Supporting RISC-V Performance Counters Through Linux Performance Analysis Tools

Joao Mario Domingos
INESC-ID, Instituto Superior Técnico
University of Lisbon, Portugal
joao.mario@tecnico.ulisboa.pt

Tiago Rocha
INESC-ID, Instituto Superior Técnico
University of Lisbon, Portugal
tiagolopesrocha@tecnico.ulisboa.pt

Nuno Neves
INESC-ID, Instituto Superior Técnico
University of Lisbon, Portugal
nuno.neves@inesc-id.pt

Nuno Roma
INESC-ID, Instituto Superior Técnico
University of Lisbon, Portugal
nuno.roma@inesc-id.pt

Pedro Tomás
INESC-ID, Instituto Superior Técnico
University of Lisbon, Portugal
pedro.z.tomas@tecnico.ulisboa.pt

Leonel Sousa
INESC-ID, Instituto Superior Técnico
University of Lisbon, Portugal
las@inesc-id.pt

***Abstract*—Increased attention to RISC-V open Instruction Set Architecture (ISA), a base ISA with a variety of optional extensions, 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), 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, a comprehensive set of extensions and modifications to the Performance analysis tools for Linux (perf/perf_events) are proposed in this paper, and a PAPI library interface is presented. These new extensions comprise not only the Linux kernel but also the OpenSBI interface, and aim to achieve full support for the RISC-V performance monitoring specification. The conducted testing and evaluation were carried out on a HiFive Unmatched board and on a CVA6 core, but the proposed extensions, and the corresponding implementation, are easily portable to other systems.**

***Index Terms*—RISC-V Processors, RISC-V-Based Accelerators, Performance Monitoring, Perf, 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 modeling 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 dependent on custom/vendor-specific solutions, with no complete support for common performance monitoring software tools, such as Performance Application Programming Interface (PAPI) [7] or the Linux kernel monitoring tool Perf [8]. Currently, only fixed counters are supported without event configuration, and no control over the counters is provided (e.g., pausing, enabling, disabling).

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

- An OpenSBI extension for privileged interaction with the RISC-V performance monitoring hardware;
- Support for the latest RISC-V HPM specification in the Linux Kernel through perf_events;

This work was partially supported by national funds through Fundação para a Ciência e a Tecnologia (FCT) under projects UIDB/50021/2020, 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.

- Support for configurable RISC-V events in Perf and in the PAPI library;
- Support for multiple platforms with distinct sets of events.

Considering the multiple available RISC-V implementations, and their dissociated performance monitoring hardware implementations, we consider coping strategies such as 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, we set specification version 1.11 [9] as our primarily supported target, and try to make the software flexible to support the majority of implementations. We note that although RISC-V previleged 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 [10]–[14]. 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 [15], 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 [16] introduced control over the privileged counter accesses. A counter-enable mask was introduced by means of 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 [17].

C. Additional Features and Future Objectives

In version 1.11 [9], 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. Discussion

Currently, the RISC-V HPM is still significantly less complex than the x86 counterpart [18] 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. Considering the current state of the RISC-V privileged specification, section III proposes an approach to monitor the performance counters in RISC-V through the widely established Linux Perf software framework and details extensions to the PAPI Library.

III. Proposed Approach and new extensions

The ground for the proposed approach is the current Linux Perf implementation, developed after the RISC-V privileged specification version 1.10. This Perf implementation provides basic support for adding, deleting, starting, and stopping

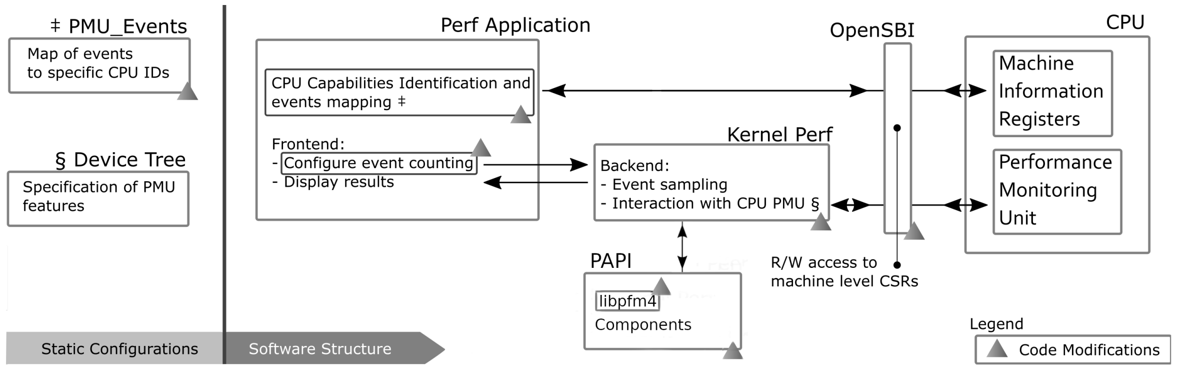
‡ PMU_Events
Map of events to specific CPU IDs
§ Device Tree
Specification of PMU features
Static Configurations
Software Structure
Perf Application
CPU Capabilities Identification and events mapping ‡
Frontend:
- Configure event counting
- Display results
Kernel Perf
Backend:
- Event sampling
- Interaction with CPU PMU §
PAPI
libpfm4
Components
OpenSBI
R/W access to machine level CSRs
CPU
Machine Information Registers
Performance Monitoring Unit
Legend
Code Modifications

Fig. 1. Overview of the system software structure.

software-side events. However, a significant limitation still exists concerning its inability to write to counters and event configuration registers, In particular, such writes require machine-level privilege, not available without a dedicated OpenSBI extension. Due to this limitation, currently, it is not possible to configure events in a specific counter, significantly limiting Perf to the fixed set of CTI counters [19].

Considering these limitations, our proposal starts by providing a mechanism to write and read on machine-level privileged counters and registers, through the introduction of a new OpenSBI extension. Additionally, since the Linux performance monitoring system is divided into the kernel driver and the Perf application, the kernel Perf driver and the Perf tool were also modified. These two modules are connected through the `perf_event_open` system call, where the kernel driver samples the events from the HPM counters.

Finally, to complement the offered performance monitoring extensions, we also extend the PAPI library to allow users to access performance measurements more easily. When considering the profiling of microprocessors for performance studies, this means offering a higher-level API to access hardware performance counters. An overview of the whole system, alongside the proposed modifications, is depicted in Figure 1.

A. OpenSBI HPM Extension

To define a new software-hardware interface and provide the required privileged access to machine-level registers, the HPM OpenSBI extension is herein adopted. The newly included OpenSBI functions are detailed in Table I, providing support for reading and writing operations over all the privileged registers defined in version 1.11 of the specification, namely:

- Generic Performance counters: `mcycle`, `mtime`, `minstret`
- Performance counters: `mhpmcounter#`
- Event configuration registers: `mhpmevent#`
- Lower privilege counter access enabler/disabler: `mcounteren`
- Inhibiting counter increment, `mcountinhibit`

Moreover, we also add support for reading and writing directly to the supervisor privilege `scounteren` register and to the user-level `hpmcounter` performance counters. While this is not a mandatory feature, considering that the Linux Kernel will have a sufficient privilege level, it allows the code to access the counters through a unified interface.

Considering the return structure of the OpenSBI handler for the RISC-V environmental call (*ecall*):

```
struct sbi_ret {
    long value;
    long error;
}
```

it was determined that each counter/register read could be executed in a single environment call. Taking into account that the return variable *value* is of type *long*, the variable size will be the same as the scalar registers implementation (i.e., 64 bits in a 64-bit implementation, and 32 bits in a 32-bits implementation). As such, for a 32-bit system, the process of reading any HPM counter must be unfolded in, at least, two calls, separately reading the lower and higher 32-bit portions of `mhpmcounter#`. Additional calls may eventually be necessary to compensate for the lower 32-bit counter overflow.

The proposed OpenSBI extension was named HPM, after the RISC-V Hardware Performance Monitor specification, and is identified by the value 0x48504d (as the direct conversion of ”HPM” from ASCII to hexadecimal). Currently, the HPM is experimental and is thus included in the experimental extension space with the corresponding ID (0x0848504d).

B. Linux Kernel Driver Modifications

The proposed software-level changes do not impact the majority of the Linux kernel source code. In particular, they are limited to specific parts, such as the Perf tool code and the Perf-related RISC-V kernel portion (arch/riscv/kernel).

As mentioned in the beginning of this section, the current RISC-V Perf kernel implementation only provides basic support for the RISC-V HPM specification, being restricted to fixed-event counters, i.e., each event can only be counted from a continuously running, non-stoppable, and non-changeable counter. Moreover, the only supported events are the cycle and instret counters, having no means to read other HPM counters. Hence, it is not compatible with the current RISC-V HPM specification, which allows for event configuration, counter inhibition and to control counter access. However, it does provide a basic structure to work on, which we extend in this work. We also built upon a Request for Comments kernel patch suggested by Zong Li [20], which introduced some support for the HPM counters through raw events, and device-tree bindings to support platform-specific hardware events (although

TABLE I
OPENSBI HPM EXTENSION FUNCTION CALLS DEFINITION.

HPM Function	**Output**	**Arguments**	**Errors**
hpm_get_mevent	event id	mHPM event id (3 - 31)	SBI_ERR_NOT_SUPPORTED: if register not implemented
hpm_set_mevent		mHPM event id, event id	SBI_ERR_NOT_SUPPORTED: if register not implemented
hpm_get_[m/u]counter	value	mHPM counter id (0 - 31)	SBI_ERR_NOT_SUPPORTED: if counter not implemented
hpm_set_[m/u]		mHPM counter id, value	SBI_ERR_NOT_SUPPORTED: if counter not implemented
hpm_get_[m/s]counteren	32-bits bitmask		SBI_ERR_NOT_SUPPORTED: if not implemented
hpm_set_[m/s]counteren		32-bits bitmask	SBI_ERR_NOT_SUPPORTED: if not implemented
hpm_get_mcountinhibit	32-bits bitmask		SBI_ERR_NOT_SUPPORTED: if not implemented
hpm_set_mcountinhibit		32-bits bitmask	SBI_ERR_NOT_SUPPORTED: if not implemented SBI_ERR_DENIED: on trying to inhibit time counter

such patch was not merged into the kernel). The introduced support for raw events allows the kernel driver to configure raw, not general, performance monitoring events, providing the necessary interfaces for adding, enabling, disabling, and removing events that are related to the HPM counters. In addition, the support for device-tree bindings allows for each platform to specify its own features, such as:

- Width of Base Counters (cycle, time, instret)
- Width of Event Counters (`mhpmevent#`)
- Number of Event Counters
- Hardware Event Map
- Hardware Cache Event Map

The Hardware Event Map is provided as a list of key-value pairs, where each pair matches a hardware event generic to Perf (key) to an implementation raw hardware event (value). An example is to use a key-value pair of `branch_misses: 0x05`, where the hardware event 0x05 matches the Perf branch_misses event. The Hardware Cache Event Map is a similar structure to Hardware Event Map. However, it maps events related to cache structures, such as L1 Read Misses or L2 Write Accesses.

The kernel Perf implementation is responsible for two independent procedures: event sampling (in a general way) and interaction with the CPU Performance Monitoring Unit (through OpenSBI, or directly). While any event is being sampled, the kernel driver will enable and start an event, proceed to take samples, and then, stop and disable the respective event. The interaction with the CPU PMU is handled at each of the mentioned steps. The introduced modifications include the interaction between the driver and the configuration registers (`mhpmevent#`, `mcounteren`), which are accomplished through OpenSBI, to provide machine-level access. Furthermore, and to decrease performance monitoring overheads, we also configure `mcounteren` to allow for supervisor-level read access, to get direct read access from the kernel to the HPM counters.

Additionally, some other changes had to be introduced in the internal procedure that matches the events to each counter. While an event could be matched to any counter (where an implementation would provide from 0 to 29 completely generic HPM counters), we alternatively propose that the mapping of each event is constrained to a specific set of counters (providing native support for hardware-friendly implementations where counters and associated events are constrained to specific pipeline stages). To achieve this, any raw event identifier contains two parameters: the event itself and the counter map. The event identifier is a numeric value to be interpreted by the CPU PMU through the `mhpmevent#` registers, not constrained to any particular logic (e.g., event classes and sub-classes). In contrast, the counter map is proposed as a 32-bit value. Each event has one and only one associated counter map, where each active bit indicates that the corresponding HPM counter is unable to count the event, as depicted in Figure 2. This parameter allows any event to be matched to any number and selection of HPM counters.

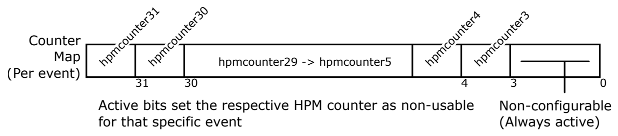
Counter Map (Per event)
hpmcounter31
hpmcounter30
hpmcounter29 -> hpmcounter5
hpmcounter4
hpmcounter3
31
30
4
3
0
Active bits set the respective HPM counter as non-usable for that specific event
Non-configurable (Always active)

Fig. 2. Counter Map for event to HPM counter matching.

C. *Perf Tool Modifications*

While the kernel driver is responsible for the actual sampling, the Perf tool acts as the front-end for event counting and provides a user interface for event listing (perf list), performance analysis (perf stat, monitor, record, report), and a set of simple benchmarks (perf bench). The modifications introduced by this work attempt to give support for raw events in a flexible and platform-specific way. In particular, they can be divided in CPU identification and events mapping.

To map the set of events that are available in a specific processor, system, or platform, we need to identify which CPU is executing Perf. According to the RISC-V ISA and OpenSBI specifications, each RISC-V implementation has a publicly available architecture ID [21], that is readable through an OpenSBI read of the RISC-V CSR *marchid*. Considering that specific implementations can be under the same architecture ID, it is possible to get an additional identification of the implemented CPU through another OpenSBI read to the CSR *mimpid*, getting the specific implementation identifier. Taking into account both the architecture and implementation identifiers, we consider that an absolute identification can be made by merging together the lower 24 bits of the architecture identifier and the lower 8 bits of the implementation identifier (see Figure 3). Since such bit-width choice can provide up to $2^{24} \approx 17\ million$ different architectures and $2^8 = 256$ implementations of each architecture, it is expected that these values will not be a future constraint.

Each CPU Identifier can be mapped to a set of files containing fully described events. This is achieved through a mapping file (in CSV format) with the following structure:

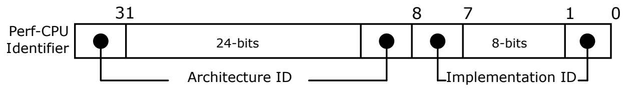

Fig. 3. CPU unique identification for Perf events.

```
CPU Identifier, File Version, Events Filename, Events Type
0x300         , 0           , CVA6           , core
0x500         , 0           , SPIKE          , core
0x200         , 0           , BOOM           , core
...
```

Although the File Version field is currently unused, the Events Filename is set to the name of the directory containing the events description, and the Events Type describes the type of events the PMU specifies. Each directory specified by the Events Filename column can contain multiple files in the Java Script Object Notation (JSON) format. Usually, each file describes an event group from one specific category (e.g., pipeline, memory, instructions, etc.), and each of the JSON files will contain one to multiple events, with the following structure:

```
{
    "Public Description": "This is an example event,
                           for demonstration purposes.",
    "Brief Description" : "This is an example event."
    "Event Code"        : "0x11",
    "Counter Mask"      : "0xF8FF",
    "Event Name"        : "EXAMPLE_EVENT",
}
```

In this example, the *Event Code* `0x11` will be used to configure the `mhpmevent#` registers of the available counters selected by the Counter Mask value, where 0xF8FF specifies that counters 8, 9, and 10 can be used to sample the event.

When monitoring the processor performance, the selected events will be forwarded to the kernel driver which, in turn, will handle the event-to-counter mapping and HPM event configuration. The kernel driver will, in turn, schedule each event or, alternatively, multiplex a set of events in the respective register, allowing for multiple events to be sampled in one workload execution, at the cost of the samples' accuracy. This process is depicted in Figure 1.

D. 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_events, this subsystem of the Linux kernel has been adopted as the main mechanism (used by PAPI library) to access the processor hardware performance counters. This architecture is shown in Figure 4. 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_events library (as illustrated in Figure 5).

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 4). To do so, we defined all supporting functions with inline assembly calls (to make direct use of RISC-V-specific instructions) and all

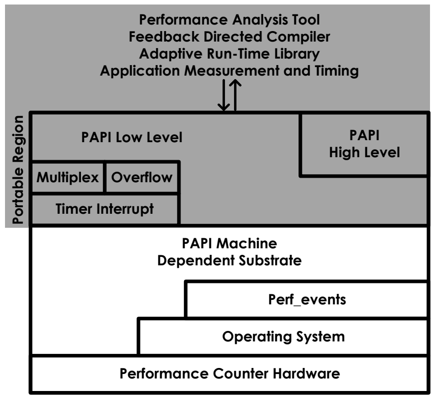
Performance Analysis Tool
Feedback Directed Compiler
Adaptive Run-Time Library
Application Measurement and Timing
Portable Region
PAPI Low Level
PAPI High Level
Multiplex
Overflow
Timer Interrupt
PAPI Machine Dependent Substrate
Perf_events
Operating System
Performance Counter Hardware

Fig. 4. PAPI architecture, modified from [23].

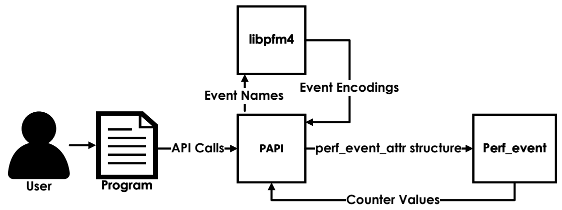

Fig. 5. Basic structure for hardware performance counting using PAPI.

necessary system parameters. 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 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 and 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_events 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.

IV. Performance Monitoring Validation

With the goal of validating our Linux Perf and PAPI library extension, we implemented the developed tools in different platforms. In particular, to allow a complete validation of the toolchain on an off-the-shelf RISC-V system, we implement Perf and PAPI on a HiFive Unmatched board. Then, to take a first step towards providing performance monitoring capabilities to RISC-V-based accelerators, we devise a Perf implementation in a QEMU simulation environment and then we implement a basic HPM unit on the CVA6 RISC-V core (often used as base architecture for the development of accelerators) and prototype it on an FPGA device.

A. Methodology

As previously referred, to evaluate our implementation of the perf and PAPI libraries we first used a SiFive Unmatched board running Linux 5.12.19. The corresponding U74-MC processor supports the Cycles and Instructions Retired fixed-event counters (counters `mcycle` and `minstret`, respectively) required by the RISC-V privileged specification [15]. Additionally, it supports two programmable-event counters (`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 II [24].

Hence, we extended our perf and PAPI implementation to fully describe the U74-MC processor PMU and interface it with Perf_events to map it to the corresponding HPM events. With the preset and native events implemented and matched to the Perf_events HPM events, we support the counting of every event listed by SiFive on the U74-MC manual [24]. [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 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 HiFive Unmatched system.

Finally, to validate the use of the devised performance tools for RISC-V-based accelerator facilities, we demonstrate the Perf tool running on a CVA6 [13] core (previously named Ariane), deployed on a Xilinx VCU128 FPGA, targeting an operating frequency of 100 MHz, running Linux 5.7.0 with BusyBox 1.31.1. To do so, we implement version 1.11 of the privileged HPM specification with sole support for the fixed-event counters detailed in Table III.

The validation of the Perf tool running the CVA6 core is conducted by running the CoreMark v1.0 [25] benchmark.

[1]The Perf_events 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
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

TABLE III
Available fixed performance events for the CVA6 platform.

Event	Counter
Cycles	`mcycle`
Instructions Retired	`minstret`
ICache Misses	`mhpmcounter3`
DCache Misses	`mhpmcounter4`
ITLB Misses	`mhpmcounter5`
DTLB Misses	`mhpmcounter6`
Loads	`mhpmcounter7`
Stores	`mhpmcounter8`
Taken Exceptions	`mhpmcounter9`
Exceptions Returned	`mhpmcounter10`
Branches and Jumps	`mhpmcounter11`
Calls	`mhpmcounter12`
Returns	`mhpmcounter13`
Mispredicted Branches	`mhpmcounter14`
Scoreboard Full	`mhpmcounter15`
Instruction Fetch Empty	`mhpmcounter16`

TABLE IV
MEASURED EXECUTION TIME OVERHEADS (AVERAGE) OF PAPI LIBRARY CALLS IN RELATION TO PERF_EVENTS CALLS.

Function Call	**Perf_events**	**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 V
PAPI LIBRARY AND PERF_EVENTS EVENT MONITORING COMPARISON WITH GEMM BENCHMARK.

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%

B. Toolchain Validation - HiFiVe Unmatched

To fully validate and profile the implemented Perf_events and PAPI library extensions, we first measure and compare the average total overhead for initialization, termination, and events start, read, and stop on both tools. To do so, we measure the total time elapsed while running each step with the `time.h` library `clock_gettime()` function. As detailed in Table IV, 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, 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_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, which compare with the 57.98µs and 15.79µs for Perf_events.

Although the observed overheads are consistent with previous research [26], 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.

We further validate the measurements obtained by using PAPI and Perf_events 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 V) 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. 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 V).

C. Performance Monitoring for RISC-V-based Accelerators

To validate Perf in the CVA6-based system we first verify the integrity of our extensions and modifications. This is done by successfully running Perf *list*, outputting the following list of available events (shortened for conciseness):

Listing 1. Output (abbreviated) of Perf *list* in CVA6.

```
  branch-instructions OR branches     [Hardware event]
  branch-misses                       [Hardware event]
  cache-misses                        [Hardware event]
  ...
  alignment-faults                    [Software event]
  ...
  iTLB-load-misses              [Hardware cache event]

branch:
  ariane_branch_jump
       [Branches/jumps count]
  ...
  ariane_ret
       [Returns count]

cache:
  ariane_dtlb_miss
       [Data TLB miss]
  ...
  ariane_store
       [Data loads]

pipeline:
  ariane_exception
       [Exceptions count]
  ...
  riscv_cycles
       [CPU cycles]
```

Finally, CoreMark was executed in the CVA6 core and monitored with Perf, achieving a performance of 174.59 points at 100 MHz. The Perf *stat* command reported the following monitored events during execution:

Listing 2. CoreMark event counter measurements from Perf *stat* in CVA6.

```
     Performance counter stats for '/bin/coremark':
         236011286      ariane_branch_jump
           5312578      ariane_call
          44038701      ariane_mis_predict
           1406812      ariane_ret
              1118      ariane_dtlb_miss
           6869722      ariane_itlb_miss
           2786559      ariane_l1_dcache_miss
           8443755      ariane_l1_icache_miss
         229104327      ariane_load
          64628214      ariane_store
             22486      ariane_exception
             22486      ariane_exception_ret
         239773306      ariane_if_empty
           9094173      ariane_sb_full
        2368685119      riscv_cycles
        1467339227      riscv_instret
```

TABLE VI
COMPUTED METRICS FROM PERFORMANCE MONITORING OF THE COREMARK BENCHMARK.

Metric		Events
Branch MissRate	18.14%	Mispred. / Branches, Calls, Returns
L1D MissRate	0.95%	L1D Misses / Loads, Stores
L1I MissRate	0.58%	L1I Misses / Instructions
ScoreBoard Full (cycles)	0.38%	ScoreBoard Full Cycles / Cycles
Inst. Fetch Empty (cycles)	10.12%	IF Empty Cycles / Cycles
Inst. Per Cycle	0.6195	Inst. / Cycles
Transl. MissRate (Data)	0.00%	Data TLB Misses / Loads, Stores
Transl. MissRate (Inst.)	0.47%	Inst. TLB Misses / Inst.

```
      23.779291520 seconds time elapsed

      23.578690000 seconds user
       0.139518000 seconds sys
```

By adding metrics support in Perf, the values in Table VI can be automatically computed after monitoring application performance. Through multiple executions of the CoreMark benchmark, with and without Perf monitoring, it was determined a performance penalty of 0.283% from using Perf.

V. CONCLUSIONS

This paper proposes a RISC-V-compatible performance monitoring and analysis framework that allows developers to optimize platform-specific code for RISC-V-based processors and accelerators. The presented tool makes use of some existing (but somewhat limited) support for RISC-V performance monitoring (using perf/perf_events), but introduces a set of new extensions and modifications, together with a new PAPI library interface, that confer RISC-V software developers with similar monitoring and analysis tools already available for other architectures (e.g., Intel, AMD, and ARM). Besides the required modifications to the kernel Perf driver and the Perf tool, the implemented extensions comprise especially devised mechanisms to read/write on machine-level privileged counters and registers through the introduction of a new OpenSBI extension. A machine-specific substrate was also developed to give support for RISC-V architectures in the PAPI library.

The validation of the proposed tool was conducted on an off-the-shelf SiFive U74-MC processor equipping a HiFive Unmatched board. It was shown not only full support for the set of counters provided by this RISC-V computing platform, but the obtained values (using the several coexisting monitoring tools) demonstrated to be closely coherent, with insignificant deviations that did not exceed 1.5%. The devised performance tools were also validated with a RISC-V-based accelerator platform, by using the modified Perf tool on a CVA6 core when executing the CoreMark benchmark.

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] 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.

[8] V. M. Weaver, "Linux perf_event features and overhead," in *The 2nd International Workshop on Performance Analysis of Workload Optimized Systems, FastPath*, vol. 13, p. 5, 2013.

[9] 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.

[10] 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.

[11] 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.

[12] 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.

[13] 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.

[14] 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.

[15] 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.

[16] 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.

[17] 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.

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

[19] A. Kao and The kernel development community, "Supporting PMUs on RISC-V platforms — The Linux Kernel documentation," Mar. 2018.

[20] Z. Li, "LKML: Zong Li: [RFC PATCH 0/6] Support raw event and DT for perf on RISC-V," June 2020.

[21] RISC-V Foundation, "Open-Source RISC-V Architecture IDs," 2021.

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

[25] S. Gal-On and M. Levy, "Exploring coremark a benchmark maximizing simplicity and efficacy," *The Embedded Microprocessor Benchmark Consortium*, 2012.

[26] 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.