A Centralized System Performance Monitoring Infrastructure

by

Mohammed Sajjad Jafri

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, 2024

© Mohammed Sajjad Jafri 2024

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

In this thesis, we introduce a centralized performance monitoring infrastructure. In the current computing landscape, performance monitoring architectures are becoming more and more important for different academic and industrial applications. Performance counters reveal valuable insight into the functioning of the platform. This information can then be exploited for debugging applications, improving performance, identifying bottlenecks, and much more. In our proposed infrastructure, we envision a configurable Advanced Performance Monitoring Unit (APMU) connected to a set of monitoring Event Units (EVU) that are installed in various hardware system IPs across the platform. These EVUs send hardware event information to the APMU. The APMU has smart counters that are capable of operating on the incoming events, and an instruction processor that can implement any desired software mechanisms on the counter data. Our design allows for an efficient collection and correlation of event data, allowing the APMU to get a more holistic insight into the system behaviour, revealing microarchitecture-specific information. We intend to allow users the ability to develop EVUs for IPs relevant to them. For instance, in the implementation phase of this work, we developed an AXI4-based Snooping Unit as a concrete example of a custom-EVU. Therefore, to help integrate such custom EVUs with an APMU infrastructure, we also standardize an EVU-APMU interface. We provide the specification for this interface, ensuring that users can connect any custom-EVU to an APMU, as long as both abide by the interface specification.

In this work, we implement two design IPs. One is the previously mentioned AXI4-based Snooping Unit and the other is a RISC-V compliant APMU. We also provide a software stack to support programming on its processor. The implemented design is emulated on an AMD Virtex UltraScale+ FPGA VCU118 device. To evaluate the implementation of our design, we present the hardware synthesis results for the FPGA, and the execution results of a latency-based regulation case study, demonstrating the functionality of our design.

Acknowledgements

I want to express my deepest gratitude to Professor Rodolfo Pellizzoni, my supervisor, for his invaluable guidance, expertise, and patience during the course of my research. Working under him has been one of the most enlightening experiences of my life. I am grateful and honoured to have had this opportunity.

I sincerely thank Professors Hiren Patel and Seyed Majid Zahedi for taking the time and effort to review my thesis and providing valuable feedback.

I offer my heartfelt appreciation to Abdur Rahman and Gopishankar Thayyil for their assistance in my research. Their contributions and efforts have added much value to this endeavour. Thanks also goes out to Aravind, Danesh and Wafic, for making my experience at the University of Waterloo a pleasant one.

Last but not least, I would like to thank Zahra for her kindness, and for impacting my life in many positive ways.

Dedication

This work is dedicated to my beloved parents for their endless love, patience, and unwavering support. Your sacrifices have made my present a reality. Everything that I am is because of you.

To my lovely sister, I am proud of the person you have become. Thank you for adding to my happiness.

To my love, Nida, you are my sunshine. My life is a brighter place with you in it.

Table of Contents

Author's Declaration ii

Abstract iii

Acknowledgements iv

Dedication v

List of Figures ix

List of Tables xi

1 Introduction 1

1.1 Motivation 1

1.2 Problem Statement 2

1.3 Solution 4

1.4 Contribution 5

1.5 Thesis Structure 5

1.6 Acknowledgement 6

2 Background and Related Works 7

2.1 Background: Performance Monitoring in COTS Architectures 7

2.1.1 Hardware Performance Counters . . . 7
2.1.2 Hardware Monitoring and Management Frameworks . . . 8
2.1.3 SoC Performance Monitoring . . . 11
2.1.4 Tracing . . . 13
2.1.5 Software Support . . . 13
2.2 Related Works: Managing Resource Contention . . . 14
2.2.1 Software Approaches: General Purpose and Cloud . . . 15
2.2.2 Software Approaches: Real-Time Systems . . . 16
2.2.3 Hardware Approaches . . . 18

3 System Design **21**
3.1 List of Example Applications . . . 22
3.2 Proposed Design . . . 26
3.2.1 EVU-APMU Interface . . . 27
3.2.2 Event Units (EVU) . . . 30
3.2.3 Advanced Performance Monitoring Unit (APMU) . . . 32
3.3 Software Design . . . 42

4 Implementation **46**
4.1 AXI4 Snooping Unit . . . 46
4.1.1 AXI4 Fundamentals . . . 47
4.1.2 Event Table for the SPU . . . 49
4.1.3 Hardware Design of the SPU . . . 51
4.2 Implemented APMU . . . 57
4.2.1 APMU Interconnect . . . 59
4.2.2 APMU Core and SPMs . . . 59
4.3 Software Support . . . 64

5 Evaluation 65
5.1 Evaluation Platform 65
5.1.1 Introduction 66
5.1.2 Addition of the APMU and AXI4 SPUs to the platform 70
5.2 Hardware Synthesis Results 71
5.3 Case Study: Latency-based Regulation 73
5.3.1 Implementation 78
5.3.2 Interference Tests on PULP 81
5.3.3 Regulation Results 83

6 Conclusions 88

References 91

APPENDICES 100

A Specification: EVU-APMU Interface 101

B Specification: Advanced Performance Monitoring Unit 104
B.1 APMU Control and Status Registers 105
B.2 APMU Ports 106
B.3 Event Filter 107
B.4 Counter Block 107
B.5 APMU Core and Memory Infrastructure 114
B.6 APMU Interconnect 115
B.7 Software Design Considerations 115

List of Figures

3.1 Typical multicore platform, with our proposed centralized performance monitoring infrastructure. . . . 22

3.2 EVU-APMU Interface. . . . 27

3.3 The fields of an event packet. . . . 27

3.4 Event Info bits transmitting request latency and transaction size. . . . 28

3.5 APMU Block Diagram with M EVUs. . . . 33

3.6 Event Filter of an APMU Counter Block. . . . 35

3.7 Counter Block. . . . 36

3.8 Format of a `XLEN`-bit APMU Counter. . . . 37

3.9 `Status` register of the APMU core. . . . 42

3.10 `BootAddr` register of the APMU core. . . . 42

4.1 Connection of an SPU between Manager and Subordinate modules. Figure (a) shows the original AXI bus between the two modules. Figure (b) shows the bus getting routed through a SPU. . . . 47

4.2 SPU Pipeline. . . . 52

4.3 SPU Block Diagram. . . . 53

4.4 An example to explain the functioning of the SPU-CAM. . . . 58

4.5 Ibex Pipeline [1] (writeback stage not included). . . . 61

4.6 Counter-Read Instruction Format. Figure (a) shows the standard RISC-V I-type instruction format. Figure (b) shows the format of our custom counter-read instruction. . . . 62

4.7 Counter-Write Instruction Format. Figure (a) shows the standard RISC-V S-type instruction format. Figure (b) shows the format of custom counter-write instruction. . . . 62
4.8 Wait-for-X Instruction Formats. Figure (a) shows the standard RISC-V R-type instruction format. Figure (b) shows the format of custom WFP instruction. Figure (c) shows the format of custom WFO instruction. . . . 63
4.9 APMU Software Stack. . . . 64

5.1 Evaluation Platform based on PULP, along with APMU and AXI4 SPUs. . 66
5.2 Interference test results. . . . 84
5.3 Regulation Results for the Synthetic Benchmarks. . . . 85
5.4 Regulation Results for the SDVB Benchmark Suite. . . . 86
5.5 Timestamp of the regulation decisions taken by the APMU core during disparity at $\alpha = 50\%$. . . . 87

A.1 The fields of an event packet. . . . 101
A.2 Event Info bits transmitting request latency and transaction size. . . . 102

B.1 APMU Architecture. . . . 105
B.2 `Timer` register of the APMU core. . . . 106
B.3 `BootAddr` register of the APMU core. . . . 106
B.4 `Status` register of the APMU core. . . . 106
B.5 APMU Counter Block. . . . 109
B.6 Format of a `XLEN`-bit APMU Counter. . . . 109
B.7 Format of `EventSelCfg` register. . . . 110
B.8 Format of `EventInfoCfg` register. . . . 110

List of Tables

2.1 PMU events supported by CVA6 family of RISC-V cores. 9

3.1 AXI4 SPU Event Table 32
3.2 ALU Operations and their opcodes 39
3.3 Example of an APMU Memory Map. 44

4.1 A typical address mapping for a computing platform. 51

5.1 Overall resource utilization cost of the system and its primary IPs. 74
5.2 Resource utilization cost of the different SPU configurations. 74
5.3 Resource utilization breakdown of an SPU. 75

A.1 An exemplified EVU Event Table. 103

B.1 Counter Mode of Operation. 108
B.2 ALU operations and their opcodes. 113
B.3 Example of an APMU Memory Map. 116

Chapter 1

Introduction

1.1 Motivation

The saturation of Moore's law has slowed down the yearly performance growth that the computer industry once came to accept as perpetual. However, the performance demands of contemporary computer systems are still growing rapidly, especially in computing and data-intensive domains such as machine learning, scientific modelling, blockchain, etc.

To meet the growing performance demands, computer architects have shifted towards heterogeneous computer architectures that support extensive parallel computing capabilities and consist of multiple processors of different types, multi-tiered memory hierarchies and dedicated hardware accelerators, etc. While this approach has shown considerable improvements in power and compute efficiency over homogeneous computer systems, it has increased the complexity of computing platforms. To derive high performance in such systems, it has become important to ensure that platform resources such as cache, main memory, system bus, I/O devices etc., are being shared efficiently. However, the systems' complexity makes it challenging to evaluate the impact that various components have on one another. Due to the abundant parallelism in modern computers, it becomes difficult to isolate sources of contention. There can be an incredible amount of overlap between services of different applications, leading to complicated patterns of transitive delays, i.e., a request being delayed by another request that itself is delayed due to other requests. For instance, a read request to the last-level shared cache from a core can be stalled due to a contention miss caused by the eviction of the requested cacheline to accommodate a fetch request made by another core. To reload the requested cacheline, the last-level cache

(LLC) has to send a fetch request to the main memory, which could be busy serving misses from other cores.

Additionally, the problem is not solely limited to performance. With the adoption of computers in various fields, users impose additional constraints on these systems. For instance, with the growth of the cloud computing market and the introduction of software-as-a-service (SaaS), platform-as-a-service (PaaS), and infrastructure-as-a-service (IaaS) paradigms, resource fairness has also become a major topic of academic and industrial research. In such systems, the task of extracting high performance is coupled with the objective of ensuring fair resource sharing among all users. Similarly, the adoption of embedded controllers in mixed critical and real-time domains such as aviation, automotive, healthcare, etc., has also seen a rapid increase in the past few decades. In such systems, it is crucial to ensure that certain system-level timing and/or efficiency guarantees are always met. Any failure to meet such guarantees can have disastrous consequences in terms of lives lost and/or infrastructure and resource wasted. Therefore, the task of extracting higher performance from computer systems while ensuring that these additional requirements are met is even more complicated.

1.2 Problem Statement

Researchers in the domain of computer architecture have spent considerable effort in tackling the problem of high-performance requirements coupled with other constraints. Over the years, they have proposed countless approaches which can be broadly categorized on the basis of their technology level, i.e., whether they are software-based, hardware-based, or an intricate mix of the two. A detailed study of some of the proposed approaches is presented in Section 2.2 of Chapter 2. A majority of these approaches rely on real-time information about the system and its various components. In [2], researchers at Google have demonstrated that for warehouse-scale data centers and cloud computing systems, access to hardware-level performance metrics not only leads to accurate isolation of software and hardware bottlenecks in the system, but also aids in implementing better resource management policies. The same argument can be extended to embedded systems as well.

Hardware performance counters (HPC) are widely used in both industrial ([3], [4], [5]) and academic computing platforms to provide researchers and system designers access to such low-level performance metrics. Performance counters are installed in various hardware modules, such as processors, caches, memory controllers, etc., to study inter-component interference, the modules' and the overall system behaviour, etc. These small groups of distributed counters are capable of counting hardware events incoming to that specific

module, e.g., the number of hits and misses to a platform-level cache made by different cores, the number of open and closed row accesses to the DRAM, the request latencies on the system interconnect, etc. However, this decentralized approach has an inherent limitation. Since the counters are localized to their corresponding modules, correlation of events occurring in different modules becomes complicated and time-consuming.

For instance, let us consider resource regulation policies. In recent years, the monitoring infrastructure has seen increased usage in the domain of run-time resource regulation policy. A standard resource regulation policy, such as ([6], [7], [8], [9], [10]), broadly has three phases:

1. **Information Collection Phase**

   The mechanism accesses the distributed hardware counters and collects the stored hardware event information.

2. **Reconstruction and Correlation Phase**

   In this phase, the collected information is studied to reconstruct a model of the system or component behaviour, and to correlate the usage of different resources by different compute units. This is often achieved using complex mathematical algorithms which are implemented either in software or via specialized hardware.

3. **Regulation Phase**

   This information is then used in the last stage to implement the desired regulation policy. A regulation or allocation policy controls the distribution and allocation of various system-level resources, such as the platform cache, system bus bandwidth, I/O devices, etc.

As the counters are distributed across multiple modules, the task of reconstructing system behaviour and correlating various event information becomes challenging. There is the added overhead of accessing these counters and the increased software complexity of combining the collected information, both of which, in many cases, are non-negligible. Since many of the allocation algorithms are periodically executed with very small intervals (in ms and μs ranges), any additional overhead is highly undesirable. It either reduces the quality of the allocation policy because less counters can be polled and have their data processed, or increases the time period of the policy, which reduces its efficiency, since the policy now takes longer to adapt to changes in the execution workload and consequent system behaviour.

1.3 Solution

Our objective is to design and implement a centralized performance monitoring infrastructure. We aim to develop an infrastructure where users can install custom monitoring units, called Event Units (EVUs), in individual system components, like processors, caches, memory controllers, etc. These units are responsible for transmitting event information to an Advanced Performance Monitoring Unit (APMU). The APMU is capable of collecting and processing the received information. We provide a design specification for the APMU. The APMU comprises a set of configurable smart counters that can perform arithmetic and logical operations on the received event information, and a specialized instruction processor that is capable of executing software programs on the counter data.

We also allow users to develop their own EVUs targeting components relevant to their application. To support easy integration, we standardize the interface used to connect EVUs to the APMU and provide a specification for the same. The interface specification allows EVUs to be developed independently from the APMU, because different EVUs and APMU implementations can always be connected, provided they both adhere to this specification. In fact, researchers have developed similar monitoring units for measuring contention with shared data for caches and memory ([11], [12]).

Compared to state-of-the-art industrial frameworks such as Intel's RDT [4] and Arm's MPAM [3] that only supports a limited number of resource management mechanisms, our proposed work is designed to be flexible and re-programmable. Our goal is to provide users an infrastructure that supports the execution of complex event-based software mechanisms.

Given that EVUs and the central APMU can aid in the collection and correlation of events from different hardware IPs, they can be used to implement a variety of software mechanisms. Performance monitoring architectures are used extensively to implement runtime regulation and resource management policies. Additionally, they are also being used in other domains. For example, researchers have suggested using performance counters to implement profile-guided optimizations (PGOs) ([13], [14]). Similarly, works such as ([15], [16], [17]) have developed anomaly and intrusion detection systems, employing hardware counters. A recent work [18] also showed that hitherto undisclosed PMU events in Intel CPUs could be effectively used for transient execution attack and side channel attack detection.

Keeping these works in mind, we believe that our design can also be used to implement profiling and tracing functionalities, security mechanisms such as hardware-software control-flow integrity checks, intrusion and anomaly detection, etc.

As a proof of concept, we have implemented an AXI4-based ([19]) Snooping Unit (AXI4

SPU) as an example of a custom EVU, and a RISC-V-compliant APMU as a concrete realization of our APMU specification. We also have used our implementation to execute a regulation case study.

1.4 Contribution

In this thesis, we present a configurable and centralized performance monitoring infrastructure that is implemented from the ground up in RTL, written in SystemVerilog. The infrastructure is then used to implement a resource regulation case study. Our major contributions are the following:

1. We present the specification for a parameterizable Advanced Performance Monitoring Unit (APMU) that supports data collection and processing functionalities. The APMU has a specialized instruction processor that can execute software programs on the collected data. The processor is extended to support additional features.

2. Our vision is for users to develop custom Event Units (EVUs) to monitor hardware IPs fitting their circumstance. With this in mind, we provide an interface specification for connecting EVUs to the APMU, while keeping the design constraints for custom EVUs as nonrestrictive as possible.

3. We implement the proposed specification in RTL for a RISC-V-based FPGA platform. We evaluate the hardware resource cost of the design and compare it against existing components in the system. We also provide a software stack, with library support and pre-defined macros, that eases the use of the APMU and its instruction processor.

4. We execute a latency-based regulation case study to showcase the functionality of our design, and demonstrate how it can be used to implement other advanced software policies.

1.5 Thesis Structure

The rest of the thesis is structured as follows:

1. **Chapter 2** provides relevant background information on performance monitoring infrastructure prevalent in industrial and academic hardware platforms. As a proof of concept, we aim to implement a latency-based regulation mechanism using our proposed architecture. To that extent, we also discuss the related works from that domain in this chapter.

2. **Chapter 3** introduces our proposed design. We begin with the EVU-APMU interface specification, followed by a discussion on the development philosophy for generalized EVUs. Finally, we introduce the APMU design specification.

3. **Chapter 4** demonstrates our RTL implementation of the design proposed in Chapter 3. The implementation was developed for a RISC-V compliant platform. For that reason, the APMU has a specialized RISC-V processor. Additionally, we also present the AXI4-based Snooping Unit, an EVU designed to snoop point-to-point AXI4 buses.

4. **Chapter 5** presents our evaluation of the design, when emulated on an FPGA board. In this chapter, we implement a regulation case study and discuss the results. Additionally, we also present the hardware synthesis reports for the design.

5. **Chapter 6** concludes the thesis with our final remarks and potential future works.

1.6 Acknowledgement

I would like to thank Abdur Rahman, a member of our research team from the University of Waterloo, for providing his support in creating and maintaining the software framework for the AMD Virtex UltraScale+ FPGA VCU118 board [20], and for performing initial software testing. He developed the software infrastructure necessary to run the San Diego Vision Benchmark(SDVB) Suite [21] on our chosen platform.

I would also like to thank Gopishankar Thayyil for his work on the last-level cache (LLC). He implemented a partitioning mechanism in the LLC that allows us to map way-based cache partitions to individual cores; a feature useful for implementing memory regulation policies on multicore platforms. Additionally, he substituted the LLC's random replacement policy with a pseudo-least recently used (pLRU) policy.

Chapter 2

Background and Related Works

This chapter is divided into two sections. In Section 2.1, we present the necessary background information discussing the existing performance monitoring frameworks, and how they are utilized in both static program optimization and dynamic resource management.

Additionally, since in Chapter 5, we focus our evaluation and case study on the issue of contention for shared memory resources, specifically in a real-time setting, we review the work done in this domain in Section 2.2.

2.1 Background: Performance Monitoring in COTS Architectures

Hardware performance monitoring systems are a crucial component of modern computing platforms. They are used extensively in profiling, bandwidth regulation, resource management, etc. For that reason, modern architectures, both open-source and COTS, support performance counters. In this section, we discuss the various performance monitoring frameworks available and their limitations.

2.1.1 Hardware Performance Counters

A standard hardware monitoring framework consists of a number of counters and associated hardware installed in a processor to monitor various low-level events. For instance, Arm and Intel have dedicated performance monitoring units embedded in their processors ([22],

[23], [24]) that count events such as level 1 cache accesses and refills, memory errors, exceptions taken, etc. In some cases, the events can also be filtered on the basis of the privilege level of the processor that initiated them. These counters can be enabled, read from, written to, and programmed to count specific events either by using specialized instructions or by writing to dedicated control and status registers, depending on the PMU framework. Many of these frameworks also support overflow interrupt functionality, in which the processor receives an interrupt when one of its performance counters overflow. Even though these performance counters make a rich amount of hardware information available to programmers, they remain cumbersome to use directly. This is because PMU specifications often undergo extensive changes between CPU generations, as has been the case for both Intel and Arm.

As a final note, we also present the current RISC-V PMU framework as proposed in the “Zicntr” and “Zihpm” extensions [25]. The “Zicntr” extension proposes three counters: CYCLE, TIME and INSTRET, which are used to count CPU cycles, real-time clock cycles, and number of instructions retired, respectively. The “Zihpm” extension proposes 29 additional 64-bit hardware performance counters. However, the set of events and other details are platform-specific. In Table 2.1, we introduce the events supported by the PMU of the CVA6 family of RISC-V cores, which are conveniently also the processors used in our evaluation platform, discussed in Chapter 5. Unfortunately, the current list of performance monitoring events in the CVA6 family is lacking. Events such as number of unaligned accesses to and writebacks by the data cache, unconditional jumps (`j`, `jal`, `jr`, `jalr`), etc., are missing. Another major limitation of RISC-V performance counters is that they are only accessible by the processor itself. This means that a core cannot read the counters of another core directly, which makes the execution of a multicore mechanism difficult.

2.1.2 Hardware Monitoring and Management Frameworks

A major limitation of in-core performance counters is that they are unable to track transactions through the platform’s memory hierarchy, leaving the system architect unaware of information such as the number of platform cache hits or misses generated by a particular core or application. In the current environment, this information is becoming more and more crucial due to growing performance demands. Works such as ([13], [14]) have proposed using hardware monitoring to implement different profile-guided optimization (PGO) frameworks. By exploiting the information about data access patterns across the memory hierarchy, provided by performance counters embedded in the memory subsystem, software designers can implement more efficient memory-aware programs, for high-

Event ID	Event Name	Description
1	L1 I-Cache Misses	Number of misses in L1 I-Cache
2	L1 D-Cache Misses	Number of misses in L1 D-Cache
3	ITLB Misses	Number of misses in ITLB
4	DTLB Misses	Number of misses in DTLB
5	Load Accesses	Number of data memory loads
6	Store Accesses	Number of data memory stores
7	Exceptions	Valid Exceptions encountered
8	Exception Handler Returns	Return from an exception
9	Branch Instructions	Number of branch instructions encountered
10	Branch Mispredicts	Number of branch mispredictions
11	Branch Exceptions	Number of valid branch exceptions
12	Call	Number of call instructions
13	Return	Number of return instructions
14	MSB Full	Scoreboard is full
15	Instruction Fetch Empty	Number of invalid instructions in IF
16	L1 I-Cache Accesses	Number of accesses to Instruction Cache
17	L1 D-Cache Accesses	Number of accesses to Data Cache
18	L1 Cache Line Eviction	Number of Data Cache line eviction
19	ITLB Flush	Number of ITLB Flushes
20	Integer Instructions	Number of Integer instructions
21	Floating Point Instructions	Number of Floating point instructions
22	Pipeline Stall	Number of cycles the pipeline is stalled during read operands
23-31	Reserved	Reserved
23-31	Reserved	Reserved

Table 2.1: PMU events supported by CVA6 family of RISC-V cores.

performance computing (HPC) and embedded systems. For this reason, many industrial manufacturers have developed their own monitoring frameworks.

Additionally, given the growing complexity of modern computer systems, these monitoring platforms are also being used to manage the problem of shared resource contention. In systems running multiple virtual machines (VMs) or applications, system resources such as cache, main memory and system bus are expected to be distributed fairly among all VMs. However, if one of the VMs starts consuming more resources than its expected share, this can adversely affect the performance of other VMs sharing the same resources. For

example, suppose two VMs with differing criticality levels share the same LLC. In that case, the VM with lower criticality may evict the cache lines of the more critical VM. This problem, called the noisy neighbour problem, is very prominent in cloud computing platforms. To resolve this specific problem, Arm proposed the Memory System Resource Partitioning and Monitoring (MPAM) [3], which is an extension to their application profile (A-profile) architectures. The A-profile architecture targets high-performance markets where the impact of shared resource contention is more prominent. The MPAM extension divides the system into a set of partitions wherein each partition is a software environment, defined by its partition space ID and partition number. The partitions also define the different sources for memory transactions. MPAM specifies two categories of partitioning techniques: those for caches and those for memory bandwidth regulation. For caches, the specification provides:

- Cache-portion partitioning, wherein the cache is divided into portions of equal sizes and the portions are allocated to different partitions based on their resource requirements.
- Cache-capacity partitioning, wherein each partition has a minimum and maximum capacity assigned to it. When a victim cache line is to be selected for eviction, the decision is based on the cache capacity occupied by partitions of candidate cache lines. Cache lines belonging to partitions that have exceeded their cache capacity will be evicted first.
- Cache maximum associativity partitioning, wherein each MPAM partition gets allotted a maximum number of ways to use within any cache set, depending on its criticality and requirements.

For memory-bandwidth regulation, MPAM proposes the following:

- Memory-Bandwidth Min-Max Partitioning, wherein each partition is allotted a minimum and maximum bandwidth. The priority of memory transactions made by different partitions depends on the bandwidth consumed by the partitions and the minimum and maximum bandwidth allotted to them. Accesses made by partitions that have not exhausted their minimum bandwidth have a higher priority than those made by partitions that have consumed their minimum but not their maximum bandwidth. Accesses by partitions that have spent their maximum bandwidth have the lowest priority.

- Memory-bandwidth proportional-stride partitioning, wherein each partition has a stride, which is a scaled reciprocal of its weight. Each incoming access is allotted a deadline defined by considering the stride of the accessing partition and the frequency of its accesses.
- Priority partitioning, wherein each partition is assigned a priority. Memory transactions with higher priority take precedence over lower-priority ones.

In a similar vein, Intel proposed the Resource Directory Technology (RDT) Framework [4]. In the RDT Framework, each core is tagged with a Resource Monitoring ID (RMID). Multiple cores can be tagged with the same RMID, which allows users to monitor and regulate applications and VMs running on multiple cores simultaneously. The RDT framework comprises five mechanisms, which work as follows:

- Cache Allocation Technology (CAT) allows for hardware way-based partitioning of the LLC in Intel processors. The user determines the partitioning ratio, considering the criticality and requirements of the workload.
- Cache Monitoring Technology (CMT), when provided with an RMID, offers information about the LLC occupancy for applications or cores associated with said ID. This allows users to monitor the cache utilization of different applications, which aids in profiling.
- Memory Bandwidth Allocation (MBA) allows allocating memory bandwidth to specific VMs or applications. The platform throttles applications that exhaust their allotted bandwidth.
- Code and Data Prioritization (CDP) allows users to exercise software control over code placement and data in the LLC. This mechanism can protect the code of certain VMs or applications, limiting the contention caused by programs with large code footprints.
- Memory Bandwidth Monitoring (MBM) enables users to track multiple VMs or applications and monitor their memory bandwidth for each running thread separately.

2.1.3 SoC Performance Monitoring

Arm's MPAM and Intel's RDT are both technologies that try to ensure quality of service (QoS) guarantees when running multi-tenant applications. Using these frameworks, system

architects can better comprehend the impact that a particular VM or application has on the memory subsystem, and how it affects other VMs or applications that are running in parallel. However, in practice, a modern system-on-chip (SoC) comprises a lot of different components and peripherals.

Keeping this in mind, SoC vendors often include other performance monitoring units (PMUs) tied to specific components. Such PMUs expose registers that can be read through the system interconnection. For example, besides cache and memory-bandwidth utilization, bus latency is another crucial metric in studying contention. By observing the average latency of different applications when run in isolation and in contention, system architects can better understand how resource sharing in the system interconnect impacts the overall performance and QoS of the system. This insight is crucial for making informed decisions regarding resource allocation, load balancing and system optimization to ensure that applications coexist harmoniously while maintaining predictable performance levels. For this purpose, AMD for instance, has a set of AXI Performance Monitor (APM) [26] in its Zynq UltraScale+ MPSoC devices, such as, the ZCU102 [27]. The APMs are a collection of programmable monitors located at multiple points on the PS AXI interconnect for profiling real-time activity over it. Each APM contains a set of programmable event counters that can be configured by software to measure system performance. It also supports cross-probe trigger functionality between counters and logging of event information. Along with the APMs, the Zynq UltraScale+ MPSoCs also have a Platform Monitoring Unit [28], which is responsible for system initialization, power management and system error handling. The PMU contains a programmable user processor that can be used to execute software programs. However, the PMU processor does not have access to event information regarding other platform components, thus limiting its usage.

On the PL side, one can instantiate additional performance monitoring IPs. For instance, the APMs discussed above are based on the AXI Performance Monitoring IP [5] provided by AMD Xilinx, for measuring the performance of AXI-based (AXI full-burst, AXI-Lite, AXI Stream) systems. The IP can measure the bus latency suffered by a specific manager/subordinate in a system. It can be used to count the occurrence of read/write requests, the number of bytes written to/by the agent, the number of clock cycles wasted by the manager in confirming a handshake, i.e., the number of idle cycles between the assertion of the `VALID` signal by the manager and the `READY` signal by the subordinate. Similar to the APM, the IP can also be configured to log the timestamps of specified events, allowing the user to reconstruct the AXI transactions later. The counter and tracing functionality can also be used for real-time profiling of software applications.

2.1.4 Tracing

A problem with the hitherto discussed frameworks is that they collect aggregate metrics, but do not allow system designers and programmers to see information pertaining to individual events. Therefore, information that might vary between individual events of the same type, such as the address accessed by a memory instruction, is not collected by them. This problem has traditionally been solved by generating an execution trace of the program using debuggers, potentially with custom interfaces. Tracing transactions on the system interconnects helps users reconstruct the system behaviour. This is immensely useful for profiling and debugging programs, analyzing performance, identifying bottlenecks, and much more. Considering the potential usefulness of a standardized tracing infrastructure, Arm has proposed the Arm CoreSight specification [29].

Arm CoreSight provides a set of hardware and software components that assist with debugging and tracing capabilities in Arm-based systems. The collected traces make it easier to analyze and optimize software execution on Arm processors. The CoreSight specification provides a range of tools to support debugging of embedded systems. CoreSight offers three categories of trace macrocells: Embedded Trace Macrocell (ETM), which is used to collect processor-driven trace, which includes information about program execution and data; System Trace Macrocell (STM) (and its predecessor Instrumentation Trace Macrocell (ITM)), which provides application-driven trace, i.e., the software writes to memory-mapped STM channels and the STM generates a trace corresponding to the register and value written, and the AHB Trace Macrocell (HTM), which provides detailed trace information for traffic on the CoreSight AHB bus. The generated information is merged into a stream and sent to a trace sink, an endpoint for the trace data. CoreSight supports both on-chip trace storage in RAM via the Embedded Trace Buffer (ETB) and off-chip connections to debug tools via the Trace Port Interface Unit (TPIU). CoreSight also supports cross-triggering, using which multiple cores in a multicore system can trigger or access each other for synchronized debugging and profiling operations.

2.1.5 Software Support

As mentioned earlier, these PMU frameworks often undergo massive changes across generations. Therefore, any knowledge that one might have of a framework implementation might not even port to another implementation of the same framework. Certain events might only be available to specific processors, as is the case for Arm cores. In certain architectures, the counters might not be available to lower privilege levels of the processors, requiring a trap to a higher privilege level for access rights. All this combined with the

fact that there are a limited number of counters, and that interpreting hardware results is often difficult, makes these counters extremely unwieldy for most software programmers.

To ease this burden, professionals have developed a variety of software tools that can act as an interface between a programmer and the hardware PMU. We briefly discuss three of the most common performance monitoring tools below:

- Linux *perf* is the official Linux performance analyzer, with its source code being part of the Linux kernel ([30], [31]). It allows for hardware- and software-based monitoring, with specialized support for Intel and AMD processors. *perf* can be used to count occurrences of a given event in the kernel and user space. It supports three sampling modes: a) event-based sampling, wherein a sample can be recorded when a event threshold is reached; b) time-based sampling, wherein samples are recorded at a given frequency; and c) instruction-based sampling (only supported for ADM64 processors), wherein a processor monitors the instructions it is executing, and samples the events created during the execution. *perf* also supports tracing, allowing users to either set up pre-defined tracepoints in software (static tracing) or by placing tracepoints using uprobes (user) or kprobes (kernel) (dynamic tracing).

- Arm MAP [32] is a software profiling tool developed for Arm-based applications. It is designed for performance debugging and profiling of C, C++, Fortran and Python applications to locate bottlenecks in the system. The tool uses adaptive sampling methods to combine data from multiple processes running on multiple compute units. Arm MAP supports parallel, single- and multi-threaded profiling with typically less than 5% runtime overhead.

- Intel's VTune [33] is a performance analysis software designed for Intel processors based on x86 series of instruction sets. VTune is supported on Linux and Windows operating systems and can work with multiple languages: C, C++, C#, Fortran, OpenCL, Python, Google Go, etc. VTune offers performance analysis for single-threaded and multi-threaded applications and memory debugging capabilities.

2.2 Related Works: Managing Resource Contention

In Chapter 5, we evaluate the functionality of our design by implementing a runtime latency-based regulation policy. Therefore, in this section, we discuss related works in the domain of resource contention, which primarily fall into two categories: a) those that evaluate the impact of resource contention for multi-tenant platforms, and b) those that

regulate different applications based on these evaluations. The proposed literature for (a) recommends using monitoring or tracing infrastructures to gather insight about contention and its ramifications on system performance ([34], [35], [36], [37], [38], [11]). On the other hand, those in (b) leverage the performance characteristics gathered through these monitoring techniques while using counters to monitor and count events whilst regulation ([6], [7], [8], [9], [10]).

2.2.1 Software Approaches: General Purpose and Cloud

The complexity of executing multi-application or multi-threaded workloads on multicore systems leads to high variability in the overall execution time of the application. As mentioned earlier, different applications running on different cores still affect one another because of contention in shared resources.

([34], [35], [36], [37], [38]) are some of the software approaches that aim to evaluate the impact of memory contention for different workloads in, primarily, data center and cloud computing systems. These techniques aim to quantify performance degradation caused due to co-location of memory-intensive applications. The approaches can broadly be categorized as: *static* and *dynamic*. Static approaches such as [34], [35] although effective at predicting application slowdown due to contention, require *apriori* knowledge of the application's behaviour. This limits their deployment in systems that often run unknown applications. Take for instance, the infrastructure-as-a-service (IaaS) cloud service models, where the users provide their own workload to be executed on the monitored infrastructure. Dynamic approaches such as [36], [37] and [38] evaluate applications' slowdown at run time via the hardware performance counters present in the memory subsystem and the processors. The performance of an application under contention can be measured by simply reading the corresponding counters. However, measuring the performance of a running application as if in absence of other applications is challenging. [36] attempts this by periodically pausing all other applications for a very short time. By stalling other applications, the work allows the application-under-analysis to have unfettered access to the system resources, which would be comparable to their performance in isolation. This approach has considerable overhead and non-negligible margin of error because a) all other applications have to be periodically halted, and b) the caches would not be warmed up in the beginning of the measurement period and thus, the peak isolation performance would not be accurately captured. In absence of memory partitioning, this also hurts the other applications because their data might get evicted. [38] attempts to limit the number of pauses that are incurred by applications that are not under analysis to reduce overhead. It does this by categorizing the application-under-analysis' execution into different phases.

It makes the assumption that the performance of an application is relatively unchanged in one phase. Therefore, the framework only has to measure isolation performance once at every phase change. Change in execution of other applications constitute a phase change only when they have massive effect on the application-under-analysis. This is done to limit frequent pausing of other applications. Since these techniques are software-based, they incur high overheads in terms of execution time and are coarse-grained when compared to hardware alternatives. However, they can be readily employed in commercial-off-the-shelf (COTS) platforms to evaluate the impact of contention.

2.2.2 Software Approaches: Real-Time Systems

It is important to develop robust models that can predict performance degradation due to resource contention, because a) they provide system designers with an astute understanding of their architecture, and b) it allows for the implementation of more precise regulation policies. From the perspective of tackling memory interference, the non-trivial solution when dealing with high-criticality tasks has always been to pause all other tasks. But the system takes a massive performance hit in those cases, and researchers in the real-time and other embedded domains have proposed several regulation policies to protect non-critical tasks or the cores running them from the critical ones ([6], [7], [8], [9], [10], [39]).

In [6], the authors propose a memory bandwidth reservation system called MemGuard. The core idea behind the work was to provide guaranteed memory bandwidth to applications by regulating the sum of memory requests made by each core to the DRAM. [6] ensures that the total sum of memory request rates by all cores is always less than the minimum DRAM service rate r_{min}. The minimum DRAM service rate is the minimum bandwidth supported by the memory in worst-case scenarios. This r_{min} service rate is then distributed amongst all cores. This is called the MemGuard budget of the cores. Each core is also assigned a performance counter that increments every time the core makes a memory request. The counters are programmed to overflow when the budget of their corresponding core exhausts. Upon overflow, an interrupt halts the offending core. As [6] is a software-based approach, it runs at each scheduler tick. This is called the MemGuard Period. At the start of each period, the regulator resumes all halted cores, measures the budget consumed by each core in the previous round and replenishes their budgets. The knowledge of the budget previously consumed allows MemGuard to implement a reclamation policy wherein a core that is not fully consuming its own budget can donate it to another that is regularly exhausting its own. In [7], MemGuard is implemented on Linux to counter Denial-of-Service (DoS) attacks. The original MemGuard is extended to separately monitor read and write requests, allowing the OS to set a high budget for

reads and a low budget for writes. Thus protecting the system from writeback buffer DoS attacks. Meanwhile [8] proposed a new implementation of MemGuard that could work for mixed-criticality systems, implemented on Linux KVM-based virtualized environment. In this work, guest applications on Linux communicate their memory bandwidth requirements to the MemGuard kernel module. The design combines MemGuard regulation policy with Completely Fair Scheduler (CFS), the default Linux scheduler, allowing MemGuard to regulate memory accesses made by guest applications.

However, since MemGuard was originally proposed for systems employing COTS memory controllers, it has a large time period which leads to coarse-grained regulation. Additionally, due to the memory overhead of interrupts—that also need to be considered when defining the budgets—MemGuard can only be implemented to regulate one or two memory bandwidth metric at once.

[9] proposed a fine-grained regulation solution that allows defining complex regulation algorithms that combine the data from multiple performance counters. The work, called MemPol, implemented the regulation policy on an auxiliary processor, allowing it to poll the counters at microsecond-level frequency. Additionally, the processor had access to built-in on-chip debug facilities that allowed it to halt cores without causing extra memory accesses via interrupts. The work also suggested employing a global regulator alongside the per-core local regulators. This help redistribute the unused bandwidth between cores. Compared to MemGuard, MemPol allows us to aggregate multiple counter together to implement more holistic regulation policies. This is especially important in the current scenario, since in a modern system, no singular metric can successfully account for all the complexities of the design. However, unlike MemGuard, it needs hardware support in the form a dedicated processor with access to counter information of other processors and IPs. [9] was implemented on the Xilinx ZCU102 [27] board and could run its polling algorithm with a time period of 6.25 μs. This fine-grained approach to regulation better handles the bursty nature of memory transactions, preventing starvation of other cores. In MemGuard, due to the 1 ms time period, it was possible that a single core could drive the DRAM utilization to 100% for a short period of time before getting regulated.

A commonality in the works discussed above is that they all rely on memory bandwidth regulation when trying to limit memory interference between different applications or cores. A different approach proposed by [10] is to directly measure the memory request latencies, and then argue on the properties of the observed latency distribution. The presence of performance counters either in the system interconnect or the memory controller is essential to enable this work. In [10], the authors execute the core-under-analysis in isolation, compute a *timeliness* objective by imposing constraints on the probability of achieving a target execution time. When running in contention, the proposed regulation policy

a) characterizes the memory request latency of the core-under-analysis as a probability distribution function, and b) attempts to alter its shape via regulation of other cores, to ensure that the observed distribution curve complies with the target. A drawback of this approach is that due to hardware limitations, the actual implementation could only generate a discretized distribution function where request latencies are slotted into observation bins. This work employed the APM event monitors introduced in Section 2.1.3 to generate the latency distribution curves.

As a side note, we also present the work done by [39], which relies on a instruction processor, similar to MemPol to methodically assess the progress made by an application while it is running in contention. [39] proposes a Milestone-Based Timely Progress Assessment (MB-TPA) which is based on comparing the intermediate progress of the application against a collection of statically generated milestone instruction addresses. In [39], the control flow of the application-under-analysis is thoroughly profiled by running representative set of inputs to generate its control flow graph (CFG). This CFG is then processed to generate a milestone graph which is then coupled with temporal information. This timed milestone graph represent the intermediate progress stages that the application-under-analysis must traverse. Using CoreSight hardware, the program counter of the core executing the application-under-analysis were compared against this timed milestone graph to gain an insight into the progress made by said application. The authors argue that TPA when coupled with regulation approaches such as the ones mentioned above, can help ensure Timely Progress Integrity (TPI) of the system as a whole. Additionally, TPA can also be utilized to detect execution anomalies.

Lastly, as mentioned before, high-criticality systems depend on tight real-time bounds to meet their guarantees and be safe. However, this is where both Arm's MPAM and Intel's RDT fall short. Works such as [40] and [41] have analyzed the MPAM and RDT frameworks respectively, and observed the following: a) for MPAM, the specification is underspecified at multiple places, as it fails to accurately define how the mechanisms are intended to work, and b) for RDT, the various mechanisms often do not work as expected, and in some cases, their measurement values might even be incorrect. Due to ambiguities and uncertainties in these frameworks, they become unsuitable for real-time applications.

2.2.3 Hardware Approaches

Over the years, researchers have proposed several hardware-based approaches to expand existing COTS monitoring functionalities to better estimate interference in memory subsystem.

The problem of contention in the memory subsystem, especially caches, is often exacerbated when shared data is involved, as shown by ([42], [43]). The works implemented synthetic benchmarks consisting of multiple threads in different execution settings: sequentially i.e., one after the other on a single processor, and in parallel on different processors, accessing different amount of private and shared data. [42] observed a 3.8x slowdown in the execution time for the parallel with shared data scenario compared to the sequential one. The authors argued that in case of real-time and processing intensive applications, the memory coherency overhead, if not accounted for, can lead to deadline violations.

Given the impact of contention due to cache coherence, works such as [11] have attempted to develop frameworks to accurately measure coherence-related contention. [11] proposed a cache coherence monitoring infrastructure called Remote Protocol-Contention Tracking (RPCT) which relies on performance counters in the private and shared caches of the cores. When a core requests exclusive access to a cache line that is already present in the private cache(s) of other core(s), it suffers a delay because the said cache line has to be invalidated from cache(s) of other core(s). Under RPCT, coherence-related contention is tracked from the perspective of the cores generating it instead of the cores suffering it. In other words, the cache of core i which owns the data starts counting contention cycles suffered by core j only when it receives a fetch request from core j for that data. In RPCT, if there are N cores then the cache controller of core i needs $N-1$ counters to measure contention caused by core i on every other core. In shared caches, the RPCT protocols demands that there must be $(N-1)^2 \approx N^2$ counters, to count contention between every two pairs of cores.

RPCT showcases the need for a centralized performance monitoring system. In the proposed infrastructure, if one wants to measure the contention caused by core i on all other cores then they need to aggregate all counters in the cache controller of core i and all the counters in the shared cache that measure contention caused by core i. Performing these kind of aggregation operations can induce latency overheads, which can worsen in case of memory hierarchies with multiple levels of shared caches. This makes the proposed-RPCT infrastructure difficult to employ for dynamic regulation or resource allocation policies. However, if the coherence-related contention information for each core was stored in a centralized location then it could be used to implement less pessimistic and more accurate regulation policies.

[12] proposed a similar work for tracking contention in the memory controller, by developing a hardware mechanism, called MCTrack to capture inter-core interference in memory. The authors developed a monitor that can be installed in the memory controller to track the state of requests of different cores in the controller. The monitor contains a set of contention counters, one per core, which increment on the basis of information gathered

from the controller, i.e., which core is getting delayed due to which other core. Similar to our vision with the monitoring EVUs, the authors implemented their monitor in a non-intrusive manner, such that it does not affect the performance or behaviour of the memory controller. In fact, both RPCT and MCTrack are good examples of custom-EVUs that can be standardized using our specification provided in Chapter 3 and hooked up to the APMU to collect contention information across the entire memory subsystem.

Lastly, researchers have also proposed hardware mechanisms to monitor contention due to concerns of resource fairness, especially for cloud markets. We briefly discuss some of them here. [44] attempts to estimate the *unfairness* in the memory subsystem, which is defined as the ratio of slowdown an application suffered compared to that suffered by another. The slowdown is the ratio of the time taken by the application when running in contention to that in isolation. This work attempts to calculate the excess delay the application suffers running in contention, accounting for shared cache interference, DRAM bus and bank conflict interference, and DRAM row-buffer interference. If the unfairness between applications exceeds a specified threshold, then the system will dynamically throttle the requests of applications that have enjoyed an unfair advantage over others. A different approach by [45] involves the use of auxiliary tag stores to measure the interference cycle count for a portion of the cache sets. The cycle count calculated by sampling those sets is then extended to account for interference in all cache requests. Both [44] and [45] can also be seen as examples of custom-EVUs, capable of monitoring fairness in the caches and DRAMs.

Chapter 3

System Design

This chapter proposes a centralized performance monitoring architecture, independent of the hardware platform and the technology employed. In the proposed architecture, we envision a system with Event Units (EVUs) embedded into various platform components, such as the processors, interconnections, caches, memory controllers, I/O ports, etc., to monitor the hardware events associated with them. An example of such a system is given in Figure 3.1. These EVUs can transmit events and additional event-related information to an Advanced Performance Monitoring Unit (APMU). The APMU comprises a number of event counters and a specialized instruction processor. The counters can be configured to either count the occurrences of these events or operate on the additional event data. The APMU processor can be programmed to execute any desired software algorithm. The APMU is connected to the platform bus, facilitating communication between the APMU and different system modules. It is also connected to the system interrupt controller, allowing it the ability to trigger interrupts when warranted.

To ensure users freedom in designing their own custom-EVUs, we do not define an EVU specification. But we do recommend a set of design guidelines. We need to ensure compatibility between the APMU and these custom-EVUs, and to that effect, we specify a standardized interface between the two. Following our interface specification, any custom-EVU can be connected to any APMU implementation. In the APMU specification, we specify a minimum set of functionalities for the APMU IP, so that software mechanisms can be developed independently of the APMU implementation. For the sake of conciseness, we focus on the main design rationale in this chapter. A full specification for the AMPU and the EVU-APMU interface is given in the appendices.

This chapter has three sections. In Section 3.1, some of the intended applications of

the architecture are discussed. These applications help justify our design decisions. The proposed hardware design is explained in Section 3.2. And the software component of the design is discussed in Section 3.3.

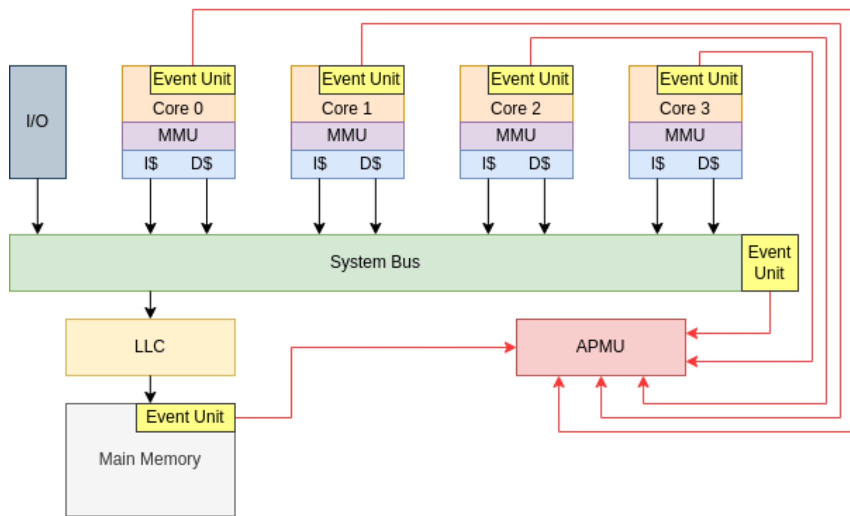


Figure 3.1: Typical multicore platform, with our proposed centralized performance monitoring infrastructure.

3.1 List of Example Applications

As mentioned, the APMU has a customized instruction processor that can process counter data and execute any desired software mechanism. A non-exhaustive list of techniques that can be implemented through the APMU is provided below.

1. **Resource Allocation Techniques**

   As discussed in Section 2.1.2, both Intel and Arm have proposed several variants of cache allocation techniques. These variants are targeted at high-performance systems that aim to meet certain QoS guarantees. The shared platform cache is partitioned between multiple tenants on the basis of either cache capacity or associative ways.

   These techniques can be implemented dynamically using the APMU. Depending on the number of accesses made, hits and misses suffered etc., the APMU core can write to the configuration registers in the cache and update its allocation policy.

Moreover, similar techniques can be applied to other hardware resources as well, such as the memory controller, system bus, I/O ports, etc. Application cores that have consumed less memory bandwidth, suffered longer request latencies, or those that have been waiting for I/O devices for long, etc., can be prioritized during request arbitration for those resources. We believe that the APMU is well-suited for this application because it can support any custom software allocation policy, combining and correlating information from multiple hardware IPs simultaneously.

Must be noted that for this purpose, we are dependent on the presence of configuration registers in the given resource that allows us to configure parts of its functionality. For example, the LLC that we are using in our evaluation in Chapter 5 supports way-based partitioning. By writing to the internal configuration of the LLC, we can map different ways to different application cores.

2. **Regulation Techniques**

   The APMU can be used to implement the different regulation techniques discussed in Section 2.2.2. Using EVUs distributed over the memory subsystem, the APMU can collect a variety of pertinent information such as the number of requests made, response latency suffered, etc. This information can then be used to implement various regulation mechanisms such as MemGuard, MemPol, etc.

   The works ([6], [7], [8], [9]) propose different variants of access-based regulation policies, i.e., each core is allowed to make a certain number of reads and writes to the memory subsystem, with their budget being replenished at regular intervals. If a core exceeds its allocated budget, the regulator can step in and either halt it entirely or reduce the priority of its requests until it acquires a new budget. A different approach was suggested by [10] that proposed a latency-based regulation policy that regulates cores when the latency distribution of the core-under-analysis is stochastically larger than a predefined target distribution. Latency-based regulation has less pessimistic bounds compared to access-based regulation. This is because the total latency suffered due to memory requests is a better indicator of the program's execution time than the number of memory accesses made. But complex hardware is required to measure individual request latency, which was not available in the COTS platform used by [10]. Their platform could only slot requests into discrete observation bins on the basis of their latency, forcing them to implement a discretized version of their proposed policy.

   The APMU can be used to implement the above policies efficiently. Because the system information is centralized in the APMU counters, the APMU processor suffers

low access penalty compared to traditional designs where the counters are localized within the hardware IP. By combining the information gathered from multiple modules, such as the system bus, caches, memory controllers, etc., the APMU can implement regulation policies comprising multiple metrics, unlike MemGuard and its variants ([6], [7], [8]) which can only focus on one or two metrics at once due to high access overhead. Using custom-EVUs, the APMU can also measure the latency of individual requests, which can be used to more accurately implement works such as [10]. Moreover, the APMU core also supports specialized hardware functions, discussed in Point 5 of Section 3.2.3, allowing it to poll counters more effectively, compared to existing software solutions such as those employed by MemPol ([9]).

3. **Timely Progress Assessment**

   The work [39] proposed a mechanism that uses timestamped milestone information to assess the progress made by a core. They define a milestone as an addresses that helps convey information about the processor's progress, thus acting as a progress *milestone*. In the work, a milestone-flow graph is developed through profiling and then provided to a monitoring core. Whenever the application core reaches a milestone address, as detected by specialized hardware in the core, the monitoring core is notified, allowing it to keep track of the core's progress. Following the notification, the monitoring core updates this specialized hardware component with the next set of potential milestone addresses from the graph. Our proposed APMU can be easily employed to execute such a mechanism, provided that there is a custom-EVU present in the application core. Moreover, using events information from other system components, the APMU can also help isolate the cause for delay in the application core during different phases of program execution.

4. **System Traces for Profiling and Debugging**

   As discussed in Section 2.1.4, many industrial designs and specifications have tracing capabilities, such as AMD Xilinx's AXI Performance Monitoring IP [5] that supports the logging of specific AXI events with timestamps, and Arm's CoreSight [29] which supports extensive tracing functionality. The system or processor traces generated by such designs are used extensively for profiling and debugging programs, conducting performance analysis, etc.

   The APMU can be used to generate detailed execution traces of the system. The key idea being that upon receiving events from multiple EVUs, the APMU processor can process them before generating the trace packets. For example, the APMU and custom-EVUs can be set up to track accesses to a subset of memory addresses

through the memory subsystem. EVUs in the caches, and memory controller can be configured to track the nature of these accesses: whether they result in a cache hit or miss, in an open- or close-row access in the memory controller, etc. Moreover, EVUs in the application cores can also help convey information about their execution phase using the milestone-based progress assessment technology discussed above, in Point 3. By combining all this information the APMU core can generate a latency decomposition for such accesses, categorized across the execution phases of the overall system.

5. **Security Mechanisms**

   Since the APMU has a bird-eye view of the entire platform, it can be easily adapted to implement a plethora of security mechanisms, such as control-flow integrity (CFI) checks, intrusion detection systems (IDS), policies to protect against denial-of-service (DoS) and side-channel attacks, etc.

   Several forward-edge control-flow integrity (CFI) mechanisms ([46], [47], [48], [49]) rely on control-flow checks to verify control flow switches of a processor. In the embedded domain, these checks can be made against the program's known control-flow graph (CFG) ([47]). By embedding an EVU in the application processor, the APMU can extract information about the direct and indirect jumps made by the program and compare it to the CFG stored in APMU's memory. In case of a CFI violation, the APMU can interrupt the offending core within a reasonable time window.

   Current literature has also proposed different variants of intrusion and anomaly detection systems (IDS, AIDS, respectively) that utilize hardware performance counters ([50] [15], [16], [17]). Using the system information gathered by these counters, different types of intrusion and anomaly detection models can be developed. The current literature categorizes AIDS system as follows:

   (a) **Knowledge-based AIDS**

   These models have a *priori* knowledge of the expected system behaviour and they reinforce this behaviour using description languages or finite-state machines (FSMs).

   (b) **Statistical AIDS**

   Statistical AIDS involves the collection and examination of every item in a set of events signifying expected usage, which is then used to build a statistical model of normal user behaviour. At run-time, the system behaviour is compared against this model to ensure correctness.

(c) **Machine Learning-based AIDS**

In this approach, large quantities of data is extracted from the system to train different machine learning (ML) models, such as clustering algorithms, neural networks, decision trees, genetic algorithms etc. Once trained, these models can then be used to ensure correct system behaviour at run-time.

All of the above works can benefit from a centralized performance monitoring framework. The APMU can collect a variety of system-related information in its counters and as system traces, which can then be used to either develop a statistical or knowledge-based AIDS, or train an ML model. Additionally, the developed model can then be implemented on the APMU processor directly.

Likewise, considering the susceptibility of shared resources to DoS attacks, several works have proposed solutions to detect and protect against them ([7], [51], [52]). By analyzing the resource access pattern of different applications, the APMU can detect suspicious behaviours. By combining and correlating information from multiple EVUs, the APMU can simultaneously monitor the entire platform and provide protection against multiple security vulnerabilities.

3.2 Proposed Design

This section describes our generalized architecture that consists of a) an Advanced Performance Monitoring Unit (APMU), and b) a set of distributed Event Units (EVUs). The EVUs can transmit hardware events and additional event information from the module they are installed in to the APMU. The APMU receives and operates on this information according to its configuration, which can be changed at run-time.

This work aims to limit the number of restrictions placed on EVU designers to ease their burden of developing component-specific EVUs. To that extent, we do not provide an EVU specification beyond a set of recommended guidelines. However, to guarantee integration with the performance monitoring infrastructure, an EVU-APMU interface specification is provided.

This section is laid out as follows: Section 3.2.1 describes the EVU-APMU interface. Section 3.2.2 provides a generalized description of EVUs along with a set of guidelines for development, and Section 3.2.3 furnishes the details of the APMU.

3.2.1 EVU-APMU Interface

The interface specification helps standardize the connection between any custom-EVU and the APMU. The output of an EVU has to be connected to a port on the APMU through a physical interface, as shown in Figure 3.2. Each APMU port must have a unique `Port ID`. An EVU can be connected to multiple APMU ports, if it so requires. The technical specification of the interface is provided in Appendix A.

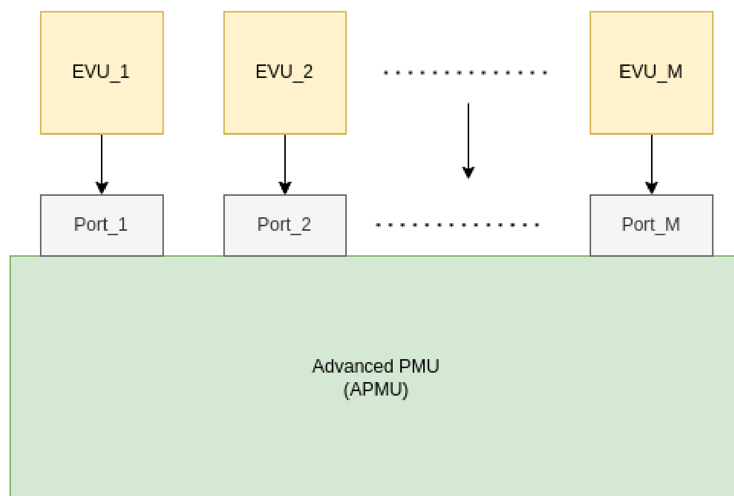


Figure 3.2: EVU-APMU Interface.

The interface is divided into two layers: a) the logical layer, which specifies the information that must be sent to the APMU per event, and b) the physical layer, which defines how the EVU is physically connected to the APMU. The two layers are described below:

1. **Logical Layer**

Source ID	Event Info	Event ID

Figure 3.3: The fields of an event packet.

For each observed event, the EVU transmits an *event packet* to the APMU. The event packet contains information that helps the APMU recognize the event and some additional metadata that provides a deeper insight into that event. An event packet, as shown in Figure 3.3, is divided into three fields: the `Event ID`, `Event Info` and `Source ID`. The fields are described below:

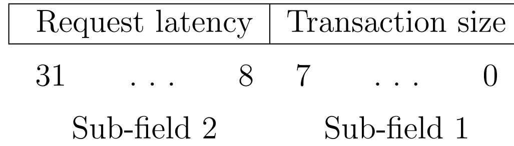


Figure 3.4: Event Info bits transmitting request latency and transaction size.

(a) **Event ID** (Mandatory)
Every type of event observed by an EVU is mapped to a unique `Event ID`, encoded as an unsigned integer. For example, an EVU in a processor can map different observable events such as memory instructions executed, branches taken, etc., to unique `Event ID`s.

(b) **Event Info** (Optional)
The specification allows EVUs to pack additional metadata in the `Event Info` alongside each event that they transmit to the APMU. This field helps convey additional information about the event. EVUs are allowed to group more than one type of metadata with an event. In such cases, the `Event Info` field must be split into multiple sub-fields. For example, in a system implementing the AXI4 protocol [19], it is possible to have an EVU connected to an AXI4-based port to monitor both request latency and transaction size of individual requests. To do so, the `Event Info` field could be divided into two sub-fields as shown in Figure 3.4, where 8 bits are reserved for transaction size and 24 bits for request latency.

(c) **Source ID** (Optional)
In many hardware IPs, the events are generated due to the action of entities external to said IP. For example, requests sent across a communication bus are generated by manager and subordinate modules connected externally to the bus. Or in caches, the memory accesses are made by external compute units. In such cases, transmitting information about the *source* entity that initiated said events is beneficial. Keeping this in mind, the interface specification supports the assignment of a unique `Source ID` to each potential event source for the component monitored by the EVU. This field is also encoded as an unsigned integer.

Each EVU-APMU interface is defined by an *event table* that contains all the necessary information to help users program the APMU with respect to this EVU. The event table is expected to have the `Event ID` and `Event ID` encodings, the supported

`Event Info` types, etc. This document is platform-dependent, so it must be drafted after the platform has been designed, including all the EVUs and the monitored component.

Lastly, the `Event ID` and `Source ID` encodings do not need to be consistent across EVUs of different types. This means that EVUs of different types can use the same `Event ID` and `Source ID` to represent different events and sources, respectively.

2. **Physical Layer**

   In this section, we list some of the possible physical connections that can be established between an EVU output port and an APMU input port. The simplest way of implementing this physical layer is to use a set of parallel wires, which is also what we implemented for our custom event unit, the AXI4-based Snooping Unit, discussed in Section 4.1. In this interface, the EVU sends one event packet to the APMU per clock cycle. The size of the interface is dictated by the size of the event packet, which can, if not limited, lead to expensive wiring cost for the design. However, using a parallel interface we can send more information per event packet, including `Event Info` and `Source ID` fields.

   An alternative solution is to implement a one-hot interface where the interface has a fixed number of bits confirmed at synthesis time, with each bit mapping to a specific hardware event. Using this interface, the EVU can transmit multiple event packets in one clock cycle. However, each event packet is restricted to just one bit, which can only confirm whether the event was observed in or not. Those event packets do not contain any `Event Info` or `Source ID` fields. It might be possible to transmit that information through this interface by adding extra wires, but it is only feasible when all the events in the interface share the same `Event Info` and `Source ID` values. Because in that case, these fields only need to be transmitted once per clock cycle, through some additional wires. If different events have different `Event Info` or `Source ID` values then they would each require extra bits to send them, but then the wire cost of the interface would quickly explode. Users are free to develop other types of EVU-APMU interfaces if they deem so necessary. However, then they must provide the specification for these interfaces.

   Lastly, the operating frequency of an EVU is likely to depend on that of the IP it is monitoring. Therefore, it is possible that this operating frequency is different from that of the APMU. This introduces additional design considerations because if an EVU and the APMU lie in different clock domains, i.e., they are asynchronous then they must be connected via clock domain crossing FIFOs (CDC FIFOs). If the EVU

clock is faster than that of the APMU, then it might need internal buffers to store excess event packets.

3.2.2 Event Units (EVU)

EVUs are hardware monitoring units distributed across various system components to monitor microarchitectural events. These components, such as processors, caches, memory controllers, interconnects, etc., are called *monitored* components. Each EVU output is connected to an APMU port as per the interface specification described above, in Section 3.2.1. Apart from the interface, we do not specify any other constraints for EVU development. However, an EVU should be designed keeping the following considerations in mind:

1. **Correctness**

   The addition of an EVU to a monitored component should not affect the correctness of said component.

2. **Minimally Invasive**

   The EVU must be minimally invasive, i.e., its addition to the monitored component should ideally not affect its performance. But in cases where unavoidable, the performance degradation should be limited.

3. **Interface Considerations**

   The interface for an EVU has to be decided considering a variety of factors such as, the number of potential events, the amount of information sent per event packet, the event generation rate, etc., which might yield certain interface types more suitable than others. For example, the simplest EVU-APMU physical interface—a parallel interface—has its bit width equal to the bit size of its event packet. The size of an event packet is the summation of all the bits in the `Event ID`, `Event Info`, and `Source ID` fields. Therefore, in an EVU with a parallel interface, it is important to keep the number of wires minimal. For other interfaces, the number of wires might be less of a concern, but sending a large transaction including the metadata of the event can still consume significant power.

4. **Optimization**

   The EVU should be optimized for area and efficiency. Since the APMU can operate on the collected event data, all unnecessary computations should be offloaded to the APMU.

5. **Configurability**

   EVUs can be designed to support different configurations. They can be configurable at synthesis time, run-time or even both. However, each of these configurability options have their own trade-offs. EVUs that are not configurable are easier to design. However, in that case, they are likely to transmit all event types to the APMU; forcing the APMU to filter them as it receives them at run-time. In this configuration, the APMU gets access to all the possible events that the EVU can monitor. But this design is not feasible for EVUs with large number of architectural events.

   EVUs that are designed for complex components such as processors, etc., tend to support hundreds or even thousands of different architectural events. But it is more likely that a user only cares about a subset of these events, depending on their application. Therefore, it does not make sense to transmit all these events to the APMU for filtering. Even more so, considering the expensive wiring cost.

   One way to get around this problem is to make the EVU configurable at synthesis time. This would allow users to specify the list of event they want the EVU to actively monitor. Moreover, doing so at synthesis would also reduce the area of the EVU, and the size of its interface(s) because architectural events that are not being monitored by the EVU would not need any circuitry. But this makes the EVU inflexible at run-time as users cannot change the list of actively monitored events.

   It is also possible to make the EVU configurable at run-time. In this case, the EVU can transmit a set of existing `Event ID`s. The users can then specify what architectural event maps to each `Event ID`, by writing to configuration registers in the EVU. The EVU would then only transmit information about these architectural events to the APMU using the `Event ID`s mapped to them. Also, since the list of supported `Event ID`s is much smaller than the number of supported architectural events, the resulting interface size would also be reduced. This also makes the EVU design flexible at run-time but with additional power and area costs.

An example of a functional EVU is presented in Section 4.1. We designed this EVU for the AMBA AXI4 protocol to monitor ongoing AXI transactions. It should be connected between an AXI4 manager and subordinate pair. A brief description of the AXI4 protocol is given in Section 4.1.1. The event table for the AXI-based Snooping Unit (AXI4 SPU) is given in Table 3.1. Information about the `Source ID` encoding is omitted for now. Each AXI event can also be associated with some additional event information. The EVU can be configured at synthesis to select the list of additional metadata that must be

transmitted alongside each event type. The request latency event information is essential in implementing latency-based regulation policies similar to the one discussed in Point 2 of Section 3.1.

Event	**Event ID**	**Event Info**
No event	0	NA
Read request	1	Size of transaction, unaligned transfer, etc.
Write request	2	
Read response	3	Request latency, etc.
Write response	4	

Table 3.1: AXI4 SPU Event Table

3.2.3 Advanced Performance Monitoring Unit (APMU)

The proposed APMU can receive, collect and operate on event packet received from EVUs distributed in the system. As shown in Figure 3.5, it comprises a configurable number of counters of `XLEN`-bit width—where `XLEN` is a parameter configurable at synthesis time—and their associated configuration registers and ALUs, a specialized processor and its instruction and data scratchpad memories (SPMs), and an interconnect to facilitate communication between the various intra-APMU components. The APMU is also connected to the system interrupt controller, allowing it to trigger interrupts in the application cores. The technical specification of the APMU is provided in Appendix B, but an explanation of each APMU components is given here:

1. **APMU Timer**

   The APMU has a 64-bit timer which increments every clock cycle. A timer is important for implementing mechanisms that require accurate timekeeping. It is also useful for generating timestamps for logging trace packets.

2. **Ports**

   The APMU can support a configurable number of ports. Each EVU in the system must connect to a separate APMU port, with a unique `Port ID`. The APMU port

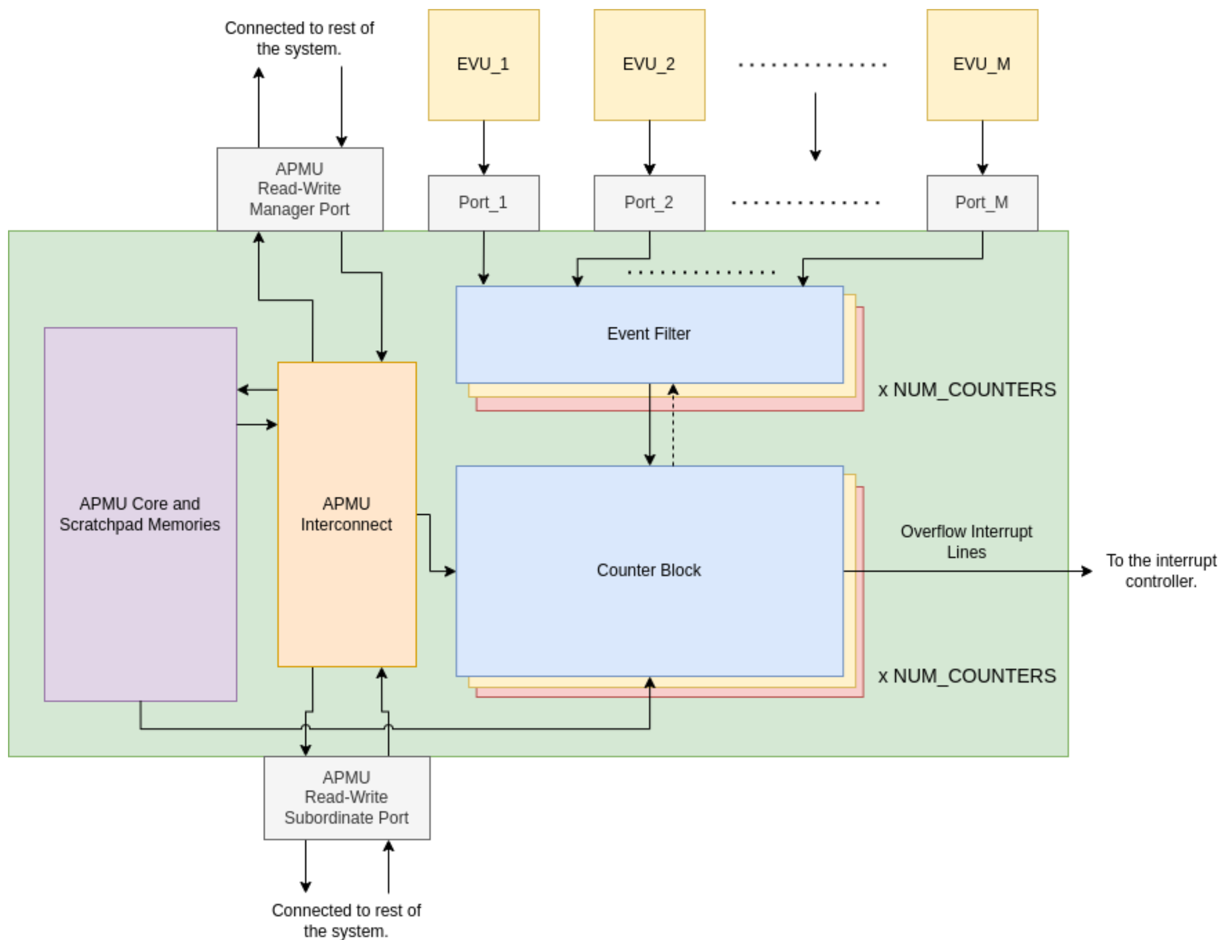


Figure 3.5: APMU Block Diagram with M EVUs.

receives event packets from an EVU and transmits it downstream to a subset of event filters, introduced in Point 3. This subset can range from one to all the event filters in the APMU.

Recall from Section 3.2.1 that the `Event ID` and `Source ID` encodings are not likely to be consistent across EVUs. This implies that the `Port ID` must be used to first specify the EVU whose events we want to count or operate on. Depending on the selected EVU, the `Event ID` and `Source ID` encodings of that EVU as provided in its event table are then used to filter in desired events. Lastly, the bit width of each APMU port depends on the EVU it is connected to and its choice of interface.

3. **Event Filters**

   Event filters process the event packets received through the APMU ports, as shown in Figure 3.6. The output of every APMU port is fanned out to either all or a subset of event filters. All counters have their own event filters. They process the event packets by comparing them against the configuration registers of their corresponding counters. These configuration registers, introduced in Point 4, are programmed to specify events that the event filter should accept, called *selected events*. The remaining events are discarded.

   Depending on the interface, the event filters can receive multiple event packets in the same clock cycle, be it from multiple APMU ports or a single one. Moreover, the configuration registers can also specify more than one selected events for its counter. Therefore, the event filter should be capable of handling multiple selected events in the same clock cycle.

   One way of implementing an event filter module is given below. In this design, the event filter sends three output signals to its respective counter block: a) an increment value, b) a selected event info and c) an event valid. The increment value represents the total number of selected events the event filter received in that clock cycle. For example, consider a counter programmed to count events with ID 5 and 7 from EVUs connected to ports 1 and 3, respectively, of the APMU. At time t, the APMU receives event packets with `Event ID`s 5 and 7 on ports with `Port ID`s 1 and 3, respectively. In this case, the counter should increment by 2, one for each selected event.

   Meanwhile, the selected event info signal represents the `Event Info` field of the selected event that was received in the current clock cycle. This allows the counter block to operate on the additional event information that is packed within an event packet. In case of multiple selected events, the event filter can arbitrarily pick the event metadata belonging to any of the selected event packets. This is why when the counter is configured to operate on the `Event Info` field, the event filters should be programmed to focus on a single port. Otherwise the counters risk missing out on event info from other selected events received in the same clock cycle.

   The event valid signal is only set to high when the filter receives at least one selected event in a clock cycle. Otherwise it is set to low.

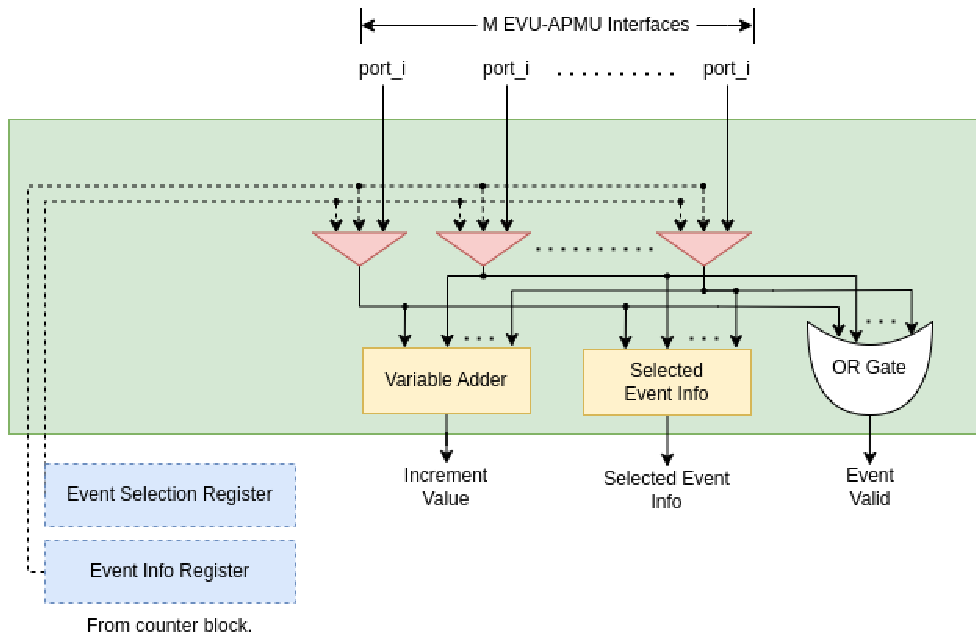


Figure 3.6: Event Filter of an APMU Counter Block.

4. **Counter Blocks**

Each APMU counter has its own counter block. A counter block constitutes an `XLEN`-bit counter, where `XLEN` is a parameter specified during synthesis, its configuration registers and an ALU. An example design of the counter block is shown in Figure 3.7. Each APMU counter is associated with two configuration registers: the Event Selection register (`EventSelCfg`), and the Event Info register (`EventInfoCfg`). This section only discusses the features of the configuration registers. A bit-by-bit description of the configuration registers is provided in the specification in Appendix B.

The structure of an APMU counter is given in Figure 3.8. Each counter has `XLEN` bits. The two most significant bits of a counter are called its pending and overflow bits. The remaining `XLEN-2` bits are used for standard counter operations.

(a) **Pending bit**

The most significant bit of the counter is called its pending bit. This bit is set whenever the counter is updated during regular operation. When the counter is programmed to only count events without operating on the event info bits, it is

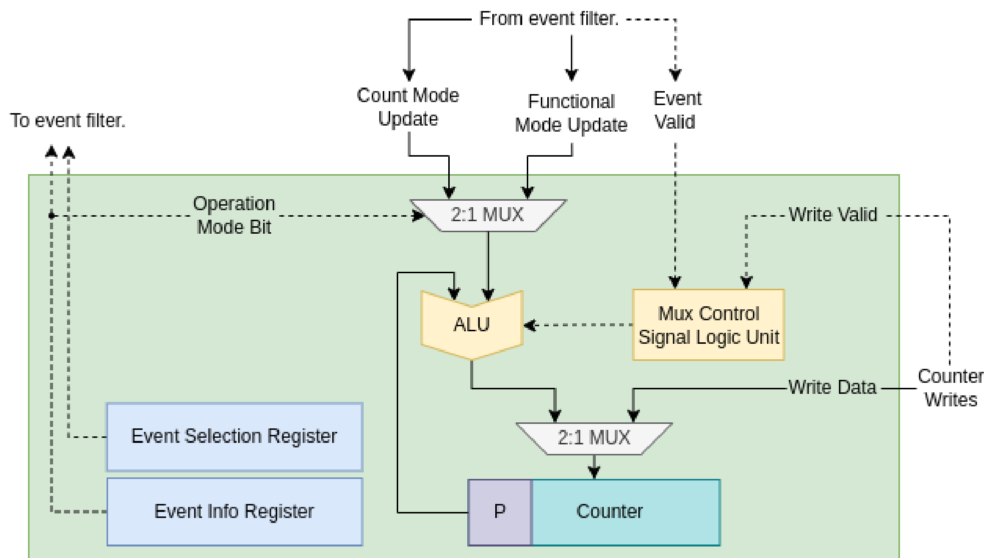


Figure 3.7: Counter Block.

said to be in *count* mode. In this mode, the counter is updated by the number of selected events received in that clock cycle. This information is conveyed by the input signals sent from the event filter. On the other hand, if the counter block is programmed to operate on the event info bits and then update accordingly, it is said to be in *functional* mode. In this mode, the counter is updated according to the ALU operation specified in the `EventInfoCfg` register. The pending bit is always set when the counter is updated during either operation modes.

The pending bit is reset by the hardware when the core exits its Wait-for-Pending mode, described in Point 5. Apart from that, it can be also reset by writing 0 to it through software.

(b) **Overflow bit**

The second most significant bit is called its overflow bit. When this bit is set, the counter is said to have overflow. The counter can send an interrupt signal to the platform interrupt controller upon overflow. Additionally, by setting the overflow bit of a counter, the APMU core can also trigger an interrupt. This functionality is crucial to several case studies discussed in Section 3.1; for instance, the regulation policies such as MemGuard and its variants ([6], [7], [8]). In these mechanisms, each application core is assigned a specific budget for some metric, such as the number reads or writes to memory, etc., with the budget being reset at regular intervals. If the core exceeds it allotted budget

then the mechanism forces it to halt. If the counter is reset to its overflow value ($2^{\texttt{XLEN-1}}$) minus the regulation budget, it will automatically overflow when the budget is exhausted. Thus triggering an interrupt.

If the overflow bit is set while the counter was updated in either count mode or functional mode, then it must be reset via software. But it is also reset by the hardware when the core exits the Wait-for-Overflow mode, as described in Point 5.

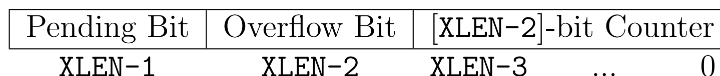


Figure 3.8: Format of a `XLEN`-bit APMU Counter.

The operation mode of a counter is decided by the `EventInfoCfg` register. Additionally, the two configuration registers are also responsible for controlling all other aspects of the counter block.

The `EventSelCfg` register is used to specify events that its respective counter should either count or operate upon. These selected events are processed by the event filter privy to said counter.

The `EventInfoCfg` register is used to specify operations that the ALU in that counter block should perform on the event metadata of a selected event packet. As discussed in Figure 3.4, the `Event Info` field can have various event-related metadata merged together. Therefore, it is necessary to allow ALU the functionality to operate on only a subset of these bits. The `EventInfoCfg` register can be used to specify a contiguous set of bits from the `Event Info` field for the ALU to operate on. This specified set of event info bits from a selected event is referred to as the *sliced event info* bits. The `EventInfoCfg` register is also used to enable or disable the interrupt functionality for counter overflow.

Each counter has a dedicated ALU that can perform various arithmetic and logical operations on the selected event info bits input from the filter. Firstly, the ALU isolates the sliced event info bits from the `Event Info` field. Then it performs the selected operation. An ALU operation is specified by writing its opcode to the `EventInfoCfg` register. The supported ALU operations and their opcodes are given in Table 3.2. A brief description of the various operations is given below:

(a) The `Addition` operation adds the sliced event info bits to the counter data. For example, using this ALU operation, a counter can cumulatively count the total

latency an application core suffers on each memory request, which is crucial for implementing the latency-based regulation policies discussed in Point 2 of Section 3.1. For example, consider a latency-based regulation policy that attempts to ensure that the core-under-analysis does not suffer an average request latency greater than some set target, while suffering memory contention from other cores. In this case, the APMU can set up a counter to count the number of requests made by said core, and another to count the total latency suffered. This can be done by installing EVUs in the memory subsystem. For example, the Snooping Unit we present in Section 4.1. Meanwhile, the APMU core can be programmed to calculate the average request latency using the data stored in the two aforementioned counters, and take necessary actions when the average exceeds a set target.

(b) The `KeepMin` and `KeepMax` operations compare the sliced event info signal to the counter value, and keep the minimum or maximum of two, respectively. For example, this ALU operation can be used to study the best- and worst-case request latencies suffered by an application under different phases of contention.

(c) The `Increment: Eq` operation compares the sliced event info bits to another value, written to a sub-field, of the `EventInfoCfg` register. If the two are equal, the counter increments by 1.

The ALU supports also other operations like `Increment: NotEq`, etc., which are similar except for the chosen relational operator(s). For example, these operations can be used to monitor when an application core accesses a particular memory region, assuming that an EVU is present in the core. An application core, if allowed by the OS or hypervisor, can access any memory-mapped platform component in the system. But in cases where it is hard to vet the program being executed on the core, there is a risk of security attacks, such as denial-of-service (DoS) on caches, memory, I/O devices, etc. The APMU can be used to protect vulnerable IPs from such attacks. The EVU can transmit information signifying the component accessed by a core as part of the `Event Info` of an event packet. This can be done trivially by comparing the address of the load or store instruction to the platform memory map. If the core made can access to a specific memory location lying within a particular address range the counter will increment. This will consequently set its pending bit. Through the Wait-for-Pending functionality, the APMU core will be informed and it can then run a software check to verify the validity of the access.

(d) Operations like `Add: NotEq`, `Add: NotEq`, `Add: LessThan`, etc., are similar to `Increment: X`, except that in this case, if the relational operation evaluates to

true, then the sliced event info bits are added to the counter.

Operation	Opcode
`Addition`	00000
`KeepMax`	00001
`KeepMin`	00010
`Increment: Eq`	00011
`Increment: NotEq`	00100
`Increment: LessThan`	00101
`Increment: GreaterThan`	00110
`Increment: LessThanEqual`	00111
`Increment: GreaterThanEqual`	01000
`Increment: InRange`	01001
`Increment: NotInRange`	01010
`Add: Eq`	01011
`Add: NotEq`	01100
`Add: LessThan`	01101
`Add: GreaterThan`	01110
`Add: LessThanEqual`	01111
`Add: GreaterThanEqual`	10000
`Add: InRange`	10001
`Add: NotInRange`	10010

Table 3.2: ALU Operations and their opcodes

5. **APMU Core and Memory Infrastructure**

   The APMU has a processing core that can be programmed at runtime to execute complex software mechanisms such as those presented in Section 3.1. Most of these mechanisms are event-based, i.e., they only make supervisory decisions when specific events occur. For that reason, the specification proposes additional functionalities in the core that build upon the existing APMU architecture, to improve its program execution time further when implementing these mechanisms.

The choice of a suitable processing core considering the trade-off in area, power, and efficiency is left to the discretion of the system designer and their platform's use case. A multi-issue out-of-order processor with more efficient functional units, such as a single-cycle multiplier, floating point ALU, etc., will generally have higher instructions per cycle (IPC) than a single-issue in-order processor for the same program, but at the cost of more power and area consumption. However, in certain use cases such as those of real-time systems, the latter might be a more suitable choice because its worst-case execution time (WCET) can be calculated more predictably than the former. Nevertheless, the specification proposes the following functionalities for any chosen APMU processor:

(a) **Support to read and write to the APMU counters and registers**

Since the APMU core is expected to process the counter data, it is reasonable to add support to the core so that it can access them effectively. The core can read the counters and re-evaluate its policies according to the program loaded into it. This functionality is essential in implementing any of the event-based software mechanisms presented in Section 3.1.

(b) **Support to read and write to rest of the system**

The APMU core should be able to read and write to other sections of the system memory and I/O devices via the shared platform bus. This feature is required to implement dynamic resource allocation and arbitration techniques, similar to those discussed Point 1 of Section 3.1. For instance, consider a mechanism that based on various metrics, such as the number of cache accesses, hit-miss ratio, access latency, etc., updates the cache partition sizes for the application cores. Shared caches generally have configuration registers that control their partition allocation. Therefore, to implement such a mechanism, the APMU core must be able to write to the configuration registers of these caches. Using this feature, the APMU can also transmit tracing information to off-chip memory at runtime.

(c) **Wait-for-X Functionality**

Most of the software mechanisms discussed in Section 3.1 either wait for the occurrence of certain events, or wait until certain events have occurred a specific number of times before executing their policies. For example, in budget-based regulation techniques, an application core is only halted when it has spent its allocated budget. In [39], hardware in the application core would inform a smaller monitoring core when it had reached a new progress milestone, along with information about the milestone reached. Taking this into account, the specification proposes two additional functionalities:

i. **Wait-for-Pending**

Recall from Point 4 of Section 3.2.3, that the most significant bit of every APMU counter is called its pending bit, which is set whenever the counter is updated during its count or functional mode. The Wait-for-Pending functionality allows the APMU core to enter a no-operation or low-power state, while polling on a set of counters. The core only exits this Wait-for-Pending mode when any one of the specified counter has its pending bit set.

For example, this feature can be used to implement CFI mechanisms, where a counter is programmed to increment only when an application core executes a branch or jump instruction. The APMU core can enter a Wait-for-Pending mode on this counter, only waking up when it increments. After which the core can verify the branch or jump instruction using any of the CFI mechanisms described in Point 5 of Section 3.1.

ii. **Wait-for-Overflow**

As mentioned in Point 4 of Section 3.2.3, the second most significant bit of the counter is called its overflow bit. This bit is set when the `XLEN-2` bit wide counter overflows. The Wait-for-Overflow functionality, similar to the Wait-for-Pending functionality, puts the counter in a no-operation or low-power state, only waking it up when any one of the specified counters has overflown.

For example, this feature can be used to implement budget-based regulation policies. The counter value can be initialized to the overflow value, ($2^{\texttt{XLEN-1}}$), minus its budget. Therefore, when the counter reaches its budget, it overflows, triggering the APMU core.

We envision that such functionalities can be implemented by extending the ISA with specific instructions.

(d) **Support to trigger interrupts**

The ability to halt and resume application cores is crucial in implementing most of the techniques discussed in Section 3.1. As such, the APMU core should have the functionality to send interrupt signals to the platform interrupt controller. This can be done by setting the overflow bit of the counters to 1.

The core should be complemented with instruction and data SPMs. The SPMs should be read- and write-able through the manager and subordinate ports of the APMU that connect it to the platform. Using these ports, any external manager, such as, an application core can access the SPMs. By updating the instruction SPM (ISPM)

during runtime, the APMU core can be repurposed to implement other programs dynamically. The data SPM (DSPM) eases the execution of programs that require a software stack, and can also be used to store system information for profiling, analysis, and debugging purposes.

To safely re-program the core, it needs a mechanism that allows external managers, such as the application cores, to halt and restart it. In that respect, the core has the Status (`Status`) and Boot Address registers (`BootAddr`), as shown in Figure 3.9 and Figure 3.10 respectively. The bit width of the `BootAddr` is defined as `YLEN` and it depends on the address width of the memory interface of the APMU core. The 0th bit of the `Status` register is the `Stall` bit, and the remaining bits are currently unused. The core is stalled if the `Stall` bit is set to 1. The reset value of this register is 1. This implies that at system reset, the core is already stalled. When 0 is written to this bit, the core restarts from the ISPM address specified in `BootAddr` register. The `BootAddr` register is initialized with the value of the ISPM base address. When the core is started, it fetches instructions from the boot address and executes them. The core can be reset again by writing a `1` to the `Stall` bit.

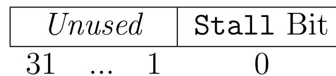


Figure 3.9: `Status` register of the APMU core.

Figure 3.10: `BootAddr` register of the APMU core.

3.3 Software Design

The infrastructure of the APMU is designed such that it can be integrated into a virtualized environment. In virtualized platforms, the hypervisor has exclusive control over security-critical components. Since the APMU is capable of halting any application core in the system, it is essential to restrict unauthorized access to it. Moreover, due to its criticality, it is important to ensure that the APMU is booted in a safe and controlled manner. Keeping this in mind, we envision a boot up process in which the hypervisor owns an image for

the APMU, similar to it having an image for each VM. It then loads the VMs on the application cores and the APMU image on the APMU. The hypervisor needs to initialize the APMU. This includes performing functions such as programming the configuration registers, setting up interrupts, etc. Once the hypervisor has completed all APMU-related operations, it can start the deviceS.

In some cases, the system designer might want to allow the operating systems, running on top of the hypervisor, access to certain portions of the APMU counters. For example, a virtualized OS that needs to run a resource regulation policy on its own user-level program might need to use some APMU counters. This can be done by letting the hypervisor emulate access to the counters on behalf of the virtualized OS, but this method would result in large software overheads, increasing the polling cost significantly. In the case of a hypervisor that supports two-level translation, a more efficient solution would be to map the memory addresses of some APMU counters to the virtual memory of the OS. This would allow the OS to access some of the APMU counters directly.

For that reason, the counters are spaced out in physical memory so that the hypervisor can put different counters in different physical pages and map them to the virtual pages of different OSes. Thus placing the counters under the control of their respective OSes. But critical registers such as those belonging to the APMU core and the configuration registers of the counters are not spaced out as such, and should not be mapped to the virtual pages of OSes directly. If an OS needs to access any of these registers, it should do so via hypervisor calls that need to be defined in the software framework of the hypervisor. An example memory map of the APMU is given in Table 3.3. In the given memory map, all registers, except the timer are be 32-bit wide. The timer is 64-bit wide.

Similarly, the OS might need to run a program on the APMU core. In this case, the hypervisor can also map part of the core's ISPM and DSPM to the OS's virtual memory. By letting OSes access portions of the ISPM and DSPM, the system can run mechanisms that can take input from these OSes at regular intervals. For example, a multi-level regulation policy where the hypervisor assigns a budget to all its virtual machines, and where the VMs are allowed to distribute their budget internally between their user-level programs. The hypervisor can map small portions of the ISPM to all the VMs so that they can write the budget allocation of their user-level programs in their allocated ISPM portion. The program running on the specialized core can read these allocations and regulate the various programs accordingly. Moreover, the data produced by the APMU processor can also be accessed by the OSes through the DSPM. An important use case would be when an OS wants to monitor the system trace of one of its user-level programs. The core can write the trace information into the DSPM, and the hypervisor can then allot the updated DSPM regions to the said OS.

Counter Block	Register	Address
NA	APMU Timer APMU Core Status Register APMU Core Boot Address	base base + 0x8 base + 0xc
0	Event Selection Register Event Info Register	base + 0x10 base + 0x14
1	Event Selection Register Event Info Register	base + 0x18 base + 0x1c
...	...	...
31	Event Selection Register Event Info Register	base + 0x108 base + 0x10c
0	Counter	base + page_size
1	Counter	base + 2 × page_size
...	...	...
31	Counter	base + 32 × page_size

Table 3.3: Example of an APMU Memory Map.

Since the APMU counters have several configuration settings and the processor has custom functionalities, we recommend providing software support for programming the APMU to help users write programs for the processor. This can be done by supplying a compiler specific to the APMU to help compile code, along with a set of library functions and macros to support common functionality.

To ensure that the specialized APMU core is not executing malicious programs, the hypervisor can restrict access to the core's ISPM, run source code analysis on any program code that is to be run on the APMU core, impose strict programming rules, perform security checks, etc. This would ensure that no rogue OS can take control of the APMU processor, ensuring a safe execution environment. However, the specification does not specify this.

Lastly, since the APMU is a platform device, it needs to be protected from malicious attackers. By design, any platform manager can write to the APMU counters and the SPMs. This means that if an OS controls a DMA peripheral, it could maliciously configure it to write/read to the APMU. To avoid this, we need an I/O Memory Management Unit (MMU). For example, both Arm and RISC-V have the System MMU [53] and I/O MMU ([54], [55]) architecture specifications, respectively; designed to protect the platform

memory from malicious DMA traffic.

Chapter 4

Implementation

This chapter presents the hardware implementation of the design specification proposed in Chapter 3 as a proof of concept, along with basic software support. This implementation is designed considering a RISC-V architecture that employs AXI4-based system interconnects. The hardware design consists of an AXI4 Snooping Unit (AXI4 SPU) and the Advanced Performance Monitoring Unit (APMU). The AXI4 SPU is developed to monitor AXI packets sent between AXI manager and subordinate pairs. It generates event packets for the APMU in accordance with the EVU-APMU interface specification presented in Section 3.2.1. We envision the AXI4 SPUs as being embedded between various inter-platform components, such as the processors and the shared LLC, the LLC and main memory, etc. This would allow the APMU access to information about the LLC which can then be exploited to run some of the case studies discussed in Section 3.1.

This chapter has three sections. In Section 4.1, the AXI4 SPU is discussed. The RISC-V-compliant APMU and its supporting software stack are presented in Sections 4.2 and 4.3, respectively.

4.1 AXI4 Snooping Unit

The AXI4 Snooping Unit (from here on, referred to as the SPU) is designed to snoop AXI packets sent between a pair of monitored manager-subordinate components. The SPU has a manager and a subordinate port. It is connected between the monitored components, as shown in Figure 4.1. The manager IP is connected to the subordinate port of the SPU, and the manager port of the SPU is connected to the subordinate IP. The AXI

request and response packets sent through the SPU are processed via dedicated filters, which output different AXI-related event packets. These event packets are then forwarded to the APMU using a parallel interface. The event table of the SPU is presented in Section 4.1.2, and the hardware design is discussed in Section 4.1.3. Before that, in Section 4.1.1 we introduce to the AXI4 protocol to familiarize the readers with the fundamentals of the AXI4 specification.

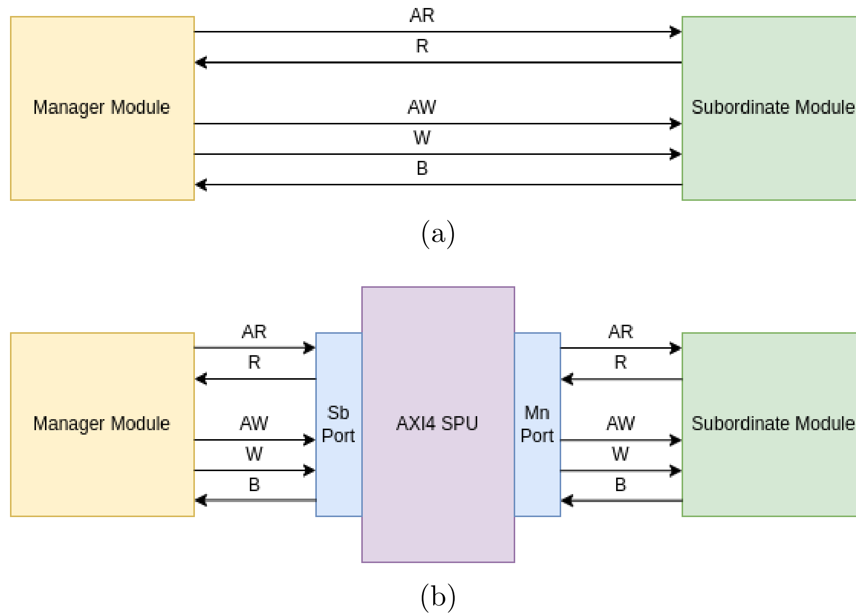


Figure 4.1: Connection of an SPU between Manager and Subordinate modules. Figure (a) shows the original AXI bus between the two modules. Figure (b) shows the bus getting routed through a SPU.

4.1.1 AXI4 Fundamentals

The AMBA AXI4 protocol [19] specified by Arm, is an on-chip point-to-point burst-based communication protocol. AMBA4 is the 2010 revised version of the AMBA specification. The revised protocol specifies three AXI4-based variants: full AXI4, AXI4-Lite and AXI4-Stream. This section presents the relevant information for the full AXI4 protocol (referred to as AXI4, from now on) which supports burst transactions, allowing multiple data packets, also called beats, to be transferred in response to a single request.

1. Under the specification guidelines, communication over an AXI4 bus can only occur between a single initiator, called a *manager*, and a single target, called a *subordinate*, as shown in Figure 4.1a. However, the specification provides detailed descriptions and signals, allowing users to extend the protocol to develop their custom multi-manager multi-subordinate interconnects.

2. **Communication channels**

   The AXI4 specification has five communication channels, namely, write address (AW), write data (W), write response (B), read address (AR) and read data (R). Each channel is composed of multiple signals, such as, `xVALID`, `xREADY`, `AxADDR`, `xDATA`, etc.

   To ensure correct communication over a channel, AXI4 defines a handshake mechanism. This mechanism uses a set of signals: `xVALID` and `xREADY` to transfer the data payload over the channel. A payload consists of all the other source-related signals in the channel except for `xVALID`. The source sets `xVALID` to indicate that the payload on the channel is valid and can be read from that cycle onwards. The destination module sets the `xREADY` signal to indicate that it is ready to read the payload. When both signals are high in the same cycle, the payload is considered transferred. After which, the source can send a new payload over the channel from the next clock cycle onwards. An individual data transfer, that is when both the `xVALID` and `xREADY` signals are high for one or more clock cycles to transfer one payload, is defined as a beat.

   As shown in Figure 4.1a, the manager IP is the source module for the AR, AW and W channels, while it is the destination module for the R and B channel. The converse is true for the subordinate IP. Since the manager initiates a data transfer, it is responsible for providing the details of the transfer, such as the memory address to be accessed, the amount of data to be transferred, the type of request: read or write, etc. A read comprises the manager sending a read request on the AR channel and the subordinate's response on the R channel. In order to do so, the manager provides all the necessary information such as the read address, the size of each beat, the number of beats, the burst type, etc., using the respective signals of the AR channel. Upon receiving a read request, the subordinate responds with the read data in the form of a series of beats, following the manager's constraints, over multiple clock cycles. The subordinate sets the `RLAST` signal on the R channel to signify the last beat of the transaction.

   On the other hand, in case of a write, the manager sends a write request on the AW channel and the write data on the W channel. This involves sending the necessary

write information on the AW channel, followed by the data to be written on the W channel. Upon completion of the write, the subordinate responds on the B channel.

3. **Type of bursts**

   The specification supports burst-based transfer. In AXI4, a data transaction is called a burst, which is made of multiple beats of valid data. The address accessed by the first beat of the burst is sent through the `AxADDR` signal. However, the addresses of the remaining beats are computed by the subordinate module based on the burst type, indicated by the `AxBURST` signal. AXI4 supports three burst types: incremental (`INCR`), fixed (`FIXED`) and wrap (`WRAP`). In `INCR` burst, the following address is computed by adding the beat size, calculated as $2^{\texttt{ARSIZE}}$, to the address of the current beat. In `FIXED` burst, the address for the following beats does not change. This mode is used to access fixed I/O ports continuously. For instance, the TX or RX registers used by the UART module. The third mode, `WRAP`, is similar to the `INCR` mode except that the next address will wrap around to the starting address if it crosses the burst boundary. This mode is used in critical word first cache refills.

4. **Transaction IDs**

   In AXI4, each transaction is associated with a transaction ID. However, the AXI4 protocol allows managers to reuse transaction IDs for new transactions, even if there are ongoing transactions with the same ID. This means that there is no unique key that can be used to tag a transaction. But the AXI4 specification does impose a few restrictions for AXI4-based subordinate modules. The one most relevant for understanding the SPU is the following: read or write requests with the same transaction ID targeting the same subordinate must be resolved in order if they share the same communication channel.

4.1.2 Event Table for the SPU

This section provides a logical description of the SPU events and their associated metadata. The hardware design developed to capture these events is presented in Section 4.1.3. As discussed in Section 3.2.1, an EVU event packet has the following three fields: `Event ID`, `Event Info`, and `Source ID`. The event table detailing the various events and their associated metadata is given in Table 3.1. The `Source ID` encoding is omitted in the event table but it is discussed in Point 1 of Section 4.1.3. The SPU is configurable and allows users to choose what additional event information they want to transfer with each type of event.

The read and write request events indicate that the manager has initiated a new read or write transaction. Meanwhile, the read and write response events indicate that the subordinate has completed an ongoing read or write transaction. A detailed explanation of the various events is given below:

1. **No event**

   The `Event ID` of 0 is reserved for when the SPU does not observe any valid event on the AXI bus.

2. **Read request**

   The read request event indicates that the manager has initiated a new read transaction. Under the AXI4 protocol, a manager initiates a new read transaction by sending a valid read address to a subordinate. When the SPU detects a new valid read address on the AR channel, it generates a read request event packet. Along with the read event packet, we can also transmit additional information regarding the read request, such as the index of the addresses it is accessing, the size of the transaction requested, and whether the address is aligned or not. The EVU can also be configured to transmit the ID of the manager that initiated the transaction via the `Source ID` field.

   The SPU maps read addresses to unique indices based on an address mapping that must be provided at synthesis time. A typical example of such a mapping is given in Table 4.1. This allows the APMU to focus on AXI transactions that only affect a particular module, such as the main memory, etc. This feature is essential in implementing regulation policies, or in monitoring and restricting accesses to specific memory regions or I/O devices for security purposes.

   The SPU can compute the size of the requested transaction in either a) the number of bytes or b) the number of cachelines (or rows in main memory) touched. Along with this information, the SPU can also determine whether the transaction is aligned. A transaction is considered aligned if it occurs at cacheline (or row) boundaries. This feature is primarily aimed at cache and main memory transactions.

3. **Write request**

   Similar to Point 2, a write request event indicates that a new write transaction has been initiated by the manager. The event packet can also contain the `Source ID` of the initiating manager and additional information about the address range the write is directed to, the size of the write transaction and whether it is aligned or not.

Index	Address Range	Mapped to
0	`0x0000 - 0x0FFF`	Debug ROM
1	`0x1000 - 0x7FFF`	Main Memory
2	`0x8000 - 0x81FF`	UART
3	`0x8200 - 0x8FFF`	Reserved

Table 4.1: A typical address mapping for a computing platform.

4. **Read response**

   The read response event indicates that the subordinate has completed an ongoing read transaction. In AXI4, a read transaction completes when the subordinate sends the last read data beat, as indicated by the last signal (`RLAST`) on the read data (R) channel. A read response event packet can also contain additional information, such as the index of the addresses accessed and the response latency. It can also contain the `Source ID` of the manager that initiated the read.

   The address mapping and the associated indices are the same as the one explained in Point 2. However, the read response latency is measured from the clock cycle the transaction is initiated, i.e., a valid read address is sent to the subordinate to the clock cycle in which the subordinate sends back the last data beat for the same transaction.

5. **Write response**

   Similar to Point 4, a write response event indicates that an ongoing write transaction has been completed. In AXI4, a write transaction completes when the subordinate sends back a write response signal (`bRESP`) on the write response (B) channel. A write response event packet can contain the same set of additional event information, namely, the index of the addresses accessed and the response latency.

   In this case, the response latency is measured from the clock cycle the transaction is initiated to the clock cycle in which a valid write response is observed for the same transaction. Lastly, the event packet can also transmit the `Source ID` of the manager that initiated the write.

4.1.3 Hardware Design of the SPU

The SPU is connected between a manager and subordinate pair through AXI4 ports, as shown in Figure 4.1. In the current implementation, the subordinate and manager

ports of the SPU are connected directly without any pipeline registers. This ensures that adding an SPU does not increase the delay suffered by the requests. However, on platforms where adding an SPU between two modules breaks timing closure, pipelining might become necessary.

The SPU is connected to the APMU through a parallel interface. As shown in Figure 4.2, the SPU registers request and response packets traversing between the AXI manager and subordinate before processing them, a clock cycle after. The AXI packets are processed via dedicated filters that produce AXI-related event packets. Each SPU event from Table 3.1 has a separate pipeline. The generated event packets are stored in their respective FIFOs. The output port of the SPU that is connected to the APMU is selected from these FIFOs in a round-robin (RR) fashion via an RR arbiter. If there is no valid event packet, the SPU transmits a no-event packet. The entire SPU design is presented in Figure 4.3.

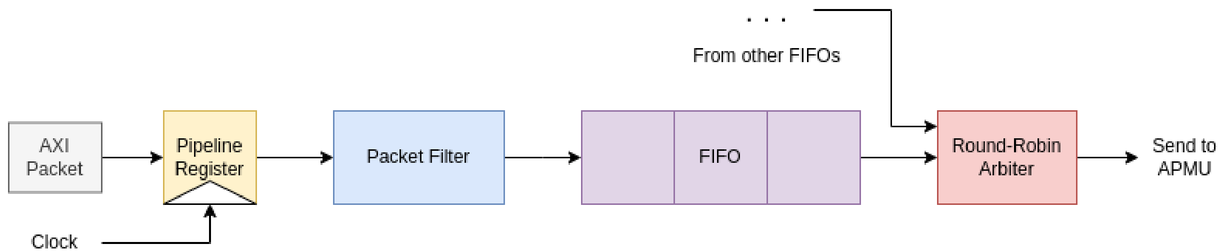


Figure 4.2: SPU Pipeline.

The SPU packet filters are discussed in detail below:

1. **AR Channel Filter**

   The AR channel filter generates the read request event packets explained in Point 2 of Section 4.1.2. It takes in the registered AXI packet sent on the AR channel. When the `ARVALID` signal on this channel is set, it indicates that the manager has initiated a new read transaction and the filter generates a new event packet containing details of the initiated read, such as the index of the accessed address, the size of the transaction, and the alignment of the transaction.

   The filter compares the address accessed on the `ARADDR` signal against the address mapping programmed in the SPU at synthesis time to decode its index.

   The SPU can be programmed to compute transaction addresses in either a) bytes or b) the number of cachelines (or rows) accessed. In the first case, the size of an AXI

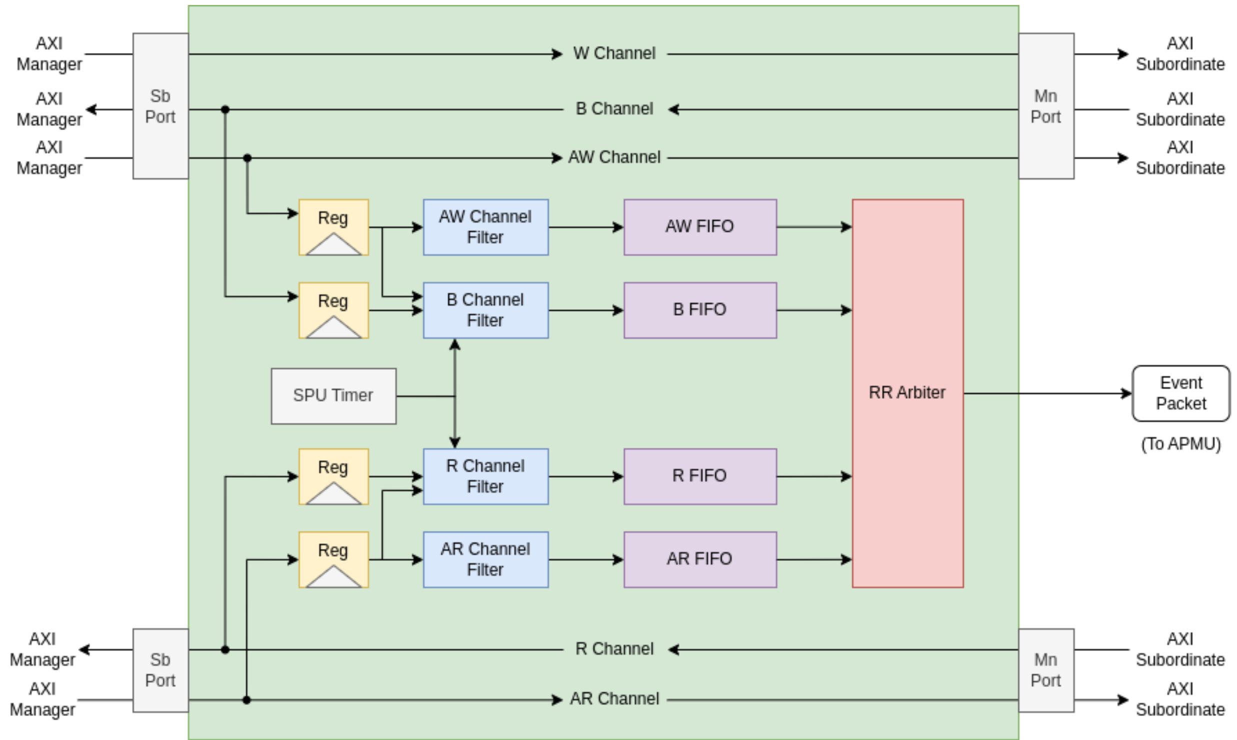


Figure 4.3: SPU Block Diagram.

transaction in bytes can be calculated using the `ARLEN` and `ARSIZE` signals.

$$\text{Transaction Size} = (\texttt{ARLEN+1}) \times 2^{\texttt{ARSIZE}} \tag{4.1}$$

Where `ARLEN` is the number of beats in the AXI burst, and `ARSIZE` is the $\log_2$ of the size of each beat.

The SPU can also compute the transaction size in the number of cachelines (or rows) affected. However, the current filter design only supports this for `INCR` burst type. Burst types are described in Point 3 of Section 4.1.1. To calculate the transaction size in the number of cachelines, we need to check if the transaction is going to cross over to multiple cachelines. Since each AXI4 transaction has a defined transaction size in bytes, we need to confirm whether the available bytes in the current cacheline is greater than or equal to the transaction size of the request. If it is greater, then the request will only access this one cacheline. Otherwise, we need to compute the remaining subsequent cachelines that will be accessed. This is because in `INCR` burst type, the addresses of the upcoming beats are incremental. The pseudo-code for this

operation is given as Algorithm 1. It works as follows:

(a) Let `r_addr` be the starting address of the read request, and `t_size` be the transaction size of the request in bytes, computed using Equation 4.1. Let `LineOffset` be the number of offset bits in the cache addressing scheme. Lastly, let the address width of the bus be 64-bits.

(b) We first need to calculate the address of the current cacheline. Since AXI4 supports unaligned transactions, a read request can target addresses from the middle of a cacheline. Therefore, to get the address of the current cacheline we discard the offset bits, as shown in Line 2.

(c) Compute the address of the next cacheline. This can be done by incrementing the index of the current cacheline by 1, as shown in Line 5.

(d) Calculate the number of *available* bytes in the current cacheline. These are the bytes that can potentially be read during a read transaction.

For example, consider a cache with 8 B cachelines. If the current cacheline starts at address `0x0`, while the read request is targeting address `0x5` and onwards, then the bytes from `0x0` to `0x4` are unavailable bytes, as they can never be read in this transaction. But bytes `0x5`, `0x6`, `0x7` in this cacheline and those in the subsequent cachelines can potentially be read, depending on the transaction size of the request. The number of available bytes in the current cacheline is calculated as shown in Line 8.

(e) In lines: 11-14, we compare if the transaction size of the request (`t_size`) is greater than the number of available bytes in the current cacheline. If yes, then we calculate the remaining number of bytes left, divide it by the total number of bytes in a cacheline ($2^{\texttt{LineOffset}}$) to get the number of additional cachelines accessed, on top of the current one. Since the number of bytes in a cacheline is a power of 2, the division can be replaced with a bitwise right shift operation. If the transaction size is equal or less than the number of available bytes in the current cacheline, then it implies that the request is only accessing this one cacheline.

Algorithm 1 Pseudo-code to compute transaction size in cachelines.

```
1  // Current cacheline address.
2  curr_line_address = (r_addr[63:LineOffset] << LineOffset);
3
4  // Next cacheline address.
5  next_line_address = this_line_address + (1 << LineOffset);
```

```
6
7    // Number of available bytes on current line.
8    n_available_bytes = next_line_address - r_addr;
9
10   // To account for the current cacheline.
11   cachelines_affected = 1;
12   if (n_available_bytes < t_size) begin
13      cachelines_affected += ((bytes_in_burst - bytes_on_cur_line)
14                               >> LineOffset) + 1;
15   end
```

On the other hand, the address alignment can be computed quickly by applying the logical-OR operation to the cacheline offset bits of the address. Aligned addresses must have their offset bits as 0.

The filter also transmits the `Source ID` of the initiating manager. However, the design to discern the `Source ID` has to be platform-dependent. Under the AXI4 protocol, the manager IDs do not need to be sent along with the transaction packets. They are omitted entirely in some AXI4-based interconnect architecture. This means a conscious effort must be made to make this information available to the SPU. This can be done in various ways. For example, one can send the manager ID using the reserved *user* bits. The SPU must be designed to split out the user bits from the incoming AXI packets to isolate the `Source ID`s.

Our evaluation platform PULP, discussed in Section 5.1, uses the AXI4 implementation developed by [56]. The proposed interconnect in this work prepends the manager ID to the packet's original transaction ID before forwarding it to the subordinate module. The subordinate module also responds with the same extended transaction ID. This allows the interconnect to route packets belonging to different managers but sharing the same transaction ID. Thus, when connected between the manager port of the interconnect and a subordinate module, the SPU computes the `Source ID` by truncating the transaction ID of the incoming and outgoing packets to retrieve the manager ID.

2. **AW Channel Filter**

   The AW channel filter generates the write request event packets discussed in Point 3 of Section 4.1.2. When a new write is detected on the AW channel as indicated by the `AWVALID` signal, the filter creates a new event packet. The hardware design of this filter is similar to that of the AR channel filter presented above in Point 1.

3. **R Channel Filter**

   The R channel filter generates the read response event packets explained in Point 4 of Section 4.1.2. It monitors both the AXI packets on the AR channel sent by the manager and those on the R channel transmitted by the subordinate. The packets sent on the AR channel signify a new read request. The ones on the R channel indicate when an ongoing read transaction is finished. When the `RLAST` is set for an ongoing transaction, it implies that the subordinate is sending the last data packet of that transaction. In this case, the filter generates a new read response event packet and additional information, such as the index of the accessed address and the response latency.

   The address index is computed in a manner similar to the method described in Point 1. However, the logic required to compute the response latency is more complicated. As mentioned in Point 4 of Section 4.1.1, there is no unique key that can be used to identify individual requests. Therefore, to compute the latency of a request, the SPU uses a content-addressable memory (CAM) that creates a new entry when a new read transaction is started. When an ongoing transaction is completed, its corresponding entry is removed.

   Apart from the transaction ID, each entry has a few additional properties: the time at which the request arrived at the channel filter, whether this request is the oldest one in the set of active requests sharing the same transaction ID, or whether it is the youngest, and a pointer to the next-oldest request in the set. The SPU has its own timer that it uses to store the timestamp of the arriving requests. When a CAM entry is popped out, the response latency is calculated as the difference between the current SPU timer value and the timestamp at which the request first arrived.

   An example to explain the functioning of the CAM is presented through Figure 4.4. Entries with pointers as "*" are pointing to themselves. For the oldest and youngest columns, "Y", "N" imply yes, no, respectively. Let R_i be the set of all read requests with transaction ID i. Let $P(r_k)$ be the pointer of request r_k that points to the next-oldest request in the set to which r_k belongs. All timestamps are from when the request arrived at the R channel filter.

   Consider an empty CAM that has no valid requests. When a new request, r_1, with ID 1 arrives at time $T = 2$, it is populated into the CAM. Since there are no active requests with ID 1, this request is both the oldest and youngest in the set R_1. Its pointer, in this case, points back to itself. A new request, r_2, with ID 2, arrives at $T = 5$. This request, too, populates another entry in the CAM, being both the oldest and youngest in its set, R_2, with its pointer again pointing back to itself. At $T = 7$,

a new request, r_3, with ID 2 arrives. This request will then become the youngest request in the set R_2. The pointer of the previous youngest request, i.e., r_2, will now point to r_3. At $T = 8$, another request, r_4, with ID 2 enters the CAM. This request is now the youngest in R_2, and the previous youngest r_3 will now point to r_4. r_2 is still the oldest request in R_2. Therefore, $P(r_1) \to r_1$, $P(r_2) \to r_3$, $P(r_3) \to r_4$ and $P(r_4) \to r_4$.

At $T = 11$, the last packet for a transaction with ID 2 is received. Since the AXI4 protocol ensures that requests with with the same transaction ID targeting the same channel are ordered, it must be the last packet for the oldest read request in the set R_2. Thus, r_2 is popped from the CAM, its arrival timestamp is subtracted from the current clock cycle, and the response latency is calculated, which is 6. The request that $P(r_2)$ points to now becomes the oldest request in the set R_2. Therefore, r_3 is now the oldest request for R_2. At $T = 12$, the last packet for a transaction with ID 1 is received. The only request with transaction ID 1 is r_1. It is popped from the CAM, and its response latency is 11. The set R_1 becomes empty again. The CAM now only has two entries: r_3 and r_4.

4. **B Channel Filter**

   Under the AXI4 protocol, a written transaction is completed only when the subordinate module sends back a write response to the initiating manager via the B channel. For the SPU, the W channel is unused.

   The B channel filter generates the write response event packets discussed in Point 5 of Section 4.1.2. The hardware design of this filter is similar to the R channel filter presented in Point 3. It has a separate CAM, because under the AXI4 protocol, a read and a write transaction with the same transaction ID do not need to be ordered as they are targeting different channels.

   When a new write request is observed on the AW channel, a new CAM entry is populated with the same entry properties as that of the CAM in the R channel filter. A write request with a particular ID is terminated when a write response is observed on B, causing the CAM to pop out the oldest request from the set of all requests with the same ID.

4.2 Implemented APMU

The APMU is designed for a RISC-V compliant platform with AXI4-based interconnections, as per the specification explained in Section 3.2.3. In this section, we primarily

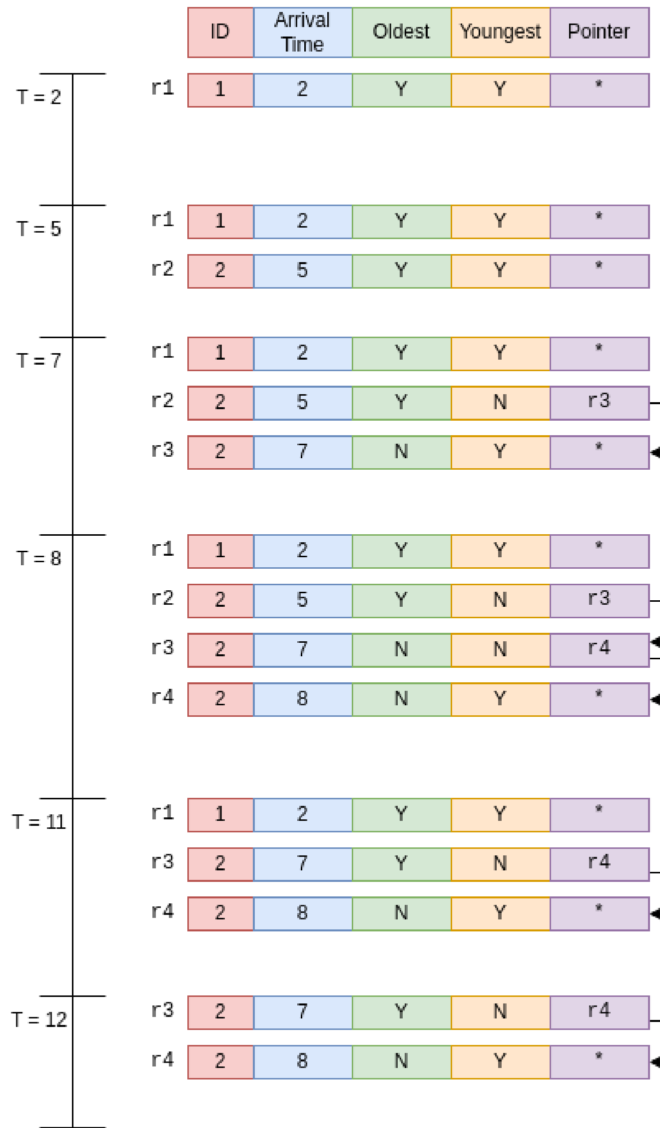


Figure 4.4: An example to explain the functioning of the SPU-CAM.

discuss the technology-specific design decisions made while developing the APMU, namely, the APMU memory map, its internal interconnect, and the APMU core.

Our APMU implementation uses an extended version of the Ibex RISC-V 32-bit processor ([57] [1], [58]). As the Ibex core can only support 32-bit operations, the implementation is constrained. The maximum number of counters in the design is limited to 32, with each

counter being 32-bit wide. As the counters support the overflow functionality, each counter has a separate wire that must be connected to the platform interrupt controller in the system. The APMU uses an internal AXI4-Lite interconnection. Lastly, the APMU memory map is the same as the one presented in Section 3.3, except that the page size is 4 KB which is the standard size for RISC-V pages.

4.2.1 APMU Interconnect

The APMU has an internal AXI4-Lite crossbar. We re-used the IP developed by the authors of [56]. The AXI4-Lite interface is similar to the full AXI4 interface discussed in Section 4.1.1 except that it lacks the infrastructure to support burst operations. All AXI4-Lite transactions are made of one beat only. We chose an AXI4-Lite crossbar over full AXI because we expect most of the accesses made to the APMU to be control accesses, i.e., accesses made to configure various APMU counter blocks. This reduces the area and power consumption of the APMU, as discussed later in Section 5.2. Only continuous accesses made to the instruction and data SPMs would benefit from burst support that comes with a full AXI4 crossbar. However, besides run-time tracing functionality, other mechanisms discussed in Section 2.2 and in Section 3.1 such as those implementing resource regulation and management, CFI-checkers, etc., do not need support for a large number of data accesses during their execution.

The APMU has both a manager and a subordinate port, allowing it to send and receive read/write transactions to and from the system. This functionality is important because it allows the APMU core to update other components of the platform. For example, the APMU core can alter the cache partitioning scheme dynamically depending on either user input or an internal allocation policy.

The APMU interconnect also allows the core to update the counter blocks by accessing their physical address. This feature allows the APMU core to reprogram counters to count different events as required.

4.2.2 APMU Core and SPMs

We chose the Ibex core by lowRISC ([57] [1], [58]) for the APMU instruction processor because of its low area footprint; the hardware synthesis results are presented in Section 5.2. Ibex is an open-source RISC-V-compliant processor designed specifically for control-oriented applications with limited arithmetic capabilities. The open-source implementation is highly configurable providing support for a branch predictor, instruction

cache, single-cycle multiplier, RISC-V extension for bit manipulation, etc. However, for our implementation, we have chosen the following processor configurations:

1. The Ibex core supports the RV32IMC ISA. RV32I is the base 32-bit instruction set for RISC-V processors; M indicates support for multiplication, and C indicates support for compressed 16-bit instructions.

2. The open-source implementation provides two pipeline options: a 2-stage pipeline where the two stages are instruction fetch (IF) and instruction decode plus execute (ID/EX), and a 3-stage pipeline where the write-back operation (WB) is split out from the previous ID/EX. For our purpose, we chose the smaller 2-stage pipelined version, as shown in Figure 4.5.

3. The chosen core has a fast multi-cycle multiplier which offers a reasonable trade-off between area and performance. It takes 3-4 cycles to complete a multiple operation. `MUL` instructions take 3 clock cycles, while `MULH` instructions take 4 clock cycles. In the RISC-V ISA, `MUL` instruction multiplies two 32-bit registers and returns the lower 32 bits, and `MULH` performs signed multiplication of two 32-bit registers and returns the upper 32 bits. The division hardware implements a multi-cycle long division algorithm which is slow and takes up to 37 clock cycles.

4. The processor does not have an instruction cache but rather a prefetch buffer, which reads from the instruction SPM. The core also does not have a branch predictor.

Apart from the above configuration, we added support for the functionalities presented in Point 5 of Section 3.2.3. This was done by extending the RISC-V ISA. We assigned the hitherto unused opcode `7’000_0111` to the APMU instructions, which are the following:

1. **Counter-read**

   The counter-read instruction transfers value from counters to registers. The instruction is encoded in the I-type format of RISC-V instructions, shown in Figure 4.6, with all unused bit fields set to 0. The `funct3` for this instruction is `3’b000`.

   Each counter in the APMU has a unique counter ID. The ID of the counter that is to be read is written into `rs1`. Counter-read reads the counter value and copies it into the register specified in `rd`.

2. **Counter-write**

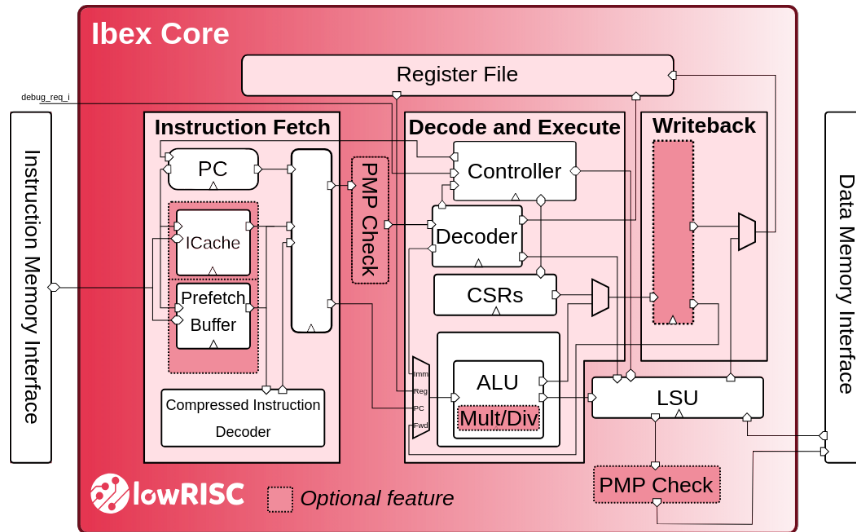


Figure 4.5: Ibex Pipeline [1] (writeback stage not included).

The counter-write instruction transfers value from registers to counters. The instruction is encoded in the S-type format, shown in Figure 4.7, with all unused bit fields set to 0. The `funct3` for this instruction is `3'b001`.

The ID of the counter that is to be updated is written into `rs1`. Counter-write copies the value from the register referred by `rs2` to the counter.

3. **Wait-for-Pending (WFP)**

   The WFP instruction stalls the core until the pending bit, introduced in Point 4 of Section 3.2.3, of one of the specified counters is set. The instruction is encoded in the R-type format, shown in Figure 4.8b, with the unused `rs2` field set to 0. The `funct3` and `funct7` fields of this instruction are `3'b010` and `7'h000_0000` respectively.

   The set of counters that have to be monitored is written as a bitmap into the `rs1` register wherein each bit maps to a APMU counter. Since the processor registers in Ibex are 32-bit wide, the maximum number of counters supported by the APMU is currently restricted to 32. The instruction will actively poll only those counters whose corresponding bit was set in `rs1`. If it observes that the pending bit of any one of these counters is set, the core exits the Wait-for-Pending mode and resumes operation. Counters that were not selected for polling, i.e., their corresponding bit is set to 0 in `rs1`, do not affect the processor behaviour. This implies that the core will not resume operation even if their pending bit is set.

31	30 25	24 21	20	19 15	14 12	11 7	6 0
imm[11]	imm[10:5]	imm[4:1]	imm[0]	rs1	funct3	rd	opcode

(a)

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

(b)

Figure 4.6: Counter-Read Instruction Format. Figure (a) shows the standard RISC-V I-type instruction format. Figure (b) shows the format of our custom counter-read instruction.

31	30 25	24 20	19 15	14 12	11 8	7	6 0
imm[11]	imm[10:5]	rs2	rs1	funct3	imm[4:1]	imm[0]	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)

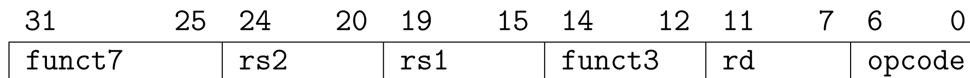
Figure 4.7: Counter-Write Instruction Format. Figure (a) shows the standard RISC-V S-type instruction format. Figure (b) shows the format of custom counter-write instruction.

When the instruction observes that the pending bit of one the selected counters is set, it also sets the corresponding bit for that counter in the `rd` register to 1. If multiple selected counters have their pending bit set in the same clock cycle, then all their corresponding bits are set in `rd`. The hardware automatically resets the pending bit when the APMU core exits the Wait-for-Pending mode.

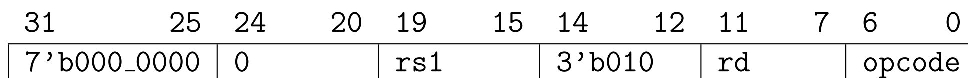
4. **Wait-for-Overflow (WFO)**

   The WFO instruction stalls the core until the overflow bit, introduced in Point 4 of Section 3.2.3, of one of the specified counters is set. The instruction is encoded in the R-type format, shown in Figure 4.8c, with the unused `rs2` field set to 0. The `funct3` and `funct7` fields of this instruction are `3'b010` and `7'h000_0001` respectively.

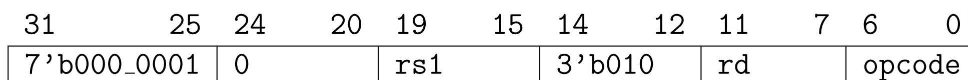
   Similarly to WFP, the set of counters that have to be monitored by this instruction is written as a bitmap into the `rs1` register wherein each bit corresponds to a counter.



(a)



(b)



(c)

Figure 4.8: Wait-for-X Instruction Formats. Figure (a) shows the standard RISC-V R-type instruction format. Figure (b) shows the format of custom WFP instruction. Figure (c) shows the format of custom WFO instruction.

> The instruction will actively poll only those counters whose corresponding bit was set in `rs1`. If it observes that the pending bit of any one of these counters is set, the core exits the Wait-for-Pending mode and resumes operation. Counters that were not selected for polling, i.e., their corresponding bit is set to 0 in `rs1`, do not affect the processor behaviour.
>
> When the instruction observes that a counter has overflown, it sets the corresponding bit for that counter in the `rd` register to 1. If multiple counters overflow in the same clock cycle, then all their corresponding bits are set in `rd`. The hardware automatically resets the overflow bit when the APMU core exits the Wait-for-Pending mode.

The APMU core has specialized read-and-write instructions for counters because accessing the counters via the internal AXI4-Lite crossbar can have up to 4 clock cycle delay. Instead, the counter-read and counter-write instructions only take 2 clock cycles. Comparatively, the load-store unit (LSU) in the Ibex core connected to the data SPM also takes 2 clock cycles. Lastly, a wire from the APMU core status register had to be connected to the controller IP of the Ibex processor, allowing the register to reset the core when its `Stall` bit was set.

4.3 Software Support

As discussed in Section 3.3, the APMU has several custom features: its counters can be configured to perform a variety of operations on the incoming event packets and its processor supports multiple custom instructions. These customizations can make its difficult for a user to use the APMU; particularly in writing code for its processor.

Considering this, we have developed a basic software stack for programming the APMU. It includes a set of library functions, and macros that can be used to implement custom APMU instructions in C. The stack includes a RISC-V GNU compiler [59] that can generate an assembly program from the input C code. This program can then be passed through our custom assembler, which converts it into bytecode to be uploaded into the APMU ISPM, as shown in Figure 4.9.

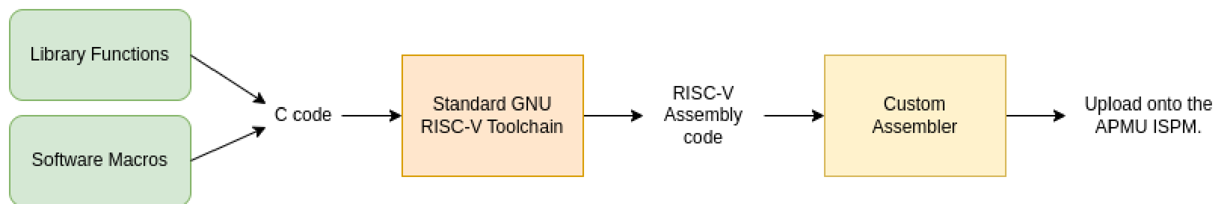


Figure 4.9: APMU Software Stack.

Chapter 5

Evaluation

This chapter evaluates the hardware implementation of our centralized performance monitoring infrastructure in terms of hardware utilization and its functionality through a latency-based regulation case study. In order to do this, we first define the evaluation platform used to test our infrastructure. Specifically, we are using a PULP-variant platform that we extended for our purpose. Our design is used to profile the platform based on synthetic programs to extract timing characteristics for the cores and memory modules, followed by the regulation case study and comparison against related works.

This chapter has three sections. In Section 5.1, we introduce the platform that we used to evaluate the functionality and performance of our infrastructure. In Section 5.2, we present the hardware synthesis results of our infrastructure and provide architectural insights into the design to explain the results. In Section 5.3, we define and implement our regulation case study. In this section, we also present a comparison against other related works.

5.1 Evaluation Platform

PULP, short for Parallel Ultra Low Power ([60], [61]), is an open-source silicon-proven RISC-V platform developed through collaboration between ETH Zurich and the University of Bologna. It is an open-source, scalable, and energy-efficient solution for embedded computer systems. To test out the platform, we realized a FPGA emulator on top of the AMD Virtex UltraScale+ FPGA VCU118 board [20].

5.1.1 Introduction

This section introduces the evaluation platform, including the changes we introduced to make it suitable for executing a latency-based regulation case study, which is defined in Section 5.3.

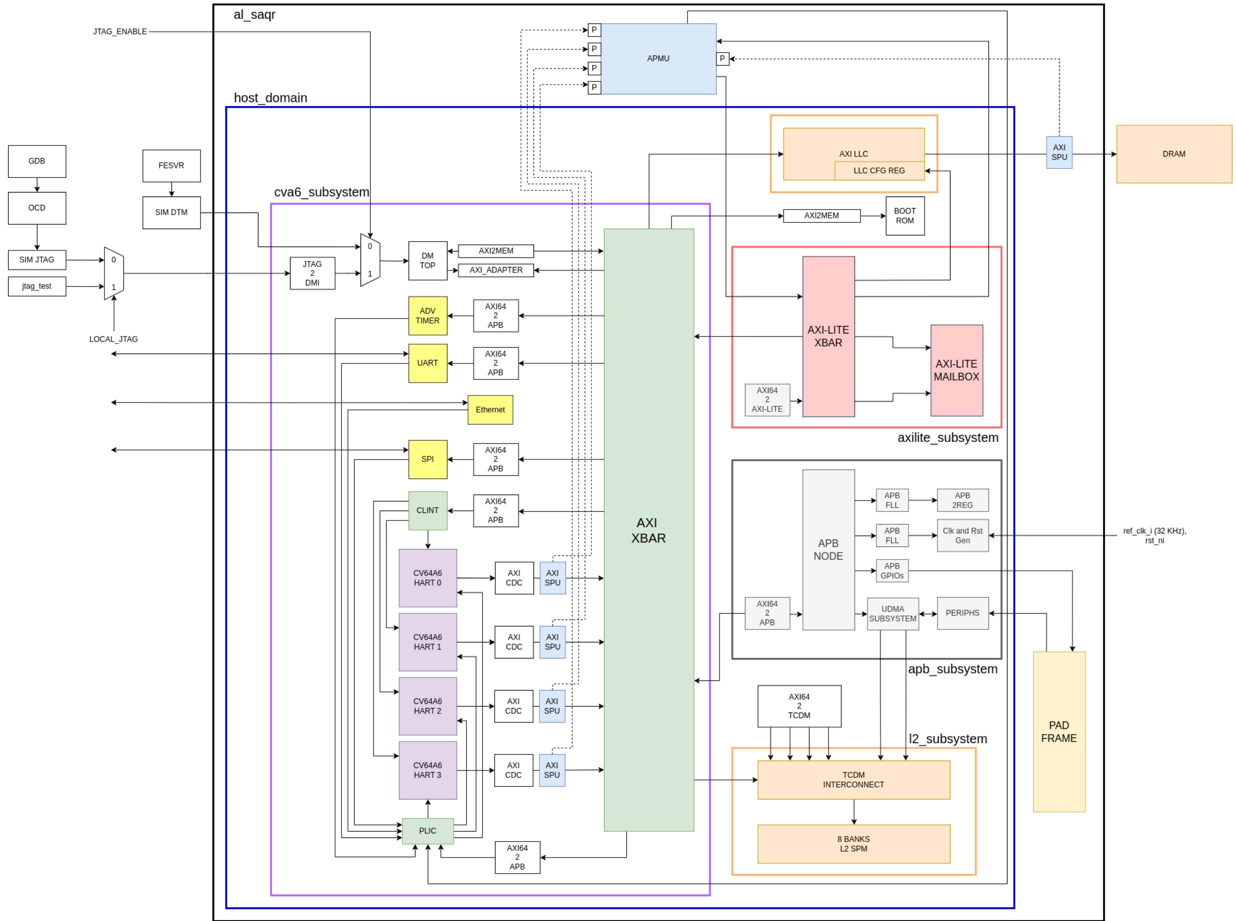


Figure 5.1: Evaluation Platform based on PULP, along with APMU and AXI4 SPUs.

The platform has multiple interconnections that connect various hardware modules together, as shown in Figure 5.1. The primary system interconnect is a full AXI4-based crossbar, which follows the AMBA AXI4 bus protocol discussed in Section 4.1.1. It is implemented using the open-source non-coherent on-chip communication architecture presented in [56]. The AXI4 crossbar has multiple managers, we describe the relevant managers below:

1. **CV64A6 cores**

   The original PULP-variant that we started with was a single-core platform. However, to generate meaningful results for our resource contention-based case study, we had to extend it to a multicore setup. The extended PULP-variant is a non-coherent multicore platform that has four CV64A6 cores, which belong to a class of 64-bit RISC-V application cores with a 6-stage, single-issue pipeline. It is an in-order core that supports RV64IMAC ISA, as specified in the RISC-V Volume I: Unprivileged ISA [62]. RV64I is the base 64-bit instruction set for RISC-V processors. M indicates support for multiplication, A for atomic operations, and C for compressed 16-bit instructions. This is because base RV64I instructions are 32-bit wide, while instructions from the compressed instruction set only occupy 16 bits. The core also implements the standard RISC-V Volume II: Privileged ISA [63], which includes three privilege levels: machine mode (M), supervisor mode (S), and user mode (U). The core also supports the Hypervisor extension [64], which adds the Hypervisor (H) mode to the privilege levels and replaces S and U modes with virtual supervisor (VS) and virtual user (VU), respectively. The machine level is the highest privilege level of the core and is mandatory for RISC-V processors; the remaining levels are optional. CV64A6 with the Hypervisor Extension is capable of running hypervisors like Bao [65].

   Each application core is identified by a unique `HART ID`. Both the platform interrupt controller and the debug module require the `HART ID` of the core to send an interrupt or debug request to it, respectively. Additionally, each CV64A6 core has a private L1 instruction and a private write-through, no-write-allocate data cache. Both caches have a cacheline size of 16 B. The L1 instruction cache has four ways, 256 sets, and a total size of 16 kB, while the L1 data cache has eight ways, 256 sets, and a size of 32 kB. Each core also has a four entry deep store buffer, i.e., until a store instruction is committed it remains in the store buffer. This also means that while reads are blocking, writes are non-blocking until the store buffer is filled up.

2. **Debug Module**

   The platform has a debug module based on the SiFive debug specification, version 0.13 from SiFive [66]. The CV64A6 core is compatible with this external debug specification and can be debugged using Open On-Chip Debugger (OpenOCD) [67] and the GNU debugger (gdb) [68]. The debug module can halt any core by sending it a debug request signal via a dedicated wire. Upon receiving a debug request, the core enters the debug mode, only exiting it when the debug module updates core-specific flags in its memory space. The debug module is responsible for booting up the application processors after the binary is loaded onto the main memory via the

JTAG interface. Additionally, the debug module can also be accessed via the AXI4 system interconnect.

For the thesis, we had to slightly modify the debug unit to allow the APMU to use it to regulate application cores as warranted. For this, we added a set of registers in the module. The APMU could write the `HART ID` of the core it wants to regulate to these specific registers using the AXI4 system crossbar. Depending on the register updated, the debug module would send out a halt or start request to the core with that specific `HART ID`.

The crossbar is responsible for routing read and write requests from managers to the subordinate modules and subsystems connected to it. It is important to note here that the AXI4 protocol does not put many restrictions on the transaction IDs associated with each data transfer. It is challenging to route multiple packets with the same transaction ID from different managers to different subordinates. To overcome this problem, a simple solution proposed by [56] is to prepend the transaction ID from the manager side with the manager ID. This implies that the transaction ID, as seen by the subordinate, includes the ID of the manager who sent the transaction request. For example, if core 0 with the manager ID `3'h3` (`3'b011`) sends a read request with transaction ID `4'hA` (`4'b1010`) to the LLC, then the crossbar will prepend the manager ID to the received transaction ID, and the subordinate will see a read request with transaction ID `7'h3A` (`7'b011_1010`). Having said this, a brief description of the various AXI subordinates modules and subsystem is given below:

1. **AXI4-Lite Subsystem**

   The AXI4-Lite system interconnect has two applications: a) it is used to write to the configuration register of the last-level cache (LLC), as discussed below in Point 2, and b) it also connects to a platform mailbox used for synchronization purposes.

2. **Memory Subsystem**

   The memory subsystem comprise of the platform LLC and the DDR4 main memory.

   (a) **Last-level cache (LLC)**
   The LLC has an AXI4-based subordinate port to receive fetch requests from external managers via the AXI4 crossbar. It is configurable in terms of size and can support either 2, 4, 8, 16, or 32 ways. The LLC is write-back with write-allocate, and can support multiple outstanding transactions. It also has

a set of configuration registers which can be read from and written to via the AXI4-Lite subsystem.

The original LLC was modified to introduce a way-based partitioning mechanism, that allows us to map AXI4 managers to particular ways. We also replaced the original random replacement policy of the LLC with a pseudo-least recently used (pLRU) replacement policy.

Apart from the above modifications, we also had to update the LLC pipeline to ensure that the transaction IDs of the requests were being sent out to the main memory. This is necessary if we want to distinguish between requests to main memory on the basis of the application core that initiated them.

All the functionalities of the LLC mentioned above, including the partitioning scheme, are controlled by their appropriate configuration registers. The LLC has a 32-bit register that can be used to set different ways of the LLC as an SPM. It also has registers controlling the LLC's flushing functionality and cache partitioning registers to designate ways to AXI4 managers. The BIST output of the tag SRAM is also stored in a configuration register that the application cores can access.

For our evaluation, the LLC is configured to have 1024 sets (S), 32 ways in each set (W), and eight blocks in each way per set (B), where one block is 64-bit wide (b). The size of the LLC is defined as $S \times W \times B \times b = 2MB$.

(b) **DDR4 Main Memory**

The LLC is further connected to the on-chip DDR4-2400 main memory through a AXI4-based interface. The Virtex VCU118 board has a dual 80-bit DDR4 component memory. Each component memory comprise a 2.5 GB set of five 256 MB x 16 (80-bit) DDR4 SDRAM devices.

3. **Interrupt Controllers**

The PULP platform, following standard RISC-V specification, has two interrupt controllers: the CLINT and the PLIC.

(a) **Core-Local Interrupt Controller (CLINT)**

The CLINT ([69], [70]) module is the RISC-V-compliant interrupt controller responsible for generating local interrupts. It manages timer interrupts and software interrupts for the application cores. This module is also used for generating inter-processor interrupts (IPIs).

(b) **Platform-level Interrupt Controller (PLIC)**

The PLIC module [71] is the RISC-V-compliant platform interrupt controller responsible for handling external interrupts triggered on the application cores. The PLIC allows users to set up different external devices,for example, the UART, µDMA, Ethernet, etc., as interrupt sources.

4. **Other Subordinate**

   The AXI4 interconnect is also used to program various other system peripherals and subsytems such as the UART, Ethernet, SPI, APB subsystem and L2SPM subsystem. The Advanced Peripheral Bus (APB) subsystem, based on the AMBA APB protocol [72], is used to route communication packets to various system peripherals such as the clock and reset generator module, the L2SPM subsystem, and the I/O DMA subsystem (µDMA). Meanwhile. the Level-2 SPM (L2SPM) subsystem, with 8 memory banks, is used for facilitating data transfers between I/O devices and the main memory via the µDMA.

Lastly, the primary operating frequency of the original platform, including the application cores, the system interconnections, the LLC, etc., is 50 MHz. The APMU, including the APMU core, and the SPUs are also running at 50 MHz. However, the DDR4 main memory operates at 650 MHz, which means that the current platform hardware is not sufficient to saturate the main memory.

5.1.2 Addition of the APMU and AXI4 SPUs to the platform

We added five AXI4 Snooping Units to the platform, as shown in Figure 5.1.

1. Four of them were placed between the output of the application cores' clock-domain crossing FIFOs (CDC FIFOs) and their respective subordinate ports on the AXI4 crossbar. These SPUs are responsible for monitoring the AXI requests sent out and responses received by the application cores, which includes all fetches and writebacks by the L1 instruction and data caches to the LLC, and all non-cacheable requests made to other system components, such as the UART, debug module, etc.

2. The last AXI4 SPU was connected between the LLC and the DDR4 main memory, allowing us access to the requests sent out by the LLC. The SPU is able to distinguish between requests initiated by different cores using their transaction IDs. The SPU in this case encodes this information in the `Source ID` bits of the event packet.

Combining the information gathered from the five SPUs, we can count the number of read and write requests to the LLC and main memory generated by each application core. We can also measure their cumulative request latencies and transaction sizes. The SPUs connected to the output of the application cores are able to monitor read and write transactions between the L1 cache of the application core and the LLC. The L1 cache of the application cores is write-through, write no-allocate. This means that read requests to the LLC are only generated due to read misses in the L1 private caches, both instruction and data. Meanwhile the writes to the LLC are caused due to the writes to the data cache which being write-through, sends it down the memory hierarchy right away. On the other hand, the SPU between the LLC and main memory can monitor reads and writes to the DRAM. Apart from read-misses, write-misses in the LLC also initiate a read from main memory because the LLC is write-back, write allocate. However, the SPU cannot distinguish between reads caused due to read misses versus those due to write misses as this information is consumed internally within the LLC. If these misses in the LLC evict a dirty cacheline, they will also initiate a write-back to the main memory. But again, we cannot distinguish between write-backs due to read misses, versus those due to write misses.

Each of the five SPUs are connected to separate ports in the APMU IP. The APMU is also connected to the AXI4-Lite subsystem as both a manager and a subordinate. This allows the user to write to the configuration registers of the counters and program the APMU core, while also allowing the core the ability to write to other system components. For example, the APMU can write to the debug module to halt or resume the application cores at will. We use this functionality in our regulation case study in Section 5.3. Apart from this, the overflow interrupts of the APMU counters are also connected to the PLIC as interrupt sources, one source per counter. This allows individual counters to trigger interrupts on the application cores upon overflow.

5.2 Hardware Synthesis Results

The PULP-variant introduced in Section 5.1 has been synthesized for the AMD Xilinx Virtex VCU118 board [20] using Vivado v2021.2. In the Virtex Ultrascale+ FPGA family ([73], [74]), each CLB contains one slice. There are two type of slices: SLICEL (Slice as Logic) and SLICEM (Slice as Memory). SLICEL LUTs are only used to implement logical circuits, while SLICEM can be also be configured as distributed-RAM. Each slice consists of eight 6-input LUTs and sixteen flip-flops.

The synthesis results are presented below. The CLB column represents the number of

CLBs utilized by the design. CLB registers represents the number of flip-flops used within the CLBs. LUT as Logic and LUT as Memory provide the distinction of how many LUTs have been used as logic (can be either SLICEL or SLICEM) and how many have been used as memory (only SLICEM). BRAM tiles and DSPs represent the number of Block RAM and DSP tiles used, respectively.

Table 5.1 shows the resource cost of the overall system, and the split between the resources utilized by the application cores, the APMU, the SPUs and the remaining system. In the design, the APMU has 32 counters, with a 4 KB instruction and a 32 KB data SPM. All the SPUs have a CAM with a depth of 16 entries and a FIFO of depth 8 entries. In the overall design, the APMU uses 7.13% of CLBs, 2.93% CLB registers, 7.41% LUTs and 3.11% BRAM tiles. However it has comparatively higher cost in the LUTs used as logic and memory, when compared to the other resourrces. This is primarily due to two factors:

1. The APMU counters and registers that are accessible via the AXI4-Lite APMU crossbar are implemented using flip-flops and registers. We have used the AXI4-Lite register IP provided by [56] because it is compatible with the AXI4-Lite crossbar IP; both have been developed by the same authors. The AXI4-Lite register IP is expensive because the current design has 32 32-bit counters, each with two 32-bit configuration registers, along with with two 32-bit APMU registers and one 64-bit APMU timer. Even though, the IP only allows one register to be read from or written to per clock cycle via the APMU crossbar, it allows multiple registers to be updated in the same clock cycle, by any other custom-hardware mechanism. This feature also drives up the logic cost of the design.

   But this feature is non-negotiable for the counters since the specification allows for multiple counters to be incremented simultaneously. However, the configuration and other APMU registers, aside from the APMU timer, do not have any such requirements. Therefore, in future, the AXI4-Lite register IP can be modified to optimize out this feature for the configuration and other APMU-critical registers except the timer.

2. The APMU ISPM consumes a large number of LUTs as memory when compared to other components in the platform. Even though, the APMU DSPM has been implemented using BRAM tiles, we were unable to synthesize the ISPM using BRAMs because of complexities in the instruction fetch logic of the APMU core. In our current design, the core is able to fetch one instruction every clock cycle, but when the ISPM is synthesized using BRAMs, the APMU processor fetches one instruction every two clock cycles, which is undesirable. We believe, this issue can be resolved in future by modifying the fetch logic of the APMU core.

Apart from these issues, the APMU consumes a very modest amount of FPGA resources. The APMU core is significantly smaller than the application processors, while the internal APMU AXI4-Lite crossbar consumes approximately half the amount of resources when compared to the full AXI system crossbar. Additionally, the SPUs consume different amount of resources depending on whether they are connected between the L1 cache of the processor and the AXI4 crossbar, or between the LLC and main memory, despite sharing the same configuration. We believe this is due to the placement and routing logic of the Synthesis tool, as this trend was observed in other SPU configurations as well.

Table 5.2 shows the resource cost of SPUs for different CAM and FIFO sizes. These results are from the SPUs that were connected to the output of the application cores and are only presented to show the relative increase in resource utilization for the SPU IP. These IPs support all the `Event Info` sub-fields from the SPU event table, presented in Section 4.1.2.

Lastly, Table 5.3 gives the resource breakdown of individual components of an SPU, connected at the output of the application core, and supporting all the `Event Info` sub-fields. As discussed in Section 4.1.3, each SPU event has its own pipeline with a channel filter and a FIFO. As expected, the content-addressable memory (CAM) in the R and B channel filters consume the most resource of the SPU; with each utilizing around 30% of the overall CLB registers and 31% of the overall LUTs for logic. The CAM in the current IP is synthesized using flip-flops and registers. However, in future, it might be possible to develop an SPU design with BRAM-consuming CAMs.

5.3 Case Study: Latency-based Regulation

In this section, we define our regulation case study that is used to evaluate the functionality of the APMU. Our case study implements a latency-based regulation on a cluster of cores, where one core, the core-under-analysis (CUA) is executing a critical real-time application while others (non-CUA) are running best effort tasks. Our mechanism aims to regulate the non-CUA cores on the basis of the average request latency of the CUA core. A target latency is set for the CUA, and if the current request latency of the CUA exceeds this target, the mechanism halts the non-CUA cores. Another approach to this would involve regulation on the basis of the total latency. Under this approach, the CUA is allowed to suffer a maximum amount of request latency called $L^{\max}_{cont}$. When the delay suffered by the CUA exceeds this value, the mechanism halts all other cores permanently. However, we chose to not implement this mechanism because it can halt the cores indefinitely, massively killing their performance.

IP Name	CLB	CLB Registers	LUT as Logic	LUT as Mem	BRAM Tiles	DSPs
PULP Platform	87,060	274,235	452,901	2,670	1,030	115
└ Core Cluster	39,796	119,556	200,416	0	264	108
└─ One CVA6 core	9,832	29,311	49,769	0	66	27
└ APMU	6,209	8,033	31,672	2,088	32	1
└─ APMU Xbar		706	9,635	0	0	0
└─ Event Filters		1,351	4,228	0	0	0
└─ Counters and Registers		4,424	11,766	0	0	0
└─ APMU Core		1,390	5,005	40	0	1
└─ APMU ISPM		72	175	2048	0	0
└─ APMU DSPM		78	220	0	32	0
└ SPUs	1,963	7,677	6,869	0	0	0
└─ One Core-to-LLC SPU	369	1,424	1,313	0	0	0
└─ LLC-to-Mem SPU	487	1,981	1,617	0	0	0
└ Rest of the system	41,055	146,646	220,813	582	734	6
└─ Full AXI Xbar		15,000	19,358	0	0	0

Table 5.1: Overall resource utilization cost of the system and its primary IPs.

SPU Config (CAM Depth, FIFO Depth)	CLB	CLB Registers	LUT as Logic
SPU (8, 8)	219	992	837
SPU (16, 8)	369	1,424	1,313
SPU (32, 8)	684	2,320	2,454

Table 5.2: Resource utilization cost of the different SPU configurations.

For a given execution of the task under analysis in isolation, we are measuring the following: the execution time (total running time of the program) E, the cumulative latency of read and write memory requests L_R, L_W, respectively, and the number of read and write requests K_R, K_W, respectively. By studying these measurements, we observed that the CV64A6 core stalls on reads but due to its four-entry deep store buffer, does not stall on write requests until the buffer is full, i.e., it stalls only after four consecutive write requests. This observation is also verified in Section 5.3.2. For this reason, we define the effective memory request latency L as follows,

$$L = L_R + L_W/4. \tag{5.1}$$

IP Name	CLB	CLB Registers	LUT as Logic
SPU (16,8)	369	1424	1313
└ AR Channel Filter		0	3
└ AW Channel Filter		0	3
└ R Channel Filter		433	404
└ B Channel Filter		433	409
└ AR FIFO		66	28
└ AW FIFO		58	24
└ R FIFO		146	44
└ B FIFO		146	44
└ RR Arbiter		2	0

Table 5.3: Resource utilization breakdown of an SPU.

Similarly, we define the effective number of memory requests K as,

$$K = K_R + K_W/4. \tag{5.2}$$

Considering Equation 5.1, we can define the total execution time E of a program as the sum of the computation time C and the effective request latency L,

$$E = C + L. \tag{5.3}$$

Since, the task under analysis might be executed multiple times, we want to define an upper bound for the measurable parameters. We let $E^{\max}$ to be an upper bound for E over all possible executions (i.e., the WCET in isolation), $K^{\max}$ to be an upper bound for K. We can then compute $C^{\max}$ to be an upper bound for $E - L$. Our case study is based on the following assumptions:

A1 **Isolation**

Co-running a task under analysis with interfering tasks can increase the effective latency L of its memory requests by a factor $\Delta \geq 0$; however, it does not affect the sequence of instructions executed by the program nor the number of memory requests it issues. Note: if there is a shared cache, this requires cache partitioning, plus no shared data or coherence, as is the case of our platform. Also, we must assume that the task is not preempted in either case.

A2 **Additivity (under isolation)**

If the effective latency L is increased by a factor Δ, then the execution time increases no more than Δ. We expect this behaviour, because as mentioned above, the computation time is not affected by contention. Moreover, in case of reads, extra read latency Δ_R can stall the core but by no more than Δ_R; and in the case of writes, extra read latency Δ_W can stall the core by only $\Delta_W/4$ because four writes need to be stored in the buffer consecutively. This increase represents the worst-case because if the four consecutive writes are more evenly spread out then the core would not have to stall at all.

Given a regulation parameter, $\alpha > 1$, the goal of our regulation mechanism is to ensure that the execution time of the task when running with interfering tasks is bounded by $E^{\max} \cdot \alpha$. This is called the *target setpoint*. Keeping assumption A2 in mind, we can calculate the maximum acceptable request latency, $L^{\max}(\alpha)$ for this case as follows,

$$L_{cont}^{\max}(\alpha) = E^{\max} \cdot \alpha - C^{\max}. \tag{5.4}$$

At any point in time during the execution, let L_{cont} and K_{cont} be the effective latency of memory requests and the effective number of memory requests made by the task under analysis following the same definitions from Equations 5.1 and 5.2, respectively. Therefore, these terms can be decomposed as follows,

$$L_{cont} = L_{cont,R} + \frac{L_{cont,W}}{4}, \tag{5.5}$$

$$K_{cont} = K_{cont,R} + \frac{K_{cont,W}}{4}, \tag{5.6}$$

where $L_{cont,R}$, $L_{cont,W}$ are the cumulative read and write request latencies at that point of time, and $K_{cont,R}$, $K_{cont,W}$ are the number of read and write request made by the task so far.

With these definitions, we want to prove the following: if at the end of the task, $L_{cont}/K_{cont} \leq L_{cont}^{\max}(a)/K^{\max}$ holds, then it must be $E_{cont} \leq E^{\max} \cdot \alpha$, where E_{cont} is the execution time of the program.

By the isolation (A1) and additivity (A2) property, the following must hold,

$$C_{cont} \leq C \tag{5.7}$$

$$\implies E_{cont} \leq E + L_{cont} - L, \tag{5.8}$$

where E and L are the execution time and memory latency of the corresponding execution in isolation, respectively, such that $\Delta = L_{cont} - L$.

By definition we have $C^{\max} \geq E - L$, because it is an upper bound over $E - L$.

$$\therefore E_{cont} \leq E + L_{cont} - L \leq L_{cont} + C^{\max} \tag{5.9}$$

Moreover, our stated assumption is,

$$\frac{L_{cont}}{K_{cont}} \leq \frac{L^{\max}_{cont}(a)}{K^{\max}} \tag{5.10}$$

$$\implies L_{cont} \leq \frac{L^{\max}_{cont}(a)}{K^{\max}} \cdot K_{cont} \tag{5.11}$$

By the isolation property (A1), it must be $K_{cont} \leq K^{\max}$,

$$\therefore E_{cont} \leq \frac{L^{\max}_{cont}(a)}{K^{\max}} \cdot K_{cont} + C^{\max} \tag{5.12}$$

$$\leq L^{\max}_{cont}(a) + C^{max} \tag{5.13}$$

$$= E^{\max} \cdot \alpha - C^{\max} + C^{\max} \tag{5.14}$$

$$= E^{\max} \cdot \alpha \tag{5.15}$$

Thus, the property holds.

Based on the detailed property, we describe our run-time mechanism as follows:

1. The mechanism first computes $L^{\max}_{cont}(\alpha)$, from Equation 5.4. Then it computes the average latency $avg_lat = L^{\max}_{cont}(\alpha)/K^{\max}$.

2. At run-time, the mechanism calculates the effective request latency L_{cont}, from Equation 5.5, where $L_{cont,R}$ and $L_{cont,W}$ are collected by reading the corresponding APMU counters.

3. The mechanism also reads the values of $K_{cont,R}$, $K_{cont,W}$ from the APMU counters to compute K_{cont}, as defined in Equation 5.6.

4. The mechanism stops the other cores so that the value of L_{cont}/K_{cont} does not become larger than avg_lat i.e., we aim to ensure that the running average of the request latencies is always less than or equal to that of the maximum acceptable average. And if the value of L_{cont}/K_{cont} falls below the avg_lat, it resumes them.

5.3.1 Implementation

In this section, we present the C code used to implement the regulation mechanism discussed above in Section 5.3. The program, given in Algorithm 2, expects the target average latency (L_{cont}/K_{cont}) to be written into the DSPM at address `DSPM_BASE + 0x80`. The program takes 20 clock cycles to start up, this includes the time taken to initialize various registers, the stack pointer, and to load up the target average latency from the DSPM. The polling period of the program is 93 clock cycles, i.e., the core re-computes the running average latency every 93 clock cycles. As discussed in Section 4.2.2, the APMU core consumes 2 clock cycles per memory and counter instruction. Additionally, the multiplier in the worst-case can take up to 4 clock cycles. To regulate cores, the APMU needs to communicate with the debug module through the system AXI4 crossbar. As described in Section 5.1, the APMU can halt and resume cores by sending their `HART IDs` to the debug module at specific addresses. A halt or resume request takes 8 clock cycles to be routed to the debug module via the system AXI4 crossbar and the module takes one clock cycle further to process it. Once the application core receives a halt request, it further takes around 5 clock cycles to halt. Therefore, the platform consumes 14 clock cycles to halt or resume an application core.

Algorithm 2 Regulation Algorithm

```
1  // APMU-specific addresses.
2  #define DEBUG_HALT    0x200
3  #define DEBUG_RESUME  0x208
4
5  #define TIMER_ADDR    0x10404000
6  #define DSPM_BASE_ADDR 0x10427000
7
8  // Macros for 32-bit read and write to memory addresses.
9  #define read_32b(addr)      (*(volatile int *)(long)(addr))
10 #define write_32b(addr, val_) (*(volatile int *)(long)(addr) = val_)
11
12 // Macros to program the Debug module.
13 #define debug_halt(core_id)  write_32b(DEBUG_HALT, core_id)
14 #define debug_resume(core_id) write_32b(DEBUG_RESUME, core_id)
15
16 // Macros for custom APMU core instructions.
17 // Counter read: rd = cnt[rs1]
18 #define counter_read(rd, rs1)    asm volatile ("cnt.rd\t%0,%1" : "=r" (rd) :
       "r" (rs1));
```

```
19  // Counter write: cnt[rs1] = rs2
20  #define counter_write(rs1, rs2) asm volatile ("cnt.wr\t%0,%1" :: "r" (rs1),
        "r" (rs2));
21
22  #define HALT          1
23  #define RESUME        0
24
25  void main () {
26      // Load the DSPM base address into SP.
27      asm volatile ("li sp,DSPM_BASE_ADDR");
28
29      // Number of requests and latencies, read from APMU counters.
30      volatile int unsigned cnt_n_read, cnt_read_lat;
31      volatile int unsigned cnt_n_write, cnt_write_lat;
32      // Used for total latency computation and comparisons.
33      volatile int unsigned current_lat = 0;
34      volatile int unsigned target_lat = 0;
35
36      // Used to loop over non-CUA cores.
37      int unsigned core_idx  = 0;
38      // Used to loop over APMU counters.
39      int unsigned counter_idx = 0;
40
41      // To verify correct operation.
42      int unsigned halt_status  = RESUME;
43
44      // This is the target avg_lat that APMU should maintain for CUA.
45      volatile int *target_addr = (int*)(DSPM_BASE_ADDR + 0x80);
46      volatile int unsigned target_avg_lat = read_32b(target_addr);
47
48      // Regulation mechanism starts.
49      while (1) {
50          // Read necessary APMU counters.
51          // Counter 0: Number of read requests
52          counter_read(cnt_n_read, 0);
53          cnt_n_read = cnt_n_read & 0x7FFFFFFF;
54
55          // Counter 1: Number of write requests
56          counter_read(cnt_n_write, 1);
57          cnt_n_write = cnt_n_write & 0x7FFFFFFF;
```

```
58
59            // Counter 2: Cumulative read latency
60            counter_read(cnt_read_lat, 2);
61
62            // Counter 3: Cumulative write latency
63            counter_read(cnt_write_lat, 3);
64
65            current_lat = cnt_read_lat + (cnt_write_lat >> 2);
66            target_lat = (cnt_n_read + (cnt_n_write >> 2)) * target_avg_lat;
67
68            if ((current_lat <= target_lat) && (halt_status == HALT)) {
69                // Resume all cores if halted.
70                debug_resume(1);
71                debug_resume(2);
72                debug_resume(3);
73                halt_status = RESUME;
74            } else if ((current_lat > target_lat) && (halt_status == RESUME)) {
75                // Halt all cores if not already halted.
76                debug_halt(1);
77                debug_halt(2);
78                debug_halt(3);
79                halt_status = HALT;
80            }
81        }
82    }
```

As mentioned in Section 2.2, MemPol [9] had also implemented a regulation policy using a small monitoring core that enjoyed access to performance counters in the application cores; however, it had tested a different regulation mechanism. In this section, we compare against those results of MemPol such as the access time of the counters, the time taken to halt or resume an application core, and the polling period. MemPol was implemented on the Zynq Ultrascale+ ZCU102 device, using the smaller real-time Cortex R5 core as the regulator core. The results presented in [9] are in ns and μs but since our platforms are running at different clock speeds, using time taken as a comparison metric does not give very meaningful insights. Therefore, we convert MemPol's results into clock cycles, using 500 MHz as the operating frequency for Cortex R5; as it is the default configuration on Vivado.

The Cortex R5 core consumes 137 and 108 clock cycles to read from and write to the performance counters in the application core, respectively; compared to the 2 clock cycle

delay per read/write operation for the APMU. Additionally, MemPol has a polling period of 3125 clock cycles, this is because of the large access time of the performance counters distributed across multiple processors. The work also assumes a worst-case poll-control delay (D) of 2000 clock cycles, where D is defined as the delay between observing that a core has exhausted its budget and the point where the core stops initiating further memory accesses. For our mechanism, the polling period is 93 clock cycles and the worst-case poll control delay D is 107 clock cycles, which is much faster than MemPol. These results though mechanism-dependent still show that our APMU infrastructure is more suitable for implementing fine-grained event-based software mechanisms than existing COTS platform.

5.3.2 Interference Tests on PULP

In this section, we present the timing characteristics for our evaluation platform, introduced in Section 5.3. To generate these characteristics we sweep the memory with one CV64A6, making read and write transactions, with different strides. This is our core-under-analysis (CUA). All these memory sweeps are made under different cases of contention: when no other core is running, when one other core is running, when two other cores are running, when three are running. The other cores are referred to as non-CUA cores. As our platform is non-coherent, we had to ensure that the memory regions used by different cores do not overlap. We allocate 25 MB of memory space for each core. The size of the LLC is 2 MB. Additionally, since the LLC is partitioning, the cores can only affect each other's request latencies and not their hit-miss ratio in the LLC.

As mentioned in Section 5.2, the platform clock runs at 50 MHz but the DDR4 memory has a clock speed of 650 MHz, because of this difference in clock frequency, we see relatively small increase in the response latency of the memory controller even under contention. However, the increase in overall response latency, combining both the LLC and main memory, is much larger because the effects of contention are more prominent for the LLC. As mentioned in Section 5.3, in worst-case every read can be blocking but we need four consecutive writes to have the same effect. In this section, we also verify our claim. Considering Equation 5.9 from Section 5.3, we have,

$$\therefore E_{cont} \leq L_{cont} + C^{\max} \tag{5.16}$$

$$\implies L_{cont} \geq E_{cont} - C^{\max}, \tag{5.17}$$

As L_{cont} is the effective latency during contention, it defined in Equation 5.5 as follows,

$$L_{cont} = L_{cont,R} + \frac{L_{cont,W}}{4}. \tag{5.18}$$

From here on, we have two cases: a) when the CUA is only making read requests, i.e., $L_{cont,W}$, $K_{cont,W} = 0$, and b) when the CUA is only making write requests, i.e., $L_{cont,R}$, $K_{cont,R} = 0$.

1. Case a) is as follows,

$$\because L_{cont} \geq E_{cont} - C^{\max} \tag{5.19}$$

$$\implies L_{cont,R} \geq E_{cont} - C^{\max} \tag{5.20}$$

   $L_{cont,R}$ is the cumulative latency observed by the APMU counters and can be defined as $L_{cont,R} = l_R \cdot K_{cont,R}$, where l_R is the latency of an individual read request. Therefore, we get,

$$l_R \geq \frac{E_{cont} - C^{\max}}{K_{cont,R}} \tag{5.21}$$

   This equation represents the lower bound for read request latencies.

2. Similarly, we have for case b),

$$L_{cont,W} \geq 4 \cdot (E_{cont} - C^{\max}) \tag{5.22}$$

   As $L_{cont,W}$ is the cumulative latency observed by the APMU counters, it can also be defined as $L_{cont,W} = l_W \cdot K_{cont,W}$, where l_W is the latency of an individual write request. Therefore, we get,

$$l_W \geq 4 \cdot \left(\frac{E_{cont} - C^{\max}}{K_{cont,W}} \right) \tag{5.23}$$

   This equation represents the lower bound for write request latencies.

The above results can be used to verify our claim regarding the nature of read and write requests in the platform and their impact on the CUA core. Figure 5.2, shows all the interference test results. The horizontal axis is the number of non-CUA cores that are interfering with the memory subsystem by making their own requests. The non-CUA

cores can make four type of requests: reads that hit in the LLC, reads that miss in the LLC, writes that hit in the LLC, and writes that miss in the LLC. The vertical axis is the request latency as measured by the APMU in clock cycles. In Figure 5.2a, we plot the contention suffered by the CUA while making read requests that hit in the LLC. Each solid line represent the latency observed by the APMU (l_R) for that type of contention, the corresponding dotted line represents the lower bound derived in Equation 5.21.

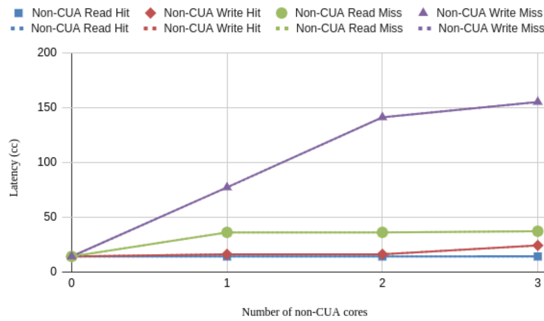
Similar results are obtained when the CUA initiates writes that hit in the LLC, as shown in Figure 5.2b. However, the case for read and write misses must be further categorized. A request that misses in the LLC can cause a write-back if it evicts a dirty cacheline. Considering this, we collect the latency of read and write miss requests made by the CUA, for both cases, when they cause a write-back and when they do not. It must be pointed out that since we generate non-CUA requests by sweeping over large independent memory regions with read or write requests, each write request that misses in the LLC will inevitably generate a write-back as well. This might explain why the contention is worse for write misses.

As can be seen in the results, for all the cases where the CUA is generating read requests, the solid and dotted lines overlap. This is because in the experiments the CUA is blasting the LLC with reads and every read stalls it. However, in case of writes the dotted line being the lower bound either overlaps or is under the solid line. However, in a few cases such as the non-CUA write miss case in Figure 5.2f, the dotted line is slight above the solid line; we suspect this is due to rounding errors, as the difference between the two is very small, of 1 clock cycles.

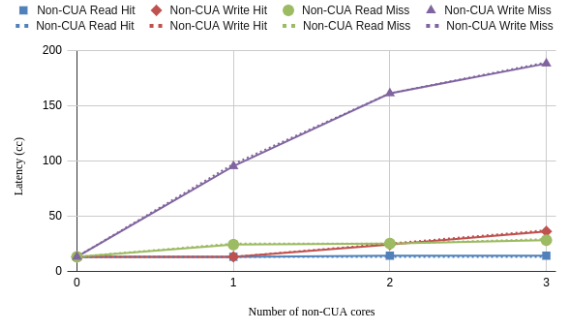
5.3.3 Regulation Results

In this section, we present the results of our case study. The experiments were conducted on a set of synthetic benchmarks, followed by the San Diego Vision Benchmark (SDVB) Suite [21]. The synthetic tests were conducted to achieve the following objectives: 1) to explore the functionality of the APMU, 2) verify the operation of our implementation, and 3) validate our regulation mechanism.

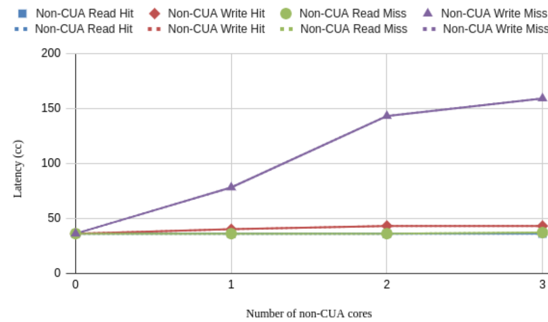
Synthetic Benchmarks: In these experiments, we executed synthetic all-read and all-write programs on the CUA core. Following the steps detailed out in Section 5.3, we first gathered results on the worst-case execution time of the task under isolation, to compute $E^{\max}$, $C^{\max}$ and $K^{\max}$. Next, we executed the regulation policy considering a set of target setpoints ($E^{\max} \cdot \alpha$), calculated by varying the regulation parameter α, from 0 to 50% with increments of 10%. The regulation program loaded into the APMU core ensured



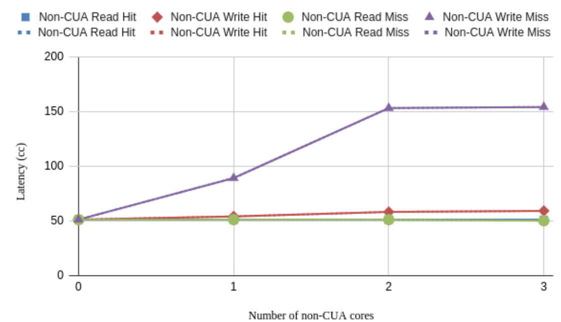
(a) CUA core makes read hits in LLC.



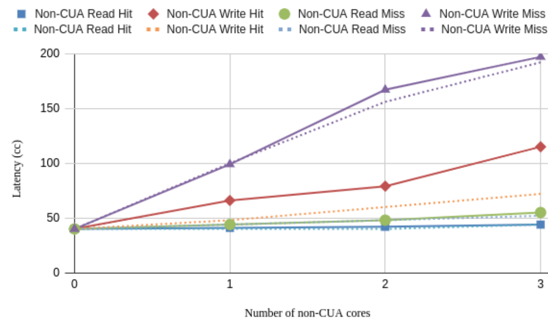
(b) CUA core makes write hits in LLC.



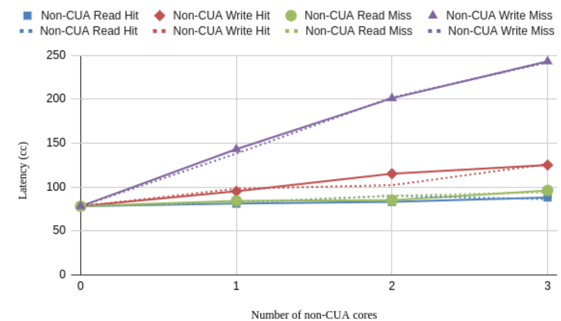
(c) CUA core makes read misses without write-backs in LLC.



(d) CUA core makes read misses with write-backs in LLC.



(e) CUA core makes write misses without write-backs in LLC.

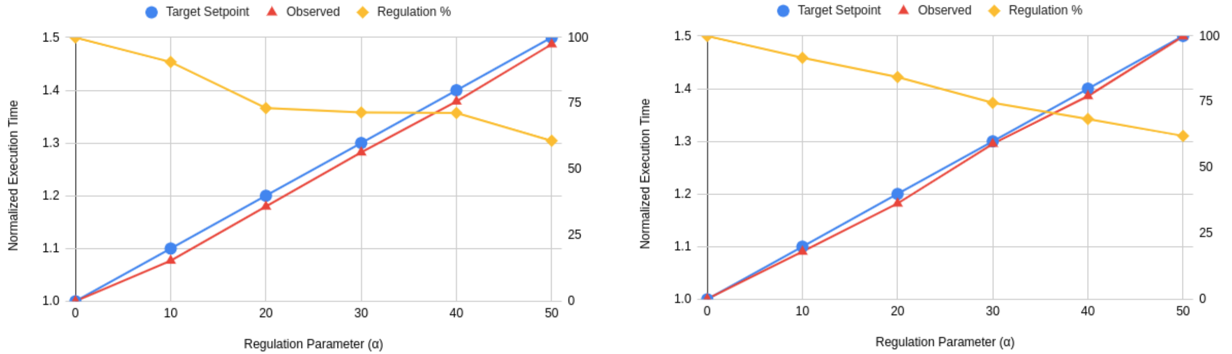


(f) CUA core makes write misses with write-backs in LLC.

Figure 5.2: Interference test results.

that the current average of the effective request latency is never larger than the maximum acceptable average.

The results are shown in Figure 5.3. The graphs show the difference between the target setpoint ($E^{\max} \cdot \alpha$) and the observed execution time (E_{cont}) at the end of the program. The regulation % shows the total amount of the time the non-CUA cores were halted compared to the total program run-time. The regulation % falls as the regulation parameter is increased, which is to be expected. More importantly, as we can see that the difference between the target setpoint and observed execution times are close (2.1% in the worst-case). This also shows that in the worst-case, while every read can stall the processor, four consecutive writes are needed to have the same impact.



(a) Regulation result for synthetic all-reads benchmark.

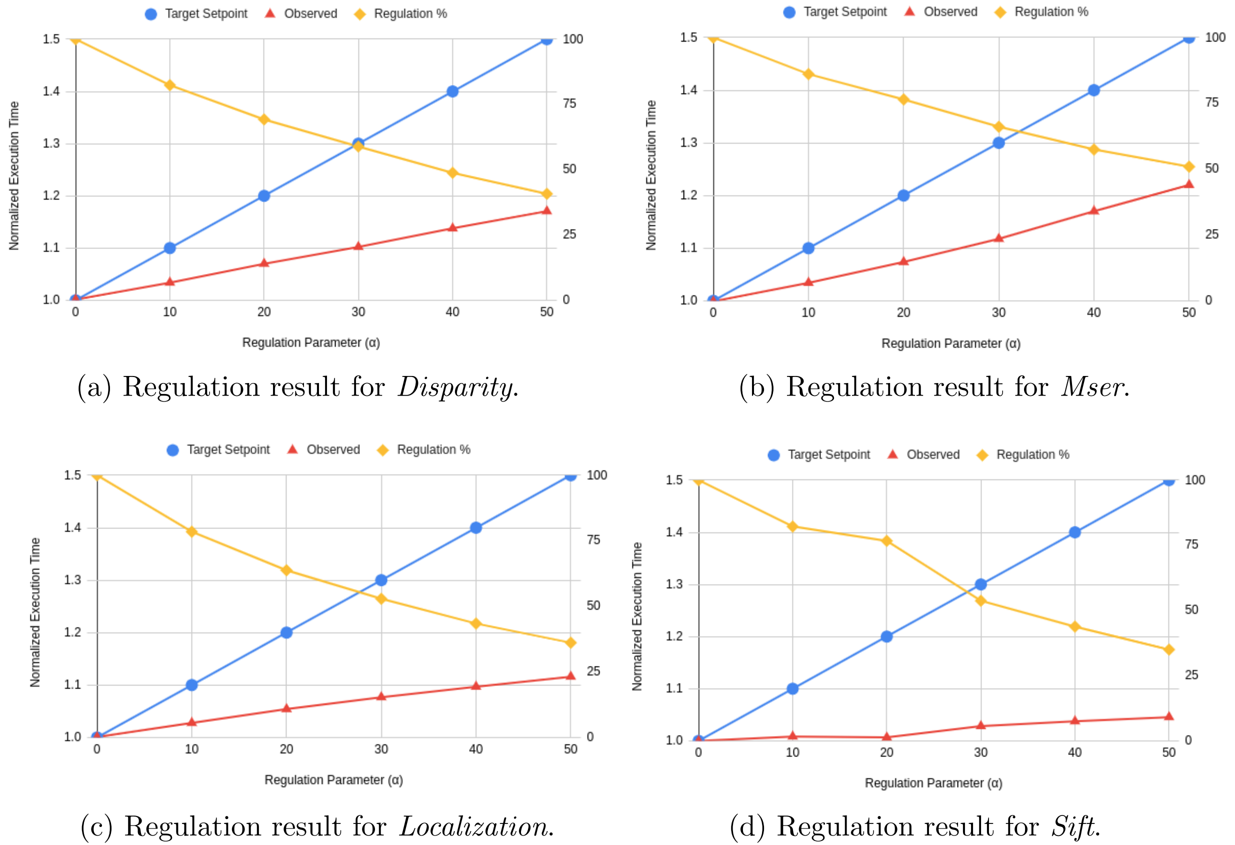
(b) Regulation result for synthetic all-writes benchmark.

Figure 5.3: Regulation Results for the Synthetic Benchmarks.

San Diego Vision Benchmarks: In this section, we aim to assess the performance of the APMU using the regulation mechanism described in Section 5.3 by executing real benchmarks. We evaluate our design by executing a set of representative benchmarks, similar to the work done by ([10], [9]). Figure 5.4 shows our results for *disparity*, *mser*, *localization* and an*sift* benchmarks. As discussed under synthetic benchmarks, the graphs shows the difference between the target ($E^{\max} \cdot \alpha$) and the observed (E_{cont}) execution times. The regulation % shows the total amount of time the non-CUA cores were halted during the program run.

In the worst-case, the difference between the target and observed execution is around 30.3%, which implies that the mechanism is over-regulating. Moreover, since the localization and sift are computationally intensive benchmarks with comparatively fewer memory accesses, their over-regulation margin is higher. This is because in our latency-based analysis, we assume that in the worst-case every four consecutive writes halt the core, which leads to a large upper bound in the worst-case effective latency L. Since it is possible that

if the writes are spread out enough, none of them will stall the core. But because we do not have information about the true nature of the writes due to the lack of an EVU in the application core, the mechanism is forced to over-regulate.



(a) Regulation result for *Disparity*.

(b) Regulation result for *Mser*.

(c) Regulation result for *Localization*.

(d) Regulation result for *Sift*.

Figure 5.4: Regulation Results for the SDVB Benchmark Suite.

Lastly, we also plot a timestamped graph in Figure 5.5 showing the regulation mechanism in action, as it halts and resumes the non-CUA cores for the disparity benchmark, with $\alpha = 50\%$. This graph also shows the different execution phases of the CUA core, when the core starts making a lot of memory accesses, the mechanism is forced to halt the non-cores.

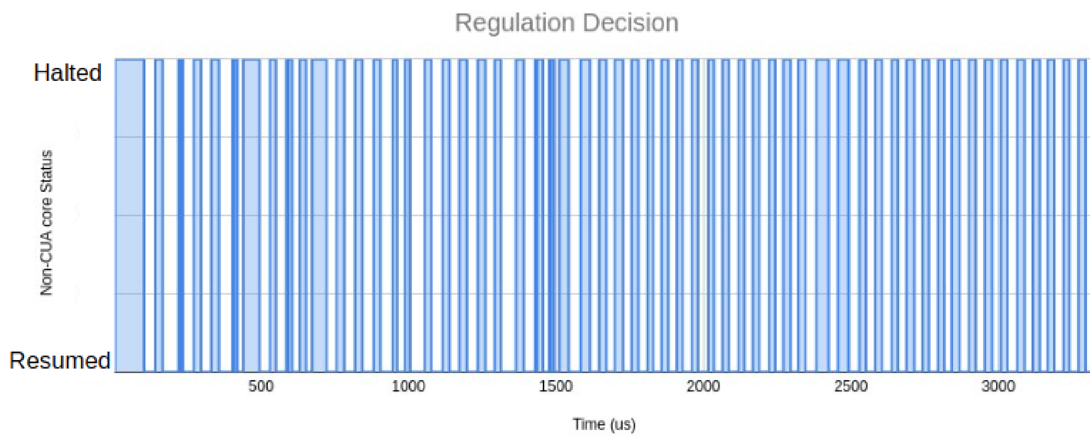


Figure 5.5: Timestamp of the regulation decisions taken by the APMU core during disparity at $\alpha = 50\%$.

Chapter 6

Conclusions

As highlighted in Chapters 1 and 2, performance monitoring frameworks are a crucial component of modern computer systems. They are used in a variety of applications from program profiling, optimization, tracing, to resource regulation and management, security checks, etc. Major industrial chip designers such as Arm and Intel have developed and proposed their own performance monitoring frameworks targeting, both, individual processors, and the overall system. However, the existing monitoring frameworks are decentralized, which means that the performance counters are embedded in the modules individually. This limits their reach in implementing complex software mechanisms spanning multiple hardware IPs, because of the access overhead and the added complexity in combining the various events to get a cohesive model of the system. Keeping the aforementioned problems in mind, we propose a centralized system performance monitoring infrastructure that comprises an Advanced Performance Monitoring Unit (APMU) and small Event Units (EVUs) that are spread out across the platform. These EVUs are dedicated monitoring units that when installed in a hardware module can be used to generate low-level event information about said module. The EVUs collect this information transmit it to the APMU in the form of a standardized event packet. The APMU consists of a set of smart performance counters that can operate on this incoming packet and store it. This allows the APMU to store event information from a variety of platform components, allowing users access to a rich amount of low-level system information. The APMU also has an instruction processor that was extended with additional functionality to provide support for the software mechanisms that are often used in conjunction with such devices and frameworks. In Section 3.1, we discuss several such applications that actively poll performance counters and take decisions accordingly.

Along side our proposed design, we also provide a detailed specification of the APMU,

and the EVU-APMU interface. For the sake of compatibility and easy integration, we have standardized the EVU-APMU interface. This allows users to connect any APMU-compatible EVU to the APMU, regardless of its implementation, allowing users a great deal of freedom in developing their own EVUs targeting IPs relevant to their use case. The specifications have been developed with the aim of extensibility in mind. Our discussion on the specification is presented in Chapter 3. Following which in Chapter 4, we introduce our implementation of the architecture, targeting an open-source RISC-V platform, with AXI4-based system interconnections. To implement the additional functionalities for the APMU core, we extend the RISC-V ISA to include our own custom instructions. We additionally also developed a software setup that allows us to efficiently write programs for the APMU processor, using these custom instructions. Our implementation also includes an AXI4 Snooping Unit, an EVU, designed for snooping AXI transactions.

In Chapter 5, we present our evaluation of the implemented design comprising the hardware synthesis results for the AMD Virtex UltraScale+ FPGA VCU118 device and the results of a latency-based regulation case study. In terms of hardware optimization, we believe that more work can be done in the APMU, as discussed in Section 5.2. Using the information available through our custom-EVU, we were to able implement a mechanism that regulates application cores on the basis of latency, which is more accurate than bandwitdh regulation. Our synthetic results validate the functionality of our design. However, the benchmarks results are less promising. This is because our chosen latency-based regulation mechanism is unable to accurately account for stalls caused by writes, due to the presence of the store buffer in the application cores. This issue is further exacerbated by the write-through nature of the L1 data cache of the cores; as this balloons up the number of writes, making our lack of information about the impact of the writes on application core even more glaring, causing us to over-regulate the other cores. However, our results show that the APMU is able to efficiently use the performance counters to run complex regulation policies.

In future, we plan to run our APMU device in a virtualized environment, running a hypervisor that can support the APMU. Using two-stage address translation, the hypervisor can map individual APMU counters to different virtual machines for faster access in a safe and controlled manner. The current APMU hardware already supports this feature. Additionally, by developing a software stack, we can ease the integration of the APMU and its custom-EVUs into the virutalized environment. This would also include an hypervisor-initiated initialization routine during boot time, software libraries to implement dynamic APMU configuration functionalities, etc. We also plan to implement our design on other platforms such as the Berkeley's FireSim [75].

On other fronts, we anticipate that our APMU design can be extended to implement

many of the applications listed in Section 3.1, provided that the necessary EVUs are developed. Moreover, we believe that the centralized aspect of the design can aid in the development of more complex software mechanisms that can effectively use the information the APMU provides.

References

[1] Ibex: An embedded 32 bit RISC-V CPU core; Ibex Documentation 0.1.dev50+gc9f4a32.d20240105 documentation. https://ibex-core.readthedocs.io/en/latest/. [Accessed 06-01-2024].

[2] Svilen Kanev, Juan Pablo Darago, Kim Hazelwood, Parthasarathy Ranganathan, Tipp Moseley, Gu-Yeon Wei, and David Brooks. Profiling a warehouse-scale computer. *SIGARCH Comput. Archit. News*, 43(3S):158–169, jun 2015.

[3] Memory System Resource Partitioning and Monitoring (MPAM), for A-profile architecture. https://developer.arm.com/documentation/ddi0598/latest/. [Accessed 30-11-2023].

[4] Intel® Resource Director Technology (Intel® RDT) framework. https://www.intel.com/content/www/us/en/architecture-and-technology/resource-director-technology.html. [Accessed 30-11-2023].

[5] AMD Xilinx. AXI Performance Monitor, Zynq UltraScale+ Device Technical Reference Manual (UG1085) — xilinx.com. https://www.xilinx.com/products/intellectual-property/axi_perf_mon.html. [Accessed 30-11-2023].

[6] 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, 2013.

[7] Michael Garrett Bechtel and Heechul Yun. Denial-of-service attacks on shared cache in multicore: Analysis and prevention. *CoRR*, abs/1903.01314, 2019.

[8] Nicolas Dagieu, Alexander Spyridakis, and Daniel Raho. Memguard, memory bandwidth management in mixed criticality virtualized systems - memguard kvm scheduling. 2016.

[9] Alexander Zuepke, Andrea Bastoni, Weifan Chen, Marco Caccamo, and Renato Mancuso. Mempol: Policing core memory bandwidth from outside of the cores. In *29th IEEE Real-Time and Embedded Technology and Applications Symposium, RTAS 2023, San Antonio, TX, USA, May 9-12, 2023*, pages 235–248. IEEE, 2023.

[10] Ahsan Saeed, Denis Hoornaert, Dakshina Dasari, Dirk Ziegenbein, Daniel Mueller-Gritschneder, Ulf Schlichtmann, Andreas Gerstlauer, and Renato Mancuso. Memory Latency Distribution-Driven Regulation for Temporal Isolation in MPSoCs. In Alessandro V. Papadopoulos, editor, *35th Euromicro Conference on Real-Time Systems (ECRTS 2023)*, volume 262 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 4:1–4:23, Dagstuhl, Germany, 2023. Schloss Dagstuhl – Leibniz-Zentrum für Informatik.

[11] 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*, SAC '23, page 461–470, New York, NY, USA, 2023. Association for Computing Machinery.

[12] A. de Lecea, M. Hassan, E. Mezzetti, J. Abella, and F. J. Cazorla. Improving timing-related guarantees for main memory in multicore critical embedded systems. In *2023 IEEE Real-Time Systems Symposium (RTSS)*, pages 265–278, Los Alamitos, CA, USA, dec 2023. IEEE Computer Society.

[13] Ali-Reza Adl-Tabatabai, Richard L. Hudson, Mauricio J. Serrano, and Sreenivas Subramoney. Prefetch injection based on hardware monitoring and object metadata. *SIGPLAN Not.*, 39(6):267–276, jun 2004.

[14] Deok-Jae Oh, Yaebin Moon, Do Kyu Ham, Tae Jun Ham, Yongjun Park, Jae W. Lee, Jung Ho Ahn, and Eojin Lee. Maphea: A framework for lightweight memory hierarchy-aware profile-guided heap allocation. *ACM Trans. Embed. Comput. Syst.*, 22(1), dec 2022.

[15] Malcolm Bourdon, Pierre-François Gimenez, Eric Alata, Mohamed Kaaniche, Vincent Migliore, Vincent Nicomette, and Youssef Laarouchi. Hardware-performance-counters-based anomaly detection in massively deployed smart industrial devices. In *2020 IEEE 19th International Symposium on Network Computing and Applications (NCA)*, pages 1–8, 2020.

[16] Mohamed El Bouazzati, Russell Tessier, Philippe Tanguy, and Guy Gogniat. A lightweight intrusion detection system against IoT memory corruption attacks. In

2023 26th International Symposium on Design and Diagnostics of Electronic Circuits and Systems (DDECS), pages 118–123, 2023.

[17] Abraham Peedikayil Kuruvila, Sayar Karmakar, and Kanad Basu. Time series-based malware detection using hardware performance counters. In *2021 IEEE International Symposium on Hardware Oriented Security and Trust (HOST)*, pages 102–112, 2021.

[18] Yihao Yang, Pengfei Qiu, Chunlu Wang, Yu Jin, Dongsheng Wang, and Gang Qu. Exploration and exploitation of hidden pmu events, 2023.

[19] AMBA AXI and ACE Protocol Specification Issue F.b, Arm Ltd., 2017. https://developer.arm.com/-/media/Arm%20Developer%20Community/PDF/Learn%20the%20Architecture/102202_0100_01_Introduction_to_AMBA_AXI.pdf?revision=369ad681-f926-47b0-81be-42813d39e132, 2020. [Accessed 12-12-2023].

[20] AMD Virtex UltraScale+ FPGA VCU118 Evaluation Kit — xilinx.com. https://www.xilinx.com/products/boards-and-kits/vcu118.html. [Accessed 11-02-2024].

[21] 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, 2009.

[22] ARM® Cortex®-A72 MPCore Processor Technical Reference Manual Revision r0p3 — developer.arm.com. https://developer.arm.com/documentation/100095/0003/?lang=en. [Accessed 13-02-2024].

[23] Intel® Microarchitecture Codename Nehalem Performance Monitoring Unit Programming Guide (Nehalem Core PMU). https://www.intel.com/content/dam/develop/external/us/en/documents/30320-nehalem-pmu-programming-guide-core.pdf. [Accessed 13-02-2024].

[24] Intel® 64 and IA-32 Architectures Software Developer Manuals — intel.com. https://www.intel.com/content/www/us/en/developer/articles/technical/intel-sdm.html. [Accessed 16-02-2024].

[25] “Zicntr” and “Zihpm” Counters — RISC-V Extension. https://wiki.riscv.org/display/HOME/Recently+Ratified+Extensions, March 2023.

[26] AMD Xilinx. AXI Performance Monitor, Zynq UltraScale+ Device Technical Reference Manual (UG1085) — docs.xilinx.com. https://docs.xilinx.com/r/en-US/ug1085-zynq-ultrascale-trm/AXI-Performance-Monitor. [Accessed 25-02-2024].

[27] AMD Xilinx. Zynq UltraScale+ MPSoC ZCU102 Evaluation Kit — xilinx.com. https://www.xilinx.com/products/boards-and-kits/ek-u1-zcu102-g.html. [Accessed 20-11-2023].

[28] AMD Xilinx. Platform Management Unit, Zynq UltraScale+ Device Technical Reference Manual (UG1085) — docs.xilinx.com. https://docs.xilinx.com/r/en-US/ug1085-zynq-ultrascale-trm/Platform-Management-Unit.

[29] CoreSight Architecture — developer.arm.com. https://developer.arm.com/Architectures/CoreSight%20Architecture. [Accessed 30-11-2023].

[30] Perf Wiki — perf.wiki.kernel.org. https://perf.wiki.kernel.org/index.php/Main_Page. [Accessed 13-12-2023].

[31] Arnaldo Carvalho de Melo. The New Linux 'perf' tools. http://vger.kernel.org/~acme/perf/lk2010-perf-paper.pdf, 2010. [Accessed 12-12-2023].

[32] Profiling with Arm MAP; Arm Documentation 2013 — developer.arm.com. https://developer.arm.com/documentation/102732/1910/Low-overhead-profiling-for-production-and-test-wrokloads-at-any-scale. [Accessed 16-02-2024].

[33] Intel® VTune™ Profiler User Guide — intel.com. https://www.intel.com/content/www/us/en/docs/vtune-profiler/user-guide/2023-1/overview.html. [Accessed 16-02-2024].

[34] Jason Mars, Lingjia Tang, Robert Hundt, Kevin Skadron, and Mary Lou Soffa. Bubble-up: Increasing utilization in modern warehouse scale computers via sensible co-locations. In *2011 44th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 248–259, 2011.

[35] Sriram Govindan, Jie Liu, Aman Kansal, and Anand Sivasubramaniam. Cuanta: Quantifying effects of shared on-chip resource interference for consolidated virtual machines. In *Proceedings of the 2nd ACM Symposium on Cloud Computing*, SOCC '11, New York, NY, USA, 2011. Association for Computing Machinery.

[36] Alex D. Breslow, Ananta Tiwari, Martin Schulz, Laura Carrington, Lingjia Tang, and Jason Mars. Enabling fair pricing on hpc systems with node sharing. In *SC '13: Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, pages 1–12, 2013.

[37] Hailong Yang, Alex Breslow, Jason Mars, and Lingjia Tang. Bubble-flux: Precise online qos management for increased utilization in warehouse scale computers. In *Proceedings of the 40th Annual International Symposium on Computer Architecture*, ISCA '13, page 607–618, New York, NY, USA, 2013. Association for Computing Machinery.

[38] Ram Srivatsa Kannan, Michael Laurenzano, Jeongseob Ahn, Jason Mars, and Lingjia Tang. Caliper: Interference estimator for multi-tenant environments sharing architectural resources. *ACM Trans. Archit. Code Optim.*, 16(3), jun 2019.

[39] Weifan Chen, Ivan Izhbirdeev, Denis Hoornaert, Shahin Roozkhosh, Patrick Carpanedo, Sanskriti Sharma, and Renato Mancuso. Low-Overhead Online Assessment of Timely Progress as a System Commodity. In Alessandro V. Papadopoulos, editor, *35th Euromicro Conference on Real-Time Systems (ECRTS 2023)*, volume 262 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 13:1–13:26, Dagstuhl, Germany, 2023. Schloss Dagstuhl – Leibniz-Zentrum für Informatik.

[40] Matteo Zini, Daniel Casini, and Alessandro Biondi. Analyzing arm's mpam from the perspective of time predictability. *IEEE Transactions on Computers*, 72(1):168–182, 2023.

[41] Parul Sohal, Michael Bechtel, Renato Mancuso, Heechul Yun, and Orran Krieger. A closer look at intel resource director technology (rdt). In *Proceedings of the 30th International Conference on Real-Time Networks and Systems*, RTNS '22, page 127–139, New York, NY, USA, 2022. Association for Computing Machinery.

[42] Giovani Gracioli and Antônio Auguto Fröhlich. On the influence of shared memory contention in real-time multicore applications. In *2014 Brazilian Symposium on Computing Systems Engineering*, pages 25–30, 2014.

[43] Ayoosh Bansal, Rohan Tabish, Giovani Gracioli, Renato Mancuso, Rodolfo Pellizzoni, and Marco Caccamo. Evaluating the memory subsystem of a configurable heterogeneous mpsoc. In *Workshop on Operating Systems Platforms for Embedded Real-Time Applications (OSPERT)*, volume 7, page 55, 2018.

[44] Eiman Ebrahimi, Chang Joo Lee, Onur Mutlu, and Yale N. Patt. Fairness via source throttling: A configurable and high-performance fairness substrate for multicore memory systems. *ACM Trans. Comput. Syst.*, 30(2), apr 2012.

[45] Kristof Du Bois, Stijn Eyerman, and Lieven Eeckhout. Per-thread cycle accounting in multicore processors. *ACM Trans. Archit. Code Optim.*, 9(4), jan 2013.

[46] Tanmaya Mishra, Thidapat Chantem, and Ryan Gerdes. Survey of control-flow integrity techniques for real-time embedded systems. *ACM Trans. Embed. Comput. Syst.*, 21(4), oct 2022.

[47] Fardin Abdi Taghi Abad, Joel Van Der Woude, Yi Lu, Stanley Bak, Marco Caccamo, Lui Sha, Renato Mancuso, and Sibin Mohan. On-chip control flow integrity check for real time embedded systems. In *2013 IEEE 1st International Conference on Cyber-Physical Systems, Networks, and Applications (CPSNA)*, pages 26–31, 2013.

[48] Tyler Bletsch, Xuxian Jiang, and Vince Freeh. Mitigating code-reuse attacks with control-flow locking. In *Proceedings of the 27th Annual Computer Security Applications Conference*, ACSAC '11, page 353–362, New York, NY, USA, 2011. Association for Computing Machinery.

[49] Victor van der Veen, Dennis Andriesse, Enes Göktaş, Ben Gras, Lionel Sambuc, Asia Slowinska, Herbert Bos, and Cristiano Giuffrida. Practical Context-Sensitive CFI. In *Proceedings of the 22nd Conference on Computer and Communications Security (CCS'15)*, October 2015.

[50] Ansam Khraisat, Iqbal Gondal, Peter Vamplew, and Joarder Kamruzzaman. Survey of intrusion detection systems: techniques, datasets and challenges. *Cybersecurity*, 2:20, Jul 2019.

[51] Michael Bechtel and Heechul Yun. Memory-aware denial-of-service attacks on shared cache in multicore real-time systems. *IEEE Transactions on Computers*, 71(9):2351–2357, 2022.

[52] Thomas Moscibroda and Onur Mutlu. Memory performance attacks: denial of memory service in multi-core systems. In *Proceedings of 16th USENIX Security Symposium on USENIX Security Symposium*, SS'07, USA, 2007. USENIX Association.

[53] Arm System MMU Support — developer.arm.com. https://developer.arm.com/Architectures/System%20MMU%20Support. [Accessed 21-02-2024].

[54] RISC-V Technical Specifications - Home - RISC-V International — wiki.riscv.org. https://wiki.riscv.org/display/HOME/RISC-V+Technical+Specifications. [Accessed 25-02-2024].

[55] RISC-V IOMMU Architecture Overview — open-src-soc.org. https://open-src-soc.org/2022-05/media/slides/RISC-V-International-Day-2022-05-05-14h10-Perinne-Peresse.pdf. [Accessed 25-02-2024].

[56] Andreas Kurth, Wolfgang Rönninger, Thomas Benz, Matheus A. Cavalcante, Fabian Schuiki, Florian Zaruba, and Luca Benini. An open-source platform for high-performance non-coherent on-chip communication. *CoRR*, abs/2009.05334, 2020.

[57] Pasquale Davide Schiavone, Francesco Conti, Davide Rossi, Michael Gautschi, Antonio Pullini, Eric Flamand, and Luca Benini. Slow and steady wins the race? a comparison of ultra-low-power risc-v cores for internet-of-things applications. In *2017 27th International Symposium on Power and Timing Modeling, Optimization and Simulation (PATMOS)*, pages 1–8, 2017.

[58] An update on Ibex, our microcontroller-class CPU core; lowRISC: Collaborative open silicon engineering — lowrisc.org. https://lowrisc.org/news/2019/06/an-update-on-ibex-our-microcontroller-class-cpu-core/. [Accessed 06-01-2024].

[59] GitHub - riscv-collab/riscv-gnu-toolchain: GNU toolchain for RISC-V, including GCC — github.com. https://github.com/riscv-collab/riscv-gnu-toolchain. [Accessed 25-02-2024].

[60] Davide Rossi, Igor Loi, Francesco Conti, Giuseppe Tagliavini, Antonio Pullini, and Andrea Marongiu. Energy efficient parallel computing on the pulp platform with support for openmp. In *2014 IEEE 28th Convention of Electrical & Electronics Engineers in Israel (IEEEI)*, pages 1–5, 2014.

[61] PULP platform. PULP Platform — pulp-platform.org. https://pulp-platform.org, 2014. [Accessed 30-11-2023].

[62] Andrew Waterman, Krste Asanović, and et al. *The RISC-V Instruction Set Manual, Volume I: User-Level ISA, Document Version 20191213*. December 2019.

[63] Andrew Waterman, Yunsup Lee, Rimas Avižienis, David Patterson, and Krste Asanović. *The RISC-V Instruction Set Manual, Volume II: Privileged Architecture, Document Version 1.7-draft.* May 2015.

[64] Andrew Waterman, John Hauser, Krste Asanović, and et al. *The RISC-V Instruction Set Manual, Volume II: Privileged Architecture (with Hypervisor Extension), Document Version 20211203.* December 2021.

[65] 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.

[66] Tim Newsome, Megan Wachs, and et al. *RISC-V Debug Specificatoin, Document Version 1.0-STABLE.* March 2019.

[67] Open On-Chip Debugger — openocd.org. https://openocd.org/. [Accessed 20-02-2024].

[68] GDB: The GNU Project Debugger — sourceware.org. https://www.sourceware.org/gdb/. [Accessed 20-02-2024].

[69] GitHub - CLIC (Core-local Interrupt Controller). https://raw.githubusercontent.com/riscv/riscv-fast-interrupt/master/clic.pdf. [Accessed 19-02-2024].

[70] GitHub - pulp-platform/clint: RISC-V Core Local Interrupt Controller (CLINT) — github.com. https://github.com/pulp-platform/clint. [Accessed 20-02-2024].

[71] GitHub - riscv/riscv-plic-spec: PLIC Specification — github.com. https://github.com/riscv/riscv-plic-spec. [Accessed 20-02-2024].

[72] AMBA APB Protocol Specification; Arm Documentation 2013 — developer.arm.com. https://developer.arm.com/documentation/ihi0024/latest/. [Accessed 19-02-2024].

[73] AMD Xilinx. UltraScale™ Architecture and Product Data Sheet: Overview (DS890) — docs.xilinx.com. https://docs.xilinx.com/v/u/en-US/ds890-ultrascale-overview. [Accessed 22-02-2024].

[74] AMD Xilinx. UltraScale Architecture Configurable Logic Block User Guide (UG574) — docs.xilinx.com. https://docs.xilinx.com/v/u/en-US/ug574-ultrascale-clb. [Accessed 22-02-2024].

[75] Sagar Karandikar, Howard Mao, Donggyu Kim, David Biancolin, Alon Amid, Dayeol Lee, Nathan Pemberton, Emmanuel Amaro, Colin Schmidt, Aditya Chopra, Qijing Huang, Kyle Kovacs, Borivoje Nikolic, Randy Katz, Jonathan Bachrach, and Krste Asanović. FireSim: FPGA-accelerated cycle-exact scale-out system simulation in the public cloud. In *Proceedings of the 45th Annual International Symposium on Computer Architecture*, ISCA '18, pages 29–42, Piscataway, NJ, USA, 2018. IEEE Press.

APPENDICES

Appendix A

Specification: EVU-APMU Interface

This specification discusses the interface that connects the output port of an Event Unit IP (EVU) to an input port on the Advanced Performance Monitoring Unit IP (APMU). An EVU is a hardware monitoring IP that can be designed and installed in any desired platform component to monitor hardware events and collect additional information related to them. The component in which the EVU is embedded for monitoring is called a *monitored* component. The EVU sends the observed event information to the APMU for counting and other purposes.

The interface is composed of two layers: the logical layer, which describes the information that the EVU sends to the APMU for each observed event, and the physical layer, which describes the physical connection used to transmit this information. These layers are detailed below:

1. **Logical Layer**

   The EVU sends an event packet for each individual event that it observes in the monitored component. Each event packet is composed of three fields: `Event ID`, `Event Info`, and `Source ID`, as shown in Figure A.1. They are explained below:

Source ID	Event Info	Event ID

   Figure A.1: The fields of an event packet.

   (a) **Event ID** (Mandatory)

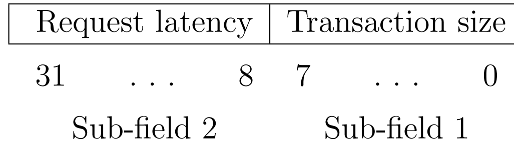


Figure A.2: Event Info bits transmitting request latency and transaction size.

The `Event ID` field encodes the type of event that was observed by the EVU in its monitored component. The `Event ID` is encoded as an unsigned integer. All events of an EVU must be mapped to unique `Event IDs`.

(b) **Event Info** (Optional)

An EVU can also transmit additional information alongside each event using the `Event Info` field. This field can be split into sub-fields where each sub-field represents a specific type of event information.

For example, in the case of an EVU monitoring read or write requests through a bus, transmitting information about the response latency and the transaction size might be beneficial. Therefore, the number of sub-fields for the `Event Info` field for these events would be 2, as shown in Figure A.2.

(c) **Source ID** (Optional)

The *source* of an event is the component external to the monitored component whose behaviour led to the generation of said event. For example, the processor that initiates a cache access is the source of that access and all events that are generated due to that access. Each potential event source is represented by a unique `Source ID`. Similar to `Event ID`, this field is also encoded as an unsigned integer.

Each EVU-APMU interface must have a document that helps users program the APMU to filter the event packets being transmitted through it. This document is called the *event table*. The event table contains the list of events the EVU can transmit through that interface, along with its `Event ID` encoding, the `Event Info` bits associated with each event, and the `Source ID` encoding. The example of a reduced event table is given in Figure A.1. This is event table of an AXI-based EVU. The `Source ID` encoding is omitted in this example.

Lastly, the `Event ID` and `Source ID` encodings are not expected to be consistent across EVUs tpyes. Therefore, different types of EVUs can re-use the same `IDs` to represent events/source.

<table>
<tr><th>Event</th><th>Event ID</th><th>Event Info</th></tr>
<tr><td>No event</td><td>0</td><td>NA</td></tr>
<tr><td>Read request</td><td>1</td><td rowspan="2">Size of transaction, unaligned transfer</td></tr>
<tr><td>Write request</td><td>2</td></tr>
<tr><td>Read response</td><td>3</td><td rowspan="2">Request latency, clock spent in contention</td></tr>
<tr><td>Write response</td><td>4</td></tr>
</table>

Table A.1: An exemplified EVU Event Table.

2. **Physical Layer**

   The physical layer defines the physical connection that is used to connect the EVU to a port on the APMU. The EVU can choose one of the provided physical layers for its interface, or it can choose to implement a different one; but in that case, they must provide a specification for the interface.

   (a) **Parallel Interface**

   The parallel interface is the simplest interface, wherein the EVU sends one event packet per clock cycle using a set of parallel wires. The size of the interface is equal to the size of the event packet of the EVU. This interface can only send one event packet per clock cycle but each packet can have additional event-related information packed in it, along side the `Event ID`.

   (b) **One-hot Interface**

   In this interface, the EVU sends a set of parallel wires to the APMU port, where each bit of the wire represents an individual event. This interface allows the EVU to send multiple event packets per clock cycle, where each packet is composed of only one bit.

   If the EVU and APMU are running in different clock domains then the interface must have clock domain crossing FIFOs (CDC FIFOs) to help facilitate asynchronous communication between the IPs.

Appendix B

Specification: Advanced Performance Monitoring Unit

This specification presents the Advanced Performance Monitoring Unit (APMU) that enables users to implement various event-dependent software mechanisms. The APMU, as shown in Figure B.1, has the following features:

1. Each EVU output port connects to a dedicated port on the APMU, represented by a unique `Port ID`.
2. A configurable number of counter blocks, wherein each counter block has one counter, two configuration registers (`EventSelCfg` and `EventInfoCfg`) and an arithmetic-logic unit (ALU). The counter blocks have support to generate interrupts in case of overflow.
3. Each counter block has its separate event filter which processes the event packets received from all APMU ports to generate the corresponding signals to update its counter.
4. An instruction processor, with complementary instruction and data scratchpad memories (SPMs), capable of executing program code stored in said SPMs.
5. A subordinate port that allows components, external to the APMU, to program the counter blocks, the processor and its SPMs.
6. A manager port that connects to the rest of the platform, allowing the APMU core to read/write to non-APMU mapped memory addresses.

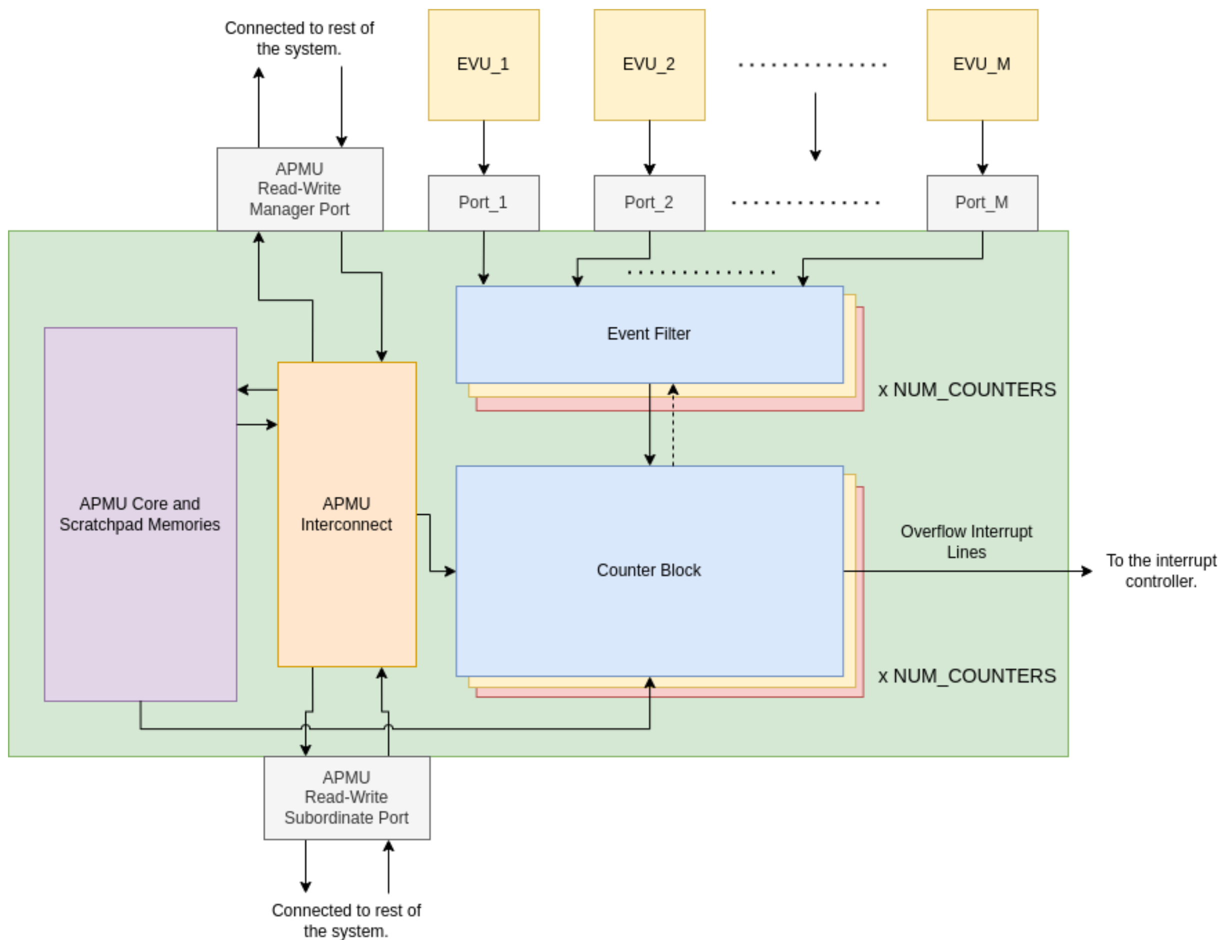


Figure B.1: APMU Architecture.

B.1 APMU Control and Status Registers

The core has three control and status registers:

1. **APMU Timer**

   The APMU has a 64-bit cycle counter. It is a read-only register necessary for time-keeping operations.

2. **Boot Address Register**

Figure B.2: `Timer` register of the APMU core.

The APMU has a boot address register (`BootAddr`), which is read- and write-able. The core always boots from the address written into this register. The bit width of the register (`YLEN`) is dependent on the bit width of the APMU processor.

Figure B.3: `BootAddr` register of the APMU core.

3. **Status Register**

   The status register (`Status`) is a 32-bit read- and write-able register that is required to reset the APMU core. In the current specification, the 0th bit of the register is called the `Stall` bit. If this bit is set, the APMU core stalls. The core will restart from the address written into `BootAddr` register, described in Point 2 of B.1.

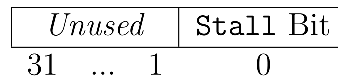


Figure B.4: `Status` register of the APMU core.

B.2 APMU Ports

The APMU has multiple ports, one for each EVU interface connected to it. An EVU can be connected to multiple APMU ports through multiple interfaces. The output of every APMU port is passed downstream to a set of counters through their respective event filter. For each port, the size of this counter set can range from one to all the counters in the APMU.

Each APMU port is represented by a unique `Port ID`. Since EVUs of different types can re-use the same `Event ID` for different events, it is important to add another layer of

identification to the design. This is achieved through the `Port ID`, as the `Port ID` helps identify the EVU connected to it. Once the EVU is known, the user can also decode the events received by comparing its `Event ID` to the event table of the interface.

B.3 Event Filter

Each counter block, discussed in Section B.4, has its own event filter. This module processes all the event packets received from the APMU ports and depending on the configuration registers of its counter, filters out unnecessary events. The configuration registers specify a set of events, that its counter should either count or operate upon, using the `Event ID`, `Source ID` and `Port ID` fields. These events are called *selected events*. The event filters should be capable of accepting multiple selected events received simultaneously from one or more ports. As mentioned in Section B.2, each APMU port is fanned out to either all or a subset of event filters in the APMU. The ports can send the event packets received from their respective EVU downstream to only those the event filters, and by extension counters, that are connected to them.

Before continuing this section, please read Table B.1.

B.4 Counter Block

Each APMU counter is bundled together with its configuration registers and a programmable ALU to constitute a *counter block*, as shown in Figure B.5. The specification does not limit the number of counter blocks in the APMU. This section is divided into four parts, explaining each component of the counter block.

1. **APMU Counter**

   The APMU counter has a parameterizable bit width `XLEN`, as shown in Figure B.6. The two most significant bits of the counter are called its pending and overflow bits, while the rest are used for standard counter operation. The pending bit is set whenever the counter is incremented during its count or functional operation mode. The overflow bit is set when the `XLEN-2` bit-wide counter overflows. If overflow interrupt is enabled in the `EventInfoCfg` register then the counter triggers an interrupt upon overflow.

Counter Mode of Operation
An APMU counter can be programmed to operate in two modes: *count* or *functional* mode. The operation mode of the counter is controlled via the `Operation` Bit in the `EventInfoCfg` register, discussed in Point 2b of Section B.4. The two modes are explained below: 1. **Count mode** When the `Operation` bit is reset to 0, the counter is said to be operating in count mode. In this mode, the counter is incremented by the number of selected events received across all APMU ports connected to its event filter. 2. **Functional mode** When the `Operation` bit is set to 1, the counter is said to be operating in functional mode. In this mode, the event info bits associated with the selected event packet are operated upon by the ALU of the counter. The output computed by the ALU is then updated into the counter. This mode should only be set in scenarios where the counter is not expected to receive multiple selected events in one clock cycle. If the event filter does receive multiple selected events while its counter is in functional mode then the filter can arbitrarily pick the event info bits of any one of the selected event packets. This is because otherwise the ALU hardware will become unnecessarily complicated if it is expected to operate on event info bits from multiple event packets simultaneously.

Table B.1: Counter Mode of Operation.

2. **Configuration registers**

   The counter block has two configuration registers: the Event Selection Register (`EventSelCfg`), which is used to specify the IDs of the selected event, and the Event Info Register (`EventInfoCfg`), which is used to specify details regarding the functional operation mode of the counter. The formats of the registers are detailed below:

   (a) **Event Selection Register (`EventSelCfg`)**

   The `EventSelCfg` register has the fields described in Figure B.7. Let `ESLEN` be the bit width of the `EventSelCfg` register. The `Event ID Value` and `Event ID Mask` are used to specify the set of selected events that the counter should count or

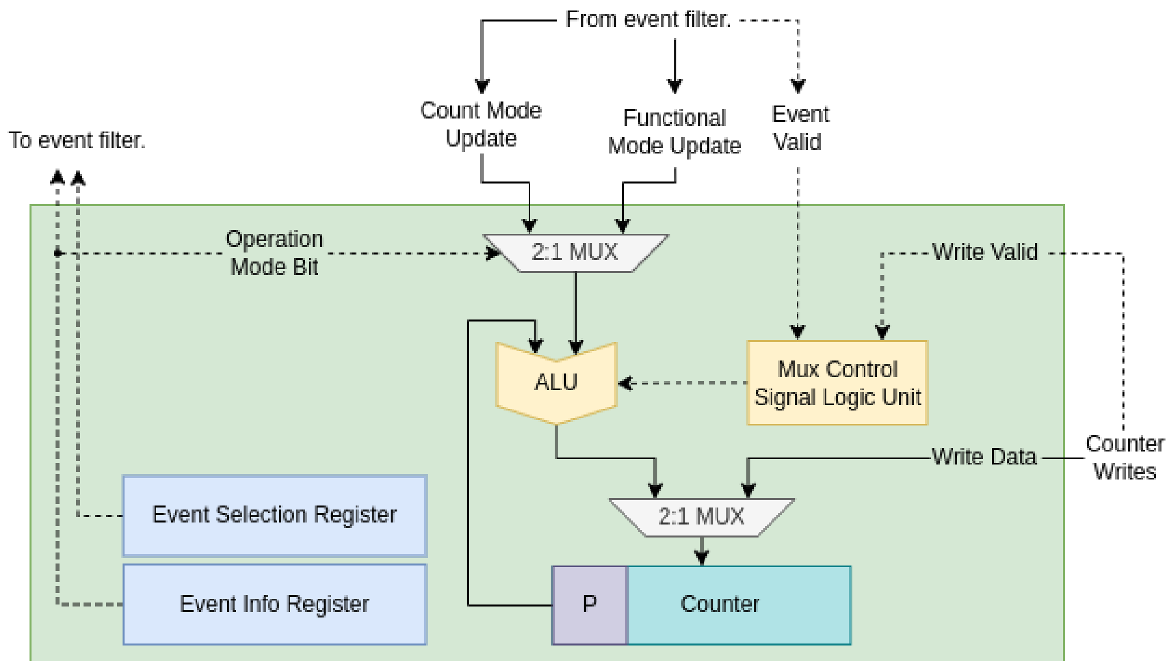


Figure B.5: APMU Counter Block.

Pending Bit	Overflow Bit	[XLEN-2]-bit Counter
XLEN-1	XLEN-2	XLEN-3 ... 0

Figure B.6: Format of a XLEN-bit APMU Counter.

operate on. The event filter will only accept those events whose Event IDs satisfy the following boolean expression:

(Event ID AND Event ID Mask) == Event ID Value

The size of these fields is equal to the minimum number of bits needed to specify the Event ID of any of the events that can be sent by the EVUs connected to this particular counter through the APMU ports.

Similarly, the event filter can be configured to filter event packets based on their Source ID and the Port ID of the port that received the packet. Counters will only increment for events whose Event ID, Source ID, and receiving port's Port ID are in the set specified by the EventSelCfg register. The size of the Source ID Mask and Source ID Value is computed similar to the

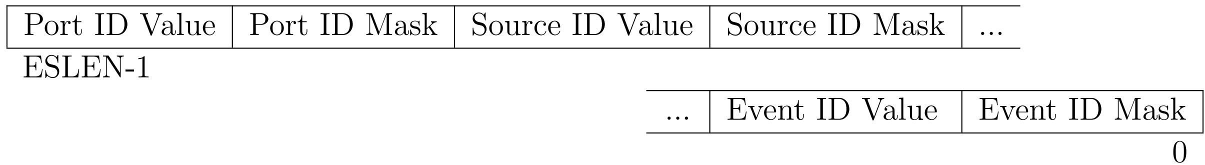


Figure B.7: Format of `EventSelCfg` register.

`Event ID Mask` and `Event ID Value`. On the other hand, the size of the `Port ID Mask` and `Port ID Value` is equal to the minimum number of bits needed to the store the `Port ID` of any of the APMU ports connected to the counter.

Note: There are no restrictions on the number of bits used for any of the `ID`s. For instance, consider an APMU connected to three EVUs. It will only have three ports. In this case, the system architect can use the following `Port ID`s: `3'b001`, `3'b010`, `3'b100`. This allows them to configure counters to count events transmitted by different combinations of ports. If `Port ID Mask` and `Port ID Value` are set to `3'b001` and `3'b000` then any event packet arriving at a port with ID `3'bXX0` will be accepted; here `X` represent "do not care" bits. This means both ports `3'b100` and `3'b010` will be considered but not `3'b001`.

(b) **Event Info Register (`EventInfoCfg`)**

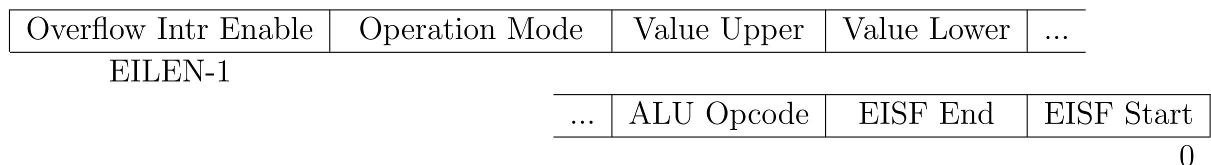


Figure B.8: Format of `EventInfoCfg` register.

The `EventInfoCfg` register has the fields described in Figure B.8. Let `EILEN` be the bit width of the `EventInfoCfg` register. The bit width of some of the fields is dependent on the types of EVUs connected to the APMU. The explanation of each field is as follows:

i. **EISF Start** and **EISF End**
   Event Info Sub-field Start (EISF Start) and End (EISF End) fields are used to specify the starting and ending bits of the `Event Info` field that the ALU should operate upon. This is because these bits can be subdivided to

pack various metadata together, as discussed in Point 1b of Appendix A. Therefore, the user needs to specify which of the metadata sub-field, i.e., which contiguous set of the event info bits, the ALU should operate upon. Both of these fields can take any value between 0 and the maximum number of bits used for the `Event Info` field out of all the connected EVUs. The ALU slices the `Event Info` field associated with a selected event according to these starting and ending bits, both inclusive, as written in `EISF Start` and `EISF End`, respectively. The ALU uses these *sliced event info bits* as an input operand, while the other input operand is the counter data.

ii. **ALU Opcode**
The ALU of the counter block can be programmed to perform specific operations by writing to this field. A set of supported opcode and their corresponding operations are given in Table B.2. The supported ALU operations are described below:

A. `Addition`
The ALU adds the sliced event info bits to the APMU counter.

B. `KeepMax`
The ALU compares the sliced event info bits against the counter value and stores the larger of the two back into the counter. Using this operation, the ALU can be programmed to capture the maximum encountered value of an event metadata.

C. `KeepMin`
The ALU compares the sliced event info bits against the counter value and stores the smaller of the two value back into the counter. Using this operation, the ALU can be programmed to capture the minimum encountered value of an event metadata.

D. `Increment: Eq`
The ALU compares the value of the sliced event info bits to the value written in the `Value_L` field of `EventInfoCfg`. If the two values are equal then the counter is incremented by 1.

E. `Increment: NotEq`
The ALU compares the value of the sliced event info bits to the value written in the `Value_L` field of `EventInfoCfg`. If the two values are not equal then the counter is incremented by 1.

F. `Increment: LessThan`
   The ALU compares if the value of the sliced event info bits is less than the value written in the `Value_L` field of `EventInfoCfg`. If yes then the counter is incremented by 1.

G. `Increment: GreaterThan`
   The ALU compares if the value of the sliced event info bits is greater than the value written in the `Value_L` field of `EventInfoCfg`. If yes then the counter is incremented by 1.

H. `Increment: LessThanEqual`
   The ALU compares if the value of the sliced event info bits is less than or equal to the value written in the `Value_L` field of `EventInfoCfg`. If yes then the counter is incremented by 1.

I. `Increment: GreaterThanEqual`
   The ALU compares if the value of the sliced event info bits is greater than or equal to the value written in the `Value_L` field of `EventInfoCfg`. If yes then the counter is incremented by 1.

J. `Increment: InRange`
   The ALU compares if the value of the sliced event info bits is in the range [`Value_L`, `Value_U`] (both inclusive). If yes then the counter is incremented by 1.

K. `Increment: NotInRange`
   The ALU compares if the value of the sliced event info bits is in the range [`Value_L`, `Value_U`] (both inclusive). If not then the counter is incremented by 1.

L. `Add: X`
   These operations are similar to the `Increment: X` operations. The only difference is that if the evaluation condition is true then the sliced event info bits are added to the counter. Here, `X` can be any of the following relational operator: `Equal`, `NotEqual`, `LessThan`, `GreaterThan`, `LessThanEqual`, `GreaterThanEqual`, `InRange`, `NotInRange`.

iii. **Value Lower** (**Value_L**) and **Value Upper** (**Value_U**)
   The Value Lower and Value Upper fields are used in conjunction with the

Operation	Opcode
`Addition`	`00000`
`KeepMax`	`00001`
`KeepMin`	`00010`
`Increment: Eq`	`00011`
`Increment: NotEq`	`00100`
`Increment: LessThan`	`00101`
`Increment: GreaterThan`	`00110`
`Increment: LessThanEqual`	`00111`
`Increment: GreaterThanEqual`	`01000`
`Increment: InRange`	`01001`
`Increment: NotInRange`	`01010`
`Add: Eq`	`01011`
`Add: NotEq`	`01100`
`Add: LessThan`	`01101`
`Add: GreaterThan`	`01110`
`Add: LessThanEqual`	`01111`
`Add: GreaterThanEqual`	`10000`
`Add: InRange`	`10001`
`Add: NotInRange`	`10010`

Table B.2: ALU operations and their opcodes.

ALU to evaluate the specified boolean condition. The values written into these two fields are treated as `Unsigned Integer`. The specification does not limit the size of the two fields.

iv. **Operation Mode**
This bit is used to specify the operation mode of the APMU counter, as discussed in Table B.1. If this bit is reset to 0 then the counter will only track the number of selected events and the ALU will not perform any event info-specific operation. If this bit is set to 1 then the counter is in functional mode and will update based on the ALU computation result.

v. **Counter Overflow Enable bit**
The MSB of the `EventInfoCfg` register is the `Counter Overflow Enable` bit. If the bit is set then upon overflow the counter will generate an overflow

interrupt. Otherwise, it will not generate any interrupt. In both cases, the counter resumes operation after an overflow without resetting its overflow bit. This must be handled in software.

B.5 APMU Core and Memory Infrastructure

The APMU has an instruction processor capable of executing program code. It should be supported by instruction and data scratchpad memories (SPMs). The specification also endorses additional functionalities for the APMU core to improve its performance when executing event-dependent programs, such as an exclusive datapath to access counters and to extend its capabilities, such as the ability to write to other memory locations in the system. These functionalities are mentioned below:

1. **Support to read and write counters efficiently.**

   The core should have an efficient datapath to reduce the access latencies for the APMU counters.

2. **Support to read and write to rest of the platform memory.**

   The core should be able to read and write from non-APMU mapped memory addresses. This allows the core to interact with other system components and can be useful for implementing event-dependent mechanisms.

3. **Support for Wait-for-X functionality.**

   The counters have two special bits: pending and overflow, as detailed in Section B.4. The pending is set to 1 whenever the counter increments in count or functional mode. The overflow bit is set to 1 when the `XLEN-2` bit counter overflows. To effectively use these bits, the specification proposes two functionalities:

   (a) **Wait-for-Pending**

   Under this functionality, the core enters a busy wait mode, polling the pending bit of a subset of counters. It exits the busy wait mode when the pending bit of any of the specified counters is set to 1. When the core exits the Wait-for-Pending, it also resets the pending bit of the counters that were set.

   (b) **Wait-for-Overflow**

   Similarly, under this functionality, the core enters a busy wait mode, polling the overflow bit of a subset of counters. It exits the busy wait mode when the

overflow bit of any of the specified counters is set to 1. When the core exits the Wait-for-Overflow, it also resets the overflow bit of the counters that were set.

4. **Support to trigger interrupts.**

   The core should be able to trigger interrupts in other application cores and devices.

5. **External support to reset and halt the core.**

   The APMU should have support to halt and reset the APMU core using external platform components. This feature is aided by the `Status` and `BootAddr` registers, discussed in B.1.

B.6 APMU Interconnect

The APMU also has an internal interconnect that enables communication between the APMU core and other APMU components such as the SPMs, counters, configuration registers, etc. The APMU has a manager port that allows the APMU core to access non-APMU memory addresses in the platform. Additionally, the APMU also has a subordinate port that allows other components in the platform to access internal APMU components. This allows external platform components to initialize and configure the APMU for operation.

B.7 Software Design Considerations

To facilitate integration in virtualized systems, the APMU counter blocks are spaced apart in the physical memory. This allows the hypervisor to place different counter blocks on different physical pages. Each of these pages can then be mapped to the virtual memory of a different virtual machine. An example memory map is shown in Table B.3 where each counter block is spaced exactly one page apart.

Counter Block	Register	Address
NA	APMU Timer APMU Core Status Register APMU Core Boot Address	base base + 0x8 base + 0xc
0	Event Selection Register Event Info Register	base + 0x10 base + 0x14
1	Event Selection Register Event Info Register	base + 0x18 base + 0x1c
...	...	...
31	Event Selection Register Event Info Register	base + 0x108 base + 0x10c
0	Counter	base + page_size
1	Counter	base + 2 × page_size
...	...	...
31	Counter	base + 32 × page_size

Table B.3: Example of an APMU Memory Map.