Co-Exploration of RISC-V Processor Microarchitectures and FreeRTOS Extensions for Lower Context-Switch Latency

Markus Scheck*
scheck@esa.tu-darmstadt.de
Technical University of Darmstadt
Darmstadt, Hesse, Germany

Tammo Mürmann*
muermann@esa.tu-darmstadt.de
Technical University of Darmstadt
Darmstadt, Hesse, Germany

Andreas Koch
koch@esa.tu-darmstadt.de
Technical University of Darmstadt
Darmstadt, Hesse, Germany

Abstract

Embedded real-time systems must respond to external events within tightly bounded timeframes to ensure safety, correctness, and reliability. While Real-Time Operating Systems (RTOSes) ease the development of complex applications by providing abstractions for multi-tasked execution, they introduce overheads in the form of task switching latency and jitter, which impact timing predictability. Minimizing these effects is essential for reducing response times and enabling robust worst-case timing analysis. We present *RTOSUnit*, a configurable hardware acceleration unit designed to reduce context-switch latency and jitter in embedded real-time systems. By integrating the RTOSUnit into three RISC-V cores of varying complexity, we demonstrate its portability. Through a range of configurations, from lightweight scheduling acceleration to full context-switch and -preloading support, RTOSUnit achieves up to 76 % reduction in mean context-switch latency and can be configured to completely eliminate jitter on selected cores. Area overheads in 22 nm ASIC implementations range from negligible (within EDA tool heuristics noise) to 44 %, with all configurations maintaining viable operating frequencies and power envelopes suitable for embedded systems. RTOSUnit offers a flexible and efficient, open-source[1] foundation for hardware-assisted real-time scheduling, paving the way for broader integration into future embedded SoCs.

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

Keywords: FreeRTOS; Context-Switch; Latency; Jitter; Real-Time; RTOS; RISC-V

ACM Reference Format:
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 30th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2 (ASPLOS '26), March 21–26, 2026, Pittsburgh, PA, USA.* ACM, New York, NY, USA, 15 pages. https://doi.org/10.1145/3779212.3790141

1 Introduction

Embedded real-time systems must respond to external stimuli within strict timing bounds to ensure safety, correctness, and reliability. While such guarantees can be achieved with low-level "bare-metal" software implementations, *Real-Time Operating Systems (RTOSes)* simplify development by offering a multi-tasked programming model. This abstraction allows developers to define multiple concurrent tasks that interact via synchronization primitives, promoting modularity and maintainability.

However, this simplification introduces overhead: the RTOS manages task execution by scheduling, state tracking, and context switching – services that, while useful, increase code size and introduce latency. During a context switch, the RTOS saves the current task's state to the stack, invokes the scheduler to select the next task, and restores the selected task's state. The total time required for this process is known as *context-switch latency*.

Context switching directly impacts the system's response time to external events. While simple event handling can be performed within the interrupt service routine (ISR), more complex logic must be deferred to tasks, requiring a full context switch before handling can commence. Consequently, reducing context-switch latency improves the minimal response time to such events. While this work focuses on the deferred handling case, note that our context storing optimizations also accelerate the non-deferred handling case.

Additionally, many scheduling algorithms introduce variability in context-switch durations, reducing predictability and complicating worst-case execution time (WCET) analysis. We define the difference between the maximum and minimum observed context-switch latency as *jitter*. Minimizing both latency and jitter is essential for predictable timing, especially in control loops or under high system load.

*Both authors contributed equally to this research.

[1] https://github.com/esa-tu-darmstadt/RTOSUnit_Integration

ASPLOS '26, Pittsburgh, PA, USA.
© 2026 Copyright held by the owner/author(s).
ACM ISBN 979-8-4007-2359-9/2026/03
https://doi.org/10.1145/3779212.3790141

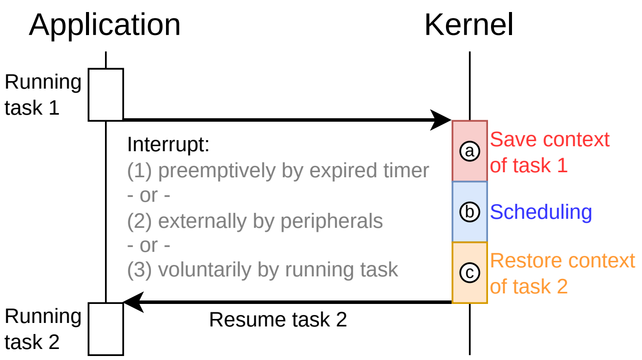

Application
Kernel
Running task 1
Interrupt:
(1) preemptively by expired timer
- or -
(2) externally by peripherals
- or -
(3) voluntarily by running task
ⓐ Save context of task 1
ⓑ Scheduling
ⓒ Restore context of task 2
Running task 2
Resume task 2

Figure 1. Simplified context switching in a typical real-time system. During an interrupt, the processor saves the current task's context, selects the next task, restores its context, and resumes execution from where it was suspended.

To address these challenges, this work introduces a configurable hardware unit into the processor for accelerating scheduling and context switching in FreeRTOS. By offloading key RTOS functions to hardware, our approach reduces worst- and best-case context-switch latency as well as jitter – resulting in faster, more predictable responses. These improvements support the development of embedded real-time systems that must adhere to strict deadlines and response-time bounds, and that require low execution-time variability for effective modeling and certification.

The key contributions of this work are as follows:

- We present a highly configurable hardware unit dedicated to scheduling and context-switching.
- We demonstrate the portability of our approach and its applicability through integration into three different RISC-V processors, ranging from microcontroller (MCU) to powerful superscalar out-of-order cores.
- We greatly reduce context-switch latency and jitter
- We analyze overheads in power, maximum core clock frequency, and area using a 22nm ASIC process.

The remainder of this paper is structured as follows: Section 2 reviews related work. Section 3 describes the target RISC-V cores and the scheduling and context-switching mechanisms of FreeRTOS. Section 4 details the implementation of RTOSUnit, while Section 5 discusses its integration into the RISC-V processors. Section 6 presents an evaluation of RTOSUnit in terms of performance, implementation overhead, and power requirements. Finally, Section 7 summarizes the findings and discusses directions for future work.

2 Related Work

Many attempts at hardware-supported scheduling and context-switching have been made. STRON by Nakano et al. [18] offloads semaphores, task tracking, interrupts, and scheduling to a coprocessor, while context management remains the responsibility of the main processor. A similar hardware/software partitioning is used by Andrews et al. [1] and Vetromille et al. [25]. Both identify task scheduling as a major source of jitter and thus a key candidate for hardware acceleration. Morton et al. [17] quantify the benefits of selective hardware offloading, showing scheduling acceleration provides significant gains with minimal area cost. Chandra et al. [5] also focus on scheduling and synchronization acceleration, opting to design their hardware through high-level synthesis rather than manual HDL development.

However, these approaches rely on *memory-mapped I/O* (MMIO) for communication with the host processor, introducing higher latencies and bus contention. Our approach uses tightly coupled *custom instructions*, reducing overhead and allowing efficient context save/restore acceleration via direct processor integration.

Balas et al. [3] reduce context-switch latency by *snapshotting* half the register file upon interrupt entry,[2] and storing the snapshot values in parallel using a dedicated memory port.[3] The other half of the register file is saved concurrently in software via the LSU's regular memory port. In contrast to our bank switching approach, their method accelerates only *part* of the task's context saving and does *not* provide hardware support for context restoration or scheduling. Additionally, their work is deeply integrated into the CV32E40P processor, while we aim for (and demonstrate) better portability *across* cores via our modular approach.

Rafla et al. [20] integrate custom instructions for context save and restore into a MIPS processor. These instructions interact with a banked context memory and require two cycles to transfer the register state. Similarly, Grunewald et al. [11] propose assigning each task its own register file. Context switching is then reduced to simply switching register banks. While effective, both approaches likely incur high area costs, especially when supporting many concurrent tasks, though neither work actually quantifies this overhead. Both approaches inherently limit the number of tasks to which context save and restore acceleration can be applied. In contrast, our method uses main memory to store task state, making the acceleration *independent* of task count and thus more scalable. We can optionally accelerate scheduling in hardware, as well, but our solution would then also be limited to a statically configured ceiling in the number of tasks.

Some commercial ISAs offer similar mechanisms. ARM AArch32 introduces shadow registers for the Fast Interrupt (FIQ) mode, reducing context-switch latency by removing the need to save/restore the banked registers [2]. Microchip's

[2] https://github.com/pulp-platform/cv32e40p/blob/6c5c2d6bdfb9c5d743aad240ce6e1ae7a0baf1b8/rtl/cv32e40p_register_file_ff.sv#L194-L308

[3] Note that the paper is inconsistent in this regard: While their Fig. 3 depicts a *bank-switching* mechanism, the released implementation actually performs *snapshotting*.

PIC32MZ expands this to seven shadow register sets for fast handling of multiple interrupt sources [16]. While these techniques work well for short non-deferred interrupt handling, they are unsuitable for general-purpose context switching, as the inactive register banks are inaccessible to software, inhibiting a replacement of their contents. Our approach, by contrast, accelerates both context management and task scheduling, supporting deferred and non-deferred interrupt handling alike. Intel's x86 provides a hardware context-switching instruction [7], but it proved to be slower than software solutions under flat memory models as used by modern operating systems, and was removed in x86-64 [8, 12].

Other works [22, 29] follow a vastly different approach for reducing context-switch overhead. Rather than accelerating context saving and restoring directly in hardware, the authors modify the *compiler* to identify so-called *switch points*, i.e., program locations where only a small subset of registers is live. For each switch point, specialized software routines for context saving and restoring are generated, minimizing the size of state that must be preserved. However, when an interrupt occurs, the hardware must defer handling until execution actually reaches the *next* switch point, introducing additional jitter. As a consequence, fully preemptive task scheduling is effectively transformed into a coroutine-like execution model with restricted preemption points. Moreover, this approach is evaluated solely in simulation and has not been implemented in hardware, nor does it provide any acceleration of the scheduling mechanism itself.

Nurmi et al. [19] propose the HETI architecture as an alternative to RTOS-based systems. Their approach models tasks as *interrupt-triggered routines* that are executed directly within an ISR. To reduce interrupt latency, the processor is extended with a dedicated register file bank per interrupt, which is automatically selected on interrupt entry. While their design enables very fast context switching by eliminating register save and restore operations, the hardware area overhead scales rapidly with the number of supported interrupts. Furthermore, the programming model is restrictive compared to an RTOS, supporting only a single main thread and one ISR per interrupt.

Cheshmikhani et al. [6] propose a hardware mechanism that *caches* interrupt handler addresses to avoid repeated handler lookups. Their work targets application-class processors running Linux, where interrupt dispatch involves dynamic and potentially costly handler resolution. In contrast, the small embedded systems that we target commonly rely on either a single shared interrupt handler or a small, statically defined interrupt vector table with constant-time handler selection, which leaves little opportunity for such caching mechanisms to reduce interrupt latency in this domain.

Because scheduling and context switching both contribute significantly to total context-switch latency, and scheduling is a major factor in jitter, combining both in hardware is a natural progression. Zagan et al. [26] replicate all state elements across the processor pipeline and register file, allowing their scheduler coprocessor to select the active context without flushing the pipeline. While this enables near-instantaneous switching, it significantly increases the area and lowers the maximum operating frequency. Doidu et al. [9] present a similar design where all pipeline and register states are banked, and the scheduler runs on the negative clock edge. They report a 0.5-cycle context-switch latency but do not evaluate frequency or area overhead. Given the similarity to Zagan et al., it is likely that their design suffers from the same frequency degradation and area overheads. FASTCHART [14, 15, 24] pushes scheduling, synchronization, and context handling fully into hardware. However, their implementation assumes the entire register file can be stored to memory in a single cycle, which the authors themselves acknowledge as unrealistic [15]. Our approach avoids this impracticality by using register banking and latency-hiding techniques. Furthermore, their approach uses one scheduling queue per priority level, while we implement a unified queue. In consequence, our approach offers greater flexibility by only limiting the *total* number of tasks, not the number of tasks per *priority level*.

In contrast to prior work, our approach combines low-latency scheduling and context switching within a single, tightly integrated hardware unit. Additionally, while some work focuses exclusively on scheduling or context management, and others compromise frequency and area, our solution strikes a practical overall balance. We support real-time constraints by improving both worst-case and best-case latency, reducing jitter, and maintaining compatibility across multiple RISC-V cores. To the best of our knowledge, this combination of configurability, portability, and efficiency has not been achieved previously.

3 HW/SW Platform Selection

To enable the practical implementation and evaluation of our approach, we must select both a processor core and an RTOS for extension.

We adopt the RISC-V Instruction Set Architecture due to its openness, extensibility, and extensive ecosystem of open-source cores spanning a wide range of complexity. Critically, RISC-V supports custom instructions, which we leverage to tightly integrate the proposed functionality. We demonstrate our approach on three representative RISC-V processors: (1) CV32E40P, a microcontroller-class, 4-stage pipeline processor [10], (2) CVA6, an application-class, 6-stage pipeline processor [27], and (3) NaxRiscv, a powerful superscalar out-of-order processor supporting variable-latency instructions

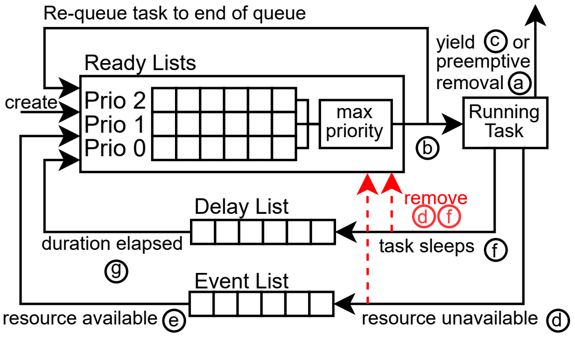


Figure 2. FreeRTOS scheduling algorithm. Tasks are scheduled in a round-robin manner within priorities, while the next task is always drawn from the highest priority of the ready tasks. In addition to the *ready lists*, FreeRTOS maintains a *delay list* for tasks suspended for a specific number of time slices, and *event lists* for tasks waiting on events.

[23]. These processors span a broad spectrum of implementation complexity and are chosen to demonstrate the portability of our approach across diverse RISC-V cores. All three processors are evaluated in their **RV32IM_Zicsr** configuration.

Although RISC-V is particularly well suited to our work, other architectures could be supported with modest engineering effort, provided they offer well-defined interrupt entry and exit boundaries, a clearly specified architectural state to be saved, and support for custom instructions.

For our evaluation, we use the popular FreeRTOS, which is lightweight, open-source, and supports RISC-V. Other RTOSes – such as Zephyr [28] or ChibiOS [21] – feature similar architectures and scheduling algorithms, and could also benefit from RTOSUnit with moderate engineering effort. We expect such extensions to yield comparable results and, consequently, solely focus on FreeRTOS in this work.

Figure 2 shows an overview of FreeRTOS's scheduling algorithm. To enforce fair scheduling among tasks of equal priority, FreeRTOS uses *time slices*: a timer interrupt *preempts* the running task after a fixed duration (a), after which the highest-priority ready task is selected (b). Tasks may also voluntarily yield (c) by triggering a software interrupt.

FreeRTOS provides synchronization primitives such as mutexes and semaphores. When a task attempts to acquire a blocked resource, it is removed from the ready list and placed into the event list (d). It returns to the ready list when the resource becomes available (e). Tasks may also suspend themselves for a defined time, during which they are placed in a delay list (f). This list is updated on every timer interrupt, and tasks whose delay has expired are re-added to the ready list (g). If a task waits on a resource with a timeout, the first event (either unblocking *or* timeout) causes reactivation.

Table 1. Overview of the proposed custom instructions

Custom Instruction	Description	Required for
ADD_READY	Insert task into ready list	HW scheduling
ADD_DELAY	Insert task into delay list	HW scheduling
RM_TASK	Remove task from HW lists	HW scheduling
SET_CONTEXT_ID	Set the next task	**w/o** HW scheduling
GET_HW_SCHED	Get next task from HW	HW scheduling
SWITCH_RF	Switch back to the APP RF	Context storing **w/o** loading

All of these lists store pointers to *Task Control Blocks (TCBs)*, which are expressed as C *structs* that hold essential information for each task. A global variable, **currentTCB**, points to the TCB of the currently running task and is updated for the next task during scheduling.

While the scheduler selects the next task, context switching is responsible for transitioning execution between tasks. As illustrated in Figure 1, context switching comprises three phases: (a) *saving* the current task's context, (b) *scheduling* to select the next task, and (c) *restoring* its context.

The context of a task is saved by pushing its general-purpose registers, processor configuration (e.g., interrupt settings), and program counter onto its stack (cf. Figure 4 (a)). The address of the saved context (i.e., the stack pointer) is stored in the task's TCB, allowing the stored state to be retrieved later (cf. Figure 4 (b)). The RISC-V architecture includes 32 general-purpose registers, one of which is hard-wired to zero. The **global pointer** register provides a fixed offset for accessing global variables and is typically initialized during program startup. The **thread pointer** register points to thread-local storage, with usage defined by the OS. In FreeRTOS, neither register is modified after initialization. Consequently, the **global pointer** and **thread pointer** registers only contain static data in the RTOS execution scenario, leaving 29 registers to be preserved. Additionally, RISC-V uses *Control and Status Registers* (CSRs) to store the processor's configuration and auxiliary information. The **mstatus** CSR (processor state) and the **mepc** CSR (program counter at the time of interruption) are additionally saved to enable task resumption and state restoration. In total, a context comprises 31 32-bit words.

Saving the context not only enables task switching but also frees the registers for use during scheduling. Restoring the new task's context involves retrieving the address of the saved state from its TCB and loading the saved values (cf. Figure 4 (d)). Context switching is initiated by an interrupt originating from a hardware timer (for time slicing), external sources (for deferred interrupt handling), or software (voluntary yield). Execution resumes in the new task using the **mret** instruction.

4 RTOSUnit Implementation

This section presents the RTOSUnit, a configurable hardware accelerator for task scheduling and context switching.

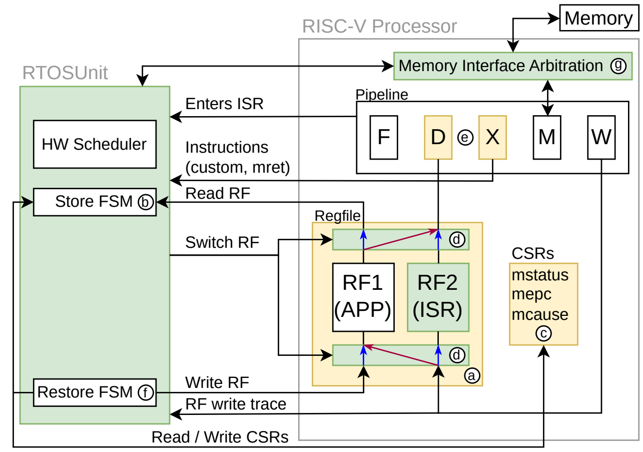


Figure 3. Proposed architecture overview. Added components are shown in green; required processor changes in yellow. Depending on the configuration, the RTOSUnit accelerates scheduling and/or context switching via custom instructions. Additionally, the *register file (RF)* must be duplicated. Blue and magenta arrows indicate active datapaths during ISR and application execution, respectively. A generic five-stage pipeline is shown, while depth and stage functions differ between specifically supported processors.

The design emphasizes fine-grained configurability, allowing users to offload context storing, loading, and/or scheduling to hardware. We select those features since they have the largest impact on context switch duration and, hence, are prime targets for acceleration. We use a combination of letters to describe which functionality has been moved to hardware. (`vanilla`) refers to the baseline case solely implemented in software, while (`SLT`) (Store Context, Load Context, Task Scheduling) has *all* of these operations accelerated by the RTOSUnit.

Unlike memory-mapped approaches, our RTOSUnit uses *custom instructions* for tight integration with the processor, minimizing latency and communication overhead. Table 1 summarizes the new instructions; Figure 3 outlines the architecture and interface required to support the full RTOSUnit feature set. The interface is designed for minimal intrusion into the processor and hence is specific to the use case. The following subsections describe the necessary processor modifications and internal RTOSUnit components by feature.

4.1 Baseline (`vanilla`)

In the baseline case, context switching is handled entirely in software. It is triggered by an interrupt, which may come from a timer (time-slice expiration), software (voluntary yield), or external events. Upon an interrupt, the processor enters an ISR, where the current task's context is saved to its stack (cf. Figure 4 (a)), a ready task is selected (c), and its context is restored (d). If triggered by a timer, delayed tasks are also updated and moved to the ready list during scheduling if their delay has expired. Additionally, the timer is reset only on timer interupts, that is, after its expiry. The ISR concludes with an **`mret`** instruction, resuming execution in the selected task.

4.2 Context Storing (`S`)

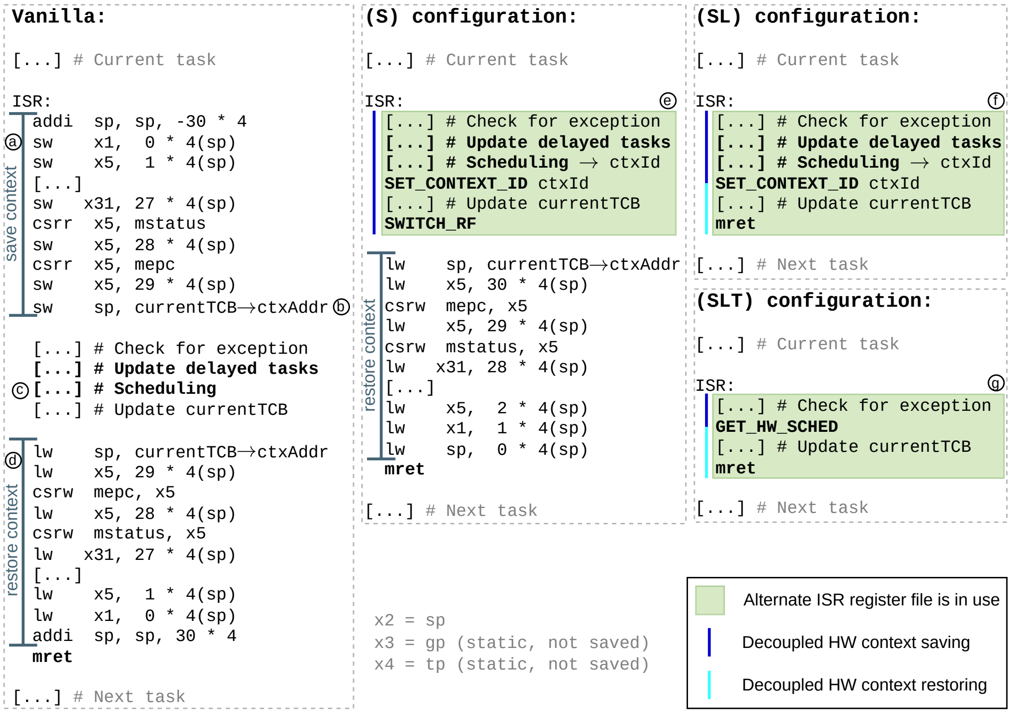
The (`S`) configuration accelerates context storing by enabling immediate scheduler execution after an interrupt, eliminating the need for a software routine to first save the current task's context. Instead, we employ a latency-hiding scheme and decoupled hardware-assisted background context storing.

To this end, we add an *alternate register file bank* to the processor (cf. Figure 3 (a)), duplicating the 29 general-purpose registers required for context representation. The **`mstatus`** and **`mepc`** CSRs do not require replication, since they do not need to be freed for executing the scheduler. Upon an interrupt, the processor frees the register file, making fresh registers immediately available, while a dedicated hardware finite-state machine (cf. Figure 3 (b)) asynchronously stores the previous context to memory. This allows the scheduler to execute without delay (cf. Figure 4 (e)). A similar technique has been validated by Balas et al. [3], but we introduce *additional optimizations* to reduce hardware overhead:

(1) Balas et al. [3] employ *snapshotting* to immediately free *half* of the register file upon interrupt entry. While effective, this approach copies all 16 freed registers in parallel into a second register file bank within a *single cycle*, incurring substantial wiring overhead and routing congestion. To avoid the need for such additional register file ports, we instead use a sparse MUX structure combined with bank switching: only **`RF1`** is connected to both the core and the RTOSUnit, whereas **`RF2`** is connected exclusively to the core (cf. Figure 3(d)). This reduced overhead allows us to duplicate the *entire* register file efficiently, thereby freeing *all* general-purpose registers upon interrupt entry.

(2) We remove the need for a second memory port by *arbitrating* memory access between the processor and RTOSUnit, where the processor has priority (cf. Figure 3 (g)). Hence, both components share a *single* memory port, where the RTOSUnit utilizes the dead/idle cycles not required by the processor.

(3) We reserve a fixed memory region inside the processor's DMEM to save the tasks' contexts and assign each task a 32-word-sized chunk. Additionally, we introduce a unique task ID that allows for quickly deriving the address of the assigned context memory by indexing the 32-word chunks with said ID. Thus, the memory addresses of saved contexts remain *constant*, avoiding updating the saved contexts' memory address in the TCB in contrast to (`vanilla`) (cf. Figure 4 (b)). The size of this fixed memory region defines an upper bound for the task count, but can be chosen arbitrarily. Since we use 32-word-sized chunks, we over-provision by



```
Vanilla:

[...] # Current task

ISR:
 addi  sp, sp, -30 * 4
 sw    x1,  0 * 4(sp)
 sw    x5,  1 * 4(sp)
 [...]
 sw   x31, 27 * 4(sp)
 csrr  x5, mstatus
 sw    x5, 28 * 4(sp)
 csrr  x5, mepc
 sw    x5, 29 * 4(sp)
 sw    sp, currentTCB→ctxAddr

 [...] # Check for exception
 [...] # Update delayed tasks
 [...] # Scheduling
 [...] # Update currentTCB

 lw    sp, currentTCB→ctxAddr
 lw    x5, 29 * 4(sp)
 csrw  mepc, x5
 lw    x5, 28 * 4(sp)
 csrw  mstatus, x5
 lw   x31, 27 * 4(sp)
 [...]
 lw    x5,  1 * 4(sp)
 lw    x1,  0 * 4(sp)
 addi  sp, sp, 30 * 4
 mret

[...] # Next task
```

```
(S) configuration:

[...] # Current task

ISR:
 [...] # Check for exception
 [...] # Update delayed tasks
 [...] # Scheduling → ctxId
 SET_CONTEXT_ID ctxId
 [...] # Update currentTCB
 SWITCH_RF

 lw    sp, currentTCB→ctxAddr
 lw    x5, 30 * 4(sp)
 csrw  mepc, x5
 lw    x5, 29 * 4(sp)
 csrw  mstatus, x5
 lw   x31, 28 * 4(sp)
 [...]
 lw    x5,  2 * 4(sp)
 lw    x1,  1 * 4(sp)
 lw    sp,  0 * 4(sp)
 mret

[...] # Next task
```

```
(SL) configuration:

[...] # Current task

ISR:
 [...] # Check for exception
 [...] # Update delayed tasks
 [...] # Scheduling → ctxId
 SET_CONTEXT_ID ctxId
 [...] # Update currentTCB
 mret

[...] # Next task
```

```
(SLT) configuration:

[...] # Current task

ISR:
 [...] # Check for exception
 GET_HW_SCHED
 [...] # Update currentTCB
 mret

[...] # Next task
```

```
x2 = sp
x3 = gp (static, not saved)
x4 = tp (static, not saved)
```

save context
restore context
restore context
ⓐ
ⓑ
ⓒ
ⓓ
ⓔ
ⓕ
ⓖ
🟩 Alternate ISR register file is in use
Decoupled HW context saving
Decoupled HW context restoring

Figure 4. ISR execution in software and hardware across different RTOSUnit configurations. As more features are offloaded, the software ISR shortens and hardware parallelism increases. With full acceleration, the ISR only updates **currentTCB**.

one word (a context consists of 31 words), but this simplifies the address generation to just shifting the task ID to the left and prepending the offset of the context memory region.

In addition to the logic required for actually storing the context, the RTOSUnit must be informed *where* to store the context. To this end, we introduce a custom instruction, **SET_CONTEXT_ID**, which is issued *after* the software scheduler selects the next context. The instruction passes the selected tasks ID to the RTOSUnit, which latches the ID and uses it during the subsequent context switch to generate the store address by indexing into the context memory region. Alternatively, the full context address could be passed to achieve the same effect. However, because our later scheduling configurations operate exclusively on IDs, we use IDs here for consistency.

In theory, context storing completes in 31 cycles, plus any stalls caused by prioritized processor memory accesses. In practice, this latency is hidden by the *significantly* longer time required to execute the scheduler concurrently.

After scheduling, the context of the selected task must be restored. Due to our sparse MUX structure, the processor must actively switch back to the application register file using the **SWITCH_RF** custom instruction. This switch-back gets delayed while storing is in progress. While the (**S**) configuration never actually required such stalling in our testing, we have implemented it to ensure correctness and support advanced configurations which are introduced later.

Integration-wise, (**S**) requires modifications to the instruction decoding of the processor, such that the added custom instructions are known and reported to the RTOSUnit. In addition to custom instructions, interrupt entries must be traced as well. Furthermore, the register file must be duplicated, (sparsely) multiplexed, and connected to the RTOSUnit. Lastly, the RTOSUnit must be connected to the CSRs of the processor.

4.3 Context Loading (SL)

Context loading is implemented similarly to context storing by adding a restore FSM (cf. Figure 3 (f)). Once the custom instruction **SET_CONTEXT_ID** announces the next task ID, the RTOSUnit triggers its restore FSM to load the new context concurrently (cf. Figure 4 (f)). This FSM performs the inverse of the store FSM: It loads the context from memory and writes it into the application register file and CSRs.

In this (**SL**) mode, the instruction **SWITCH_RF** for switching back to the application register file is no longer required. Instead, the register file bank switch is handled *automatically* during the **mret** instruction, which ends the ISR and resumes execution at the saved address in **mepc**. The **mret** instruction is stalled until the context restore completes, preventing a premature return to task execution before restoration has finished.

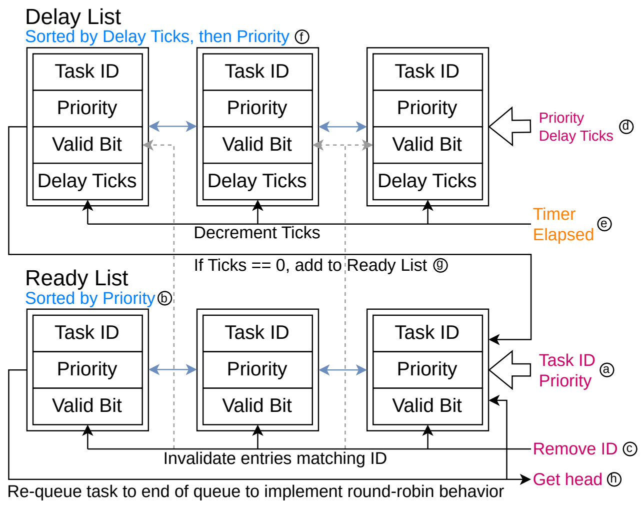


Figure 5. Hardware scheduler architecture. Magenta indicates events triggered by custom instructions; orange marks external events. The hardware scheduler implements the FreeRTOS ready and delay lists, maintaining correct order via iterative sorting. Timer interrupts decrement delay counters, moving tasks to the ready list when reaching zero.

Integration-wise, (`L`) only works in *conjunction* with (`S`), and additionally requires `mret` tracing and stalling signals to be added to the processor.

4.4 Task Scheduling (`T`)

Hardware-based task scheduling can be used independently or alongside context storing and loading. In (`T`) mode, FreeRTOSes ready and delay lists are moved to *hardware*, while the event list remains in *software*, as the RTOSUnit does not manage synchronization primitives. Three custom instructions are added: (1) `ADD_READY` to add a task to the ready list, (2) `ADD_DELAY` to add the running task to the delay list, and (3) `RM_TASK` to remove a task from either hardware-maintained list. Figure 5 illustrates the hardware scheduler structure. Adding a task to the ready list requires its ID and priority (a). The list is iteratively sorted in hardware by priority, preserving the order among tasks with identical priorities (b). Entries marked as invalid are sorted toward the tail of the list. Our prototype uses bubble sort as the sorting algorithm because it has low area requirements and a sufficient time elapses between inserting a task and querying the head of the list. For larger numbers of tasks in the system (and a correspondingly longer ready list), faster algorithms may be necessary to avoid stalls.

Task removal is achieved by matching the task ID provided by the `RM_TASK` instruction and clearing the *valid* bit of all matching entries (c). Since only the currently running task can be delayed, `ADD_DELAY` requires only the task's priority and delay duration as parameters (d).

On timer interrupts (detected via `mcause`), all delay counters are decremented (e). The delay list is sorted by remaining delay time, with ties broken by priority (f). When a task's delay expires, it is moved to the ready list automatically (g).

In (`T`) mode, the custom instruction `GET_HW_SCHED` replaces `SET_CONTEXT_ID` and returns the head of the ready list while moving that entry to the tail of the ready list (h). To further reduce the ISR execution time, we modify the RISC-V hardware timer – used exclusively for preemptive scheduling interrupts – to auto-reset, eliminating the need for software-based counter reads and compare register updates. Thus, given the configuration (`SLT`), the ISR reduces to merely updating `currentTCB` (cf. Figure 4 (g)). Since our hardware scheduling uses task IDs rather than TCB pointers, we include a software lookup table to map each task ID to its corresponding TCB, enabling the required update of `currentTCB`.

Processor modifications for (`T`) are minimal: support for custom instructions, interrupt source tracing, and auto-resetting of the hardware timer. Note that the maximum number of tasks is fixed at design time by the hardware queue size. However, once this threshold is exceeded, the system can fall back to software scheduling at the cost of forfeiting the benefits of hardware scheduling.

The presented features allow for the following configuration permutations: (`S`), (`SL`), (`T`), (`ST`), (`SLT`). We additionally support *optional* features for further mean-case latency reduction, though at the expense of larger jitter. These options are *dirty bits* (`D`), *load omission* (`O`), and *preloading* (`P`).

4.5 Dirty Bits (`D`)

Dirty bits accelerate context storing by avoiding unnecessary memory writes when tasks modify only a subset of registers. To exploit this, each register in the application register file is augmented with a *dirty bit*, which is set upon write. During context storing, the RTOSUnit saves only the registers marked as dirty. After ISR completion, all dirty bits are cleared. Integrating the (`D`) option requires a register write tracing interface within the processor.

4.6 Load Omission (`O`)

Load omission skips context loading if the *previous* and *next* selected tasks are identical. In this case, the application register file already contains the correct context, eliminating the need to reload it from memory.

4.7 Preloading (`P`)

When context storing, loading, and scheduling are all enabled, the ISR reduces to updating `currentTCB` based on the next scheduled task ID. In this case, the context-switch latency is bounded by the memory bandwidth.

Since hardware scheduling exposes the head of the ready list – the task with the highest priority – it can be accessed even outside the ISR. This enables an optional context *preloading* mechanism that *speculatively* loads the context of the task at the head of the ready list, which is likely to run next. A dedicated 31-word buffer is used to hold this preloaded context. Because the RTOSUnit accesses memory at a lower arbitration priority than the processor, preloading does *not* interfere with ongoing computation, as long as no cache is used, or, if one is, the RTOSUnit can bypass it.

Preloading operates in lockstep with context storing: as each register is saved to memory, it is immediately overwritten with the corresponding preloaded value. Due to this coupling, preloading is *incompatible* with the dirty-bit option.

Since tasks may enter the ready list immediately following an interrupt, for instance, when a timer interrupt unblocks delayed tasks, the speculatively preloaded context may not match the *actual* next task. In such cases, the correct context must be loaded from memory, resulting in the same latency as if preloading had not been used in the first place.

If the preloaded context is correct, context loading can be skipped entirely, significantly reducing context-switch latency.

5 RTOSUnit Integration

While the main functionality resides within the RTOSUnit, integrating it with a processor requires several architectural modifications. Specifically, the register file must be replicated and multiplexed (cf. Figure 3 (d)), custom instructions must be integrated (cf. Figure 3 (e)), interrupt and **mret** signaling must be managed, CSR access must be handled, and memory arbitration must be implemented (cf. Figure 3 (g)).

We identified three micro-architectural features and two integration decisions that significantly affect integration complexity: instruction reordering, speculative execution, register renaming, configuration of the RTOSUnit, and memory arbitration at the bus or *Load/Store Unit (LSU)* level. All proposed custom instructions (cf. Table 1) update the RTOSUnit's internal state and therefore must execute *in-order* and *non-speculatively*. More complex cores commonly reorder and speculatively issue instructions to improve instruction-level parallelism. Additionally, they use register renaming to eliminate false dependencies (i.e., WAW and WAR hazards), which may complicate register file replication and RTOSUnit interfacing.

Among all custom instructions, **SWITCH_RF** is unique in that it alters the processor's microarchitectural state by switching to the application register file. Since this change affects subsequent instruction execution, it requires hazard-handling logic: all *prior* instructions must commit to the ISR RF, while all *following* instructions must utilize the APP RF. Depending on the pipeline depth, both prior and following instructions may be in the pipeline simultaneously. If *both* storing and loading are enabled, register banks are switched automatically upon interrupt entry and exit (using the **mret** instruction). These events act as synchronization points (e.g., pipeline flushes), allowing the switch to occur without additional control logic. Additionally, **SWITCH_RF** must be delayed while context storing is in progress to prevent premature RF switching and the resulting overwriting of data that has not yet been stored. We observed the need for such stalling logic only in the (**ST**) and (**SDT**) configurations. In the remaining configurations that use **SWITCH_RF**, software scheduling introduces sufficient delay for context storing to complete. Nevertheless, we implement this stalling mechanism in all configurations employing **SWITCH_RF** to ensure correctness.

The location of the memory arbitration in the pipeline also affects integration complexity, particularly in the presence of a cache. If arbitration occurs within the *LSU*, the RTOSUnit can share the processor's cache, potentially reducing context-switch latency for cached contexts, especially with high-latency memory. However, this may also increase *worst-case latency* and *jitter* due to additional cache miss penalties. In turn, arbitration at the bus-level leaves more request cycles available for the RTOSUnit, since some of the processor's requests now get handled by the cache, and potentially reduces *jitter* by eliminating cache hit/miss variability. However, bus-level arbitration increases the *mean latency*.

The remainder of this section discusses the integration of the RTOSUnit into three representative RISC-V microarchitectures of increasing complexity.

5.1 CV32E40P / Pipelined In-Order Processor

Among the evaluated cores, CV32E40P has the simplest architecture. It features a 4-stage pipeline that executes instructions strictly in order, with no instruction reordering or register renaming. As it lacks a cache, the LSU directly interfaces with the memory bus, and arbitration requires only simple multiplexers on the outgoing memory signals.

Although CV32E40P performs speculative instruction fetches, branches are resolved early, and mispredicted instructions are never executed. Thus, no additional handling for speculation is required. In configurations using the **SWITCH_RF** instruction, no additional hazard logic is needed due to the shallow pipeline and lack of speculation. The switch can safely occur during operand reading.

5.2 CVA6 / Out-of-Order WB

Like CV32E40P, CVA6 issues instructions in-order and employs only limited speculation, enabling the RTOSUnit to be integrated as a standard functional unit. However, unlike CV32E40P, CVA6 incorporates register renaming to enable *out-of-order* write-back. This renaming is handled inside the scoreboard module, leaving the register file to contain committed values only. Thus, the RTOSUnit can access the architectural state directly, making register file replication and multiplexing as straightforward as in CV32E40P.

Configurations using the **SWITCH_RF** instruction require extra logic to avoid hazards, as instructions prior to **SWITCH_RF** must write to the ISR RF while those following **SWITCH_RF** already read from the APP RF, with both potentially coexisting in the pipeline during a switch.

Despite CVA6 using a write-through cache, we chose to arbitrate the memory at the *bus level* to reduce jitter.

5.3 NaxRiscv / Full Out-of-Order Execution

While integrations with CV32E40P and CVA6 were straightforward, adapting the RTOSUnit to the more complex NaxRiscv core required additional considerations, due to its fully *out-of-order execution* model, aggressive *speculative execution*, and *register renaming*. As we will show, our RTOSUnit can also be integrated with such a complex base core.

Unlike CVA6, which only reorders *write-back*, NaxRiscv *issues* and *executes* instructions out-of-order and speculatively. Since custom instructions irrevocably affect RTOSUnit state, they must execute strictly *in program order* and *non-speculatively*. To this end, we introduced a *queue* that buffers custom instructions (cf. Figure 6) while their final speculation status (correctly speculated or misspeculated) is not yet known. We preserve the program order by inserting the custom instructions into this queue during the in-order decode stage (a). Once issued, the resolved argument values (RS1, RS2) are added alongside the buffered instructions (b). If the instruction turns out to be on a misspeculated path, the queue is flushed (c). Otherwise, when the instruction reaches the commit stage (d), it is passed to the RTOSUnit (e) for execution, preserving program order and correctness. We found a short queue size of just two entries to be sufficient to prevent pipeline stalling due to a full queue. Alternatively, one could stall the decode stage to ensure correct execution and ordering of our custom instructions. However, this would also block all subsequent *normal* instructions, whereas the addition of the queue allows for out-of-order execution without stalling the pipeline.

Register renaming, as explained in Figure 7, enables NaxRiscv's instruction reordering. In contrast to CVA6, renaming is not concealed within a scoreboard. Instead, NaxRiscv is equipped with a *larger* physical register file capable of storing more than the 31 writable architectural registers (a). In this fashion, the register file is employed to store both the *current* architectural state and as-yet *uncommitted* values. In order to differentiate between the current state and uncommitted results, as well as to translate an architectural register to the correct physical register, NaxRiscv contains additional logic (b) to keep track of the mapping between the architectural and physical registers.

As explained in Subsection 4.2, for a context-switch, 29 architectural registers must be saved and restored from memory. Given the RTOSUnit's micro-architecture independence, it uses architectural register addresses (c), which must be *translated* into physical addresses before accessing the physical register file. To this end, the existing translation logic is tapped into, and the translated address is forwarded to the register file.

To provide a dedicated register file for the ISR, the physical register file and translation logic were duplicated (c). Notably, NaxRiscv allocates physical registers only upon *first write*. As a result, the alternate register file can be smaller than the full 32 registers, depending on actual usage.

We found that as few as *eleven* registers were sufficient to execute our FreeRTOS benchmarks. However, such a reduction in the register file size inherently imposes constraints on the code that can be executed during an ISR. For maximum generality, we thus chose a full-size (32 registers) alternate register file for ISR use that still allows for *arbitrary* code execution in the ISR.

Finally, the RTOSUnit's memory interface must be arbitrated with the processor's memory requests. Due to the large and complex architecture of NaxRiscv, it is more likely to be connected to high-latency memories, such as external DRAM. Therefore, we extended the core's LSU to allow the RTOSUnit to *share* the processor's write-back cache (cf. Figure 8). The existing LSU includes speculative load (a) and store (b) queues, which are flushed on misprediction. Given the non-speculative nature of the RTOSUnit's requests, a dedicated queue, designated as **ctxQueue** (c), was added here. This approach avoids alterations in the core's flushing logic and significantly simplifies the prioritization of non-RTOSUnit memory requests (d).

The **ctxQueue** combines the functionalities of the load and store queues. It supports *out-of-order* memory access and uses a simple tag to indicate load or store (e). Each entry contains address and data fields (f). Since the RTOSUnit expects the read responses in the same order as the requests, we use the data field to latch the cache's result until it can be forwarded to the RTOSUnit. To balance performance and area, we evaluated different queue sizes and identified *eight* entries as a Pareto-optimal solution. Further reducing the queue size would negatively impact context-switch latency, while larger sizes offer no performance gain.

Entires in the **ctxQueue** are allocated and freed in-order, and the RTOSUnit issues load and store addresses in identical order. Given this behavior, a load and store (or two stores) to the same address could coexist in the queue only for a queue depth of at least 32 entries. With our chosen depth of eight, such aliasing cannot occur. Hence, store-to-load forwarding and handling of aliasing stores within the **ctxQueue** are unnecessary, ensuring safe out-of-order operation. These assumptions may be violated when dirty bits are used, in which case context loading is stalled until the **ctxQueue** is empty.

The design assumes *exclusive access* to the context memory region by either the core or the RTOSUnit at any given time. In consequence, *no* snooping or store-to-load forwarding needs to be implemented between the **ctxQueue** and NaxRiscv's existing queues. When both context loading and

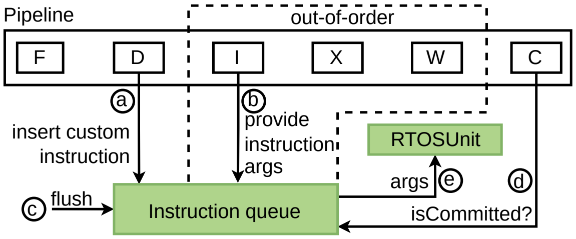


Figure 6. Integration of RTOSUnits' custom instructions into a full out-of-order core. The instruction queue is used to collect out-of-order issued custom instructions (a)/(b) and passes them in-order to the RTOSUnit (e).

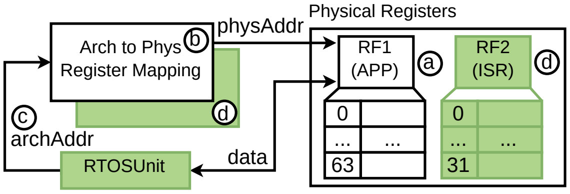


Figure 7. Overview of NaxRiscv's renaming architecture. The 64-entry physical register file (a) holds committed and in-flight values. The register-mapping (b) translates architectural to physical registers (e.g., for the RTOSUnit (c)) and disambiguates between committed and in-flight values. Green shaded boxes indicate added logic for the ISR register file and RTOSUnit integration.

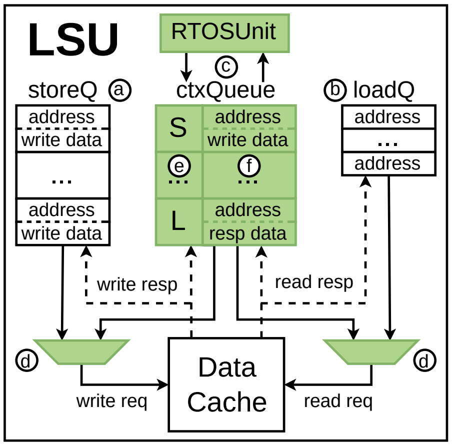


Figure 8. Overview of the required changes to NaxRiscv's LSU to arbitrate memory requests from the RTOSUnit. Green blocks show added logic. The **`ctxQueue`** (c) handles memory requests from the RTOSUnit, enabling out-of-order execution. Multiplexers (d) arbitrate between core and RTOSUnit, prioritizing the core's **`loadQ`** and **`storeQ`**.

storing are performed in hardware, only the RTOSUnit exclusively accesses the reserved context memory. If only context storing is enabled, the core loads contexts in software but never writes to the context memory region. If the scheduler immediately switches back to the suspended task, the **`SWITCH_RF`** instruction waits for all pending stores in the **`ctxQueue`** to complete, ensuring that context saving finishes *before* the register file is switched and the same context is reloaded in software. After the **`SWITCH_RF`** instruction has executed, a *reschedule event* is triggered, similar to a misprediction, to re-execute the instructions following **`SWITCH_RF`** on the switched register file.

6 Evaluation

We assess RTOSUnit's performance, silicon area, and power overheads as 22nm ASICs using the letter-based configuration naming scheme introduced earlier: Store (**`S`**), Load (**`L`**), Task Scheduling (**`T`**), Load Omission (**`O`**), Dirty Bits (**`D`**), and Preloading (**`P`**). As before, (**`vanilla`**) refers to an unmodified processor. Additionally, we provide a direct comparison with related work (**`CV32RT`**) [3]. To this end, we have re-implemented their CV32E40P-only solution for CVA6 and NaxRiscv. In all three (**`CV32RT`**) variants, a dedicated memory port is used to save 16 of the 31 words of a context. For NaxRiscv, the dedicated port bypasses the write-back data cache. To prevent cache incoherence, the single cache line containing the bypassed 16 words is explicitly invalidated. The remaining context is saved in software and thus normally cached. We also tested flushing the cache and marking the context region as uncacheable; however, both approaches resulted in significantly worse latency compared to the software-only (**`vanilla`**) solution. In contrast, all other configurations of NaxRiscv arbitrate *within* the LSU, as described in Subsection 5.3, and do *not* require any cache invalidation, while allowing the entire context to be cached. All hardware list lengths are set to eight entries unless explicitly mentioned otherwise.

6.1 Context-Switch Latency

We evaluate context-switch latency in RTL simulation using 20 iterations of all tests provided by the RISC-V port of *RTOS-Bench* [4]. The ready and delay lists are configured with eight entries each. The context load omission optimization (**`O`**), described in Subsection 4.6, is enabled *only* for configurations that use dirty bits (**`D`**) or preloading (**`P`**), as these techniques already trade lower average latency for increased variability, making them unsuitable for real-time applications that need tighter time bounds. All cores are assumed to use tightly coupled, single-cycle latency on-chip SRAM. Latency is measured from interrupt trigger to the execution of the **`mret`** instruction, and jitter is reported as the difference between maximum and minimum latencies. We report latency in cycles rather than time at f_{max}, as embedded systems typically operate at much lower frequencies to optimize for area and power consumption. The results are aggregated in Figure 9.

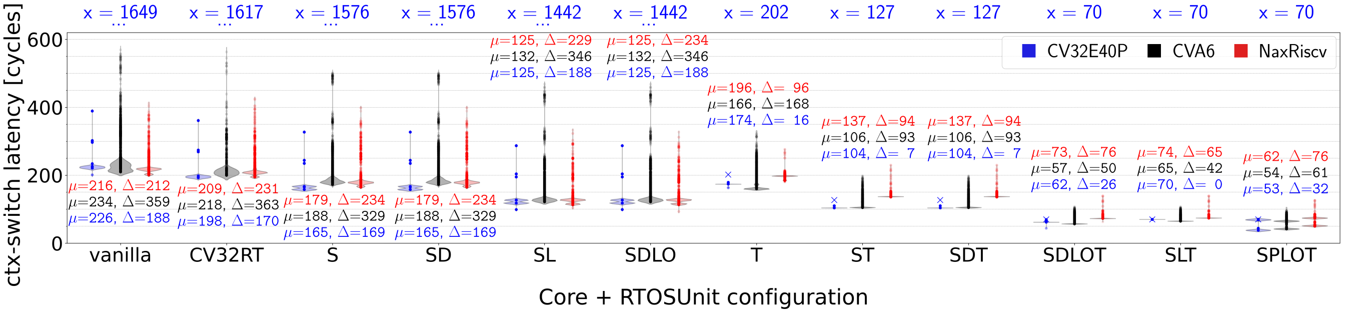


Figure 9. Context-switch latencies with different processors and configurations. Store (**S**), Load (**L**), Task Scheduling (**T**), Load Omission (**O**), Dirty Bits (**D**), and Preloading (**P**). (**CV32RT**) corresponds to the approach by Balas et al. [3]. Δ is the jitter, as the difference between the measured worst- and best-case switching latency. μ represents the average switching latency. Note that some WCET values, marked with an x, greatly *exceed* the y-axis limit.

Compared to software context switching (**vanilla**), the approach of Balas et al. (**CV32RT**) achieves only modest average latency reductions, ranging from 3 % to 12 %. In contrast, our simplest configuration (**S**) yields substantially larger improvements, between 17 % and 27 %. This improvement is likely attributable to our ability to overlap the *entire* context saving with code execution, whereas the approach of Balas et al. accelerates only *half* of the context saving in hardware. In both cases, the jitter remains comparable.

Across all cores, offloading context storing and loading to hardware (**ST**) *reduces* mean latency while *maintaining* similar jitter. This offloading removes overhead from the processor, while the inherently variable-latency scheduler remains in *software*. Offloading scheduling alone to hardware (**T**) *significantly* reduces jitter, especially on CV32E40P, dropping from 188 to just 16 cycles, a reduction of more than 90 %. Average latency is reduced by 23 %, 29 %, and 9 % for CV32E40P, CVA6, and NaxRiscv, respectively. Moving both context-switching and scheduling (**SLT**) to hardware minimizes both latency and jitter. On CV32E40P, jitter is eliminated *entirely*, while jitter on CVA6 and NaxRiscv is reduced by up to 88 %. The remaining jitter is likely due to microarchitectural features like caches and speculative execution, which cannot be directly influenced by our approach. In the (**SLT**) configuration, context storing and restoring become the primary bottlenecks, as the remaining ISR operations require *fewer* than 62 cycles, the minimum to fully hide context handling.

Enabling dirty bits and load omission (**SDLOT**) yields a modest reduction in mean latency at the expected cost of increased jitter. For CV32E40P, some cases show latency below 50 cycles. Omitting hardware scheduling in (**SDLO**), dirty bits and load omission show no improvement over (**SL**), suggesting that switching time is more influenced by software *scheduling* than *memory bandwidth*. Jitter remains largely unchanged, except for a minor deviation on NaxRiscv, likely due to altered memory access patterns due to the dirty bit optimization.

The fastest average switching latency is achieved using our *preloading* strategy (**SPLOT**). Selecting the head of the *ready* list proves to be a simple yet effective heuristic for preloading the next context. Results fall into two clusters of similar size: one with higher latency due to *mispredicted* preloads, and another with lower latency from correctly predicted contexts, where latency is reduced by up to 31 cycles compared to (**SLT**). For mispredicted preloads, (**SPLOT**) performs comparably to (**SLT**).

6.2 Worst-Case Timing Guarantees

Due to our decoupled context-switch acceleration, the WCET of the ISR is the maximum latency required either by the RTOSUnit or the remaining software in the ISR. To calculate the worst-case software latency, we analyze the longest instruction path, assuming maximum latency for every instruction and accounting for pipeline flushes and stalls due to dependencies. For software-scheduled setups, we assume that eight tasks are delayed and must be moved to the ready lists by the ISR. Note that larger task counts would yield even higher worst-case latencies. For RTOSUnit FSM latency, we analyze both the hardware and ISR code, considering stalls from processor memory accesses.

Figure 9 shows the worst-case context-switch latencies for CV32E40P. Our approach significantly reduces WCET: for instance, the baseline shows a WCET of 1649 cycles, while offloading context-switching (**SL**) or task scheduling (**T**) reduces it to 1442 cycles and 202 cycles, respectively. When offloading both to hardware (**SLT**), the WCET is reduced to 70 cycles, matching the measured context-switch latency.

Due to the complexity of CVA6 and NaxRiscv, WCET analysis for these cores is considered out of scope. Their latencies are highly dependent on factors such as cache configurations and memory access patterns, which makes WCET predictions potentially overly pessimistic. WCET analysis for out-of-order cores remains an ongoing research topic. Li et al. [13] propose a method for such an analysis, which requires very precise modeling of the core's pipeline.

6.3 ASIC

We assess the area and maximum frequency of our designs using commercial ASIC tools (EDA), targeting a 22 nm technology node, and perform the implementation down to chip layout for accurate power analysis. To ensure stable area measurements, the target frequency is fixed at or above the maximum frequency achievable by each *unmodified* base core. Timing overheads for our RTOSUnit are reflected as *negative worst-case setup slack*, as reported by the EDA tools. These slowdowns are reported as $f_{\max}$ drops. For NaxRiscv, we manually instantiate optimized process-specific SRAM macros to support caches and branch prediction tables efficiently. However, their area is *excluded* from the reported values to ensure a fair comparison, as the other cores are evaluated *without* caches. Figure 10 shows the area of each core alongside all supported RTOSUnit configurations, while Figure 11 presents the corresponding maximum frequencies achieved. Although these GHz frequencies *are* technically achievable, frequencies in the hundreds-of-megahertz range are more common in embedded systems, given their smaller area and power / thermal envelopes.

Among the evaluated designs, CV32E40P is the smallest core and thus most sensitive to area changes. The (**S**) configuration increases area by 21.9 %. This is only slightly larger than the 21.2 % area increase observed for (**CV32RT**), while (**S**) improves the average latency by 27 %, compared to (**CV32RT**)'s 12 %. Introducing hardware scheduling (**T**) adds no observable area overhead compared to (**vanilla**). However, combining scheduling with context-storing (**ST**) results in a 33 % area increase. Adding context-loading functionality (**SLT**) introduces negligible additional overhead, while the integration of dirty bits falls within the typical range of "heuristics noise" of the algorithms in the EDA tools. The (**SPLOT**) configuration, which implements context preloading via dedicated registers, increases area by approximately 44 %. Across all configurations, except for (**CV32RT**), CV32E40P experiences a frequency drop of roughly 15 %, yet remains *well above* typical operating frequencies for small embedded systems, its primary target domain.

CVA6 and NaxRiscv, being larger cores, exhibit similar but less pronounced trends. For CVA6, (**S**) results in a 3-5 % area increase. For NaxRiscv, the overhead reaches up to 15 %, likely due to the duplication of register renaming logic in addition to the alternate register file. (**CV32RT**) shows an overhead of 2 % for CVA6. Overhead remains lower than for CV32E40P, as routing congestion proves less critical during synthesis. In contrast, NaxRiscv exhibits an area increase of 19 %, which exceeds that of the most advanced configuration (**SPLOT**). This higher overhead stems from NaxRiscv's use of *register renaming*: snapshotting cannot just rely on static addresses to access the register file and therefore requires the costly addition of 16 extra read ports. As with CV32E40P, enabling the (**T**) configuration does not introduce measurable area overhead. In CVA6 and NaxRiscv, further acceleration of scheduling and context switching increases area by up to 8 % and 13 % respectively, rising to 14 % and 15 % when preloading is also enabled.

The added pipeline hazard handling logic required for the **SWITCH_RF** custom instruction in (**S**), (**SD**), (**ST**), and (**SDT**) on CVA6 leads to area increases beyond the corresponding configurations with hardware context loading support ((**SL**), (**SDLO**)), (**SLT**), and (**SDLOT**)). NaxRiscv exhibits the opposite behavior, where omitting hardware context loading results in reduced area overhead, as those hazards are instead handled through pipeline rescheduling.

CVA6 experiences a slight frequency drop of around 8 % across configurations, while NaxRiscv *maintains* stable frequency throughout, with the exception of the (**SPLOT**) configuration, which exhibits a modest 4 % drop.

We also evaluate how area overhead *scales* with increasing lengths of the ready and delay lists. We synthesize the CV32E40P core with hardware scheduling only (**T**), varying the number of slots in both lists while keeping them equal in length. Figure 12 presents the absolute area across different list sizes. A list length of zero denotes the unmodified core. While minor fluctuations appear, especially at smaller list sizes, most likely due to "heuristics noise", the area increases approximately linearly, reaching a 14 % overhead at 64 slots.

Lastly, we evaluate the *power impact* of our RTOSUnit. We perform highly accurate *gate-level simulations* on the actual chip layouts for the **mutex_workload** test on all cores running at 500 MHz. This approach is chosen because it includes the precise signal transitions of an *actual* workload execution, and not just estimated average toggle rates, and thus reflects a realistic operating scenario. Using the resulting logic waveforms and implemented ASIC designs, we derive the average power consumption using industry-standard commercial power analysis EDA tools. The results, shown in Figure 13, reflect *average power draw* over the *full workload duration*, rather than isolating the individual context-switching phases. This approach is motivated by the fact that total power consumption across the processor's operation, not just during context switches, influences implementation decisions in low-power designs. For NaxRiscv, we *excluded* the caches' SRAM power draw from reports to ensure a fair comparison.

As expected for the 22 nm technology node (and below!), we observe a strong correlation between area and power draw, primarily due to the impact of *static* power draw. For CV32E40P and CVA6, (**CV32RT**) exhibits the lowest additional power consumption. In the case of CVA6, the power consumption of the (**S**) configuration is nearly identical to that of (**CV32RT**), while providing a substantial improvement

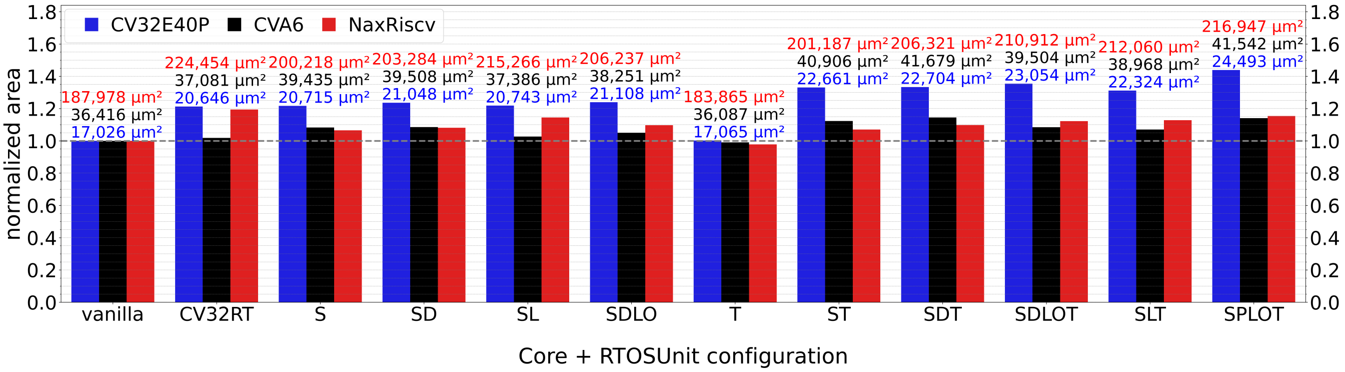


Figure 10. Normalized ASIC area w.r.t. the baseline of the processors under different RTOSUnit configurations. Store (**S**), Load (**L**), Task Scheduling (**T**), Load Omission (**O**), Dirty Bits (**D**), and Preloading (**P**). (**CV32RT**) corresponds to the approach by Balas et al. [3]. The absolute area is reported above the bars.

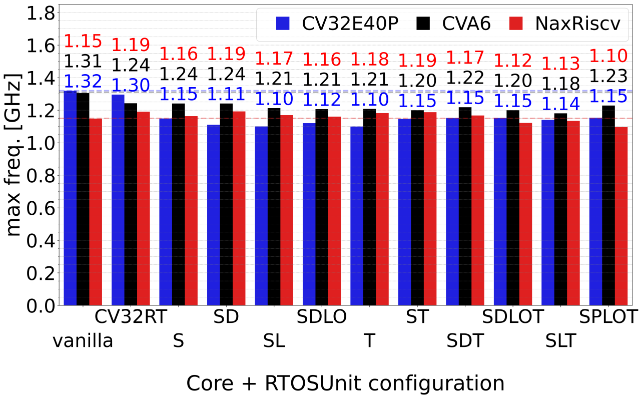


Figure 11. ASIC f_{max} of the base processors under different configurations.

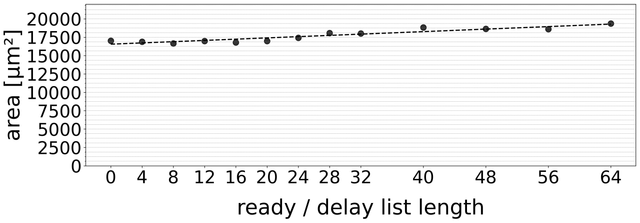


Figure 12. ASIC area scaling with increasing scheduler list length.

in context-switch latency. For NaxRiscv, the scheduling-only configuration (**T**) incurs the lowest additional power consumption, with less than 2 mW. With the exception of NaxRiscv, the highest power draw is observed for the (**SPLOT**) configuration. For NaxRiscv, however, the (**CV32RT**) configuration exhibits the highest power consumption, as it also results in the largest area overhead. While CV32E40P's *relative* power increases by up to 72 %, the *absolute* increases remain small. CVA6 shows a similar trend, with a relative

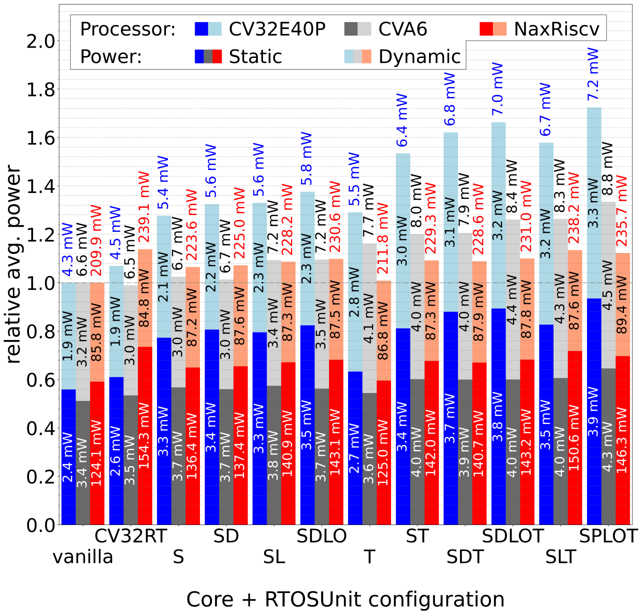


Figure 13. Power estimation under different RTOSUnit configurations.

power increase of up to 33 %. Due to NaxRiscv's higher baseline power usage, the *relative* increase is more modest, up to 13 % (excluding (**CV32RT**)).

6.4 Configuration Selection

The choice of configuration depends on the relative importance of latency, jitter, and area overhead in the target application. No single configuration dominates across all metrics; instead, each represents a different point in the design space.

The (**SLT**) configuration emerges as a strong all-round solution. It provides substantial improvements in average latency and jitter while maintaining predictable behavior across workloads. However, these benefits come at a noticeable area cost, which may limit their applicability in highly area (cost)-constrained systems. When average context-switch latency is the primary optimization goal and area overhead is only of secondary concern, the (**SPLOT**) configuration is the preferred choice. By enabling context preloading, it achieves the lowest average latency among all evaluated designs, albeit with the highest area and power overheads.

For systems where silicon area is the dominant constraint, the scheduling-only configuration (**T**) represents an attractive option. It introduces virtually no area overhead while still providing meaningful reductions in jitter and reasonable improvements in latency. This makes (**T**) particularly well suited for cost-sensitive embedded designs.

The (**SL**) configuration occupies an intermediate position between (**T**) and (**SLT**). Compared to (**T**), it offers improved average latency at the cost of a moderate area increase, while jitter remains largely unchanged. Relative to (**SLT**), it reduces area overhead but also sacrifices some of the benefits in jitter reduction, making it suitable when latency improvements are desired without fully committing to the cost of the more comprehensive solution.

Overall, these configurations allow designers to trade off area, latency, and jitter in a controlled manner, enabling adaptation to a wide range of embedded real-time use cases.

7 Conclusion and Future Work

We introduced the RTOSUnit, a novel hardware acceleration unit for tightly-coupled context-switching acceleration in real-time systems. Our approach is portable, as demonstrated by integration into three RISC-V processors of varying complexity. By its configurable architecture, the RTOSUnit enables *fine-grained* trade-offs between area, mean context-switch latency, and jitter, making it easily adaptable for different application requirements.

Our most advanced hard real-time-focused configuration (**SLT**), which includes full context-switching and scheduling acceleration, eliminates jitter *entirely* on some processors and achieves a *mean* latency reduction of up to 69 %, with an area overhead of 31 %. Meanwhile, our most *area-efficient* variant (**T**) has costs within EDA tool "heuristics noise", while reducing jitter by 91 % and lowering mean latency by 23 %. Across *all* configurations, operational frequencies remain within practical bounds for embedded deployments.

While our RTOSUnit provides significant improvements for *single-core* FreeRTOS systems, its current design is limited to uniprocessor configurations. Future work could explore support for multi-core systems, extending RTOSUnit's applicability to multiprocessing scenarios.

Additionally, hardware acceleration of common synchronization primitives, such as the semaphores or mutexes examined in prior work, could further offload the processor and reduce overhead in coordination-intensive workloads. Similarly, context storing and loading could be extended to support floating-point or vector registers.

Finally, although similar in architecture, supporting additional real-time operating systems such as Zephyr [28] or ChibiOS [21] would further broaden the scope and adoption potential of the design in embedded environments.

Acknowledgments

We thank our reviewers for their valuable feedback. We acknowledge the financial support by the German Federal Ministry of Research, Technology and Space in the project “Scale4Edge” (grant: 16ME0139).

AI tools were used for language refinement.

References

[1] David Andrews, Wesley Peck, Jason Agron, Keith Preston, Ed Komp, Mike Finley, and Ron Sass. 2005. hthreads: A hardware/software co-designed multithreaded RTOS kernel. In *2005 IEEE Conference on Emerging Technologies and Factory Automation*, Vol. 2. IEEE, 8–pp.

[2] Arm Ltd. 2019. Registers in AArch32 state. https://developer.arm.com/documentation/100076/0200/instruction-set-overview/overview-of-aarch32-state/registers-in-aarch32-state. Accessed: 2025-08-01.

[3] 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* 32, 6 (2024), 1032–1044. doi:10.1109/TVLSI.2024.3377130

[4] Brannium. 2024. RTOSBench. https://github.com/Brannium/RTOSBench. Accessed: 2025-03-26.

[5] Sathish Chandra, Francesco Regazzoni, and Marcello Lajolo. 2006. Hardware/software partitioning of operating systems: A behavioral synthesis approach. In *Proceedings of the 16th ACM Great Lakes symposium on VLSI*. 324–329.

[6] Elham Cheshmikhani and Hamed Farbeh. 2025. Interrupt Caching: A Hardware-Assisted Interrupt Handling to Enhance System Responsiveness. *IEEE Access* (2025).

[7] Intel Corporation. 1986. *80386 Programmer's Reference Manual.* Section 7.5: Task Switching.

[8] Advanced Micro Devices. 2006. *AMD64 architecture programmer's manual volume 2: System programming.* Section 12.3: Hardware Task-Management in Legacy Mode.

[9] Eugen Dodiu 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*. IEEE, 1–5.

[10] Michael Gautschi, Pasquale Davide Schiavone, Andreas Traber, Igor Loi, Antonio Pullini, Davide Rossi, Eric Flamand, Frank Gurkaynak, and Luca Benini. 2017. Near-Threshold RISC-V Core With DSP Extensions for Scalable IoT Endpoint Devices. doi:10.1109/TVLSI.2017.2654506

[11] 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*. IEEE, 592–599.

[12] Intel Corporation. 2023. *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3A: System Programming Guide, Part 1.* Section 7.7: Task Management in 64-Bit Mode.

[13] Xianfeng Li, Abhik Roychoudhury, and Tulika Mitra. 2006. Modeling out-of-order processors for WCET analysis. *Real-Time Systems* 34, 3 (2006), 195–227.

[14] L. Lindh. 1991. Fastchart-a fast time deterministic CPU and hardware based real-time-kernel. In *Proceedings. EUROMICRO '91 Workshop on*

Real-Time Systems. 36–40. doi:10.1109/EMWRT.1991.144077

[15] Lennart Lindh and Frank Stanischewski. 1991. FASTCHART-Idea and Implementation. In *Proceedings of the 1991 IEEE International Conference on Computer Design on VLSI in Computer & Processors (ICCD '91)*. IEEE Computer Society, USA, 401–404.

[16] Microchip Technology Inc. 2015. *PIC32MZ Embedded Connectivity with Floating Point Unit (EF) Family Data Sheet*. Accessed: 2025-08-01.

[17] 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* (Nicosia, Cyprus) *(SAC '04)*. Association for Computing Machinery, New York, NY, USA, 869–875. doi:10.1145/967900.968077

[18] Takumi Nakano, Yoshiki Komatsudaira, Akichika Shiomi, and Masaharu Imai. 1999. Performance evaluation of STRON: A hardware implementation of a real-time OS. *IEICE transactions on fundamentals of electronics, communications and computer sciences* 82, 11 (1999), 2375–2382.

[19] 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).

[20] 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)*. IEEE, 1–4.

[21] Giovanni Di Sirio. 2025. ChibiOS Real-Time Operating System. https://www.chibios.org/. Accessed: 2025-05-10.

[22] Jeffrey S Snyder, David B Whalley, and Theodore P Baker. 1995. Fast context switches: Compiler and architectural support for preemptive scheduling. *Microprocessors and Microsystems* 19, 1 (1995), 35–42.

[23] SpinalHDL. 2025. NaxRiscV. https://github.com/SpinalHDL/NaxRiscv. Accessed: 2025-03-26.

[24] Frank Stanischewski. 1993. FASTCHART-Performance, Benefits and Disadvantages of the Architecture. In *Fifth Euromicro Workshop on Real-Time Systems*. IEEE Computer Society, 246–247.

[25] Melissa Vetromille, Luciano Ost, César AM Marcon, Carlos Reif, and Fabiano Hessel. 2006. RTOS scheduler implementation in hardware and software for real time applications. In *Seventeenth IEEE International Workshop on Rapid System Prototyping (RSP'06)*. IEEE, 163–168.

[26] Ionel Zagan and Vasile Gheorghiță Găitan. 2022. Designing a Custom CPU Architecture Based on Hardware RTOS and Dynamic Preemptive Scheduler. *Mathematics* 10, 15 (2022). doi:10.3390/math10152637

[27] Florian Zaruba and Luca Benini. 2019. The Cost of Application-Class Processing: Energy and Performance Analysis of a Linux-Ready 1.7-GHz 64-Bit RISC-V Core in 22-nm FDSOI Technology. doi:10.1109/TVLSI.2019.2926114

[28] Zephyr Project. 2025. Zephyr Real-Time Operating System. https://www.zephyrproject.org/. Accessed: 2025-05-10.

[29] Xiangrong Zhou and Peter Petrov. 2006. Rapid and low-cost context-switch through embedded processor customization for real-time and control applications. In *Proceedings of the 43rd annual Design Automation Conference*. 352–357.