A First Look at RISC-V Virtualization From an Embedded Systems Perspective

Bruno Sá, José Martins, and Sandro Pinto

Abstract—This article describes the first public implementation and evaluation of the latest version of the RISC-V hypervisor extension (H-extension v0.6.1) specification in a Rocket chip core. To perform a meaningful evaluation for modern multi-core embedded and mixed-criticality systems, we have ported Bao, an open-source static partitioning hypervisor, to RISC-V. We have also extended the RISC-V platform-level interrupt controller (PLIC) to enable direct guest interrupt injection with low and deterministic latency and we have enhanced the timer infrastructure to avoid trap and emulation overheads. Experiments were carried out in FireSim, a cycle-accurate, FPGA-accelerated simulator, and the system was also successfully deployed and tested in a Zynq UltraScale+ MPSoC ZCU104. Our hardware implementation was open-sourced and is currently in use by the RISC-V community towards the ratification of the H-extension specification.

Index Terms—Virtualization, RISC-V, H-extension, hypervisor, partitioning, mixed-criticality, embedded systems

✦

1 INTRODUCTION

IN the last decade, virtualization has become a key enabling technology for servers, but also for several embedded industries such as the automotive and industrial control [1], [2]. In the embedded space, the number of requirements has been steadily increasing for the past few years. At the same time, the market pressure to minimize size, weight, power, and cost (SWaP-C) has pushed for the consolidation of several subsystems, typically with distinct criticality levels, onto the same hardware platform [3], [4]. In response, academia and industry have focused on developing hardware support to assist virtualization (e.g., Arm Virtualization Extensions), and adding upstream support for these technologies in mainstream hypervisor solutions [5], [6].

Embedded software stacks are progressively targeting powerful multi-core platforms, endowed with complex memory hierarchies [7], [8]. Despite the logical CPU and memory isolation provided by existing hypervisor layers, there are several challenges and difficulties in proving strong isolation, due to the reciprocal interference caused by micro-architectural resources (e.g., last-level caches, interconnects, and memory controllers) shared among virtual machines (VM) [2], [5]. This issue is particularly relevant for mixed-criticality applications, where security- and

safety-critical applications need to coexist along non-critical ones. In this context, a malicious VM can either implement denial-of-service (DoS) attacks by increasing their consumption of a shared resource [3], [9] or to indirectly access other VM's data leveraging existing timing side-channels [10]. To tackle this issue, industry has been developing specific hardware technology (e.g., Intel Cache Allocation Technology) and the research community have been very active proposing techniques based on cache locking, cache/page coloring, or memory bandwidth reservations [5], [8], [11], [12], [13].

Recent advances in computing systems have brought to light an innovative computer architecture named RISC-V [14]. RISC-V distinguishes itself from traditional platforms by offering a free and open instruction set architecture (ISA) featuring a modular and highly customizable extension scheme that allows it to scale from tiny embedded microcontrollers up to supercomputers. RISC-V is going towards mainstream adoption under the premise of disrupting the hardware industry such as Linux has disrupted the software industry. As part of the RISC-V privileged architecture specification, hardware virtualization support is specified through the hypervisor extension (H-extension) [15]. The H-extension specification is currently in version 0.6.1, and no ratification has been achieved so far. To date, the H-extension has achieved function completeness with KVM and Xvisor in QEMU. However, none hardware implementation is publicly available yet, and commercial RISC-V cores endowed with hardware virtualization support are not expected to be released in the foreseeable future.

In this work, we share our experience while providing the first public hardware implementation of the latest version of the RISC-V H-extension in the Rocket core [16]. While the specification is intended to fulfil cloud and embedded requirements, we focused our evaluation on modern multi-core embedded and mixed-criticality systems (MCS). In this context, we have ported Bao [2], a type-1,

• The authors are with the Centro ALGORITMI, Universidade do Minho, 4710-057 Braga, Portugal. E-mail: bruno.vilaca.sa@gmail.com, {jose. martins, sandro.pinto}@dei.uminho.pt.

open-source static partitioning hypervisor, to RISC-V. In the spirit of leveraging the hardware-software codesign opportunity offered by RISC-V, we have also performed a set of architectural enhancements in the interrupt controller and the timer infrastructure aiming at guaranteeing determinism and improving performance, while minimizing interrupt latency and inter-hart interference. The experiments carried out in FireSim [17], a cycle-accurate, FPGA-accelerated simulator, and corroborated in Zynq UltraScale+ MPSoC ZCU104, demonstrate significant improvements in performance ($<$1% overhead for hosted execution) and interrupt latency ($>$89% reduction for hosted execution), at a fraction of hardware costs (11% look-up tables and 27-29% registers). We released our hardware design as open source[1] and the hardware is currently being used as a reference implementation by the RISC-V International[2] to ratify the H-extension specification.

In summary, with this work, we make the following contributions:

- the first public and open source implementation of the latest version of the RISC-V H-extension (v0.6.1) in a Rocket Chip core (Section 3);
- a set of hardware enhancements, in the platform-level interrupt controller and the architectural timer infrastructure, to tune virtualization support for embedded and mixed-criticality requirements (Section 4);
- the port of the open source Bao hypervisor for RISC-V (Section 5);
- the development of an open source ad-hoc testing framework that enable the raw functional validation of fine-grain features of the hypervisor specification (Section 6.1);
- the first public and cycle-accurate evaluation of the H-extension in a RISC-V core. We focused on hardware costs, performance overhead, inter-VM interference, and interrupt latency (Section 6);

2 RISC-V VIRTUALIZATION SUPPORT

The RISC-V privilege architecture [15] features originally three privilege levels: (i) *machine* (M) is the highest privilege mode and intended to execute firmware which should provide the supervisor binary interface (SBI); (ii) *supervisor* (S) mode is the level where an operating system kernel such as Linux is intended to execute, thus managing virtual-memory leveraging a memory management unit (MMU); and (iii) *user* (U) for applications. The modularity offered by the ISA allows implementations featuring only M or M/U which are most appropriate for small microcontroller implementations. However, only an implementation featuring the three M/S/U modes is useful for systems requiring virtualization. The ISA was designed from the ground-up to be classically virtualizable [18] by allowing to selectively trap accesses to virtual memory management control and status registers (CSRs) as well as timeout and mode change instructions from supervisor/user to machine mode. For instance, *mstatus*'s trap virtual memory (TVM) bit enables
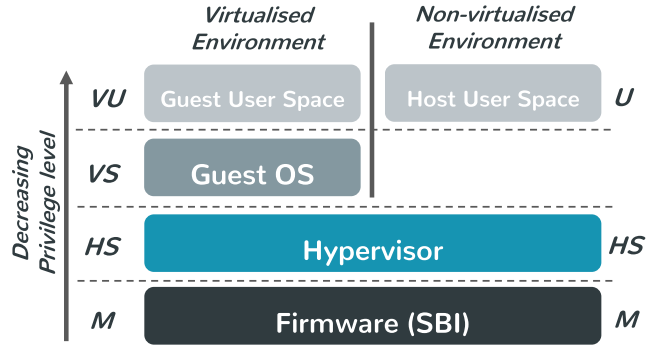

Fig. 1. RISC-V privileged levels: machine (M), hypervisor-extended supervisor (HS), virtual supervisor (VS), and virtual user (VU).

trapping of *satp*, the root page table pointer, while setting the trap sret (TSR) bit will cause the trap of the *sret* instruction used by supervisor to return to user mode. Furthermore, RISC-V provides fully precise exception handling, guaranteeing the ability to fully specify the instruction stream state at the time of an exception. The ISA simplicity coupled with its virtualization-friendly design allow the easy implementation of a hypervisor recurring to traditional techniques (e.g., full trap-and-emulate, shadow page tables) as well as the emulation of the hypervisor extension, described further ahead in this section, from machine mode. However, it is well-understood that such techniques incur large performance overheads.

2.1 Hypervisor Extension

Like most other mainstream ISAs, the latest draft of the RISC-V privilege architecture specification offers hardware virtualization support, i.e., the optional hypervisor extension ("H"), to increase virtualization efficiency. As illustrated by Fig. 1, the H-extension modifies the supervisor mode to an *hypervisor-extended supervisor mode* (HS-mode), which similarly to Intel's VT-x root mode, is orthogonal to the new *virtual supervisor mode* (VS-mode) and *virtual user mode* (VU-mode), and therefore can easily accommodate both bare-metal and hosted (a.k.a. type-1 and -2) as well as hybrid hypervisor architectures. Unavoidably, the extension also introduces two-stage address translations where the hypervisor controls the page tables mapping guest-physical to host-physical addresses. The *virtualization mode* is controlled by the implicit V bit. When V is set, either VU- or VS-mode are executing and 2nd stage translation is in effect. When V is low, the system may execute in M-, HS- or U-mode. The extension defines a few new hypervisor instructions and CSRs, as well as extends existing machine CSRs to control the guest virtual memory and execution. For example, the *hstatus* allows the hypervisor to track and control virtual machine exception behavior, *hgatp* points to the 2nd-stage root page table, and the *hfence* instructions allow the invalidation of any TLB entries related to guest translations. Additionally, it introduces a set of virtual supervisor CSRs which act as shadows for the original supervisor registers when the V bit is set. A hypervisor executing in HS-mode can directly access these registers to inspect the virtual machine state and easily perform context switches. The original supervisor registers which are not banked must be manually managed by the hypervisor.

---

1. https://github.com/josecm/rocket-chip/tree/hyp
2. former RISC-V Foundation

An innovative mechanism introduced with the H-extension is the hypervisor virtual machine load and store instructions. These instructions allow the hypervisor to directly access the guest virtual address space to inspect guest memory without explicitly mapping it in its own address space. Furthermore, because these accesses are subject to the same permissions checks as normal guest access, it precludes against confused deputy attacks when, for example, accessing indirect hypercall arguments. This capability can be extended to user mode, by setting *hstatus* hypervisor user mode (HU) bit. This further simplifies the implementation of (i) type-2 hypervisors such as KVM [6], which hosts device back-ends in userland QEMU, or (ii) microkernel-based hypervisors such as seL4 [19], which implement virtual machine monitors (VMMs) as user-space applications.

With regard to interrupts, in RISC-V, there are three basic types: external, timer, and software (essentially used as IPIs). Each interrupt can be re-directed to one of the privileged modes by setting the bit for target interrupt/mode in a per-hart (hardware-thread, essentially a core in RISC-V terminology) interrupt pending bitmap, which might be directly driven by some hardware device or set by software. This bitmap is fully visible to M-mode through the *mip* CSR, while the S-mode has a filtered view of its interrupt status through *sip*. This concept was extended to the new virtual modes through the *hvip*, *hip*, and *vsip* CSRs. As further detailed in Section 4, in current RISC-V implementations, a hardware module called the core-local interrupter (CLINT) drives the timer and software interrupts, but only for machine mode. Supervisor software must configure timer interrupts and issue IPIs via SBI, invoked through *ecall* (environment calls, i.e., system call) instructions. The firmware in M-mode is then expected to inject these interrupts through the interrupt bitmap in the supervisor. The same is true regarding VS interrupts as the hypervisor must itself service guest SBI requests and inject these interrupts through *hvip* while in the process invoking the machine-mode layer SBI. When the interrupt is triggered for the current hart, the process is inversed: (i) the interrupt traps to machine software, which must then (ii) inject the interrupt in HS-mode through the interrupt pending bitmap, and then (iii) inject it in VS mode by setting the corresponding *hvip* bit. As for external interrupts, these are driven by the platform-level interrupt controller (PLIC) targeting both M and HS modes. The hypervisor extension specifies a guest external interrupt mechanism which allows an external interrupt controller to directly drive the VS external interrupt pending bit. This allows an interrupt to be directly forward to a virtual machine without hypervisor intervention (albeit in an hypervisor controlled manner). However, this feature needs to be supported by the external interrupt controller. Unfortunately, the PLIC is not yet virtualization-aware. A hypervisor must fully trap-and-emulate PLIC accesses by the guest and manually drive the VS external interrupt pending bit in *hvip*. The sheer number of traps involved in these processes is bound to impact interrupt latency, jitter, and, depending on an OS tick frequency, overall performance. For these reasons, interrupt virtualization support is one of the most pressing open-issues in RISC-V virtualization. In Section 4, we describe our approach to address this issue.
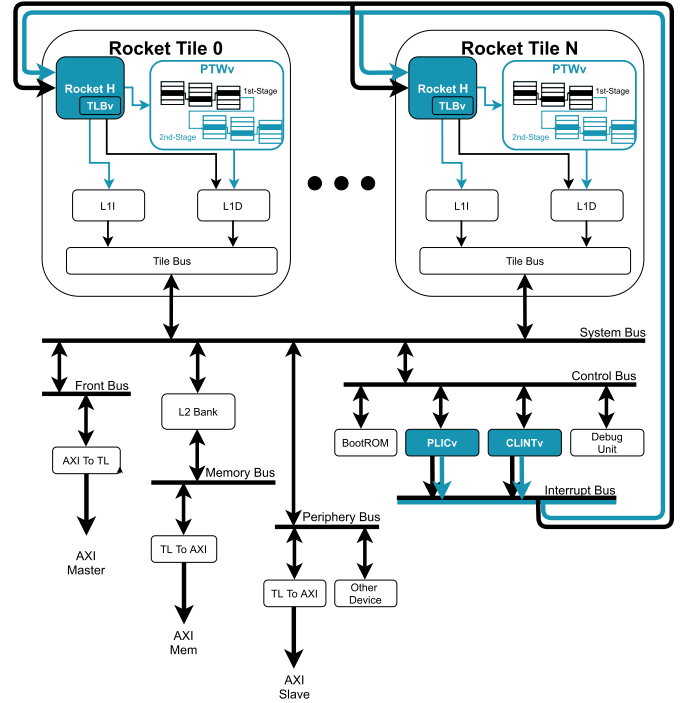

Fig. 2. Rocket chip diagram. H-extension and interrupt virtualization enhancements (CLINTv and PLICv) highlighted in blue. Adapted from [16].

The hypervisor extension also augments the trap encoding scheme with multiple exceptions to support VS execution. For instance, it adds guest-specific page faults exceptions for when translations at the second stage MMU fail, as well as VS-level ecalls which equate to hypercalls.

Finally, it is worth mentioning that the specification is tailored to seamlessly support nested virtualization; however, nested virtualization is out-of-scope of this article.

3 ROCKET CORE HYPERVISOR EXTENSION

We have implemented the RISC-V hypervisor extension in the open-source Rocket core, a modern 5-stage, in-order, highly configurable core, part of the Rocket chip SoC generator [16] and written in the novel Chisel hardware construction language [20] (Fig. 2). Despite being possible to configure this core according to the 32- or 64-bit variants of the ISA (RV32 or RV64, respectively), our implementation currently only supports the latter. The extension can be easily enabled by adding a *WithHyp* configuration fragment in a typical Rocket chip configuration.

Hypervisor CSRs and Instructions. The bulk of our H-extension implementation in the Rocket core revolves around the CSR module which implements most of the privilege architecture logic: exception triggering and delegation, mode changes, privilege instruction and CSRs, their accesses, and respective permission checks. These mechanisms were straightforward to implement, as very similar ones already exist for other privilege modes. Although most of the new CSRs and respective functionality mandated by the specification were implemented, we have left out some optional features. Specifically:

TABLE 1
Current State of Hypervisor Extension Features Implemented in the Rocket Core: ● Fully-Implemented; ◐ Partially Implemented; ○ Not Implemented

CSRs	hstatus/mstatus	●
	hideleg/hedeleg/mideleg	●
	hvip/hip/hie/mip/mie	●
	hgeip/hgeie	●
	hcounteren	●
	htimedelta	○
	mtval2/htval	●
	mtinst/htinst	○
	hgapt	◐
	vsstatus/vsip/vsie/vstvec/vsscratch vsepc/vscause/vstval/vsatp	●
Instructions	hlv/hlvx/hsv	●
	hfence.vvma/gvma	◐
Exceptions & Interrupts	Environment call from VS-mode	●
	Instruction/Load/Store guest-page fault	●
	Virtual instruction	●
	Virtual Supervisor sw/timer/external interrupts	●
	Supervisor guest external interrupt	●

1) *htimedelta* is a register that contains the difference between the value obtained by a guest when reading the *time* register and the actual *time* value. We expect this register to be emulated by the firmware running in M-mode as it is the only mode allowed to access *time* (see CLINT background in Section 4.1);

2) *htinst* and *mtinst* are hardwired to zero. These registers expose a trapping instruction in an easy and pre-decoded form so that the hypervisor can quickly handle the trap while avoiding reading the actual guest instruction and polluting the data cache;

3) *hgatp* is the hypervisor's 2nd-stage root page-table pointer register. Besides the pointer it allows to specify a translation mode (essentially, page-table formats and address space size) and the VMID (virtual machine IDs, akin to 1st-stage ASIDs). The current implementation does not support VMIDs and only allows for the *Sv39x4* translation mode;

4) the *hfence* instructions, which are the hypervisor TLB synchronization or invalidation instructions, always invalidate the full TLB structures. However, the specification allows to provide specific virtual addresses and/or a VMID to selectively invalidate TLB entries.

Nevertheless, all the mandatory H-extension features are implemented and, therefore, our implementation is fully compliant with the RISC-V H-extension specification. Table 1 summarizes all the included and missing features, and Fig. 3 highlights the major architectural changes to the Rocket core functional blocks.

Two-Stage Address Translation. The next largest effort focused on the MMU structures, specifically the page table walker (PTW) and translation-lookaside buffer (TLB), in particular, to add to support for the 2nd-stage translation. The implementation only supports the *Bare* translation mode (i.e., no translation) and the *Sv39x4*, which defines a specific page table size and topology which results in guest-physical addresses with a maximum width of 41-bits. The modification to the PTW extends the module's state-machine so that it switches to perform 2nd-stage translation

at each level of the 1st translation stage (Fig. 4). At each step it merges the results of both stages. When a guest leaf page table entry (PTE) is reached, it performs a final translation of the targeted guest-physical address. This proved to be one of the trickiest mechanisms to implement, given the large number of corner cases that arise when combining different page sizes at each level and of exceptions that might occur at each step of the process. TLB entries were also extended to store both the direct guest-virtual to host-physical address as well as the resulting guest-physical address of the translation. This is needed because even for a valid cached 2-stage translation, later accesses might violate one of the RWX permissions, and the specification mandates that the guest-physical address must be reported in *htval* when the resulting exception is triggered. Note that the implementation does not support VMID TLB entry tagging. We have decided to neglect this optional feature for two mains reasons. First, at the time of this witting, the Rocket core did not even support ASIDs. Second, static partitioning hypervisors (our main use case) do not use it at all. A different hypervisor must invalidate these structures at each context-switch. As such, the implemented support for *hfence* instructions ignores the VMID argument. Furthermore, they invalidate all cached TLB or walk-cache entries used in guest translation, despite it specifying a virtual address argument, or being targeted at only the first stage (*hfence.hvma*) or both stages (*hfence.gvma*). To this end, an extra bit was added to TLB entries to differentiate between the hypervisor and virtual-supervisor translations. Finally, we have not implemented any optimizations such as dedicated 2nd-stage TLBs as many modern comparable processors do, which still leaves room for important optimizations.

Hypervisor Virtual-Machine Load and Store Instructions. Despite most of the implementation being straightforward, because it mainly involved replicating or extending existing or very similar functionality and mechanisms, the most invasive implemented feature was the support for hypervisor virtual machine load and store instructions. This is because, although RISC-V already provided mechanisms for a privilege level to perform memory accesses subject to the same translation and restrictions of a lower privilege (such as by setting the modify privilege - MPRV - bit in *mstatus*), these sync the pipeline instruction stream which results in the actual access permission modifications being associated with the overall hart state and not tagged to a specific instruction. As such, we added dedicated signals that needed to be propagated throughout the pipeline starting from the decode stage, going through the L1 data cache up to the data TLB to signal the memory access originates from a hypervisor load/store. If this signal is set, the TLB will ignore the current access privilege (either HS or HU), and fetch the translation and perform the privilege checks as the access was coming from a virtual machine (VS or VU). Similar signals already existed for the fence instructions.

Other Modifications. The Rocket chip generators truncate physical address width signals to the maximum needed for the configured physical address space. Thus, another issue we faced was the need to adapt the bit-width of some buses and register assumed to be carrying physical addresses to support guest-physical addresses, essentially virtual addresses. Fin-
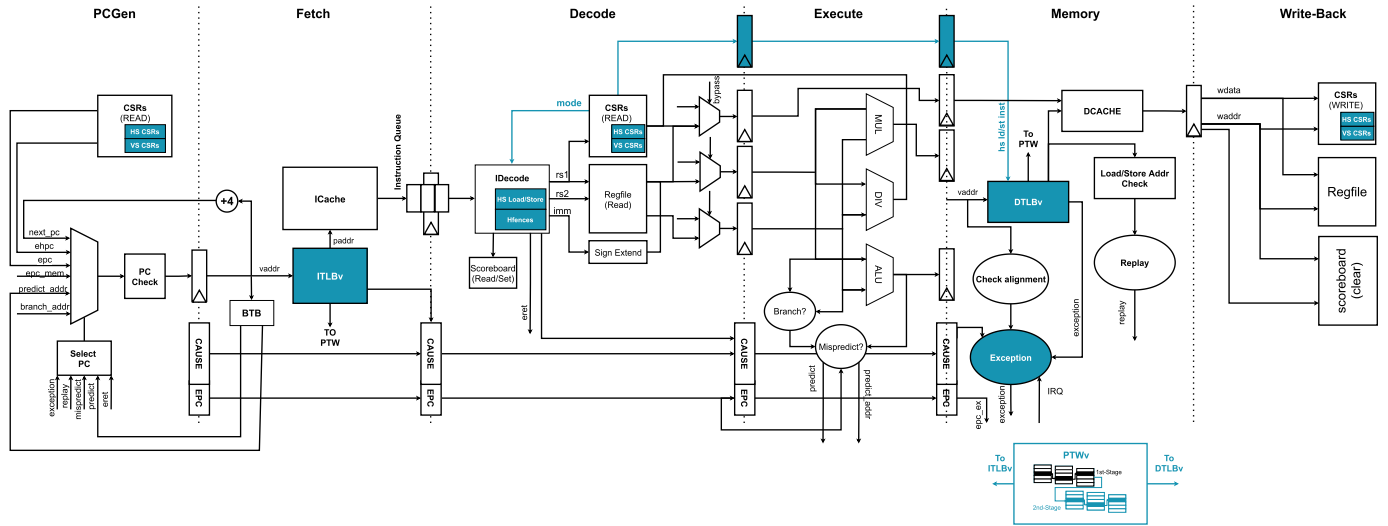
Fig. 3. Rocket core microarchitecture overview featuring the H-extension. Major architectural changes to the Rocket core functional blocks (e.g., Decoder, PTW, TLBs, and CSRs) are highlighted in blue. Adapted from[3].

ally, we also needed to slightly modify the main core pipeline to correctly forward virtual exceptions, including the new virtual instruction exception, to the CSR module along with other hypervisor-targeted information such as faulting guest-physical addresses to set the *htval/mtval* registers.

4 INTERRUPT VIRTUALIZATION ENHANCEMENTS

As explained in the previous sections, RISC-V support for virtualization still only focuses on CPU virtualization. Therefore, as illustrated in Fig. 2, we have also extended other Rocket chip components, namely the PLIC and the CLINT, to tackle some of the previously identified drawbacks regarding interrupt virtualization.

4.1 Timer Virtualization

CLINT Background. CLINT is the core-level interrupt controller responsible for maintaining machine-level software and timer interrupts in the majority of RISC-V systems. To inject software interrupts (IPIs in RISC-V lingo) in M-mode, the CLINT

facilitates a memory-mapped register, denoted *msip*, where each register is directly connected to a running CPU. Moreover, the CLINT also implements the RISC-V timer M-mode timer specification, more specifically the *mtime* and *mtimecmp* memory-mapped control registers. *mtime* is a free-running counter and a machine timer interrupt is triggered when its value is greater than the one programmed in *mtimecmp*. There is also a read-only *time* CSR accessible to all privilege modes, which is not supposed to be implemented but converted to a MMIO access of *mtime* or emulated by firmware. Thus, M-mode software implementing the SBI interface (e.g., OpenSBI) must facilitate timer services to lower privileges via ecalls, by multiplexing logical timers onto the M-mode physical timer.

CLINT Virtualization Overhead. Naturally, this mechanism introduces additional burdens and impacts the overall system performance for HS-mode and VS-Mode execution, especially in high-frequency tick OSes. As explained in Section 2, a single S-mode timer event involves several M-mode traps, i.e., first to set up the timer and then to inject the interrupt in S-mode. This issue is further aggravated in virtualized environments as it adds extra HS-mode traps. The simplest solution to mitigate this problem encompasses providing multiple independent hardware timers directly available to the lower privilege levels, HS and VS, through new registers analogous to the M-mode timer registers. This approach is followed in other well-established computing architectures. For instance, the Armv8-A architecture has separate timer registers across all privilege levels and security states.

CLINT Virtualization Extensions. As detailed in Table 2, we added read-only *stime* and *vstime*, as well as read/write *stimecmp* and *vstimecmp* memory-mapped registers to the CLINT. Furthermore, we implemented a memory-mapped version of the *htimedelta* CSR, which defines a drift of *time* as viewed from VS- or VU-mode perspectives. In our implementation (see Fig. 5), *htimedelta* will be reflected in the value of *vstime*, by adding it to the value of *mtime*. Adopting such approach would enable S-mode software to directly
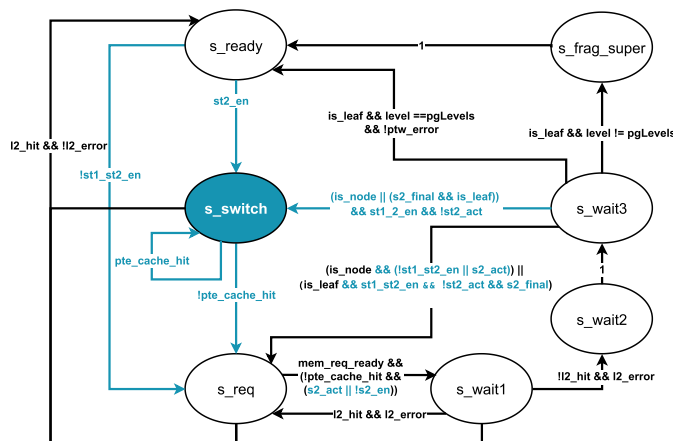

Fig. 4. PTW state machine featuring the two-stage translation. Modifications to the state machine, including a new state (*s_switch*) to switch the translation between the two stages (e.g., change the root page-table pointer), are highlighted in blue.
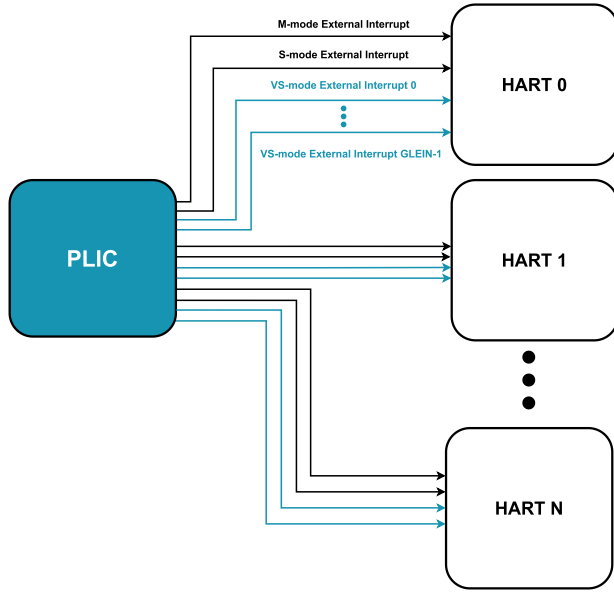
3. https://inst.eecs.berkeley.edu/~cs250/fa13/handouts/lab2-riscv.pdf

TABLE 2
CLINT Memory Map

Field	Offset(hex)	Description
msip n	0x00000 + (n*4)	M-mode hart n software interrupt
mtimecmp n	0x04000 + (n*8)	Compare value for the M-mode hart n timer
mtime	0x0bff8	Current time value
stimecmp n	**0x0c000 + (n*8)**	**Compare value for the S or HS-mode hart n timer**
stime	**0x1bff8**	**Current time value for S-mode (RO replica of mtime)**
vstimecmp n	**0x1c000 + (n*8)**	**Compare value for the VS-mode hart n timer**
vstime n	**0x14000 + (n*8)**	**Current time value for the VS-mode hart n (mtime + htimedelta n)**
htimedelta n	**0x24000 + (n*8)**	**Holds the current time delta between the value of the time CSR and the value returned in VS-mode hart n**

In bold, the new HS and VS timer registers.

interact with its timer and receive interrupts without firmware mediation. In CLINT implementations each type of timer registers are mapped onto separate pages. However, hart replicas of the same register are packed contiguously in the same page. As such, with this approach, the hypervisor still needs to mediate VS- register access as it cannot isolate VS registers of each individual virtual hart (or vhart) using virtual memory. Nevertheless, traps from HS- to M-mode are no longer required, and when the HS and VS timer expires, the interrupt pending bit of the respective privilege level is directly set.

CLINTv versus Other Proposals. Concurrently to our work, there have been some proposals discussed among the community to include dedicated timer CSRs for HS and VS modes. The latest one which is officially under consideration, at a high-level, is very similar to our implementation. However, there are differences with regard to: first, it does not include *stime* and *vstime* registers but only the respective timer compares; second, and more important, we add the new timer registers as memory-mapped IO (MMIO) in the CLINT, and not as CSRs. The rationale behind our decision is based on the fact that the RISC-V specification states that the original M-mode timer registers are memory-mapped, due to the need to share them between all harts as well as due to power and clock domain crossing concerns. As the new timers still directly depend on the original *mtime* source value, we believe its simpler to implement them as MMIO, centralizing all the timer logic. Otherwise, every hart would have to continuously be aware of the global *mtime* value, possibly through a dedicated bus. Alternatively, it would be possible to provide the new registers, as well as *htimedelta*, through the CSR interface following the same approach as the one used for *time*, i.e., by converting the CSR accesses to memory accesses. This approach would, however, in our view, add unnecessary complexity to the pipeline as supervisor software can always be informed of the platform's CLINT physical address through standard methods (e.g., device tree).

Fig. 5. CLINT microarchitecture with virtualization enhancements. Architectural changes to include hardware support for S and VS mode timers are highlighted in blue.

4.2 PLIC Virtualization

PLIC Background. The PLIC is the external interrupt controller used in most current RISC-V systems. The PLIC is capable of multiplexing multiple devices interrupts to one or more hart contexts. More specifically, up to 1023 devices interrupt lines can be connected to the PLIC, each with its configurable priority. PLIC contexts represent a set of registers and external interrupts lines, each targeting a specific privilege level within a given hart (see Fig. 6). Each line will drive the corresponding privilege bit in the hart global external interrupt pending bitmap. Part of each context, the PLIC provides registers for interrupt enabling as well as for interrupt handling: upon entry on an interrupt handler, software reads a context claim register which returns the interrupt ID that triggered the interrupt. To complete the interrupt handling process, the hart must write back to the *complete register* the retrieved interrupt ID. The claimed interrupt will not be re-triggered until the interrupt during this process. Beyond that, PLIC also supports context interrupt masking through the *threshold register*, i.e., interrupts with priority lower than the threshold value are not delivered to the context. Importantly, each set of claim/complete/threshold registers is mapped onto a different physical memory page.

PLIC Virtualization Overhead. Currently, only M-mode and S-mode contexts are supported (grey lines in Fig. 6), meaning the PLIC specification does not provide additional interrupt virtualization support. The hypervisor is then responsible for emulating PLIC control registers accesses and fully managing interrupt injection into VS-mode. Emulating PLIC interrupt configuration registers, such as *enable* and *priority registers*, may not be critical as it is often a one-time-only operation performed during OS initialization. However, the same does not apply to the *claim/complete registers*, which must be accessed before and after every

Fig. 6. High-level virtualization-aware PLIC logic.

TABLE 3
Extended Memory Map for the Virtualization-Aware PLIC

Field	Offset	Description
priority i	0x0000000 (i * 4)	Interrupt ID i priority register
pending	0x0001000	Interrupt source pending bits. Up to 32 sources per register.
enable c	0x0002000 + (c * 0x80)	Interrupt source enable registers bits for context c. Upto 32 sources per register.
threshold c	0x0200000 + (c * 0x1000)	Priority threshold register for context c
claim/ complete c	0x0200004 + (c * 0x1000)	Claim/Complete register for context c
vcibir c	**0x4000000 + (c * 4)**	**Virtual context c injection block ID register**
viir j **block** n	**0x4010000 + (n * 0x1000) + (j * 4)**	**Virtual interrupt injection register j of block n**
ibmsr block n	**0x4110000 + (n * 4)**	**Injection block management and status register for block n**

In bold, the new virtual interrupt injection registers.

interrupt handler. For physical interrupts directly assigned to guests, this is further aggravated, since it incurs in the extra trap to the hypervisor to receive the actual interrupt before injecting it in the virtual PLIC. These additional mode-crosses causes a drastic increase in the interrupt latency and might seriously impact overall system performance, especially for real-time systems that rely on low and deterministic latencies.

PLIC Virtualization Enhancements. Based on the aforementioned premises, we propose a virtualization extension to the PLIC specification[4] that could significantly improve the system's performance and latency by leveraging the guest external interrupt feature of the RISC-V hypervisor extension (see Section 2). We had four main requirements: (i) allow direct assignment and injection of physical interrupts to the active VS-mode hart, i.e., without hypervisor intervention; (ii) minimize traps to the hypervisor, in particular, by eliminating traps on claim/complete register access; (iii) allow a mix of purely virtual and physical interrupts for a given VM; and (iv) a minimal design with a limited amount of additional hardware cost and low overall complexity. We started by adding *GEILEN* VS-mode contexts to every hart. *GEILEN* is a hypervisor specification "macro" that defines the maximum (up to 64) available VS external interrupt contexts that can be simultaneously active for a hart, in a given implementation. This is a configurable parameter in our implementation. The context for the currently executing vhart is selected through the VGEIN field in the *hstatus* CSR. Fig. 6 highlights, in blue, the external interrupt lines associated with VS contexts that directly drive the hart's VS mode external interrupt pending bits. With additional virtual contexts and independent context's *claim/complete register* pair available on separate pages, the hypervisor can allow direct guest access to *claim/complete registers* by mapping them in the guests' physical address space, mitigating the need to trap-and-emulate such registers. However, access to configuration registers such as *priority* and *enable*

registers are still trapped since these registers are not often accessed and are not in the critical path of interrupt handling. A hypervisor might assign physical interrupts to a guest as follows. When a guest sets configurations register fields for a given interrupt, a hypervisor commits it to hardware in the vhart's context if the interrupt was assigned to its VM. Otherwise, it might save it in memory structures if it corresponds to an existing virtual interrupt fully managed by the hypervisor.

PLICv Pure Virtual Interrupts Extensions. With direct access to *claim/complete registers* at the guest level, injection of purely virtual interrupts must also be done through the PLIC, so there are unified and consistent forwarding and handling for all the vhart's interrupts. To this end, and inspired by Arm's GIC list registers, we added three new memory-mapped 32-bit wide register sets to the PLIC to support this operation (see Table 3 and Fig. 7): *Virtual Interrupt Injection Registers* (VIIR), *Virtual Context Injection Block ID Registers* (VCIBIR), and *Injection Block Management and Status Registers* (IBMSR). VIIRs are grouped into page-sized *injection blocks*. The number of *VIIRs* in a block and the number of available blocks are implementation-defined (up to a maximum of 1,000 and 240, respectively); however, if a given block exists, at least one register must be implemented. There are *GEILEN VCIBIR* per-hart which are used to specify the source injection block used for a given context's virtual interrupt injection. In this way, virtual interrupts for multiple harts belonging to a specific VM can be injected through a single injection block, precluding the need for complex synchronization across hypervisor harts. Also, this allows a hart to directly inject an interrupt in a foreign vhart without forcing an extra HS trap. Setting a context's *VCIBIR* to zero indicates that no injection block is attached.

PLICv Virtual Interrupt Injection Mechanism. Virtual interrupt injection is done through the *VIIRs* which are composed of three fields: *inFlight*, *interruptID*, and *priority*. Setting the VIIR with a *interruptID* greater than 0 and the *inFlight* bit not set would make the interrupt pending for

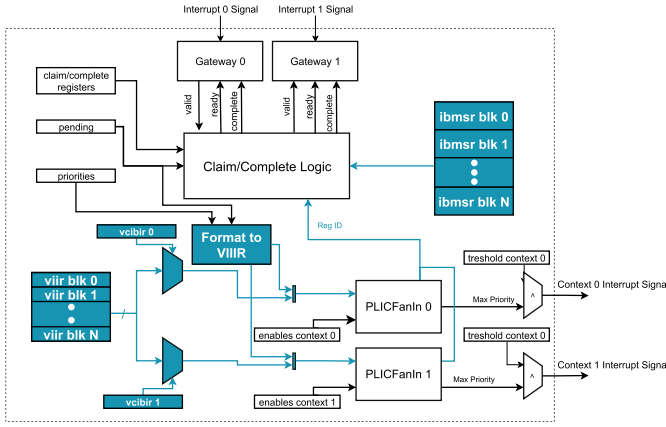4. https://github.com/josecm/riscv-plic-spec/tree/virt

Fig. 7. PLIC microarchitecture with virtualization enhancements. Architectural changes to include support for virtual interrupt injection are highlighted in blue.

the virtual contexts associated with its block. The bit *inFlight* is automatically set when the virtual interrupt is pending and a claim is performed indicating that the interrupt is active, preventing the virtual interrupt from being pending. When the *complete register* is written with an ID present in a VIIR, that register is cleared, otherwise an interrupt might be raised to signal it, as explained next. On a *claim register* read, the PLIC selects the higher priority pending interrupt of either a context's injection block or enabled physical interrupts. Each block is associated with a *block management interrupt* (akin to GIC's maintenance interrupt) fed back through the PLIC itself with implementation-defined IDs. It serves to signal events related to the block's *VIIRs* lifecycle. Currently, there are two well-defined events: (i) no *VIIR* pending, and (ii) claim write of an non-present *interruptID*. The enabling of each type of event, signaling of currently pending events and complementary information are done through a corresponding *IBMSR*. Note that all of the new PLIC registers are optional as the minimum number of injection blocks is zero. If this is the case, a hypervisor might support either (i) a VM with only purely virtual interrupts, falling back to the full trap-and-emulate model, or (ii) a VM with only physical interrupts.

PLICv Context Switch. An important point regarding the PLIC virtualization extensions we have somewhat neglected in our design is its impact on vhart context-switch. At first sight, it might seem prohibitively costly due to the high number of MMIO context and block registers to be saved and restored. However we believe this is minimized first (i) due to the possibility of having up to *GEILEN* virtual contexts for each hart and a number of *injection blocks* larger than the maximum possible number of active vharts; second (ii) we expect that only a small number (1 to 4) VIIRs are implemented in each block; and third (iii) as we expect that physical interrupt assignment will be sparse for a given virtual machine, an hypervisor can keep a word-length bitmap of the enable registers that contain these interrupts, and only save/restore the absolutely needed.

5 BAO RISC-V PORTING

Bao in a Nutshell. Bao [2] is an open-source static partitioning hypervisor developed with the main goal of facilitating the

straightforward consolidation of mixed-criticality systems, thus focusing on providing strong safety and security guarantees. It comprises only a minimal, thin-layer of privileged software leveraging ISA virtualization support to partition the hardware, including 1-to-1 virtual to physical CPU pinning, static memory allocation, and device/interrupt direct assignment. Bao implements a clean-slate, standalone component featuring about 8 K SLoC (source lines of code), which depends only on standard firmware to initialize the system and perform platform-specific tasks such as power management. It provides minimal inter-VM communication facilities through statically configured shared memory and notifications in the form of interrupts. It also implements from the get-go simple hardware partitioning mechanisms such as cache coloring to avoid interference in caches shared among the multiple VMs.

Bao for RISC-V. Bao originally targeted only Arm-based systems. So, we have initially ported to RISC-V using the QEMU implementation of the H-extension. Later, this port was used in the Rocket core with the extensions described in this article without any friction. Given the simplicity of both the hypervisor and RISC-V designs, in particular, the virtualization extensions and the high degree of similarity with Arm's Aarch64 ISA, the port was mostly effortless. It comprised an extra 3,731 SLoC compared to Arm's 5,392 lines of arch-specific code. Nevertheless, a small step-back arose that forced the need to modify the hypervisor's virtual address space mappings. For security reasons, Bao used the recursive page table mapping technique so that each CPU would map only the memory it needs and not map any internal VM structures besides its own or any guest memory. RISC-V impose some constrains given that each PTE must either serve has a next-table pointer or a leaf PTE. Therefore, we had to modify Bao to identity-map all physical memory, to enable performing software page walks to build guest page tables. Another RISC-V-specific facility we had to implement in Bao was the read guest memory operation. This need arose because neither QEMU nor our Rocket implementation of the H-extension provides the pre-decoded trapped instruction on *htinst*. Therefore, on a guest trap due to a page-fault that requires emulation (e.g., PLIC access), Bao must read the instruction from guest memory and decode it in software. Nevertheless, Bao never directly maps guest memory in its own address space. It reads the instruction via hypervisor load instructions. The RISC-V Bao port relies on a basic SBI firmware implementation featuring the IPI, Timer, and RFENCE extensions. As of this writing, only OpenSBI has been used. Bao provides SBI services to guests so these can program timer interrupts and send IPI between vharts. However, Bao mostly acts as a shim for most VS- SBI calls, as the arguments are just processed and/or sanitized and the actual operation is relegated to the firmware SBI running in machine mode.

Bao RISC-V Limitations. It is also worth mentioning there are still some gaps in the ISA to suors like Bao. For example, cache maintenance support operations are not standardized. At the moment, core implementations must provide this functionality via custom facilities. The Rocket core provides a custom machine-mode instruction for flushing the L1 data cache. As Bao relies on these operations to implement some of its features (e.g., cache coloring), we have

implemented a custom cache flush API in OpenSBI that Bao calls when it needs to clean caches. Another issue regards external interrupt support. Due to the PLIC's virtualization limitations (see Section 4.2), Bao's implementation must fully trap and emulate guest access to the PLIC, i.e., not only on configuration but also on interrupt delivery and processing. As we show in Section 6.4, this adds significant overheads, especially on interrupt latency. Finally, Bao relies on IOMMU support to be able to directly assign DMA-capable devices to guest OSes. However, there is no standard IOMMU currently available in RISC-V platforms (see Section 8).

6 EVALUATION

The evaluation was conducted for three different SoC configurations, i.e., dual-, quad-, and six-core Rocket chip with per-core 16 KiB L1 data and instruction caches, and a shared unified 512 KiB L2 LLC (last-level cache). The software stack encompasses the OpenSBI (version 0.9), Bao (version 0.1), and Linux (version 5.9), and bare metal VMs. OpenSBI, Bao, and bare metal VMs were compiled using the GNU RISC-V Toolchain (version 8.3.0 2020.04.0), with -O2 optimizations. Linux was compiled using the GNU RISC-V Linux Toolchain (version 9.3.0 2020.02-2). Our evaluation focused on functional verification (Section 6.1), hardware resources (Section 6.2), performance and inter-VM interference (Section 6.3), and interrupt latency (Section 6.2).

6.1 Functional Verification

The functional verification of our hardware implementation was performed on a Verilator-generated simulator and on a Zynq UltraScale+ MPSoC ZCU104 FPGA.

We have developed an ad-hoc testing framework as a bare-metal application. Our goal was to test individual features of the hypervisor specification without any additional system software complexity and following a test-driven development (TDD) like approach. During development, we have written a comprehensive test suite spanning features such as two-stage translation, exception delegation, virtual exceptions, hypervisor load-store instructions, CSR access, just to name a few. To be able to test out individual features, the framework provides an API which easily allows: (i) fully resetting processor state at beginning of each test unit; (ii) fluidly and transparently changing privilege mode; (iii) easy access a guest virtual address with any combination of 1st and 2nd stage permissions; and (iv) easy detection and recovery of exceptions and later checking of its state and causes. Nevertheless, the framework still has some limitations such as not allowing user mode execution or experimenting with superpages. This hypervisor extension testing framework and accompanying test suite are openly available[5] and can be easily adapted to other platforms. We have also run our test suite in QEMU, unveiling bugs in the hypervisor extension implementation, for which patches were later submitted.

As a second step to validate our implementation we have successfully ran two open-source hypervisors: Bao and XVisor [21]. XVisor also provides a "Nested MMU Test-suite"

TABLE 4
Rocket Chip Hardware Resource Overhead With Virtualization Extensions

		Dual-Core	Quad-Core	Six-Core
Rocket Cores	LUTs	50922/**11%**	101744/**12%**	152957/**12%**
	Regs	25086/**30%**	50172/**30%**	75258/**30%**
CLINT	LUTs	68/**375%**	196/**296%**	269/**373%**
	Regs	194/**297%**	324/**336%**	454/**277%**
PLIC	LUTs	90/**140%**	144/**236%**	220/**263%**
	Regs	83/**325%**	116/**412%**	149/**460%**
Others	LUTs	11207/**2%**	13242/**3%**	91821/**0, 5%**
	Regs	4257/**0, 1%**	4628/**0, 2%**	4728/**2%**
Total	LUTs	62287/**11%**	115356/**11%**	167753/**11%**
	Regs	29620/**27%**	55250/**28%**	80589/**29%**

which mainly exercises the two-stage translation. At the time of this writing, our implementation fully passes this test suite. Some bugs uncovered while running these hypervisors were translated into tests and incorporated into our test suite.

6.2 Hardware Overhead

To assess the hardware overhead, we synthesized multiple SoC configurations with an increasing number of harts (2, 4, and 6). We used Vivado 2018.3 targeting the Zynq Ultra-Scale+ MPSoC ZCU104 FPGA. Table 4 presents the post-synthesis results, depicting the number of look-up tables (LUTs) and registers for the three SoC configurations. For each cell, there is the absolute value for the target configuration and, in bold, the relative increment (percentage) compared to the same configuration with the hypervisor extensions disabled. We withhold data on other resources (e.g., BRAMs or DSPs) as the impact on its usage is insignificant.

According to Table 4, we can draw two main conclusions. First, there is an overall non-negligible cost to implement the hypervisor extensions support: an extra 11% LUTs, and 27-29% registers. Diving deeper, we observed that this overhead comes almost exclusively from two sources: the CSR and TLB modules. The CSR increase is explained given the number of HS and VS registers added by the H-extension specification. The increase in the TLB is mainly due to the widening of data store to hold guest-physical addresses (see Section 3) and the extra privilege-level and permission match and check complexity. The second important point is that, although the enhancements to the CLINT and PLIC reflect a large relative overhead, as these components are simple and small compared to the overall SoC infrastructure, there is no significant impact on the total hardware resources cost. Lastly, we can also highlight that increasing the number of available harts in the SoC does not impact the relative hardware costs.

6.3 Performance and Inter-VM Interference

To assess performance overhead and inter-hart / inter-VM interference, we used the MiBench Embedded Benchmark Suite. MiBench is a set of 35 benchmarks divided into six suites, each one targeting a specific area of the embedded
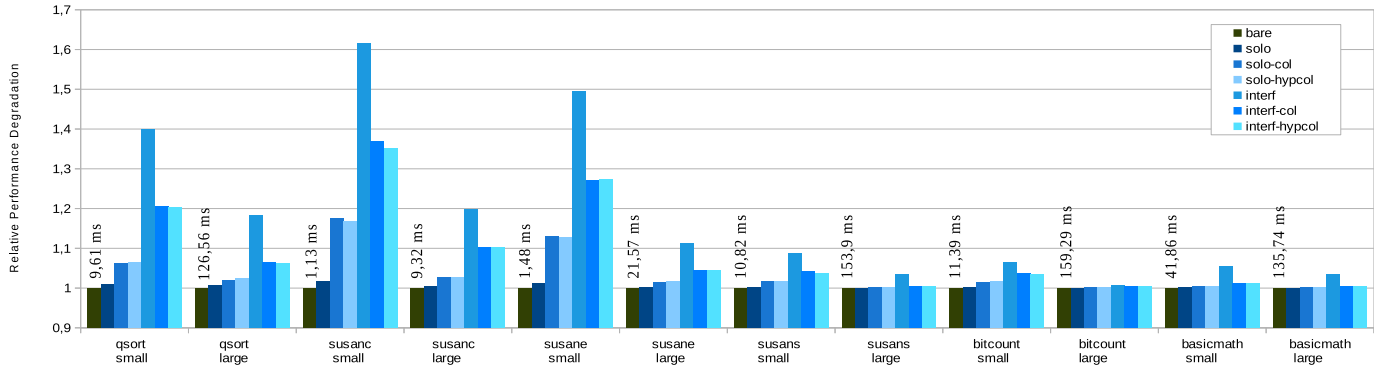
5. https://github.com/josecm/riscv-hyp-tests

Fig. 8. Relative performance overhead of MiBench automotive suite for different system configurations, relative to bare-metal execution. Absolute value indicated at the top of the solo bar.

market. We focus our evaluation on the automotive subset. The automotive suite includes three high memory-intensive benchmarks, i.e., more susceptible to interference due to LLC and memory contention (qsort, susan corners, and susan edges).

Each benchmark was ran for seven different system configurations targeting a six-core design: (i) guest native execution (bare); (ii) hosted execution (solo); (iii) hosted execution with cache coloring for VMs (solo-col); (iv) hosted execution with cache coloring for VMs and the hypervisor (solo-hypcol); (v) hosted execution under interference from multiple colocated VMs (interf); (vi) hosted execution under interference with cache coloring for VMs (interf-col); and (vii) hosted execution under interference with cache coloring for VMs and the hypervisor (interf-hypcol). Hosted scenarios with cache partitioning aim at evaluating the effects of partitioning micro-architectural resources at the VM and hypervisor level and to what extent it can mitigate interference. We execute the target benchmark in a Linux-based VM running in one core, and we add interference by running one VM pinned to five harts, each running a bare-metal application. Each hart runs an ad-hoc bare-metal application that continuously writes and reads a 1 MiB array with a stride equal to the cache line size (64 bytes). The platform's cache topology allows for 16 colors, each color consisting of 32 KiB. When enabling coloring, we assign seven colors (224 KiB) to each VM. The remaining two colors were reserved for the hypervisor coloring case scenario.

The experiments were conducted in Firesim, an FPGA-accelerated cycle-accurate simulator, deployed on an AWS EC2 F1 instance, running with a 3.2 GHz simulation clock. Fig. 8 presents the results as performance normalized to bare execution, meaning that higher values report worse results. Each bar represents the average value of 100 samples. For each benchmark, we added the execution time (i.e., absolute performance) at the top of the bare-metal execution bar.

According to Fig. 8, we can draw six main conclusions. First, hosted execution (solo) causes a marginal decrease of performance (i.e., average 1% overhead increase) due to the virtualization overheads of 2-stage address translation. Second, when coloring (solo-col and solo-hypcol) is enabled, the performance overhead is further increased. This extra overhead is explained by the fact that only about half of the L2 cache is available for the target VM, and that coloring precludes the use of superpages, significantly increasing

TLB pressure. Third, when the system is under significant interference (inter), there is a considerable decrease of performance, in particular, for the memory-intensive benchmarks, i.e., qsort (small), susan corners (small), and susan edges (small). For instance, for the susan corners (small) benchmark, the performance overhead increases by 62%. Fourth, we can observe that cache coloring can reduce the interference (inter-col and inter-hypcol) by almost 50%, with a slight advantage when the hypervisor is also colored. Fifthly, we can observe that the cache coloring, per se, is not a magic bullet for interference. Although the interference is reduced, it is not completely mitigated, because the performance overhead for the colored configurations under interference (inter-col and inter-hypcol) is different from the ones without interference (solo-col and solo-hypcol). Finally, we observe that the less memory-intensive benchmarks (i.e., basicmath and bitcount) are less vulnerable to cache interference and that benchmarks handling smaller datasets are more susceptible to interference.

The achieved results for RISC-V share a similar pattern to the ones assessed for Arm [2]. In our previous work [2], we have deeply investigated the micro-architectural events using a performance monitoring unit (PMU). As part of future work, we plan to conduct a deeper evaluation of micro-architectural interference effects while proposing additional mechanisms to help mitigate inter-VM interference.

6.4 Interrupt Latency

To measure interrupt latency and respective interference, we use a custom bare-metal benchmark application and a custom MMIO, auto-restart, incrementing timer that drives a PLIC interrupt input line. This application sets up the timer to trigger an interrupt at 100Hz (each 10 ms). The latency corresponds to the value read from the timer counter at the start of the interrupt handler. We invalidate the L1 instruction cache at each measurement using the *fence.i* instruction as we believe it is more realistic to assume this cache is not hot with regard to the interrupt handling or the hypervisor's interrupt injection code. We ran the benchmark using Firesim with the same platform and configurations described in Section 6.3. For guest configurations we took samples for both trap-and-emulate and PLIC interrupt direct injection. The average of the results obtained from 100 samples (the first 2 discarded) for each configuration are depicted in Fig. 9.
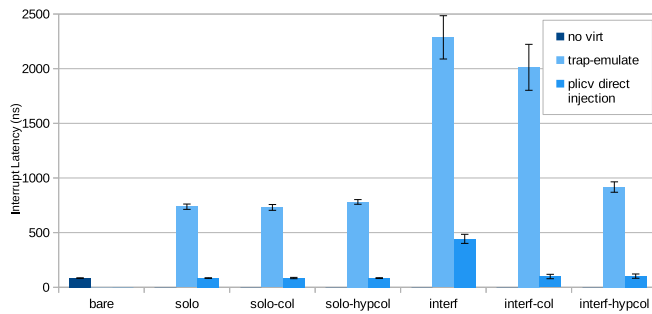
Fig. 9. Interrupt latency in nanoseconds for bare-metal execution and for guest execution with injection following a trap-and-emulate approach or direct injection with hardware support.

The interrupt latency for the bare (in Fig. 9, no virt) execution is quite low (approx. 80 ns) and steady. The trap-and-emulate approach introduces a penalty of an order of magnitude (740 ns) that is even more significant under interference (up to 2,280 ns, about 300%) both in average and standard deviation. Applying cache partitioning via coloring helps to mitigate this, which shows that most of the interference happens in the shared L2 LLC. The difference between inter-col and interf-hypcol shows that it is of utmost importance to assign dedicated cache partitions to the hypervisor: the interfering VM also interferes with the hypervisor while injecting the interrupt and not only with the benchmark code execution itself.

Fig. 9 also shows that the effect of the direct injection achieved with guest external interrupt and PLIC virtualization support can bring guest interrupt latency to near-native values. Furthermore, it shows only a fractional increase under interference (when compared to the trap-and-emulate approach) which can also be attenuated with cache coloring. As the hypervisor no longer intervenes in interrupt injection, for this case, it suffices to color guest memory. A small note is that the use of cache coloring does not affect the benchmark for solo execution configurations, given that the benchmark code is very small. Thus, the L1 and L2 caches can easily fit both the benchmark and hypervisor's injection code. Finally, we can conclude that with PLIC virtualization support, it is possible to significantly improve external interrupt latencies for VMs.

7 RELATED WORK

There is a broad spectrum of hardware virtualization related technologies and hypervisor solutions. Due to the extensive list of works in the literature, we will focus on (i) COTS and custom hardware virtualization technology and extensions and (ii) hypervisors and microkernels solutions for RISC-V.

Hardware Virtualization Technology. Modern computing architectures such as x86 and Arm have been adding added hardware extensions to assist virtualization to their CPUs for more than a decade. Intel has developed the Intel Virtualization Technology (Intel VT-x) [22], the Advanced Programmable Interrupt Controller (APIC) virtualization extension (APICv), and Intel Virtualization Technology for Directed I/O (Intel VT-d). Intel has also included nested virtualization hardware-based capabilities with Virtual Machine Control Structure (VMCS) Shadowing. Arm included the virtualization

extensions (Arm VE) since Armv7-A and developed additional hardware to the Generic Interrupt Controller (vGIC) for efficient virtual interrupt management. Recently, Arm has announced a set of extensions in the Armv8.4-A that includes the addition of secure virtualization support [23] and the Memory System Resource Partition and Monitoring (MPAM) [24]. There are additional COTS hardware technologies that have been leveraged to assist virtualization, namely the MIPS virtualization module [25], AMD Virtualization (AMD-V), and Arm TrustZone [4], [26]. The academia has also been focused on devising and proposing custom hardware virtualization support and mechanisms [27], [28]. Xu *et al.* proposed vCAT [27], i.e., dynamic shared cache management for multicore virtualization platforms based on Intel's Cache Allocation Technology (CAT). With NEVE [28], Lim *et al.* developed a set of hardware enhancements to the Armv8.3-A architecture to improve nested virtualization performance, which was then included in Armv8.4-A. Within the RISC-V virtualization scope, Huawei has also presented extensions both to the CLINT and the PLIC [29], including one of the timer extension proposals mentioned in section 4. Regarding the PLIC, comparing to our approach, their design is significantly more invasive and complex, as it uses memory-resident tables for interrupt/vhart and vhart/hart mappings. Their approach significantly complicates the PLIC implementation as it must become a bus-master. Nevertheless, this might bring some advantages, e.g., speeding-up VM context-switches. Furthermore, they also propose to extend the CLINT not only to include supervisor and virtual-supervisor timers but also to allow direct send and receive of HS/VS software interrupts (i.e., IPIs) without firmware/hypervisor intervention

Hypervisors and Microkernels for RISC-V. KVM [30] and Xvisor [21] were the first hypervisors adding support for the RISC-V H-extension in QEMU. KVM [30] is a type-2 hosted hypervisor integrated into Linux's mainline as of 2.6.20. KVM targets mainly enterprise virtualization setups for data centers and private clouds. Xvisor [21] is a type-1 monolithic hypervisor targeting embedded systems with soft real-time requirements. Both hypervisors are officially part of the RISC-V Software Ecosystem[6] and naturally have been used by technical groups as reference implementations to validate and evolve the H-extension. RVirt[7] is an S-mode trap-and-emulate hypervisor for RISC-V, written in Rust. Contrarily to KVM and XVisor, RVirt can run in RISC-V processors without hardware virtualization support. Diosix[8] is another lightweight bare-metal hypervisor written in Rust for RISC-V. Similar to RVirt, Diosix can run in RISC-V cores that lack the H-extension, leveraging the physical memory protection (PMP) to achieve isolation. Xtratum, a hypervisor primarily developed for safety-critical aerospace applications, has also recently been ported to support RISC-V ISA [31], following the same PMP-based concept for isolation as Diosix. Xen [32] and Jailhouse [33], two widely used open-source hypervisor solutions, have already given preliminary steps towards RISC-V support. However, as of this writing, upstream support for RISC-V is not yet available, but it is expected to be included in the foreseeable future.

---

6. https://github.com/riscv/riscv-software-list
7. https://github.com/mit-pdos/RVirt
8. https://diosix.org/

seL4, a formally verified microkernel, is also verified on RISC-V [34]. Other commercial microkernels already support RISC-V. Preeminent examples include the SYSGO PikeOS and the Wind River VxWorks.

8 DISCUSSION

Hypervisor Extension Specification State. The RISC-V H-extension is currently in its 0.6.1 version and is being developed within the privileged specification working group of RISC-V International, following a well-defined extension development lifecycle. The specification draft has been stable for quite some time and therefore is approaching a frozen state, after which it will enter a period of public review before finally being ratified. However, to enter a frozen state it will need both (i) open RTL core implementations suitable for deployment as soft-cores on FPGA platforms and (ii) hypervisor ports that exercise its mechanisms and provide feedback. Until the extensions are ratified, we do not expect any commercial IP or ASIC implementations to be available. With this work, we have contributed with one open RTL implementation but more are needed. Xvisor and KVM have been the reference open-source hypervisors used in the extension development process. We have further contributed with the Bao port, but the more hypervisor ports are available to evaluate the suitability of the H-extension for different hypervisor architectures, the better.

Missing Architectural and Virtualization Features. As discussed in the article, there are still some gaps in RISC-V, in particular with respect to virtualization. At the ISA level, features like cache management operations are needed. Fortunately, there is already a working group defining these mechanisms. At a platform level, timer and external interrupt virtualization support is needed. Our results show the importance of these mechanisms to achieve low and deterministic interrupt latency in virtualized real-time systems. There are already efforts within the RISC-V community to provide this support: a new extension proposal is on the fast track to include dedicated timers for HS- VS-modes; and a new interrupt controller architecture featuring support for message-signaled interrupts (MSI) and virtualization support is under development within the privileged specification working group. Another missing component critical for virtualization is the IOMMU. An IOMMU is needed to implement efficient virtualization, by allowing the direct assignment of DMA-capable devices to VMs, while guaranteeing strong isolation between VMs and the hypervisor itself. Static partitioning hypervisors such as Bao completely depend on IOMMU, as they do not provide any kind of device emulation and only pass-through access. At the moment, in a RISC-V platform, a Bao guest that wishes to use a DMA device must have all its memory configured with identity mapping. Unfortunately, this still completely breaks encapsulation, serving only for experimentation and demonstration purposes, not being suitable for production.

Multi-Core Interference Mitigation. In Section 6, we have demonstrated something well-understood and documented in the literature [2], [5], [8], [11], [12], [13], i.e., that (i) in multi-core platforms there is significant inter-core interference due to shared micro-architectural resources (e.g., caches, buses, memory controllers), (ii) which can be minimized by mechanisms such as page coloring used to partition shared caches. Other techniques such as memory bandwidth reservations [11] and DRAM bank partitioning [35] can minimize interference further ahead in the memory hierarchy. These partitioning mechanisms are important in embedded mixed-criticality systems both from the security and safety perspectives by protecting against side-channel attacks and guaranteeing determinism and freedom-from-interference required by certification standards (e.g., ISO26262). They are also useful for server systems by helping to guarantee quality-of-service (QoS) and increase overall utilization [36]. However, software-based approaches typically have significant overheads and increase the trusted computing base (TCB) complexity. Academic works such as Hybcache [37] or the bandwidth regulation unit (BRU) [13] propose the implementation of this kind of mechanisms in RISC-V cores (Ariane [38] and Rocket respectively). SafeSU [39] provides similar features by relying on a hardware statistics unit that measures inter-core interference in commercial space-graded RISC-V MPSoC. SiFive has provided cache partitioning mechanisms in hardware via way-locking [40]. We have started experimenting with these mechanisms using Bao and will present our findings in future work. During this work, we found it would be useful to have a standard set of mechanisms and interfaces to rely on. We argue that RISC-V is also missing a standard extension to provide such facilities. Other ISAs have already introduced these ideas, e.g., Intel's CAT and Arm's MPAM [24]. MPAM functionality is also extended to other virtualization-critical system-bus masters including the GIC and the SMMU (Arm's interrupt controller and IOMMU, respectively), something that should also be taken into account when developing similar RISC-V specifications.

Alternative Partitioning Approach. Even without virtualization support, it is possible to implement static partitioning in RISC-V leveraging the trap-and-emulate features described in Section 2 and using the PMP for memory isolation instead of two-stage translation. The PMP is a RISC-V standard component that allows M-mode software to white-list physical address space regions on a per-core basis. This results in a kind of para-virtual approach, as the guest must be aware of the full physical address space and possibly recompiled for different system configurations. To provide direct assignment of DMA devices, the host platform would also need to provide IOPMPs (akin to IOMMU, without translation), which is a specification already on course. Furthermore, the hypervisor would be forced to flush micro-architectural state such as TLBs or virtual caches at each context switch resulting in significant performance overheads. The use of VMIDs, part of the H-extension, tackles this issue. Notwithstanding, this is not a real problem for statically partitioned systems. Thus, once there is no commercial hardware featuring the H-extension available in the market, this is the approach of some of the hypervisors mentioned in Section 7. We are currently developing a customized version of Bao to run in RISC-V platforms without H-extension support (e.g., Microchip PolarFire SoC Icicle or the upcoming PicoRio). Nevertheless, we believe the hypervisor extension is still a better primitive for implementing these systems, given the higher flexibility and scalability it provides.

9 CONCLUSION

In this article, we have presented the first implementation of the RISC-V H-extension in a real RISC-V core, i.e., Rocket core. We have also proposed a set of hardware enhancements to the interrupt controller and the timer infrastructure aiming at tackling mixed-criticality systems requirements of minimal latencies, determinism and predictability. To validate and evaluate our hardware implementation, we have also ported the Bao open-source hypervisor to RISC-V. We achieved functional verification of our implementation in a Verilator-generated simulator and a Zynq UltraScale+ MPSoC ZCU104 FPGA. We carried out an extensive set of experiments in FireSim, a cycle-accurate simulator, to assess performance, inter-VM interference, and interrupt latency. The results demonstrated that the H-extension, per se, introduces a reduced performance penalty, but without additional hardware support interference and interrupt latency can impose a prohibitive cost for MCSs. Our proposed architectural enhancements considerably minimize these effects, by reducing interrupt latency and interference by an order of magnitude. Lastly, we discussed identified gaps existing in RISC-V with regard to virtualization and we outlined internal ongoing efforts within RISC-V virtualization. Our hardware design was made freely available for the RISC-V community and is currently the single reference implementation available to ratify the H-extension specification.

REFERENCES

[1] G. Heiser, "Virtualizing embedded systems - Why bother?," in *Proc. 48th ACM/EDAC/IEEE Des. Autom. Conf.*, 2011, pp. 901–905.

[2] J. Martins, A. Tavares, M. Solieri, M. Bertogna, and S. Pinto, "Bao: A lightweight static partitioning hypervisor for modern multicore embedded systems," in *Proc. Workshop Next Gener. Real-Time Embedded Syst.*, 2020, vol. 77, pp. 3:1–3:14.

[3] M. Bechtel and H. Yun, "Denial-of-service attacks on shared cache in multicore: Analysis and prevention," in *Proc. IEEE Real-Time Embedded Technol. Appl. Symp.*, 2019, pp. 357–367.

[4] S. Pinto, H. Araujo, D. Oliveira, J. Martins, and A. Tavares, "Virtualization on TrustZone-enabled microcontrollers? Voilà!," in *Proc. IEEE Real-Time Embedded Technol. Appl. Symp.*, 2019, pp. 293–304.

[5] T. Kloda, M. Solieri, R. Mancuso, N. Capodieci, P. Valente, and M. Bertogna, "Deterministic memory hierarchy and virtualization for modern multi-core embedded systems," in *Proc. IEEE Real-Time Embedded Technol. Appl. Symp.*, 2019, pp. 1–14.

[6] C. Dall and J. Nieh, "KVM/ARM: The design and implementation of the linux ARM hypervisor," in *Proc. 19th Int. Conf. Architectural Support Program. Lang. Oper. Syst.*, 2014, pp. 333–348.

[7] P. Burgio *et al.*, "A software stack for next-generation automotive systems on many-core heterogeneous platforms," *Microprocessors Microsystems*, vol. 52, pp. 299–311, 2017.

[8] M. Xu *et al.*, "Holistic resource allocation for multicore real-time systems," in *Proc. IEEE Real-Time Embedded Technol. Appl. Symp.*, 2019, pp. 345–356.

[9] M. Bechtel and H. Yun, "Exploiting DRAM bank mapping and Huge-Pages for effective denial-of-service attacks on shared cache in multicore," in *Hot Topics Sci. Secur. Symp. (HotSoS'20)*, Lawrence, KS, New York, NY, USA: ACM, Apr. 2020, p. 2., doi: 10.1145/3384217.3386394.

[10] Q. Ge, Y. Yarom, D. Cock, and G. Heiser, "A survey of micro-architectural timing attacks and countermeasures on contemporary hardware," *J. Cryptographic Eng.*, vol. 8, no. 1, pp. 1–27, Apr. 2018.

[11] H. Yun, G. Yao, R. Pellizzoni, M. Caccamo, and L. Sha, "MemGuard: Memory bandwidth reservation system for efficient performance isolation in multi-core platforms," in *Proc. IEEE 19th Real-Time Embedded Technol. Appl. Symp.*, 2013, pp. 55–64.

[12] R. Mancuso, R. Dudko, E. Betti, M. Cesati, M. Caccamo, and R. Pellizzoni, "Real-time cache management framework for multicore architectures," in *Proc. IEEE 19th Real-Time Embedded Technol. Appl. Symp.*, 2013, pp. 45–54.

[13] F. Farshchi, Q. Huang, and H. Yun, "BRU: Bandwidth regulation unit for real-time multicore processors," in *Proc. IEEE Real-Time Embedded Technol. Appl. Symp.*, 2020, pp. 364–375.

[14] K. Asanović and D. A. Patterson, "Instruction sets should be free: The case for RISC-V," Univ. California, Berkeley, Berkeley, CA, Tech. Rep. UCB/EECS-2014–146, 2014.

[15] A. Waterman, K. Asanović, and J. Hauser, "The RISC-V instruction set manual volume II: Privileged architecture, document version 1.12-draft," *RISC-V Foundation*, 2020.

[16] K. Asanović *et al.*, "The rocket chip generator," Univ. California, Berkeley, Berkeley, CA, Tech. Rep. UCB/EECS-2016–17, 2016.

[17] S. Karandikar *et al.*, "FireSim: FPGA-accelerated cycle-exact scale-out system simulation in the public cloud," in *Proc. ACM/IEEE 45th Annu. Int. Symp. Comput. Archit.*, 2018, pp. 29–42.

[18] G. J. Popek and R. P. Goldberg, "Formal requirements for virtualizable third generation architectures," *Commun. ACM*, vol. 17, no. 7, pp. 412–421, Jul. 1974.

[19] G. Klein *et al.*, "SeL4: Formal verification of an OS kernel," in *Proc. SIGOPS 22nd Symp. Oper. Syst. Princ.*, 2009, pp. 207–220.

[20] J. Bachrach *et al.*, "Chisel: Constructing hardware in a scala embedded language," in *Proc. Des. Autom. Conf.*, 2012, pp. 1216–1225.

[21] A. Patel, M. Daftedar, M. Shalan, and M. W. El-Kharashi, "Embedded hypervisor xvisor: A comparative analysis," in *Proc. Euromicro Int. Conf. Parallel Distrib. Netw.-Based Process.*, 2015, pp. 682–691.

[22] R. Uhlig *et al.* , "Intel virtualization technology," *Computer*, vol. 38, no. 5, pp. 48–56, 2005.

[23] Arm Ltd., "Isolation using virtualization in the Secure world Secure world software architecture on Armv8.4," 2018. [Online]. Available: https://armkeil.blob.core.windows.net/developer/Files/pdf/Isolation%20using%20virtualization%20in%20the%20Secure%20World_Whitepaper.pdf

[24] Arm Ltd., "Arm architecture reference manual supplement - Memory system resource partitioning and monitoring (MPAM), for Armv8-A," 2018. [Online]. Available: https://documentation-service.arm.com/static/5f8da597f86e16515cdb85fb?token=

[25] C. Moratelli, S. Johann, and F. Hessel, "Exploring embedded systems virtualization using MIPS virtualization module," in *Proc. ACM Conf. Comput. Frontiers*, 2016, pp. 214–221.

[26] S. Pinto and N. Santos, "Demystifying arm TrustZone: A comprehensive survey," *ACM Comput. Surv.*, vol. 51, no. 6, Jan. 2019, Art. no. 130.

[27] M. Xu, L. Thi, X. Phan, H. Choi, and I. Lee, "vCAT: Dynamic cache management using CAT virtualization," in *Proc. IEEE Real-Time Embedded Technol. Appl. Symp.*, 2017, pp. 211–222.

[28] J. T. Lim, C. Dall, S.-W. Li, J. Nieh, and M. Zyngier, "NEVE: Nested virtualization extensions for ARM," in *Proc. 26th Symp. Oper. Syst. Princ.*, 2017, pp. 201–217.

[29] S. Zhao, "Trap-less virtual interrupt for KVM on RISC-V," in *KVM Forum*, 2020. [Online]. Available: https://riscv.org/news/2020/12/trap-less-virtual-interrupt-for-kvm-on-risc-v-by-siqi-zhao-kvm-forum/

[30] U. Lublin, Y. Kamay, D. Laor, and A. Liguori, "KVM: The Linux virtual machine monitor," in *Proc. Linux Symp.*, 2007. [Online]. Available: http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.488.2278

[31] F. Gómez *et al.*, "De-RISC–Dependable real-time infrastructure for safety-critical computer systems," *ADA USER*, vol. 41, no. 2, 2020, Art. no. 107.

[32] J. Hwang *et al.*, "Xen on ARM: System virtualization using xen hypervisor for ARM-based secure mobile phones," in *Proc. 5th IEEE Consumer Commun. Netw. Conf.*, 2008, pp. 257–261.

[33] R. Ramsauer, J. Kiszka, D. Lohmann, and W. Mauerer, "Look mum, no VM exits!(almost)," *CoRR*, vol. abs/1705.06932, 2017. [Online]. Available: http://arxiv.org/abs/1705.06932

[34] G. Heiser, "seL4 is verified on RISC-V!" in *Proc. RISC-V Int.*, 2020. [Online]. Available: https://riscv.org/blog/2020/06/sel4-is-verified-on-risc-v/

[35] H. Yun, R. Mancuso, Z. Wu, and R. Pellizzoni, "PALLOC: DRAM bank-aware memory allocator for performance isolation on multicore platforms," in *Proc. IEEE Real-Time Embedded Technol. Appl. Symp.*, 2014, pp. 155–166.

[36] D. Lo, L. Cheng, R. Govindaraju, P. Ranganathan, and C. Kozyrakis, "Heracles: Improving resource efficiency at scale," in *Proc. ACM/IEEE 42nd Annu. Int. Symp. Comput. Archit.*, 2015, pp. 450–462.

[37] G. Dessouky, T. Frassetto, and A.-R. Sadeghi, "HYBCACHE: Hybrid side-channel-resilient caches for trusted execution environments," in *Proc. 29th USENIX Conf. Secur. Symp.*, 2020, pp. 451–468.

[38] 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 Trans. Very Large Scale Integr. Syst.*, vol. 27, no. 11, pp. 2629–2640, Nov. 2019.

[39] G. Cabo *et al.*, "SafeSU: An extended statistics unit for multicore timing interference," in *Proc. IEEE Eur. Test Symp.*, 2021, pp. 1–4.

[40] L. Gwennap, "Deterministic processing for mission-critical applications deterministic processing for mission-critical applications," Linley Group, Mountain View, CA, Tech. Rep. Sep. 2020. [Online]. Available: https://www.linleygroup.com/uploads/sifive-deterministic-processing-wp.pdf

Bruno Sá received the master's degree in electronics and computer engineering with specialization in Embedded Systems and Automation, Control and Robotics. He is currently working toward the PhD degree in the Embedded Systems Research Group, University of Minho, Portugal. His research interests include operating systems, virtualization for embedded systems, computer architectures, IoT systems, and artificial intelligence.
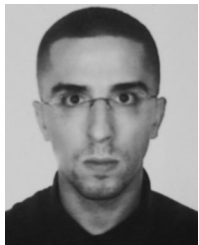
José Martins received the master's degree in electronics and computer engineering, during his master's, he also was a visiting student at the University of Wurzburg, Germany. He is currently working toward the PhD degree and teaching assistant in the Embedded Systems Research Group, University of Minho, Portugal. He has a significant background in operating systems and virtualization for embedded systems. Over the last few years, he has also been involved in several projects on the aerospace, automotive, and video industries. He is the main author of the Bao hypervisor.

Sandro Pinto received the PhD degree in electronics and computer engineering. He is currently a research scientist and invited professor at the University of Minho, Portugal. During his PhD, he was a visiting researcher at the Asian Institute of Technology, Thailand, University of Wurzburg, Germany, and Jilin University, China. He has a deep academic background and several years of industry collaboration focusing on operating systems, virtualization, and security for embedded, cyber-physical, and IoT-based systems. He has published several scientific papers in top-tier conferences/journals and is a skilled presenter with speaking experience in several academic and industrial conferences.

▷ **For more information on this or any other computing topic, please visit our Digital Library at** www.computer.org/csdl.