

Hardware Support for Deterministic User-Level Interrupts in Real-Time Embedded Systems

Hongbin Yang
Shandong University
Qingdao, Shandong, China
hongbinyang@mail.sdu.edu.cn

Huanle Zhang
Shandong University
Qingdao, Shandong, China
dtczhang@sdu.edu.cn

Runyu Pan*
Shandong University
Qingdao, Shandong, China
rypan@sdu.edu.cn

Abstract

Modern real-time embedded systems demand strong component isolation for safety and security. Normally, interrupts are handled in kernel mode before they are passed on to the application, which induces high latency if isolated components need to react to external events. Thus, hardware-assisted, low-latency user-level interrupts are highly desirable. Nevertheless, existing hardware mechanisms only optimize average-case performance while neglecting worst-case latency for sensitive applications including autonomous vehicles and robots. To overcome this limitation, we propose a novel open-source¹ hardware extension that integrates spatial isolation setup, execution budget management, cross-domain nesting, and accelerated context switching for user-level interrupts with tight latency. It enables direct, deterministic delivery upon user-level interrupt arrival—without kernel intervention—even when the target process is dormant. Across the entire design space, the best-performing design reduces worst-case latency by over 50×, while the lowest-overhead design incurs only a 23% increase in core area (2% of total die area) and a 3% increase in dynamic power. To the best of our knowledge, this is the first integrated mechanism to guarantee multi-tenant user-level interrupt delivery with a nanosecond-scale yet bounded worst-case latency. It provides a holistic solution for developing secure, dependable, and low-latency cyber-physical systems in real-time consolidation scenarios.

CCS Concepts: • Computer systems organization → Real-time system architecture; Real-time operating systems; Embedded hardware; Embedded software.

Keywords: User-level interrupts; Latency; Jitter; Real-time systems; Security; Safety; Isolation; RISC-V

*Corresponding author.

¹[10.6084/m9.figshare.33511450](https://doi.org/10.6084/m9.figshare.33511450)



This work is licensed under a Creative Commons Attribution 4.0 International License.

ASPLOS '27, Heraklion, Greece.

© 2027 Copyright held by the owner/author(s).

ACM ISBN 979-8-4007-3012-2/2027/04

<https://doi.org/10.1145/3845814.3850053>

ACM Reference Format:

Hongbin Yang, Huanle Zhang, and Runyu Pan. 2027. Hardware Support for Deterministic User-Level Interrupts in Real-Time Embedded Systems. In *Proceedings of the 32nd ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 1 (ASPLOS '27)*, April 11–15, 2027, Heraklion, Greece. ACM, New York, NY, USA, 26 pages. <https://doi.org/10.1145/3845814.3850053>

1 Introduction

With the rise of autonomous vehicles [6, 100], embodied intelligence robots [32], and smart medical devices [8, 72], real-time systems execute increasingly complex binaries composed of software components from heterogeneous sources. This diversity arises from code reuse, multi-stakeholder development, and cost-saving consolidation, deviating significantly from the traditional paradigm where a single team maintains control over the codebase [6, 68].

When diverse components coexist within an unisolated address space, faults or compromises can act as *trampolines*, enabling system-wide fault propagation or hijacking. To safeguard these systems, various component isolation strategies have been proposed [14, 40, 43]. Of these approaches, the microkernel scheme [24, 50, 68, 88], e.g., TockOS [48], has gained popularity due to its ease of adoption and has helped to secure *millions of systems* [80]. It isolates each component as a process, while a privileged microkernel manages scheduling, memory allocation, and device access. The processes are isolated (1) *spatially* so that they only access a specific set of memory and peripherals that are explicitly allocated to them, and (2) *temporally* so that they cannot exceed their execution budget and interfere with the timely execution of other components by monopolizing the processor. Each process may host real-time applications: a typical use case is an integrated flight controller, which concurrently handles high-speed communication and low-jitter rotor control with distinct processes [11].

However, a critical challenge arises with latency-sensitive real-time interrupt handling for each process. Ideally, interrupt handlers belonging to untrusted components should run at the user-level to stay spatially and temporally confined. However, forwarding interrupts from the kernel to user processes inflates latencies by *an order of magnitude*, a latency overhead that is prohibitive for tasks where deterministic response times are *non-negotiable*. For example, in vehicle

Table 1. Comparison with representative works; in this table, isolation means attack resilience, rather than functional safety alone.

Work	Spatial Isolation	Temporal Isolation	Multi-process	Interrupt Latency	Determinism Level
TockOS [48]	✓	✓	✓	High	Medium
Hopter [56]	✗	✗	✗	Medium	Medium
CV32RT [9]	✗	✗	✗	Low	High
RTOSUnit [77]	✗	✗	✗	Low	High
UINTR [7, 35]	✓	✗	✓	Low	Low
SBI [67]	✓	✓	✗	Low	High
This work	✓	✓	✓	Low	High

engine control or high-speed servo drives, high latency can cause *large control errors or even system failure*.

Recent hardware like Intel’s UINTR [35] and RISC-V’s N extension [98] deliver interrupts to the user-level, but they are *nondeterministic*. They rely on the target process being scheduled to take the *fast path*; when the target process is dormant, the kernel must intervene to switch to it, creating a *slow path*. This produces an unpredictable bimodal latency distribution that improves only *average-case* throughput [7] and fails to enhance *worst-case* responsiveness (see §5.4). To make things worse, when the background runs most of the time, frequent foreground interrupts always take the slow path, wasting more energy per interrupt, as well as *reducing the processing time available to the background task* (see §5.5). Another approach is TrustZone-M-based SBI [67]. Although it preserves the microkernel’s support for multiple processes, it supports deterministic user-level interrupt acceleration *for only one of those processes* per core. Interrupts belonging to other unaccelerated processes still incur kernel latencies, rendering it impractical for realistic multi-domain real-time consolidation scenarios.

Without full hardware support for deterministic user-level interrupts, developers are forced down a *vicious waterfall*: to obtain tight *worst-case* latencies, interrupt handlers must be compiled into the kernel; and to prevent them from launching attacks, certification or software fault isolation is required, precluding the use of *native binary code*, which is the most efficient form.

To *remove this isolation-determinism trade-off*, we propose a novel hardware extension for deterministic user-level interrupt handling, which *satisfies real-time system requirements without compromising isolation*. It (1) enforces secure execution of user-level handlers via temporal and spatial confinement, (2) enables deterministic user-level interrupt delivery even when the target process is not scheduled, and (3) achieves low delivery latency comparable to that of kernel-level interrupts, with moderate area and power overheads.

Contributions. As shown in Table 1, this work presents a hardware extension that enables deterministic, low-latency, and secure multi-tenant user-level interrupt delivery by *activating each interrupt handler together with its spatial and*

temporal protection domains. The **individual sections** make the following *respective contributions*:

- **we identify an isolation-determinism trade-off (§1)** that we eliminate by *proposing a unified architectural abstraction for deterministic multi-tenant user-level interrupts* that (1) binds spatial and temporal protection domains to interrupts, (2) automates domain reconfiguration upon activation, (3) supports rapid nested cross-domain preemption, and (4) accelerates secure register context switching;
- **we explore the hardware design space (§3)** for the proposed deterministic user-level interrupt delivery abstraction, elucidating *possibilities with different trade-offs that reduce worst-case user-level interrupt latency by up to 50×*, when combined;
- **we propose a system-level abstraction (§3)** compatible with existing microkernel operating systems to *efficiently leverage the proposed hardware mechanism* with moderate adaptation effort;
- **we implement select variants (§4)** on the RISC-V architecture using a 45 nm process, discussing *the Performance, Power, and Area (PPA) trade-offs, demonstrating minimum area and power overheads of only 23% and 3%, respectively*;
- **we evaluate key implementations (§5)** with FPGA prototyping, demonstrating *up to 60× jitter reduction on latency-sensitive tasks and 10× processor wastage reduction on colocated best-effort tasks* in real-world applications.

To the best of our knowledge, this work is the first integrated mechanism (§6) that *bridges the determinism gap between swift user-level interrupt delivery and real-time responsiveness*, enabling consolidation of multiple real-time tasks without sacrificing interrupt latency.

2 Background and Related Work

2.1 User-level Interrupt: Nondeterministic

The emergence of kernel-bypassing technologies has transformed high-performance cloud computing. User-level scheduling [25], networking [20], and storage stacks [84] shift functions traditionally handled by the kernel to the user space, eliminating unnecessary processor state transitions, leading to improved system efficiency. A recently available feature is user-level interrupt delivery, which delivers interrupts directly to the user space without kernel intervention.

The most notable implementations are Intel’s UINTR extension [35] and RISC-V’s N extension [73, 98]. The former is commercially available in x86-64 processors codenamed “Sapphire Rapids”, while the latter is an unratified draft at the time of writing. Intel’s UINTR extension enables user-level interrupt delivery by having the kernel configure a User Interrupt Target Table (UITT). A sender thread issues a SENDUIPI instruction with a UITT offset, and the hardware resolves the target thread, triggering an Inter-Processor Interrupt (IPI) via the Advanced Programmable Interrupt Controller (APIC) to activate the handler. While designed to reduce

cross-core communication overhead, the original Sapphire Rapids implementation restricts senders to threads, excluding devices [7, 35], which limits its relevance for real-time systems. In contrast, the S, H, and N extensions of RISC-V introduce a unified interrupt delegation model, allowing delegation registers to route interrupts to less privileged modes regardless of whether the sender is a thread or a device.

Some research has accumulated around user-level interrupts, encompassing applications like accelerated user-level scheduling [52] and tail latency optimization [37]. Additionally, one study focused on optimizing hardware for highly efficient interrupt delivery [7]. In a broader context, user-level interrupts can be seen as a subtype of *deprivileging interrupts*, delivering interrupts to a lower privilege level instead of a higher one. This definition shares similarities with virtualization extensions like Intel's VT-d posted interrupts, which have been leveraged to achieve performance boosts similar to those found in user-level interrupts [39]. However, existing research on user-level interrupts prioritizes average performance over deterministic latency, as the kernel still manages scheduling and protection domain switching.

2.2 Hardware Thread Schedulers: No Isolation

The real-time community has long been using hardware schedulers to bring down scheduling latencies, e.g., [17, 62, 63, 77, 86, 90]. In this scheme, the hardware may handle scheduling decisions, register context switching, or even synchronization.

When scheduling algorithms are implemented in hardware, they have a low and deterministic latency, especially for algorithms like Earliest Deadline First (EDF) [91], or for systems with many processor cores [15, 42, 57, 66, 93, 101, 107]. However, pure hardware schedulers require customized hardware and fixed task parameters, which can be limiting. To overcome this, some research uses prioritization logic of interrupt controllers [10, 12, 23, 31] to perform scheduling.

When register context switching is implemented in hardware, software saving or restoring sequences are bypassed, resulting in lower latency, as demonstrated in [71, 77]. Two approaches exist: (1) automatic register pushing and popping on interrupt entries and returns [9, 34, 102], and (2) providing additional partial [53] or full [17, 19, 27, 36, 64, 99, 101] register banks for different processing contexts. Most register context switch mechanisms don't involve process switches, except for x86's outdated task gate [35]. However, this mechanism is poorly optimized at the hardware level and introduces unpredictable latencies due to Translation Look-aside Buffer (TLB) flushing, both of which probably led to its removal in 64-bit mode.

It is noteworthy that certain solutions [67, 70] leverage TrustZone-M to accelerate user-level interrupt delivery. However, they have multiple downsides, including (1) the lack

of multi-process user-level interrupt capability, (2) the absence of a defined mechanism for user-level interrupt nesting and preemption, (3) the tight coupling between execution budgets and host processes precluding flexible spatial and temporal domain composition, and (4) the dependence on ARMv8-M-specific hardware features, making them less suited to general-purpose real-time consolidation. As these works reuse TrustZone-M [67, 70], they do not address the unique challenges that only surface when multi-tenant support is needed. We leave the combination of our extension with a TrustZone-M-like security extension for future work.

To sum up, we are not aware of any multi-tenant consolidating hardware mechanism that performs all of register context switching, budget-enforcing scheduling, and access permission switching in a swift fashion.

2.3 Microkernels: High Latency Overhead

Modern real-time processors feature privilege levels and Physical Memory Protection (PMP; known as the Memory Protection Unit or MPU in ARM), enabling microkernel deployment. In contrast to a general-purpose Memory Management Unit (MMU), PMP omits hardware address translation and TLB structures, using software-managed, segment-based physical address protection with deterministic access latency. Such a choice is deliberate as TLB misses introduce unpredictable latency; however, this restricts these systems to Single Address Space Operating Systems (SASOSes).

Many notable SASOS microkernels [24, 48, 50, 68, 88] have leveraged PMP as an isolation facility. Besides, other works have explored PMP-based isolation approaches [12, 65, 75] with similar takes. However, these works all suffer from significant execution overhead that arises from processor mode switches, with interrupt latencies reaching *hundreds of clock cycles or more*. While this is acceptable for less demanding cases, such delays exceed the tolerance for scenarios where tens of clock cycles are considered the limit.

2.4 Alternative Methods: Multiple Limitations

In addition to microkernel-based isolation approaches, software surrogates can also set up isolation within a single address space. Some approaches rely solely on software and can go without hardware isolation facilities of any sort. This includes (1) Hopter [56], ARMor [105], CRT-C [40] and Minion [41] that leverage bounds checking or compiler-based verifications, and (2) eBPF [43, 103], Java [58], Python [59], Javascript [22, 61] and even WebAssembly [28, 69] that rely on managed execution environments. Some approaches resort to a combination of software and hardware, such as TockOS [48], uXOM [46] and Pip-MPU [14], for efficiency. Outside the embedded realm, researchers have explored isolation facilities that combine software with Intel MPK, ARM memory tagging, or hardware capabilities [2, 3, 18, 30, 44, 49, 76, 81, 95, 97]. Notably, these efforts also focus on a single address space such as a process or unikernel.

While these approaches have their merits, they often come with significant limitations. Some require toolchain modifications that are unfeasible in a production setting, while others rely on tedious manual isolation boundary annotations. Additionally, these approaches often restrict the language selection, neglect availability concerns, assume asymmetric isolation, cannot be linked against third-party binaries or require binary scanning, adding complexity and overhead. Furthermore, some of them incur high execution overheads and large memory footprints, making them suitable only for resource-rich environments. Compared with these surrogates, our extension shifts the trusted mechanism from an updatable but broader software stack to a narrow, fixed hardware dispatch path with bounded state and well-defined interfaces, making it more amenable to focused *formal verification*.

In addition to software methods, current hardware methods that achieve deterministic latency also exhibit severe limitations: (1) adding Field-Programmable Gate Arrays (FPGAs) that have high area, power, and cost overheads [45], (2) deploying application-specific peripherals that have limited programming flexibility [74], or (3) utilizing individual polling-based System-on-Chips (SoCs) for each latency-sensitive task, raising firmware management complexity [96].

3 Design Space Exploration

The primary objective of this extension is to ensure deterministic user-level interrupt delivery by entirely eliminating software mediation. Rather than committing to a single ad-hoc design, we *systematically explore the design space and refine it toward practical implementations*. To this end, the design pursues the following **goals**:

- G1: Deterministic latency.** User-level interrupt handlers must commence execution with a deterministic and low latency upon activation, regardless of the currently scheduled process, ensuring system responsiveness.
- G2: Spatial confinement.** User-level interrupt handlers must have their memory and device accesses confined to authorized memory regions, protecting system confidentiality and integrity.
- G3: Temporal confinement.** User-level interrupt handlers must execute within a rigidly enforced budget, preventing unbounded execution from interfering with other components and safeguarding system availability.
- G4: Minimal hardware overhead.** The extension must incur minimal silicon area and power overheads, respecting the tight Size, Weight, and Power plus Cost (SWaP-C) constraints of embedded real-time systems.
- G5: Ecosystem compatibility.** The extension must incur minimal invasiveness to the bus interface and execution pipeline of the original processor to ensure applicability across diverse architectures, and provide a companion

software abstraction to facilitate integration into existing microkernels.

Latency vs. area vs. power trade-off. We expect that lower user-level interrupt latency comes at the cost of increased hardware complexity and power consumption to support parallel microarchitectural operations. Nevertheless, we aim to characterize the latency–area–power trade-off and strike a balance that meets real-time requirements without excessive overheads.

Assumed threat model. We assume mutually distrusting user-level interrupt handlers that may attempt spatial (illegal access; harms confidentiality and integrity) or temporal (processor monopolization/denial of service; harms availability) violations.

Assumed task model. We model user-level interrupt handlers as run-to-completion aperiodic tasks hosted on deferrable servers [87] with strict budget constraints, which are enforced by the hardware extension.

3.1 Real-time Processor Core

Table 2. Comparison of typical real-time processors using official manuals. BP is Branch Prediction, PMP is the number of entries, interrupt latency is in CPU cycles, and CoreMark is per MHz when running from zero-wait-state memory. *No data in official manual.

Processor	Stages	BP	PMP	FPU	CoreMark	Latency
Cortex-M0	3	Static	None	No	2.33	16
Cortex-M3	3	Static	8	No	3.45	12
Cortex-M4	3	Static	8	Single	3.54	12
DSP C28x	8	Static	None	Single	1.33*	16
Cortex-R4	8	Dynamic	12	Single	2.83	20
QingKe V4	3	Dynamic	4	Single	3.47	*

To explore the design space, we develop our own baseline core rather than reusing a full-blown processor. Since the mechanism only touches interrupt handling logic, a bare-minimum core helps us to isolate the cost of the mechanism itself instead of burying it inside potential unrelated microarchitectural complexity. In this light, we design a RISC-V core with a 3-stage in-order pipeline, static branch prediction, a single-precision FPU, an 8-entry PMP, and a performance of around 3 CoreMark/MHz [26], which is consistent with the popular cores shown in Table 2. We have chosen RISC-V due to its openness and flexibility. We focus on a single-core architecture in this study and reserve multi-core as future work, as supporting it would require global interrupt routing and arbitration [10] in addition to per-core context management.

Note that this core is without an MMU, superscalar capability, or caches, aligning with the area and power performance requirements of major real-time applications. Since our extensions only modify interrupt handling logic, for cores employing instruction reordering or speculative execution, only *stalling the fetch and draining the reorder buffer* are needed to ensure precise architectural state updates; the

trade-offs of eager flushing or tracking [7] are *beyond our scope*. For readers interested in extending this work to out-of-order application processors in edge and cloud computing scenarios, §B.14 in the appendix demonstrates the portability and simplicity of our proposed extension.

3.2 Hardware Extension Synopsis

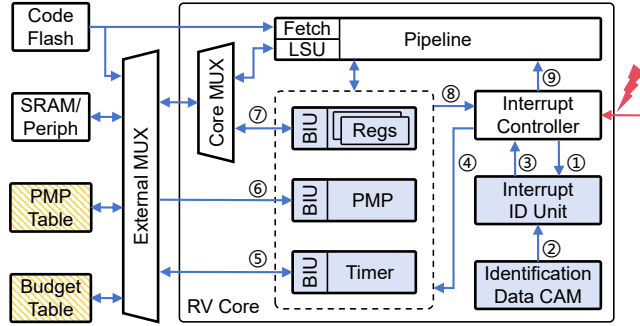


Figure 1. Proposed extension (V5 variant; for detailed description, see §4.8) architecture and data flow. The extra memory blocks are in striped yellow, and the intra-core hardware extension is in blue.

The extension confines each deterministic user-level interrupt handler to (1) a spatial protection domain described by a set of PMP configurations that reflect process access permissions, and (2) a temporal protection domain enforced by a hardware budget countdown timer. Upon interrupt activation, the hardware switches to these domains, saves the full register set, and zeroizes it. The extension also distinguishes user-level interrupt sources from traditional kernel-level interrupt sources, maintaining backwards compatibility. We showcase the operation of the extension with a concrete implementation that we call the V5 variant (see §4.8), as shown in Figure 1.

Interrupt entry. Upon interrupt activation, the Interrupt Controller (IC) calls upon the Interrupt Identification Data Unit (IIDU) to determine ① whether it is a user-level interrupt. If it is, IIDU identifies ② the address of the PMP configuration and budget value from the Interrupt Identification Data (IID) it manages, which is a Content Addressable Memory (CAM) in this variant, and notifies the IC ③. Subsequently, the IC triggers ④ parallelizable hardware operations in which (1) the execution budget is loaded ⑤, in this variant, from Tightly Coupled Memory (TCM), (2) the PMP configuration is loaded ⑥ from its TCM as well, (3) the registers are saved by switching banks or spilling to memory ⑦ and are then zeroized. Once completed, a finish signal is sent ⑧ to the IC, after which it resumes ⑨ the interrupt vector and the timer countdown. When the interrupt vector executes, if (1) the PMP detects an out-of-bounds access, which indicates a spatial violation, or (2) the timer counts down to zero, which indicates a temporal violation, a user-level interrupt return is forced to maintain strong isolation.

Interrupt return. Upon interrupt return, the register context and PMP configuration are restored, with the remaining budget written back to the budget table ⑤. If spilling occurred during saving, the register context is restored from memory ⑦.

Low invasiveness. From the concrete example above, it may be observed that the extension exhibits modest hardware intrusion by design. It only

- modifies the interrupt controller to integrate the IID mechanism (§3.3),
- interfaces the PMP to enable configuration autoloading from memory (§3.4),
- introduces a countdown timer for bounding interrupt execution duration (§3.4), and
- taps into the register banks to enable hardware-accelerated context switching (§3.5),

all of which target modular, identifiable components without altering the core pipeline stages. We further explore the design space of each hardware modification and analyze the associated trade-offs before implementation. Additionally, since our extension supports kernel-preempting cross-domain nested user-level interrupts, we also elaborate on the nesting mechanism (§3.6) and exception semantics (§3.7).

3.3 User-level Interrupt Identification

Identifying which interrupt source should be directed to the user-level and what spatial and temporal protection domains it targets requires querying the hardware-managed IID. Instead of entangling this with the interrupt vector table and breaking compatibility with RISC-V standards, we *implement it as a standalone structure* managed by the IIDU. The design must balance lookup latency against silicon area, as this step is on the critical path.

IID location. Placing the IID in CPU registers ensures deterministic, single-cycle lookup for its target protection domain information, but increases the core area. Conversely, storing the IID in RAM saves logic area but introduces additional bus accesses, undermining latency.

IID organization. The IID maps interrupt numbers to two protection domain numbers and can be designed in two ways. A linear table indexed by interrupt number offers simplicity but requires substantial storage proportional to the vector table size. Alternatively, we *may use a compressed form*—suitable given that only select interrupts are user-level—to store explicit interrupt and domain address entries, necessitating the use of CAM for parallel lookups.

Table 3. IID design summary. Feasible options are in bold.

IID	In CPU	In RAM
As table	Infeasible; table too big to fit into CPU	More latency less core area
As CAM	Less latency more core area	Infeasible; CAM needs parallel lookup

Discussion. Combining these factors, we dismiss the area-prohibitive CPU-Table approach and the latency-prohibitive RAM-CAM approach, narrowing the design space to two viable options shown in Table 3.

3.4 PMP Table and Budget Table

The PMP table stores spatial confinements for each domain, while the budget table enforces temporal isolation. Both the PMP table and budget table are linear tables, indexed directly by the respective pointers in the IID entry of each interrupt. Each PMP table entry contains the PMP configuration to load into the PMP configuration registers when switching processes. Conversely, each budget table entry contains a 32-bit integer holding the execution budget in CPU clock cycles, which is loaded into the countdown timer when the handler starts executing. Given their size, both tables must reside in memory, necessitating a trade-off between bus interface complexity and the potential for bus contention.

Tables in the main SRAM. Given that both tables are consulted sequentially after the IID lookup, they can coexist with the IID in the same SRAM. However, parallelization of context saving and table accesses creates potential contention on the shared SRAM bus, introducing latency spikes. On the bright side, this approach requires no additional memory ports and maintains full compatibility with the original core’s Bus Interface Unit (BIU).

Tables in tightly-coupled memory. Alternatively, we can place the PMP and budget table in *one shared or two dedicated* SRAM blocks to eliminate bus contention. However, this requires one or even two additional memory ports, complicating the bus interface design. Fortunately, many processors *come architecturally with optional instruction or data TCM ports*, which we can take advantage of.

PMP table in flash. Similar to TCM, many SoCs feature a dedicated bus for read-only embedded code flash that offers throughput comparable to SRAM with higher access latency, which can accommodate the tables. However, the remaining budget must be written back to the budget table upon interrupt return to adhere to bandwidth-based servers, and the kernel may periodically replenish these budgets. Consequently, only the PMP table may reside in flash, while the budget table must be held in RAM.

Table 4. Protection table design summary.

Tables in	Advantages	Disadvantages
Main SRAM	Simplistic design; no changes to bus interfaces	Potential bus conflicts; longer latency
Data TCM	No potential bus conflicts; likely less latency	Requires TCM interface; complicates hardware
Code Flash (PMP Only)	Simplistic design; no potential bus conflicts	Flash cannot be modified; slightly longer latency

Discussion. All three options shown in Table 4 are feasible, each with its own trade-offs. However, we will only explore

the first two options in the implementation section, as flash is a restricted surrogate of TCM for our purposes.

3.5 Register Context Saving

To prevent untrusted user-level handlers from peeking into the register set of threads or handlers that belong to other processes, the hardware must save the full register set and zeroize it. The `mepc`, `mcause`, and `mtval` Control and Status Registers (CSRs), alongside the internal state of the potentially preempted interrupt, including its ID, priority, PMP entry pointer, and budget entry pointer, must also be saved. As we enforce quad-word alignment, this yields a total context size of 160 bytes with two reserved words. Given the size of the whole context, this is likely the *primary bottleneck* for latency, and *conflicts with other activities should be avoided*.

Context shadow banking. Similar to [77], we reserve a dedicated register bank for handling interrupts, thereby drastically reducing context switching latency by eliminating the need for register stacking. However, this approach (1) significantly increases the core area and (2) does not support nesting unless multiple banks are available.

Context hybrid spilling. To support *unbounded nesting without an unbounded area*, we introduce a lightweight fallback mechanism. When shadow banks are exhausted (or when none is available), the hardware automatically spills the context to a stack in memory. The spill can target SRAM—which may introduce bus conflicts and prolong interrupt latency—or utilize a dedicated TCM port.

Table 5. Register context design summary.

Context in	Extra banks with spilling	Stacking only (no banks)
Main SRAM	Spills less but more area; longer stacking latency; simple bus interface	Always spills but less area; longer stacking latency; simple bus interface
Dedicated TCM	Spills less but more area; shorter stacking latency; additional TCM interface	Always spills but less area; shorter stacking latency; additional TCM interface

Discussion. We explore all four options listed in Table 5. Since the number of banks primarily affects nesting depth before latency issues arise, we only vary the number of banks when examining its impact on the core area.

3.6 Protection Domain Saving

Since the extension preempts all underlying execution, including the kernel, the preempted protection domain information must be saved to avoid disturbing the underlying software and to remain compatible with the ecosystem. This includes (1) the kernel-managed PMP configuration, which reflects the permissions of the current process, and (2) the system timer value, which reflects the budget of the current kernel-managed thread. Furthermore, user-level interrupts must be capable of preempting one another.

User-level interrupt being preempted. This is a straightforward case: since all PMP configurations for user-level

interrupts are stored in the PMP table, they simply need to be reloaded upon resumption. Regarding the budget timer, however, there is a subtle distinction: when preemption occurs, the remaining time must be saved to the budget table to account for the time expended, and subsequently restored when the interrupt resumes.

Kernel or thread being preempted. This presents a complex scenario, as the PMP at the point of preemption is managed by the kernel. To address this, we can either (1) provide an extra bank of shadow PMP registers or (2) spill the PMP registers to memory. Additionally, if the system timer directly tracks the remaining budget of preempted kernel-managed threads, it must be paused while user-level interrupts execute.

3.7 Ancillary Hardware Mechanisms

Asynchronous exception handling. Handling exceptions in our user-level interrupts poses a challenge because they preempt kernel execution. As exception handlers are themselves kernel code, synchronous exception handling for user-level interrupts may cause unsafe kernel reentrancy. To address this issue, we force the user-level interrupt handlers to return immediately upon any exception or budget exhaustion, with the cause recorded in an *optional* dedicated CSR which the kernel may inspect asynchronously after all nested user-level interrupts have returned. This prevents exception handling on-the-spot; however, it lifts the blocking of the delivery of subsequent user-level interrupts for the full duration of exception processing, which a malicious interrupt can leverage to cause undue interference when exceptions are handled synchronously. In the end, as user-level interrupts *are part of the application*, it is the responsibility of the application to ensure their correct run-to-completion operation.

Tail-chaining opportunity. The context switch for user-level interrupts is heavier than for kernel-level interrupts, as it entails protection domain switching. To minimize switches during back-to-back execution, we may employ the tail-chaining mechanism [102] to elide the previous interrupt's context restore and the next interrupt's context save if they target the same process. Although this improves *average-case* performance, we do not implement it since optimizing *worst-case* latency is the focus of this work.

FPU context lazy saving opportunity. Real-time cores have long been equipped with FPUs, with recent cores like Cortex-M85 featuring even SIMD extensions, highlighting the growing importance of floating point capabilities. To optimize stack pushing latency, we can reserve space for FPU registers and only trigger actual saves and restores when FPU instructions are actually used, as such uses are infrequent in interrupt handlers [102]. Since FPU handling falls outside the scope of our paper's focus, we do not implement the FPU, let alone this optimization.

3.8 System-level Abstraction and Integration

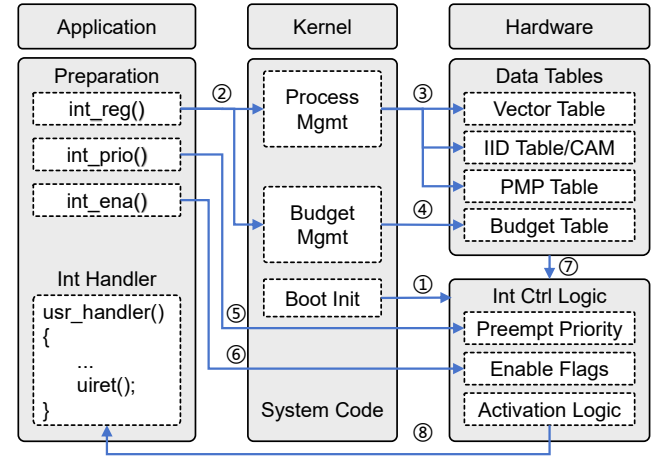


Figure 2. User-level interrupt API and system workflow.

To enable ecosystem integration, we present the system-level Application Programming Interface (API) shown in Figure 2 that maps interrupts to run-to-completion hardware threads.

At boot time, the kernel initializes the extension ① to enable user-level interrupts for applications. To register a user-level interrupt handler, the application first invokes the `int_reg(int_id, handler, policy)` ②, which passes the interrupt number, handler address, and desired scheduling policy to the kernel's process and budget management modules after due permission checks. The process management module allocates an entry in the IID table, registers the handler in the interrupt vector table, and allocates a PMP entry to enforce the access permissions of the process ③; if a PMP entry is already allocated to the process, the existing entry is reused. The budget management module programs the execution budget into the corresponding budget table entry ④, where the allocation, reuse, and replenishment of the budget entry are governed by the scheduling policy. The allocation of IID, PMP, and budget table entries may be managed through a pool-based scheme analogous to ASID management in seL4 [16], or administered by a trusted user-level process, enabling flexible adaptation to diverse system architectures. The application then sets the priority via `int_prio(prio)` ⑤ and enables the interrupt via `int_ena()` ⑥. When the interrupt is triggered, the control logic loads the configuration from the tables ⑦ and redirects the execution to the handler ⑧ that contains a custom `uiret` instruction at the end.

Budget replenishment. Following the standard definition of deferrable servers, the kernel may elect to replenish the budget at the beginning of each server period. For benign periodic interrupts that never deplete the server budget, the classic recurrence [5, 38, 47, 94] may be used to validate schedulability.

Application synchronization. For most SoCs, individual interrupt masking is supported at the peripheral level, and applications may temporarily disable user-level interrupts for buffer synchronization if the corresponding peripheral registers are within the memory map of the corresponding process. However, applications are prohibited from manipulating interrupt priorities or global interrupt enables, and cannot enable any interrupt disabled at the interrupt controller unless explicitly granted the corresponding privileges. When individual interrupt masking is unavailable or undesired, synchronization can also be realized via low-overhead lockless atomic operations or ping-pong buffers with a modest memory footprint.

4 Implementation and Optimization

4.1 Implementation Primer

We implement our RISC-V core on the Nangate 45 nm library using OpenROAD [1], targeting a typical frequency of 200 MHz with 70% core utilization, and on an XC7K480T FPGA to generate runnable instances. The power metrics are measured using OpenROAD’s gate-level flow (TT corner, 25 °C, 1.1 V), simulating a convolutional neural network computation where the extension is not actively involved. To capture dynamic power characteristics when our extension is constantly engaged, we also test back-to-back interrupt workloads where the next interrupt is triggered immediately following the return of the previous one. Finally, we use silicon area, core power, and interrupt latency to guide our search, eliminating the *bottleneck* at each stage iteratively.

Do note that the interrupt latencies discussed in this section correspond to isolated events; for concurrent arrivals, a higher-priority interrupt arriving immediately after a lower-priority interrupt doubles the baseline hardware latency, which applies to conventional interrupt handling as well. Also note that the interrupt latencies reported in this section are *theoretical values* of each implementation; for measured values, refer to §5.

4.2 Baseline RISC-V Real-time Core

To evaluate the extension, we first implement the baseline RISC-V processor supporting the RV32IM instruction set, and it achieved 2.77 CoreMark/MHz when realized with the FPGA, which meets our expectations in §3. The core also integrates a single-precision FPU [29] that is powered but not clocked or supported, which is *for area and static power accounting*, as it has nothing to do with the integer pipeline.

Table 6. Characteristics of different processor configurations. CoreMark is per MHz, area is in mm², latency is in CPU cycles, and power is in static (mW) / dynamic (μW/MHz) format. “Naked” is the integer logic, “+PMP” adds an 8-entry PMP, “+FPU*” adds an idle single-precision FPU, “+SBP” adds static branch prediction, “+MDU” adds multiplier and divider, “Base” includes all of the above.

Processor	CoreMark	Floorplan Area	Latency	Power	
Naked (RV32I)	0.86	0.0253 (35%)	5	0.528	19.2
Naked+PMP		0.0340 (48%)		0.720	20.0
Naked+FPU*		0.0512 (72%)		1.030	19.15
Naked+SBP	0.92	0.0267 (37%)	5	0.552	21.35
Naked+MDU	2.69	0.0364 (51%)		0.753	19.4
Base	2.77	0.0715 (100%)	5	1.5	21.75
Cortex-M3 (DS)	3.45	0.0875	12	N/A	
IBEX (RV32IMC)	2.36	0.0475	6	N/A	

As shown in Table 6, its interrupt latency is less than that of Cortex-M3 [4] from the ARM DesignStart, and is on par with IBEX [55], another RISC-V core. The core places and routes up to 240 MHz when more frequency pressure is applied, which achieves our frequency goal.

Discussion. Our core rivals existing cores in both interrupt latency and floorplan area. CoreMark shows multipliers improve performance significantly at modest area expenses, while branch prediction gives a little boost for a little area. Being a minimal core, it is particularly *sensitive to area and power overheads*; we therefore expect the relative overhead to be *even lower for larger, feature-rich processors*.

4.3 Control and Status Register Scheme

We define several CSRs for our deterministic user-level interrupt extension; hence their “dui” prefix. The `duictl` acts as the master control: bits [31:2] hold the word-aligned base address of the IID when implemented as a table, bit 1 indicates hardware support for the extension, and bit 0 globally enables or disables the extension. When disabled, the processor operates as a standard RISC-V core, preserving full backward compatibility. The `duistk` stores the base address of a stack onto which the hardware automatically saves registers upon a user-level interrupt. The `duiepc`, similar to standard RISC-V `mepc`, preserves the program counter of the interrupted instruction. Upon user-level interrupt nesting, `duiepc` is also pushed to the stack with the general-purpose registers for nested resumption.

When IID is implemented as a CAM, each CAM entry contains three associated registers called `iidnumX`, `iidpmpX`, and `iidtimX`, where X denotes the index. Among them, the `iidnumX` holds the interrupt number for parallel matching against incoming interrupt requests. Upon a match, the `iidpmpX` provides the pointer to the PMP configuration entry in the PMP table, while the `iidtimX` provides the pointer to the execution budget entry in the budget table.

4.4 V1: Minimal Complexity

We begin our exploration with the most straightforward variant, aiming at minimizing complexity. This variant avoids

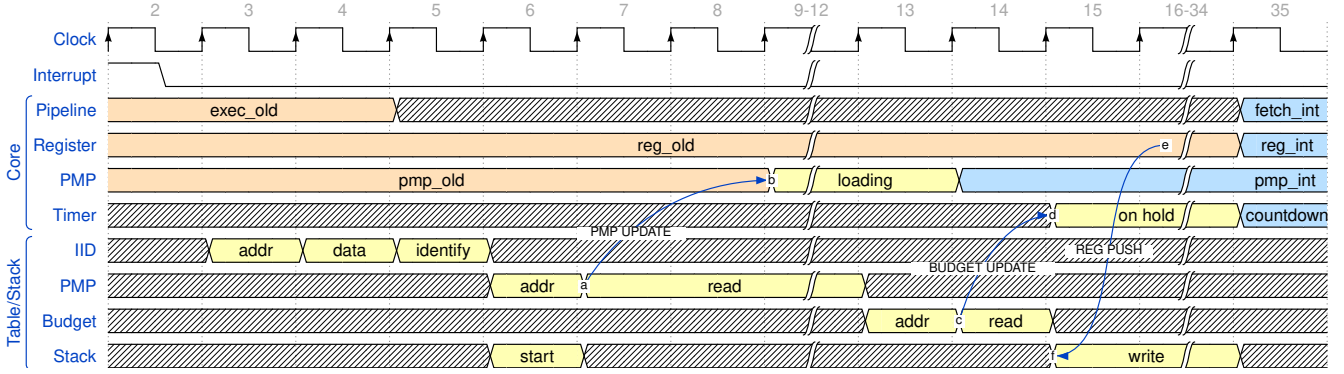


Figure 3. Timing diagram of the V1 variant upon user-level interrupt activation, with a latency of 37 cycles (2 more cycles are needed for the first fetched interrupt vector instruction to reach the execute stage). a→b shows PMP table consulting and PMP updating, c→d shows budget table consulting and timer updating, while e→f shows context stacking. The kernel-managed PMP is shadow banked and not shown.

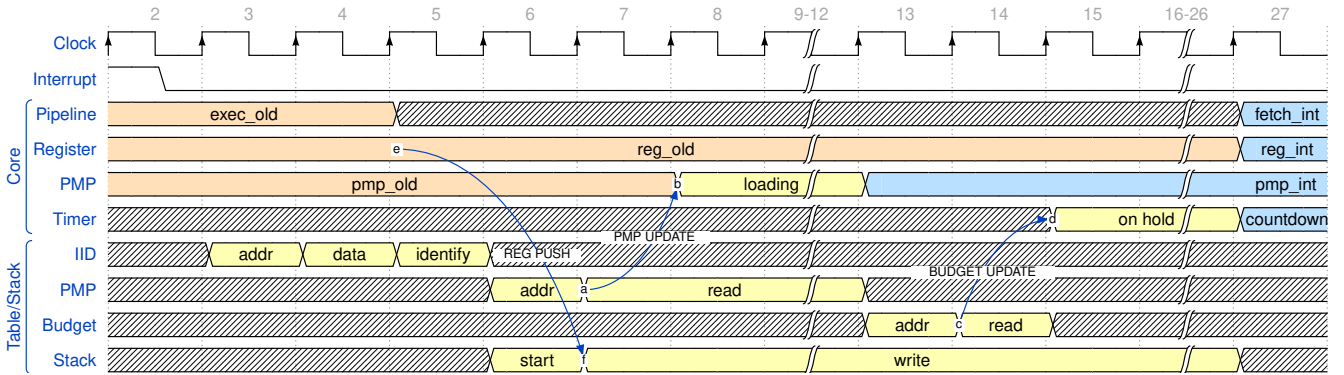


Figure 4. Timing diagram of the V2 variant upon user-level interrupt activation, with a latency of 29 cycles. The commentary is the same as that of Figure 3. The deciding factor here is still context stacking, which is 21 cycles; this includes the registers plus some internal statuses.

adding memory interfaces or register banks to the base core, instead performing all table lookups and context dumps exclusively via the main SRAM. Accordingly, we organize the IID as a linear table, and co-locate the IID table, PMP table, execution budget table, and register context stack in the main SRAM. On preemption, the kernel-managed PMP configuration is also spilled to the main SRAM.

This yields an area of 0.0870 mm^2 (+22%) and an interrupt latency of 43 cycles. As shown in Figure 3, when kernel-managed PMP context is instead stored in a dedicated shadow bank, the latency drops to 37 cycles, with an area of 0.0878 mm^2 (+23%), static power of 1.89 mW, and dynamic power of $22.4 \mu\text{W}/\text{MHz}$. When the extension is actively engaged by back-to-back interrupts, the dynamic power is $25 \mu\text{W}/\text{MHz}$, which almost doubles the $12.8 \mu\text{W}/\text{MHz}$ of the base core due to the activation of our extension, despite the pipeline stall.

Discussion. While 37 cycles represents quite an improvement over kernel paths that cost hundreds of cycles, this remains higher than kernel-level interrupts. Nevertheless, V1 serves as a choice for designs that require strict drop-in compatibility with existing bus interfaces. We draw two key

observations: (1) register context stacking is the longest memory operation unless an excessive number of PMP entries (more than 32, whose configuration is more than 40 words) is implemented, and (2) serializing all memory accesses onto a single bus leads to severe contention. Additionally, since shadow banking kernel-managed PMP context increases the area by around 0.0008 mm^2 (+1%) for a 6-cycle gain, we always adopt this optimization and omit further discussion.

4.5 V2: Hiding Protection Table Latency

To mitigate bus contention, we introduce a TCM port into V1 and dedicate it to the register context stacking, which is the longest operation, thereby *hiding PMP and budget table access latencies*. While this may compromise bus interface compatibility in some cases, many processors—including the Cortex-M and Cortex-R—come with native TCM support.

This implementation reduces the latency to 29 cycles as shown in Figure 4, with an area of 0.0905 mm^2 (+27%), static power of 1.9 mW, and dynamic power of $23.9 \mu\text{W}/\text{MHz}$. When the extension is actively engaged by back-to-back interrupts, the dynamic power is $25.9 \mu\text{W}/\text{MHz}$.

Discussion. Comparing this implementation with V1 (§4.4), we draw two key conclusions: (1) adding dedicated TCM ports introduces interface complexity but incurs little area and power overhead, and (2) *register context stacking remains dominant even with a dedicated memory port*, which renders further port dedication to the PMP table and budget table ineffective at this stage. Notably, PMP loading becomes dominant when more than 32 entries are implemented, and the TCM port should be dedicated to the PMP table instead. We omit further discussion of this rare case, as *no meaningful optimizations remain beyond IID lookup acceleration* (§4.8).

4.6 V3: Removing Context Stacking Bottleneck

To fully *eliminate the register context stacking bottleneck*, we employ additional register banks to enable zero-latency context switching, and keep the dedicated stack TCM port in V2 (§4.5) to support unbounded nesting once the banks are exhausted. The timing is similar to V2's Figure 4, except that the instruction fetching now immediately follows budget updating.

Adding one extra register bank reduces interrupt latency to 17 cycles, with a core area of 0.1057 mm^2 (+48%), static power of 2.23 mW, and dynamic power of $23.95 \mu\text{W}/\text{MHz}$. When the extension is actively engaged by back-to-back interrupts, the dynamic power is $28.65 \mu\text{W}/\text{MHz}$. Further increasing the number of extra banks to 2 raises the core area to 0.1212 mm^2 (+48%→+70%), a steep escalation.

Discussion. This implementation confirms that register banking effectively *removes the context stacking bottleneck* but incurs significant area and power overhead when compared with V2 (§4.5). Moreover, employing multiple banks incurs a prohibitive area overhead for cost- or power-sensitive systems. Nevertheless, it matches the interrupt latency of Cortex-R4, a popular automotive and medical processor.

4.7 V4: Resolving Protection Table Contention

With the stacking bottleneck fully resolved, the critical path now shifts to contention between the PMP table and the execution budget table, which are both in SRAM. We therefore choose to add one more TCM port to V3 and move the PMP table to its dedicated TCM. The timing still resembles V2's Figure 4, except that the instruction fetching now immediately follows PMP updating, and that the PMP updating happens in parallel to budget loading.

This refinement reduces interrupt latency to 14 cycles, with a core area of 0.1077 mm^2 (+51%), static power of 2.23 mW, and dynamic power of $23.95 \mu\text{W}/\text{MHz}$. When the extension is actively engaged by back-to-back interrupts, the dynamic power is $29.30 \mu\text{W}/\text{MHz}$.

Discussion. From a practical engineering standpoint, this result is already compelling: it roughly matches the kernel-level interrupt latency of the Cortex-M0, which is prevalent in Unmanned Aerial Vehicles (UAVs). The only bottleneck

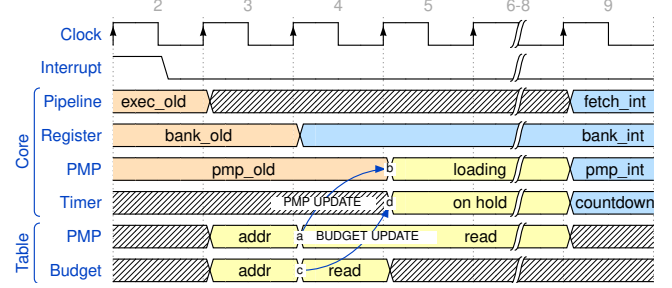


Figure 5. Timing diagram of the V5 variant upon user-level interrupt activation, with a latency of 11 cycles. The commentary is the same as that of Figure 3, except that the IID consulting costs no extra cycles, while the context stacking is eliminated by banking.

that remains is the IID, whose lookup *must be performed* before all other activities can be launched in parallel.

4.8 V5: Enabling Zero-latency IID Lookup

Based on V4 (§4.7), we *implement the IID as CAM* to embed the lookup operation directly into the interrupt acknowledge cycle. Upon a lookup, the first matching entry outputs the corresponding PMP table entry address and budget table entry address. The IID CAM must contain sufficient entries to hold all user-level interrupts for a given target scenario, and we configure it to hold up to 16 interrupts. This targets typical deployment scenarios with six to eight processes, where each process may demand one or two accelerated user-level interrupts. Since the number of user-level interrupts is limited by IID CAM capacity in this variant, the pool-based management described in §3.8 may be leveraged to govern entry allocation across applications. As this management mechanism is not on the interrupt delivery path, it has no impact on interrupt latency.

As shown in Figure 5, this final optimization design reduces interrupt latency to 11 cycles, with a core area of 0.1186 mm^2 (+66%), static power of 2.53 mW, and dynamic power of $23.60 \mu\text{W}/\text{MHz}$. When the extension is actively engaged by back-to-back interrupts, the dynamic power is $26.60 \mu\text{W}/\text{MHz}$, reversing the increasing trend across V1–V4 as IID lookup does not access memory in this variant.

Halving entries to 8 reduces the core area to 0.1123 mm^2 (+66%→+57%) with a dynamic power of $25.9 \mu\text{W}/\text{MHz}$ under back-to-back interrupts, whereas doubling entries to 32 raises the core area to 0.1295 mm^2 (+66%→+81%) with a dynamic power of $28.95 \mu\text{W}/\text{MHz}$ under back-to-back interrupts. Nevertheless, dynamic power for computation workloads is identical to the 16-entry case, as our extension remains inactive.

Discussion. This optimization delivers near-minimum latency at the expense of substantially increased area and is already faster than the Cortex-M3, which is widely deployed

in Industrial Control Systems (ICS). While the IID CAM optimization is logically independent, it only yields a 3-cycle reduction with an additional 15% area overhead, making it *attractive only with other optimizations*. In comparison, register banking—the most costly optimization—delivers a 12-cycle reduction at a cost of 21% area, which provides a marginal trade-off that is more than 2× as favorable. Notably, this variant exhibits lower dynamic power than V4 and V3, which stems from the decoupling of the IIDU from the main data path.

4.9 Implementation Summary

Table 7. Characteristics of baseline processor cores with different extension variants. Floorplan area is in mm², interrupt latency is in CPU cycles, and power is in static (mW) / dynamic-benchmark (μW/MHz) / dynamic-back-to-back-interrupt (μW/MHz) format. “Kernel” refers to the microkernel software path in [68]. Since the base core reaches 240 MHz, we apply a uniform 200 MHz timing pressure here to ensure a fair area comparison.

Processor	Interrupt latency	Floorplan area	Power		
Base	5 (kernel-level)	0.0715 (100%)	1.50	21.75	12.80
Base+V1	37	0.0878 (123%)	1.89	22.40	25.00
Base+V2	29 (-8)	0.0905 (127%)	1.90	23.90	25.90
Base+V3	17 (-20)	0.1057 (148%)	2.23	23.95	28.65
Base+V4	14 (-23)	0.1077 (151%)	2.23	23.95	29.30
Base+V5	11 (-26)	0.1186 (166%)	2.53	23.60	26.60
Kernel	634	N/A	N/A		

Table 7 summarizes the characteristics of all variants. Each resolves one critical performance bottleneck at a time, with successively diminishing returns. Area increases from Base through V2 are dominated by additional CSRs and combinational logic, whereas the steep growth from V3 through V5 arises from banked contexts and the IID CAM. Notably, the TCM in V2 through V5 does not increase the total memory footprint, as the aggregate memory can be split between main SRAM and TCM; their physical addresses can be made contiguous to avoid impacting application developers. Nevertheless, the overheads of the TCM bus logic and interface are included in our measurements.

Key variants. Considering the diverse requirements of real-world applications, we highlight three key implementation variants for in-depth evaluation:

V1: IID as table, context on stack, no extra TCM ports.

This variant minimizes area and requires no modifications to the memory interface, but incurs the longest latency. It is ideal for cores demanding strict drop-in compatibility with existing designs.

V2: IID as table, context on stack, dedicated TCM for stack. This variant reduces interrupt latency by adding a single dedicated TCM port. Although it complicates the memory interface, it introduces negligible area and power overhead. It is well-suited for cores with native TCM support or moderate area constraints.

V5: IID as CAM, context banked, dedicated TCM for PMP table, dedicated TCM for potential spills. This variant achieves the lowest interrupt latency, but incurs the highest area and power overhead, plus additional TCM ports. Unlike V1 and V2, V5 is bounded by the IID entries and register banks; the former bounds the number of interrupt sources while the latter bounds nesting depth before falling back to V2. It is exclusively for applications requiring minimum latency, or for high-end cores with rich features (e.g. double-precision FPU or cache) whose area and power dwarf this extension.

5 Evaluation

In our evaluation, we scrutinize the three key implementations listed in §4.9, and aim to answer the following questions that directly correspond to our goals listed in §3:

- Q1: Does the extension exhibit consistently low interrupt latencies regardless of the current process, when compared with the state of the art (G1)?**
- Q2: Does the extension terminate the execution of the interrupt handler forcibly when it violates spatial (G2) or temporal (G3) isolation?**
- Q3: Is the area cost acceptable, considering that the core is a portion of the entire SoC die (G4)?**
- Q4: What changes to an existing microkernel are necessary to integrate this extension (G5)?**

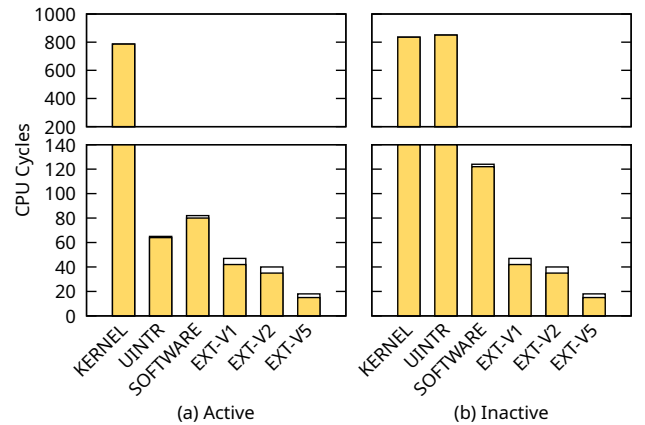


Figure 6. Raw interrupt latency, in CPU cycles. Lower is better.

5.1 Evaluation Setup

We implement the aforementioned RISC-V processors with three key variants on an XC7K480T FPGA platform. The cacheless SoC we implement runs at 50 MHz, which is typical, and integrates 128 KiB of zero-wait-state SRAM and 256 KiB of flash, with the latter emulated using SRAM. Note that for systems with external memory, our zero-wait-state

SRAM approximates ideal L1-cache hits; for longer software-mediated paths, cache misses may introduce additional latency, whereas our extension is minimally impacted as it may interface with the TCM. Therefore, the presented performance gains over kernel-level interrupts are conservative. SoCs equipped with the hardware extension but using slower or external RAM will show even greater gains than those reported here.

All test binaries are compiled using GCC 15.2.0 with the -O3 optimization level. Each measurement is repeated 10,000 times to ensure statistically robust results.

In all bar figures, average and maximum values are presented as stacked bars: *the darker foreground bar denotes the average case, while the lighter upper bar represents the observed worst case*. In all line plots, data points are measured at 10%, 20%, 50%, and 100% of each scheme's maximum application performance; intermediate values are obtained via interpolation.

For evaluation, we use the microkernel in [50], which is on par with TockOS [48], and integrate our extension into it. We compare the key implementations against three baselines: (1) **KERNEL**, the conventional mechanism relying entirely on kernel mediation, (2) **UINTR**, the Intel UINTR or RISC-V-N scheme that bypasses the kernel only for active target processes; we emulate its performance by handling interrupts directly in kernel mode if the target process is active, while falling back to **KERNEL** otherwise [7, 35, 73], and (3) **SOFTWARE**, a software scheme without hardware mode transitions: it uses minimal prologues to save context and set up PMP—still, leveraging the PMP is more efficient than software bounds checking on-the-fly [69]—before dispatching to the handler directly in kernel mode, making it a near-optimal pure-software latency baseline [12, 56].

5.2 Raw Interrupt Latency (Q1; G1)

To start, we measure the latency from the assertion of the interrupt signal pin to the execution of the first instruction in the user-level handler. To produce the interrupts, an auto-reloading decremting timer is configured to fire every 4,000 cycles, far longer than the latency of all methods. The latency is computed as the difference between the fixed overflow period and the remaining timer count captured upon handler entry, minus the timer read overhead, which is 4 cycles.

As shown in Figure 6, we measure the latency under two scenarios: (a) when the target process is actively running, and (b) when it is inactive and has been switched out. Note that **UINTR** in (a) reflects the performance of trusted kernel handlers as well, which is measured as a performance surrogate for user-level delivery. Meanwhile, the **SOFTWARE** in (a) represents software methods without extra overhead [56], given that no PMP configuration is involved upon entry. To further validate the optimality of **SOFTWARE** in (b) when PMP

configuration is involved, we run an array iteration benchmark with the Ahead-of-Time (AoT) eBPF bpftime [106] on our core, and it is 3.27× slower than native code. In other words, for a small computation that takes 20 cycles in native code, the equivalent eBPF code consumes over 60 cycles. Note that PMP configuration introduces an overhead of less than 40 cycles; even after adding this cost, the **SOFTWARE** in (b) totals below 60 cycles, which the eBPF code readily exceeds.

Discussion. All measured latencies are highly deterministic when it is known whether the target process is either active or inactive, with maximum values closely aligning with average values across all schemes. **KERNEL** exhibits poor performance, with latency exceeding 800 cycles even in the best case. Since its critical path is fully dominated by kernel mediation, latency varies little between active and inactive conditions, but its extremely high absolute latency renders it unsuitable for low-latency real-time systems. **UINTR** shows extreme performance variability: while it achieves moderate latency for active processes by bypassing the kernel, it incurs massive overhead in the inactive state as it falls back to the full **KERNEL** path. Also, considering that **UINTR** in the active case is a kernel-level surrogate, we note that this is also a lower bound for software handlers that disregard security. **SOFTWARE**, on the other hand, hits a fundamental floor, with around 80 cycles of latency when active and 120 cycles of latency when inactive. In contrast, our extensions maintain latencies below 50 cycles, with near-zero variance between active and inactive scenarios, a feat unattainable by other schemes. Notably, V5 achieves a latency of 15 cycles—which translates to 300 ns—outperforming **KERNEL** and **UINTR** by more than 50× and **SOFTWARE** by more than 8×. Even V1 and V2 outperform **SOFTWARE** by more than 3× when the target is inactive.

5.3 Isolation Effectiveness (Q2; G2, G3)

To evaluate the safety and security of our extensions, we set up malicious interrupt handlers to perform unauthorized memory accesses (confidentiality and integrity) or execute infinite loops (availability). We test three critical preemption scenarios: (1) the malicious handler preempts a user-level thread, (2) the malicious handler preempts the kernel, and (3) the malicious handler preempts another user-level interrupt handler; for the last case, we also test deeply nested interrupt handling with exhausted banks in the V5 variant.

Discussion. Our results demonstrate that all variants enforce strict temporal and spatial protection domains across all preemption scenarios. Any malicious handler attempting to violate isolation guarantees is immediately and forcibly terminated, with control returned to the preempted context, safeguarding system confidentiality, integrity, and availability. Also, when the banks in V5 are exhausted, its interrupt latency degrades gracefully to that of V2 as expected. This

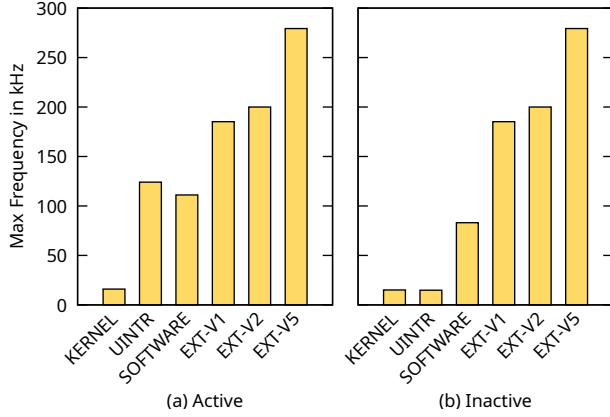


Figure 7. Maximum achievable PTO frequency. Higher is better.

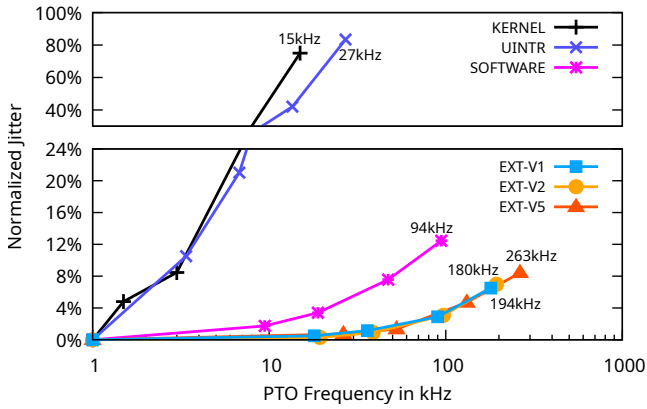


Figure 8. Normalized waveform jitter as a function of PTO frequency, under rapid process interleave interference. Lower is better.

demonstrates that temporal isolation remains guaranteed even under such conditions.

5.4 Latency-sensitive Application (Q1; G1)

To demonstrate the effectiveness of our extensions in latency- and jitter-sensitive scenarios, we apply them to the Pulse Train Output (PTO) application [82], which is a core function in Programmable Logic Controllers (PLCs), robotic systems, and UAVs. It requires the system to generate precisely timed pulse sequences to drive actuators and execute motion commands. Typically, a background task prepares the pulse train specification, including an initial pulse period and a per-pulse period delta. A foreground interrupt handler then computes the required pulse intervals on-the-fly and configures the timer for the next timeout accordingly, so that the output pin toggles timely upon interrupt firing.

We use a square-wave pulse train to measure two critical metrics: (1) the maximum sustainable PTO frequency, correlating with speed or resolution, and (2) pulse waveform jitter normalized to ideal output across frequencies, correlating

with accuracy. Accordingly, (1) Figure 7 reports the maximum sustainable PTO frequencies, measured under both (a) active and (b) inactive target process states, and (2) Figure 8 shows jitter as a function of PTO frequency under mixed process conditions, with the target and an unrelated process alternating continuously to reflect the complex background workloads of real-world robotic systems.

Frequency Discussion. The KERNEL baseline yields the lowest PTO frequencies, as its heavy kernel mediation makes it unsuitable for fast actuation, which requires at least 20 kHz [82]. The UINTR performs adequately when the target process is active, but degrades to KERNEL otherwise, while the SOFTWARE provides a middle ground. In contrast, our extensions are nearly unaffected by process state, consistently outperforming all baselines and achieving peak PTO frequencies up to nearly 300 kHz.

Jitter Discussion. Under the mixed process scenario, KERNEL and UINTR both exhibit catastrophic degradation, exceeding 40% jitter around 15 kHz, rendering them unusable for any practical control task. SOFTWARE, on the other hand, has better jitter, but remains unacceptable for applications that require less than 10% jitter at 100 kHz [82]. Instead, our extensions maintain less than 5% jitter at 100 kHz, with the V5 achieving 8% jitter at over 250 kHz, satisfying the most demanding applications. At 10 kHz, which is a common frequency, all our extensions exhibit less than 0.5% jitter, which is 60× better than the kernel-mediated approaches.

5.5 Task-colocation Application (Q1; G1)

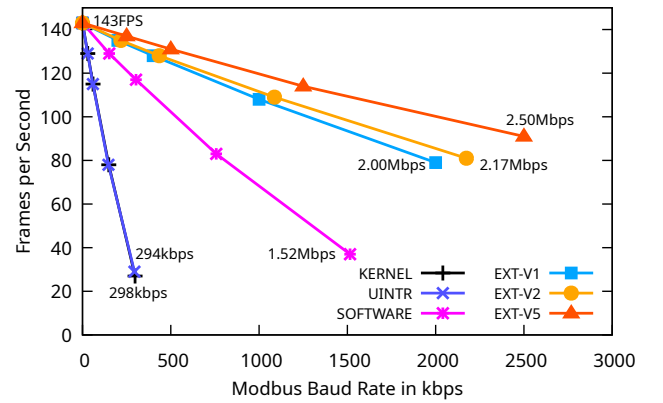


Figure 9. Achievable FPS of a background image recognition task as a function of foreground Modbus-RTU baud rate. Higher is better.

In domain controller consolidation scenarios [6], foreground real-time tasks are often colocated with background best-effort workloads to cut hardware costs, a trend also seen in cloud computing [51]. The interference is mutual: the real-time task suffers prolonged worst-case latencies preempting the continuously running best-effort workload, while the

best-effort workload suffers *average-case* throughput loss due to frequent interruptions from the real-time task.

We use Modbus-RTU—a widely adopted serial protocol in sensing and ICS [89]—as the real-time foreground communication task. It generates one interrupt per byte transmitted, resulting in a high interrupt frequency that stresses the interrupt handling pipeline. The background best-effort workload is a neural image recognition application [83] that repeatedly runs inferences on input images, a common scenario in edge vision and embodied intelligence systems. As shown in Figure 9, we measure the impact of foreground interrupt handling overhead on the background workload’s throughput (Frames per Second, FPS) as the Modbus baud rate increases.

Discussion. Under the inactive target process condition enforced in this experiment, UINTR exhibits behavior nearly identical to that of the KERNEL baseline, as it always falls back to the kernel-mediated slow path when the background task runs continuously. While both schemes readily support 115.2 kbps, a widely used industrial baud rate, their background task performance degrades severely, cutting FPS from 143 to 92, which is a 36% reduction. Furthermore, neither can sustain baud rates above 300 kbps due to their excessive interrupt latency, a critical limitation for modern UAVs and industrial systems. SOFTWARE outperforms the kernel-mediated baselines and can operate at over 1 Mbps, which is a representative high baud rate for industrial scenarios [92], but at this baud rate, FPS drops to 68, representing a 52% performance loss. Notably, all our variants support baud rates up to 2 Mbps. At 1 Mbps, our best-performing V5 achieves 120 FPS, incurring less than 17% loss; even the entry-level V1 sustains 108 FPS, which is a 24% penalty. At 115.2 kbps, all of our extensions incur at most a 5 FPS penalty, which is 10× better than kernel-mediated solutions, and 2× better than SOFTWARE.

5.6 SoC Die Area Impact (Q3; G4)

Table 8. Total die area of typical SoCs. Length and width are in mm, while area is in mm². Core% is the portion of our base core on the same assumed die area; V1%, V2%, and V5% are the extra areas that the V1, V2, and V5 would require. The reason for a small Core% is that SoCs integrate RAM, flash, and various peripherals.

Die (Domain)	L×W=Area	Core%	V1%	V2%	V5%
STM32H743 (General)	5.06×4.70=23.78	0.3%	0.07%	0.08%	0.20%
LPC55S69 (Financial)	2.90×2.78=8.06	0.9%	0.20%	0.24%	0.58%
ESP32C3 (Wi-Fi IoT)	2.28×2.27=5.18	1.4%	0.31%	0.37%	0.91%
TC387 (Automotive)	7.64×7.39=56.46	0.1%	0.03%	0.03%	0.08%

We decapped four representative SoCs implemented using 40 nm technology (more compact than our Nangate 45 nm) and obtained their die areas as shown in Table 8, to evaluate the potential area impact of our extension on the entire die.

Discussion. It is evident that the core occupies a marginal portion of the total area, particularly when noting that we

use a 45 nm process. As a result, our extension is almost negligible; even if we assume a tiny 1 mm² die, the V1 only accounts for less than 2% of extra area, and the V5 would account for 5%.

5.7 Kernel Integration Impact (Q4; G5)

Table 9. Modifications of microkernel source, in SLoC.

Kernel+HAL	V1	V2	V5
5856+4249	+216 (+2.14%)	+216 (+2.14%)	+270 (+2.67%)

To assess the adaptation effort on an existing microkernel [50], we count the Source Lines of Code (SLoC) that must be added to leverage the extension, as shown in Table 9.

Discussion. We may observe that the SLoC increases for all variants are small when compared to the entire code base, reflecting minimal adaptation effort.

6 Conclusions

This paper proposes a hardware extension that aims to achieve deterministic multi-tenant user-level interrupt delivery in real-time embedded systems. It guarantees low and bounded worst-case latency regardless of the target process state, while enforcing strict spatial and temporal confinement for untrusted user-level interrupt handlers. Through design space exploration, we identify three representative variants, the best of which reduces worst-case latency by more than 50× with modest overheads. The resulting extension provides a holistic solution for developing secure, dependable, and low-latency cyber-physical systems in real-time consolidation scenarios. It may also help to lift safety- and security-related restrictions preventing untrusted applications from employing hardware interrupt handlers in automotive and industrial contexts.

Acknowledgements

This work is supported by National Key R&D Program of China (No. 2022YFB4502001) and National Natural Science Foundation of China (No. 62402291, 62302265, 62232010, U23A20332). It is also supported by Shandong Province Natural Science Foundation (No. ZR2023QF172, 2024HWYQ-020).

The authors would like to thank the reviewers and shepherd for valuable comments and suggestions. The authors are also grateful to the Shandong Provincial Key Laboratory of Computing-Network Integration at Shandong University and its director, Xiuzhen Cheng, for their support.

Finally, the authors dedicate this work to the memory of Kexin Li, who once did hold this project in hand but was untimely taken. *That which lurks within a one-in-a-million worst case shall yet come to pass; herein lies our quest to render earthly creations securely deterministic.*

A Artifact Appendix

A.1 Abstract

This artifact provides the RTL implementations, software benchmarks, and evaluation infrastructure used to reproduce the main results presented in the paper. It includes Vivado-, OpenROAD-, and Verilator-based workflows for validating the reported area, power, and performance results for the core with our extension, as well as the performance of two real-world application studies.

The full evaluation requires Linux and Windows 10 systems with more than 100 GB of free disk space on Linux and at least 50 GB on Windows. The evaluation should reproduce the results shown in the evaluation section of our paper, albeit with possible minor deviations, and our main conclusions should be supported.

A.2 Artifact check-list (meta-information)

- **Program:** Matrix Multiply, Raw Latency Test, PTO and Modbus-RTU.
- **Compilation:** Vivado-2019.2 (Windows 10), Verilator-5.046, GCC-13.3.0, RISC-V GNU Toolchain 15.2.0, OpenROAD-flow-scripts commit b39f7e7, Python3.12, Make (CMake).
- **Transformations:** BIN-to-COE conversion using the provided Python script, and BIN-to-TXT conversion using xxd or the tool included in the artifact.
- **Model:** A TinyMaix-based MNIST model, included in the artifact.
- **Run-time environment:** Windows 10, Ubuntu 24.04, and RME/RVM/RMP microkernel/virtualization stack, which are included in the artifact. The evaluated applications run as native RME processes and do not run inside an RMP guest.
- **Hardware:** A Xilinx XC7K480T FPGA board or a comparable FPGA board with at least 384 KiB of BRAM (256 KiB + 128 KiB) and a USB-to-UART adapter. The recommended configurations are an AMD Ryzen 5 5500U processor with 16 GB of memory for the Ubuntu machine and an Intel Core i7-11700 processor with 16 GB of memory for the Windows machine.
- **Metrics:** User-level interrupt latency (CPU cycles), ASIC static power (mW) and dynamic power (μ W/MHz), and ASIC area (mm^2), PTO frequency (kHz), PTO jitter (%), Modbus baud rate (bps), MNIST framerate (FPS).
- **Output:** (1) The PTO, Modbus-RTU, and interrupt-latency experiments output their results through the UART interface, and the provided Gnuplot scripts generate PDF figures from the collected data. (2) The power and area results are generated by OpenROAD-flow-scripts.
- **Experiments:** OpenROAD-flow-scripts for ASIC implementation, Verilator simulation for SAIF generation, and an FPGA board for the application experiments.
- **How much disk space required (approximately)?:** Linux: more than 100 GB; Windows: more than 50 GB.
- **How much time is needed to prepare workflow (approximately)?:** Less than 2 hours.
- **How much time is needed to complete experiments (approximately)?:** Approximately 1 day.

- **Publicly available?:** Yes.
- **Code licenses (if publicly available)?:** GPL v2.
- **Data licenses (if publicly available)?:** GPL v2.
- **Archived (provide DOI)?:** [10.6084/m9.figshare.33511450](https://doi.org/10.6084/m9.figshare.33511450).

A.3 Description

A.3.1 How to access. We provide a 7-Zip archive named [asplos-ae-no1-submission-no325.7z](#) as the artifact to evaluators on the Figshare platform.

Note: All 7z archives referenced in the installation section are contained in [asplos-ae-no1-submission-no325.7z](#).

A.3.2 Hardware dependencies. We deploy our processor core on a Xilinx XC7K480T FPGA to run the experiments. The experimental results are transmitted through the UART interface implemented on the FPGA board. A USB-to-UART adapter is required unless another method is available for accessing the UART interface.

A.3.3 Software dependencies.

- We use Xilinx Vivado 2019.2 to synthesize the Verilog RTL, generate the bitstream, and program the FPGA board.
- We use OpenROAD-flow-scripts to run the ASIC implementation flow and evaluate the area and power of our processor core.
- The serial-port software used on the Windows host computer is LLCOM, an open-source application that supports Lua scripts.
- We use Gnuplot 5.4.6 to generate the PDF figures.

The other software dependencies are listed in the checklist above, and all companion code that we use is included in the artifact package.

A.3.4 Models. We use a pretrained MNIST model with TinyMaix for the image-recognition task in the Modbus-RTU experiment. [TinyMaix](#) is a lightweight neural-network inference library designed for running TinyML models on resource-constrained embedded systems, and we use it for the sole purpose of generating computation-intensive workloads.

A.4 Installation

This evaluation flow requires two operating systems: Windows 10 and Ubuntu 24.04. For Windows, a machine with high single-core performance is recommended for FPGA synthesis; we chose Windows to align with interactive engineers who are familiar with a GUI interface, though batch-mode synthesis under Linux is also possible. For Linux, a machine with greater parallel computing power is recommended for the ASIC design flow and Verilator simulation.

After extracting the main artifact package, verify that <ARTIFACT_ROOT> contains the following files:

```
<ARTIFACT_ROOT>/
|-- base_core.7z
```

```
|-- download.lua
|-- freq_help.py
|-- gnuplot.7z
|-- openroad.tar.zst
|-- openroad-orfs-docker.7z
|-- verilator_sim.7z
|-- system.7z
|-- xpack-riscv-none-elf-gcc-15.2.0.zip
\-- AE-README.pdf
```

• Windows environment

- Install Vivado 2019.2 and restore the `base_core` project.
- Install GNU Make, CMake, Git Bash, and the RISC-V GCC toolchain.
- Extract the RME-RVM-RMP-TinyMaix software stack.
- Build the Lat, mbrs_frame, and PTO projects.
- Install LLCOM and configure `download.lua` for UART-based BIN-file programming.

• Linux environment

- Extract the Verilator simulation environment and OpenROAD-flow-scripts.
- Install Docker and load the provided OpenROAD image.
- Install Verilator v5.046, Make, Python 3.12.

• Plotting tools

- Install Gnuplot 5.4.6.
- Extract the plotting scripts in the `gnuplot.7z`.
- Verify the scripts in the `latency`, `mbrs`, and `PTO` directories.

Please refer to `AE-README.pdf` in the artifact for detailed installation instructions. Each installation subsection includes a basic build or execution test. Successful completion of these tests confirms that the corresponding evaluation environment is correctly installed.

A.5 Experiment workflow

The evaluation consists of separate ASIC and FPGA workflows. Please refer to `AE-README.pdf` in the artifact for detailed workflow instructions.

• ASIC evaluation

- Run the OpenROAD ASIC implementation flow.
- Simulate the generated Verilog design with Verilator.
- Generate and import the SAIF files.
- Record the area, static power, and dynamic power.

• FPGA evaluation

- Generate and program the required Vivado bitstream.
- Evaluate the KERNEL, UINTR, SOFTWARE, EXT-V1, EXT-V2, and EXT-V5 cases.
- Measure active and inactive user-level interrupt latency.

- Measure MNIST FPS at different Modbus-RTU baud rates.
- Measure active and inactive PTO frequencies.
- Calculate PTO jitter at 10%, 20%, 50%, and 100% of the maximum frequency.
- Record the results in the corresponding Gnuplot `.dat` files.

A.6 Evaluation and expected results

The results of the application experiments are plotted and compared with the corresponding figures in the paper, while the ASIC area and power results are compared with the corresponding tables. Please refer to `AE-README.pdf` in the artifact for further information.

- Compare the ASIC area and power results of variants of our base core with those listed in Table 6 and Table 7. There might be subtle differences for each run (as this is the nature of ASIC tools), but the numbers should stay quite close to what we report, and the general trend or conclusion should not deviate from our paper.
- Generate and compare the following plots:
 - User-level interrupt latency: Figure 6.
 - PTO frequency: Figure 7.
 - PTO jitter: Figure 8.
 - Modbus-RTU MNIST framerate: Figure 9.

Similar to the tables above, there might be differences, but most of them should be moderate, and the general trend or conclusion should not deviate from our paper.

A.7 Methodology

Submission, reviewing, and badging methodology:

- <https://www.acm.org/publications/policies/artifact-review-badging>
- <http://cTuning.org/ae/submission-20201122.html>
- <http://cTuning.org/ae/reviewing-20201122.html>

B Supplementary Appendix

For informative purposes, we include supplementary materials below. They provide a broader view for curious readers.

B.1 Real-time Systems and Processors

Real-time systems are required to respond to external stimuli within a predefined deadline or at precise epochs. Such systems *prioritize determinism and responsiveness over resource utilization or best-effort throughput*. In short, real-time is not merely about speed, but fundamentally about *predictability*.

Accordingly, typical real-time processors adopt deterministic microarchitectural designs [33], such as in-order pipelines with limited dual-issue at most, software-managed protection mechanisms rather than hardware page table walkers, static branch prediction, and even scratchpad memories instead of caches. Even high-end real-time processors equipped with out-of-order pipelines or dynamic branch prediction—such as the Cortex-R—often support disabling some enhancement features to enable more predictable execution. This ensures real-time systems can generate responses at the required epoch precisely, thereby minimizing timing jitter.

B.2 FPU Context Handling

As this work focuses on integer pipelines, FPU context management is considered outside the scope. Nevertheless, we highlight several feasible approaches for FPU context handling, beyond the straightforward banking approach.

Exception-on-touch. To prevent untrusted user-level handlers from accessing or tampering with the interrupted FPU context, we can block FPU register access for such handlers, triggering an exception and forcing a return on any attempted access. This approach introduces little hardware complexity but forbids handlers from using the FPU for floating-point acceleration, which is rarely needed.

Lazy stacking. When some user-level interrupt handlers do require FPU access, we may employ a lazy stacking scheme analogous to that of Cortex-M [102]. Specifically, FPU registers are not stacked until they are first accessed or tampered with, at which point either all FPU registers or only the accessed ones are stacked. This allows the use of the FPU and only imposes latency penalties on handlers that benefit from the FPU.

B.3 Top-half and Bottom-Half Scheme

As the top-/bottom-half interrupt scheme is well established, it is desirable that we leverage our user-level interrupts for responsive top-half handling and kernel threads for flexible bottom-half processing. However, the preemptive nature of our user-level interrupts lacks the ability to notify the kernel with system calls, which seems to preclude bottom-half support.

This lack is *intentional*, as mapped peripherals often offer *sufficient redundant interrupt resources*, of which only a few

are practically utilized. For instance, serial ports provide transmit/receive interrupts as well as parity error, noise error, and even hardware flow control interrupts. If any of these interrupts are unused, they can be repurposed as a confined software interrupt source to notify the kernel and trigger bottom-half execution, requiring no additional hardware due to the mapping of the entire serial port to the process.

Even if all interrupt sources of the mapped serial port are occupied, unused on-chip peripherals can be repurposed by disconnecting external pin interfaces and mapping them to the user process, so that their interrupt sources may be leveraged. Full peripheral saturation is extremely rare in real-world systems, making this an economical solution. For further enhancement, future extensions can introduce dedicated registers or user-level instructions to explicitly trigger pending system calls, analogous to the PendSV mechanism in Cortex-M processors [102].

B.4 Inter-Process Communication Possibility

As many prior works focus on Inter-Process Communication (IPC) or scheduling efficiency [7, 37, 52], one may naturally ask whether our mechanism can be applied to the same domain, like [21]. We do not rule out this possibility, as this use case would resemble a thread-migration-style synchronous invocation [50]. Given that our interrupt return latency is comparable to or even lower than entry latency, a round-trip synchronous invocation could achieve under 30 cycles—far faster than kernel-mediated software-only approaches.

Nevertheless, we do not explore this use case for six reasons: (1) our user-level interrupts always preempt the kernel, elevating thread priority similarly to the Priority Ceiling Protocol (PCP) and effectively treating the entire call as a critical section, which is not always desirable for general IPC, (2) no native return value mechanism is supported, requiring shared memory and thus complicating spatial isolation, (3) invocations now consume the budget of the interrupt source rather than the initiating thread, complicating temporal isolation, (4) system calls and thread switching are disallowed within the interrupt handler as they are run-to-completion, (5) a peripheral must be mapped to a process to grant permission to initiate such IPC, and (6) one peripheral interrupt source must be consumed for each IPC channel.

Given all of the above, our mechanism is tailored for peripheral interrupt handling (IPI at most) and thus not designed for general IPC; we therefore consider this topic outside the scope of this work.

B.5 Baseline Core Microkernel Performance

To provide an impression of the microkernel system-level operation performance of our baseline core, we compare it against the SoCs in [50] and NaxRiscv [85], which is an out-of-order RISC-V core used in [77].

As shown in Figure 10, we measure (1) intra-process context switch latency (ctx/1), (2) inter-process context switch

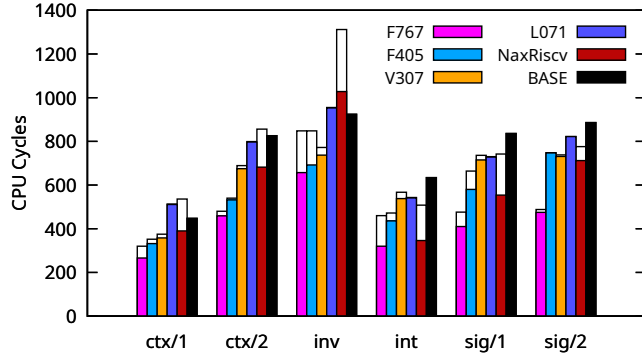


Figure 10. Comparison of microkernel basic operations. The commodity SoCs are the STM32L071, STM32F405, STM32F767, and CH32V307. Our core is on par with the STM32L071 and CH32V307, while lagging noticeably behind the dual-issue STM32F767. NaxRiscv, despite being dual-issue and out-of-order, does not yield a clear benefit due to its high mode-switch latencies, highlighting the ineffectiveness of complex microarchitectures for latency-sensitive applications. See §B.14 for detailed NaxRiscv configuration.

latency (ctx/2), (3) thread migration/return round-trip latency (inv), (4) interrupt signaling latency (int), (5) intra-process signaling latency (sig/1), and (6) inter-process signaling latency (sig/2). It can be observed that our core achieves performance roughly matching these off-the-shelf SoCs, validating its representativeness.

Table 10. Comparison of user-level interrupt latency in CPU cycles between the microkernel we chose and the TockOS.

Core	OS	TockOS [48]		Microkernel [50]	
		Average	Worst-case	Average	Worst-case
Base		1932	2121	790	792
NaxRiscv		1582	4644	565	569

We further compare the open-source microkernel [50] adopted for evaluation with TockOS [48], with results listed in Table 10; the comparison validates our kernel selection.

We also make an indirect comparison against Hopter [56], which lacks RISC-V support. We evaluate our EXT-V5 at 50 MHz on RISC-V—half the frequency of Hopter’s 100 MHz Cortex-M—while strictly following Hopter’s original pin-to-pin latency measurement methodology. Even at this lower clock frequency, our design achieves a latency of 0.51 μ s, significantly outperforming Hopter’s 1.36 μ s, while providing stronger isolation guarantees.

B.6 CoreMark Measurement Representativeness

CoreMark is a prevalent benchmark for embedded processor evaluation, analogous to the SPECint suite for general-purpose computing. Similar to SPECint results, CoreMark scores are heavily affected by the processor’s memory hierarchy configuration. Even under the ideal zero-wait-state data SRAM assumption, two distinct CoreMark evaluation

configurations exist: (1) peak performance, achieved with zero-wait-state instruction TCM; and (2) real-world performance, when instructions are fetched from an embedded Flash memory that is often augmented with a hardware prefetcher. Processor vendor datasheets typically advertise the peak performance, while the Flash-based configuration better reflects realistic deployment scenarios.

Table 11. CoreMark results under zero-wait-state instruction fetch. Data accesses are already zero-wait-state by default.

Core	Naked	+SBP	+MDU	Base
CoreMark/MHz	1.13	1.251	3.085	3.209

To provide SoC engineers with a realistic look of our extension under practical deployment conditions, our reported 2.77 CoreMark/MHz for the baseline core is measured on a 2-cycle wait-state, SRAM-emulated Flash with an eight-instruction prefetcher, which is suboptimal in terms of performance. For ideal zero-wait-state instruction TCM execution, the corresponding performance is presented in Table 11, which closely aligns with the official ARM advertisement for the Cortex-M4 core, further validating the representativeness of our core.

B.7 Correctness of the SOFTWARE Baseline

Some may notice that the standard RISC-V PMP specification supports kernel-mode access checking via the “L” bit, yet setting this bit permanently locks the corresponding PMP entry, prohibiting subsequent modification. To address this limitation, We repurpose the original “L” bit as a new “K” bit, which solely indicates whether access permissions are restricted for kernel-mode execution as well, without permanent locking.

Another potential concern involves whether PMP configuration overhead in SOFTWARE can exceed pure software checking overheads. To answer this question, we benchmark the state-of-the-art eBPF runtime [106] against native C code on our baseline processor, using a microbenchmark that calculates the average of a 1000-element array. To maximize eBPF performance, we assume all eBPF code undergoes Ahead-of-Time (AOT) compilation. Still, eBPF execution is still 3.27 $\times$ slower than native C code mainly due to its virtual machine model. Given that our PMP configuration takes approximately 40 cycles, eBPF can be expected to have higher overhead than our PMP-based surrogate for any interrupt handler exceeding 20 cycles. Beyond eBPF, WASM-based sandboxing also suffers from inherent bounds-checking drawbacks. As demonstrated in [69], WASM imposes per-linear-access bounds checks, rendering it slower than hardware-assisted PMP checking.

Prior work [54] first realizes sub- μ s user-level interrupt delivery with protection domain switching within a Linux process, representing a notable milestone for cloud computing scenarios. Given that most real-time SoCs lack virtual

address entirely, they are analogous to a single Linux process; in that sense, the concept of interrupt dispatching with call-gate-style memory isolation within a single address space has early precedents [12] in embedded systems. To construct a call gate, [54] relies on the WRPKRU of the Intel Memory Protection Key (MPK), whereas [12] employs fast SVC-based invocation on Cortex-M platforms. Specifically, [12] uses SVC *exclusively for privilege escalation* to enable inline privileged execution, forming a call gate similar to WRPKRU. Both approaches *suffer from the same prerequisite that precludes unverified binaries*: application code must be scanned to prevent undesired privilege escalations, namely the WRPKRU in [54] and SVC in [12]. Finally, this prerequisite is lifted by Donky [79], which leverages user-level interrupts to build call gates that enable system-call-like privilege transitions within a single process. In the eyes of the embedded realm, however, this can be interpreted as establishing a SASOS microkernel abstraction within a process.

Similar to SOFTWARE, all of them [12, 54, 79] still rely on software-triggered primitives for switching of unprivileged protection domains. For a fair comparison with cloud-based platforms, we expect our frequency to scale to 2 GHz with the latest semiconductor process: now the latency of EXT-V5 would drop below 10 ns in theory, which is a significant improvement over [54].

B.8 FPGA Resource Consumption and Fmax

Table 12. FPGA resources of baseline processor cores with different extension variants. Maximum frequency Fmax is in MHz.

Configuration	Slice LUTs	Slice FFs	Fmax	ASIC Fmax
Base	4700 (100%)	2377 (100%)	74	198
Base+V1	7342 (156%)	3704 (156%)	72	198
Base+V2	7242 (154%)	3706 (156%)	71	198
Base+V3	8785 (187%)	4984 (210%)	74	199
Base+V4	8857 (188%)	4985 (210%)	74	199
Base+V5	9500 (202%)	6040 (254%)	74	199

We report the resource utilization of our CPU core on the FPGA in Table 12. This excludes the FPU in order to better expose the variations in integer pipeline overheads. *As FPGA resources are coarse-grained, this data is for reference only.* Both the FPGA and ASIC Fmax values are reported as well, which confirms that our extension has little impact on Fmax.

B.9 Theoretical and Measured Latency Differences

Table 13. Difference between measured (Figure 6) and theoretical (Figure 3, Figure 4, Figure 5, Table 7) interrupt latency for our extensions, in CPU cycles.

Variant	Theoretical	Measured	Difference
EXT-V1	37	42	+5
EXT-V2	29	35	+6
EXT-V5	11	15	+4

The difference between theoretical and measured interrupt latency is shown in Table 13. Besides rounding errors, this discrepancy arises mainly because our theoretical latency only counts towards the execution (EX stage) of the first instruction in the interrupt vector. Per the standard RISC-V specification, this *must be* a jump to the actual handler, which introduces an additional delay of 3 cycles on our core.

In contrast, the Cortex-M vector table directly stores the handler address and avoids this overhead. We do note that some manufacturers choose to optimize this; however, we choose to *preserve compatibility with the standard RISC-V specification*.

B.10 Exact Data Points in Application Evaluation

Table 14. Exact data points for Figure 8. Frequency is in kHz, jitter is in percentage, normalized to the ideal output pulse width.

Exp	10%		20%		50%		100% (MAX)	
	Freq	Jitter	Freq	Jitter	Freq	Jitter	Freq	Jitter
KERNEL	1.5	4.8	3.0	8.46	7.48	25.56	14.95	75
UINTR	3.4	10.5	6.8	21.0	13.5	42.1	27.1	83.4
SOFTWARE	9.4	1.7	18.9	3.4	47.2	7.6	94.4	12.5
EXT-V1	18.0	0.5	36.0	1.2	90.0	2.9	180.0	6.5
EXT-V2	19.4	0.3	38.8	0.9	96.9	3.1	193.8	7.0
EXT-V5	26.3	0.7	52.6	1.3	131.6	4.69	263.2	8.4

Table 15. Exact data points for Figure 9. Baud rate is in kbps.

Exp	10%		20%		50%		100% (MAX)	
	Baud	FPS	Baud	FPS	Baud	FPS	Baud	FPS
KERNEL	29.8	129	59.5	115	149	78	298	27
UINTR	29.4	129	58.8	115	147	78	294	29
SOFTWARE	152	129	303	117	758	83	1515	37
EXT-V1	200	135	400	128	1000	108	2000	79
EXT-V2	217	135	435	128	1087	109	2174	81
EXT-V5	250	137	500	131	1250	114	2500	91

The exact data points used to draw Figure 8 and Figure 9 in §5 are shown in Table 14 and Table 15, for reader reference. The data points are measured at 10%, 20%, 50%, and 100% of each scheme’s maximum application performance (frequency for PTO and baud rate for Modbus); hence the data points do not align with each other.

B.11 Latency Requirements of Sample Applications

Table 16. Latency requirements of sample applications, extracted from [67].

Application	Latency	Explanation
USB I/O emulation	0.5 μ s	Emulation must be cycle-accurate
1-Wire emulation	0.5 μ s	1-Wire bus requires accurate time slots
Rotary encoders	1 μ s	Servo encoders toggle rapidly
Fast ADC data logging	1 μ s	Low jitter produces high-quality data
Machine vision LEDs	10 μ s	High-speed blinkers yield good images
IR codeword decoding	10 μ s	Codewords need to be precisely captured
Engine control units	10 μ s	Ignition timing must be accurate
Quadcopter control	100 μ s	Typical control loop is below 1 ms

To familiarize readers with additional use cases beyond those evaluated in §5, we extract representative applications from [67] and reproduce them in Table 16. For a typical 200 MHz processor, each cycle corresponds to 5 ns. Interrupt latencies exceeding 200 cycles cannot support applications demanding 1 μ s response times, and will likely fail to satisfy jitter constraints for applications requiring 10 μ s-class latency if interrupt latency is nondeterministic.

B.12 SoC Dieshots

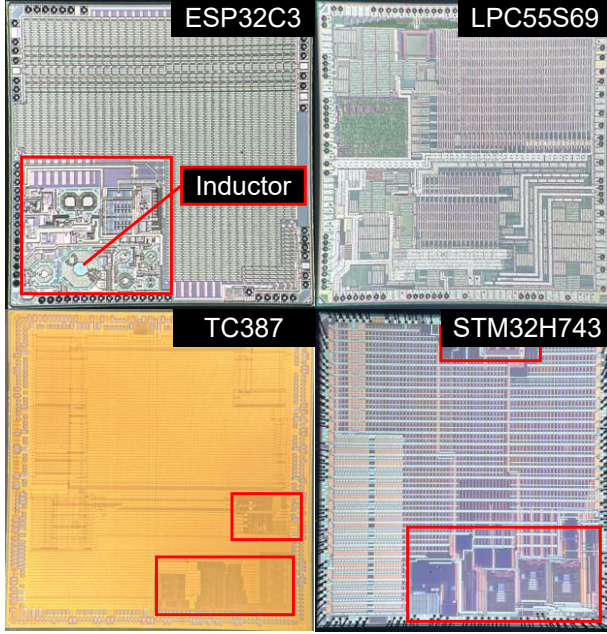


Figure 11. SoC dieshots of the ESP32C3, LPC55S69, TC387, and STM32H743, all of which are 40 nm. Dieshots are not to scale.

To familiarise readers with commercial SoC dies, we present the dieshots in Figure 11. *Potential* analog peripherals are highlighted using red bounding boxes, and the large inductor in the ESP32C3 is clearly visible due to its full integration of the radio-frequency front-end. In contrast, the processor, SRAM, and Flash block details are obscured by the top metal layers and are not directly observable without etching down to the polysilicon (POLY) layer.

B.13 Impact on Feature-rich Real-time Cores

Table 17. Characteristics of feature-rich cores with different extension variants. Floorplan area is in mm².

Configuration	Floorplan area
Base	0.0715 (24%)
Rich	0.2974 (100%)
Rich+V1	0.3132 (105%)
Rich+V2	0.3129 (105%)
Rich+V3	0.3289 (111%)
Rich+V4	0.3291 (111%)
Rich+V5	0.3395 (114%)

To simulate our extension on feature-rich processors, we augment our core with a double-precision FPU, a 4 KiB I-cache (8-byte lines, 4-way set-associative, random replacement), and a 4 KiB D-cache (8-byte lines, 4-way set-associative, random replacement), with both caches implemented using OpenCache. We then recompute the area overheads of our extensions in Table 17. Notably, high-end processors such as the STM32F767 integrate significantly larger caches (16 KiB I/D) and superscalar pipelines, making our extensions even more negligible.

B.14 Impact on Large Out-of-order Cores

To cater to curious readers who wonder how our extension would perform on out-of-order cores used in edge, mobile, or cloud computing scenarios, we select NaxRiscv [85] in its default configuration (2-decode, 3-issue, 2-retire) as our implementation testbed, owing to its high configurability and modular plugin support. Still, we adopt the OpenROAD [1] toolchain for area and power evaluation and the XC7K480T FPGA platform for runtime execution, with all settings and baselines consistent with those in §5.

Table 18. Detailed configuration of NaxRiscv used in this supplementary experiment.

Group	Item	Content
Basics	Name	NaxRiscv
	Configuration	RV32IMAFD
Pipeline	Stages	Variable; ≥ 9
	Width	2-decode, 3-issue, 2-retire
	Execution unit	$2 \times \text{ALU} + 1 \times \text{MDU/LSU}$
	Reservation station	32-entry
	Reorder buffer	64-entry
	Load queue	16-entry
	Store queue	16-entry
Register	Architectural register	32-1; unified with physical
	Physical register	64; 6-read, 4-write
	Renaming entry	$2 \times 32-1$; 6-read, 4-write
Branch	Prediction algorithm	BTB+RAS+GShare
	Global history	24-bit; 13-bit effective
	Pattern table	16k $\times$ 2-bit
	Return stack	16-entry
Protection	Target buffer	512-entry
	MMU	Turned off
Cache	PMP	8-entry; we added
	L1 Icache	16KiB; 4-way
	L1 Dcache	16KiB; 4-way
Performance	CoreMark	5.51/MHz; FPGA measured
	Interrupt latency	38 cycles; FPGA measured

Core Adaptation. Since NaxRiscv does not natively support PMP, we implement a PMP unit to validate instruction fetch and data load/store operations, and configure the core to operate without an MMU. We also equip the core with zero-wait-state SRAM on the system bus to eliminate extraneous memory performance penalties. The detailed core configuration is presented in Table 18 and is roughly comparable to Cortex-A9-class application processors, which are widely deployed in routers and IP cameras.

Design space analysis. Unlike compact in-order cores, regular out-of-order cores must wait until *all in-flight instructions*

are committed and register renaming and reordering operations are fully resolved before servicing interrupts. We refrain from invasive modifications to change the design to eager flushing, as our work is an extension rather than a full redesign. This results in an interrupt latency of 38 cycles on the vanilla NaxRiscv core, rendering it poorly suited for latency-sensitive real-time applications. As the core does not have a standalone Architectural Register File (ARF), the renaming table has to be read to determine which physical registers need to be saved.

General implementation. Given the long pipeline latency, fast interrupt identification brings no practical latency benefit, and we adopt the RAM-based IID table design. Once an interrupt is classified as user-level, the execution budget and PMP configurations are preloaded from dedicated tables. While budget loading incurs no issues, PMP loading introduces subtle hazards, since unresolved in-flight instructions from interrupted execution must still be validated against the original PMP permissions. To resolve this hazard, we introduce an additional independent PMP register bank, yielding a three-bank PMP: one for kernel privilege execution, one for active user-level interrupt contexts, and one dedicated to buffering newly loaded PMP entries during ongoing pipeline execution. Even the inherently long pipeline-draining latency, all of interrupt identification, budget fetching, and PMP re-configuration can complete entirely within this draining window. All memory accesses for these operations are scheduled on the TCM port, eliminating bus contention with residual pipeline execution.

Register context saving implementation. A challenge lies in efficient register context saving. Two strategies exist: (1) full banking of the entire rename table and physical register file, or (2) selective register preservation by renaming table lookup. While (1) enables single-cycle context swapping, we argue that it delivers diminishing returns in our scenario, since the baseline pipeline-drain latency already takes 38 cycles, the gain of single-cycle register switching is marginal at the cost of substantial area, and this will cause the extension overheads to grow with the number of physical registers. We therefore adopt (2): after the pipeline reaches full quiescence, we traverse the rename table to extract the physical registers mapped to each architectural register, zeroize their entries, and move them to a dedicated register bank. The physical register file does not come with global reset support in the original design, and only four write ports are available for zeroing; this limits the design to clearing four registers per cycle, which requires 11 cycles in total for full register banking and zeroization, and yields a total interrupt latency of 49 cycles for user-level interrupt handling. To sum up, we colocate all of the IID table, PMP table, budget table, and the context spill stack within a dedicated TCM region, and call this variant EXT-VR.

Implementation complexity and effort. In contrast to RTOSUnit [77], our implementation complexity remains minimal: we only (1) modify the interrupt controller, (2) tap into the retire-done signal, and (3) multiplex the register files to interface the TCM. The augmented PMP logic and corresponding exception signals are inserted at the same pipeline stage as the original MMU, leaving the out-of-order execution core completely untouched. We implemented this extension from scratch on vanilla NaxRiscv in merely three person-weeks, *with no prior familiarity with the SpinalHDL language, much less the NaxRiscv processor architecture.*

Table 19. Characteristics of different processor configurations. Area is in mm², latency is in CPU cycles, and power is in static (mW) / dynamic-benchmark (μ W/MHz) / dynamic-back-to-back-interrupt (μ W/MHz) format without any SRAM macro power. “+PMP” adds an 8-entry PMP, “+V5/VR” adds our hardware extension.

Processor	Latency	Floorplan Area	Power		
Base+V5	11	0.1186 (6.61%)	2.53	23.60	26.60
NaxRiscv	38	1.779 (99.2%)	9.65	564.50	469
NaxRiscv+PMP	38	1.794 (100%)	9.95	568.50	470.5
NaxRiscv+PMP/VR	49	1.833 (102%)	10.75	571.50	504.5
Kernel	565	N/A	N/A		

Area and power overheads. We evaluate area and power overheads of VR using the same methodology as in §4, as summarized in Table 19. To better expose logic overheads, we exclude all internal NaxRiscv SRAM structures, which are mainly caches, from power analysis. Our results show that both the PMP and our extension introduce negligible overhead: adding a PMP increases area by less than 1%, static power by 3%, and dynamic power by 0.7%, while our VR extension adds an additional 2% area, 8% static power, and 0.53% dynamic power. Given that commercial application processors include large L2 caches, graphics engines, and external memory controllers, we expect the overhead of our extension to be even smaller relative to a full die.

Core comparison. We further compare NaxRiscv+PMP/VR with our V5-extended core. As shown in Table 19, complex out-of-order cores *deliver a poor PPA trade-off for latency-sensitive real-time workloads*. Compared with our three-stage in-order core, NaxRiscv has 15× more area, 4× higher static power, and 24× higher dynamic power, while suffering 4× worse interrupt latency. Admittedly, NaxRiscv measured 5.51 CoreMark/MHz, almost doubling our 2.77 CoreMark/MHz. However, typical interrupt handlers are short, making general single-threaded performance gains less pronounced. This trend is also validated in Figure 10, where NaxRiscv performs poorly on round-trip synchronous invocations. Such short kernel-bound paths exhibit minimal instruction-level parallelism, leaving out-of-order execution unable to compensate for its intrinsic pipeline overhead. In practical scenarios, this latency is expected to degrade further due to external dynamic memories and enabled MMU translation.

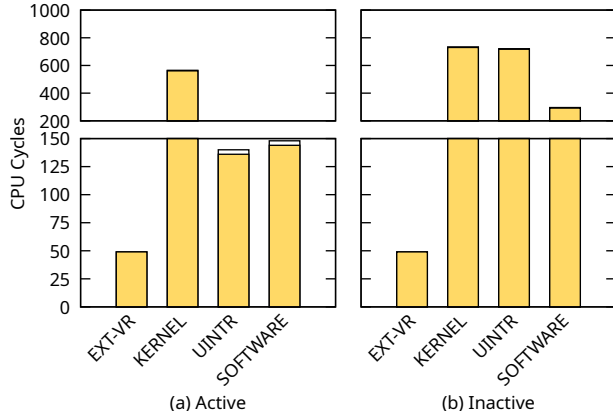


Figure 12. Raw interrupt latency, in CPU cycles. Lower is better.

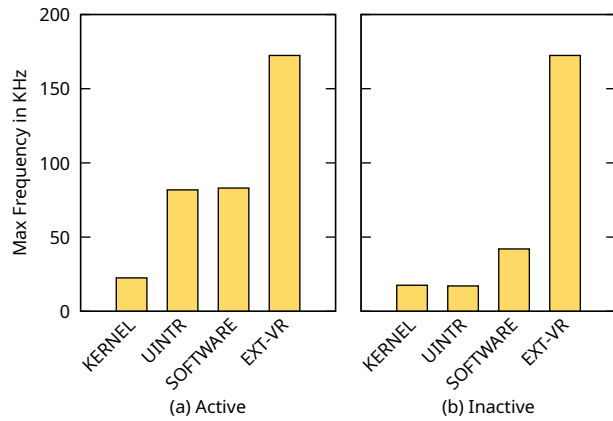


Figure 13. Maximum achievable PTO frequency. Higher is better.

Therefore, typical out-of-order cores are only suitable for systems with relaxed latency requirements, heavy background computation, or full-stack OS support such as Linux. or systems demanding both low deterministic latency and computing capability, *industry practice adopts heterogeneous core combinations*, pairing real-time cores with high-performance out-of-order cores—for instance, Cortex-M4 with Cortex-A9.

Interrupt latency comparison. We follow the comparison from §5.2 and show the results in Figure 12. Our extension still outperforms all three baselines by a significant margin, albeit with a reduced advantage: 10× faster than KERNEL and 3× faster than SOFTWARE. Notably, the gap between UINTR and SOFTWARE under active scenarios becomes negligible, as the high baseline interrupt latency dominates and diminishes the relative overhead of bounds checking, which we emulate via PMP programming. SOFTWARE remains superior in target-inactive scenarios.

Latency-sensitive application. We rerun the PTO experiment under settings identical to those in §5.4 and show the results in Figures 13 and 14. The core conclusions still hold; however, the inherently long interrupt latency of the

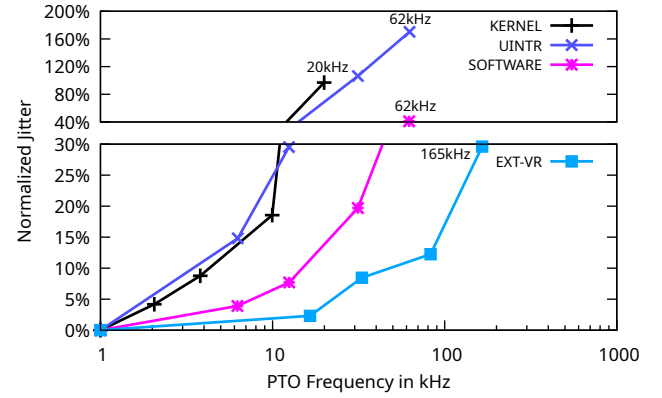


Figure 14. Normalized waveform jitter as a function of PTO frequency, under rapid process interleave interference. Lower is better.

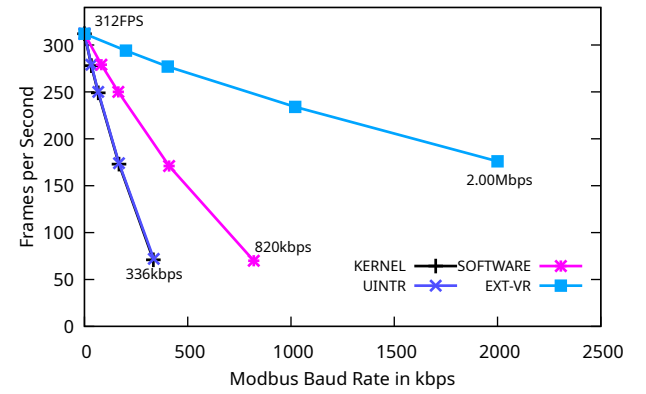


Figure 15. Achievable FPS of a background image recognition task as a function of foreground Modbus-RTU baud rate. Higher is better.

out-of-order core leads to two notable changes. First, our extension achieves a lower sustainable operating frequency than in the in-order setup, reducing the performance gap over the KERNEL baseline to 7×. Second, the out-of-order core’s execution unpredictability *drastically worsens jitter* for all three baselines, yielding a maximum jitter close to 200%, which indicates pulse skipping—an unacceptable behavior for motion control. Although our extension still substantially outperforms all baselines, it also suffers mild degradation, exhibiting up to 30% jitter at maximum attainable frequency, which is suboptimal.

Task-colocation application. We rerun the Modbus colocation experiment under settings identical to those in §5.5 and show the results in Figure 15. The out-of-order NaxRiscv achieves more than twice the 143 FPS attained by our core, consistent with its CoreMark score. Nevertheless, NaxRiscv delivers a *lower maximum baud rate than our simple core*, owing to its long interrupt latency. Still, even on this out-of-order platform, our extension surpasses all baselines in both

maximum baud rate and processor utilization. Interestingly, the KERNEL and UINTR perform better on NaxRiscv than on our in-order core, likely benefiting from the out-of-order core's higher throughput.

Conclusions. From all of the above, we conclude that (1) our extension features low implementation invasiveness and minimal complexity, (2) our extension retains low deterministic latency *even on processors it was not designed for* and significantly outperforms the baselines, (3) higher general-purpose processing performance *does not necessarily imply better real-time responsiveness*, and (4) out-of-order cores are *less suited for hard real-time applications* demanding low latency and strict predictability, further validating that *a predictable architecture is an appropriate design choice*.

Potential impact. Although our conclusions state that out-of-order cores deliver poor trade-offs for hard real-time systems, simpler in-order multi-issue cores may achieve better timing predictability [78], and both of them may remain viable for computing-intensive scenarios with relaxed latency requirements. A typical example is Robot Operating System (ROS) co-deployed with a more predictable microkernel on symmetric multiprocessor platforms. With our extension, these processors can further improve the predictability of the microkernel with negligible overhead. MMU-based processors may also benefit from our extension when equipped with extension-managed lockable TLBs; this does not preclude coexisting walker-filled TLBs, and just requires the extension-managed entries to take precedence. As a promising future direction, this extension may be ported to cloud platforms to further mitigate tail latency.

B.15 Miscellaneous Notes

Precedents. Notably, [17, 67] provided related clues. Nevertheless, [17] adopted a full hardware-RTOS perspective and did not investigate memory isolation, while [67] focused on a single-domain user-level interrupt paradigm leveraging existing hardware. Neither formulated this multi-tenant abstraction nor systematically explored the design space.

Side-channels. As all interrupt handlers execute on a shared single core, timing side channels inherently exist across them. Possible mitigation strategies include (1) restricting handlers from accessing global low-level timing counters or exploitable peripherals—certain peripherals such as Analog-to-Digital Converters (ADCs) feature mostly fixed conversion times that enable coarse-grained wall-clock time estimation, (2) pausing individual low-level counters whenever the corresponding handler is inactive, effectively virtualizing them, (3) employing simple in-order cores to eliminate timing variability introduced by out-of-order execution and other microarchitectural effects. As our extension saves and restores all architectural states automatically on interrupt entry and return, no observable storage channel exists assuming a correct operating system kernel implementation [60, 104].

DMA management. As DMAs can circumvent CPU-side protection, it is imperative that they be managed by the kernel or trusted process. Allowing application access to an unrestricted DMA engine will break all isolation boundaries.

Brief response time analysis. If we assume that (1) all interrupts are periodic or have a minimum inter-arrival time, (2) each interrupt carries an implicit deadline no greater than its period or minimum inter-arrival time, and (3) the server budgets are never exhausted, the exact worst-case response time R_i of interrupt i can be computed via the following recurrence [5]:

$$R_i^{(n+1)} = L_i^{\text{entry}} + C_i + L_i^{\text{exit}} + \sum_{j \in hp(i)} \left\lceil \frac{R_i^{(n)} + J_j}{T_j} \right\rceil (L_j^{\text{entry}} + C_j + L_j^{\text{exit}}), \quad (1)$$

where L_i^{entry} is the entry latency, C_i is the worst-case computation time, L_i^{exit} is the return latency, J_i is the release jitter, T_i is the period or minimum inter-arrival time, and $hp(i)$ is the set of interrupts with higher priority.

For our extension, the interrupt return latency is *always smaller than the entry latency*, which allows us to simplify the recurrence as follows with some pessimism:

$$R_i^{(n+1)} = 2L_i^{\text{entry}} + C_i + \sum_{j \in hp(i)} \left\lceil \frac{R_i^{(n)} + J_j}{T_j} \right\rceil (2L_j^{\text{entry}} + C_j). \quad (2)$$

If it converges and the converged R_i plus J_i is below D_i , which is the deadline of interrupt i , no misses occur.

When considering potential server budget exhaustion, servers can be modeled as schedulable entities as a whole, leading to a similar server-level recurrence [13]. A detailed analysis is omitted, as budget exhaustion corresponds to erroneous or malicious operations that require mode changes for mitigation, which falls beyond the scope of this work.

Real product considerations. The temporal isolation mechanism adopted in this work relies on conventional preemptive deferrable server semantics, while user-level handlers are prohibited from globally disabling interrupts.

Yet, several practical enhancement features can be integrated into real products to improve applicability: (1) limiting the maximum duration of each individual handler activation, which enables first-trigger detection of anomalous behaviors, (2) capping the total number of interrupts independent of budget consumption, which regularizes interference patterns on lower-priority execution, (3) supporting multiple non-preemptive sub-priorities within a single preemptive priority level, which enhances flexibility, and (4) adding a user-level custom instruction for temporarily disabling interrupts for less than the L_i^{entry} , which allows bounded critical sections for sensitive hardware manipulation without breaking temporal isolation.

References

- [1] Tutu Ajayi, Vidya A. Chhabria, Mateus Fogaça, Soheil Hashemi, Abdelrahman Hosny, Andrew B. Kahng, Minsoo Kim, Jeongsup Lee, Uday Mallappa, Marina Neseem, Geraldo Pradipta, Sherief Reda, Mehdi Saligane, Sachin S. Sapatnekar, Carl Sechen, Mohamed Shalan, William Swartz, Lutong Wang, Zhehong Wang, Mingyu Woo, and Bangqi Xu. 2019. Toward an Open-Source Digital Flow: First Learnings from the OpenROAD Project. In *Proceedings of the 56th Annual Design Automation Conference 2019 (DAC '19)*. Association for Computing Machinery.
- [2] Hesham Almatary, Michael Dodson, Jessica Clarke, Peter Rugg, Ivan Gomes, Michal Podhradsky, Peter G Neumann, Simon W Moore, and Robert NM Watson. 2022. CompartOS: CHERI compartmentalization for embedded systems. *arXiv preprint arXiv:2206.02852* (2022).
- [3] Saar Amar, Tony Chen, David Chisnall, Nathaniel Wesley Filardo, Ben Laurie, Hugo Lefeuvre, Kunyan Liu, Simon W Moore, Robert Norton-Wright, Margo Seltzer, et al. 2025. CHERIoT RTOS: An OS for fine-grained memory-safe compartments on low-cost embedded devices. In *Proceedings of the ACM SIGOPS 31st Symposium on Operating Systems Principles (SOSP)*. 67–84.
- [4] ARM Limited. 2007. *Cortex-M3 Technical Reference Manual*. ARM Limited, Cambridge, UK. Available at <https://developer.arm.com/documentation/ddi0337/e>.
- [5] Neil Audsley, Alan Burns, Mike Richardson, Ken Tindell, and Andy J Wellings. 1993. Applying new scheduling theory to static priority pre-emptive scheduling. *Software engineering journal* 8, 5 (1993), 284–292.
- [6] AUTOSAR 2020. Adaptive AUTOSAR. The AUTOSAR Runtime for Adaptive Applications, retrieved 12/14/20. <https://www.autosar.org/standards/adaptive-platform>
- [7] Berk Aydogmus, Linsong Guo, Danial Zuberi, Tal Garfinkel, Dean Tullsen, Amy Ousterhout, and Kazem Taram. 2025. Extended User Interrupts (xUI): Fast and Flexible Notification without Polling. In *Proceedings of the 30th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2*.
- [8] Weibang Bai, Qixin Cao, Pengfei Wang, Peng Chen, Chuntao Leng, and Tiewen Pan. 2017. Modular design of a teleoperated robotic control system for laparoscopic minimally invasive surgery based on ROS and RT-Middleware. *Industrial Robot: An International Journal* (2017).
- [9] Robert Balas, Alessandro Ottaviano, and Luca Benini. 2024. CV32RT: Enabling Fast Interrupt and Context Switching for RISC-V Microcontrollers. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems* (2024).
- [10] Malte Bargholz, Christian Dietrich, and Daniel Lohmann. 2022. PSIC: Priority-strict multi-core IRQ processing. In *2022 IEEE 25th International Symposium On Real-Time Distributed Computing (ISORC)*.
- [11] Zhuoqun Cheng, Richard West, and Craig Einstein. 2018. End-to-end analysis and design of a drone flight controller. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* (2018).
- [12] Daniel Danner, Rainer Muller, Wolfgang Schröder-Preikschat, Wanja Hofer, and Daniel Lohmann. 2014. SAFER SLOTH: Efficient, hardware-tailored memory protection. In *20th IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*.
- [13] Robert I Davis and Alan Burns. 2005. Hierarchical fixed priority pre-emptive scheduling. In *26th IEEE International Real-Time Systems Symposium (RTSS'05)*.
- [14] Nicolas Dejon, Chrystel Gaber, and Gilles Grimaud. 2023. Pip-MPU: Formal verification of an MPU-based separation kernel for constrained devices. *International Journal of Embedded Systems and Applications* (2023).
- [15] Danesh Derafshi, Amin Norollah, Mohsen Khosroanjam, and Hakem Beitollahi. 2020. HRHS: A High-Performance Real-Time Hardware Scheduler. *IEEE Transactions on Parallel and Distributed Systems* 31 (2020).
- [16] Philip Derrin, Dhammika Elkaduwe, and Kevin Elphinstone. 2026. seL4 reference manual. *NICTA-National Information and Communications Technology Australia* (2026).
- [17] Christian Dietrich and Daniel Lohmann. 2017. OSEK-V: application-specific RTOS instantiation in hardware. *ACM SIGPLAN Notices* (2017).
- [18] Kha Dinh Duy, Kyuwon Cho, Taehyun Noh, and Hojoon Lee. 2023. Capacity: Cryptographically-enforced in-process capabilities for modern arm architectures. In *Proceedings of the 2023 ACM SIGSAC Conference on Computer and Communications Security*. 874–888.
- [19] Eugen Dodiou and Vasile Gheorghita Gaitan. 2012. Custom designed CPU architecture based on a hardware scheduler and independent pipeline registers — Concept and theory of operation. In *2012 IEEE International Conference on Electro/Information Technology*.
- [20] DPK [n. d.]. Intel Data Plane Development Kit (DPDK). <https://dpdk.org/>.
- [21] Dong Du, Zhichao Hua, Yubin Xia, Binyu Zang, and Haibo Chen. 2019. XPC: architectural support for secure and efficient cross process call. In *Proceedings of the 46th International Symposium on Computer Architecture*. 671–684.
- [22] duktape 2020. Duktape. An embeddable Javascript engine with a focus on portability and compact footprint, <https://duktape.org>, retrieved 12/14/20.
- [23] Johan Eriksson, Fredrik Häggström, Simon Aittamaa, Andrey Kruglyak, and Per Lindgren. 2013. Real-time for the masses, step 1: Programming API and static priority SRP kernel primitives. In *2013 8th IEEE international symposium on industrial embedded systems (SIES)*.
- [24] f9micro 2013. F9 microkernel, retrieved 6/2/24. <https://github.com/f9micro/f9-kernel>
- [25] Phani Kishore Gadepalli, Runyu Pan, and Gabriel Parmer. 2020. Slite: OS Support for Near Zero-Cost, Configurable Scheduling. In *IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*.
- [26] Shay Gal-On and Markus Levy. 2012. Exploring CoreMark — a benchmark maximizing simplicity and efficacy. *The Embedded Microprocessor Benchmark Consortium* (2012).
- [27] Winfried Grunewald and Theo Ungerer. 1996. Towards extremely fast context switching in a block-multithreaded processor. In *Proceedings of EUROMICRO 96. 22nd Euromicro Conference. Beyond 2000: Hardware and Software Design Strategies*. 592–599.
- [28] Robbert Gurdeep Singh and Christophe Scholliers. 2019. WARDuino: A Dynamic WebAssembly Virtual Machine for Programming Microcontrollers. In *ACM SIGPLAN International Conference on Managed Programming Languages and Runtimes (MPLR)*.
- [29] John Hauser. 2019. Berkeley HardFloat Floating-Point Arithmetic Package. <https://www.jhauser.us/arithmetic/HardFloat.html>
- [30] Mohammad Hedayati, Spyridoula Gravani, Ethan Johnson, John Criswell, Michael L Scott, Kai Shen, and Mike Marty. 2019. Hodor: {Intra-Process} isolation for {High-Throughput} data plane libraries. In *2019 USENIX Annual Technical Conference (USENIX ATC 19)*.
- [31] Wanja Hofer, Daniel Lohmann, Fabian Scheler, and Wolfgang Schröder-Preikschat. 2009. Sloth: Threads as interrupts. In *2009 30th IEEE Real-Time Systems Symposium (RTSS)*.
- [32] Yiyang Huang, Yuhui Hao, Bo Yu, Feng Yan, Yuxin Yang, Feng Min, Yinhe Han, Lin Ma, Shaoshan Liu, Qiang Liu, and Yiming Gan. 2025. Dadu-Corki: Algorithm-Architecture Co-Design for Embodied AI-powered Robotic Manipulation. In *Proceedings of the 52nd Annual International Symposium on Computer Architecture*.
- [33] Zhe Huang, Xingyao Chen, Feng Gao, Ruige Li, Xiguang Wu, and Fan Zhang. 2024. Sophon: A Time-Repeatable and Low-Latency Architecture for Embedded Real-Time Systems Based on RISC-V.

- IEEE Transactions on Very Large Scale Integration (VLSI) Systems* 33, 1 (2024), 221–233.
- [34] Texas Instruments. [n. d.]. TMS320C28x CPU and Instruction Set Reference Guide.
- [35] Intel. 2025. Volume 3: Full System Programming Guide. *Intel 64 and IA-32 Architectures Software Developer's Manuals* (2025).
- [36] Intel Corporation. 1981. MCS-51 Family of Single Chip Microcontrollers User's Manual.
- [37] Yuekai Jia, Kaifu Tian, Yuyang You, Yu Chen, and Kang Chen. 2024. Skyloft: A General High-Efficient Scheduling Framework in User Space. In *Proceedings of the ACM SIGOPS 30th Symposium on Operating Systems Principles*. 265–279.
- [38] Mathai Joseph and Paritosh Pandya. 1986. Finding response times in a real-time system. *Comput. J.* 29, 5 (1986), 390–395.
- [39] Kostis Kaffes, Timothy Chong, Jack Tigar Humphries, Adam Belay, David Mazières, and Christos Kozyrakis. 2019. Shinjuku: Preemptive Scheduling for μ second-scale Tail Latency. In *16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19)*. 345–360.
- [40] Arslan Khan, Dongyan Xu, and Dave Jing Tian. 2023. Low-cost privilege separation with compile time compartmentalization for embedded systems. In *2023 IEEE Symposium on Security and Privacy (S&P)*.
- [41] Chung Hwan Kim, Taegyu Kim, Hongjun Choi, Zhongshu Gu, Byoungyoung Lee, Xiangyu Zhang, and Dongyan Xu. 2018. Securing Real-Time Microcontroller Systems through Customized Memory View Switching.. In *Network and Distributed System Security Symposium (NDSS)*.
- [42] Poonacha Kongetira, Kathirgamar Aingaran, and Kunle Olukotun. 2005. Niagara: A 32-way multithreaded SPARC processor. *IEEE micro* (2005).
- [43] Szymon Kubica and Marios Kogias. 2024. μ BPF: Using eBPF for Microcontroller Compartmentalization. In *Proceedings of the ACM SIGCOMM 2024 Workshop on eBPF and Kernel Extensions*.
- [44] Simon Kuenzer, Vlad-Andrei Bădoi, Hugo Lefeuve, Sharan Santhanam, Alexander Jung, Gauthier Gain, Cyril Soldani, Costin Lupu, Ștefan Teodorescu, Costi Răducanu, et al. 2021. Unikraft: fast, specialized unikernels the easy way. In *Proceedings of the Sixteenth European Conference on Computer Systems*. 376–394.
- [45] Ian Kuon and Jonathan Rose. 2006. Measuring the Gap between FPGAs and ASICs. In *Proceedings of the 2006 ACM/SIGDA 14th International Symposium on Field Programmable Gate Arrays*.
- [46] Donghyun Kwon, Jangseop Shin, Giyeol Kim, Byoungyoung Lee, Yeongpil Cho, and Yunheung Paek. 2019. uXOM: Efficient eXecute-Only Memory on ARM Cortex-M. In *USENIX Security Symposium (USENIX Security)*.
- [47] John P Lehoczy. 1990. Fixed priority scheduling of periodic task sets with arbitrary deadlines. In *Proceedings 11th Real-Time Systems Symposium (RTSS)*.
- [48] Amit Levy, Bradford Campbell, Branden Ghena, Daniel B Giffin, Pat Pannuto, Prabal Dutta, and Philip Levis. 2017. Multiprogramming a 64 kB Computer Safely and Efficiently. In *Proceedings of the 26th Symposium on Operating Systems Principles (SOSP)*.
- [49] Guanyu Li, Dong Du, and Yubin Xia. 2020. Iso-UniK: lightweight multi-process unikernel through memory protection keys. *Cybersecurity* (2020).
- [50] Junchao Li, Runsheng Hou, Guangyong Shang, Huanle Zhang, Xiuzhen Cheng, and Runyu Pan. 2025. FVM: Practical Feather-Weight Virtualization on Commodity Microcontrollers. *IEEE Trans. Comput.* (2025).
- [51] Jialun Li, Danyang Xiao, Jieqian Yao, Yujie Long, and Weigang Wu. 2023. Learning Scheduling Policies for Co-Located Workloads in Cloud Datacenters. *IEEE Transactions on Cloud Computing* (2023).
- [52] Yueying Li, Nikita Lazarev, David Koufaty, Tenny Yin, Andy Anderson, Zhiru Zhang, G Edward Suh, Kostis Kaffes, and Christina Delimitrou. 2024. Libpreemptible: Enabling fast, adaptive, and hardware-assisted user-space scheduling. In *2024 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*. 922–936.
- [53] ARM Limited. 2005. ARM v5 Architecture Reference Manual.
- [54] Jiazhen Lin, Youmin Chen, Shiwei Gao, and Youyou Lu. 2024. Fast core scheduling with userspace process abstraction. In *Proceedings of the ACM SIGOPS 30th symposium on operating systems principles*.
- [55] lowRISC. 2017. Ibex RISC-V Core. <https://github.com/lowRISC/ibex>
- [56] Zhiyao Ma, Guojun Chen, Zhuo Chen, and Lin Zhong. 2025. Hopter: a Safe, Robust, and Responsive Embedded Operating System. In *Proceedings of the 23rd Annual International Conference on Mobile Systems, Applications and Services*. 556–569.
- [57] David May. 2009. *The XMOS XS1 Architecture*. XMOS Semi. Ltd.
- [58] MicroEJ [n. d.]. MicroEJ for Microcontrollers: <http://www.microej.com>, retrieved 10/6/17.
- [59] MicroPython [n. d.]. MicroPython for Microcontrollers: <http://www.micropython.org>, retrieved 10/6/17.
- [60] MIPS Technologies, Inc. 2005. *MIPS32 Architecture For Programmers, Volume I-IV*. MIPS Technologies, Inc.
- [61] mJS [n. d.]. mJS Embedded JavaScript Engine for C/C++: <http://github.com/cesanta/mjs>, retrieved 10/6/17.
- [62] Andrew Morton and Wayne M. Loucks. 2004. A hardware/software kernel for system on chip designs. In *Proceedings of the 2004 ACM Symposium on Applied Computing*. Association for Computing Machinery.
- [63] Takumi Nakano, Andy Utama, Mitsuyoshi Itabashi, Akichika Shiomi, and Masaharu Imai. 1995. Hardware implementation of a real-time operating system. *IEEE Transactions on Electronics, Information and Systems* 116, 1 (1995), 117–124.
- [64] Antti Nurmi, Abdesattar Kalache, Henri Lunnikivi, Per Lindgren, and Timo D. Hämäläinen. 2025. Efficient and Predictable Context Switching for Mixed-Criticality and Real-Time Systems. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems* (2025).
- [65] Daniel Oliveira, Tiago Gomes, and Sandro Pinto. 2022. uTango: an open-source TEE for IoT devices. *IEEE Access* (2022).
- [66] Padauk Technology Co., Ltd. 2024. *PFC887 Industrial Grade 8-bit MTP MCU (FPPA) Datasheet*. Padauk Technology Co., Ltd.
- [67] Runyu Pan and Gabriel Parmer. 2022. SBIs: Application access to safe, baremetal interrupt latencies. In *2022 IEEE 28th Real-Time and Embedded Technology and Applications Symposium (RTAS)*.
- [68] Runyu Pan, Gregor Peach, Yuxin Ren, and Gabriel Parmer. 2018. Predictable Virtualization on Memory Protection Unit-based Microcontrollers. In *24th IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*.
- [69] Gregor Peach, Runyu Pan, Zhuoyi Wu, Gabriel Parmer, Christopher Haster, and Ludmila Cherkasova. 2020. eWASM: Practical Software Fault Isolation for Reliable Embedded Devices. In *Proceedings of the International Conference on Embedded Software (EMSOFT)*.
- [70] Sandro Pinto, Hugo Araujo, Daniel Oliveira, José Martins, and Adriano Tavares. 2019. Virtualization on TrustZone-enabled microcontrollers? Voilà!. In *IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*.
- [71] Nader I. Rafla and Deepak Gauba. 2011. Hardware implementation of context switching for hard real-time operating systems. In *2011 IEEE 54th International Midwest Symposium on Circuits and Systems (MWSCAS)*.
- [72] Mahfuzur Rahman and Bashir I Morshed. 2025. CardioResp Device: Hardware and Firmware of an Embedded Wearable for Real-Time ECG and Respiration in Dynamic Settings. *Electronics* (2025).
- [73] RISC-V. 2025. Unratified RISC-V N Extension Draft. <https://github.com/riscv/riscv-isa-manual/commit/87a1e5f400f989467831a9352265c82b4a1848aa>
- [74] Marlyn Rosales and Nima Eskandari. 2023. *C2000 ePWM Developer's Guide*. Texas Instruments.

- [75] Bruno Sá, José Martins, and Sandro Pinto. 2021. A first look at RISC-V virtualization from an embedded systems perspective. *IEEE Trans. Comput.* (2021).
- [76] Vasily A Sartakov, Lluís Vilanova, and Peter Pietzuch. 2021. Cubicleos: A library os with software componentisation for practical isolation. In *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*.
- [77] Markus Scheck, Tammo Mürmann, and Andreas Koch. 2026. Co-Exploration of RISC-V Processor Microarchitectures and FreeRTOS Extensions for Lower Context-Switch Latency. In *Proceedings of the 31st ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS), Volume 2*. ACM.
- [78] Martin Schoeberl, Wolfgang Puffitsch, Stefan Hepp, Benedikt Huber, and Daniel Prokesch. 2018. Patmos: A time-predictable microprocessor. *Real-Time Systems* (2018).
- [79] David Schrammel, Samuel Weiser, Stefan Steinegger, Martin Schwarzl, Michael Schwarz, Stefan Mangard, and Daniel Gruss. 2020. Donky: Domain keys-efficient {In-Process} isolation for {RISC-V} and x86. In *29th USENIX Security Symposium (USENIX Security 20)*.
- [80] Leon Schuermann, Brad Campbell, Branden Ghena, Philip Levis, Amit Levy, and Pat Pannuto. 2025. Tock: From Research to Securing 10 Million Computers. In *Proceedings of the ACM SIGOPS 31st Symposium on Operating Systems Principles*.
- [81] Wenyuan Shao, Xinyu Han, Evan Stella, Linnea Dierksheide, Phani Kishore Gadepalli, and Gabriel Parmer. 2025. Janus: OS Support for a Secure, Fast Control-Plane. In *2025 IEEE 31st Real-Time and Embedded Technology and Applications Symposium (RTAS)*.
- [82] Siemens. 2019. *SIMATIC S7 S7-1200 Programmable Controller Manual*. Siemens AG.
- [83] Sipeed. [n. d.]. TinyMaix, a Tiny Inference Neural Network Library Specifically for Microcontrollers. <https://github.com/sipeed/TinyMaix>
- [84] SPDK [n. d.]. Intel Storage Performance Development Kit (SPDK). <http://spdk.io/>.
- [85] SpinalHDL. [n. d.]. NaxRiscv: An out-of-order RISC-V CPU core. <https://github.com/SpinalHDL/NaxRiscv>.
- [86] Frank Stanischewski. 1993. FASTCHART-Performance, Benefits and Disadvantages of the Architecture. In *Fifth Euromicro Workshop on Real-Time Systems*.
- [87] Jay K. Strosnider, John P. Lehoczky, and Lui Sha. 1995. The Deferrable Server Algorithm for Enhanced Aperiodic Responsiveness in Hard Real-Time Environments. *IEEE Trans. Comput.* 44, 1 (1995), 73–91. doi:10.1109/12.368008
- [88] Arun Kumar Sundar Rajan, Armin Feucht, Lothar Gamer, Idriz Smaili, and Nirmala Devi M. 2018. Hypervisor for consolidating real-time automotive control units: Its procedure, implications and hidden pitfalls. *Journal of Systems Architecture* 82 (2018), 37–48.
- [89] Sadik Tamboli, Mallikarjun Rawale, Rupesh Thoraiet, and Sudhir Agashe. 2015. Implementation of Modbus RTU and Modbus TCP communication using Siemens S7-1200 PLC for batch process. In *2015 International Conference on Smart Technologies and Management for Computing, Communication, Controls, Energy and Materials (ICSTM)*.
- [90] Yi Tang and Neil W. Bergmann. 2002. Technical Reference Manual for RTU operating System Accelerator.
- [91] Yi Tang and Neil W. Bergmann. 2015. A Hardware Scheduler Based on Task Queues for FPGA-Based Embedded Real-Time Systems. *IEEE Trans. Comput.* 64 (2015).
- [92] Texas Instruments. 2023. *ISO15, ISO35, ISO15M, ISO35M Isolated 3.3-V Half- and Full-Duplex RS-485 Transceivers*. <https://www.ti.com/lit/ds/symlink/iso15.pdf>
- [93] James E Thornton. 1964. Parallel operation in the control data 6600. In *Proceedings of the October 27-29, 1964, fall joint computer conference, part II: very high speed computer systems*.
- [94] Ken W Tindell, Alan Burns, and Andy J. Wellings. 1994. An extendible approach for analyzing fixed priority hard real-time tasks. *Real-Time Systems* (1994).
- [95] Anjo Vahldiek-Oberwagner, Eslam Elnikety, Nuno O Duarte, Michael Sammler, Peter Druschel, and Deepak Garg. 2019. ERIM: Secure, efficient in-process isolation with protection keys (MPK). In *28th USENIX Security Symposium (USENIX Security 19)*. 1221–1238.
- [96] Kizheppatt Vipin. 2018. CANNoC: An open-source NoC architecture for ECU consolidation. In *IEEE International Midwest Symposium on Circuits and Systems (MWSCAS)*.
- [97] Alexios Voulimeneas, Jonas Vinck, Ruben Mechelinck, and Stijn Volckaert. 2022. You shall not (by) pass! practical, secure, and fast pku-based sandboxing. In *Proceedings of the Seventeenth European Conference on Computer Systems*. 266–282.
- [98] Andrew Waterman, Krste Asanovic, and John Hauser. 2025. Volume II: Privileged Architecture. *The RISC-V Instruction Set Manual* (2025).
- [99] WCH. 2025. *QingKe v4 Processor Technical Reference Manual*. WCH Co., Ltd.
- [100] Guoqi Xie, Gang Zeng, Zhetao Li, Renfa Li, and Keqin Li. 2017. Adaptive dynamic scheduling on multifunctional mixed-criticality automotive cyber-physical systems. *IEEE Transactions on Vehicular Technology* 66, 8 (2017), 6676–6692.
- [101] Nobuyuki Yamasaki, Ikuo Magaki, and Tsutomu Itou. 2007. Prioritized SMT architecture with IPC control method for real-time processing. In *13th IEEE Real Time and Embedded Technology and Applications Symposium (RTAS'07)*.
- [102] Joseph Yiu. 2014. *The definitive guide to the ARM Cortex-M3 and Cortex-M4 processors*. Newnes.
- [103] Koen Zandberg, Emmanuel Baccelli, Shenghao Yuan, Frédéric Besson, and Jean-Pierre Talpin. 2022. Femto-containers: lightweight virtualization and fault isolation for small software functions on low-power IoT microcontrollers. In *ACM/IFIP International Middleware Conference (Middleware)*.
- [104] Xin Zhang, Chang Liu, Jiajun Zou, Yi Yang, Qingni Shen, Zhi Zhang, and Trevor E Carlson. 2026. Towards Practical Interrupt Side-Channel Attacks on macOS for Apple Silicon. In *2026 ACM/IEEE 53rd Annual International Symposium on Computer Architecture (ISCA)*.
- [105] Lu Zhao, Guodong Li, Bjorn De Sutter, and John Regehr. 2011. ARMor: Fully Verified Software Fault Isolation. In *Proceedings of the Ninth ACM International Conference on Embedded Software (EMSOFT)*.
- [106] Yusheng Zheng, Tong Yu, Yiwei Yang, Yanpeng Hu, Xiaozheng Lai, Dan Williams, and Andi Quinn. 2025. Extending applications safely and efficiently. In *19th USENIX Symposium on Operating Systems Design and Implementation (OSDI 25)*.
- [107] Michael Zimmer, David Broman, Chris Shaver, and Edward A Lee. 2014. FlexPRET: A processor platform for mixed-criticality systems. In *2014 IEEE 19th Real-Time and Embedded Technology and Applications Symposium (RTAS)*.