TÉCNICO LISBOA

Performance Monitoring and Event-based Sampling for RISC-V

Tiago Alfredo Lopes Rocha

Thesis to obtain the Master of Science Degree in

Electrical and Computer Engineering

Supervisor: Prof. Dr. Pedro Filipe Zeferino Aidos Tomás
Dr. Nuno Filipe Simões Santos Moraes da Silva Neves

Examination Committee

Chairperson: Prof. Dr. António Manuel Raminhos Cordeiro Grilo
Supervisor: Prof. Dr. Pedro Filipe Zeferino Aidos Tomás
Member of the Committee: Prof. Dr. Aleksandar Ilic

December 2023

Acknowledgements

At this moment, the end of my academic journey draws near. It was a fun and challenging part of my life I'm happy to have shared with so many wonderful people. Many of which were indispensable for me to complete this chapter and to whom I must thank.

First I would like to thank some of the Professors who crossed my path. Starting, of course with my thesis advisors, Prof. Pedro Tomás and Dr. Nuno Neves, as well as Prof. Nuno Roma. Without them and their ideas, suggestions and corrections this work would have not been possible. One more Professor from IST I owe a special thanks is Prof. João Paulo Ferreira da Silva, who truly made me believe that with hard work and dedication, I could complete this degree. Also, to my high school Professor who unfortunately is not among us anymore, José Luís Moraes, in whose classes I learned that Engineering was the right path for me.

Then, to my partner Patícia, who has always been by my side and supported me through the highs and lows of not only this work but also of my life, never letting me forget who I am and giving me unconditional support.

To my parents, Cristina and Rui, for raising me into the person I am today and always being proud of what I achieve. Also to my sister, Telma, who was my first teacher, showing me how to write and count; the true beginning of my academic education.

I would also like to thank Diogo and Vitor, for sharing these years with me, helping me in times of trouble and celebrating with me in times of victory.

Finally, but not less importantly, I thank everyone I met along the way, every friend I made and every Professor I was taught by. I'm certain they all helped me reach this point.

This work was partially supported by national funds through Fundação para a Ciência e a Tecnologia (FCT) under projects UIDB/50021/2020, 2022.06780.PTDC, and from the European High Performance Computing Joint Undertaking (JU) under Framework Partnership Agreement No 800928 and Specific Grant Agreement No 101036168 (EPI SGA2). The JU receives support from the European Union's Horizon 2020 research and innovation programme and from Croatia, France, Germany, Greece, Italy, Netherlands, Portugal, Spain, Sweden, and Switzerland.

Declaration

I declare that this document is an original work of my own authorship and that it fulfills all the requirements of the Code of Conduct and Good Practices of the Universidade de Lisboa.

Abstract

Increased attention to RISC-V open Instruction Set Architecture (ISA), has fueled its move from embedded devices to the high-performance computing arena, with the proliferation of RISC-V-based accelerators. However, the absence of powerful performance monitoring tools often results in poorly optimized applications and, consequently, limited computing performance. While the RISC-V ISA already defines a Hardware Performace Monitor and offers support for the Linux *perf_events* subsystem, research and development on RISC-V-based devices have been more focused on architectures and compilers rather than tools to support monitoring performance. To overcome this limitation, the introduction of *PAPI* library support for RISC-V processors is proposed in this thesis, and a Precise Event Sampling system specification compatible with future *PAPI* integration is presented along with a minimal implementation proof-of-concept. The conducted testing and evaluation of the *PAPI* port were carried out on a SiFive Unmatched board, but the proposed changes, and the corresponding implementation, are easily portable to other systems. The proof of concept for RISC-V Precise Event Sampling was implemented on a CVA6 processor.

It was found that, when compared to directly using *perf_events*, *PAPI* presents a large overhead; 83360μs in comparison with *perf_events* 100.24μs. Nevertheless, most of it (81200μs) is concentrated in the initialization of the library, which only occurs once per program execution.

Keywords: RISC-V Processors, Performance Monitoring, Precise Event Sampling, PAPI

Resumo

A crescente atenção dada à arquitetura de código aberto RISC-V impulsionou a sua transição de dispositivos embedded para a arena de computação de alto desempenho, com a proliferação de aceleradores baseados em RISC-V. No entanto, a ausência de ferramentas de monitorização de desempenho frequentemente resulta em aplicações pouco otimizadas e, consequentemente, num desempenho de computação limitado. Embora a arquitetura RISC-V já defina um Monitor de Desempenho de Hardware e ofereça suporte para o subsistema *perf_events* do Linux, a pesquisa e o desenvolvimento em dispositivos baseados em RISC-V têm-se concentrado mais em arquiteturas e compiladores do que em ferramentas de suporte à monitorização de desempenho. Para superar essa limitação, a introdução do suporte à biblioteca *PAPI* para processadores RISC-V é proposta neste documento, juntamente com a apresentação de uma especificação do sistema de Amostragem Precisa de Eventos compatível com uma futura integração no *PAPI*, juntamente com uma prova de conceito de implementação mínima. Os testes e a avaliação do *PAPI* foram realizados em uma placa SiFive Unmatched, mas as alterações propostas e a implementação correspondente são facilmente portáveis para outros sistemas. A prova de conceito para a Amostragem Precisa de Eventos em RISC-V foi implementada num processador CVA6.

Foi constatado que, em comparação com o uso direto do *perf_events*, o *PAPI* apresenta um overhead significativo; 83360μs em comparação com 100.24μs do *perf_events*. No entanto, a maior parte (81200μs) está concentrada na inicialização da biblioteca, que ocorre apenas uma vez por execução do programa.

Palavras-Chave: Processadores RISC-V, Monitorização de Desempenho, Amostragem Precisa de Eventos, PAPI

Contents

List of Figures

List of Tables

Listings

Glossary

CSR	Control Status Register
EPI	European Processor Initiative
EU	European Union
HPC	High Performance Computing
HPM	Hardware Performace Monitor
ISA	Instruction Set Architecture
PMC	Perfomance Monitoring Counter
PMU	Performance Monitoring Unit
PLIC	Platform-Level Interrupt Controller
PES	Precise Event Sampling
SMP	Symmetric Multiprocessing
SoC	System on Chip
WARL	Write Any Read Legal

CHAPTER 1

Introduction

In recent decades, microprocessors and computing technologies have found a way into almost all aspects of both our personal and collective lives. They have allowed humanity to develop in most areas of knowledge and given us the tools to enhance our standard of living to previously unimaginable heights. For example, areas like autonomous systems, medicine research and climate modeling have substantially benefited from the advancements in computing technology. In more recent times, the pursuit of High Performance Computing (HPC) in conjunction with data storage improvements allowed for a proliferation of cloud, data science and machine learning applications that together have created a platform in which even more scientific work can be developed.

For this trend to continue, all of the aforementioned technologies must be optimized to their greatest potential. The performance of computing technologies is evaluated not only purely in calculations per second, but also in how energetically efficient they are, being that the latter is recognized as the main challenge to achieve the milestone of exascale computing, or in other terms, executing 10^{18} calculations per second. Even though this barrier has been crossed, there is much to be done in terms of efficiency.

The European Commission has acknowledged the strategic benefits of energetically efficient HPC by supporting the creation of the EuroHPC Joint Undertaking, an entity that has the objective of pooling European resources to buy, build and deploy supercomputers in the European Union (EU). EuroHPC is trying to achieve this by doing two things: firstly, creating a pan-European supercomputing infrastructure; and secondly, supporting the research and innovation of microprocessor and computing technologies inside the EU. The latter culminated in the formation of the European Processor Initiative (EPI), an initiative to develop a high-end, RISC-V-based microprocessor. The ISA represents the most widely adopted for open-source architecture projects and has gained substantial support not only from academia but also from emerging commercial vendors.

To achieve a goal such as exascale computing or even beyond that, programmers must be able to utilize as efficiently as possible all of the available hardware resources. This is a significant challenge, as it is generally hard to pinpoint the cause of a slowdown given the ever-increasing complexity of the microprocessor's architecture. In particular, this often requires the application of profiling techniques to correctly identify the bottleneck. On the other hand, during architecture codesign, this may be useful to optimize the architecture to the target application domains.

There are several ways to profile the performance of a system when running an application. One can use simple profiling techniques, like counting the time elapsed, but even if the programmer manages to identify the part of the source code that generates the unexpected slowdown, this technique does not explain why the slowdown occurred. If a well-tailored model of the processor exists, a simulator can be used to go over the execution and profile the usage of each component of the system, but this is often not the case, either due to commercial constraints or intellectual property limitations. Moreover, there are always modeling limitations that may impose a bias in the observation of performance bottlenecks.

Another way would be to utilize an Hardware Performace Monitor (HPM), usually packaged inside modern microprocessors. These devices can count a set of fixed and/or programmable events that occur inside the processor's pipeline (like instruction-cache misses or branches taken). With this information, a better image of the architecture components responsible for the performance bottlenecks can be observed.

This, in turn, allows tuning the application to account for the limitations found, or the design of the hardware to mitigate the observed problem.

Additionally, larger microprocessor manufacturers such as Intel or Arm offer catch-all tools that combine HPM event sampling with other techniques to aid the user with identifying which parts of their software need more attention. Again, open-source architectures based on the RISC-V ISA rarely, if ever, have this level of support from their manufacturers.

Although the RISC-V Foundation specifies that performance counters and a HPM must be packaged into a RISC-V core implementation [26], its open-source nature provides limited software support for accessing hardware performance counters. The official Linux kernel only supports the sampling of the cycles and instructions retired counters, leaving a lot of functionality out of bounds for anyone who desires to run Linux on their RISC-V processor.

This gap started to be bridged when support for the latest RISC-V HPM specification was introduced in the Linux Perf Kernel Driver [3], but there is yet no way to easily access the performance counters through a higher level interface library, like *PAPI*, and no further functionality is offered to the user besides manually specifying what events should be counted and reading them explicitly.

These are clear disadvantages when comparing RISC-V to any other already established architecture and may prevent the porting of user-level applications to the platform by setting a high barrier to entry for application developers who wish to utilize the available hardware to its full potential.

1.1 Objectives

It has become clear that, with today's computing technology, to achieve higher productivity and performance it is pivotal that great attention is given to process optimization, even if that means tailoring an application to a specific architecture to take advantage of its specificities or, if early enough in the architecture co-design phase, tailoring the architecture to the target application.

It is also evident that although support for the profiling tool *perf_events* is available for RISC-V systems, no higher-level library is available to easily access the Hardware Performance Counters inside the application source code, as is usual to be done on other popular architectures.

Furthermore, the RISC-V specification for the HPM is still very simple when compared with the feature-rich implementations of architectures such as *x86* or *ARM*.

To help more efficient RISC-V software development and facilitate architecture codesign, this work is defined by two objectives. The first is to port an open-source and commonly used C library that offers higher-level access to the HPM. The second is to propose a new hardware/software system that offers more in-depth detail of the processor's state when profiling an application and presents a proof-of-concept implementation of it.

1.2 Contributions

This work hopes to offer to the RISC-V community the following contributions:

- Port the *PAPI* library to RISC-V while providing support for the SiFive U74-MC processor;
- Explanation of what steps need to be replicated or changed if *PAPI* support for new RISC-V processors is desired;
- Proposal of a Precise Event Sampling (PES) facility inside of RISC-V's HPM that could be integrated into *PAPI*;
- Minimum viable implementation of a PES facility on a CVA6 processor.

1.3 Document Outline

This dissertation will continue with the background and state-of-the-art in Chapter 2, where the concepts and background knowledge necessary to understand the work developed are presented, as well as modern uses and techniques for HPMs. Then, in Chapter 3, the changes necessary to port the *PAPI* library to RISC-V are discussed and the results of the implementation are presented and analyzed. After, Chapter 4 goes over the design, implementation, results and shortcomings of a proof-of-concept PES facility for RISC-V processors. Finally, Chapter 5 wraps up the document with the achieved contributions and future work.

CHAPTER 2

Background and State-Of-The-Art

Following the objectives exposed in Section 1.1, the bulk of this work will revolve around HPM implementations on RISC-V processors and how the information obtained through them can be used to aid the development of more efficient software that fully utilizes the resources available to it.

Hence, this chapter describes the current state-of-the-art uses of hardware performance counters and the background to understand how they are defined in the context of RISC-V, how they are implemented, and modern tools to access them when using a Linux-based operating system.

In practice, Sections 2.1, 2.2 and 2.3 focus on the RISC-V definition for HPMs, the SiFive U7 implementation of HPM and the OpenHardware Group CVA6 implementation of HPM, respectively. Section 2.4 reviews some current developments on how Hardware Performance Counters are being improved. Sections 2.5 and 2.6 go over modern software that simplifies access to the HPM facilities in a processor. Section 2.7 explains how PES can be used to generate more useful information by studying how Intel PEBS and AMD IBS work.

2.1 RISC-V ISA

RISC-V [26], [27] is an open source ISA that was designed to support computer architecture research and education and also to become a free and open standard for industry implementations. The RISC-V ISA is purposely defined without implementation details to guarantee freedom for the vendor to adapt an architecture implementation to its needs.

This subsection will start with an overview of what the base RISC-V integer ISA is and the standard extensions available and defined by the RISC-V Foundation. Then, the specification given for a RISC-V HPM in [27] is run through.

2.1.1 Base Integer ISA and Extensions

A RISC-V ISA is defined as a base integer ISA, that is required for all implementations, in conjunction with optional extensions. The base integer ISA is meant to offer sufficient functionality for custom accelerators and educational purposes. Together with standard extensions, it can become an ISA suitable for more advanced computing and even general-purpose and high-performance computing. For this reason, and although RISC-V is usually referred to as a single ISA, it is a family of related ISAs. RISC-V ISAs are named based on a code that identifies the specific base ISA: "RV" followed by a number that indicates the length of the address space and either the letter "I", which indicates the existence of the standard number of 32 general purpose registers, or the letter "E", that stands for embedded and means that the only 16 general purpose registers exist. Additionally, a variety of letters can be concatenated to the end of the base name to indicate the ISA extensions present in the specific implementation. The address space length is also referred to as MXLEN (i.e., an RV64 system has MXLEN=64 and an RV32 system has MXLEN=32).

2.1.2 Hardware Performance Monitor

As mentioned in Section 2.1, the RISC-V specification does not provide implementation details. With this in mind, the present subsection goes over the general definition for a RISC-V HPM as described in [27]. Later sections will go over specific implementations of HPMs for the cores used in the development and testing of this thesis.

The HPM specification was first introduced in version 1.7 of the privileged specification and had the most recent change introduced in version 1.11 of the same document.

Fixed And Programmable Event Counters

The counters available in RISC-V HPM are: `mcycle` that counts the number of executed cycles in the core, `minstret` that counts the number of retired instructions and an additional 29 programmable event counters `mhpmcounter3-mhpmcounter31`. All of these counters are machine-level read and write Control Status Registers (CSRs) and have 64-bit precision in both RV64 and RV32 systems.

Event selector CSRs `mhpmevent3-mhpmevent31` are MXLEN-bit wide registers that are used to control which event is being counted on the corresponding counter (e.g., `mhpmevent3` controls the event that causes `mhpmcounter3` to increment). The meaning of the programmable events and their encoding should be defined by the vendor of the specific implementation except event 0 which always means "no event". The specification indicates that all 29 programmable counters and their respective event selectors should be implemented while also stating a valid implementation is to make unused programmable counters and their respective selectors read-only 0.

As previously stated, the programmable event counters always have 64-bit precision. To comply with this in RV32 access to 64-bit registers is made in two steps: reads/writes of `mcycle`, `minstret` and `mhpmevent3-mhpmevent31` return/change only bits 31-0 and reads/writes of `mcycleh`, `minstreth` and `mhpmevent3h-mhpmevent31h` return/change bits 63-32.

Counter Availability To Lower Privilege Levels

The `mcycle`, `minstret` and `mhpmevent3-mhpmevent31` registers are machine-level read and write CSRs, meaning reads/writes from lower privilege levels will trigger an illegal instruction exception. The `cycle`, `instret` and `hpmevent3-hpmevent31` CSRs are read-only shadows of `mcycle`, `minstret` and `mhpmevent3-mhpmevent31`. Their availability to the next implemented privilege level is controlled through the `mcounteren` CSR. The `mcounteren` is a 32-bit register and is organized in a one-hot encoding of all hardware performance counters plus the memory-mapped `mtime` register, as shown in Figure 2.1. As an example, when bit 0 is clear, reads of `cycle` from a privileged mode lower than machine mode will generate an illegal instruction exception. When bit 0 is set, access for reads is permitted to the next implemented privilege mode (i.e., supervisor-mode if implemented or user-mode otherwise).

The `mcounteren` must always be implemented if the system implements user-mode but it may be set to read-only 0.

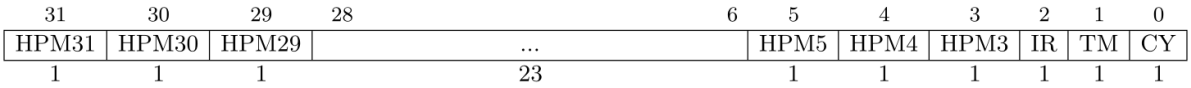

Figure 2.1: Counter-enable register `mcounteren`, [27].

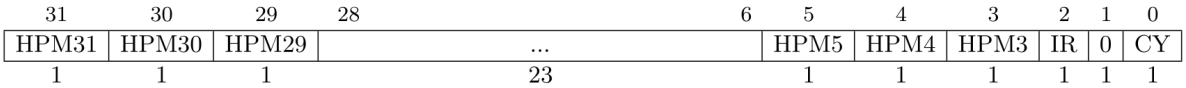

Figure 2.2: Counter-inhibit register `mcountinhibit`, [27].

Counter Inhibition

The counter-inhibit register `mcountinhibit` is a 32-bit register that controls whether a hardware performance counter increments when the programmed event occurs or not. It is implemented using a one-hot encoding of the available counters, as shown in Figure 2.2. As an example, when bit 0 of `mcountinhibit` is clear, the `mcycle` register will increment normally. When bit 0 of `mcountinhibit` is set, `mcycle` register will not increment.

The `mcountinhibit` register can not be implemented and, in that case, the implementation should behave as if the register were set to read-only 0.

Overflow Or Threshold Handling

At the time of writing, no specification for a mechanism that generates an interrupt when an HPM counter overflows or reaches a previously chosen threshold exists.

2.2 SiFive U7 Core

The U7 core was designed by SiFive and is included in the U74-MC core complex [23]. A block diagram of U74-MC is presented in Figure 2.3. Its ISA is RV64IMAFDC, which means, a base RISC-V integer ISA with a 64bit address space that also has the standard extensions: "M" (integer multiplication), "A" (atomic instructions), "F" (single-precision floating-point), "D" (double-precision floating-point) and "C" (compressed instructions). The standard instructions "M", "A", "F" and "D" are often bundled together into the single denomination "G", as they are essential for a general-purpose microprocessor.

The U7 core processor core implements a basic HPM that is divided into two classes of counters: fixed-function counters and event programmable counters.

2.2.1 Fixed-Function Performance Monitoring Counters

The fixed class of counters consists of a set of fixed counters and their corresponding counter-enable registers. As the name indicates, a fixed-function performance monitor counter is hardwired to one specific event and that event cannot be changed. The only flexibility provided by this class of counters is the ability to enable or disable the counting through the counter-enable registers and to change the counter value.

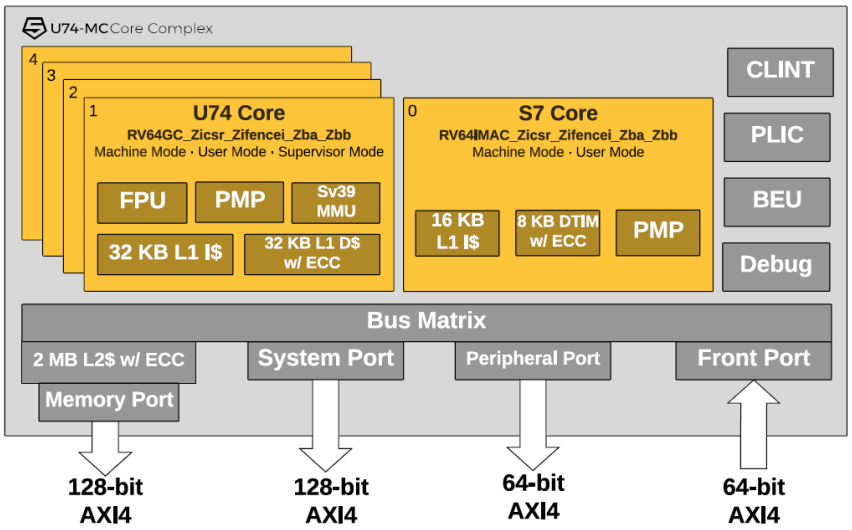

Figure 2.3: U74-MC Series Block Diagram [23]

The following fixed-function counters are provided in the U74-MC core:

`mcycle`

The fixed-function performance monitoring counter `mcycle` holds the count for the number of clock cycles executed since an arbitrary time in the past. It is 64 bits wide and reading the CSR will return all 64 bits.

`minstret`

Likewise, the fixed-function performance monitoring counter `minstret` holds the count of the number of instructions retired since an arbitrary time in the past. The counter is 64 bits wide and reading the CSR will return all 64 bits.

This implementation of the fixed-function performance monitor counters corresponds with the requirements defined by the RISC-V Foundation in [27].

2.2.2 Event-Programmable Performance Monitoring Counters

The programmable class of counters is composed of event-programmable counters and event-selector registers. The U7 HPM boasts two of these counters: `mhpmcounter3` and `mhpmcounter4`. They are implemented as 40-bit counters and can be written to initialize their count value. To control the event being counted by the programmable counters, two event-selector CSRs are provided: `mhpmevent3` and `mhpmevent4`. These CSRs are 64-bit wide and are partitioned into three fields (as shown in Figure 2.4), with the 8 least significant bits representing the event class; bits 8 through 55 forming a mask for the events in said class; bits 56 and 57 are reserved and do not currently have an associated functionality; bits 58 through 62 inhibit the count of events when the core is in the associated privilege level (refer to Table 2.1 for specific privilege level related to each bit), which means that if all are set to zero, the count

Machine Hardware Performance Monitor Event Register			
Bits	**Name**	**Attr**	**Description**
[7:0]	Class	WARL	Selects the Event Class to make available for counting
[55:8]	EventSel	WARL	Bit-mask of Event(s) to count
[57:56]	Reserved	-	
58	VUINH	WARL	Reserved
59	VSINH	WARL	Reserved
60	UINH	WARL	If set, counting of events in U-mode is inhibited
61	SINH	WARL	If set, counting of events in S-mode is inhibited
62	MINH	RW	If set, then counting of events in M-mode is inhibited
63	OF	RW	Overflow status and interrupt disable bit. Set by hardware when counter overflows.

Table 2.1: `mhpmeventX` Register bit layout for overflow and filtering [23].

occurs in all privilege levels; and bit 63 indicates whether the associated counter as overflown and in such case remains set until it is written by software.

According to the RISC-V Foundation [27], the programmable counters `mhpmcounter5` through `mhpmcounter31` must also physically exist in the processor but is a legal implementation to have both the unused counters and their corresponding event selector CSR be read-only zero registers.

In Section 2.2.3 an explanation of how the events are encoded into the `mhpmeventX` CSR is presented and an example is given.

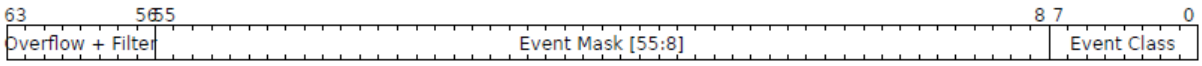

Figure 2.4: Event selector fields [23].

2.2.3 Event Selector Encodings

On the U7 core, events are categorized into classes, which are selected by setting the correct bits in the `mhpmeventX` CSR. Specific events are selected by setting the corresponding bit on the Event Mask field of `mhpmeventX`. Multiple events from a given class can have their corresponding bit set on a given `mhpmeventX`. In that case, the count is incremented every time any of the selected events occurs. If all the bits in the Event Mask are set to 0, nothing is counted.

All the events available to program the two programmable counters, their class and their Event Mask encoding are present in Table 2.2. In the implementation of the U7 core used during the development of this work, setting a bit in the Event Mask that is not defined in the specified Event Class does not affect the count, but is discouraged as it might make software incompatible with future implementations that may expand the list of available events.

As a quick example: if the `mhpmevent3` CSR is set to `0x4200` (activating solely bits 9 and 14), it can be verified by referencing Table 2.2 that `mhpmcounter3` will increment every time either the events "Integer load instruction retired" or "Conditional branch retired" occur.

Machine Hardware Performance Monitor Event Register	
Instruction Commit Events, `mhpmeventX[7:0]=0x0`	
Bits	**Description**
8	Exception taken
9	Integer load instruction retired
10	Integer store instruction retired
11	Atomic memory operation retired
12	System instruction retired
13	Integer arithmetic instruction retired
14	Conditional branch retired
15	JAL instruction retired
16	JALR instruction retired
17	Integer multiplication instruction retired
18	Integer division instruction retired
19	Floating-point load instruction retired
20	Floating-point store instruction retired
21	Floating-point addition retired
22	Floating-point multiplication retired
23	Floating-point fused multiply-add retired
24	Floating-point division or square-root retired
25	Other floating-point instruction retired
Microarchitectural Events, `mhpmeventX[7:0]=0x1`	
Bits	**Description**
8	Address-generation interlock
9	Long-latency interlock
10	CSR read interlock
11	Instruction cache/ITIM busy
12	Data cache/DTIM busy
13	Branch direction misprediction
14	Branch/jump target misprediction
15	Pipeline flush from CSR write
16	Pipeline flush from other event
17	Integer multiplication interlock
18	Floating-point interlock
Memory System Events, `mhpmeventX[7:0]=0x2`	
Bits	**Description**
8	Instruction cache miss
9	Data cache miss or memory-mapped I/O access
10	Data cache write-back
11	Instruction TLB miss
12	Data TLB miss
13	UTLB miss

Table 2.2: `mhpmevent` Register on SiFive U7 [23].

Machine Hardware Performance Monitor Event Register	
Value	**Description**
0x1	L1 I-Cache misses
0x2	IL1 D-Cache misses
0x3	ITLB misses
0x4	DTLB misses
0x5	Load accesses
0x6	Store accesses
0x7	Exceptions
0x8	Exception handler returns
0x9	Branch instructions
0xA	Branch mispredicts
0xB	Branch exceptions
0xC	Call
0xD	Return
0xE	MSB full
0xF	Instruction fetch empty
0x10	L1 I-Cache accesses
0x11	L1 D-Cache accesses
0x12	Eviction
0x13	I-TLB flush
0x14	Integer instructions
0x15	Floating point instructions
0x16	Pipeline bubbles/Stall

Table 2.3: `mhpmevent` Register on CVA6.

2.3 CVA6

Designed to provide insight into the energy cost and tradeoffs associated with projecting a RISC-V core with support for fully-fledged operating systems [33], CVA6 (previously known as Ariane), is a 6-stage, single-issue, in-order CPU that implements an RV64IMAFDC ISA. It supports the M, S and U RISC-V privilege modes, which allows compatibility with Unix-like operating systems.

2.3.1 Performance Monitoring Unit

Similarly to the implementation presented for the SiFive U7 core (Section 2.2), the CVA6 core implements the `mcycle` and `minstret` fixed counters, following the RISC-V privileged specification [27].

CVA6 implements 6 programmable counters, all of which can count all events but without the ability to multiplex the counters to more than 1 event at a time. The `mhpmcounterX` counters are programmed by setting the corresponding `mhpmeventX` CSR to the mask that encodes the desired event, with `mhpmeventX` set to `0` meaning no event is to be counted. All available events and their encoding are provided in Table 2.3.

Additionally, the HPM implementation includes the `mcountinhibit` CSR to allow for the counting to be halted in a specific counter, also following the most recent RISC-V privileged specification.

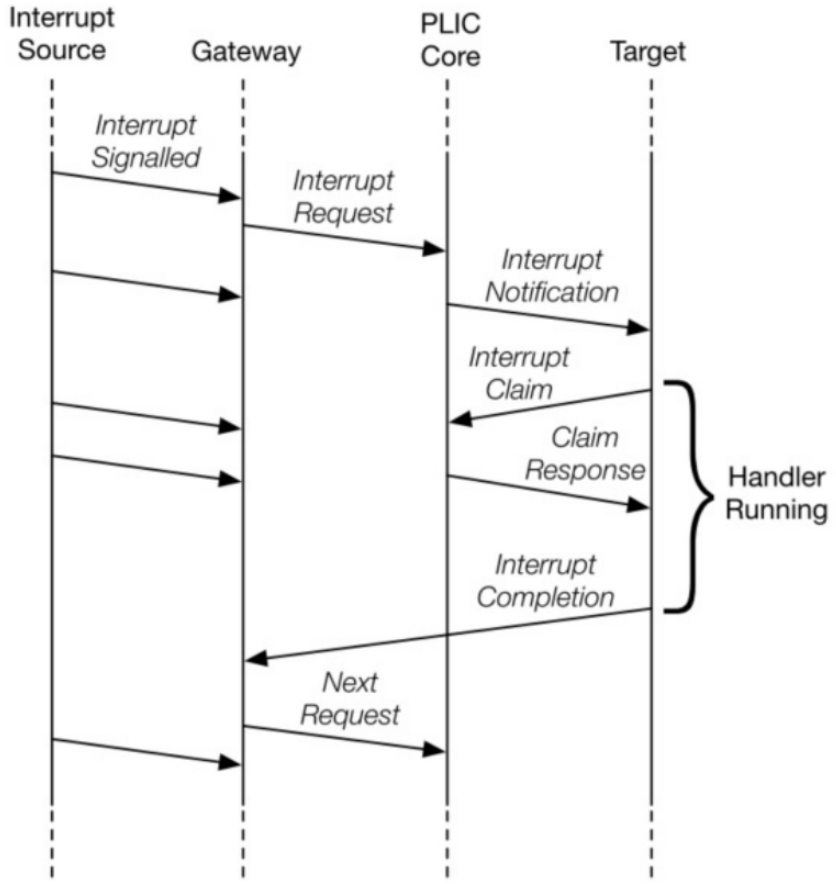

Figure 2.5: PLIC communication flow, [10]

2.3.2 Platform-Level Interrupt Controller

The PLIC included in the CVA6 processor follows the RISC-V International specification, [10], and is used in RISC-V computers to generate external interrupts that the cores can then handle.

The working of the PLIC can be divided into 4 steps. First, an interrupt source (like a keyboard or a signal coming from the core itself) activates letting the PLIC gateway know that the source device needs the core to handle something. The gateway then sends an interrupt request to the PLIC core, and if the Interrupt Enable for that source is set, the target for that external interrupt is not pending the resolution of a previous interrupt and the interrupt priority for the desired interrupt overcomes the currently defined priority threshold, the PLIC core sends an interrupt pending signal to the correct privilege mode of the target hart, or what the specification calls a context.

When the RISC-V core is notified that it has an interrupt pending it should read a specified memory location (as an example, the M mode of Hart 0 should write to the memory location `base + 0x200004`, with base meaning the PLIC base memory address which in CVA6 is `0xC000000`). When the core finishes the execution of the interrupt handler, it should write the value it read before to that same memory address to let the PLIC know the handler has completed and the context is ready to receive external interrupts. The specification mentions that the value written is not verified so although the value written should be the same it will work even if it is not. This communication flow is illustrated in Figure 2.5.

2.4 Modern development on Hardware Performance Counters

2.4.1 Hardware Multiplexing

Due to the limited number of physical counters usually available in comparison with the number of selectable events to count on a processor (as an example, the SiFive U7 can count 35 different programmable events but only offers 2 programmable counters), in 2001 [15] proposed a novel way to count more than one event at a time in one programmable counter. The *MPX* method, as it was called, proposed that if the occurrence of an event e during a certain time T was fairly constant, the time T could be divided into equally long slots in which more than one event would be counted in round-robin fashion. The actual amount of times that e occurred would be approximately equal to the counted amount of e multiplied by the ratio between the total time T and the time event e was being counted. Or, in other words, *MXP* proposed that Hardware Performance Counters should be time-multiplexed. The results shown in the paper were positive, managing to have their prediction for four concurrent events stay within 5% of measurements done without multiplexing, in most cases. However, due to the inherent memory inefficiency of the software that controls *MXP*, demonstrated inaccuracies reach as high as 70% and there is no bound for how inaccurate a count can be due to the unknown consistency of the event occurrence.

In order to try and overcome these limitations, several solutions were proposed over the years and, in 2021, [32] suggested a paradigm shift from deterministic counters to an architecture based on approximate counting algorithms. This architectural philosophy allows for great reductions of memory usage, which was the *MXP*s Achilles' heel. As such, hardware events related to memory are more accurate when compared to the other method of time multiplexation. Also, the approximate algorithms counter approach has a theoretical maximum for relative error: 89%.

2.5 Perf_events

As previously discussed, most modern processors include in their architecture some implementation of the concept of hardware performance monitoring through event counters. Often, direct access to these counters is restricted to code executing at the supervisor-level. To overcome this restriction, libraries that provide an interface between the user and the operating system kernel started emerging, like *Oprofile*, *Perfctr* or *Perfmon2*.

Until 2009, the Linux kernel did not ship with integrated support for performance counter access and all existing implementations had to be patched into the kernel. With the release of Linux kernel version 2.6.31, the *perf_events* subsystem was introduced [30] as the default interface for accessing hardware performance counters. It was designed with functionality and abstraction in mind to make it simple to operate.

Unlike previously available interfaces, that used pseudo-filesystems or emulated devices to access the performance counters, *perf_events* uses the `perf_event_open()` [11] system call to allocate file descriptors with the events to be counted specified at the open time in the fields of the `perf_event_attr`

[11] structure and counters can be enabled and disabled with `ioctl()` or `prctl()` system calls. As the communication with the kernel is done through file descriptors, a `read()` system call is used to read the values counted.

The `perf_event_attr` has a high number of variables and not all of them are relevant to this work. The ones that deserve the most attention in this context are:

type

This variable specifies the general event type to be configured. It can take predefined values such as `PERF_TYPE_HARDWARE` if it is a general event provided by the kernel or `PERF_TYPE_RAW` if the user intends to use an implementation-specific event to configure the hardware performance counter.

config

This variable specifies the actual event to be set. It works following the value set in the *type* variable to determine the event. Following the previous examples, if *type* is `PERF_TYPE_HARDWARE`, the user can select `PERF_COUNT_HW_CPU_CYCLES` to select an event that counts executed processor cycles for example; if *type* is `PERF_TYPE_RAW`, this variable needs to be set to a specific value determined by the vendor. The *libpfm4* library can be used to translate between event names and their hexadecimal encoding; more in Section 2.6.2.

The front-end of the *Perf_events* subsystem is the *perf* [7] profiler. *perf* is especially well suited to conduct microprocessor analysis as it is capable of profiling processor stack traces, tracing CPU scheduler behaviors and probe performance monitoring counters.

In the context of this work, *Perf_events* is used to interface with the performance monitoring counters of RISC-V processors. Every specific implementation of a processor has its configuration codes to indicate what hardware events should be counted by the programmable performance monitoring counter, like the codes discussed in Section 2.2.3. *perf* does not expect the user to know every single code a processor might have listed in its manual and provides human-readable mappings for supported processor implementations. As an example, instead of the user having to remember the code for counting exceptions taken on the SiFive U74-MC is `(0x0100)`, they simply have to remember the event name "exception_taken".

Full support for every event mappable in the SiFive U7 processor core is available on *Perf_events* [3], a work previously developed within EPI.

2.6 Performance API - PAPI

Even when most processor platforms started adopting hardware performance monitoring counters in the late 1990s, access to them was still poorly documented, unstable and not available to user-level programs. These conditions made it difficult for performance tool developers to implement their solutions and for users to easily use such tools or to monitor and profile the processor usage of their applications.

PAPI [1], [5], [25], [31] was created with the main focus of providing an easy-to-use set of interfaces that could gain access to the hardware performance counters of the major processor platforms. This could free developers of the unnecessary difficulties of obtaining such information to tune and optimize their software, run performance analysis or model the processor usage of their implemented solutions.

PAPI provides two interfaces to the hardware counters. One is a simple and high-level interface to acquire simple measurements and another is low-level and allows for more refined functionality. The high-level interface only allows the user to start, stop and read counters for a specified list of events. On the low-level interface, the user can manually manage the hardware event groups to be measured; which are called *EventSets*.

PAPI was also implemented with portability in mind. When a program is written using *PAPI* functions to profile its performance, that code should not have to be altered if it is to be used in another computer for which the architecture is supported by *PAPI* and it is installed.

2.6.1 Original Design

In this subsection, a brief discussion of how *PAPI* works is presented.

Layers

Figure 2.6 is a simple representation of *PAPI* layered approach. *PAPI* is internally split into two parts: a portable part and a machine-dependant part.

The portable part contains both the high and low-level interfaces and is independent of the architecture it is running on, so it does not require much work when porting *PAPI* to new implementations of already supported architectures. All the API functions and utilities that manage state handling, memory management, data structure manipulation and thread safety are defined here.

The machine-dependent code composes the internal *PAPI* layer and is denominated substrate. The portable layer calls function defined on the substrate as a means to access the hardware counters. Having its interface and functionality well defined but no implementation method specifics, the substrate can be easily updated and changed with no great incompatibilities. Any architecture/operating system combination needs only one new substrate layer to provide *PAPI* support.

Portability

The *PAPI* API allows source code portability through a common interface, but it does not solve any problems related to decoding the machine-specific settings for accessing HPMs. To address this, *PAPI* defines a list of predefined events, or presets, that represent most of the events that are commonly used in a modern processor. Some of these events are, for example, total cycles, total instructions used, total branch instructions completed or floating-point instructions completed per second.

A *PAPI* substrate should implement as many presets as possible without providing misleading or wrong results. This simplification could potentially confuse if the same preset event is compared between different systems which may, of course, have distinct implementations for counting the specified event. As such, directly comparing two systems is not the intention of *PAPI* providing presets, but rather to introduce

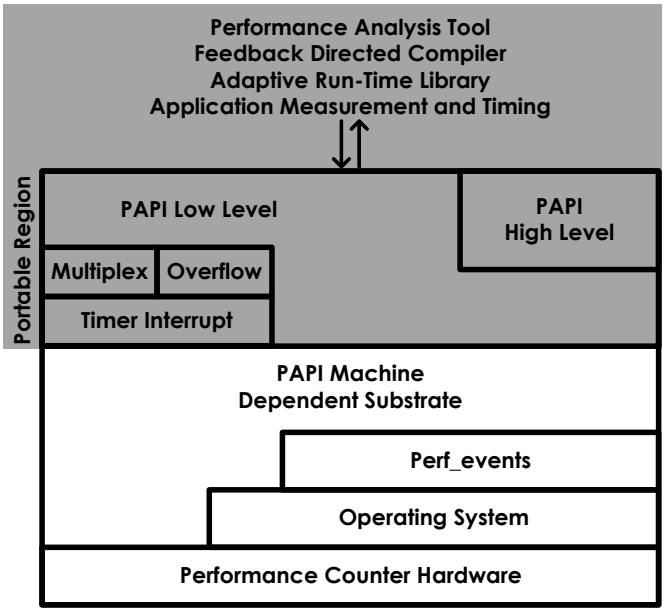

Figure 2.6: PAPI architecture, modified from [1].

standard names for the metrics available. Because of this, it is still necessary for the user to possess a working knowledge of the system that they are testing to be able to correctly interpret the data collected with *PAPI*.

EventSets

EventSets are used in *PAPI* as an abstraction from singular hardware events and are composed of events the user wishes to count as a group. Two major reasons are presented to follow this approach, one technical and one practical.

The first one is that aggregating events together increases efficiency when accessing counters through the operating system. Since most operating systems support using a single system call to move the value of more than one counter, grouping events greatly reduces the overhead of multiple system calls. This is especially important when *PAPI* is used to monitor small regions of code inside loops that iterate many times.

The second reason for the existence of EventSets is to allow users to create their counter groupings, catered to their specific application areas. The necessity for such groupings arises from the fact that in some instances, a singular event count is not enough to explain the performance of a certain region of code. Commonly, the more relevant information is only attainable by relating different metrics.

When using *PAPI*, the user is allowed to create as many EventSets as needed. Providing the substrate can provide the necessary resources. They are also allowed to use them simultaneously and share counters between them. If more events are added to the EventSet than are simultaneously countable by the HPM and the user has not explicitly enabled software multiplexing, an appropriate error is returned. An error is also returned if the user tries to use an EventSet improperly.

Multiplexing

Although most modern processor architectures provide hardware performance counters, usually only a very small number of them are present, and consequently, the amount of events that can be counted at the same time is also low. This constraint means that the quantity of information acquirable in each run is sharply diminished and, in the eventuality that the application under study takes a long time to execute, the necessity to obtain various metrics might compound the run time of the test into several hours, days, or even weeks.

To avoid such limitation, *PAPI* subdivides the usage of counting hardware over time, or what is called multiplexing. By using multiplexing the user perceives that more hardware events are countable at the same time.

Although multiplexing incurs a small overhead and negatively affects the counting accuracy, the advantage of monitoring several metrics concurrently vastly outweighs the disadvantages. Even so, to prevent the user from unwittingly using multiplexing without being aware of the loss of precision, in both the low and high-level interfaces, it has to be activated manually with a specific API call.

Threshold Handling

PAPI provides the user with the possibility to define handlers for when a specific hardware event surpasses a predetermined threshold. This functionality is implemented by using a high-resolution interval timer and setting a timer interrupt handler. If the system does not support counter overflow, *SIGPROF* and *ITIMER_PROF* are used instead.

The handler is called from the signal context bundled with some arguments whenever the counter value is greater than the defined threshold. The user can then use the arguments given to determine the event that overflowed, by how much and where in the source code it happened.

PAPI uses the same functionality to handle counter value overflow.

Statistical Profiling

The functionality of constructing statistical profiling of where a certain event count overflows is also offered by *PAPI*. It uses the method described in Section 2.6.1 to detect when the event surpasses the threshold and receives a signal that contains some arguments, namely: the stack pointer and program counter. Then it uses some underlying performance tool to get the address at which the program was interrupted and hash it into a histogram. At the end of the program execution, the user is given a line-by-line analysis of where the counter overflow happened. This study can be done with any implemented event.

To create a histogram of the type described, the user can use the *PAPI_profile()* call.

Thread Support

Since Symmetric Multiprocessing (SMP) was already popular in HPC applications by the time *PAPI* was created, thread awareness was part of its implementation since the beginning.

Being thread-aware implies that every globally writeable variable or structure is locked before it is modified and unlocked afterward. *PAPI* only has one global data structure to store process-wide options and thread-specific pointer maps. As this structure is only modified by two API calls and mostly at initialization or termination time of the *PAPI* library, assuring thread awareness does not represent a great overhead drawback when accessing hardware performance counters.

Another difficulty relates to the accuracy of the counters as, to be thread-aware, the operating system has to save and restore the HPM counters when context switching between active threads and processes. *PAPI* must also keep copies of every thread's counter data structure and values. *PAPI*s thread-awareness functionalities are only activated if more than one thread initializes the *PAPI* library or executes a *PAPI* API call.

Furthermore, some main-stream threading APIs don't explicitly present the concept of user-level or kernel-level threads to the user, like *Pthreads* and *OpenMPI*. In the case user-level threads are used, the values read are likely inaccurate and so, the user must explicitly bind the threads to kernel-level. In the HPC community this should not be of much concern as kernel-level threads are the standard.

Counter Accuracy

When *PAPI* was designed, it attempted to keep the overhead as small as possible to maintain disturbance to the performance analysis to a minimum. Even then, it is impossible to not contaminate the counter values at all and some inaccuracy can still occur due to other programs contending for the resources of the system or even because of the operating system background tasks.

In the original paper [1], the authors of *PAPI* did not provide any study on the accuracy of hardware performance counters or the *PAPI* library itself but [5] briefly refers to [12], where it is concluded that some accuracy problems when using hardware performance counters are prevalent when the granularity is used is too small to guaranty that the overhead of counting the events is not prevalent in the said count. The same study also found that the accuracy when attributing a hardware event to a specific instruction in out-of-order architectures is problematic.

In [30], it was shown that *PAPI* had a combined start, stop and read performance overhead of around 14000 cycles on an Intel Core 2 system, but also shows that further optimization is possible.

2.6.2 *libpfm4*

As mentioned in Section 2.6.1, the implementation of the substrate was not defined in the original *PAPI* specification and over the years, several approaches to access the hardware performance counters have been used. Since *PAPI* version 4, *libpfm4* has been used to communicate with *perf*, [28].

libpfm4 [6] is a library used to develop monitoring tools that make use of performance monitoring events. This library can be used to convert an event name, given in a human-readable string, to a raw hardware event encoding compliant with the vendor specifications or to an OS-specific encoding, in which case it prepares the adequate data structure that the kernel requires. In the case of modern versions of *PAPI*, the events are set up through the kernel tool *perf*.

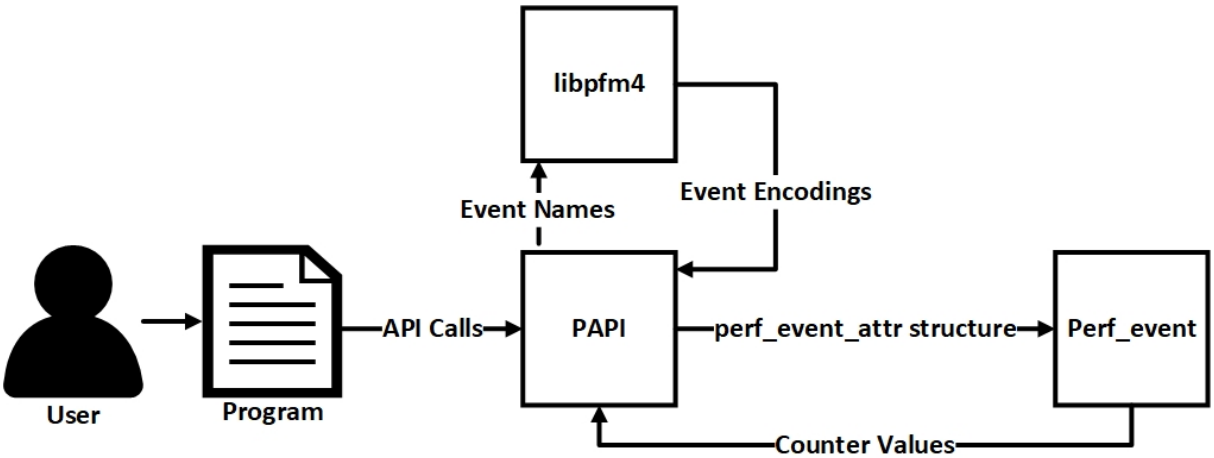

Figure 2.7: Basic structure for hardware performance counting using PAPI.

The data structure *perf* uses to interface with programs is the `perf_event_attr` [11] mentioned in Section 2.5.

2.6.3 How PAPI Reads Event Count Values

Summarizing all information in Section 2.6, what follows is a very brief explanation of how *PAPI* can be used to read hardware performance monitoring counter values. Figure 2.7 provides a visual representation of it.

The user writes a program and identifies what events they want to count to optimize or profile their software. They initialize the *PAPI* library and make the necessary API calls using event names in the form of human-readable strings. *PAPI*, in its most recent substrate version, inputs the Event name and machine-specific information to libpfm4, which translates this information into a `perf_event_attr` structure *perf* can use and gives this structure back to *PAPI*. The structure is then sent as an argument in a system call to the kernel's *perf_events* subsystem. The return is a file descriptor *PAPI* utilizes to start, stop and read the counter values.

2.7 Precise Event Sampling

PES is a robust profiling technique supported by most modern processors HPMs that can sample hardware events and locate the instructions that trigger said events, [21]. It has been merged into modern software profiling tools to better identify performance bottlenecks and has been shown to enable the detection of inter-thread coherence traffic [20], false sharing [14], long latency remote memory accesses in NUMA multicore systems [13], data locality problems [22], performance degrading bandwidth consumption [8] and conflict cache misses [19]. Because this kind of sampling allows for the analysis of instruction pointers and addresses of data being operated, fully fleshed-out implementations of PES can help pinpoint bottlenecking instructions or data objects. Additionally, these tools offer lower time and memory overheads in comparison with options like cycle-accurate hardware simulators and it achieves this by sampling hardware information directly through implemented specialized hardware without adding much software

control.

This technology is already supported by major players like Intel *PEBS*, AMD *IBS*, IBM *MRK* and ARM *SPE*. They are not implemented in precisely the same way but all share similar characteristics, like the existence of specialized hardware on the HPM to enable PES. Currently, no RISC-V implementation has been presented in detail, having only been mentioned briefly in [2] as a means to other HPM uses.

In this section, an in-depth breakdown of Intel's *PEBS* is presented. This is the PES implementation presented because the proof-of-concept presented in this thesis for a RISC-V PES facility more closely follows Intel *PEBS* than the other mentioned implementations.

2.7.1 Intel *PEBS*

In Intel processors, since the Nehalem architecture, PES facilities have been present on Performance Monitoring Unit (PMUs) and can use all the programmable event counters available in them [9]. Intel PMU is governed by what are called global control registers. They can enable and disable both the event counters themselves (similar to the `mcounteren` CSR in RISC-V) as well as the PES functionality of each counter. To use PES, the global control registers are set so a programmable counter and its PES functionality are enabled, the corresponding event select register is programmed with the mask of the event the user wants to count and an overflow is defined so that every time the count in the counter reaches the predefined overflow, a sample is gathered. This number of events is also called the sampling period. When an overflow occurs, the PMU is rigged to capture the next occurrence of the event and when that happens, *PEBS* copies the machine state to a *PEBS buffer* in the form of *PEBS records*, what is called sampling. When the amount of *PEBS records* reaches a predefined amount, an interrupt is triggered so that the profiling software can retrieve the recorded information. This process is exemplified in Figure 2.8 for the case where retired load and store instructions are being profiled. The process works as follows: (1) Global control register `IA32_PERF_GLOBAL_CTRL` enables PMC0 and PMC1 by setting its bits corresponding to both counters to one; (2) Global control register `IA32_PEBS_ENABLE` enables *PEBS* in PMC0 and PMC1 by setting the bits corresponding to both counters to one; (3) The event select registers `IA32_PERFEVTSEL0` and `IA32_PERFEVTSEL1` are programmed to make PMC0 and PMC1 count retired loads and retired stores, respectively; (4) The configured Perfomance Monitoring Counters (PMCs) are preloaded with the sampling interval, N, so that they overflow on elapsing N events. During execution, PMC1 counter overflows after N stores occur. Since *PEBS* is armed to trap the next store, PMC 1 is preloaded with N again. When another store occurs after the overflow *PEBS* traps the access and a microcode routine records the machine state in a *PEBS* buffer. If the number of records has reached a specified threshold (1 in this case), an interrupt is triggered, and an interrupt handler transfers the *PEBS* records to user space [21].

2.7.2 Precise Event Sampling Support in PAPI

As was discussed in Section 2.6.1, *PAPI* offers a simple event sampling functionality that records the overflow address and gives the user-defined handler the user thread context so it can also record any

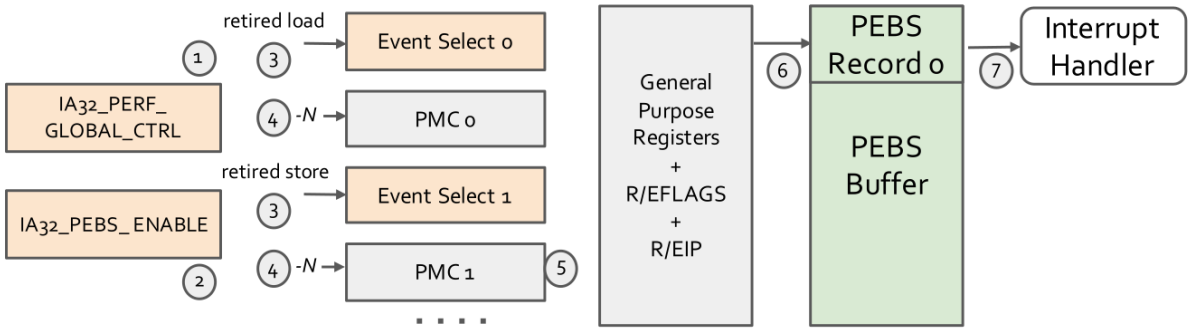

Figure 2.8: One possible execution scenario of Intel *PEBS*. [21]

information accessible through it if it so wishes to.

Nevertheless, [29] identified the problems of this approach and proposed an integration of PES functionalities already present in modern processors, like *PEBS* and *IBS* as a way to solve them.

The paper highlights that the currently present statistical sampling implementation of *PAPI* incurs a greater overhead when compared to hardware/software PES systems because every overflow implies an interruption to store the overflow address and execute the user-defined handler function. The other identified problem is that through the user thread context made available to the user-defined handler, it is impossible to access higher-privileged level information, like kernel- and machine-level register states.

To overcome this issues, new *PAPI* interfaces are proposed, from which the one that would better integrate with *PEBS* would work by setting up existing PES facilities in the processors HPM and define an internal *PAPI* buffer that would be populated from the PES buffer when it was notified by the HPM to do so, via an interrupt. When the *PAPI* buffer fills up, a user-defined handler would be called to process the data or store it in a file.

2.8 Summary

The chapter begins by highlighting the central theme of hardware performance monitoring and PES.

The background section delves into the essential concepts of hardware performance monitoring. It outlines the significance of tracking a processor's performance and the role of event counters in this process. Special attention is given to RISC-V processors and their event counters.

Moving on to *PAPI*, this chapter elucidates its pivotal role as a versatile tool for accessing hardware performance counters. *PAPI* is presented as a valuable solution to the challenge of monitoring and profiling software execution efficiently.

The functionalities of *PAPI* are unveiled through its high-level and low-level interfaces. The high-level interface simplifies the process by allowing users to initiate, terminate, and retrieve counter data for a selected list of events. On the other hand, the low-level interface provides more fine-grained control by enabling users to manage event groups known as EventSets.

A core feature of *PAPI* is its portability, facilitating compatibility across different computer architectures. It ensures that code written with *PAPI* functions remains unaltered when transferred to another system supported by *PAPI*, emphasizing ease of use and cross-platform flexibility.

The chapter touches on the concept of PES, a robust profiling technique. PES has gained prominence due to its ability to capture hardware events and identify the specific instructions or data operations responsible for these events.

Intel's implementation of PES, known as PEBS, is highlighted as an example of this technology. PEBS is integrated into Intel processors and uses the PMU to capture samples when specific event thresholds are reached. This feature provides insight into performance bottlenecks and allows the identification of problematic instructions and data objects.

This chapter also covers *PAPI*'s support for event sampling. *PAPI* provides basic event sampling, but it has limitations, including high overhead and restricted access to privileged information through user-defined handlers. To overcome these issues, [29] proposes integrating PES features like *PEBS* into *PAPI*. This integration would reduce overhead by using existing processor facilities and allow better access to performance data.

CHAPTER 3

Porting PAPI to RISC-V

During this part of the work, a great effort was made to develop the *PAPI* source code in a way that resembles the structure already existing in that project as well as in the *libpfm4* library. This was done for:

1. Possible merging of this work into the main project itself, requiring the least possible changes;
2. Ease the understanding of this work by developers already familiar with the source code of the project;
3. Facilitate the addition of support to more RISC-V processors in the future, following the already well-designed abstraction levels of the *PAPI* project source code.

This decision might have added an unnecessary burden to development in the closed context of achieving results for this thesis but offers a better chance of this work being helpful to anyone who might want to contribute to this port, available publicly at https://github.com/hpc-ulisboa/RISC-V-PAPI.

Getting to understand the underlying workings of this 24-year-old, almost 8500-commit program posed one of the most significant challenges of this thesis. And to overcome it several weeks were spent step-by-step debugging executions of *PAPI* calls and consulting the documentation to understand the data structures and functions that allow *PAPI* to abstract the access to the Hardware Performance Counters. This effort can not be observed directly by the reader but it granted the knowledge necessary to compile the relatively small list of steps that were taken to port *PAPI* to RISC-V and, in particular to the SiFive U7 core, that will be dissected in the following sections of this chapter. But, before that, it would be pertinent to point out that the process of porting *PAPI* to a RISC-V processor can be divided into two distinct parts:

- General RISC-V support - *PAPI* is not yet compatible with RISC-V architectures at all. This means that the substrate (the underlying layer of *PAPI* discussed in Section 2.6.1) must be built upon to add this new architecture to the already large list offered at this time;
- Device-Specific support - After compatibility with RISC-V is achieved, support for a specific existing RISC-V processor is needed in order to verify and test the port. For this new parts need to be added to the substrate as well.

Given this, and taking into account the objectives of this thesis, when discussing the steps required for device-specific, greater attention will be given to how to replicate them, documenting how to expand *PAPI* with more RISC-V implementations.

3.1 General RISC-V Support

The first step to port *PAPI* to RISC-V was to make it compilable on the platform. The already existing *PAPI* source code utilizes preprocessor directives to stop compilation when an architecture-specific definition is missing. So, in order to compile three of these sections had to be defined, namely: (1) a way for *PAPI* to interface with the Linux thread context of the running user-level thread; (2) inline assembly to create a memory barrier; (3) a way to read the cycle counter.

```
...
typedef unsigned long int __riscv_mc_gp_state[32];
...
#define REG_PC 0
...
typedef struct mcontext_t {
    __riscv_mc_gp_state __gregs;
    union  __riscv_mc_fp_state __fpregs;
} mcontext_t;
...
typedef struct ucontext_t {
    unsigned long int  __uc_flags;
    struct ucontext_t *uc_link;
    stack_t            uc_stack;
    sigset_t           uc_sigmask;
    char               __glibc_reserved[1024 / 8 - sizeof (sigset_t)];
    mcontext_t uc_mcontext;
} ucontext_t;
...
```

Listing 3.1: Definitions in ucontext.h.

```
...
#elif defined(__riscv)
#define REG_PC 0
#define OVERFLOW_ADDRESS(ctx) ctx.ucontext->uc_mcontext.__gregs[REG_PC]
...
```

Listing 3.2: Changes to src/linux-context.h.

3.1.1 Thread Context For Counter Overflow

Starting with the file src/linux-context.h, *PAPI* needs to be able to access the program counter of the currently running user-level thread in case of a counter overflow. It does so through the ucontext_t type structure, ctx.ucontext, that can be obtained with the getcontext() function. ctx.ucontext contains a mcontext_t type structure, uc_mcontext, that itself contains a __riscv_mc_gp_state type variable, __gregs. __riscv_mc_gp_state is a typedef of an array of 32 unsigned long integers. It is also known that the context for the program counter is stored in position 0 of a __riscv_mc_gp_state type variable. All these definitions are presented in Listing 3.1.

Given this information, the file src/linux-context.h can be appended with the contents of Listing 3.2 and *PAPI* can access the thread context in RISC-V.

3.1.2 Memory Barrier

The file src/mb.h contains the definition for the memory fence instruction in inline assembly for each supported architecture. According to the RISC-V Unprivileged Manual, [26], the memory fence instruction is fence. Also, according to GCC documentation [24], instructions that operate over memory other than those listed on the input and outputs listed in the inline assembly (which in the case of the fence instruction is none), should include the clobber "memory". As such, the changes needed in the src/mb.h file can be done by adding the lines presented in Listing 3.3.

```
...
#elif defined(__riscv)
#define rmb() asm volatile("fence":::"memory")
...
```

Listing 3.3: Changes to `src/mb.h`.

```
...
#elif defined(__riscv)
static inline long long
get_cycles( void ) {
    return 0;
}
...
```

Listing 3.4: Changes to `src/linux-timer.c`.

3.1.3 Access To Cycle Counter

The file `src/linux-timer.c` needs a definition for a `get_cycles(void)` function for each supported architecture. Because the firmware of the SiFive system blocked user-level reads of the `cycle` counter for security reasons and analyzing the solutions for other architectures this seems to be a common issue among them. The solution used is to define a dummy function that always returns a long long integer zero, as presented in Listing 3.4.

The 3 changes just discussed allow for *PAPI* to be compiled on a RISC-V computer. Following are the changes necessary for *PAPI* to abstract the Hardware Performance Counters on all RISC-V systems.

3.1.4 Vendor Identification

The next step is for *PAPI* to identify what processor vendor it is targeting. The library does it by parsing the `/proc/cpuinfo` in Linux-based systems, so to keep the new code consistent with the already existing one, the same method was followed. Nevertheless, other alternatives could also be used, like reading and interpreting the `mvendorid` CSR. It first identifies the general type of architecture by searching the file for a word that can identify the architecture and is followed by a string that can identify the specific component vendor (as an example, the string `"CPU implementer"` indicates that the target is an ARM CPU followed by a string that identifies the vendor of that specific CPU, `"ARM_FUJITSU"` for example). In the case of RISC-V computers[1], and in particular for the SiFive Unmatched board, the `/proc/cpuinfo` file is structured as shown in Listing 3.5.

The string that precedes the vendor identification string is `"uarch"` and as such, the block of code necessary to identify, in general, that the target is a RISC-V processor is presented in Listing 3.6. The `search_cpu_info()` function is already defined in *PAPI* and returns anything after the `":"` character in the line that starts with the given string, in this case, `"uarch"`. Usually, as in this case, what comes after is a string containing two parts separated by a comma. The first part is the vendor name and the second is the

[1] It is possible that vendors with tailored Linux versions define this file in a different way than the one presented here. If so, the vendor identification may require a different implementation.

```
processor       : 0
hart            : 2
isa             : rv64imafdc
mmu             : sv39
uarch           : sifive,u74-mc
...
```

Listing 3.5: `/proc/cpuinfo` file on a RISC-V computer.

```
 s = search_cpu_info(f, "uarch");
if (s) {
   char *v;
   v = strtok(s, ",");
   if (v) {
      // Nested if (or switch-case) section that would analyze the vendor string and identify
          ↪ the vendor
   }
}
```

Listing 3.6: Changes to `src/linux-common.c` for RISC-V identification.

processor name. The vendor name would then be compared against several known strings (in a nested if or switch-case) until a match is found. A code representing the vendor is stored in a `PAPI_hw_info_t` type structure kept by *PAPI* during the execution of the library.

In Section 3.2, it will be shown how the specific vendor was identified for the port to an actual RISC-V processor.

The changes presented thus far are sufficient for *PAPI*-proper code to support RISC-V, but not the *libpfm4* library it so heavily relies upon.

libpfm4 needs also to identify the correct implementation of the architecture. This is done with the `pfm_riscv_detect()` that works by using an already existing *libpfm4* function, `pfmlib_getcpuinfo_attr()` to obtain whatever information `/proc/cpuinfo` contains after the string "uarch". It then compares the returned string with all supported implementation's code strings until it finds a match. Then it sets the implementation enum defined in Listing 3.9. The definition of `pfm_riscv_detect()` is presented in Listing 3.7.

3.1.5 RISC-V Event List Entry Type, Register Type and PMU Configurtion Type

The *libpfm4* library encodes the events for each processor in a list of `[INSERT_ARCH_NAME]_entry_t` type structures that need to be defined for RISC-V as well. Given that the RISC-V privileged manual [27] does not constraint the implementations of the HPM, it is assumed that the encoding for the events is not masked or divided into sections. The same can be said for how the `mhpmeventX` registers are encoded, which is a definition that is also needed by the *libpfm4* library. Given this, the `riscv_entry_t` and `pfm_riscv_reg_t` were defined in a new `src/libpfm4/lib/pfmlib_riscv_priv.h` file, as presented in Listing 3.8.

A `pfm_riscv_cfg` variable is also needed. This variable is used by *libpfm4* to store the specific part being used when in the identification phase and later used to decide which PMU model should be used.

```c
int pfm_riscv_detect(void *this) {
    int ret;
    char buffer[128];

    ret = pfmlib_getcpuinfo_attr("uarch", buffer, sizeof(buffer));
    if (ret == -1)
        return PFM_ERR_NOTSUPP;
    if (strcmp(buffer, "[IMPLEMENTAION_CODE]") == 0)
        pfm_riscv_cfg.implementation = [IMPLEMENTAION_ENUM];
    else if (strcmp(buffer, "[ANOTHER_IMPLEMENTAION_CODE]") == 0)
        pfm_riscv_cfg.implementation = [ANOTHER_IMPLEMENTAION_ENUM];
    ...
    else
        return PFM_ERR_NOTSUPP;

    return PFM_SUCCESS;
}
```

Listing 3.7: Definition of `pfm_riscv_detect()`.

```c
...
typedef struct{
    const char *name;  /* event name */
    unsigned int code; /* event code */
    const char *desc;  /* event description */
} riscv_entry_t;

typedef union pfm_riscv_reg {
    unsigned int val;
} pfm_riscv_reg_t;
...
```

Listing 3.8: Definition of `riscv_entry_t` and `pfm_riscv_reg_t`.

```
...
typedef enum {
    // List of supported implementations
} pfm_riscv_implementation_t;

typedef struct {
    pfm_riscv_implementation_t implementation;
} pfm_riscv_config_t;

extern pfm_riscv_config_t pfm_riscv_cfg;
...
```

Listing 3.9: Definition of `pfm_riscv_cfg`.

```
...
int pfm_riscv_get_event_first(void *this) {
    return 0;
}
```

Listing 3.10: Definition of `pfm_riscv_get_event_first()`.

There are no restraints on how this variable should be defined due to every use of it also having to be defined for the new architecture supported, as will be seen later. It was chosen to implement it simply, using an enum of all supported RISC-V implementations (which is just 1 at the moment) but abstracting the new enum type so it can be easily changed in the future if, for example, it is desirable to support two versions of the same processor. As an example, if support for *PAPI* is introduced to the processors developed by the EPI project, it would be of great importance that the library can correctly identify which version of the implementation it is running on. The definitions appended to file `src/libpfm4/lib/pfmlib_riscv_priv.h` are presented in Listing 3.9.

Next are a series of functions *libpfm4* will associate with the PMU model of a RISC-V implementation that allows it to iterate over the available event list, get its encoding and *perf_events* encoding and also identify if the machine *PAPI* is being run on is, in fact, of that implementation.

3.1.6 Functions To Interact With The Event List

All functions shown in this section were declared in `src/libpfm4/lib/pfmlib_riscv_priv.h` and defined in `src/libpfm4/lib/pfmlib_riscv.c`

The *PAPI* PMU model requires a function to return the index of the first event available in said PMU.

Instead of just defining that the first event available for a processor on a given event list is the one with index 0, *PAPI* allows for the flexibility of letting the developer of the PMU model define which event is the first. The advantage here is that if two closely related versions of an implementation (let their names be `V1` and `V2`) share almost all configurable events, but `V2` has a new event not available in version 1. In this case, they can share the same event list by placing the new event in the first position of the list definition and programming the pfm_riscv_get_event_first() to return `0` if `V2` is detected and `1` if `V1` is detected.

For the context of this work, `pfm_riscv_get_event_first()` was simplified to return `0`, as is presented in Listing 3.10.

```
int pfm_riscv_get_event_next(void *this, int idx) {
    pfmlib_pmu_t *p = this;

    if (idx >= (p->pme_count - 1))
        return -1;
    return idx + 1;
}
```

Listing 3.11: Definition of `pfm_riscv_get_event_next()`.

```
int pfm_riscv_event_is_valid(void *this, int pidx) {
    pfmlib_pmu_t *p = this;
    return pidx >= 0 && pidx < p->pme_count;
}
```

Listing 3.12: Definition of `pfm_riscv_event_is_valid()`.

The function to get the next event on the list, `pfm_riscv_get_event_next()`, needs just to verify that the next index is not greater than the last index of the event list, or the number of available events in the PMU model (`pme_count`) minus 1. It is presented in Listing 3.11.

The function that, given an event index, checks if that index is valid for a specified PMU model, `pfm_riscv_event_is_valid()`, needs to check if the index is not lower than 0 and lower than or equal to the last index, or lower than the number of events available for the model. It is presented in Listing 3.12.

The function that validates the available event list (or table) for a given RISC-V implementations PMU model, `pfm_riscv_validate_table()` needs to iterate over the list of events and verify they have a name and a description and if the codes defined are all different from each other. These were the 3 variables defined to be part of the `riscv_entry_t` structure and that identify each event on a RISC-V event list.

This can be done by iterating over all events of the list, where the variable `pme_count` is the number of events in the list. In this loop, it is first checked if a `name` variable has a value different that `NULL`; if not, an error is printed to the console informing of which event index generated the problem. A similar verification is done for the event description variable, `desc`. To check for repeated event encodings a nested loop starting on the next event of the list is iterated over where the `code` variable of both events is compared.

The definition is presented in Listing 3.13.

The function that returns all available information about a given event, `pfm_riscv_get_event_info()` needs to fill out a variable (`info`) with the events information: name, description, code, equivalent (or alias event if such exists; it was assumed no aliases exist in RISC-V), index, PMU model it is associated with and additional architecture-specific attributes that may exist. As no additional attributes were created for RISC-V, the number of additional attributes (`nattrs`) must be reported as 0. The definition is presented in Listing 3.14.

3.1.7 Get Event Encoding and *perf_events*

The last two functions that *libpfm4* needs to be linked with a RISC-V PMU model are the ones that encode the events one wants to profile. Because of PAPI's modular nature and its ability to use different

```
int pfm_riscv_validate_table(void *this, FILE *fp) {
    pfmlib_pmu_t *pmu = this;
    const riscv_entry_t *pe = this_pe(this);
    int i, j, error = 0;

    for (i = 0; i < pmu->pme_count; i++)
    {
        if (!pe[i].name)
        {
            fprintf(fp, "pmu: %s event%d: :: no name (prev event was %s)\n", pmu->name, i, i
                ↪ > 1 ? pe[i - 1].name : "??");
            error++;
        }
        if (!pe[i].desc)
        {
            fprintf(fp, "pmu: %s event%d: %s :: no description\n", pmu->name, i, pe[i].name);
            error++;
        }
        for (j = i + 1; j < pmu->pme_count; j++)
        {
            if (pe[i].code == pe[j].code)
            {
                fprintf(fp, "pmu: %s events %s and %s have the same code 0x%x\n", pmu->name,
                    ↪ pe[i].name, pe[j].name, pe[i].code);
                error++;
            }
        }
    }
    return error ? PFM_ERR_INVAL : PFM_SUCCESS;
}
```

Listing 3.13: Definition of `pfm_riscv_validate_table()`.

```
int pfm_riscv_get_event_info(void *this, int idx, pfm_event_info_t *info) {
    pfmlib_pmu_t *pmu = this;
    const riscv_entry_t *pe = this_pe(this);

    info->name = pe[idx].name;
    info->desc = pe[idx].desc;
    info->code = pe[idx].code;
    info->equiv = NULL;
    info->idx = idx;
    info->pmu = pmu->pmu;

    /* no additional attributes defined for RISC-V */
    info->nattrs = 0;

    return PFM_SUCCESS;
}
```

Listing 3.14: Definition of `pfm_riscv_get_event_info()`.

```
int pfm_riscv_get_encoding(void *this, pfmlib_event_desc_t *e) {
    const riscv_entry_t *pe = this_pe(this);
    pfm_riscv_reg_t reg;

    reg.val = pe[e->event].code;
    evt_strcat(e->fstr, "%s", pe[e->event].name);
    e->codes[0] = reg.val;
    e->count = 1;

    return PFM_SUCCESS;
}
```

Listing 3.15: Definition of `pfm_riscv_get_encoding()`.

underlying middleware layers to access the Hardware Performance Counters, the event encoding occurs in two steps.

First, the `pfm_riscv_get_encoding()` function is executed to get the value one would need to set on the event select register for the event (or events) to be profiled. In the case of RISC-V, this is simply the value of the code section of a `riscv_entry_t` type variable. It is stored on a `pfm_riscv_reg_t` type variable (both types previously defined in Section 3.1.5). The definition of `pfm_riscv_get_encoding()` is presented in Listing 3.15.

Once *libpfm4* has the value to populate the event selector register, it needs a function to create a `perf_events` control structure that it can pass to the kernel driver with a system call. This function is called `pfm_riscv_get_perf_encoding()` and consists of populating a `perf_event_attr` type structure. It sets the event type to `PERF_TYPE_RAW`, meaning the `config` field will contain an implementation-specific event code, sets said field to the return of the `pfm_riscv_get_encoding()` function discussed above and manually clears the bits that exclude hyper-visor-, kernel- or user-level occurrences of the event from the count (i.e., inhibits the counter when an event occurs in a mode set to be excluded). This last part has to be added because if left unchanged *PAPI* later sets both the Hyper-Visor and kernel-level excludes on and the platform where the tests were being conducted did not support such exclusions it is unclear if the problem was caused by a lower software layer or a physical problem with the hardware used. The `pfm_riscv_get_perf_encoding()` definition is presented in Listing 3.16.

3.1.8 *libpfm4* Makefile Changes

In order for the new RISC-V files for the *libpfm4* library to be compiled changes had to be done to the `src/libpfm4/config.mk` and `src/libpfm4/lib/Makefile`.

In the first file, the target architecture for the compilation is identified. It was defined that if the architecture is `riscv64` or `riscv64` (RISC-V 32-bit), the environment variable `CONFIG_PFMLIB_ARCH_RISCV` is set and the changes are presented in Listing 3.17.

In the second file, the files are added to the to-be-compiled sources as well as the compilation flag `-DCONFIG_PFMLIB_ARCH_RISCV` so that *libpfm4* only compiles the necessary PMU models instead of always compiling all (the exact place where this flag is checked will be presented on Section 3.2, where implementation-specific changes are discussed). These changes are presented in Listing 3.18.

```c
int pfm_riscv_get_perf_encoding(void *this, pfmlib_event_desc_t *e) {
    pfmlib_pmu_t *pmu = this;
    pfm_riscv_reg_t reg;
    struct perf_event_attr *attr = e->os_data;
    int ret;

    if (!pmu->get_event_encoding[PFM_OS_NONE])
        return PFM_ERR_NOTSUPP;

    ret = pmu->get_event_encoding[PFM_OS_NONE](this, e);
    if (ret != PFM_SUCCESS)
        return ret;

    if (e->count > 1)
    {
        DPRINT("%s: unsupported count=%d\n", e->count);
        return PFM_ERR_NOTSUPP;
    }

    attr->type = PERF_TYPE_RAW;
    reg.val = e->codes[0];

    attr->config = reg.val;

    // risc-v can not set privilege levels
    attr->exclude_hv = 0;
    attr->exclude_kernel = 0;
    attr->exclude_user = 0;

    return PFM_SUCCESS;
}
```

Listing 3.16: Definition of `pfm_riscv_get_perf_encoding()`.

```
...
ifeq ($(ARCH),riscv64)
override ARCH=riscv
endif
ifeq ($(ARCH),riscv)
override ARCH=riscv
endif
...
ifeq ($(ARCH),riscv)
CONFIG_PFMLIB_ARCH_RISCV=y
endif
...
```

Listing 3.17: Changes to `src/libpfm4/config.mk`.

```
...
ifeq ($(CONFIG_PFMLIB_ARCH_RISCV),y)

ifeq ($(SYS),Linux)
SRCS += pfmlib_riscv_perf_event.c
endif

INCARCH = $(INC_RISCV)
SRCS += pfmlib_riscv.c [+ IMPLEMENTATION SPECIFIC PMU MODEL FILES]
CFLAGS += -DCONFIG_PFMLIB_ARCH_RISCV
endif
...
INC_RISCV=pfmlib_riscv_priv.h \
   events/[IMPLEMENTAION SPECIFIC EVENT LIST FILE] \
   events/[IMPLEMENTAION SPECIFIC EVENT LIST FILE] \
    [...]
   events/[IMPLEMENTAION SPECIFIC EVENT LIST FILE]
...
```

Listing 3.18: Changes to `src/libpfm4/lib/Makefile`.

3.2 SiFive U74-MC Support

Having the support for RISC-V processors implemented, the next step is to offer support to a specific implementation of the ISA. In this project, the computer chosen was a SiFive Unmatched board that is equipped with a SiFive U74-MC processor that features four SiFive U7 cores as the main computation facilities. In the context of *PAPI*, it is more usual to refer to support given to a processor and not the specific core it bolsters. As such, the processor supported shall be referred to as SiFive U74-MC to keep coherence with the presented listings as well as the publicly accessible implementation on GitHub.

As mentioned before, this section will contain some remarks on how one could add support for more RISC-V processors while taking advantage of the general RISC-V support discussed in the previous section.

3.2.1 Event List

A processor supported in *libpfm4* should have a new file created in the folder `src/libpfm4/lib/events/` named `riscv_[VENDOR NAME]_[IMPLEMENTAION NAME]_events.h` and this file should contain a static constant list of `riscv_entry_t` structures named `riscv_[VENDOR NAME]_[IMPLEMENTAION NAME]_pe[]` and its contents should be the name, code and description of each event offered by the processors HPM. An example is presented for the case of the U74-MC processor on Listing 3.19. The description and encoding for each event were taken from the processor manual [23] and the name was copied from the paper that originally gave `perf_events` support to the processor [3].

3.2.2 PMU Model

With the preparation for the definition of the PMU model almost complete, one needs to insert the model name into the enum typedef `pfm_pmu_t` that is maintained by *libpfm4* in file `src/libpfm4/include/perfmon/pfmlib.h`

```
static const riscv_entry_t riscv_sifive_u74_pe[] = {
    // Instruction Commit Events: .code[7:0]=0
    {.name = "EXCEPTION_TAKEN",
     .code = 0x0000100,
     .desc = "Exception taken"},
    [...]
};
```

Listing 3.19: Example of `riscv_sifive_u74_pe[]` event list.

```
typedef enum {
    ...
    PFM_PMU_RISCV_SIFIVE_U74,
    PFM_PMU_MAX
} pfm_pmu_t;
...
```

Listing 3.20: Changes to `libpfm4/include/perfmon/pfmlib.h`.

to encode such models. A new enum entry needs to be entered right before the `PFM_PMU_MAX` end marker. For the U74 the addition made is presented in Listing 3.20.

Finally, the implementation detection described in Section 3.1.4 presented in Listing 3.7 needs to be completed with the identification present in the Linux `/proc/cpuinfo` file where it reads `[IMPLEMENTAION_CODE]` and the chosen implementation enum name where it reads `[IMPLEMENTAION_ENUM]`. In this case, the implementation code is `"sifive,u74-mc"` and the chosen enum was `SIFIVE_U74_MC`.

With all this completed a new file can be created in the `src/libpfm4/lib/` named `pfm_riscv_[VENDOR NAME]_[IMPLEMENTAION NAME].c`, in this case `pfm_riscv_sifive_u74.c`.

This file contains a function that *libpfm4* can access through the PMU model to detect if the implementation running the library is the one modeled. This uses the `pfm_riscv_detect()` function and compares the return with the enum that was defined in Listing 3.20. This function should be named `pfm_riscv_detect_[VENDOR NAME]_[IMPLEMENTAION NAME]()`, in this case `pfm_riscv_detect_sifive_u74()` and its definition is presented in Listing 3.21.

The PMU model is of type `pfmlib_pmu_t` and should be named `riscv_[VENDOR NAME]_[IMPLEMENTAION NAME]_support`. It should contain the following variables:

```
static int pfm_riscv_detect_sifive_u74(void *this) {
    int ret;

    ret = pfm_riscv_detect(this);
    if (ret != PFM_SUCCESS)
        return PFM_ERR_NOTSUPP;

    if (pfm_riscv_cfg.implementation == SIFIVE_U74_MC)
        return PFM_SUCCESS;

    return PFM_ERR_NOTSUPP;
}
```

Listing 3.21: Definition of `pfm_riscv_detect_sifive_u74()`.

- desc - Description of the model;
- name - Name of the model without spaces;
- pmu - Enum previously defined in `src/libpfm4/include/perfmon/pfmlib.h`;
- pme_count - Number of events present on the event list;
- type - Type of model;
- supported_plm - Supported privilege levels;
- pe - List of available events;
- pmu_detect - Pointer to the `pfm_riscv_detect_[VENDOR NAME]_[IMPLEMENTAION NAME]()` function;
- num_counters - Number of programmable counters;
- num_fixed_cntrs - Number of fixed counters;
- max_encoding - Maximum number of 64-bit values that can be encoded into each event selector;
- get_event_encoding - Pointer to `pfm_riscv_get_encoding()` function;
- PFMLIB_ENCODE_PERF - Pointer to `pfm_riscv_get_perf_encoding()` function;
- get_event_first - Pointer to `pfm_riscv_get_event_first()` function;
- get_event_next - Pointer to `pfm_riscv_get_event_next()` function;
- event_is_valid - Pointer to `pfm_riscv_event_is_valid()` function;
- validate_table - Pointer to `pfm_riscv_validate_table()` function;
- get_event_info - Pointer to `pfm_riscv_get_event_info()` function.

Lastly, the model needs to be declared as an extern in file `src/pfmlib4/lib/pfmlib_priv.h` so the rest of the library can use it.

In this case, the `riscv_sifive_u74_support` model is defined as presented in Listing 3.22.

All that is left to change inside *libpfm4* is to add the new files to the Makefile. In the general case, Listing 3.18 had two device-specific fields, the first one (`[+ IMPLEMENTATION SPECIFIC PMU MODEL FILES]`) should be replaced with `pfmlib_riscv_sifive_u74.c` and the second one (`[IMPLEMENTAION SPECIFIC EVENT LIST FILE]`) with `riscv_sifive_u74_events.h`

3.2.3 *PAPI* Vendor Identification

Following what was described in Section 3.1.4 the `_linux_get_cpu_info()` function in file `src/linux-common.c` was completed with the specific vendor information for SiFive so that *PAPI* can correctly identify it. Additionally, the decode_vendor_string in the same file was also completed with the SiFive information. The `PAPI_VENDOR_RISCV_SIFIVE` was defined in file `src/papi.h` with the value 9 (next available value).

```
pfmlib_pmu_t riscv_sifive_u74_support = {
    .desc = "RISC-V SiFive U74",
    .name = "riscv_sifive_u74",
    .pmu = PFM_PMU_RISCV_SIFIVE_U74,
    .pme_count = LIBPFM_ARRAY_SIZE(riscv_sifive_u74_pe),
    .type = PFM_PMU_TYPE_CORE,
    .supported_plm = RISCV_PLM,
    .pe = riscv_sifive_u74_pe,
    .pmu_detect = pfm_riscv_detect_sifive_u74,
    .num_cntrs = 2,
    .num_fixed_cntrs = 2,
    .max_encoding = 1,

    .get_event_encoding[PFM_OS_NONE] = pfm_riscv_get_encoding,
    PFMLIB_ENCODE_PERF(pfm_riscv_get_perf_encoding),
    .get_event_first = pfm_riscv_get_event_first,
    .get_event_next = pfm_riscv_get_event_next,
    .event_is_valid = pfm_riscv_event_is_valid,
    .validate_table = pfm_riscv_validate_table,
    .get_event_info = pfm_riscv_get_event_info,
};
```

Listing 3.22: Definition of `riscv_sifive_u74_support`.

```
static void decode_vendor_string( char *s, int *vendor ) {
    ...
    else if ( strcasecmp( s, "RISCV_SIFIVE" ) == 0)
  *vendor = PAPI_VENDOR_RISCV_SIFIVE;
    ...
}
...
int _linux_get_cpu_info( PAPI_hw_info_t *hwinfo, int *cpuinfo_mhz ) {
    s = search_cpu_info(f, "uarch");
    if (s) {
        char *v;
        v = strtok(s, ",");
        if (v) {
            if ((strcasecmp(v, "sifive") == 0))
            strcpy(hwinfo->vendor_string, "RISCV_SIFIVE");
        }
    }
    ...
}
```

Listing 3.23: Changes to `src/linux-common.c` for RISC-V Vendor identification.

```
####################
# RISC-V SiFive U74 #
####################
CPU,riscv_sifive_u74
#
PRESET,PAPI_L1_DCM,NOT_DERIVED,DCACHE_MISS_MMIO_ACCESSES
PRESET,PAPI_L1_ICM,NOT_DERIVED,ICACHE_RETIRED
PRESET,PAPI_L1_TCM,DERIVED_ADD,DCACHE_MISS_MMIO_ACCESSES,ICACHE_RETIRED
PRESET,PAPI_TLB_DM,NOT_DERIVED,DATA_TLB_MISS
PRESET,PAPI_TLB_IM,NOT_DERIVED,INST_TLB_MISS
PRESET,PAPI_TLB_TL,DERIVED_ADD,DATA_TLB_MISS,INST_TLB_MISS
#
PRESET,PAPI_BR_UCN,DERIVED_ADD,JAL_INSTRUCTION_RETIRED,JALR_INSTRUCTION_RETIRED
PRESET,PAPI_BR_CN,NOT_DERIVED,CONDITIONAL_BRANCH_RETIRED
PRESET,PAPI_BR_MSP,DERIVED_ADD,BRANCH_DIRECTION_MISPREDICTION,BRANCH_TARGET_MISPREDICTION
PRESET,PAPI_BR_PRC,DERIVED_SUB,CONDITIONAL_BRANCH_RETIRED,
    ↪ BRANCH_DIRECTION_MISPREDICTION,BRANCH_TARGET_MISPREDICTION
#
PRESET,PAPI_FMA_INS,NOT_DERIVED,FP_FUSEDMADD_RETIRED
PRESET,PAPI_TOT_INS,NOT_DERIVED,INSTRUCTIONS
PRESET,PAPI_INT_INS,DERIVED_ADD,INTEGER_LOAD_RETIRED,INTEGER_STORE_RETIRED,
    ↪ INTEGER_ARITHMETIC_RETIRED,INTEGER_MULTIPLICATION_RETIRED,INTEGER_DIVISION_RETIRED
PRESET,PAPI_FP_INS,DERIVED_ADD,FP_LOAD_RETIRED,FP_STORE_RETIRED,FP_ADDITION_RETIRED,
    ↪ FP_MULTIPLICATION_RETIRED,FP_FUSEDMADD_RETIRED,FP_DIV_SQRT_RETIRED,OTHER_FP_RETIRED
PRESET,PAPI_LD_INS,DERIVED_ADD,INTEGER_LOAD_RETIRED,FP_LOAD_RETIRED
PRESET,PAPI_SR_INS,DERIVED_ADD,INTEGER_STORE_RETIRED,FP_STORE_RETIRED
PRESET,PAPI_BR_INS,DERIVED_ADD,JAL_INSTRUCTION_RETIRED,JALR_INSTRUCTION_RETIRED,
    ↪ CONDITIONAL_BRANCH_RETIRED
PRESET,PAPI_TOT_CYC,NOT_DERIVED,CYCLES
PRESET,PAPI_LST_INS,DERIVED_ADD,INTEGER_LOAD_RETIRED,FP_LOAD_RETIRED,INTEGER_STORE_RETIRED,
    ↪ FP_STORE_RETIRED
#
PRESET,PAPI_FML_INS,NOT_DERIVED,FP_MULTIPLICATION_RETIRED
PRESET,PAPI_FAD_INS,NOT_DERIVED,FP_ADDITION_RETIRED
PRESET,PAPI_FP_OPS,DERIVED_ADD,FP_ADDITION_RETIRED,FP_MULTIPLICATION_RETIRED,
    ↪ FP_FUSEDMADD_RETIRED,FP_DIV_SQRT_RETIRED,OTHER_FP_RETIRED
```

Listing 3.24: Changes to `src/papi_events.csv`.

3.2.4 *PAPI* Presets

As discussed in Section 2.6.1, preset events are a major feature of the library and should be supported to its full extent. As such, every processor supported by *PAPI* should have its entry in the file `src/papi_events.csv`, where the preset mapping is done. All existing *PAPI* preset events are defined in [16].

The contents of Listing 3.24 were appended to the `src/papi_events.csv` file. The parser for the file ignores lines that start with #. Additions to the file should specify the beginning of a new processor by starting with the line `CPU,[IMPLEMENTAION NAME]`. Then, the presets have to be inserted one per line, starting with the word `PRESET` followed by a comma and the preset event being mapped (`PAPI_L1_DCM` in the first event of the listing). Next, one needs to indicate if the mapping is derived or not. Deriving a mapping means that the preset is mapped to a combination of the events available on the processor. The combination here can mean several things, like summing up event counts (`DERIVED_ADD`), subtracting (`DERIVED_SUB`), other simple operations or even specifying a complex operation (with `DERIVED_POSTFIX`).

Function Call	perf_events	PAPI Library
Initialization	57.98μs	81200μs
Start count	11.67μs	947.11μs
Read count	7.76μs	13.95μs
Stop count and read	15.89μs	19.41μs
Termination	15.79μs	1190μs
Total	100.24μs	83360μs

Table 3.1: Measured execution time overheads (average) of *PAPI* Library calls in relation to *perf_events* calls.

Exec. Time	Event ID	perf_events (a)	PAPI Library (b)	Abs. error ((b) i.r.t (a))
10s	FP_LOAD	495450012	495450012	0.00%
	FP_STORE	124722512	124722512	0.00%
	FP_MADD	372222502	372222502	0.00%
	D$_BUSY	1639038926	1610686782	1.73%
	BR_TARGET_MISS	1659736	1665510	0.35%
	FP_INTERLOCK	1397108171	1409479181	0.89%
	I$_MISS	193767	203070	4.80%
	D$_MISS	18586915	18567696	0.10%
	D$_WB	77905	83298	6.92%
	CYCLES	12928115705	12622306130	2.37%
30s	CYCLES	38883173173	38121784778	1.96%
60s	CYCLES	84688102810	83346239838	1.58%

Table 3.2: *PAPI* Library and *perf_events* event monitoring comparison with Gemm benchmark.

3.2.5 Case-Specific Note

The *GCC* version installed on the SiFive Unmatched used for development and testing did not support one library needed to compile Fortran test files. Because of this, that compilation step was commented.

3.3 Implementation Evaluation and Result Discussion

To determine the overhead introduced by this implementation of *PAPI* and compare it to a direct interface with *perf_events* three tests were designed and conducted.

First, the average total overhead for initialization, termination, and events start, read, and stop on *PAPI* and *perf_events* were measured and compared. To do so, the total time elapsed while running each step was controlled with the `time.h` library `clock_gettime()` function. As detailed in Table 3.1, the observed overheads for *PAPI* are higher than for *perf_events*, with the typical case accounting for 13.95µs for a *PAPI* event read (vs 7.76µs on *perf_events*) and 947µs for event start (vs 11.67µs on *perf_events*). The initial *PAPI* start overhead is due to the underlying mechanism of base *PAPI* code, which requires creating an event structure (including memory allocation and structure initialization), calling the *libpfm4* library to translate event names to event codes, and then issuing a system call for *perf_events*. Therefore, when considering the typical cumulative use case of event count plus event stop and read, an average overhead of 966.52µs is observed for *PAPI* , which compares with 27.56µs for *perf_events*. Finally, there is also an initial overhead for library initialization and termination which (on average) takes 81.2ms and 1.19ms for *PAPI*, respectively, compared with the 57.98µs and 15.79µs for *perf_events*.

Although the observed overheads are consistent with previous research [30], a second analysis was performed by using *Strace* to count the system calls invoked by each library during program profiling. *Strace* showed 7716 system calls for *PAPI* and 43 calls for *perf_events*, with most calls being associated with the initialization step, particularly to allocate initial memory structures, read the configuration files for the current architecture and initialize and test *perf_events* functionalities.

The measurements obtained by using *PAPI* were further validated by performing a detailed profiling of the *GEMM* kernel [17] (configured to last about $10s$ when running on the SiFive U7 core). This is done by measuring average counts for multiple events across 100 runs of the kernel. The obtained results (presented in Table 3.2) show that the counts for deterministic events (e.g., floating-point stores, fused-multiply-add operations, etc.) present the same measured count using *perf_events* and *PAPI* . However, other events such as cache operations and cycles, show slight variations due to OS-related interference during the execution and the *PAPI* software infiltrating the counts while it starts, stops and reads them. By increasing the execution time of the GEMM kernel to $30s$ and $60s$, we observe that the OS interference is mitigated, decreasing variations from 2.37% to 1.58% (see Table 3.2).

Taking all of this into mind, it is evident the *PAPI* library causes a big overhead over simply using *perf_events* directly but most of that overhead is spent on initialization and termination, which would be more hidden in normal use cases where the application being tested would be much larger than the minute test kernel used here. It was also demonstrated that some types of events do not have their count affected at all by *PAPI* overhead, namely events that count instruction retirements or stores/loads. Nevertheless, events related to cache access or device utilization are affected indirectly by *PAPI* due to the computer having to divide its resources between the actual program being profiled and the profiling software. All that said, the point of this work is to provide developers with an easy-to-use way of gathering performance data with acceptable accuracy, and that seems to have been achieved with the results presented.

3.4 Summary

This chapter outlines the process of implementing *PAPI* support for the RISC-V architecture, with a particular focus on the SiFive U74-MC processor.

The foundation of the implementation lies in the creation of a custom RISC-V PMU model. This model encapsulates the core attributes of a PMU, including descriptions, names, event lists, and encoding methods. The model is not limited to a specific RISC-V processor but can serve as a template for other RISC-V implementations.

A central aspect of the implementation is the definition of an event list. This list contains information about the events that can be monitored on the RISC-V architecture. It encompasses event names, codes and descriptions.

The process of event encoding involves determining the values to set in the event select register for monitoring specific events.

Changes are made to the *PAPI* source code to ensure that the library correctly identifies SiFive as a vendor. While the focus is on the SiFive U74-MC processor, this is a general step applicable to all

RISC-V-based processors.

This chapter also evaluates the overhead introduced by *PAPI* compared to direct usage of *perf_events*. The measured overhead, especially during initialization and termination, highlights the trade-offs between using *PAPI* for its ease of use and the more streamlined but complex *perf_events* interface.

The evaluation extends to measuring events when executing a GEMM kernel, validating the consistency of event counts between *PAPI* and *perf_events*. This validation underscores the suitability of *PAPI* for accurate performance data gathering on RISC-V processors.

While the primary focus is on implementing *PAPI* support for the SiFive U74-MC processor, the underlying processes and concepts have broader applications across the RISC-V architecture. The implementation serves as a foundational reference for enabling performance analysis capabilities on various RISC-V-based processors, demonstrating the adaptability of *PAPI* in a diverse hardware ecosystem.

CHAPTER 4

Precise Event Sampling on RISC-V

As has been discussed, PES offers very robust profiling techniques that allow the user to acquire more in-depth information about the execution of the profiled software. This chapter focuses on proposing a specification for a RISC-V PES facility and how it could be integrated into modern profiling tools. The chapter starts by defining the objectives of the system and the requirements to achieve them. Then a minimal implementation of the system on a CVA6 processor is presented.

4.1 Precise Event Sampling Specification

When it came to designing a PES system for RISC-V, it was attempted to provide basic functionalities compatible with those of Intel *PEBS*, described in [21] and Section 2.7.1 of this document. The requirements for integration into *PAPI* were also taken into account, by considering previous work on the subject [29].

4.1.1 Objectives

The objectives for this system are to be able to sample the processor state with a sampling triggered by a performance counter reaching a predefined threshold. It is desired that once the sample is triggered, an external interrupt stops the processor's execution and stores information about the processor state in a memory-mapped buffer. This buffer should be able to be mapped with a variable size and, once full, should trigger a software interrupt so the user-level software using the PES system can gather its contents and process or store them.

The control of the new hardware introduced should be done by kernel-level software, like *perf_events*, and communicate with the hardware only through an OpenSBI interface, respecting the RISC-V CSRs Machine privilege level inside the HPM.

This user-level software using PES system could be *PAPI*, as an example, or any other sampling tool that can interface with *perf_events*.

4.1.2 Requirements

Following the aforementioned objectives for the system, the requirements are:

- A method to manage the buffer, considering its location in physical memory, the current buffer status and the next available position;
- A mechanism to control the thresholds of all counters and to trigger register storage on the shared buffer whenever this threshold is exceeded;
- A method for the system to notify the user-level tool to read the buffer.
- A method for the Kernel to control the HPM;
- A method for the user-level tool to communicate with the Kernel;

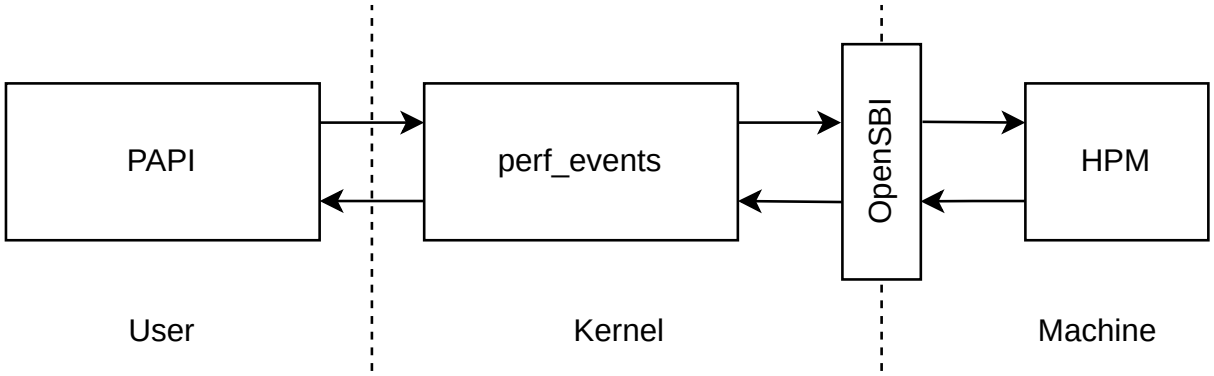

Figure 4.1: Overview of the system software structure.

4.1.3 Specification

The objectives defined require that new hardware inside the HPM is specified.

The PES buffer requires 3 new CSRs: the `mhpmmmaped` to store the address of the mapped memory, the `mhpmmsize` to store the size of the buffer and `mhpmmoffset` to keep the address of the next writable memory position. All three are `MXLEN-bit` Write Any Read Legal (WARL) registers.

A way to store the threshold values is also necessary. That can be done by having 32 new CSRs: `mhpmthresholdcyc` to store the cycle counter threshold, `mhpmthresholdinstret` to store the instruction retired counter threshold and `mhpmthreshold3-mhpmthreshold31` to store the programmable counters thresholds. All of them are `MXLEN-bit` WARL registers. When `MXLEN=32`, `mhpmthresholdcyc`,`mhpmthresholdinstret` and `mhpmthreshold3-mhpmthreshold31` store bits 31-0 of the corresponding counter threshold and new `mhpmthresholdcych` to store the cycle counter threshold, `mhpmthresholdinstreth` to store the instruction retired counter threshold and `mhpmthreshold3h-mhpmthreshold31h` CSRs are used to store bits 63-32.

For the sampling to occur, a comparator must be implemented in hardware for each performance counter. When the value of the counter is equal to or higher than the respective threshold, a signal should be sent to the PLIC so a machine-level interrupt handler can store any processor information in the memory-mapped buffer. This handler should also advance the value of the next available buffer position, `mhpmmoffset` by the size of the information just written.

When the buffer is full, detected by a comparator in hardware that checks if `mhpmmsize=mhpmmoffset`. When this is true, an interrupt must be generated for the user-level tool to gather the information stored on the buffer.

For the user-level to be able to access the HPM, kernel-level software must handle its requests and relay them to machine-level. The kernel-level software would ideally be *perf_events* as an example, but could also be a collection of tailored system calls, as long as it provides communication with the HPM. The communication between kernel-level and machine-level also requires work, namely the addition of read and write functions to new CSRs to an existing OpenSBI extension that already supports the current RISC-V HPM specification. An overview of how the software structure should look is presented in Figure 4.1.

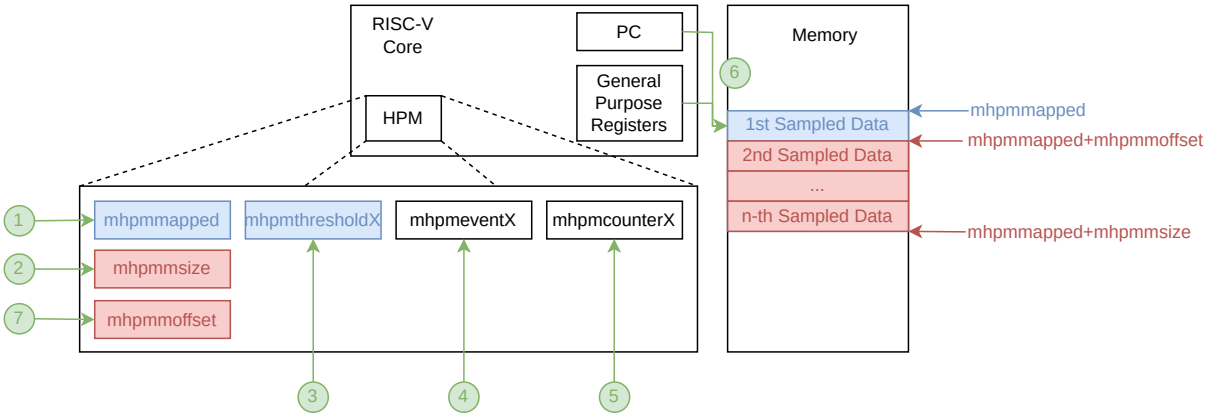

Figure 4.2: Precise Event Sampling on RISC-V system architecture. Parts in white already existed, parts in blue were implemented, and parts in red were not implemented.

4.2 Precise Event Sampling Architecture

The system implemented in the CVA6 processor follows the specification discussed in Section 4.1.3. Due to time constraints, it was not possible to implement an entire system but it was possible to demonstrate that without, significant modifications to the hardware resources and the Kernel underlying layers, this method of software profiling can be achieved.

The system architecture specified is represented in Figure 4.2 and would work as follows: (1) The kernel-level software maps a section of memory to be used as a buffer and stores it in the `mhpmmmaped` CSR; (2) The size of the buffer is stored in `mhpmmsize`; (3) The chosen threshold value to work as the sampling rate is stored in the `mhpmthresholdX` CSR; (4) The event encoding is stored in the `mhpmeventX` CSR as it happens in current implementations of RISC-V HPMs; (5) The `mhpmcounterX` increments at each event occurrence; (6) When the count is equal to the defined threshold, a signal is sent to the PLIC so that an external interrupt is generated. Some collection of data about the core's state is appended to the memory-mapped location and the counter is reset; (7) The `mhpmmoffset` is incremented by the size of the written information. Once the buffer is full, an interrupt is generated so that the profiling software can retrieve the information.

From this architecture, some parts were not implemented in the work described in the following sections, but comments will be given on what is missing in each part of the design presented to function.

4.2.1 New CSRs

In the RISC-V privileged specification [27] the CSR address ranges are explained to be comprised of encoded into the first 4 bits of the 12-bit encoding. The first 2 bits indicate if the CSR is read/write (encoded `00`, `01` or `10`) or read-only (encoded with `11`) and the second 2 bits encoded the lowest privilege level that can access them (`00` for user-level, `01` for Superior-level, `10` for Hypervisor-level and `11` for machine-level). All existing machine-level HPM CSRs are encoded in the `0xB00-0xB7F` address range.

To implement the proposed functionality, new CSRs were needed. With the RISC-V specification in mind, they are declared in the same address range as the existing machine-level HPM CSRs, making

them readable and writable only in machine-mode. Their implementation is done on the file core/include/riscv_pkg.sv and the added registers where:

- mhpmthresholdcyc with address 0xB21;
- mhpmthresholdinstret with address 0xB22;
- mhpmthreshold3-8 with addresses 0xB23-0xB28;
- mhpmmmaped with address 0xBC0.

4.2.2 Memory-Mapped Location Address

It was decided the address for the memory-mapped location should be stored in a new CSR and have its content managed inside the HPM. For this, 7 new registers were created in the file core/perf_counters.sv of the CVA6 source-code: treshhold[6:1] and mmaped_addr and are represented by the d and q signals of a D Flip Flop. Functionality was added for these registers to be readable and writeable and update their q values at each clock cycle, unless of course, the reset signal is clear (active on low), in which case the q values are set to 0. These changes are presented in Listing 4.1 where: addr_i is the CSR address passed to the HPM to read or write to and we_i is the input signal write enable.

4.2.3 SBI-Like Interface Extension

As alluded to in Section 4.2.1, the newly created CSRs, as well as the mhpmeventX and mhpmcounterX, are not writeable in User Mode and need firmware support to be written. During testing, the underlying software used to run user-level binaries was the RISC-V Proxy Kernel and Boot Loader [18], which does not support any environment calls for the user to write to the HPM CSRs. It did, however, support an SBI-like interface for other environment calls.

The interface was extended with four new calls: SBI_HPM_SET_MEVENT, SBI_HPM_SET_MTHRESHOLD, SBI_HPM_SET_MCOUNTER, SBI_HPM_SET_MMAPED. These calls are implemented in the simplified manner presented in Listing 4.2.

It is pertinent to point out that the SBI_HPM_SET_MTHRESHOLD also sets the Machine external interrupt enable bit on the machine interrupt enable CSR. This arms the PES system by letting the machine-level know that it should trap external interrupts.

SBI calls are, of course, accessed exclusively by the supervisor-level. As such it was also necessary to modify the user-level trap vector so that the Supervisor immediately runs an environment call again with the same arguments when the system call code corresponds with one of the newly defined calls.

4.2.4 External Interrupt Request Control

In the same file, functionality to alert the PLIC to a counter threshold overflow was also added. This control is done by comparing the value of the programmable counter to its corresponding threshold register and, if the count is greater or equal to the threshold the output signal perf_counter_irq_o is set. The description of this behavior is presented in Listing 4.3. This signal is propagated out of the core and

```
...
// Signal Declaration
logic [63:0] threshold_d[8:1];
logic [63:0] threshold_q[8:1];
logic [63:0] mmaped_addr_d;
logic [63:0] mmaped_addr_q;
...
always_comb begin : generic_counter
    ...
    // Unless Changed, last value is maintained
    threshold_d = threshold_q;
    mmaped_addr_d = mmaped_addr_q;
    ...
    unique case (addr_i)
        // Read from CSR_MHPM_THRESHOLD_X
        riscv::CSR_MHPM_THRESHOLD_CYC,
        riscv::CSR_MHPM_THRESHOLD_INST_RET,
        riscv::CSR_MHPM_THRESHOLD_3,
        riscv::CSR_MHPM_THRESHOLD_4,
        riscv::CSR_MHPM_THRESHOLD_5,
        riscv::CSR_MHPM_THRESHOLD_6,
        riscv::CSR_MHPM_THRESHOLD_7,
        riscv::CSR_MHPM_THRESHOLD_8 : begin data_o =
            ↪ threshold_q[addr_i-riscv::CSR_MHPM_THRESHOLD_CYC + 1];end
        ...
        // Read from CSR_MHPM_MMAPED
        riscv::CSR_MHPM_MMAPED : begin data_o = mmaped_addr_q; end
        ...
    endcase

    if(we_i) begin
        unique case(addr_i)
            // Write to CSR_MHPM_THRESHOLD_X
            riscv::CSR_MHPM_THRESHOLD_CYC,
            riscv::CSR_MHPM_THRESHOLD_INST_RET,
            riscv::CSR_MHPM_THRESHOLD_3,
            riscv::CSR_MHPM_THRESHOLD_4,
            riscv::CSR_MHPM_THRESHOLD_5,
            riscv::CSR_MHPM_THRESHOLD_6,
            riscv::CSR_MHPM_THRESHOLD_7,
            riscv::CSR_MHPM_THRESHOLD_8 : begin
                ↪ threshold_d[addr_i-riscv::CSR_MHPM_THRESHOLD_CYC + 1] = data_i; end
            ...
            // Write to CSR_MHPM_MMAPED
            riscv::CSR_MHPM_MMAPED  : begin mmaped_addr_d = data_i; end
            ...
        endcase
    end
end
...
always_ff @(posedge clk_i or negedge rst_ni) begin
    if (!rst_ni) begin
        ...
        // If reset signal is clear (active on low) CSRs are reset
        threshold_q       <= '{default:0};
        mmaped_addr_q     <= '{default:0};
    end else begin
        ...
        // Else, flip flop saves value
        threshold_q       <= threshold_d;
        mmaped_addr_q     <= mmaped_addr_d;
    end
end
```

Listing 4.1: Changes to core/perf_counters.sv.

```
void mcall_trap(uintptr_t *regs, uintptr_t mcause, uintptr_t mepc) {
  write_csr(mepc, mepc + 4);

  uintptr_t n = regs[17], arg0 = regs[10], arg1 = regs[11], retval, ipi_type;
  switch (n){
    case SBI_HPM_SET_MEVENT:
      if (arg0 >= 3 && arg0 <= 8) {
        write_csr(mhpmevent3 + arg0 - 3, arg1);
        retval = 0;}
      else
        retval = -EINVAL;
      break;
    case SBI_HPM_SET_MTHRESHOLD:
      if (arg0 >= 1 && arg0 <= 8) {
        set_csr(mie, MIP_MEIP);
        write_csr(0xb21 + arg0 - 1, arg1); // 0xb21 = mhm_threshold_3
        retval = 0;}
      else
        retval = -EINVAL;
      break;
    case SBI_HPM_SET_MCOUNTER:
      if (arg0 >= 3 && arg0 <= 8) {
        write_csr(mhpmcounter3 + arg0 - 3, arg1);
        retval = 0;}
      else
        retval = -EINVAL;
      break;
    case SBI_HPM_SET_MMAPED:
      write_csr(0xbc0, arg1); // 0xbc0 = mhm_mmaped
      retval = 0;
      break;
  ...
  }
  regs[10] = retval;
}
```

Listing 4.2: Changes to `machine/mtrap.c` in riscv-pk library.

```
always_comb begin : perf_irq_ctrl
  perf_counter_irq_o = 'b0;
  for (int unsigned i = 1; i <= 8; i++) begin
    if (generic_counter_q[i] >= threshold_q[i] && threshold_q[i] != 'b0) begin
      perf_counter_irq_o = 'b1;
    end
  end
end
```

Listing 4.3: HPM External Interrupt Request Control.

```
...
assign irq_sources[ariane_soc::NumSources-1:8] = '0;
...
assign irq_sources [7] = perf_counter_irq_i;
...
```

Listing 4.4: Changes to `corev_apu/tb/ariane_peripherals.sv`.

into the PLIC, inside the peripherals module (file `corev_apu/tb/ariane_peripherals.sv`), in the form of an interrupt request source. CVA6 supports 30 external interrupt request sources but only 7 (indexes 0 through 6) are used, having the remaining reserved. It was chosen to use source 7 to represent the performance monitoring threshold overflow, as shown in Listing 4.5. Inside the PLIC the request is treated by a gateway as described in Section 2.3.2.

4.2.5 External Interrupt Handler

When the count reaches the predefined threshold and the PLIC signals a pending external Machine interrupt, the RISC-V Proxy Kernel's trap table is called. This library did not originally support machine-mode external interrupts so that had to be added and is done in the first lines of Listing 4.5. Because no other Machine external interrupts exist, there was no case selector for which handler should be run. The handler starts by letting the PLIC know the interrupt is being treated. Then proceeds to stop the counting of `mhpmcounter3`. Then the handler saves the value of `MEPC` to the memory location whose address is stored in `mhpmmmaped`.

The reader might be already questioning the correctness of the last step, which is, in fact, at the very least incomplete. In Machine mode, the CVA6 processor does not translate memory addresses, expecting to be given a physical address instead of a virtual one. This means that the address stored `mhpmmmaped` that was mapped in User mode could not be passed directly to a store instruction on machine-level code. And, of course, this instruction runs without problem, writing to some other memory location no one claims. This error was not detected until very late in development because the simulator used to test the implementation was malfunctioning when variables were printed to stdout, which rendered reading the memory location in the User mode test program useless and not even attempted. Some way to translate the address should be added before storing the contents of `MEPC` or, the memory could be mapped in Machine mode and some kind of interrupt could be used to later retrieve the contents of the memory location in User mode.

```
    li a0, IRQ_M_EXT * 2
    bne a0, a1, .Lbad_trap

    # Yes.
    # Handler
    li a2, 0xc200004      // Signal to the PLIC that
    lw a0, 0(a2)          //the interrupt was taken
    li a0, 8              // Inhibit the counting of
    csrs 0x320, a0        //events in mhpmcounter3

    csrr a0, mepc
    csrr a2, 0xbc0
    sw a0, 0(a2)

    csrw mhpmcounter3, x0 // Reset mhpmcounter3
    li a2, 0xc200004      // Signal to the PLIC that
    sw a0, 0(a2)          //the interrupt handler is done
    li a0, 8              // Clears inhibition
    csrc 0x320, a0        //of mhpmcounter3
    j .Lmret
```

Listing 4.5: Changes to `corev_apu/tb/ariane_peripherals.sv`.

Version	Total LUTs	Logic LUTs	LUTRAMs	SRLs	FFs
Original	75749	73124	1996	629	48927
With PES	75377	72752	1996	629	49390
Δ	-372	-372	0	0	463

Table 4.1: Reported resource utilization for the CVA6 processor.

The handler continues by resetting the counter, signaling the PLIC that the handler is finished so new interrupts can be generated and re-enable the count. These three steps allow for the sampling to occur indefinitely.

Besides the error already discussed, several upgrades could be introduced to this handler to render the system more usable in real-world applications instead of being a simple proof of concept. In particular, the handler should support all counters instead of just `mhpmcounter3`, the values stored in the sampling should include more information about the core's state than just the `MEPC` and the samples should not overwrite each other, like it was defined in Section 4.1.3.

Even so, this handler manages to prove that the hardware changes introduced to the CVA6 processor are enough to support PES.

4.3 Evaluation and Result Discussion

In order to understand the impact of the added hardware on resource utilization, the CVA6 project was synthesized and implemented with and without modifications in Vivado 2019.2 with the AMD Virtex `7vx485t-ffg1761` FPGA as the target device. Although the "Runtime Optimized" strategy was used to reduce optimization-introduced discrepancies the resource utilization report, gathered in Table 4.1, shows that the implementation with the PES facilities use fewer look-up tables and marginally more flip-flops when comparing the top-level module. This means that the inherent uncertainty of the optimization path taken by Vivado was enough to overshadow the resources used in the newly added software. In terms of

the clock period, the original targeted $20ns$ was still achieved with no reported timing errors. Furthermore, analyzing the 100 lowest slacks in the timing report, it is noticeable the great majority of them, all apart from one, are unrelated to the changes, belonging to the RAM module and the single one related to the core belongs to the cache submodule and has a slack greater than $1.5ns$. In other words, the PES facilities implemented have no impact on the working frequency of the processor.

To test the PES functionality, a simple test program was written and is presented in Listing 4.6.

The application consists of doing a mmap and storing the returned address to the newly created `mhpmmmaped`, setting the count threshold to 3, programming the event "Floating Point Instructions" to be counted and executing a float sum operation 15 times.

This test serves the purpose of generating more than one sample and allows for the analysis of the simulation waveforms to confirm that the implemented hardware and software modification produce the desired result of storing samples in memory.

A waveform file was obtained from running the binary of this program in the Verilator simulator and a few snippets will be discussed below. In the images, the $ symbol means the value is in hexadecimal format, the % symbol means it is in binary and no symbol means decimal format.

In Figure 4.3 it is possible to see the values of the HPM CSRs at the beginning of the count. The mapped memory address is `0x12000`, the threshold is correctly set to 3 and the event selector to 21. It is also visible that the count started and reached 3, where it stopped and the interrupt request signal was activated. The `irq_sources_i` signal (interrupt request sources) from the PLIC almost immediately is set to `0x80` which is the one-hot encoding for the 7th source that was defined to be the count threshold interrupt request. Following, the PLIC gateway sets the interrupt pending to the same value which means it has notified the hart of the pending interruption.

Jumping ahead in time, displayed in Figure 4.4 is the initial read (indicated by the `operation=0x25` signal) of the PLIC memory position `vaddr=0xc200004` to notify of the claim. The claim is confirmed by the signal `claim` at the end of the figure being equal to the one-hot encoding for the count threshold source.

Advancing to Figure 4.5, the first thing to point out is the `mcountinhibit_q` being set to 8, meaning the `mhpmcounter3` is stopped. Then, the value of `MEPC` (represented by the signal `data=mepc_q`) is stored (indicated by the signal `operation=0x27`) in the memory location pointed to by `mhpmmmaped` (represented by the signal `vaddr=mmaped_addr_d`). This store is where the actual sampling occurs.

In Figure 4.6, the counter is reset to 0 and, as a consequence the `perf_counter_irq_o` and `irq_sources_i` signals are unset. The PLIC memory location is written to notify the handler is ending.

Finally, the `mcountinhibit` is cleared and the PLIC process ends by temporarily setting the `complete` signal with the one-hot encoding of the source, as shown in Figure 4.7.

All the snippets shown demonstrate that the implemented system has the basic functionality to sample the processor state and that this kind of facility can be easily standardized and added into RISC-V computers as a powerful tool to profile software.

```
uintptr_t sysCallMmap() {
    long result = 0;
    __asm__ volatile("addi a7, x0, 222":::); // Syscall to mmap
    __asm__ volatile("addi a0, x0, 0":::); // Address = NULL
    __asm__ volatile("addi a1, x0, 8":::); // Length = 8B
    __asm__ volatile("addi a2, x0, 3":::); // prot = PROT_READ|PROT_WRITE
    __asm__ volatile("addi a3, x0, 0x22":::); // flags = MAP_PRIVATE|MAP_ANONYMOUS
    __asm__ volatile("addi a4, x0, -1":::); // fd = -1
    __asm__ volatile("addi a5, x0, 0":::);  // offset = 0
    __asm__ volatile("ecall":::);
    __asm__ volatile("mv %0, x10":"=r"(result)::);
    return result;
}

int main(int argc, char const *argv[])
{
    float a = 0.3, b = 0.5;
    uintptr_t *ptr = NULL;

    ptr = (uintptr_t *)sysCallMmap();

    // Set mhpmmmaped address to mmaped address
    __asm__ volatile("add a1, x0, %0"::"r"(ptr):);
    __asm__ volatile("addi a7, x0, 12":::);
    __asm__ volatile("ecall":::);

    // Set threshold to 3
    __asm__ volatile("addi a1, x0, 3":::);
    __asm__ volatile("addi a7, x0, 10":::);
    __asm__ volatile("ecall":::);


    // Set event to be counted - Floating Point Instructions
    __asm__ volatile("addi a1, x0, 21":::);
    __asm__ volatile("addi a7, x0, 9":::);
    __asm__ volatile("ecall":::);

    for (int i = 0; i < 15; i++) {
        a = a + b;
    }
    return 0;
}
```

Listing 4.6: PES test program.

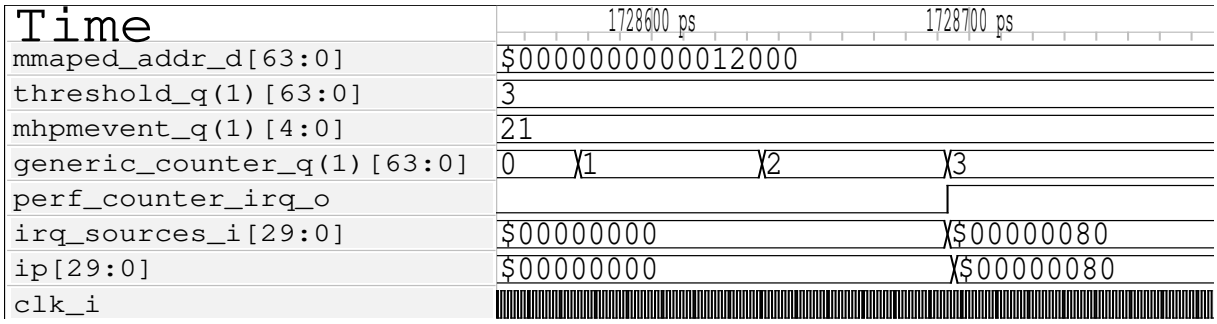

Figure 4.3: Register values during counting and interrupt request generation.

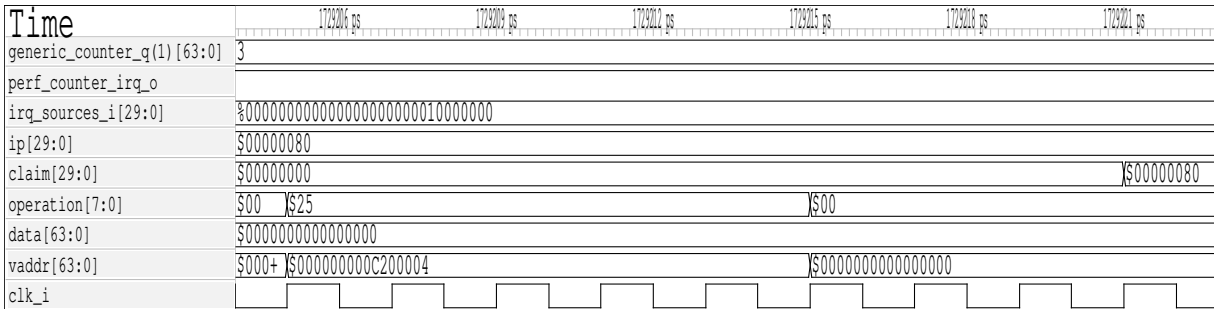

Figure 4.4: Handler claim notification.

4.4 Summary

This chapter outlines the design and implementation of a PES system for RISC-V, emphasizing its specification and integration into contemporary profiling tools. The PES system is defined with specific objectives and requirements, including the ability to sample the processor's state based on predefined performance counter thresholds. Key components of the architecture include new CSRs to manage buffer locations and threshold values. A hardware-based comparator is used to trigger interrupts when performance counters reach their specified thresholds. Communication between kernel-level software, such as *perf_events*, and the hardware occurs through an OpenSBI interface, respecting the RISC-V CSRs Machine privilege level. The chapter also discusses an SBI-like interface extension, external interrupt control, and the implementation of an external interrupt handler. A test program is used to validate the PES system's basic functionality for profiling software, showing its potential to standardize and integrate PES into RISC-V-based computers as a valuable tool for software profiling.

In summary, the chapter presents a comprehensive overview of the PES system's design and implementation for RISC-V, covering its specification, architecture, and practical validation. It highlights the system's potential to serve as a standardized and integrated solution for profiling software in RISC-V-based computing environments, offering valuable insights into its development and usage.

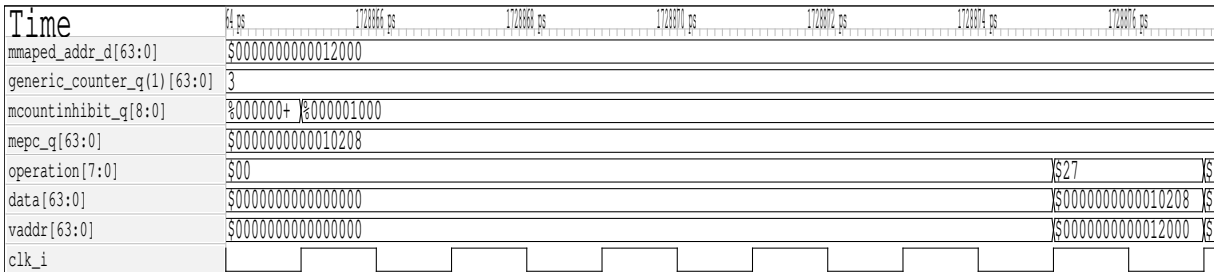

Figure 4.5: Counter inhibition and sampling of the MEPC.

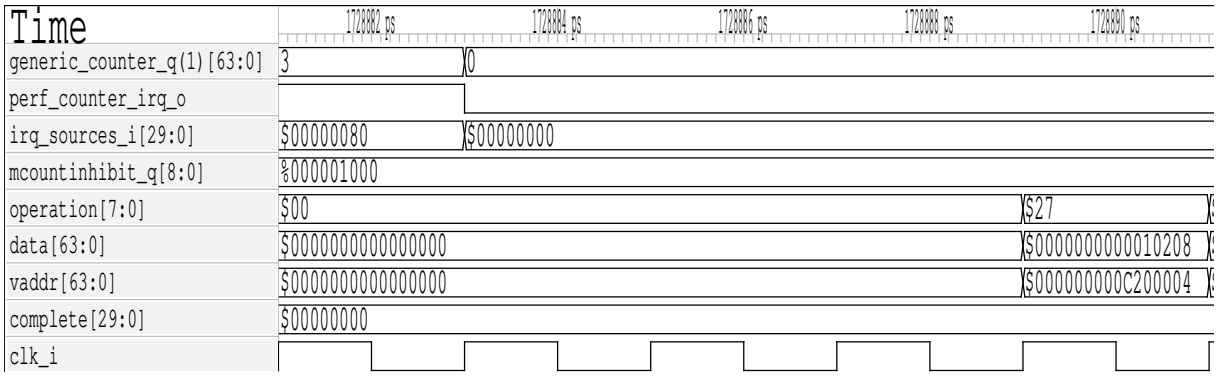

Figure 4.6: Counter reset.

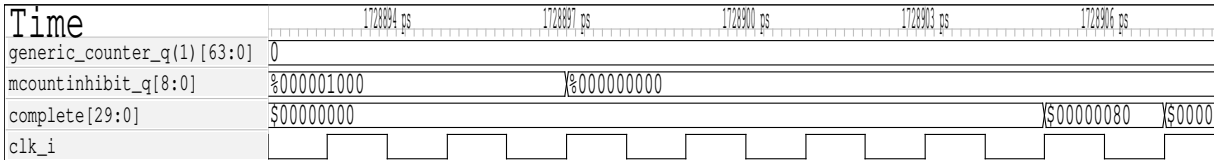

Figure 4.7: Counting resumed and PLIC complete signal.

CHAPTER 5

Conclusion

5.1 Contributions Achieved

This work set out with the objective of enabling RISC-V software development by introducing higher-level access to the RISC-V HPM and proposing a new development path to this still-simple facility by designing a proof-of-concept for PES functionality.

Higher-level access was achieved by porting the PAPI library to RISC-V and giving it support the SiFive Unmatched board in a modular way and following the principles of the already existing repository code to facilitate future integration into the main project and the introduction of any RISC-V processor that is compatible with the current HPM specification.

The implemented version of PAPI is fully usable and introduces only a small amount of overhead in comparison with directly using the *perf_events* subsystem, which is much more laborious due to the necessity of knowing the event encodings and all the necessary configuration options. Furthermore, the overhead was shown to be associated with *PAPI* initialization, which is only counted once per program execution. The work developed in this implementation of *PAPI* was published in:

- Joao Mario Domingos et al. 'Supporting RISC-V Performance Counters Through Linux Performance Analysis Tools'. In: *2023 IEEE 34th International Conference on Application-specific Systems, Architectures and Processors (ASAP)*. 2023, pp. 94–101. DOI: 10.1109/ASAP57973.2023.00027

In regards to the newly proposed PES functionality, the complete implementation plan was, unfortunately, not completed. Nevertheless, the facilities and software handlers implemented were enough to demonstrate the basic functionality of sampling the core state by using a programmable event as the sampling rate and storing the corresponding information to memory. With this in mind, it is possible to say the objective of presenting a proof-of-concept for PES functionality was successful.

The work developed for this thesis has been shown to provide developers with easier access to the HPM, not available until now and also presented a new path of development for the HPM specification that could further help the porting of software to RISC-V.

5.2 Future Work

The possible future works launched by this thesis could go in several directions.

The *PAPI* library modifications could be submitted to be merged into the main project. Support for new RISC-V processors could be added, like the ones developed by the EPI project.

The PES functionality introduced is but a very crude proof-of-concept. It should be further developed, adjusting the original design presented if so is deemed necessary, and compiled into a RISC-V extension proposition that could be submitted to the RISC-V Foundation. Bringing the PES system to a complete state would require new modifications: to the Hardware, to introduce the not yet implemented registers already deemed necessary for the full system; to the Kenel, to implement complete handlers instead of simplified versions of them; to the OpenSBI layer, to allow full communication with the new registers; to the *perf_events* driver, to allow seamless control of the system without the need for the user to manually

program and initiate the system; and to the *PAPI* library, to be able to correctly interface with *perf_events* when attempting to program the HPM to do event-based sampling.

Other possible works derived from the experience developing this work could be to develop new tools, built upon the *PAPI* library to analyze more specific metrics, such as the energy efficiency of certain operations, for example.

Bibliography

[1] Shirley Browne et al. 'A portable programming interface for performance evaluation on modern processors'. In: *The international journal of high performance computing applications* 14.3 (2000), pp. 189–204.

[2] Hongwei Cui et al. 'A Hardware-Software Cooperative Interval-Replaying for FPGA-based Architecture Evaluation'. In: *2023 Design, Automation & Test in Europe Conference & Exhibition (DATE)*. 2023, pp. 1–2. DOI: 10.23919/DATE56975.2023.10137049.

[3] Joao Mario Domingos, Pedro Tomás and Leonel Sousa. 'Supporting RISC-V Performance Counters through Performance Analysis Tools for Linux (Perf)'. In: *5th Workshop on Computer Architecture Research with RISC-V (CARRV 2021)*. June 2021.

[4] Joao Mario Domingos et al. 'Supporting RISC-V Performance Counters Through Linux Performance Analysis Tools'. In: *2023 IEEE 34th International Conference on Application-specific Systems, Architectures and Processors (ASAP)*. 2023, pp. 94–101. DOI: 10.1109/ASAP57973.2023.00027.

[5] Jack Dongarra et al. 'Using PAPI for hardware performance monitoring on Linux systems'. In: *Conference on Linux Clusters: The HPC Revolution*. Vol. 5. Linux Clusters Institute. 2001.

[6] Stephane Eranian. *libpfm4*. URL: https://sourceforge.net/p/perfmon2/libpfm4/ci/master/tree/ (visited on 26/12/2022).

[7] Brendan Gregg. *Systems performance: enterprise and the cloud*. 2nd ed. Pearson Education, 2020. Chap. 13.

[8] Christian Helm and Kenjiro Taura. 'PerfMemPlus: A Tool for Automatic Discovery of Memory Performance Problems'. In: *High Performance Computing*. Ed. by Michèle Weiland et al. Cham: Springer International Publishing, 2019, pp. 209–226. ISBN: 978-3-030-20656-7.

[9] *Intel 64 and IA-32 Architectures Software Developer's Manual - Volume 3B: System Programming Guide*. Version Order Number 325384. Chap. 20.

[10] RISC-V International. *RISC-V Platform-Level Interrupt Controller Specification*.

[11] Michael Kerrisk. *PERF_EVENT_OPEN Linux Programmer's Manual*. URL: https://www.man7.org/linux/man-pages/man2/perf_event_open.2.html (visited on 26/12/2022).

[12] Wendy Korn, Patricia J Teller and G Castillo. 'Just how accurate are performance counters?' In: *Conference Proceedings of the 2001 IEEE International Performance, Computing, and Communications Conference (Cat. No. 01CH37210)*. IEEE. 2001, pp. 303–310.

[13] Xu Liu and John Mellor-Crummey. 'A Tool to Analyze the Performance of Multithreaded Programs on NUMA Architectures'. In: *SIGPLAN Not.* 49.8 (Feb. 2014), pp. 259–272. ISSN: 0362-1340. DOI: 10.1145/2692916.2555271. URL: https://doi.org/10.1145/2692916.2555271.

[14] Liang Luo et al. 'LASER: Light, Accurate Sharing dEtection and Repair'. In: *2016 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. 2016, pp. 261–273. DOI: 10.1109/HPCA.2016.7446070.

[15] J.M. May. 'MPX: Software for multiplexing hardware performance counters in multithreaded programs'. In: *Proceedings 15th International Parallel and Distributed Processing Symposium. IPDPS 2001*. 2001, 8 pp.-. DOI: 10.1109/IPDPS.2001.924955.

[16] *PAPI Documentation, papi_common_strings.h*. URL: https://icl.utk.edu/projectsdev/papi/docs/d1/d72/papi__common__strings_8h_source.html (visited on 16/10/2023).

[17] *PolyBenchC-4.2.1*. URL: https://github.com/MatthiasJReisinger/PolyBenchC-4.2.1 (visited on 16/10/2023).

[18] *RISC-V Proxy Kernel and Boot Loader*. URL: https://github.com/riscv-software-src/riscv-pk (visited on 18/10/2023).

[19] Probir Roy et al. 'Lightweight Detection of Cache Conflicts'. In: *Proceedings of the 2018 International Symposium on Code Generation and Optimization*. CGO 2018. Vienna, Austria: Association for Computing Machinery, 2018, pp. 200–213. ISBN: 9781450356176. DOI: 10.1145/3168819. URL: https://doi.org/10.1145/3168819.

[20] Muhammad Aditya Sasongko et al. 'ComDetective: A Lightweight Communication Detection Tool for Threads'. In: *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. SC '19. Denver, Colorado: Association for Computing Machinery, 2019. ISBN: 9781450362290. DOI: 10.1145/3295500.3356214. URL: https://doi.org/10.1145/3295500.3356214.

[21] Muhammad Aditya Sasongko et al. 'Precise Event Sampling on AMD vs Intel: Quantitative and Qualitative Comparison'. In: *IEEE Transactions on Parallel and Distributed Systems* (2023), pp. 1–15. DOI: 10.1109/TPDS.2023.3257105.

[22] Muhammad Aditya Sasongko et al. 'ReuseTracker: Fast Yet Accurate Multicore Reuse Distance Analyzer'. In: *ACM Trans. Archit. Code Optim.* 19.1 (Dec. 2021). ISSN: 1544-3566. DOI: 10.1145/3484199. URL: https://doi.org/10.1145/3484199.

[23] *SiFive U74-MC Core Complex Manual*. Version 21G3.02.00. SiFive, Inc.

[24] Richard M. Stallman and the GCC Developer Community. *Using the GNU Compiler Collection*. GNU Press, 2023. URL: https://gcc.gnu.org/onlinedocs/gcc.pdf (visited on 13/10/2023).

[25] Dan Terpstra et al. ‘Collecting Performance Data with PAPI-C’. In: *Tools for High Performance Computing 2009*. Ed. by Matthias S. Müller et al. Berlin, Heidelberg: Springer Berlin Heidelberg, 2010, pp. 157–173. ISBN: 978-3-642-11261-4.

[26] Andrew Waterman and Krste Asanovič, eds. *The RISC-V Instruction Set Manual, Volume I: Unprivileged ISA*. Version 20191213. RISC-V Foundation. Dec. 2019.

[27] Andrew Waterman, Krste Asanovič and Hauser John, eds. *The RISC-V Instruction Set Manual, Volume II: Privileged Architecture*. Version 20211203. RISC-V International. Dec. 2021.

[28] Vince Weaver. ‘New features in the PAPI 5.0 release’. In: *University of Tennessee, Tech. Rep* (2012).

[29] Vincent M Weaver et al. ‘Advanced hardware profiling and sampling (PEBS, IBS, etc.): creating a new PAPI sampling interface’. In: *Technical Report UMAINE-VMWTR-PEBS-IBS-SAMPLING-2016-08. University of Maine, Tech. Rep.* (2016).

[30] Vincent M Weaver. ‘Self-monitoring overhead of the Linux perf_ event performance counter interface’. In: *2015 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*. IEEE. 2015, pp. 102–111.

[31] Vincent M. Weaver et al. ‘PAPI 5: Measuring power, energy, and the cloud’. In: *2013 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*. 2013, pp. 124–125. DOI: 10.1109/ISPASS.2013.6557155.

[32] Jingyi Xu et al. ‘Memory-Efficient Hardware Performance Counters with Approximate-Counting Algorithms’. In: *2021 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*. 2021, pp. 226–228. DOI: 10.1109/ISPASS51385.2021.00041.

[33] Florian Zaruba and Luca Benini. *The Cost of Application-Class Processing: Energy and Performance Analysis of a Linux-Ready 1.7-GHz 64-Bit RISC-V Core in 22-nm FDSOI Technology*. Version 2.0.4. July 2019. DOI: 10.1109/TVLSI.2019.2926114. URL: https://github.com/openhwgroup/cva6.