Performance Monitoring and Event-based Sampling for RISC-V

Tiago Alfredo Lopes Rocha
Instituto Superior Técnico
University of Lisbon, Portugal
tiagolopesrocha@tecnico.ulisboa.pt

***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 performance monitor (HPM) and offers support for the Linux *perf_event* 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 (PES) 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 PES was implemented on a CVA6 processor.**

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

***Index Terms*—RISC-V Processors, Performance Monitoring, Precise Event Sampling, PAPI**

I. Introduction

The introduction of RISC-V as a royalty-free Instruction Set Architecture (ISA) has significantly changed the landscape of microprocessor development. While there was a rapid adoption of RISC-V-based micro-controllers, the use of RISC-V-based systems is now becoming widespread, with multiple recent initiatives trying to develop high-performance processors.

However, naively porting prominent workloads to new computing platforms often results in limited computing performance. Naturally, many factors have to be taken into account when justifying such performance limitations, including poor software implementations that result in high computational complexities or inefficient data structures, ineffective or poor cache usage, or processor stalls due to front-end or back-end bottlenecks. This is a particular problem when considering the use of emerging technologies based on RISC-V, for which most mainstream performance monitoring tools only offer limited or no support.

When tackling application optimization, one must first monitor its execution, identify the main performance bottlenecks, and tailor the software to best fit the underlying hardware. Naturally, this procedure can hardly be performed by solely using performance metrics (e.g., execution time or clock cycles), as multiple factors come into play when mapping the software to a modern computing system (e.g., in- vs out-of-order execution engines, pipeline stages, execution ports and corresponding latencies, re-order buffers, load/store queues, cache organization, etc). Consequently, the capture and analysis of detailed performance metrics to allow in-depth architecture modelling and optimization procedures (e.g. [1], [2]) becomes a fundamental requirement.

While Intel and ARM provide proprietary performance monitoring solutions [3]–[6], which allow software developers to take the ultimate advantage of their hardware, RISC-V is still underdeveloped in this regard, having recently had its HPM supported and accessible through the *perf_event* subsystem, [7]. Nevertheless, no higher-level access to the performance counters is yet available such as through Performance Application Programming Interface (PAPI) [8] as well as little attention being given to PES facilities and how they could be introduced in the RISC-V specification, having only been briefly mentioned as a means to other HPM uses, [9].

To improve and extend the performance analysis tools for RISC-V in Linux, the following additions and modifications are herein presented:

- 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 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.

Considering the multiple available RISC-V implementations, and their dissociated performance monitoring hardware implementations, it was considered to copy strategies such as

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.

backward compatibility and implementation features discovery. Even so, it is not possible to encompass all the details and specializations of all the available implementations and RISC-V specifications at once. Therefore, it was determined specification version 1.11 [10] was the primarily supported target, and to make the software flexible to support the majority of implementations. Although RISC-V privileged ISA is currently on version 1.13, no changes to the performance monitoring specifications have been made since version 1.11.

II. RISC-V Performance Monitoring

RISC-V ISA has received continuous interest and development, from wider software compatibility to an increasing number of hardware implementations [11]–[15]. Alongside the software and hardware, the RISC-V specification also shows a persistent evolution, driven by the growing requirements of the RISC-V ecosystem. Since RISC-V privileged specification version 1.7 [16], a minimal performance monitoring interface was defined. From then, the specification has introduced additional counters and other necessary features for access control and event multiplexing.

A. Early specifications

The first RISC-V privileged specification, version 1.7, introduced the first attempt at monitoring the core's performance. Its implementation, supporting three fixed counters (Cycle, Time and Retired Instructions (CTI)), allowed for baseline performance monitoring of a RISC-V processor, enough for calculating the Instructions per Clock (IPC) metric. With v1.7, the Performance Monitoring Unit (PMU) had all the counters accessible at user and supervisor privilege levels, lacking control over non-privileged access.

Version 1.9 [17] introduced control over the privileged counter accesses. A counter-enable mask was introduced employing three registers accessible only at machine-level, and imposing read control over the CTI counters at the hypervisor, supervisor, and user levels. In addition, v1.9 introduced a set of delta counters: a counter which keeps the difference between each of the lower privilege counters and the respective machine-level counter (e.g., mstime_delta=stime-mtime). These delta counters were removed after version 1.9. At the time, RISC-V performance monitoring was still limited to the set of three fixed registers, without support for general-purpose or fixed-event performance monitoring registers.

B. Configurable events and counters

Support for 29 additional performance monitor registers was introduced with version 1.10. The Hardware Performance Monitor (HPM) counters, ranging from `hpmcounter3` to `hpmcounter31`, can be individually configured by setting an event identifier in the corresponding `hpmevent` registers, a set of XLEN-bits registers (e.g., XLEN = 64 in a 64-bit implementation). This amounts to, virtually, 2^{64} selectable events for a single register, a value that surpasses any realistic implementation and provides an overly large design flexibility. The RISC-V specification states that the number, width, and supported events of each `hpmcounter` is platform-/implementation-specific. Even so, HPM counters are limited to a maximum width of 64 bits.

When setting the `hpmevent` registers, event 0 is considered as the null event, and both the event configuration and the counter registers can be hardwired to 0, indicating that no event counting can occur. Each event counter (`hpmcounter#`) is writable in a WARL (write any, read logical) scheme, allowing for each counter to be individually reset/set [18].

C. Additional Features and Future Objectives

In version 1.11 [10], individual counter inhibition (i.e., stop counting) was introduced, allowing the software to atomically sample events. This is accomplished through the introduction of the `mcountinhibit` register, where each of the 32 bits can be set to inhibit the respective HPM counter.

Current specifications suggest that future versions could include support for common event standardization, to count ISA-level metrics, such as executed floating-point or integer instructions. Similarly, some common and widely supported micro-architectural metrics could be standardized (e.g., L1 instruction cache misses). Another feature that may appear in future specifications is the support for counter overflow interrupts, allowing the software to accurately count events that overflow the respective counters at a faster pace than the event sampling occurs. Nevertheless, the occurrence of such continuous overflowing is unlikely, considering implementations with 64-bit counters.

D. Access to the HPM Within Linux Systems

With the release of Linux kernel version 2.6.31, the *perf_event* subsystem was introduced [19] as the default interface for accessing hardware performance counters. It was designed with functionality and abstraction in mind to make it simple to operate.

perf_event uses the `perf_event_open()` [20] 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` [20] structure and counter 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.

In the context of this work, *perf_event* is used to interface with the performance monitoring counters of RISC-V processors. Full support for every event mappable in the SiFive U7 processor core was made available on *perf_event* by [7].

E. Discussion

Currently, the RISC-V HPM is still significantly less complex than the x86 counterpart [21] and not comparable to the dedicated performance analysis tools like ARM's Coresight and Intel's PCM-based monitoring solutions [3]–[6]. Even so, the RISC-V HPM specification is a flexible generic performance monitoring solution. Furthermore, by being open-source, it allows any degree of implementation freedom.

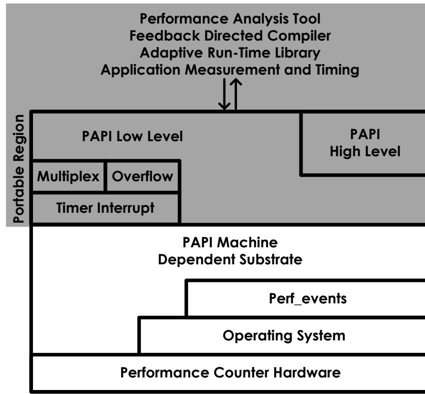

Fig. 1. PAPI architecture, modified from [23].

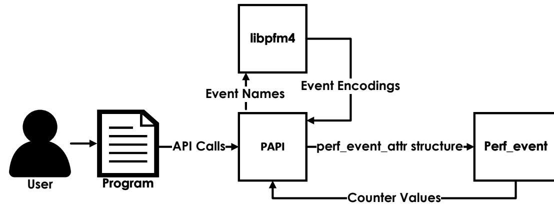

Fig. 2. Basic structure for hardware performance counting using PAPI.

Considering the current state of the RISC-V privileged specification, section III proposes a higher-level approach to monitoring the performance counters in RISC-V through the widely established PAPI Library. A proof-of-concept of how a PES system might work in RISC-V systems is also presented.

III. Proposed Approach

To complement the already existing, low-level performance monitoring tools, the PAPI library was extended to allow users to access performance measurements more easily on RISC-V. When considering the profiling of microprocessors for performance studies, this means offering a higher-level API to access hardware performance counters.

A. PAPI Library Modifications

Within the underlying structure of the PAPI library (often known as *substrate*), the communication with the HPM has to be done through another program or subsystem that has lower-level access to the hardware. Although PAPI was released before *perf_event*, this subsystem of the Linux kernel has been adopted as the main mechanism (used by the PAPI library) to access the processor hardware performance counters. This architecture is shown in Figure 1. In its current form, PAPI uses the libpfm4 [22] library to translate event names (in human-readable string form) to event encodings (determined by the hardware vendor) and build the control structure necessary to use the *perf_event* library (as illustrated in Figure 2).

Accordingly, to create the aimed support for RISC-V architectures in the PAPI library, it is first necessary to define a machine-specific substrate (see Figure 1). To do so, all supporting functions with inline assembly calls (to make direct use of RISC-V-specific instructions) and all necessary system parameters were defined. These include *i)* the program counter context location in the operating system filesystem; *ii)* a dummy function to read the Cycles control register (which, due to the RISC-V specification, is not accessible in user-level applications and, as such, returns 0); and *iii)* a function to allow PAPI to create a memory fence.

With such a baseline for RISC-V compatibility, the PMU of any RISC-V processor can be described. To do so, it is necessary to create a new pfmlib_pmu_t type structure instance to hold the processor information. It includes the name and number of programmable hardware event counters, the list that maintains all available events and the corresponding pointers to functions that allow PAPI to manipulate the events (e.g., encoding an event or iterating over event lists). The list of countable events from the processor's HPM is stored on a riscv_entry_t structure list, which holds the name, short description, and event code. The stored code matches the word that is passed to *perf_event* to program the event.

Currently, the PAPI library maintains a pre-defined list of more than one hundred events that are commonly available in processors' HPMs, named *PAPI preset events*. The mapping of these preset events to the events countable by the processor is done through a papi_events file (in CSV format). Additional events available in the HPM (that do not match preset events) are automatically included from libpfm4 listing as *native events* specific to each architecture. The event mapping itself can either be done through direct 1-to-1 mapping or by combining multiple events. The latter allows counting multiple events in parallel and calculating a derived metric from these counted events. Naturally, the complexity of these metrics is limited by the number of physically available programmable counters, although this can be sometimes overcome by the multiplexing facilities of PAPI.

B. 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, [24].

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's *PEBS*, AMD's *IBS*, IBS's *MRK* and ARM's *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 [9] as a means to other HPM uses.

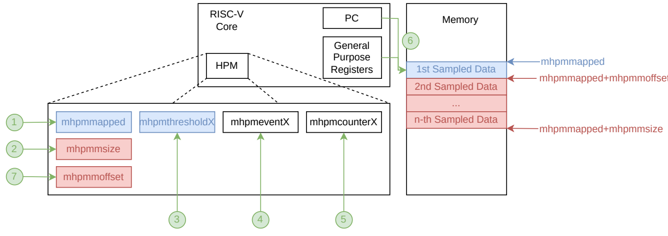

Fig. 3. Precise Event Sampling on RISC-V system architecture. Parts in white already existed, parts in blue were implemented, and parts in red were not implemented.

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 system architecture specified could be represented as in figure 3 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 `mhpmcounter4` 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.

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

IV. Performance Monitoring Validation

To validate the PAPI library extension on an off-the-shelf RISC-V system, we implement PAPI on a SiFive Unmatched board. Then, to conclude the basic functionality of the PES system, a modified CVA6 processor was simulated.

A. Methodology

The U74-MC processor, contained in the SiFive Unmatched board, supports the Cycles and Instructions Retired fixed-event counters (counters `mcycle` and `minstret`, respectively) required by the RISC-V privileged specification [16]. Additionally, it supports two programmable-event counters

TABLE I
Performance events for the SiFive U74-MC platform.

`mhpmeventX` `[7:0]`	**Bit**	**Event description**
Instruction Commit (`0x0`)	8	Exception taken
	9	Integer load instruction
	10	Integer store instruction
	11	Atomic memory operation
	12	System instruction
	13	Integer arithmetic instruction
	14	Conditional branch
	15	JAL instruction
	16	JALR instruction
	17	Integer multiplication instruction
	18	Integer division instruction
	19	Floating-point load instruction
	20	Floating-point store instruction
	21	Floating-point addition
	22	Floating-point multiplication
	23	Floating-point fused multiply-add
	24	Floating-point division or square-root
	25	Other floating-point instruction
Microarchitecture (`0x1`)	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 (`0x2`)	8	Instruction cache miss
	9	Data cache miss / memory-mapped I/O access
	10	Data cache write-back
	11	Instruction TLB miss
	12	Data TLB miss
	13	UTLB miss

(`mhpmcounter3` and `mhpmcounter4`), which selection is performed by configuring CSR registers `mhpmevent3` and `mhpmevent4`. Events are divided into 3 groups, each focused on a different architectural component, namely: instruction commit (`mhpmeventX[7:0]=0x0`), microarchitecture (`mhpmeventX[7:0]=0x1`) and memory system (`mhpmeventX[7:0]=0x2`). Selection of the actual event within each group is made by setting upper bits of the `mhpmeventX[7:0]` registers as specified in Table I [25].

Hence, the PAPI implementation was extended to fully describe the U74-MC processor PMU and interface it with *perf_event* to map it to the corresponding HPM events. With the preset and native events implemented and matched to the *perf_event* HPM events, the counting of every event listed by SiFive on the U74-MC manual [25] is supported. [1]

To assess the developed framework, specific micro-benchmarks were first developed and executed to measure the execution overheads for Perf and PAPI calls. Then an evaluation of both Perf and PAPI was conducted using a general matrix multiplication kernel (GEMM) with dataset (matrices A, B, and C) dimensions scaled so that the kernel

[1]The *perf_event* source code is available at https://github.com/hpc-ulisboa/RISC-V-Perf-Events-Unmatched and the PAPI is available at https://github.com/hpc-ulisboa/RISC-V-PAPI

TABLE II
MEASURED EXECUTION TIME OVERHEADS (AVERAGE) OF PAPI LIBRARY CALLS IN RELATION TO *perf_event* CALLS.

Function Call	*perf_event*	PAPI Library
Initialization	$57.98\mu s$	$81200\mu s$
Start count	$11.67\mu s$	$947.11\mu s$
Read count	$7.76\mu s$	$13.95\mu s$
Stop count and read	$15.89\mu s$	$19.41\mu s$
Termination	$15.79\mu s$	$1190\mu s$
Total	$100.24\mu s$	$83360\mu s$

TABLE III
PAPI LIBRARY AND *perf_event* EVENT MONITORING COMPARISON WITH GEMM BENCHMARK.

Exec. Time	Event ID	*perf_event* (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%

duration is close to 10 (A: 450×550, B: 550×500, and C: 450×500), 30 (A: 650×750, B: 750×700, and C: 650×700), and 60 (A: 850×950, B: 950×900, and C: 850×900) seconds, on the SiFive Unmatched system.

To understand the impact of the added hardware for PES 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.

Resource utilization reports show that the implementation with the PES facilities uses 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.

The functionality of the PES system was demonstrated with a simple program that sets the sampling rate as 3 floating point instructions and then proceeds to run 15 floating point adds. Each sample stores the `MEPC` contents to a predefined memory position.

B. Toolchain Validation

To fully validate and profile the implemented *perf_event* and PAPI library extensions, the average total overhead for initialization, termination, and events start, read, and stop on both tools were first measured. To do so, the total time elapsed while running each step is defined with the `time.h` library `clock_gettime()` function. As detailed in Table II, the observed overheads for PAPI are higher than for *perf_event*, with the typical case accounting for 13.95µs for a PAPI event read (vs 7.76µs on *perf_event*) and 947µs for event start (vs 11.67µs on *perf_event*). 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, which makes a kernel call to configure the corresponding CSR registers via OpenSBI. 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_event*. Finally, there is also an initial overhead for library initialization and termination which (on average) takes 81.2ms and 1.19ms for PAPI, respectively, which compare with the 57.98µs and 15.79µs for *perf_event*.

Although the observed overheads are consistent with previous research [19], 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_event*, 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_event* functionalities.

The measurements obtained by using PAPI and *perf_event* are further validated by performing a detailed profiling of the GEMM kernel (configured for $10s$). This is done by measuring average counts for multiple events across 100 runs of the kernel. The obtained results (presented in Table III) show that the counts for deterministic events (e.g., floating-point stores, fused-multiply-add operations, etc.) present the same measured count using *perf_event* and PAPI. However, other events such as cache operations and cycles, show slight variations due to OS-related interference during the execution. By increasing the execution time of the GEMM kernel to $30s$ and $60s$, an OS interference mitigation is observed, decreasing variations from 2.37% to 1.58% (see Table III).

In regards to PES, the test showed that the system correctly sampled after 3 occurrences of the event and stored the desired information in a predefined memory location. It was also demonstrated that the interrupt handler correctly resets the system so further samplings can be performed.

V. CONCLUSIONS

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_event 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 [26].

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, all while having minimal impact on resource utilization and operating frequency. 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.

REFERENCES

[1] D. Marques, A. Ilic, Z. A. Matveev, and L. Sousa, "Application-driven cache-aware roofline model," *Future Generation Computer Systems*, vol. 107, pp. 257–273, 2020.

[2] J. Alcaraz, A. Sikora, and E. César, "Hardware counters' space reduction for code region characterization," in *Euro-Par 2019: Parallel Processing: 25th International Conference on Parallel and Distributed Computing*, pp. 74–86, Springer, 2019.

[3] A. Kleen and B. Strong, "Intel ® Processor Trace on Linux," *Tracing Summit*, 2015.

[4] A. P. Su *et al.*, "Multi-core software/hardware co-debug platform with ARM CoreSight™, on-chip test architecture and AXI/AHB bus monitor," in *International Symposium on VLSI Design, Automation and Test*, pp. 1–6, Apr. 2011.

[5] S. M. A. Zeinolabedin, J. Partzsch, and C. Mayr, "Real-time Hardware Implementation of ARM CoreSight Trace Decoder," *IEEE Design Test*, vol. 38, pp. 69–77, Feb. 2021.

[6] Y. Lee *et al.*, "Using CoreSight PTM to Integrate CRA Monitoring IPs in an ARM-Based SoC," *ACM Transactions on Design Automation of Electronic Systems*, vol. 22, pp. 52:1–52:25, Apr. 2017.

[7] J. M. Domingos, P. Tomas, and L. Sousa, "Supporting risc-v performance counters through performance analysis tools for linux (perf)," 2021.

[8] H. Jagode, A. Danalis, H. Anzt, and J. Dongarra, "PAPI software-defined events for in-depth performance analysis," *The International Journal of High Performance Computing Applications*, vol. 33, pp. 1113–1127, Nov. 2019.

[9] H. Cui, S. Liang, Y. Cui, W. Zhang, H. Zhan, C. Yang, X. Liu, and X. Cheng, "A hardware-software cooperative interval-replaying for fpga-based architecture evaluation," in *2023 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, pp. 1–2, 2023.

[10] A. Waterman and K. Asanović, "The RISC-V Instruction Set Manual, Volume II: Privileged Architecture, Version 20190608-Priv-MSU-Ratified," *RISC-V Foundation*, June 2019.

[11] J. Zhao, B. Korpan, A. Gonzalez, and K. Asanovic, "SonicBOOM: The 3rd Generation Berkeley Out-of-Order Machine," *Workshop on Computer Architecture Research with RISC-V (CARRV)*, p. 7, 2020.

[12] J. Balkind *et al.*, "OpenPiton+Ariane: The First Open-Source, SMP Linux-booting RISC-V System Scaling From One to Many Cores," in *Workshop on Computer Architecture Research with RISC-V (CARRV)*, pp. 1–6, 2019.

[13] E. Matthews and L. Shannon, "TAIGA: A new RISC-V soft-processor framework enabling high performance CPU architectural features," in *2017 27th International Conference on Field Programmable Logic and Applications (FPL)*, pp. 1–4, Sept. 2017.

[14] F. Zaruba and L. 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," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 27, pp. 2629–2640, Nov. 2019.

[15] C. Chen *et al.*, "Xuantie-910: A Commercial Multi-Core 12-Stage Pipeline Out-of-Order 64-bit High Performance RISC-V Processor with Vector Extension : Industrial Product," in *2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA)*, pp. 52–64, May 2020.

[16] A. Waterman, Y. Lee, R. Avizienis, D. A. Patterson, and K. Asanovic, "The RISC-V Instruction Set Manual, Volume II: Privileged Architecture, Document Version 1.7," *EECS Department, UC Berkeley, Tech. Rep. UCB/EECS-2015-49*, vol. 49, 2015.

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

[18] A. Waterman, Y. Lee, D. A. Patterson, and K. Asanovic, "The RISC-V Instruction Set Manual, Volume I: User-Level ISA Version 2.1," *EECS Department, UC Berkeley, Tech. Rep. UCB/EECS-2016-118*, vol. 118, 2016.

[19] V. 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)*, pp. 102–111, IEEE, 2015.

[20] M. Kerrisk, "Perf_event_open linux programmer's manual."

[21] Intel, "Intel® 64 and IA-32 Architectures Developer's Manual: Vol. 3B," tech. rep., 2016.

[22] S. Eranian, "libpfm4."

[23] S. Browne *et al.*, "A portable programming interface for performance evaluation on modern processors," *The international journal of high performance computing applications*, vol. 14, no. 3, pp. 189–204, 2000.

[24] M. A. Sasongko, M. Chabbi, P. H. J. Kelly, and D. Unat, "Precise event sampling on amd vs intel: Quantitative and qualitative comparison," *IEEE Transactions on Parallel and Distributed Systems*, pp. 1–15, 2023.

[25] SiFive, Inc., *SiFive U74-MC Core Complex Manual*.

[26] J. M. Domingos, T. Rocha, N. Neves, N. Roma, P. Tomás, and L. Sousa, "Supporting risc-v performance counters through linux performance analysis tools," in *2023 IEEE 34th International Conference on Application-specific Systems, Architectures and Processors (ASAP)*, pp. 94–101, 2023.