Hardware support for Static-Priority Stack Resource Policy based scheduling

Per Lindgren*†, Pawel Dzialo* and Henri Lunnikivi†
*Luleå University of Technology
Email:per.lindgren@ltu.se, pawdzi-7@student.ltu.se
†Tampere University
Email: henri.lunnikivi@tuni.fi

***Abstract*—Stack Resource Policy based scheduling comes with unique properties such as race- and deadlock-free scheduling, bounded priority inversion and single stack execution. In this paper we survey the outset for SRP based scheduling for a set of representative modern 32-bit micro-controller families. We show that requirements for hardware accelerated SRP based scheduling are met, allowing for effective implementation by the Rust RTIC framework and a path towards predictable, robust, reliable and secure firmware for industrial applications.**

I. Introduction

Embedded software plays an increasingly important role in the functionality, reliability, safety and security of (electronic) devices. Scheduling and resource management of the embedded software is key to meet the aforementioned criteria. In this paper we survey the outset for hardware accelerated Stack Resource Policy (SRP)[1] based scheduling. In Section II we review essential properties of the SRP execution model and benefits to static analysis. Section III covers interrupt controller architectures for a set of modern representative families of 32-bit micro-controllers. Section discusses opportunities to mapping the SRP execution model onto selected micro-controllers. The paper is summarized in Section V where we based on the survey conclude that the requirements for hardware accelerated SRP based scheduling are best met by architectures providing interrupt nesting and threshold based interrupt filtering such as the Cortex M v7/v8, RISC-V CLIC/ESP32C3 and AURIX TriCore, while other architectures implies larger (yet bounded) overhead for scheduling and resource management.

II. Stack Resource Policy based scheduling

A. Formal Model

We adopt the model introduced by [1]:

- J A job J is a finite sequence of instructions to be executed on a single processor/core.
- $\mathcal{J}$ $\mathcal{J}$ denotes both a job execution request and its execution.
- $p(\mathcal{J})$ Defines the (base) priority of $\mathcal{J}$. $p(\mathcal{J}') > p(\mathcal{J})$ indicates that expediting $\mathcal{J}'$ is sufficiently important that completion of $\mathcal{J}$ is permitted to be delayed.
- $\pi(J)$ Defines the preemption level of a job, defined so that a job J' may preempt J only if $\pi(J') > \pi(J)$.
- R A non-preemptable resource R can be claimed by a job for the execution of a critical section.
- $\lceil R \rceil$ Defines the (static) current ceiling of resource R

$$\lceil R \rceil = max(\{0\} \cup \{\pi(J) | J \in L(R)\})$$

where $L(R)$ is the set of jobs that (may) request R.

- Π Denotes the (dynamic) current system ceiling.

Under SRP a job execution request for J is blocked until $\mathcal{J}$ has the highest priority of all outstanding job's execution requests and $\pi(J) > \Pi$.

B. Restricted SRP

In the original work on SRP [1], the job preemption level is set dynamically allowing e.g., Earliest Deadline First (EDF [2]) scheduling. However, by restring SRP to static preemption levels (formally $\pi(J) = p(J)$) we can leverage on commonplace interrupt hardware for accelerated scheduling as shown in [3]. Similarly the original work on SRP allowed multi-unit resource. By the restriction to single-unit resources, the computation of Π can be defined as:

$$\Pi = max(\{0\} \cup \{\pi(J)\} \cup \{\lceil R \rceil | R \in R_{claimed}\})$$

where J is the currently executing job (if any), and $R_{claimed}$ the set of currently claimed (outstanding) resources. This allows efficient (constant time) management of Π [3].

C. Properties

The original work on SRP [1] derives a set of outstanding features for preemptive scheduling onto single-core platforms:

1) Race- and deadlock free scheduling by construction.
2) Bounded priority inversion.
3) Single context switch per job request.
4) Single (shared) stack execution.
5) Response time and overall schedulability test.

The resource access guarantees 1) provides an outset for sound, robust and reliable scheduling. The scheduling guarantees 2) and 3) gives an outset for CPU efficient and deterministic execution, while 4) provides the outset for memory efficient execution[1]. Finally 5) allows compile-time guarantees to worst case timing behavior.

[1]Under SRP, a safe upper bound for required stack memory can be determined by the sum of max stack depth for each unique priority in the system, independent to the number of tasks in the system

D. Multi-core extensions

The original work on SRP targeted scheduling on single-core systems. Several extensions to multi-core scheduling have been proposed (e.g., MSRP[4] and MrsP[5]). However, shared access to resources cross cores comes with an inherent cost of non core-local blocking. An alternative approach is to statically partition the system such that all tasks accessing any common shared resource belong to the same domain. This way the outstanding properties of SRP still holds, while parallelism can be leveraged domain level [6].

E. Hardware assisted scheduling

SRP based scheduling can be effectively implemented and accelerated by exploiting the underlying hardware. These are the minimal requirements:

1) preemptive (nested) interrupt execution,
2) interrupt sources with individual priorities,
3) interrupt requests possible from both hardware and software, and
4) interrupt filtering by priority threshold or masking.

III. Interrupt Controller Architectures

Interrupt and exception mechanisms are provided by all modern micro-controllers. However supported features (Table I) vary by architecture (Table II) as shown in Table III. In the following we give a survey on interrupt and exception support for a representative set of modern 32-bit architectures.

Priorities	Number of priority levels
G Mask	Interrupts can be masked globally
S Mask	Interrupts can be masked individually per source.
Threshold	Interrupts can be masked by a threshold value

TABLE I
Legend: Hardware features

CM	Cortex M0/M0+/M1/M23 (ARMv6-M/v8-M-base) [7], [8]
CM*	Cortex M3/M4/M7/M33 (ARMv7-M/v8-M-main) [9], [8]
RV32	32-bit RISC-V basic interrupt architecture [10]
RV32-CLIC	32-bit RISC-V with CLIC support according to [11]
ESP32-C3	Espressif RISC-V [12]
AURIX	Infineon AURIX (TC3xx) [13]

TABLE II
Legend: Hardware architectures

Table III summarizes a set of modern 32-bit MCU families:

Feature	CM	CM*	RV32	RV32-CLIC	ESP32-C3	AURIX
Priorities	4	4-256	no	256	16	256
G Mask	yes	yes	yes	yes	yes	yes
S Mask	yes	yes	yes	yes	yes	yes
Threshold	no	yes	no	yes	yes	yes
HW Int	≤ 32	32 - 240	1+1+16	≤ 4096	58	≤ 1024
SW Int	≤ 32	32 - 240	1	≤ 4096	4	8

TABLE III
Hardware supported features

A. ARM Cortex-M

The interrupt mechanisms on the ARM Cortex-M architecture are controlled by the core registers together with the NVIC peripheral (Section III-A1). In the following interrupts and exceptions are used interchangeably unless stated else wise.

Priorities are ordered from -3 (Reset), -2 (NMI), -1 (HardFault), 0 (highest interrupt priority), up to 255 (lowest priority). The number of interrupt priority bits varies from 2 to 8, depending on architecture and vendor implementation. Assuming 4 priority bits, 16 non-negative priories are possible.

1) Core registers and NVIC: The core registers `PRIMASK`, `FAULTMASK` and `BASEPRI`, can be set to filter interrupts based on priority. `PRIMASK` sets the execution priority to 0, essentially disabling external interrupts. `BASEPRI` changes the priority level required for exception preemption (a 0 value disables filtering).

Exceptions and interrupts are preemptively executed (nested). Interrupts can be individually masked (disabled/enabled, masking 32 sources in parallel), but only a subset of exceptions are maskable and have configurable priorities. Interrupt pending status can be individually masked allowing software interrupt request.

Together, core registers and the NVIC fulfills the requirements for hardware accelerated scheduling in Section II-E.

2) Interrupt handling: When an interrupt is taken, the state context is saved by hardware onto the stack and the processor is set in `Handler` mode. [2] Hardware stacking follows the ARM EABI (calling convention) allowing ordinary functions to implement handlers, as the hardware stacks the *caller-saver* registers. When executing a return instruction[3] in `Handler` mode the state context is restored. Preemptive context stacking and interrupt tail-chaining ensures predictive dispatch with low overhead.

B. RISC-V

The interrupt mechanism on RISC-V is similar to ARM Cortex-M in that the interrupts are managed via control/status registers (CSRs) specified in the RISC-V Privileged ISA Specification [10], and dispatched by one or more interrupt controller devices and their associated memory-mapped register interface.

Interrupt handling on RISC-V is heavily influenced by the RISC-V privileged architecture. A RISC-V machine may have one to three different privilege levels that are effectively isolated from each other as execution environments. These privilege levels as shown in Table IV are referred to as Machine, Supervisor and User. While machine mode is mandatory for every RISC-V machine, rest are optional. [10]

Each privilege mode provides their own set of CSRs and a program counter, forming an independent hardware thread (Hart). Each Hart has a status (`x`status) register to control the operating state of the Hart, an interrupt enable (`x`ie) register

[2]For processors supporting hardware floats, these can be either pushed by hardware or lazily handled.

[3]LDM/POP that loads PC, LDR with PC as destination or BX

Level	Name	Abbreviation
0	User/Application	U
1	Supervisor	S
2	Reserved	
3	Machine	M

TABLE IV
RISC-V PRIVILEGE MODES

to control currently active interrupts and an interrupt pending (xip) register to track currently pending interrupt lines. For instance, these machine mode interrupt control registers are called `mie`, and `mip`. Table V illustrates their composition. While xip cannot be directly used to raise a software interrupt, the specification[10, p. 29] states that the platform implementation should provide a separate memory-mapped register for doing so. The specification therefore defines a way for platforms to make interrupt requests from software satisfying the software side of requirement Section II-E 3.

Index	Vector	Abbreviation
0	User Soft	USI
1	Supervisor Soft	SSI
3	Machine Soft	MSI
4	User Timer	UTI
5	Supervisor Timer	STI
7	Machine Timer	MTI
8	User External	UEI
9	Supervisor External	SEI
11	Machine External	MEI
16+	External / Custom	-

TABLE V
RISC-V M-MODE INTERRUPT REGISTER STRUCTURE, E.G., *mip*, *mie*, *mcause* & *mideleg*

RISC-V interrupt sources are commonly divided into local and global (external) interrupts. Local interrupt management is designed around two standard root interrupt sources, timer and software interrupt (inter-processor interrupt, IPI). External interrupt sources provide the ability for other platform-specific devices to interrupt the core. Together, these interrupt sources provide a way for both hardware and software to request an interrupt, satisfying completely the requirement Section II-E 3.

1) Interrupt handling: To serve the interrupts, the core follows a simple fixed priority scheme where interrupts are handled in the following order: `MEI`, `MSI`, `MTI`, `SEI`, `SSI`, `STI`. While this already satisfies the minimal requirement for interrupt sources with individual priorities in Section II-E 2, an interrupt controller can then implement a more complex policy providing more flexibility.

As may be apparent from the description, due to independent CSRs between privilege levels, RISC-V cores are able to natively preempt and serve *vertical interrupts* between privilege levels without an external interrupt controller. This however does not yet satisfy the requirement Section II-E 1 as it does not allow meaningful preemption within the application domain, thus an extension to the specification is required.

2) Core-Local Interrupt Controller (CLIC): The core RISC-V ISA specification is commonly extended with an interrupt controller specification which is used to enable fulfillment of application area specific requirements such as horizontal (i.e., intra-privilege mode) preemption, prioritization, latency constraints, or security constraints. The Core-Local Interrupt Controller (CLIC)[11] is one such proposal for an architecture extension.

The CLIC supports 4096 individually configurable, vectored, preemptive interrupts, satisfying fully requirement 1 for preemptive interrupt execution of SRP based scheduling. The CLIC specification subsumes the basic local interrupt prioritization behavior described in subsection III-B1, extending it with full per-interrupt prioritization.

Extending the RISC-V inter-privilege vertical interrupts, CLIC adds support for horizontal preemption for up to 256 interrupt levels where level zero corresponds to regular execution outside of an interrupt handler and higher-numbered levels can preempt lower-numbered levels [11]. While the exact number of supported interrupt levels is defined by a platform profile, more than one level is required to satisfy requirement Section II-E 1 for preemptive interrupts. The extension also specifies software conventions for handling nested interrupt handlers, similar to how they are handled for ARM Cortex-M and NVIC, explored in Section III-A2.

In a given privilege mode, CLIC can disable preemption for interrupts below set threshold using xintthres. This satisfies the final requirement Section II-E 4 and allows implementation of a critical section for a subset of interrupts, while still allowing preemption from interrupt levels above the threshold.

C. ESP32-C3

The ESP32-C3[12] is an implementation of the RISC-V ISA with an interrupt controller loosely based on the CLIC. The controller provides 58 peripheral and 4 software interrupt sources, satisfying requirement Section II-E 3. Each of the interrupt sources to be used must be mapped to one of the 31 provided CPU interrupts via an interrupt matrix.

A priority configuration register is available for each of the CPU interrupts, enabling individual priority levels as referenced in requirement Section II-E 2. The overall priority implementation is similar to the CLIC, with 16 available priority levels and a configurable priority threshold which enables filtering in accordance with requirement Section II-E 4. This means that horizontal preemption is possible, with full horizontal nesting implemented on the same basis as for the CLIC.

Apart from the additional functionality provided by the interrupt matrix, some of the standard control/status registers for interrupt control of the generic RISC-V (`mstatus`, `mtvec`, `mcause`, `mepc`), are implemented. Something to note is that the CPU only supports machine-mode vectored interrupts.

CPU interrupts are handled mostly as described for the vectored mode of the general RISC-V, with the core difference being the priority scheme, and the mapping of interrupts to specific indices, and consequently the resulting ISR addresses. The root sources of the RISC-V are not implemented, and instead, each of the CPU interrupts is simply enumerated from 0 to 31. From there, the ISR address is calculated as in the

general case. For instance, the CPU interrupt `x` results in the ISR address `mtvec+4*x`.

D. Infineon AURIX

The AURIX TC3xx[13] architecture adopts a flexible Interrupt Router (IR) module including the following features:

- Up to 8 ICUs.
- Support of up to 255 service request priority levels per ICU.
- Low latency service request arbitration ≤ 4 clock cycles.
- 8 General Purpose Service Requests (GPSR) per CPU that can be used as Software Interrupts.
- Priority dependent masking of service requests (for CPUs, related control registers included in the CPUs.)

In earlier work [14] it has been shown that the AURIX/TriCore architecture meets the requirements for SRP compatible scheduling (Section II-E).

IV. Hardware accelerated SRP based scheduling

A. SRP model as interrupts

For architectures meeting the requirements from Section II-E we can map an SRP model as follows:

- J A job J corresponds to a terminating function (without arguments and return values). Each job is associated to an interrupt vector $V(J)$.
- $\mathcal{J}$ Execution request of $\mathcal{J}$ amounts to an interrupt request to $V(\mathcal{J})$
- Π The system ceiling Π is implemented either by threshold based filtering (e.g., CM*-`BASEPRI` or CLIC-`xintthresh`), or by interrupt masking (e.g., CM `NVIC_ICERn`/`NVIC_ISERn`) or CLIC `clicintie`).

B. Expected overhead

On platforms with nested interrupt controllers static priority scheduling is directly performed by the hardware without additional overhead[4]. For other architectures, nesting emulation amounts to additional bookkeeping (queuing of pended, not yet dispatched tasks), where the number of interrupt sources gives the upper bound complexity.

Resource protection (concurrency control) amounts to system ceiling manipulation. On platforms with threshold based interrupt filtering resource lock amounts to reading the threshold, setting the new value and restoring the read value on unlock. For other platforms interrupts are filtered by masking, e.g., on the Cortex M0 (supporting 32 interrupt sources), this amounts to writing a 32-bit disable-mask register on resource lock and writing a 32-bit enable-mask register on unlock. For the Cortex M23 supporting 240 interrupt sources requires a maximum of 8 32-bit accesses on lock/unlock, however, the actual number of register accesses are application dependent. In any case resource management amounts to a bound number of operations, where threshold and mask values are determined statically (at compile time).

[4]RISC-V requires software based call-save register stacking while on Cortex M platforms stacking is performed as part of the interrupt hardware dispatch mechanism.

C. Example

The RTIC framework is a prominent example of hardware accelerated SRP based scheduling. The RTIC framework is implemented as a Rust language extension (procedural macro), performing the following steps at compile time:

1) The RTIC application is translated into an SRP based task/resource model and checked for consistency.
2) Resource ceilings are computed from the model.
3) Executable code is generated from the model.

The RTIC framework provides an executable model for real-time systems programming, upholding soundness invariants for the Rust language (providing strong guarantees of memory safety) as well as ensuring SRP requirements (such as LIFO access to resources). In this way, an application passing compilation can claim stronger guarantees to system wide memory safety and schedulability than other operating systems or kernels (including the formally verified seL4 microkernel[15]). They state:

"In fact, any system, no matter how secure, can be used in insecure ways."

This may hold for seL4 and other traditional kernels, however RTIC is fundamentally different. The task/resource model is unbreakable (leveraging on the Rust language and its distinction between *safe* and *unsafe* code), thus system wide properties regarding safety and security can be obtained by construction[16].

V. Conclusions

In this paper we have provided a survey of hardware supported interrupt mechanisms. Whereas there exists architectural differences, the requirements for SRP based scheduling as implemented by the Rust RTIC framework are met. We can expect best performance on architectures providing hardware supported interrupt nesting and threshold based interrupt filtering such as the Cortex M v7/v8, RISC-V CLIC/ESP32C3 and the AURIX TriCore architecture. While SRP scheduling on other architectures is still feasible by software emulation, additional (yet bounded) overhead for scheduling and resource management is to be expected.

RTIC is already widely used in industry with companies like Volvo leveraging on the outstanding guarantees of SRP based scheduling[17].

In future work we foresee to provide RTIC support for the set of architectures covered in the article. A backend implementation for the ESP32C3 is currently under evaluation. Initial results indicate performance on-par with our expectations (i.e., similar as the Cortex-M v7/v8) and set for inclusion in the next release. Additionally, initial experiments of software emulated interrupt nesting looks promising. The goal with this work is to support a wider range of RISC-V based micro-controllers at the cost of bounded additional scheduling overhead. Recently Infineon went official with their support of Rust for the safety critical micro-controllers [18].

REFERENCES

[1] T. P. Baker, "A stack-based resource allocation policy for realtime processes," *[1990] Proceedings 11th Real-Time Systems Symposium*, pp. 191–200, 1990.

[2] J. Xu and D. Parnas, "Scheduling processes with release times, deadlines, precedence and exclusion relations," *IEEE Transactions on Software Engineering*, vol. 16, no. 3, pp. 360–369, 1990.

[3] J. Eriksson, F. Häggström, S. Aittamaa, A. Kruglyak, and P. Lindgren, "Real-time for the masses, step 1: Programming API and static priority SRP kernel primitives," in *2013 8th IEEE International Symposium on Industrial and Embedded Systems (SIES 2013) : 19-21 June 2013, Porto, Portugal*, 2013, pp. 110–113, godkänd; 2013; 20130701 (pln).

[4] P. Gai, G. Lipari, and M. Di Natale, "Minimizing memory utilization of real-time task sets in single and multi-processor systems-on-a-chip," in *Proceedings 22nd IEEE Real-Time Systems Symposium (RTSS 2001) (Cat. No.01PR1420)*, 2001, pp. 73–83.

[5] A. Burns and A. Wellings, "A schedulability compatible multiprocessor resource sharing protocol – mrsp," in *2013 25th Euromicro Conference on Real-Time Systems*, 2013, pp. 282–291.

[6] J. Aparicio Rivera, "Real time rust on multi-core microcontrollers," Master's thesis, Luleå University of Technology, Computer Science, 2020.

[7] ARM, "Armv6-M Architecture Reference Manual," 2023. [Online]. Available: https://developer.arm.com/documentation/ddi0419/e/

[8] ——, "Armv8-M Architecture Reference Manual," 2023. [Online]. Available: https://developer.arm.com/documentation/ddi0553

[9] ——, "Armv7-M Architecture Reference Manual," 2023. [Online]. Available: https://developer.arm.com/documentation/ddi0403/ee/

[10] "The RISC-V Instruction Set Manual, Volume II: Privileged Architecture," 12 2021, accessed 11.3.2023. [Online]. Available: https://github.com/riscv/riscv-isa-manual/releases/download/Priv-v1.12/riscv-privileged-20211203.pdf

[11] "Core-Local Interrupt Controller (CLIC) RISC-V Privileged Architecture Extensions," accessed 11.3.2023. [Online]. Available: https://github.com/riscv/riscv-fast-interrupt/blob/master/clic.adoc

[12] Espressif, "ESP32C3 Technical Reference Manual," 2023. [Online]. Available: https://www.espressif.com/sites/default/files/documentation/esp32-c3_technical_reference_manual_en.pdf

[13] Infineon, "AURIX™ TC3xx," 2023. [Online]. Available: https://www.infineon.com/dgdl/Infineon-AURIX_TC3xx_Part1-UserManual-v02_00-EN.pdf?fileId=5546d462712ef9b701717d3605221d96

[14] W. Hofer, D. Lohmann, F. Scheler, and W. Schröder-Preikschat, "Sloth: Threads as Interrupts," *2009 30th IEEE Real-Time Systems Symposium*, pp. 204–213, 2009.

[15] S. Systems, "seL4 - If I run seL4, is my system secure?" 2023. [Online]. Available: https://docs.sel4.systems/projects/sel4/frequently-asked-questions.html

[16] M. Lindner, J. Aparicio, and P. Lindgren, "Concurrent Reactive Objects in Rust Secure by Construction," *Ada User Journal*, vol. 40, no. 1, 2019, validerad;2019;Nivå 1;2019-04-09 (inah).

[17] Volvo, "Why Rust is actually good for your car," 2023. [Online]. Available: https://medium.com/volvo-cars-engineering/why-volvo-thinks-you-should-have-rust-in-your-car-4320bd639e09

[18] Infineon, "Automotive Safety and Cybersecurity: Infineon's AURIX™ TC3xx, TC4x, TRAVEO™ T2G & PSoC families of microcontrollers support Rust," 2023. [Online]. Available: https://www.infineon.com/cms/en/about-infineon/press/market-news/2023/INFATV202303-076.html