Supporting RISC-V Performance Counters through Performance analysis tools for Linux (Perf)

JOAO MARIO DOMINGOS, PEDRO TOMAS, and LEONEL SOUSA, INESC-ID, Instituto Superior Técnico - Universidade de Lisboa, Portugal

Increased attention to RISC-V in Cloud, Data Center, Automotive and Networking applications, has been fueling the move of RISC-V to the high-performance computing scenario. However, lack of powerful performance monitoring tools will result in poorly optimized applications and, consequently, a limited computing performance. While the RISC-V ISA already defines a hardware performance monitor (HPM), current software gives limited support for monitoring performance. In this paper we introduce extensions and modifications to the Performance analysis tools for Linux (perf/perf_events), Linux kernel, and OpenSBI, aiming to achieve full support for the RISC-V performance monitoring specification. Preliminary testing and evaluation was carried out in Linux 5.7 running on a FPGA-booted CVA6 CPU, formerly named Ariane, showing a monitoring overhead of 0.283%.

CCS Concepts: • **Software and its engineering → Software notations and tools**.

Additional Key Words and Phrases: Performance Counters, Performance Monitoring, System Software

ACM Reference Format:
Joao Mario Domingos, Pedro Tomas, and Leonel Sousa. 2021. Supporting RISC-V Performance Counters through Performance analysis tools for Linux (Perf) . In *CARRV 2021: Fifth Workshop on Computer Architecture Research with RISC-V, June 17th, 2021, Worldwide.* ACM, New York, NY, USA, 7 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

In a High-Performance Computing era, naively porting workloads across computing platforms results in limited computing performance. Naturally, many factors come into play when justifying the observed performance limitations, including poor software implementations that result in high computational complexities, in inefficient data structures and/or limited cache usage, resulting in a memory bound execution, in execution bottlenecks, leading to processor stalls, etc.

Hence, to optimize an application, one must first monitor the its execution, identify the main performance bottlenecks, and then tailor the software to best fit with the underlying hardware. Naturally, this cannot be performed using sheer 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, we easily observe that, in the current computing scenario, there is an increasing need to capture and analyse detailed performance metrics, in order to allow in-depth architecture modeling and optimization procedures (e.g. [17]).

While Intel and ARM have proprietary performance monitoring solutions [11, 12, 22, 29], 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 PAPI [6, 9, 14] or Oprofile [19]. Mainly, RISC-V is still not fully supported in the Linux kernel monitoring tool Perf [5, 10, 27]. Only fixed counters are currently supported without event configuration, and no control over the counters is provided (e.g., pausing, enabling, disabling).

To improve the performance analysis tools for Linux (Perf) with support for the RISC-V performance monitoring facilities, in this paper we propose the following software additions and modifications:

- Support the latest RISC-V HPM specification in the Linux Perf Kernel Driver;
- Introduce customizable events for RISC-V, configurable through the Perf application;
- Support multiple platforms with distinct sets of events;
- An OpenSBI extension for privileged interaction with the RISC-V performance monitoring hardware.

Considering the multiple available RISC-V implementations, and their dissociated performance monitoring hardware implementations, we propose to consider coping strategies such as backwards compatibility and implementation features discovery. Even so, it is not possible to encompass all the details and specializations of the available implementations and RISC-V specifications at once, and therefore, we set specification version 1.11 [23] as our primarily supported target, and try to make the software flexible to support the majority of implementations.

This document represents early work that could be subjected to changes due to future knowledge and realizations. In a final form of the work, our aim is to arrive at a complete support for Perf and eventually for other relevant performance monitoring and optimization software (e.g., PAPI).

2 RISC-V HARDWARE PERFORMANCE MONITOR

The RISC-V ISA has seen a continuous ecosystem development, from wider software compatibility to an increasing number of hardware implementations [1, 3, 4, 16, 18, 21, 28, 30]. Alongside the software and hardware, the RISC-V specification also shows an uninterrupted evolution, driven by the growing requirements of the RISC-V ecosystem. Since RISC-V privileged specification version 1.7 [25], a minimal performance monitoring interface was defined

CARRV 2021, June 17th, 2021, Worldwide

ACM ISBN 978-x-xxxx-xxxx-x/YY/MM…$15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

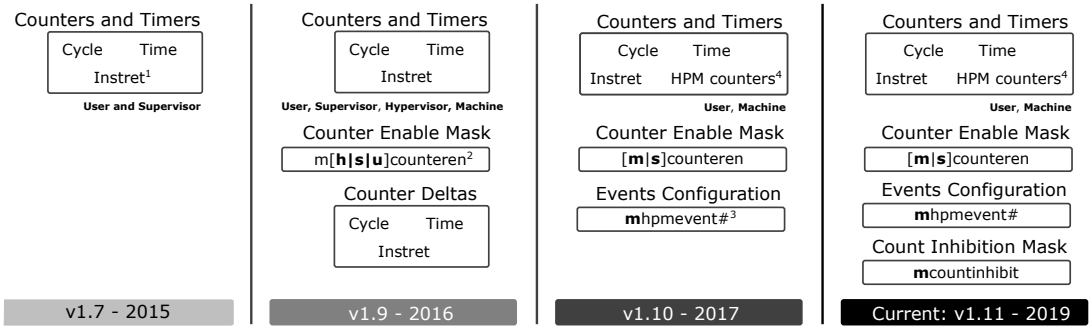

Fig. 1. Evolution of the RISC-V Hardware Performance Monitor specification

(see Figure 1). From then, the specification has introduced additional counters and necessary features for access control and event multiplexing.

2.1 Early specifications

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

Version 1.9 [24] 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 hypervisor, supervisor and user level. In addition, v1.9 introduced a set of counter deltas, a counter would keep the difference between each of the lower privilege counters and the respective machine-level counter (e.g., stime-mtime=mstime_delta). 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.

2.2 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-bits implementation). This amounts for, virtually, 2^{64} selectable events for a single register, a value that surpasses any realistic implementation, providing 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 an WARL (write any, read logical) scheme, allowing for each counter to be individually reset/set [26].

2.3 Additional Features and Future Objectives

In the latest ratified specification, version 1.11 [23], 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, as to count ISA-level metrics, such as executed floating-point or integer instructions. Similarly, some very 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 counters at a faster pace than the event sampling occurs. Although, the occurrence of such continuous overflowing is unlikely, considering implementations with 64-bits counters.

2.4 Summary

At the time, the RISC-V HPM is still significantly less complex than the x86 counterpart [8] and not comparable to the dedicated performance analysis tools like ARM's coresight and Intel's PCM-based monitoring solutions [11, 12, 22, 29]. Even so, the RISC-V HPM specification is a flexible generic performance monitoring solution, and being open-source allows any degree of implementation freedom. Considering the current state of the RISC-V privileged specification, we propose, in the following section, an approach to monitor the performance counters in RISC-V through Linux Perf.

3 PROPOSED APPROACH AND NEW EXTENSIONS

This work starts from 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 software-side events. However, a significant

limitation is the inability to write to counters and event configuration registers, as those writes require machine-level privilege, not available without a dedicated OpenSBI extension. Due to the writing limitation, it is not possible to configure events in a specific counter, significantly limiting Perf to the fixed set of CTI counters [10].

Considering the current 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 the kernel Perf driver and the Perf tool were also modified. The Linux performance monitoring system is divided into the Perf application and the kernel driver, both connected through the `perf_event_open` system call, where the kernel driver samples the events from the performance monitoring hardware counters. An overview of the system, alongside with proposed modifications, is depicted in Figure 2.

3.1 OpenSBI HPM Extension

To define an interface between software and hardware and provide the required privileged access to machine-level registers, the Hardware Performance Monitoring OpenSBI extension is herein adopted. The added OpenSBI functions are detailed in Table 1. It allows to support reading and writing to 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 an absolutely necessary feature, considering that the Linux Kernel will have sufficient privilege level, it allows the code to access the counters through an unified interface. This may be deprecated in the future if the performance impact is non-negligible and if there is no actual benefit at software level.

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 environmental call. Taking into account that the return variable *value* is of type *long*, the variable size will be the same of the implementation scalar registers (i.e., 64-bits in a 64-bit implementation, and 32-bits in a 32-bits implementation). Taking this into account, for a 32-bits system, the process of reading any HPM counter must be unfold in, at least, two calls, reading separately the lower and higher 32-bits portions of `mhpmcounter#`. Additional calls may eventually be necessary to compensate for the lower 32-bits counter overflow.

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

3.2 Linux Kernel Driver Modifications

The software-level changes proposed in this work do not impact the majority of the Linux kernel source-code. In particular, they are limited to specific areas, 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 gives 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-stopable, and non-changeable counter. Moreover, the only supported events are the cycle and instret counters, having no process for reading other HPM counters. Hence, it is not compatible with the current RISC-V HPM specification, that 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 build upon a Request for Comments kernel patch suggested by Zong Li [13], that introduced support for the HPM counters, through raw events, and device-tree bindings to support platform-specific hardware events (although 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 with 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 processes: 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 configure `mcounteren` to allow for supervisor-level read access, to achieve direct read access from the kernel to the HPM counters.

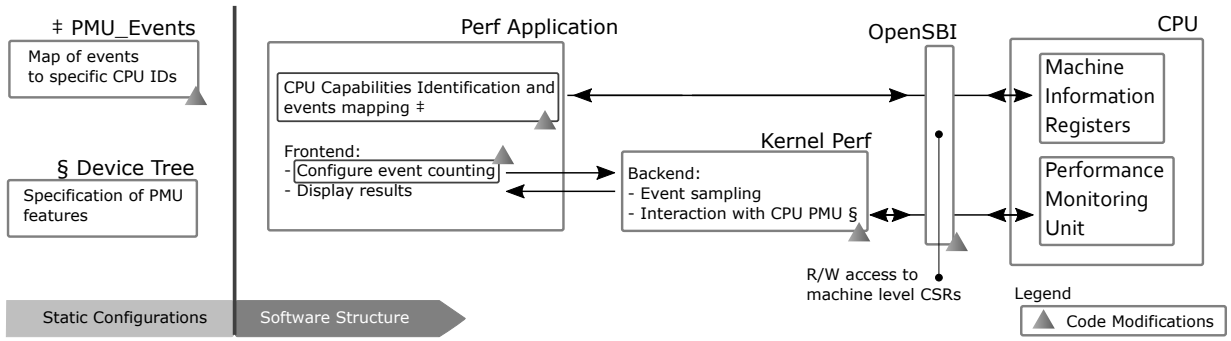

Fig. 2. Overview of the system software structure

Table 1. 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

Additionally, we introduced changes to how the events are matched 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 3. This additional parameter allows any event to be matched to any number and selection of HPM counters.

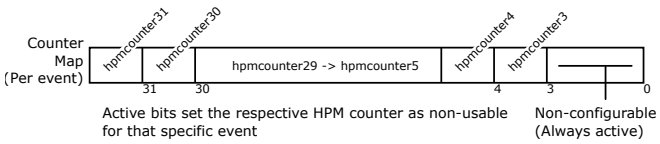

Fig. 3. Counter Map for event to HPM counter matching.

3.3 Perf Tool Modifications

The kernel driver is responsible for the actual sampling, the Perf tool acts as the frontend for event counting, providing 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, the modifications can be separated in two sets, CPU identification and events mapping.

To be able to map the set of events 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 [20], that is readable through an OpenSBI read of the RISC-V CSR *marchid*. Considering specific implementations can be under the same architecture ID, it is possible to get additional identification of the implemented CPU through another OpenSBI read to the CSR *mimpid*, getting the specific implementation identifier. Taking into account 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 to the lower 8-bits of the implementation identifier (see Figure 4). The choice of value widths can provide up to $2^{24} \approx 17$ *million* different architectures and $2^8 = 256$ implementations of each architecture, it is expect that these values will not be a future constraint.

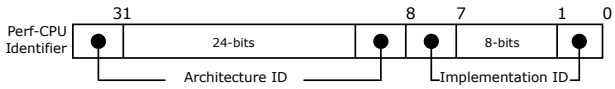

Fig. 4. CPU unique identification for Perf events.

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 example structure:

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

Where the CPU Identifier is defined according to the mentioned rules, the File Version is currently unused, the Events Filename is set to the name of 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",
}
```

For 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 the counters 8, 9 and 10 can be used to sample the event.

When monitoring performance, the selected events will be forwarded to the kernel driver, that in turn will handle the event to counter mapping and HPM event configuration. The kernel driver will, in turn, schedule each event or, in alternative, multiplex a set of events in the respective register, allowing for multiple events to be sampled in one workload execution, at the cost of samples accuracy. This process is depicted in Figure 2.

3.4 Summary

The modifications to the kernel, Perf and OpenSBI (enfatised in Figure 2), allowed us to take advantage of the RISC-V HPM specification. However, the retro-compatibility, and the divergence in the already implemented platforms poses challenges which must be faced during setup and testing of the performance monitoring software. In our current implementation, the hardware event and cache event maps (defined through the device tree bindings) are working as fixed-events, each mapping attaches the respective event to a specific counter. Consequently, this works for defined fixed-event counters, and does not have the flexibility of Perf mapping solution. Therefore, due to the inferior capabilities and increased difficulty of maintaining both solutions, the device-tree configurable events may be removed. Moreover, to better identify which processor features are available, we aim at using OpenSBI to discover registers capabilities and determine, at run-time, which features are implemented. Additionally, and to prevent illegal access to registers, the OpenSBI extension may be modified to prevent accesses that are not available in the platform, returning a descriptive error indicating that the feature is not supported. Through the OpenSBI capabilities and the Perf kernel driver we seek to achieve a good balance between a low performance overhead and powerful performance monitoring capabilities.

Table 2. 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

4 DEVELOPMENTS AND RESULTS

Initial testing and development was started in QEMU [2, 16] through the implementation of version 1.11 of the privileged HPM specification. While QEMU is a flexible, fast and powerful platform for software development, mainly when there is lack of widely available hardware, it is difficult to emulate specific hardware. In particular, while there were no major difficulties in providing read and write access to the emulated configuration registers and performance counters, counting actual performance values proved unfeasible in practice.

The first attempt of emulating performance counters was to increase each counter by a randomized value on each counter read. While the strategy was simple, there was no validity to the obtained results. A temporary alternative solution was to pause the emulated system, execute the gem5 simulator [15] with the corresponding application, and use the filesystem to retrieve gem5 statistics onto QEMU. Evidently, pausing QEMU execution during a performance counter evaluation was not feasible, and although the results could have some validity, the counter sampling sequence was not realistic, as only one of the samples would actually increment. Another solution could be to execute an actual performance monitoring of the host system through Perf, although intuitively this would result in the same problems of the prior solution. Considering these limitations, QEMU was used to develop the majority of software and to perform initial testing.

Table 3. Computed metrics from performance monitoring of the CoreMark benchmark.

Metric		Events
Branch MissRate	18.14%	Mispredictions / 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
Instruction Fetch Empty (cycles)	10.12%	IF Empty Cycles / Cycles
Instructions Per Cycle	0.6195	Instructions / Cycles
Translation MissRate (Data)	0.00%	Data TLB Misses / Loads, Stores
Translation MissRate (Instructions)	0.47%	Instructions TLB Misses / Instructions

To evaluate the solution in a real RISC-V platform, we set with the CVA6 [28] core (previously named Ariane), executing on an FPGA at 100 MHz, running Linux 5.7.0 with BusyBox 1.31.1. The choice of system was due to availability and we expect to widen the testing platforms in the future. Currently, our CVA6 implementation only supports the fixed-event counters detailed in Table 2.

With our extensions and modifications, it is now possible to list these events when running *perf list* on top of the CVA6 implementation. Outputting the following list of available events (shortened for better representation):

```
  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]
```

The CoreMark [7] benchmark was used to further test through actual performance monitoring. Version 1.0 of CoreMark was executed in the CVA6 core and monitored under (perf stat), achieving a performance of 174.59 points at 100 MHz. The Perf *stat* command reported the following monitored events during execution:

```
     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

      23.779291520 seconds time elapsed

      23.578690000 seconds user
       0.139518000 seconds sys
```

By adding metrics support in Perf, the values in Table 3 can be automatically computed after monitoring an application performance. Through multiple executions of the CoreMark benchmark, with and without Perf monitoring, the performance penalty of using Perf was determined as 0.283%, for this benchmark.

5 CONCLUSIONS AND NEXT STEPS

In this paper a RISC-V compatible performance monitoring solution is proposed, allowing for developers to do platform-specific code optimization for RISC-V processors. While there was already initial support for RISC-V performance monitoring through the Performance analysis tools for Linux (Perf), it was not comprehensive enough and supported only cycles and instructions counting. Although that version of Perf did not support the latest ISA specification, we introduce support through simple modifications to the Perf Kernel driver and to the Perf application, alongside with a new extension to OpenSBI that enables interaction with higher privilege configuration registers.

Despite Perf utilization being declining over time, mainly due to the takeover of Intel and ARM proprietary platforms, we consider this as an opportunity for Perf. Considering the open-source nature of RISC-V, it is only natural to rely on open-source tools. At this concern, Perf is a powerful, extendable and comprehensive performance monitoring tool for RISC-V, provides the natural interface for higher level libraries (e.g., PAPI).

While this proposal tackles the interface between events and the RISC-V hardware performance monitoring facility, there are other open-source initiatives, through Kernel patches, that seek to improve Perf with alternative or complimentary objectives (e.g., supporting firmware level events). Accounting that this paper represents an early work, the presented proposal may be improved by merging with upcoming opportunities or diverging ideas. Furthermore, we seek to further test the proposed performance monitoring approach

and to apply it in ASIC RISC-V hardware, such as the Hi-Five Boards and the upcoming BeagleBones BeagleV board.

ACKNOWLEDGMENTS

This work was partially supported by national funds through Fundação para a Ciência e a Tecnologia (FCT) under projects UIDB/50021/2020, and by funds from the European Union Horizon 2020 Research and Innovation programme under grant agreement No. 826647, European Processor Initiative EPI. We acknowledge the Institute of Computer Science (ICS) team at the Foundation for Research and Technology - Hellas (FORTH), in particular to the Computer Architecture and VLSI Systems (CARV) personel: Nick Kossifidis, Georgios Ieronymakis, Nikolaus Dimou, and Vassilis Papaefstathiou, for all the heavily appreciated support, tools and guidance.

REFERENCES

[1] Jonathan Balkind, Katie Lim, Fei Gao, Jinzheng Tu, David Wentzlaff, Michael Schaffner, Florian Zaruba, and Luca Benini. 2019. 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)*. 1–6.

[2] Fabrice Bellard. 2005. QEMU, a fast and portable dynamic translator.. In *USENIX annual technical conference, FREENIX Track*, Vol. 41. Califor-nia, USA, 46.

[3] Christopher Celio, Pi-Feng Chiu, Borivoje Nikolic, David Patterson, and Krste Asanovic. 2017. *BOOM v2: an open-source out-of-order RISC-V core.* Technical Report UCB/EECS-2017-157. University of California at Berkeley. 8 pages.

[4] C. Chen, X. Xiang, C. Liu, Y. Shang, R. Guo, D. Liu, Y. Lu, Z. Hao, J. Luo, Z. Chen, C. Li, Y. Pu, J. Meng, X. Yan, Y. Xie, and X. Qi. 2020. 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)*. 52–64. https://doi.org/10.1109/ISCA45697.2020.00016

[5] Arnaldo Carvalho De Melo. 2010. The new linux'perf'tools. http://www.linux-kongress.org/2010/slides/lk2010-perf-acme.pdf

[6] Jack Dongarra, Kevin London, Shirley Moore, Phil Mucci, and Dan Terpstra. 2001. Using PAPI for Hardware Performance Monitoring on Linux Systems. In *In Conference on Linux Clusters: The HPC Revolution, Linux Clusters Institute.*

[7] Shay Gal-On and Markus Levy. 2012. Exploring coremark a benchmark maximizing simplicity and efficacy. *The Embedded Microprocessor Benchmark Consortium* (2012).

[8] Intel. 2016. *Intel® 64 and IA-32 Architectures Developer's Manual: Vol. 3B.* Technical Report. https://www.intel.com/content/www/us/en/architecture-and-technology/64-ia-32-architectures-software-developer-vol-3b-part-2-manual.html

[9] Heike Jagode, Anthony Danalis, Hartwig Anzt, and Jack Dongarra. 2019. PAPI software-defined events for in-depth performance analysis. *The International Journal of High Performance Computing Applications* 33, 6 (Nov. 2019), 1113–1127. https://doi.org/10.1177/1094342019846287 Publisher: SAGE Publications Ltd STM.

[10] Alan Kao and The kernel development community. 2018. Supporting PMUs on RISC-V platforms — The Linux Kernel documentation. https://www.kernel.org/doc/html/latest/riscv/pmu.html

[11] Andi Kleen and Beeman Strong. 2015. Intel ® Processor Trace on Linux. *Tracing Summit* 2015 (2015). https://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.735.3516&rep=rep1&type=pdf

[12] Yongje Lee, Jinyong Lee, Ingoo Heo, Dongil Hwang, and Yunheung Paek. 2017. Using CoreSight PTM to Integrate CRA Monitoring IPs in an ARM-Based SoC. *ACM Transactions on Design Automation of Electronic Systems* 22, 3 (April 2017), 52:1–52:25. https://doi.org/10.1145/3035965

[13] Zong Li. 2020. LKML: Zong Li: [RFC PATCH 0/6] Support raw event and DT for perf on RISC-V. https://lkml.org/lkml/2020/6/28/374

[14] Kevin London, Shirley Moore, Philip Mucci, Keith Seymour, and Richard Luczak. 2001. The PAPI Cross-Platform Interface to Hardware Performance Counters. In *Department of Defense Users' Group Conference Proceedings*. 18–21.

[15] Jason Lowe-Power, Abdul Mutaal Ahmad, Ayaz Akram, Mohammad Alian, Rico Amslinger, Matteo Andreozzi, Adrià Armejach, Nils Asmussen, Brad Beckmann, Srikant Bharadwaj, and others. 2020. The gem5 simulator: Version 20.0+. *arXiv preprint arXiv:2007.03152* (2020).

[16] Leandro Lupori, Vanderson Rosario, and Edson Borin. 2018. Towards a high-performance RISC-V emulator. In *2018 symposium on high performance computing systems (WSCAD)*. 213–220. tex.organization: IEEE.

[17] Diogo Marques, Aleksandar Ilic, Zakhar A Matveev, and Leonel Sousa. 2020. Application-driven cache-aware roofline model. *Future Generation Computer Systems* 107 (2020), 257–273. Publisher: Elsevier.

[18] Eric Matthews and Lesley Shannon. 2017. 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)*. 1–4. https://doi.org/10.23919/FPL.2017.8056766 ISSN: 1946-1488.

[19] OProfile. 2021. OProfile - A System Profiler for Linux. https://oprofile.sourceforge.io/about/

[20] RISC-V Foundation. 2021. Open-Source RISC-V Architecture IDs. https://github.com/riscv/riscv-isa-manual/blob/master/marchid.md

[21] SiFive. 2018. The SiFive HiFive Unleashed RISC-V Board. https://www.sifive.com/boards/hifive-unleashed

[22] Alan P Su, Jiff Kuo, Kuen-Jong Lee, Ing-Jer Huang, Guo-An Jian, Cheng-An Chien, Jiun-In Guo, and Chien-Hung Chen. 2011. Multi-core software/hardware co-debug platform with ARM CoreSight™, on-chip test architecture and AXI/AHB bus monitor. In *Proceedings of 2011 International Symposium on VLSI Design, Automation and Test*. 1–6. https://doi.org/10.1109/VDAT.2011.5783594

[23] Andrew Waterman and Krste Asanović. 2019. The RISC-V Instruction Set Manual, Volume II: Privileged Architecture, Version 20190608-Priv-MSU-Ratified. *RISC-V Foundation* (June 2019).

[24] Andrew Waterman, Yunsup Lee, Rimas Avizienis, David Patterson, and Krste Asanović. 2016. The RISC-V Instruction Set Manual, Volume II: Privileged Architecture, Document Version 1.9. *EECS Department, UC Berkeley, Tech. Rep. UCB/EECS-2016-129* 129 (2016).

[25] Andrew Waterman, Yunsup Lee, Rimas Avizienis, David A Patterson, and Krste Asanovic. 2015. The RISC-V Instruction Set Manual, Volume II: Privileged Architecture, Document Version 1.7. *EECS Department, UC Berkeley, Tech. Rep. UCB/EECS-2015-49* 49 (2015).

[26] Andrew Waterman, Yunsup Lee, David A Patterson, and Krste Asanovic. 2016. The RISC-V Instruction Set Manual, Volume I: User-Level ISA Version 2.1. *EECS Department, UC Berkeley, Tech. Rep. UCB/EECS-2016-118* 118 (2016).

[27] Vincent M Weaver. 2013. Linux perf_event features and overhead. In *The 2nd International Workshop on Performance Analysis of Workload Optimized Systems, FastPath*, Vol. 13. 5.

[28] Florian Zaruba and Luca Benini. 2019. 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* 27, 11 (Nov. 2019), 2629–2640. https://doi.org/10.1109/TVLSI.2019.2926114 Conference Name: IEEE Transactions on Very Large Scale Integration (VLSI) Systems.

[29] Seyed Mohammad Ali Zeinolabedin, Johannes Partzsch, and Christian Mayr. 2021. Real-time Hardware Implementation of ARM CoreSight Trace Decoder. *IEEE Design Test* 38, 1 (Feb. 2021), 69–77. https://doi.org/10.1109/MDAT.2020.3002145 Conference Name: IEEE Design Test.

[30] Jerry Zhao, Ben Korpan, Abraham Gonzalez, and Krste Asanovic. 2020. SonicBOOM: The 3rd Generation Berkeley Out-of-Order Machine. *Workshop on Computer Architecture Research with RISC-V (CARRV)* (2020), 7.