A Rust Framework for Real-Time Parallel Programming

Hugo Silva
SoftCPS/ISEP, Porto, Portugal

Tiago Carvalho
SoftCPS/ISEP & INESC TEC, Porto, Portugal

Luis Miguel Pinho
SoftCPS/ISEP & INESC TEC, Porto, Portugal

Abstract

Real-time systems increasingly rely on parallel execution to meet performance and timing requirements. While several programming languages provide mechanisms for combining real-time and parallel programming, Rust currently lacks dedicated frameworks that address both aspects in an integrated way.

In previous work, we proposed a high-level design of a framework for real-time parallel programming in Rust. In this paper, we describe the design of a prototype implementation of this framework as a Rust library. The prototype provides abstractions for creating and managing real-time threads with priorities, as well as thread pools that enable structured parallel execution while respecting priority-based scheduling.

We describe the architecture of the prototype, its implementation and illustrate its use through examples. This implementation demonstrates the feasibility of supporting real-time parallel programming patterns in Rust and serves as a foundation for future extensions of the framework.

2012 ACM Subject Classification Computer systems organization → Real-time languages; Software and its engineering → Language features

Keywords and phrases Real-time systems, Parallel programming, Rust

Digital Object Identifier 10.4230/OASIcs.AEiC.2026.5

Funding The work is co-funded by the Portuguese Innovation Agency (ANI) through the COMPETE 2030 and NORTE 2030 Programmes, with the support of the European Regional Development Fund (ERDF), within projects GenerIoT (COMPETE2030-FEDER-00355700) and HomePot (NORTE2030-FEDER-01241500).

Acknowledgements The authors would like to thank the anonymous reviewers for their helpful comments and feedback.

1 Introduction

Real-time systems are a challenging research area with applications ranging from the automotive industry to robotics and the Internet of Things. These systems differ from general-purpose systems due to the importance of time-related non-functional requirements, such as deadlines and bounded execution intervals, usually running in resources constrained environments [21].

The development of real-time systems has faced limitations when relying solely on sequential execution models [7]. The widespread adoption of multi-core processors has become the primary approach for increasing computational performance. However, exploiting these architectures efficiently requires software capable of expressing and managing parallel execution. Sequential programming models do not scale well to multi-core platforms and therefore fail to fully utilize the available processing resources. At the same time, the

expansion of real-time systems into domains involving close human–machine interaction introduces even stricter timing requirements, further increasing the need for efficient parallel real-time programming models [1, 15].

Real-time parallel programming has been explored in several programming languages. In C/C++, solutions often rely on extensions or directive-based models such as OpenMP [19], while Ada includes language-level support for concurrency and real-time scheduling. Despite these efforts, the programmability of parallel real-time systems remains an active research topic [1, 15].

Rust [13] is an emerging systems programming language that provides strong guarantees regarding memory safety and concurrency correctness. Its ownership-based memory model allows many classes of errors commonly found in low-level languages to be detected at compile time. These characteristics make Rust an appealing candidate for the development of critical and real-time systems [10]. Rust offers several libraries for parallel programming and there is ongoing work on real-time execution support. However, there is still a lack of frameworks that combine both aspects in a unified programming model.

In previous work [4], we proposed the design of a framework aimed at supporting real-time parallel programming in Rust. In this paper, we present an initial proposal for such framework. The paper is structured as follows. The next section provides some background and related work for parallel programming models in real-time systems. Afterward, Sections 3 and 4 provide the design and architecture of the proposed framework, followed by the programming interface in Section 5 and internal execution and runtime behaviour in Section 6. Finally, Section 7 provides the results of experiments done with the proposed implementation, and Section 8 draws the final conclusion and potential future work.

2 Parallel Programming for Real-Time Systems

Programming models for real-time systems typically provide abstractions that capture key elements of the real-time execution model [1]. These abstractions include the notion of time, the specification of real-time tasks and their scheduling properties, mechanisms for communication and synchronization between tasks, and control over how computations are mapped onto the underlying execution platform.

Two main approaches have traditionally been used to support these requirements. The first relies on general-purpose sequential programming languages, where concurrency and real-time behaviour are implemented through libraries, operating system primitives, and runtime support. This approach is commonly used with the C programming language [9] together with real-time operating systems and POSIX real-time extensions [11]. The second approach consists of using programming languages where concurrency and real-time constructs are provided as first-class language features, such as in Ada [2] or in Real-Time Java [6].

When considering both concurrency and parallelism, programming real-time systems must address more challenges. In addition to defining concurrent tasks that represent the structure of the application, the programming model must also support the expression of parallelism within the algorithms executed by these tasks. As a result, the programming model must combine mechanisms for defining concurrent activities with abstractions that enable parallel execution inside these activities.

Furthermore, mechanisms are required to control how parallel computations are mapped onto scheduling entities. Although mapping and scheduling are typically handled by the runtime system or the operating system, programming models for real-time parallel systems must expose sufficient control to allow developers to influence these decisions. This results in

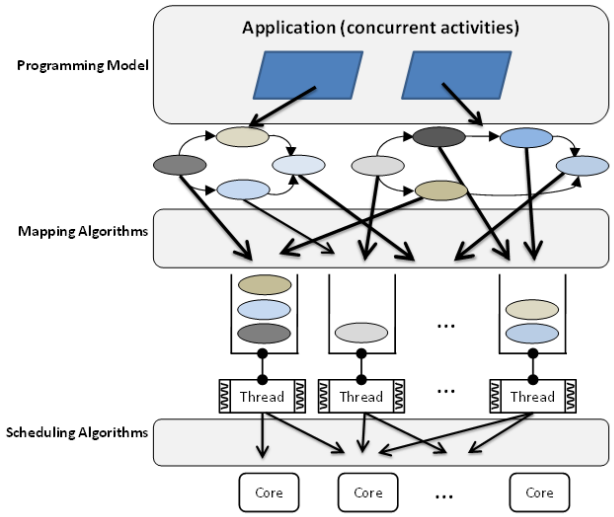

Figure 1 Vertical stack of mapping and scheduling parallel computation (adapted from [16]).

an integrated vertical stack where the programming model, runtime, and operating system cooperate to manage the mapping and scheduling of parallel computations, as illustrated in Figure 1 [16]: application code is structured as graphs of tasks, which are mapped and executed by underlying threads, themselves scheduled by the operating system. Such control may be exposed through runtime libraries and system calls, or also integrated into the programming model itself.

Despite research efforts, the programmability of parallel real-time systems is still challenging. Among the existing approaches, two representative solutions are particularly relevant: the use of a restricted OpenMP tasking model for real-time systems and the parallel programming capabilities integrated into the Ada language.

OpenMP [5] is a widely adopted API for shared-memory parallel programming, extensively used in high-performance computing. Its applicability has gradually expanded to embedded and critical computing domains, such as video processing in autonomous driving systems. Although OpenMP already provides mechanisms for expressing parallelism, it does not natively support key real-time scheduling properties such as deadlines or worst-case execution time constraints. Serrano et al. [19] proposed an OpenMP profile targeting critical real-time systems, introducing restrictions and extensions to make the tasking model more suitable for real-time execution. While this work establishes important foundations, further research is still needed, particularly regarding the integration of multiple parallel real-time activities and the specification of real-time scheduling properties.

Ada [2] is a high-level programming language designed for safety-critical and real-time systems. It provides built-in support for concurrency through its tasking model, as well as mechanisms for real-time scheduling and synchronization. More recently, Ada has incorporated constructs that support parallel programming [8], allowing developers to express parallel execution within the language while preserving its real-time semantics. As a result, Ada represents a language-driven approach integrating real-time and parallel programming.

Although both OpenMP and Ada adopt fine-grained parallel models, they originate from different communities and design perspectives. OpenMP emerged from the high-performance computing community and provides a comprehensive and flexible parallel programming model. In contrast, Ada was developed primarily for safety-critical and real-time applications

and therefore offers extensive support for real-time execution. Consequently, integrating real-time and parallelism follows different directions in these technologies: OpenMP attempts to incorporate real-time properties into an existing parallel programming model, whereas Ada introduces parallel constructs into a language that already supports real-time concurrency [14].

Beyond these two approaches, Maia et al. [12] proposed an approach combining the Java fork/join model with the Real-Time Specification for Java, which was not followed up. Schmid et al. [17] performed a preliminary analysis of real-time execution in frameworks such as Embedded Multicore Building Blocks, Intel Threading Building Blocks, and High Performance ParalleX, concluding that the dynamic nature and limited real-time capabilities of the underlying scheduler in these frameworks makes it a challenge to provide real-time behaviour. A current relevant work is on providing soft real-time behaviour with static memory allocation and probabilistic time bounds, in a manner which is transparent to the programmer [18].

To the best of our knowledge, there is still no framework that integrates structured parallel programming with explicit real-time scheduling control in Rust, with [4] providing some initial ideas.

2.1 Implications for Rust-Based Frameworks

The analysis of the described approaches allows extracting requirements for the design of a framework intended for real-time parallel programming in Rust, to ensure control over parallel program execution and scheduling, with a structured parallel programming approach, while supporting the inherent memory safety rules of Rust.

As a first requirement, the framework needs to provide mechanisms for controlling the scheduling properties of the underlying execution entities (Listing 1, from [4]). In real-time systems it is essential to be able to specify policies for the scheduling process (e.g. periods or priorities), to ensure that critical tasks are completed within the required time period. In this case, a thread is created (*thread::spawn*), with the functional code (*some_task*), and a set of scheduling parameters (*task_sched*). To create these parameters, first a `task_params` object instantiates a *RT_Thread::FixedPriority* struct, specifying priority and periodicity attributes. Then, a *RT_Thread::SchedulingParameters* object specifies the scheduling class (First-in-first-out fixed priority), a core (in this case the thread is not pinned to a core), and the Task parameters.

Listing 1 Example of a thread being spawn with a SCHED_FIFO scheduling class.

```
fn some_task() {...}

fn main(){
  let task_params = RT_Thread::FixedPriority {
    priority: 255,
    period: None,
  };
  let task_sched = RT_Thread::SchedulingParameters {
    class: RT_Thread::SchedulingClass::FIFO,
    core: None,
    params: task_params
  };
  thread::spawn(some_task, task_sched);
}
```

Another requirement is the support of thread pools (Listing 2, from [4]). While some operations might demand the creation of individual threads with specific parameters, fine-grained parallel programming will benefit from the creation of thread pools, where parallel execution can be spawned and managed.

Listing 2 Example of a threadpool with specific scheduling algorithms per thread.

```
fn main(){

  let task_params = RT_Thread::ConstantBandwitdhServer {
     runtime: Duration::from_millis(2),
     period: Duration::from_millis(50),
     deadline: Duration::from_millis(10),
  };

  let task_sched = RT_Thread::SchedulingParameters {
     class: RT_Thread::SchedulingClass::CBS,
     params: task_params
  };


  RT_Thread::create_periodic_task(task_sched, body: move | {
     let n_workers = 4;
     let pool = ThreadPool::new_inherited(n_workers);

     let params = vec![];
     let param_0 : RT_Thread::get_current_sched_params();
     param_0.core = 0;
     params.push(param_0);  // inherit parameters for core zero
     params.push(SchedulingParameters { //fixed priority in core 1
        class: SchedulingClass::FIFO,
        core: 1,
        params: FixedPriority {priority: 255}
     });
     params.push( SchedulingParameters { //CBS on core 2
        class: SchedulingClass::CBS,
        core: 2,
        params: ...
     });
     // ...

     let pool = ThreadPool::new_rt(n_workers, params);
     ...
     pool.execute(num_thread, some_task); //in specific thread
  });
}
```

Finally, safety is an important consideration, a fundamental Rust concern. The framework needs to allow developers to write real-time programs without having to resort to "unsafe" operations, a factor important for the validation of real-time systems.

3 Framework Design

In order to support the high-level design goals described previously, the internal architecture of the framework is organized around a set of custom Rust *structs* (Figure 2) that encapsulate both the lifecycle of threads and the associated scheduling logic. Although the user interacts mainly with high-level public abstractions, the library internally coordinates thread creation, job execution, and scheduling through a modular architecture. This separation allows the framework to provide a simple and safe programming interface while still enabling detailed control over execution behaviour.

To achieve this objective, two primary public abstractions are exposed to the programmer: one representing *isolated threads* and another representing *thread pools*. This separation allows the framework to maintain internal control over thread management while still providing users with flexible mechanisms to configure and interact with execution entities. Both abstractions are implemented as public *structs* whose internal state remains private and is managed exclusively by the library. This design ensures that critical thread management data remains encapsulated within the framework, preserving safety guarantees while hiding unnecessary complexity from the user.

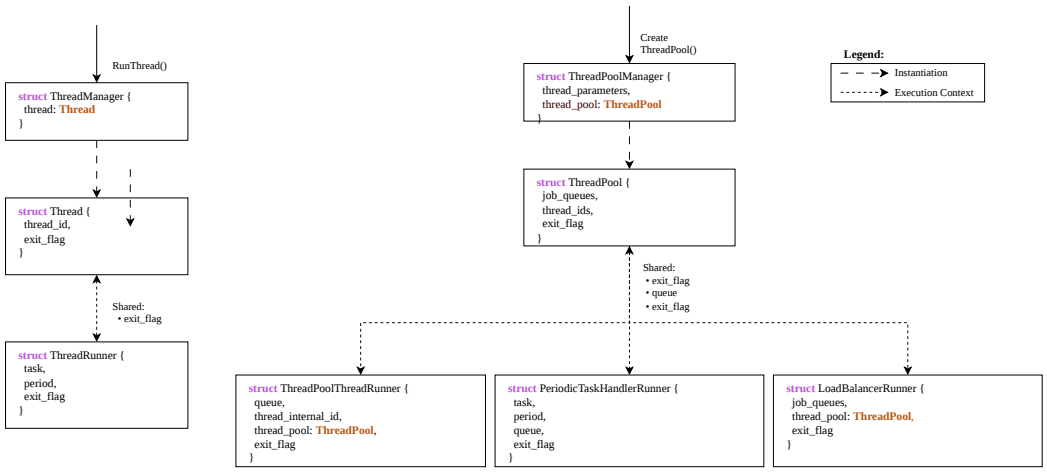

Figure 2 Framework design: Isolated Threads (left) and Thread Pools (right).

3.1 Isolated Threads

For isolated threads (Figure 2, left), the `ThreadManager`, a public object created by the user, internally instantiates (dashed arrow) an additional `Thread` struct, responsible for configuring and managing the thread's execution state. This internal structure stores the information required to manage the thread lifecycle and is owned by the outer abstraction exposed to the user.

Since isolated threads represent a single execution entity, the information required by the framework is relatively limited. The internal structure (a *Thread*) stores the operating system thread identifier and a shared termination flag. This flag is used to signal the thread when it must stop execution and is set when the user invokes the corresponding termination method.

When the thread itself is created, a separate execution context (a *ThreadRunner*) is initialized (dotted arrow) containing the elements required for its operation. These include the task assigned to the thread, the execution period specified by the user (when periodic execution is required), and a reference to the shared termination flag. This mechanism allows the running thread to monitor the flag and terminate gracefully when requested.

3.2 Thread Pools

The design of thread pools (Figure 2, right) follows a similar philosophy of maintaining a simple user-friendly interface, while delegating execution management to internal structures. The public abstraction (`ThreadPoolManager`) stores the parameters defined by the user, together with an internal *ThreadPool* structure responsible for coordinating the worker threads.

The *ThreadPool* structure contains the information necessary to manage a group of worker threads. In particular, it maintains the operating system identifiers of the created threads and several shared flags used to control termination and runtime coordination.

A key design difference from traditional thread pool implementations lies in the job scheduling model. Instead of relying on a single global work queue shared by all worker threads, the framework maintains an individual queue for each thread. This design enables users to explicitly assign tasks to specific threads when required, allowing greater control over execution placement and improving predictability in real-time scenarios.

To support the execution of worker threads within the pool, an additional structure named *ThreadPoolThreadRunner* is necessary. This structure contains the shared termination flags, an internal thread identifier used by the framework, the job queue associated with that worker thread, and a reference to the parent *ThreadPool*. These elements allow each worker thread to manage its assigned workload while maintaining coordination with the rest of the pool.

3.3 Periodic Task Handling

In order to support periodic job execution, the framework introduces a dedicated internal structure named *PeriodicTaskHandlerRunner*. When a periodic task is created, a helper thread is spawned to manage the periodic activation of that task. This approach simplifies the logic executed by the worker threads themselves, as the periodic scheduling responsibilities are handled externally.

The *PeriodicTaskRunner* structure stores the parameters of the periodic task, the queue of the worker thread to which the task should be dispatched, and the shared termination flag used to coordinate safe shutdown of the system.

3.4 Load Balancing

The framework also allows users to submit tasks without explicitly specifying the worker thread on which they should execute. In such cases, task assignment is handled internally by the system.

To support this functionality, the thread pool creates a dedicated internal thread responsible for distributing these tasks across the available workers. This component is implemented through the *LoadBalancerRunner* structure. The structure maintains references to the job queues of all worker threads, the shared termination flag, and the parent *ThreadPool* structure.

Using this information, the load balancer can dynamically assign tasks to worker threads. Additionally, similar to the other thread pool components, it can create periodic task handlers when required.

4 Framework Architecture

The proposed framework builds upon concepts present in existing Rust thread pool libraries, particularly the *ThreadPool* crate [3], while extending them to address the requirements of real-time systems. The framework introduces mechanisms that provide greater control over scheduling attributes and thread management, while maintaining a simplified programming interface for developers.

Several operations that would normally require direct interaction with low-level system interfaces (typically through libraries such as *libc*) are encapsulated within safe abstractions. This design allows developers to access system-level functionality without explicitly writing unsafe code, thereby reducing the risk of memory management errors.

The framework is organized into four main modules (Figure 3). The *lib* module exposes the public interface and contains the methods directly accessible to the user. The remaining modules implement the internal logic of the system.

The *thread_manager* module contains the data structures and methods responsible for creating and managing threads and thread pools. The *thread_runner* module implements the execution cycle of the threads, including the mechanisms responsible for executing submitted

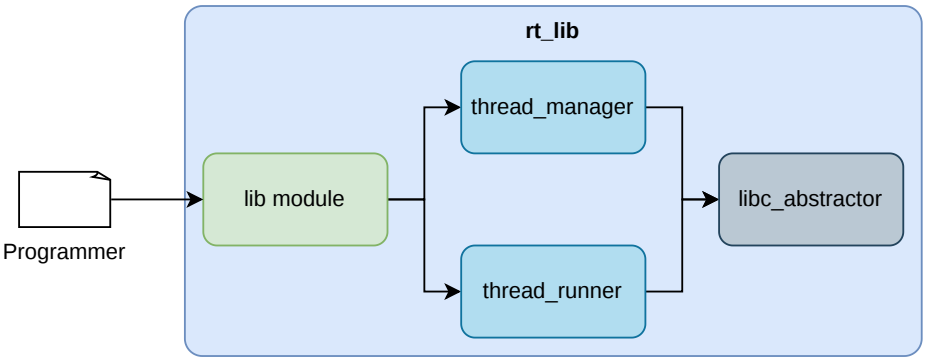

Figure 3 Overview of the framework architecture.

tasks and enforcing periodic behaviour. Finally, the *libc_abstractor* module encapsulates all interactions with the external *libc* library and contains the unsafe code required to access low-level system functionality.

By isolating these operations in a dedicated module, the framework maintains cleaner and safer code across the remaining components. This modular architecture helps organize the framework's functionality while keeping complex or low-level operations hidden from the user.

5 Programming Interface

This section describes the mechanisms provided by the framework to create and manage threads from the developer's perspective.

5.1 Scheduling Parameters

To enable developers to configure real-time scheduling properties, the framework introduces the *SchedulingParameters* structure. This structure allows the programmer to define the attributes required for a real-time thread, namely the scheduling class, processor affinity, and scheduling priority (see Listing 3).

The scheduling class is defined using the *SchedulingClass* enumerator, which includes the real-time scheduling policies supported by the framework, namely *FIFO* and *Round-Robin*. The scheduling class and processor affinity are optional parameters. If these parameters are not provided, the operating system defaults are used.

Listing 3 Example of scheduling parameter definition.

```
params = framework::SchedulingParameters {
    class: Some(SchedulingClass::FIFO),
    core: Some(0),
    priority: 9,
};
```

5.2 Isolated Threads

The framework provides the *ThreadManager* abstraction for creating isolated threads (Listing 4). The method *create_running_thread* receives the scheduling parameters, an optional execution period, and the job to be executed.

If a period value is provided, the task is executed periodically. Otherwise, the task is executed once and the thread terminates afterward.

Listing 4 Creating an isolated thread.

```
params = framework::SchedulingParameters { ... };

job = move || {
    ...
};

period_ms = Some(1500);

thread = match ThreadManager::create_running_thread(params, period_ms, job) {
    Ok(thread) => thread,
    Err(e) => ...
};
```

The method returns a *Result* type containing either the created *Thread* structure or an error message. The returned structure provides two blocking management operations: *exit_running_thread*, which terminates the thread gracefully, and *kill_running_thread*, which terminates it immediately.

Using the exit action is the safest approach to stop a thread, allowing the thread to finish its work before stopping. However, as there might be circumstances where an abrupt stop is necessary, the kill action can be performed. Due to how evasive kill action is, it should be a last resort and carefully used, in order to mitigate possible safety issues.

5.3 Thread Pools

Thread pools are created using the *ThreadPoolManager* abstraction (Listing 5). The method *new_rt* allows the creation of a pool with a specified number of threads and a set of scheduling parameters.

Listing 5 Creating a thread pool with custom parameters.

```
params = vec![
    framework::SchedulingParameters { ... },
    framework::SchedulingParameters { ... },
    framework::SchedulingParameters { ... }
];

number_of_threads = 3;

pool = match ThreadPoolManager::new_rt(number_of_threads, params) {
    Ok(thread_pool) => thread_pool,
    Err(e) => ...
};
```

The framework also supports inheriting scheduling parameters from the parent execution context (Listing 6).

Listing 6 Creating a thread pool with inherited parameters.

```
pool = match ThreadPoolManager::new_inherited(3) {
    Ok(thread_pool) => thread_pool,
    Err(e) => ...
};
```

5.4 Task Execution

Once a thread pool has been created, tasks can be submitted using the *execute* method (Listing 7). The method receives an optional thread identifier (its internal index), an optional execution period, and the task closure.

Listing 7 Submitting tasks to a thread pool.

```
params = vec![
    framework::SchedulingParameters { ... }
];

number_of_threads = 2;

pool = ThreadPoolManager::new_rt(number_of_threads, params).unwrap();

job = ...;
thread = Some(0);

pool.execute(thread, None, job);

period_ms = Some(1000);

pool.execute(None, period_ms, job);
```

When a thread index is provided, the task is executed on that specific thread. Otherwise, the framework dynamically assigns the task to the first available thread. Note that a thread index is essentially determined by the order they are configured in the thread pool. For instance, the first thread configured in Listing 7 is accessible by index 0, and so forth.

5.5 Thread Pool Management

To terminate threads in the pool, the framework provides the methods *exit_threads* and *kill_threads* (Listing 8). The *kill_threads* operation immediately terminates all worker threads.

Listing 8 Managing thread pool execution.

```
sched_params_0_2 = pool.get(Some(vec![0,2]));

pool.exit_threads(false);

pool.exit_threads(true);
```

When the argument of *exit_threads* is set to *false*, only threads without periodic tasks are terminated. When set to *true*, all threads are terminated after finishing their current execution iteration.

5.6 Inspection Utilities

The framework also provides helper methods that allow developers to inspect thread attributes without interacting directly with unsafe code. These include:

- *get_policy*
- *get_cpu*
- *get_priority*
- *get_tid*

These functions return the scheduling policy, processor affinity, priority, and identifier of the thread, respectively, facilitating debugging and validation of real-time applications.

6 Internal Execution and Runtime Behaviour

In order to provide the level of simplicity exposed in the programming interface, a significant portion of the complexity involved in thread creation and management is handled internally by the framework. This includes low-level interactions with the *libc* library, management of scheduling attributes, and coordination of the thread lifecycle.

The *lib* module contains the public-facing structures *ThreadPoolManager* and *ThreadManager*, together with the methods exposed to the user. Internally, this module interacts with the *thread_manager* module, which contains the internal *Thread* and *ThreadPool* structures responsible for storing and managing the data associated with created threads and thread pools. These structures are not accessible to the user and encapsulate the internal state of the runtime.

Finally, the *thread_runner* module contains the execution logic that runs within each thread created by the framework. This module implements the runtime behaviour of isolated threads, worker threads in thread pools, load-balancing threads, and periodic task handlers. The following sections describe the behaviour of these components in greater detail.

6.1 Isolated Thread Execution

When the user invokes the *create_running_thread* method to create an isolated thread, the framework performs several configuration steps internally. The process begins by initializing the thread attributes using the POSIX *pthread_attr_t* structure. This structure stores all configuration parameters associated with the thread and ensures that scheduling attributes are explicitly defined rather than inherited from the parent thread.

If the user specifies a processor core in the scheduling parameters, the framework sets the corresponding CPU affinity, effectively pinning the thread to that core. The scheduling policy is then configured according to the scheduling class defined by the user. This value is translated into the corresponding *libc* constant and applied to the thread attributes. Finally, the priority associated with the thread is set according to the chosen scheduling policy.

After configuring the attributes, the framework prepares the job provided by the user. The job, represented as a closure, is packaged together with the optional execution period and a shared termination flag used to signal when the thread should exit. This structure is allocated on the heap and converted into a raw pointer so that it can be safely passed to the system-level thread creation routine.

The actual thread is created using the *libc* thread creation function. If the operation succeeds, the thread identifier is stored in the internal *Thread* structure. Otherwise, the framework releases any allocated resources and returns an error to the caller. Once created, the thread begins executing the *single_thread* routine, which represents the main execution loop of an isolated thread. At the beginning of each iteration, the thread checks the shared termination flag. If the flag is set, the loop terminates and the thread exits gracefully. Otherwise, the thread executes the user-defined job.

After executing the job, the framework checks whether a periodic interval was specified. If the job is periodic, the thread sleeps for the defined duration before starting the next iteration. If no period is defined, the thread exits after executing the job once. These routines are implemented within the *thread_runner* module.

To ensure safe concurrent access to shared data such as job closures and termination flags, synchronization primitives are used internally. These mechanisms guarantee that access to shared resources remains safe even when multiple threads are executing concurrently. The methods *exit_running_thread* and *kill_running_thread* are also implemented within

the internal *Thread* structure. The former sets the shared termination flag, allowing the thread to terminate safely after completing its current job. The latter uses the stored thread identifier to terminate the thread immediately.

From the user's perspective, the result is a fully configured real-time thread executing the provided job, either periodically or as a single execution, without requiring the use of unsafe code or direct interaction with low-level system interfaces.

6.2 Thread Pool Creation

The implementation of thread pools requires significantly more coordination due to the presence of multiple worker threads and the need to distribute tasks among them. When a thread pool is created using either the *new_rt* or *new_inherited* methods, the framework first constructs a collection of *SchedulingParameters*. These parameters are then passed to the thread creation routines implemented in the *thread_runner* module.

For each worker thread, the framework creates a dedicated task queue. These queues store the jobs assigned to each thread and are consumed by the worker threads during execution. In addition to the per-thread queues, a global queue is also created to store tasks that are submitted without an explicit thread assignment.

Alongside these queues, the framework creates all worker threads according to the user-defined parameters. Additionally, an extra thread is created internally to act as a load balancer. This thread is responsible for monitoring the global queue and redistributing tasks to worker threads whose queues become empty.

All runtime data associated with a thread pool is stored within the internal *ThreadPool* structure. This structure contains the task queues, the identifiers of all active threads, and shared flags used to control thread termination.

Figure 4 illustrates how threads are created and associated with their respective queues within a thread pool.

6.3 Load Balancer Behaviour

The load balancer thread executes the *load_balancer_thread* routine. This routine continuously checks whether the framework has requested termination of periodic tasks. If the termination flag is not set, the thread inspects the global task queue.

If no jobs are available, the thread sleeps briefly before repeating the loop to avoid unnecessary CPU usage. When a job is present, the load balancer searches for an available worker thread with an empty queue. If such a thread is found, the job is transferred from the global queue to the corresponding worker queue.

If no worker thread is immediately available, the load balancer retries the operation during the next iteration. This mechanism allows tasks to be distributed dynamically without requiring the user to manually manage workload distribution (a work-stealing scheduler could have been implemented, however this approach allows an easier analysis of interference). It is nevertheless to be used when threads have the same scheduling parameters.

6.4 Worker Thread Behaviour

Worker threads execute the *run_thread* routine. This function represents the core execution behaviour of worker threads within a thread pool.

At the start of each loop iteration, the thread checks whether it should terminate. The termination condition depends on whether the thread is currently executing periodic tasks or non-periodic tasks, as different flags are used for each case. If the thread is not scheduled for

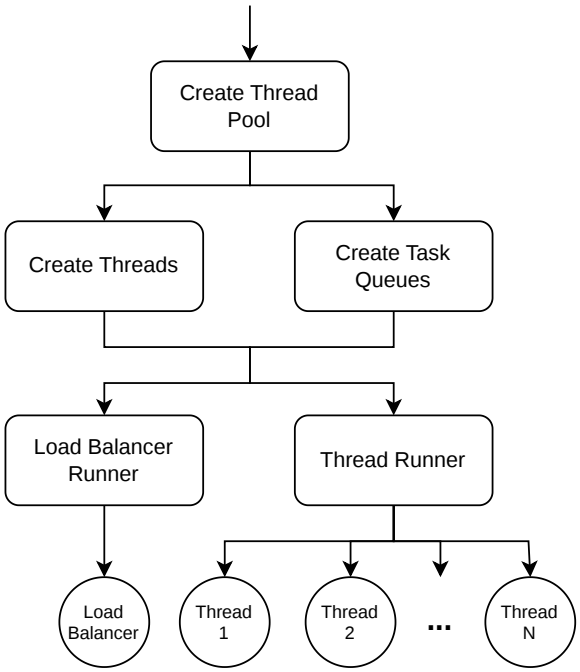

Figure 4 Creation of threads in the ThreadPool.

termination, it checks its task queue. If the queue is empty, the thread sleeps briefly before repeating the loop. If a job is available, the thread retrieves it and executes the associated closure.

When the job corresponds to a periodic task, the framework verifies whether a handler thread has already been created for it. If not, a periodic handler thread is spawned to manage future activations of that task.

6.5 Periodic Task Handling

Periodic tasks are managed by dedicated handler threads created through the method *spawn_periodic_handler_threads* (as illustrated in the first `execute` invocation, on the left side of Figure 5). Each periodic task is associated with a *PeriodicTaskHandlerRunner* structure containing the task information, the identifier of the worker thread responsible for execution, and the shared termination flag for periodic threads.

The handler thread executes the *periodic_handler_thread* routine. Since the first execution of the task is performed by the worker thread, the handler begins by sleeping for the configured period. After waking, it checks whether the periodic termination flag has been set. If not, it re-inserts the task into the worker thread's queue and repeats the loop. This mechanism ensures that periodic tasks are re-scheduled without blocking worker threads or interfering with other tasks.

Figure 5 illustrates how tasks are handled within the thread pool, including periodic task handling (left), single shot task assigned to a particular thread (middle) and a (thread pool) global queue task (left).

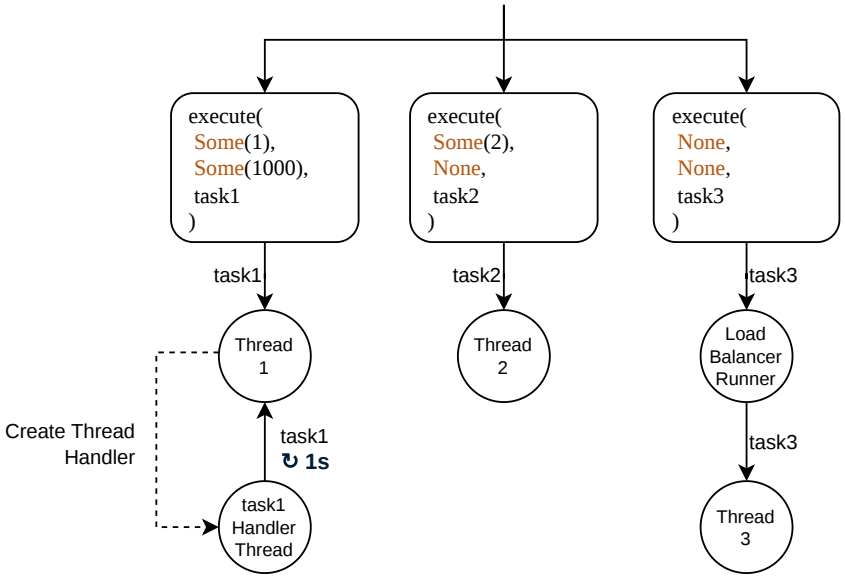

Figure 5 Handling of tasks in the ThreadPool. From left to right: periodic task, task in specific thread, task where load balancer decides where to execute.

6.6 Thread Pool Shutdown

To terminate threads gracefully, the framework provides the *exit_threads* method. Internally, this method accesses the *running_threads_id* collection stored in the *ThreadPool* structure. This collection contains the identifiers of all active threads, including worker threads, the load balancer, and periodic handler threads.

When the user requests termination of all threads, the framework sets the appropriate termination flags and iterates through the collection, joining each thread. Joining ensures that the main program waits until each thread finishes execution before continuing.

If the user requests termination only of threads not executing periodic tasks, only the corresponding termination flag is set. In this case, periodic threads remain active. An alternative method, *kill_threads*, is also available. This method iterates through the same thread identifier collection but immediately terminates the threads using system-level operations. Although this approach is faster, it may lead to resource leaks and therefore contradicts the safety philosophy of Rust.

6.7 Interaction with *libc*

Several operations within the *thread_runner* module require direct interaction with the *libc* library. Since *libc* originates from the C programming language, these operations cannot benefit from Rust's usual memory safety guarantees and therefore must be executed inside *unsafe* blocks.

To maintain code clarity and minimize the use of unsafe constructs across the framework, all interactions with *libc* are isolated within the *libc_abstractor* module. This design keeps the remaining modules free from unsafe code and improves maintainability and readability of the overall implementation.

7 Experimental Evaluation

The proposed framework was designed with real-time programming requirements in mind, where timing guarantees, responsiveness, and predictability are critical for system correctness. Therefore, evaluating the performance of the framework is essential to determine its suitability for time-sensitive applications.

A set of experiments was conducted to analyse the behaviour of the framework under different execution scenarios (full description can be found in [20]). The evaluation focused on key metrics such as thread creation time, task execution duration, scheduling latency, and task distribution across worker threads. To collect these measurements, the experimental examples were instrumented with timestamp collection using the Rust *std* library, namely the *Instant* type, which allows precise measurement of elapsed time. Synchronization primitives such as *Mutex* were used to safely propagate timing information from worker threads to the main program. Basic statistical metrics including minimum, maximum, mean, and standard deviation were computed using a Rust statistics library.

Each experiment was executed one hundred times in order to obtain a sufficiently large sample set for analysis. The experiments were conducted on a virtual machine running Ubuntu with four CPU cores and 8 GB of RAM, hosted on a desktop machine equipped with an Intel Core i7-9700K processor and 16 GB of RAM. Although the use of virtualization may introduce additional scheduling variability due to resource sharing, it also provides reproducibility and isolation. Consequently, the results presented here should be interpreted as conservative estimates of performance; improved timing behaviour is expected when executing on dedicated hardware.

Several scenarios were evaluated. The first experiment measured the overhead of creating isolated threads and executing simple tasks. Results showed that thread creation times were consistently below one millisecond on average, while task execution durations closely matched the expected workload timing. Some variability was observed due to the virtualized environment, but most executions remained within predictable bounds.

The second experiment evaluated hierarchical thread creation, where a parent thread dynamically spawned multiple child threads executing periodic tasks. The results showed that child thread creation overhead was very small, and although the coordination performed by the parent thread introduced additional execution time, the framework handled hierarchical thread management efficiently.

A third experiment focused on the thread pool implementation. The results demonstrated that thread pool initialization incurs slightly higher overhead due to the creation of multiple worker threads and internal queues. However, once the pool is created, job dispatch latency becomes extremely small, typically measured in microseconds, indicating that the framework can efficiently reuse worker threads to execute tasks.

A further experiment evaluated task distribution across worker threads when tasks are submitted without explicitly specifying a target thread. The load balancing mechanism showed a relatively even distribution of tasks across available worker threads, confirming the effectiveness of the internal queueing and load balancing strategy. Execution times remained stable even under increased workload.

Finally, a scenario was evaluated in which thread pools were created dynamically from within already running threads using inherited scheduling parameters. While this configuration introduces additional initialization overhead, the results indicate that such dynamic creation remains feasible and reasonably efficient for runtime task management.

Overall, the experimental results indicate that the framework introduces very low overhead for thread creation and task dispatching, while maintaining predictable execution behaviour in most scenarios. Although some timing variability was observed due to the virtualized execution environment, the median values across all experiments remained stable and consistent with the requirements of real-time systems.

8 Conclusions

Real-time systems increasingly require the ability to exploit parallel hardware while maintaining strict control over timing and scheduling behaviour. Although Rust has gained significant attention as a systems programming language due to its strong safety guarantees and modern concurrency model, support for structured real-time parallel programming remains limited.

This paper presented the design and prototype implementation of a framework for real-time parallel programming in Rust. The framework introduces abstractions for creating and managing real-time threads with configurable scheduling parameters, as well as thread pools that support structured parallel task execution. The design aims to provide developers with explicit control over priorities, processor affinity, and task placement while preserving the safety guarantees offered by Rust.

The prototype implementation demonstrates that it is feasible to combine real-time scheduling mechanisms with parallel programming constructs within the Rust ecosystem. The presented architecture hides low-level complexity from the user while still exposing the necessary controls required for real-time applications. Experimental results indicate that the framework introduces low overhead for thread creation and task dispatch, suggesting that the approach is suitable for time-sensitive environments.

This work represents an initial step toward providing practical support for real-time parallel programming in Rust. Future work will focus on extending the framework with additional programming abstractions, scheduling mechanisms and synchronization primitives. Furthermore, integration with existing real-time operating systems, such as Zephyr, will be explored to allow to further evaluate the overheads and predictability of the framework implementation.

References

1 Alan Burns and Andrew J Wellings. *Real-time systems and programming languages: Ada 95, real-time Java, and real-time POSIX*. Pearson Education, 2001.

2 Alan Burns and Andy Wellings. *Concurrent and real-time programming in Ada*. Cambridge University Press, 2007.

3 Brian Burton. Rust threadpool library. `https://github.com/rust-threadpool/rust-threadpool`, 2024. Accessed: 2026-03-10.

4 Tiago Carvalho, Hugo Silva, and Luís Miguel Pinho. A real-time parallel programming approach for rust. *Ada Lett.*, 43(2):57–61, 2024. `doi:10.1145/3672359.3672366`.

5 Rohit Chandra, Leo Dagum, David Kohr, Ramesh Menon, Dror Maydan, and Jeff McDonald. *Parallel programming in OpenMP*. Morgan kaufmann, 2001.

6 Peter Dibble. *Real-time Java platform programming*. Prentice Hall Professional, 2002.

7 Mark D Hill and Michael R Marty. Amdahl's law in the multicore era. *Computer*, 41(7):33–38, 2008. `doi:10.1109/MC.2008.209`.

8 ISO/IEC. Information Technology – Programming Languages – Ada, ISO/IEC 8652:2023, 2023. Ada 2022 Language Standard.

9 Brian W Kernighan and Dennis M Ritchie. *The C programming language*. Pearson Education Asia, 2002.

10 Steve Klabnik and Carol Nichols. *The Rust programming language*. No Starch Press, 2023.

11 Donald Lewine. *POSIX programmers guide*. O'Reilly Media, Inc., 1991.

12 Cláudio Maia, Luís Nogueira, and Luís Miguel Pinho. Combining rtsj with fork/join: a priority-based model. In *Proceedings of the 9th International Workshop on Java Technologies for Real-Time and Embedded Systems*, pages 82–86, 2011. doi:10.1145/2043910.2043924.

13 Nicholas D Matsakis and Felix S Klock. The rust language. *ACM SIGAda Ada Letters*, 34(3):103–104, 2014.

14 Luis Miguel Pinho. Real-time parallel programming for homogeneous multicores. In *2024 IEEE 14th International Symposium on Industrial Embedded Systems (SIES)*, pages 1–9, 2024. doi:10.1109/SIES62473.2024.10768043.

15 Luis Miguel Pinho, Vincent Nélis, Patrick Meumeu Yomsi, Eduardo Quiñones, Marko Bertogna, Paolo Burgio, Andrea Marongiu, Claudio Scordino, Paolo Gai, Michele Ramponi, et al. P-socrates: A parallel software framework for time-critical many-core systems. *Microprocessors and Microsystems*, 39(8):1190–1203, 2015. doi:10.1016/J.MICPRO.2015.06.004.

16 Luis Miguel Pinho, Eduardo Quinones, Marko Bertogna, Andrea Marongiu, Vincent Nelis, Paolo Gai, and Juan Sancho. *High-Performance and Time-Predictable Embedded Computing*. River Publishers, Wharton, TX, USA, 2018.

17 Michael Schmid, Florian Fritz, and Juergen Mottok. Parallel programming in real-time systems. In *ARCS Workshop 2019; 32nd International Conference on Architecture of Computing Systems*, pages 1–7. VDE, 2019.

18 Michael Schmid, Florian Fritz, and Jürgen Mottok. Fine-grained parallelism framework with predictable work-stealing for real-time multiprocessor systems. *Journal of Systems Architecture*, 124:102393, 2022. doi:10.1016/j.sysarc.2022.102393.

19 Maria A Serrano, Sara Royuela, and Eduardo Quiñones. Towards an openmp specification for critical real-time systems. In *Evolving OpenMP for Evolving Architectures: 14th International Workshop on OpenMP, IWOMP 2018, Barcelona, Spain, September 26–28, 2018, Proceedings 14*, pages 143–159. Springer, 2018. doi:10.1007/978-3-319-98521-3_10.

20 Hugo Silva. Real-time parallel programming in rust. Master's thesis, Instituto Superior de Engenharia do Porto, 2025. Accessed: 2026-03-10. URL: http://hdl.handle.net/10400.22/31230.

21 Paul T Ward. *Structured development for real-time systems: Vol. I: Introduction and tools*. Pearson Education, 1986.