. Software-based cache partitioning for real-time applications. In *Third International Workshop on Responsive Computer Systems*, 1993.
- [53] Xilinx. *Zynq UltraScale+ Device Technical Reference Manual*, 2023. UG1085.
- [54] Xilinx. ZCU102 Evaluation Board. <https://www.xilinx.com/products/boards-and-kits/ek-u1-zcu102-g.html#information>, 2024. Accessed: 2024-11-13.
- [55] Hailong Yang, Alex Breslow, Jason Mars, and Lingjia Tang. Bubble-flux: Precise online qos management for increased utilization in warehouse scale computers. *ACM SIGARCH Computer Architecture News*, 41(3):607–618, 2013.
- [56] Heechul Yun, Gang Yao, Rodolfo Pellizzoni, Marco Caccamo, and Lui Sha. Memguard: Memory bandwidth reservation system for efficient performance isolation in multi-core platforms. In *2013 IEEE 19th Real-Time and Embedded Technology and Applications Symposium (RTAS)*, pages 55–64. IEEE, 2013.

- [57] Xiao Zhang, Sandhya Dwarkadas, and Kai Shen. Towards practical page coloring-based multicore cache management. In *Proceedings of the 4th ACM European conference on Computer systems*, pages 89–102, 2009.
- [58] Alexander Zuepke, Andrea Bastoni, Weifan Chen, Marco Caccamo, and Renato Mancuso. Mempol: policing core memory bandwidth from outside of the cores. In *2023 IEEE 29th Real-Time and Embedded Technology and Applications Symposium (RTAS)*, pages 235–248. IEEE, 2023.

APPENDICES

Appendix A

APMU Software Stack

A.1 Linker and Sample Memory Map for APMU

In this section we detail a sample memory map for the APMU Core Software through the linker script. As shown below, we define three macros: `STACK_SIZE`, `DYN_COMP_ISPM_SIZE` and `DYN_COMP_DSPM_SIZE`. Further we defines the ISPM and DSPM memory regions. In the ISPM, the text section is placed followed by the section for installation of text section of dynamic components, called `dyn_comp_ispm` section. This section is aligned to 4 byte boundary and given the respective size previously defined. The DSPM is allocated the data, rodata, bss, stack and the `dyn_comp_dspm` sections. The sample linker script is shown below:

```
1  STACK_SIZE      = 0x300;
2  DYN_COMP_ISPM_SIZE = 0x400;
3  DYN_COMP_DSPM_SIZE = 0x400;
4
5  ENTRY(_start)
6
7  /* Memory regions for Instruction and Data Memories */
8  MEMORY
9  {
10     ISPM (rx) : ORIGIN = 0x10426000, LENGTH = 0x1000
11     DSPM (rw) : ORIGIN = 0x10427000, LENGTH = 0x1000
12 }
13
```

```

14 SECTIONS
15 {
16     .text : {
17         . = ALIGN(4);
18
19         *(.text)
20         *(.text.*)
21         . = ALIGN(4);
22     } > ISPM
23
24     .dyn_comp_ispm (NOLOAD) : {
25         . = ALIGN(4);
26         _dyn_ispm_start = .;
27         . += DYN_COMP_ISPM_SIZE;
28         _dyn_ispm_end = .;
29     } > ISPM
30
31     .rodata : {
32         . = ALIGN(4);
33         /* Small RO data before large RO data */
34         *(.srodata)
35         *(.srodata.*)
36         *(.rodata);
37         *(.rodata.*)
38         . = ALIGN(4);
39     } > DSPM
40
41     .data : {
42         . = ALIGN(4);
43         _sdata = LOADADDR(.data);
44         _sdata = .;
45
46         /* Small data before large data */
47         *(.sdata)
48         *(.sdata.*)
49         *(.data);
50         *(.data.*)
51         . = ALIGN(4);
52         _edata = .;
53     } > DSPM

```

```

54
55     .bss :
56     {
57         . = ALIGN(4);
58         _bss_start = .;
59         /* Small BSS before large BSS */
60         *(.sbss)
61         *(.sbss.*)
62         *(.bss)
63         *(.bss.*)
64         *(COMMON)
65         _bss_end = .;
66         . = ALIGN(4);
67     } > DSPM
68
69
70     .stack (NOLOAD) : {
71         . = ALIGN(4);
72         _stack_start = .;
73         . += STACK_SIZE;
74         _stack_end = .;
75     } > DSPM
76
77     .dyn_comp_dspm (NOLOAD) : {
78         . = ALIGN(4);
79         _dyn_dspm_start = .;
80         . += DYN_COMP_DSPM_SIZE;
81         _dyn_dspm_end = .;
82     } > DSPM
83 }

```

A.2 Configuration Files

In this section we discuss two configuration files, one for the APMU Core runtime software and the other which Bao already uses to define the Virtual Machines.

A.2.1 APMU Configuration

For the APMU Core runtime software, first, the configuration file describes the number of counters implemented in hardware. Then we define the print configuration which determines if print to memory region is desired instead of printing to UART peripheral. If it is desired, the start address of the memory region as well as the size of the region is provided. This configuration file also contains the description of the communication queues, by defining the queue ID, the start address, size and if the queue is request queue or a response queue. A sample configuration file for the APMU Core software is shown below:

```
1  struct config config = {
2      // Number of implemented counters
3      .num_counter = 8,
4
5      // Print memory configuration
6      .print_config = {
7          .is_print_memory = 1,    // Enable print to memory instead of UART
8          .print_start_addr = 0x10428C00, // Ignored if is_print_memory = 0
9          .print_size = 0x200,      // Ignored if is_print_memory = 0
10     },
11
12     // Communication queue configuration
13     .num_queues = 2,
14     .queues = {
15         {
16             .queue_id = 1,          // Queue ID
17             .start_address = 0x10429000, // Start address of queue
18             .size = 0x400,          // Size of queue in bytes
19             .is_request = 1         // This is request queue
20         },
21         {
22             .queue_id = 2,          // Queue ID
23             .start_address = 0x10429400, // Start address of queue
24             .size = 0x800,          // Size of queue in bytes
25             .is_request = 0         // This is response queue
26         }
27     }
28 };
```

A.2.2 VM Configuration in Bao

Bao hypervisor provides a configuration file for the description of the VMs. This file specifies details such as the number of VMs, the path to each VM's binary file, the number of CPUs allocated to each VM, and the devices assigned to them. Below is an example snippet of the configuration file:

```
1  VM_IMAGE(baremetal_image,  
    ↪  XSTR(/path/to/bao-baremetal-guest/build/cva6/baremetal.bin));  
2  VM_IMAGE(linux_image, XSTR(/path/to/bao-baremetal-guest/build/cva6/linux.bin));  
3  
4  struct config config = {  
5      CONFIG_HEADER  
6      .vmlist_size = 2,           // Number of VMs  
7      .vmlist = {  
8          {  
9              .image = {           // Image physical addresses  
10                 .base_addr = 0x80200000,  
11                 .load_addr = VM_IMAGE_OFFSET(baremetal_image),  
12                 .size = VM_IMAGE_SIZE(baremetal_image),  
13                 .inplace = true  
14             },  
15             .entry = 0x80200000,  // Entry Address  
16             .platform = {  
17                 .cpu_num = 1,      // CPU allocated to VM  
18                 .region_num = 1,   // VM virtual memory regions  
19                 .regions = (struct vm_mem_region[]) {  
20                     {  
21                         .base = 0x80200000,  
22                         .size = 0x20E00000,  
23                     }  
24                 },  
25                 .dev_num = 5,      // Devices allocated to VM  
26                 .devs = (struct vm_dev_region[]) {  
27                     {  
28                         .pa = 0x10000000,  // Physical Address  
29                         .va = 0x10000000,  // Virtual Address  
30                         .size = 0x00010000, // Size  
31                         .interrupt_num = 1, // Interrupts  
32                         .interrupts = (irqid_t[]) {1}
```

```

33         },
34         // And so on ...

```

To allocate pages of APMU DSPM or the counter and configuration registers, the device configuration option is used in which the physical address, the virtual address with respect to the VM and the size of the mapping is described. Other configuration options for device description can be left empty. This would automatically tell Bao to map the specified physical address into the VM's address space when Bao boots up.

To enable Bao to be able to load the APMU binaries into the APMU ISPM and DSPM upon bootup, we use the existing infrastructure to specify VM images, which puts certain global variables within the code which are picked by the APMU bootup driver. Furthermore, we have to specify the addresses of ISPM and DSPM. Additional macros are also used to allow the APMU bootup driver to locate the binaries and copy them over to the ISPM and DSPM respectively. A diff of the added configuration options is shown below with some sample values:

```

1  VM_IMAGE(baremetal_image,
    ↪  XSTR(/path/to/bao-baremetal-guest/build/cva6/baremetal.bin));
2  VM_IMAGE(linux_image, XSTR(/path/to/bao-baremetal-guest/build/cva6/linux.bin));
3  +VM_IMAGE(apmu_text, XSTR(/path/to/apmu-os/build/apmu_text.bin));
4  +VM_IMAGE(apmu_data, XSTR(/path/to/apmu-os/build/apmu_data.bin));
5
6  struct config config = {
7      CONFIG_HEADER
8  +  .apmu_config = {
9  +      .is_apmu_enabled = 1,           // true
10 +      .hyp_loads_code = 1,           // true
11 +      .ispm_base_addr = 0x10426000,   // Base address of APMU ISPM
12 +      .text_load_addr = VM_IMAGE_OFFSET(apmu_text),
13 +      .text_size = VM_IMAGE_SIZE(apmu_text),
14 +      .dspm_base_addr = 0x10427000,   // Base address of APMU DSPM
15 +      .data_load_addr = VM_IMAGE_OFFSET(apmu_data),
16 +      .data_size = VM_IMAGE_SIZE(apmu_data),
17 +      .dyn_comp_start_id = 5,
18 +      .dyn_comp_num = 5,
19 +  }
20      .vmlist_size = 2,                 // Number of VMs
21      .vmlist = {

```

A.3 Supported Relocation Types

The dynamic linker supports a subset of relocations which are specified in the RISC-V ELF Specifications [17]. These relocations are enough for the current envisioned use cases for the APMU dynamic components. The supported relocation types are shown in Table A.1.

Table A.1: Supported RISC-V Relocation Types

Relocation Type	Description
R_RISCV_NONE	No relocation required.
R_RISCV_32	Absolute 32-bit address relocation.
R_RISCV_BRANCH	Relocation for branch instructions, adjusting relative offsets based on the relocated symbol address.
R_RISCV_JAL	Relocation for jump and link (JAL) instructions, adjusting the offset for jumps to relocated symbol addresses.
R_RISCV_CALL_PLT	Relocation for function calls using the Procedure Linkage Table (PLT), combining AUIPC and JALR.
R_RISCV_PCREL_HI20	High 20 bits of a PC-relative address relocation.
R_RISCV_PCREL_LO12_I	Low 12 bits of a PC-relative address relocation (I-type instruction).
R_RISCV_PCREL_LO12_S	Low 12 bits of a PC-relative address relocation (S-type instruction).
R_RISCV_HI20	Relocation for the high 20 bits of a symbol address.
R_RISCV_LO12_I	Relocation for the low 12 bits of a symbol address (I-type instruction).
R_RISCV_LO12_S	Relocation for the low 12 bits of a symbol address (S-type instruction).