POLITECNICO
MILANO 1863

SCUOLA DI INGEGNERIA INDUSTRIALE
E DELL'INFORMAZIONE

Executive Summary of the Thesis

EVA: A Rust-based Embedded Real-time Operating System

Laurea Magistrale in Computer Science and Engineering - Ingegneria Informatica

Author: **Davide Mor**

Advisor: **Prof. Federico Terraneo**

Academic Year: **2025-26**

1. Introduction

As our world evolves, our reliance on embedded systems increases. This includes safety-critical applications, where the highest standards of coding are required. Despite this, the way we write embedded software has not changed fundamentally. As of 2023, 70% of embedded devices still rely on C and C++ as their main language of choice [1]. These languages are by design memory unsafe, moving the burden of validating memory accesses to the programmer. This has proven to be a great source of bugs and issues. The Rust programming language aims at solving this by providing a memory safe programming language that relies almost entirely on static compile-time checks.

1.1. Ecosystem Gap

Despite wide adoption in several industries, such as cloud infrastructure and high performance computing, Rust still has not gained much traction in the embedded world. We believe the culprit to be an inconsistent ecosystem, still dominated by C and C++. Many widely adopted and mature embedded operating systems have poor Rust support. While Rust-based embedded operating systems do exist, they suffer from the opposite problem of being too Rust centric and lacking support for existing infrastructure. This last issue needlessly heightens Rust commitment cost, reducing Rust adoption.

1.2. Contributions

In this thesis we showcase EVA[1], our vision for a Rust-based embedded operating system. It focuses on four main goals:

- Being written and designed in Rust from the outset.
- Multi-language support, none of the features of this operating system should be restricted to Rust alone.
- Optional support for higher level APIs such as the Rust std library and POSIX.
- Portability across a wide spectrum of embedded devices.

2. EVA

EVA is architecturally divided into layers. Figure 1 illustrates these layers and how they interact. This diagram also includes greyed out components, which are not implemented in the current EVA release but are planned for future developments.

2.1. Language Library Layer

To achieve multi-language support, the interface between the application layer and kernel layer

[1] https://github.com/Tazdevil971/eva

needs to be specifically engineered. We provide two almost identical APIs to the kernel. The first is based on the C ABI, which was chosen due to its status of de-facto lingua franca when it comes to interfacing different languages. The second one is based on the Rust ABI directly, which allows for seamless Rust to Rust interaction (a Rust kernel application to EVA). Apart from a direct interface with the kernel, usable in constrained environments, we also provide higher abstraction layers in the form of libraries. The full Rust std library is better for ease of development but slows down performance and, most importantly, increases code size. The EVA klib provides some abstraction over the raw interface with the kernel, but with a much smaller footprint. It is currently provided as part of the kernel crate and can only be accessed by depending on the kernel directly. Finally, libc support is planned for languages other than Rust.

2.2. Board Support Package Layer

Portability is a critical aspect of embedded operating systems. In EVA this is provided through the portability interface, whose objective is to decouple the particular hardware portability implementation from the kernel itself. The portability implementation can be anywhere in the dependency graph. This service is provided by the Board Support Package (BSP), which also supplies applications with drivers and other abstractions over the hardware. EVA provides some standard BSPs, but applications can also use out-of-tree BSPs, meaning that hardware support can be easily augmented in case of need.

The portability interface itself works by defining a standard interface inside the kernel itself marked as "extern". Since a strong interaction is required between the portability implementation and the kernel, we decided that the portability interface is only accessible from Rust.

2.3. Kernel Layer

EVA main component is the kernel. We decided on a unikernel design [2], which is adopted widely as the architecture of choice for the embedded world due to its simplicity and low overhead. Our kernel provides several features:

- A real-time, highest priority first round-robin scheduler.

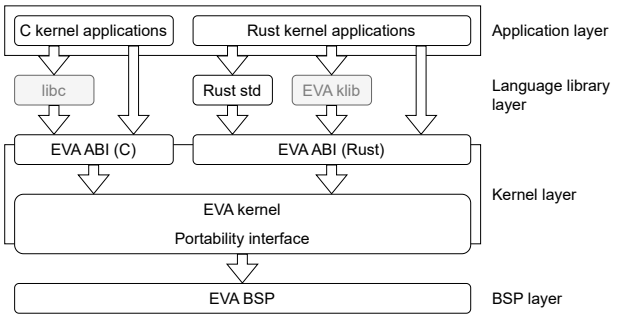

Figure 1: EVA architecture.

- A dynamic memory allocator, based on a linked list and first fit algorithm.
- Textual input/output interface for easy communication with the outside world.
- A timing subsystem, used for task preemption and timekeeping.

2.3.1. Kernel Interfaces and Linking

Rust dependency system is acyclic, which imposes strict limitations on how we structure our system. For example, with our current design it would be impossible to implement the BSP, as it depends on the kernel for bootstrapping, and the kernel depends on it for the portability implementation. We circumvent this issue by manually specifying interfaces used across crates. By tagging functions with "extern" the final linker stage is now responsible for resolving the dependency. As such we can put the declaration and definition separately anywhere in the dependency tree, allowing much more flexibility.

However, this does come at a cost. Since those functions are now opaque to Rust, a lot of optimizations, such as inlining, are no longer possible. We can regain the lost optimization potential by enabling fat Link Time Optimization (LTO). This compiler feature allows the codegen backend (in our case LLVM) to perform an extra step of optimization after linking, at the cost of longer compilation times. The effect of this is shown more concretely later in Section 3.

2.3.2. Threads

A thread is a single execution flow, which can be time-multiplexed by the scheduler. Each thread is associated with a Thread Control Block (TCB) containing thread status and other data needed for scheduling. Alongside the TCB, each thread also requires a switch context, used to store data for switching in and out of threads, and a stack.

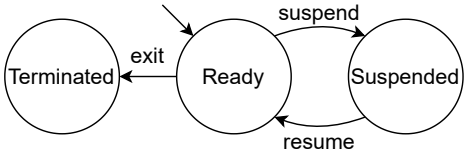

Figure 2: Thread state machine.

All of which are partitioned from a single allocated chunk.

Thread behavior is controlled by an internal state machine, shown in Figure 2. Ready is the only state where a thread can be executed. Suspended is used to halt execution for the suspend and resume operations (explained in Section 2.3.3, layer 2). Finally, terminated is a sink state used to identify a thread that has concluded execution but can be still interacted with.

Thread lifetime is managed via three operations: spawn, join, and detach. Spawn creates a thread in the ready state and registers it to the scheduler. Join waits for the target thread to terminate and deallocates its resources. Detach either deallocates the target thread resources immediately or it signals to the target thread to clean up its resources upon termination.

2.3.3. Scheduler

Scheduling is based on a highest priority first, round-robin algorithm. We support 33 different priority levels, 32 of which are usable by applications, and one is reserved for the idle task. Currently, the system only supports single-core devices. Scheduling uses linked list queues for each priority level and an upper level bitset to easily query for the highest non-empty priority queue. The primitives exposed by the scheduler are subdivided into four layers, shown in Figure 3. Each layer is allowed to use the primitives below it, but not above. Switching the current active task is an operation known as a context switch, which is performed by invoking the scheduler code. The mechanism through which the scheduler gets invoked depends on the particular portability implementation, but it is usually performed via a software triggered interrupt. The scheduler can either be invoked via code directly, through a yield operation, or periodically by the timing subsystem to perform preemption and time quantum expiration.

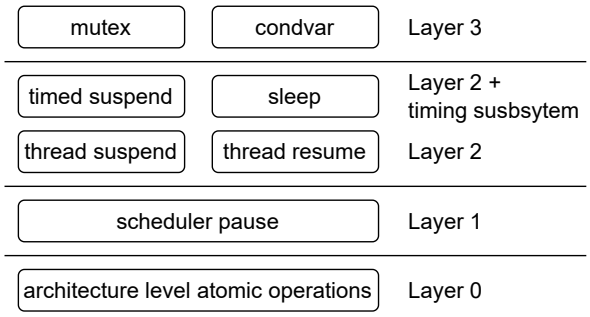

Figure 3: Layers of scheduling primitives.

Layer 0: The first layer consists of atomic operations guaranteed at the architectural level. Currently, EVA relies on three classes of operations: atomic load and stores, atomic Compare And Swap (CAS), and finally atomic swap. Atomic loads and stores do not require special hardware considerations on single-core systems and are translated into regular loads and stores by the compiler. Atomic CAS and atomic swap operations are not always supported at the hardware level, for example, on the Cortex-M0. On those platforms these operations are emulated by disabling and re-enabling interrupts globally, creating a critical section that cannot be interrupted in any way. For this reason these last two operations are used sparingly.

Layer 1: The second layer includes the scheduler pause, which is a synchronization mechanism that prohibits the scheduler from running while inside the critical section. A scheduler pause can be acquired and released, and code running between the two actions is guaranteed to never be preempted. If a scheduler invocation is requested while a scheduler pause is active, for example due to a time quantum expiration, a pending flag is set. Upon releasing the scheduler pause this flag gets checked, and, if set, the scheduler is immediately invoked via a yield operation. By allowing certain variables to be accessed only when a scheduler pause is active, we guarantee that the thread holding the scheduler pause has exclusive access to those resources. Due to the performance considerations highlighted previously, the scheduler pause is implemented only with atomic loads and stores.

Layer 2: On top of the scheduler pause, we implemented thread suspend and resume. A thread can temporarily disable execution of itself by invoking a suspend operation. The thread will no

longer be scheduled until another thread invokes resume on the original thread. This allows us to implement wait and signal mechanisms, the basis for most other synchronization primitives.

At this same level we can also find timed variants of the suspend operation. Threads can therefore suspend themselves with the guarantee of being resumed upon expiration of the timeout. The scheduler itself is responsible for resuming threads using a suspended timeout.

Both the timed and normal variants of the suspend operation are guaranteed to only terminate when a matching resume is invoked. This means that spurious wakeups do not affect the system. Despite this, we do not make any assumption about the absence of spurious wakeups in the kernel and encourage application developers to do the same. Since invoking resume is a safe operation, code must behave correctly even if the resume is unexpected, which could be critical in certain applications, such as mutexes.

Layer 3: The final layer is only implemented on layer 2 and layer 1 and does not depend in any way on the kernel internals. This layer currently provides a mutex and condition variable, both of which are well known and commonly supported synchronization primitives.

The present version of the mutex does not protect against priority inversion. Both primitives respect priority (higher priority threads are serviced before lower priority ones) and fall back to order of arrival (FIFO) for threads with the same priority.

Both primitives support timed operations, which guarantee a maximum wait time, after which the operation simply fails.

2.3.4. Avoiding IRQ Disable in The Kernel

A notable difference between EVA and other embedded unikernels is the lack of a kernel level way of disabling and enabling interrupts. This is by design, as we believe this offers several advantages:

- Management of interrupts is left to the drivers, which, by working at a much lower level, can have more granular control over them.
- Since the kernel rarely disables interrupts internally (the only exception being emulation of atomic CAS and swap operations), we retain a much greater responsiveness to interrupts.
- A kernel pause and an interrupt disable have a similar but slightly different role, which is a common source of confusion about what primitive to use, often leading to deadlocks and other issues. By only having one of them, we avoid this confusion completely.

The only major downside is that kernel pauses are not accessible in every context, as interrupt handlers are not guaranteed to be able to acquire one. This implies that certain operations, such as thread resume, cannot be performed inside those contexts. We solve this issue by providing a subset of the kernel API, which is accessible from any context. Those APIs work through a concept we call “kernel operation defer queues”. These are lock-free queues, implemented using a lock-free stack (also known as a Treiber stack [3]). The first step performed by the scheduler code is checking these queues and performing all deferred operations. Currently, only thread resumes are supported using this mechanism, but in the future we could potentially expand the interface to include more operations.

2.3.5. Idle thread

To guarantee that there is always at least one thread available for execution at any given time, we introduce an idle thread. Since this thread should only run when no other thread is ready, it is given the lowest priority possible, and it’s the only thread allowed with that priority. This thread is also used to perform “garbage collection” tasks, such as cleaning up resources of terminated detached threads.

2.4. Porting the Rust Std Library

To tap into the existing Rust ecosystem, we already provide support for the Rust std library. To utilize this feature, a custom compiler is required[2], as Rust does not support building a custom std library separately from the compiler.

[2] https://github.com/Tazdevil971/rust/tree/tazdevil971/eva-std

The Rust std requires a minimum amount of features to be present on the target platform to be ported. These include:

- A dynamic memory allocator.
- Mutexes and condition variables.
- Thread Local Storage (TLS) (either dynamic or native). As of the current release we only support dynamic TLS.

To provide extended functionality, we also provide support for:

- Textual input and output.
- Thread lifetime management (spawn, join, and detach).

2.5. Running on Linux

Another major feature is support for a Linux-based BSP. This allows EVA to run entirely inside a Linux process, opening access to existing tooling, such as debuggers and fuzzers. We use signals to emulate interrupts and interrupt handlers. For example, the scheduler code is invoked via one of the Linux real-time signals. The timing subsystem relies on a timer to periodically send the scheduler signal. Since we cannot rely on the libc to implement our functionality, as it could conflict with the future EVA libc, all Linux invocations are performed via system calls directly and no system libraries are used.

3. Benchmarks

To show the feasibility of EVA design, we provide some benchmarks. These benchmarks compare EVA with a mature operating system, Miosix [4], specifically the latest stable version: 2.81. We tested two different versions of EVA, the normal unstable branch and an ad hoc modified version called the "unchecked" version. This altered version is stripped of the majority of safety runtime checks that Rust performs. Comparing this version with the base one allows us to evaluate how these checks affect performance. We also evaluate two build configurations for both versions of EVA: a normal release build and a more aggressive "releaseplus" configuration. The latter reduces codegen units to one and enables fat LTO, maximizing the optimization potential.

The benchmarks are conducted by toggling a pin on and off, an operation requiring at most four instructions. We then use an external tool (an oscilloscope) to measure the time of the two fronts. All the benchmarks use a microbenchmarking style, only testing a singular operation. Finally, our testing platform is an STM32F767ZI, with a SysClk and CPU clock of 16 MHz.

The five benchmarks are structured to measure:

1. Time between a yield and execution of the next thread.
2. Time between a condition variable signal and execution of the thread waiting on that condition variable.
3. Time of an uncontended mutex lock followed by an uncontended mutex unlock.
4. Time between a contended mutex lock and the execution of the next thread.
5. Time between a contended mutex unlock and the execution of the thread waiting on that lock.

Results are reported in Table 1, using calculated CPU cycles as a measure of time.

These benchmarks validate a few key insights:

- EVA is slower when compared to Miosix. On average, we see a -40% speed decrease. Our goal was not performance, yet we are still not too far from a well optimized and mature operating system. We hope that in the future we will be able to close the gap.
- Performing a fat LTO optimization pass has a considerable effect on performance, roughly a $+45\%$ increase.
- Rust runtime checks definitely play a role in performance, but they are arguably a worthy tradeoff. For release builds, the speed-up obtained by removing checks is $+17\%$, while for "releaseplus" builds it's $+5\%$. This once again highlights the importance of fat LTO.

4. Conclusions

We have built EVA, an embedded real-time operating system written entirely in Rust. Through intentional design, we have provided first class multi-language support, vast hardware portability, and Rust std library support. Finally, we benchmarked EVA, showing not only how Rust runtime checks are an affordable performance cost but also how EVA could realistically scale to compete with other embedded operating systems.

Experimental data (cycles)

OS	Bench 1	Bench 2	Bench 3	Bench 4	Bench 5
Miosix 2.81 (-O2)	428	572	84	480	486
EVA (release)	716	1600	259	1008	1472
EVA unchecked (release)	576	1216	281	832	1248
EVA (releaseplus)	492	1139	160	720	1056
EVA unchecked (releaseplus)	454	1032	166	720	944

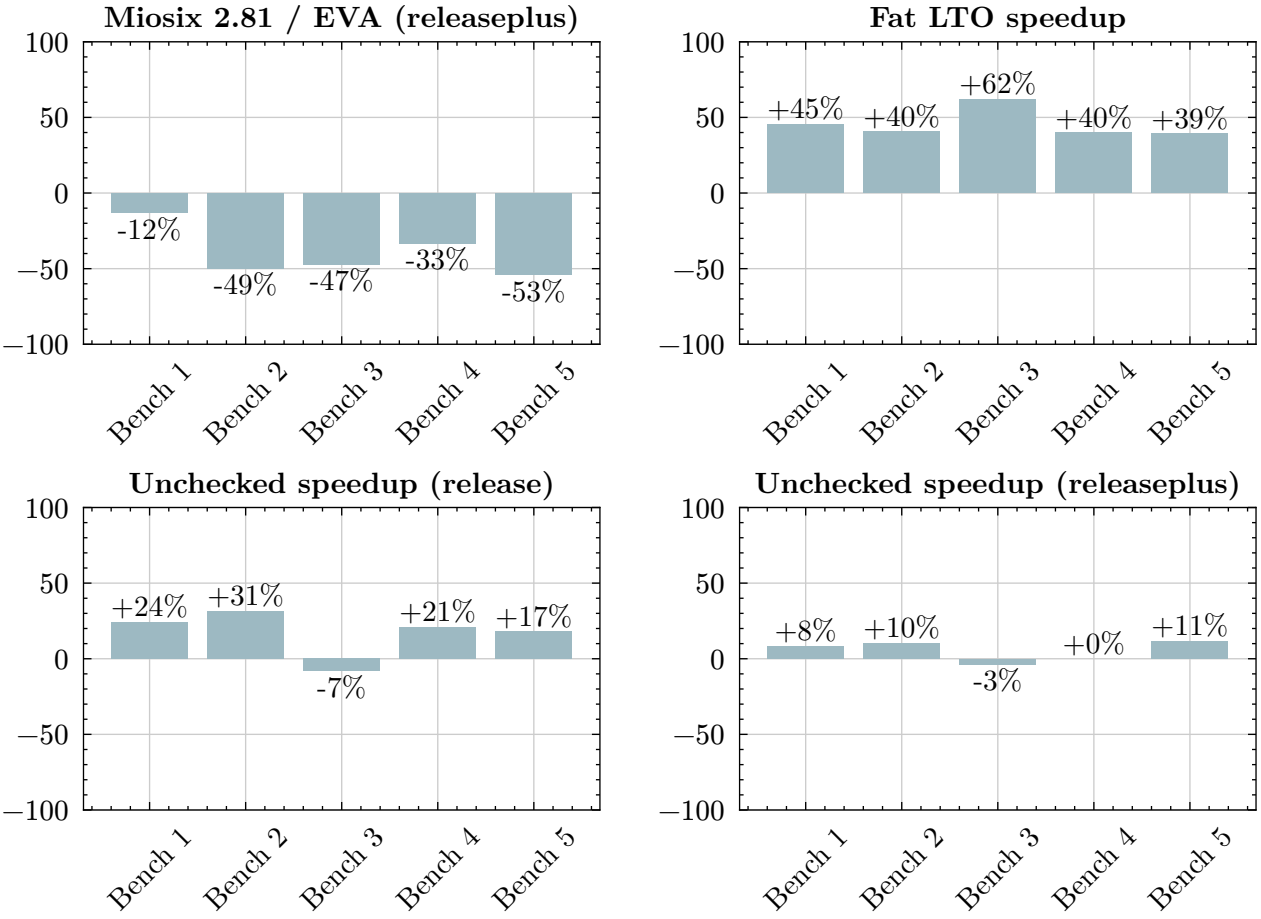

Table 1: Results of experimental benchmarks.

References

[1] Embedded.com, "Embedded Market Study 2023 (Programming Languages in Embedded Systems)." [Online]. Available: https://www.embedded.com/wp-content/uploads/2023/05/Embedded-Market-Study-For-Webinar-Recording-April-2023.pdf

[2] A. Madhavapeddy *et al.*, "Unikernels: library operating systems for the cloud," *SIGARCH Comput. Archit. News*, vol. 41, no. 1, pp. 461–472, Mar. 2013, doi: 10.1145/2490301.2451167.

[3] R. K. Treiber, "Systems Programming: Coping with Parallelism," Yorktown Heights, NY, USA, technical report RJ5118, 1986.

[4] A. Leva, M. Maggio, and F. Terraneo, "Performance analysis of operating systems schedulers realised as discrete-time controllers," in *2012 IEEE International Conference on Control Applications*, 2012, pp. 745–750. doi: 10.1109/CCA.2012.6402429.