RISC-V for Real-time MCUs - Software Optimization and Microarchitectural Gap Analysis

Robert Balas*, Luca Benini*
*Integrated Systems Laboratory, ETH Zurich, Switzerland
balasr@iis.ee.ethz.ch, lbenini@iis.ee.ethz.ch

***Abstract*—Processors using the RISC-V ISA are finding increasing real use in IoT and embedded systems in the MCU segment. However, many real-life use cases in this segment have real-time constraints. In this paper we analyze the current state of real-time support for RISC-V with respect to the ISA, available hardware and software stack focusing on the RV32IMC subset of the ISA. As a reference point, we use the CV32E40P, an open-source industrially supported RV32IMFC core and FreeRTOS, a popular open-source real-time operating system, to do a baseline characterization. We perform a series of software optimizations on the vanilla RISC-V FreeRTOS port where we also explore and make use of ISA and micro-architectural features, improving the context switch time by 25% and the interrupt latency by 33% in the average and 20% in the worst-case run on a CV32E40P when evaluated on a power control unit firmware and synthetic benchmarks. This improved version serves then in a comparison against the ARM Cortex-M series, which in turn allows us to highlight gaps and challenges to be tackled in the RISC-V ISA as well as in the hardware/software ecosystem to achieve competitive maturity.**

***Index Terms*—RISC-V, FreeRTOS, real-time, open-source**

I. Introduction

Embedded applications running on low-power microcontrollers require multiple tasks to execute concurrently, to synchronize their execution with timers, external events, and with other tasks. Although dedicated multi-threading and multi-tasking hardware is considered too expensive for embedded microcontrollers, on the software side, some abstraction is needed to ease the life of programmers in managing these interactions among the various hardware and software components when dealing with real-time applications.

Several software libraries have been provided by major IPs and MCU companies to hide the complexity of the underlying hardware and provide a standardized way to handle peripherals and multi-tasking in commercial MCUs. Among them, Real-Time Operating Systems (RTOS) allow a modular approach to the development of real-time applications by providing commonly used primitives and abstractions such as tasks, inter-task object synchronization, and inter-task data transferring, lowering time-to-market, and simplifying certification by providing a pre-certified software stack. On the downside, some fine-grained control has to be given up. Nevertheless, RTOSes prove to be the standard starting point to develop real-time applications.

RISC-V [1] is an emerging open-source ISA finding increasing use in IoT and embedded systems in the MCU segment. The value proposition of the RISC-V ISA over other available solutions is no licensing fees and royalties and freedom to add instructions. Many real-life use cases in this segment and their application domains have real-time constraints. Since RISC-V is a relatively recent ISA and RISC-V cores and under active development. RISC-V hardware and software are lacking maturity, especially for what concerns real-time support.

Among several RISC-V open-source cores, the CV32E40P [2] [3] is a 4-pipeline stages processor implementing RVC32IMFC ISA, as well as extensions for energy-efficient digital signal processing in the IoT domain. CV32E40P is maintained by the OpenHW group, an industry association developing, verifying and documenting the IP, and is considered as one of most solid implementations of RISC-V cores in the MCU domains. CV32E40P is integrated into the PULPissimo open-source platform [1], a state of the art microcontroller with support for standard MCU peripherals, low-power features, and the standardized RISC-V interrupt controller, called CLINT.

The paper analyzes the current status of RT support for RISC-V with respect to ISA, hardware and software focusing on RV32IMC (MCU profile), and using an open-source, industrially supported core, and one of the leading [4] open RT operating systems (FreeRTOS). The analysis covers baseline RT performance characterization, a number of optimizations to improve it, with careful exploration of ISA and micro-architectural features of the core. The proposed optimizations improve performance by 25% on context switching and by 33% in the average and 20% in the worst case on interrupt response. We then compare the achieved performance with what is claimed by the leading commercial ISA and MCU in the same market segment and highlights gaps and challenges to be tackled by the RISC-V ISA as well as hardware and software ecosystem achieve competitive maturity.

II. Background

A. FreeRTOS

FreeRTOS [5] is an industrially supported and maintained open-source RTOS released for the first time in 2003 and written in portable C99. A vast array of primitives such as tasks, mutexes/semaphores for object synchronization and queues/mailboxes for inter-task data transfer are supported. Resources can be either statically or dynamically allocated, depending on the application's need. The scheduler can be configured to run in a pre-emptive, priority-based mode where equal priority tasks are picked in a round-robin fashion or cooperative mode. For low power operation, a so-called tick-less mode can be engaged, which suppresses the periodic system

[1]available on `https://www.github.com/pulp-platform/`

ticks when the program is idling until the next task is to be run. Certain architectures allow a type of interrupt that is not blocked by critical sections of the RTOS (Section II-D). Official add-on libraries are available such as a TCP/UDP/IP stack, FAT filesystem support, a POSIX abstraction layer, HTTPS support, saving development time when writing IoT applications.

B. CV32E40P

The CV32E40P [6] is an open-source industrially supported RISC-V core that implements base integer instructions with hardware multiplier and compressed instruction support (RV32IMC). Additionally, the RISC-V Xpulp extension presented in [3] is available. Instruction fetches and data read/writes can happen concurrently without interference, and the memory has zero wait states. The CV32E40P, combined with the custom interrupt controller described in Section II-E, forms the hardware basis for the comparison against ARM-based solutions.

C. Interrupts in RISC-V

The Core-level Interrupt Controller (CLINT), a standard defined in the RISC-V privilege mode specification [7], is the interrupt interface within the core. It defines a fixed priority interrupt scheme, default interrupt sources, and how interrupts interact with a RISC-V core with respect to privilege modes and machine state. On demand, external interrupts can be vectorized if the hardware implements that feature, but regular exceptions always end up at the vector table base address.

The Platform-Level Interrupt Controller (PLIC) [8] is a standard, specifically defined for RISC-V, multiplexing multiple interrupt lines that can be prioritized onto the external interrupt lines of the CLINT. Handling interrupts requires intervention from the core side. Serving interrupts requires write and read operations to configuration and control registers to demultiplex the incoming interrupt and accept it. This software control loop creates an overhead on the latency required to serve the interrupts.

D. Interrupt Types

In most embedded applications, interrupts constitute a critical interface with external events. When the RTOS interacts with interrupts, we often consider two classes of interrupt service routines (ISR): ISRs that rely on RTOS specific resources that need to be protected with critical sections (RTOS ISR) and ISRs that do not make use of such things (non-RTOS ISR). The former are affected by the longest critical section in the program, where in the worst-case, your interrupt latency is lower bounded by the whole critical section length plus the hardware intrinsic interrupt latency. The latter should ideally be allowed to run during critical sections, to allow for an interrupt latency that is independent of the execution state of the RTOS.

E. Interrupt Controller

Aiming for a low interrupt latency, we do not use a PLIC. Instead, we use an interrupt interface of the CV32E40P that allows tightly coupled interrupt handling: Interrupts are processed in an external interrupt controller that directly confirms taken interrupts with a hardware-based handshake. This is essentially a CLINT that allows for interrupts to be completed in hardware.

F. Power Control Unit Firmware

We use FreeRTOS and the CV32E40P in a power control unit for HPC platforms as a practical usage scenario. This firmware is used to interface with the physical sensors and actuators in high-performance computing systems, periodically reading the process, temperature, workload of the main processing elements and setting voltage and frequency according to the power management policies implemented. The real-time requirement is given by the need to react timely to PVT variations as well as application-level power peaks. The key RT-related functionality needed in this scenario is to guarantee the power capping constraint as well as the thermal stability of the system [9]. We target optimization of these key primitives to enable more reactive control and to free up as many processor cycles as possible for the control tasks.

III. Optimizations

A. ABI

Currently, RISC-V has two Application Binary Interfaces (ABIs) specified: The regular ABI, and the embedded ABI [10] (EABI). The former is mainly meant to be used in generic situations, while the latter has a focus on embedded systems that are sensitive to long interrupt latency. Compared to the regular ABI, the EABI reduces the number of argument registers from 8 to 4, consequently reducing the number of registers needed to be saved before calling the body of the interrupt handler. We add support to FreeRTOS for the EABI.

B. RV32E

The RISC-V standard defines an optional replacement for the base integer instruction set (RV32I) for 32-bit architectures, called RV32E, explicitly designed for embedded systems. The only difference between is RV32I and RV32E is that the number of available integer registers is reduced from 32 to 16. Note that any processor that implements RV32I can be targeted by a compiler emitting RV32E specific assembly. For real-time workloads that are not register-starved, this is a good trade-off, since the reduced number of registers allows for faster context-switches. We add support for RV32E in a configurable manner allowing the programmer to chose according to her needs.

C. Optimized Task Selection

FreeRTOS ports can optimize the task selection part, which is central in a context switch for choosing the next task to be scheduled. When using optimized task selection, priorities of ready tasks are recorded in a bitfield. We make use of Xpulp's `p.fl1` instruction to find the first bit set looking from the MSB. This is then the highest priority for which a schedulable task is ready. By replacing the compiler intrinsic `__clzsi2`, which counts leading zeros for signed integers that would have been used instead, we save roughly 8 cycles.

D. Vectorized Interrupts

The default FreeRTOS port is not making use of vectorized interrupts, instead funneling all exceptions through a single entry point. This unnecessarily increases the context switch time and interrupt latency due to additional code that needs to be run to

TABLE I
AVERAGE CONTEXT SWITCH TIMES IN FREERTOS FOR POWER CONTROL UNIT FIRMWARE

FreeRTOS version	*# cycles*	*# instructions*
vanilla RISC-V[a]	162	137
optimized RISC-V[b]	129	101
optimized RISC-V[c]	121	93
ARM Cortex-M3 [5]	96	-

[a]`gcc-9.2.0 -O3 -march=rv32imac -mabi=ilp32`
[b]`gcc-9.2.0 -O3 -march=rv32emac -mabi=ilp32e`
[c]Optimized Task Selection with `Xpulp` extension

determine the origin of the interrupt. On the CV32E40P, we support vectored interrupts.

E. Non-RTOS and Nested Interrupts

The current RISC-V FreeRTOS version does not make use of nested interrupts and consequently does not support non-RTOS ISR. We implement non-RTOS ISRs for our custom interrupt controller by only partially masking a set of interrupts depending on a user-defined configuration parameter and quickly re-enabling global interrupts after saving all necessary state. Note that interrupts are disabled on entry to the ISR as dictated by the CLINT specification.

IV. PERFORMANCE ANALYSIS

We run the original, vanilla FreeRTOS port of the power control unit firmware and show the improvement when we apply the optimization from Section III. We then measure the numbers of cycles spent in the RTOS primitives of interest. These measurements are cycle-accurate: the RTL code that describes the hardware design consisting of the CV32E40P, its interrupt controller, and all peripherals are simulated.

A. Context Switch Time

The context switch in FreeRTOS consists of saving the context i.e. registers and other machine state, determining the next task to be scheduled, and finally restoring the context of the potentially new, different task. The results are summarized in Table I. In total, we improve the context switch latency by 25% when applying all optimizations at once to the CV32E40P compared to the vanilla RISC-V FreeRTOS port.

B. Interrupt Latency

We measure both the interrupt latency for the RTOS and non-RTOS ISR case. The latency given in any case is from when the interrupt is registered at the interrupt controller to the first executed instruction in the ISR.

1) non-RTOS ISR: The non-RTOS ISR latency is program independent, assuming we have zero interference on the bus, no wait states in the memory, and no other interrupts pending. The important factors are the implementation of the core itself dictating in how many cycles the register are saved in the memory and the ABI, which determines the number of caller-save registers. For RISC-V, we will measure both the effect of the regular ABI and the EABI. In Table II, we give a summary of the worst-case interrupt latency for non-RTOS ISR.

TABLE II
WORST CASE INTERRUPT LATENCY (NON-RTOS) ON CV32E40P AND ARM CORTEX-M3

Configuration	*# cycles*	*# caller-save*
CV32E40P, regular ABI	33 (23 + 4 + 6)	16
CV32E40P, ABI and RV32E	27 (17 + 4 + 6)	10
CV32E40P, EABI and RV32E	24 (14 + 4 + 6)	7
ARM Cortex-M3, AAPCS [11] [12]	12	6

TABLE III
AVERAGE AND WORST CASE INTERRUPT TIMES IN FREERTOS FOR POWER CONTROL UNIT FIRMWARE

FreeRTOS version	*avg #cycles*	*wcet #cycles*
vanilla RISC-V	71	298
optimized RISC-V	47	239

To be able to implement non-RTOS ISR for the CV32E40P, we have to quickly re-enable interrupts, which are by default disabled when we enter the handler. In order to do that, we have to save machine state registers (`mstatus, mepc, mcause`). During this brief time, interrupts are disabled and effectively form a short critical section of 6 cycles, which we account for in Table II. For each of the configurations of the CV32E40P, the decrease is in line with expectations seeing as how the number of cycles required decreases proportionally with the number of caller-save registers. In the cases outlined in Table II, the CV32E40P requires 6 cycles for the interrupt to propagate through the interrupt controller and make the core jump to the trap handlers entry point, 4 cycles to save the old stack and set up the interrupt stack, k cycles to save k registers and an additional 6 cycles when we hit the worst case critical section.

2) RTOS ISR: For the power control unit firmware, we measure the interrupt latency for a RTOS ISR. The results are summarized in Table III. The worst-case RTOS interrupt latency is estimated by finding the longest continuous critical section and then adding the non-RTOS ISR latency on top of it. With all optimizations applied, we improve the RTOS ISR latency by 33% in the average and 20% in the worst case.

V. COMPARISON WITH ARM

We compare the CV32E40P against an ARM Cortex-M3. The ARM Cortex-M series processors use an interrupt controller called the Nested Vectored Interrupt Controller (NVIC) [13]. It supports hardware-based saving of registers on interrupts, basically completing the caller-save part of the calling convention automatically. Interrupt sources can be divided into levels that pre-empt each other and further subdivided into priorities that resolve ties when several interrupts on the same level happen concurrently.

Since interrupts are vectored and are not disabled when entering their respective handler that, combined with the hardware-assisted saving of registers, allows for low jitter and latency on interrupts. This is because the Cortex-M3 is automatically saving all caller-save registers and the machine state (`xPSR, PC`) to the stack whenever an interrupt happens. That means the core is concurrently able to fetch already the

ISR's address and jump there while the interrupt context is pushed onto the stack.

On the software side, we use the AAPCS ABI [14], which is ARM's version of an embedded ABI. The ARM Cortex-M3 context switch needs 25 fewer cycles, meaning the context switch of the CV32E40P is 26% slower (Table I). The interrupt latency for non-RTOS ISR is 12 cycles, which is twice as fast for the same amount of work (Table II). Furthermore, the machine state is also saved, allowing for nested interrupts. This difference is due to the NVIC and its automatic register stacking. The RISC-V EABI is competitive compared to the ARM AAPCS ABI in terms of the number of caller-save registers, only requiring a single additional register to be saved.

VI. Discussion

Despite optimizing FreeRTOS for the CV32E40P improves real-time performance, a significant gap with respect to the Cortex-M3 remains.

On the software front, we squeezed out most of the performance that we believe is possible, meaning that the remaining gap with respect to context switch time and interrupt latency is due to hardware architectural differences summarized below. Software related constraints for low-latency interrupts on embedded systems have been mostly resolved by introducing the EABI. The Core-Local Interrupt Controller (CLIC) [15] is a promising proposal attempting to deliver faster interrupts to RISC-V. It allows for preemptible, edge-triggered interrupts to be acknowledged in hardware, introduces a prioritization/level scheme to interrupts such as ARMs with the corresponding level-based disabling of them, allowing fine-grained control of which interrupts can enter a critical section, and event makes whether interrupt sources are vectored configurable.

Unfortunately, the fundamental issues are still there: Interrupts are disabled when entering an interrupt handler. The instruction sequence required to save enough state to re-enable them has been shortened by fusing state registers, but it still remains, negatively impacting worst-case interrupt latency and jitter for higher priority or non-RTOS interrupts. No automatic register stacking or equivalent solution is available. We believe that either such a scheme has to be introduced to become competitive with ARM or a whole other approach has to be taken, such as for example, tighter integration of RTOS software primitives with hardware-based solutions [16].

VII. Conclusion

We analyze the current state of real-time support for RISC-V by starting with the CV32E40P, an open-source industrially supported RV32IMFC core and a vanilla, unoptimized FreeRTOS as reference point. We tailor FreeRTOS to the CV32E40P by performing a series of software optimization obtaining a 25% improved context switch time and a 33% better interrupt latency in the average and 20% in the worst case when tested on a power control unit firmware and synthetic benchmarks.

For the non-RTOS interrupt latency, we achieve an improvement of 27%. Nevertheless, a gap remains when compared to the ARM Cortex-M3/4 processor. The non-RTOS interrupt latency of the CV32E40P is with 24 cycles in the optimized version still twice as long as the Cortex-M3's in a comparable setting and the context switch 26% slower. While the emergence of the RISC-V EABI helped lowering the number of caller-save registers, more aggressive methods are required to close the gap.

The Core-Local Interrupt Controller (CLIC) is a promising proposal that tackles some problems, such as hardware acknowledgment of vectored interrupts, interrupt levels and priorities, seamless integration with the CLINT, and we plan to work on automatic register stacking and similar ideas, which remain unaddressed.

References

[1] A. Waterman, "Design of the RISC-V Instruction Set Architecture," Ph.D. dissertation, University of California, Berkeley, Jan. 2016. [Online]. Available: https://www2.eecs.berkeley.edu/Pubs/TechRpts/2016/EECS-2016-1.html

[2] O. Group, "OpenHW Group | OpenHW Group." [Online]. Available: https://www.openhwgroup.org/

[3] M. Gautschi, P. D. Schiavone, A. Traber, I. Loi, A. Pullini, D. Rossi, E. Flamand, F. K. Gürkaynak, and L. Benini, "Near-Threshold RISC-V Core With DSP Extensions for Scalable IoT Endpoint Devices," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 25, no. 10, pp. 2700–2713, Oct. 2017, conference Name: IEEE Transactions on Very Large Scale Integration (VLSI) Systems.

[4] Aspencore, "Embedded Market Study," Tech. Rep., 2019. [Online]. Available: https://www.embedded.com/wp-content/uploads/2019/11/EETimes_Embedded_2019_Embedded_Markets_Study.pdf

[5] "FreeRTOS - Market leading RTOS (Real Time Operating System) for embedded systems with Internet of Things extensions." [Online]. Available: https://www.freertos.org/index.html

[6] "openhwgroup/cv32e40p," Sep. 2020, original-date: 2016-02-18T18:21:33Z. [Online]. Available: https://github.com/openhwgroup/cv32e40p

[7] "The RISC-V Instruction Set Manual, Volume II: Privileged Architecture, Document Version 1.12-draft," RISC-V Foundation, Tech. Rep., Jun. 2019.

[8] A. Chang, P. Dabbelt, and D. Barbier, "RISC-V Platform-Level Interrupt Controller Specification," Tech. Rep., Mar. 2020. [Online]. Available: https://github.com/riscv/riscv-plic-spec

[9] A. Bartolini, D. Rossi, A. Mastrandrea, C. Conficoni, S. Benatti, A. Tilli, and L. Benini, "A PULP-based Parallel Power Controller for Future Exascale Systems," in *2019 26th IEEE International Conference on Electronics, Circuits and Systems (ICECS)*, Nov. 2019, pp. 771–774.

[10] "Proposal for a RISC-V Embedded ABI (EABI)," Tech. Rep., May 2019. [Online]. Available: https://github.com/riscv/riscv-eabi-spec

[11] "Cortex-M3 Technical Reference Manual," Feb. 2010. [Online]. Available: https://developer.arm.com/documentation/ddi0337/h

[12] "Measuring Interrupt Latency," Apr. 2018, documentNumber: AN12078. [Online]. Available: https://www.nxp.com/docs/en/application-note/AN12078.pdf

[13] J. Yiu, *The Definitive Guide to ARM® Cortex®-M3 and Cortex®-M4 Processors*. Newnes, Oct. 2013, google-Books-ID: 9YxqAAAAQBAJ.

[14] "Procedure Call Standard for the Arm Architecture," Jun. 2020. [Online]. Available: https://developer.arm.com/documentation/ihi0042/j/

[15] "RISC-V Core-Local Interrupt Controller (CLIC) Version 0.9-draft-20200908," Tech. Rep., Sep. 2020. [Online]. Available: https://github.com/riscv/riscv-fast-interrupt

[16] F. Mauroner and M. Baunach, "mosartMCU: Multi-core operating-system-aware real-time microcontroller," in *2018 7th Mediterranean Conference on Embedded Computing (MECO)*, Jun. 2018, pp. 1–4.