POLITECNICO
MILANO 1863

SCUOLA DI INGEGNERIA INDUSTRIALE
E DELL'INFORMAZIONE

Tesi di Laurea Magistrale in
Computer Science and Engineering
Ingegneria Informatica

Priority aware asynchronous programming for embedded real-time systems

Author:
Luca Romanò

Student ID:
10656514

Advisor:
Prof. Federico Terraneo

Co-Advisor:
Tomas Antonio Lopez

Academic Year:
2025-2026

Dedicated to my family.

Abstract

In this thesis, we worked on making asynchronous calls more efficient, targeting real-time operating systems suited for embedded systems, characterized by their low resources. We applied our research on Miosix, a real-time operating system developed at Politecnico di Milano.
In most general-purpose asynchronous runtime environments, like the one implemented by the standard library of C++, when an asynchronous call is scheduled, a thread is created, allocating memory space on ram, and then executed. In case of a high number of asynchronous calls, especially if resources are low, this standard approach can lead to a high resource consumption and an overall slow down of the system.
Working on a real-time operating systems means scheduling tasks based on their priority, calculated in relation to some task deadline or period, where the ones with the closest deadline needs to be scheduled earlier to ensure the correct functioning of the system.
Our work centers around the implementation of an event-based scheduler that supports the scheduling of asynchronous calls based on priority.
The main idea of our approach is to create an executor that instantiate threads for executing asynchronous calls, and it executes them based firstly on their priority, and then considering a FIFO order between the ones with the same priority.
Our work is focused on avoiding the creation of new memory space for every asynchronous call, and guaranteeing that the number of executing tasks will not be higher than the number of CPU cores. As a result, we have lower memory usage and faster execution time, increased determinism, and absence of priority inversions.

Keywords: real-time operating systems, embedded systems, asynchronous calls, scheduling algorithms

Abstract in lingua italiana

In questa tesi, abbiamo lavorato per rendere piu efficienti le chiamate a funzioni asincrone in un contesto di sistemi operativi real-time applicati su sistemi embedded, che sono caratterizzati dalle scarse risorse hardware.
Abbiamo applicato il nostro lavoro su Miosix, un sistema operativo real-time sviluppato al Politecnico di Milano.
Nella maggior parte degli ambienti di esecuzione asincroni, come quello implementato dalla libreria standard di C++, quando viene chiamata una funzione asincrona, viene creato un thread che alloca spazio nella memoria RAM e poi viene eseguito. Nel caso di un numero elevato di chiamate asincrone, specialmente se le risorse sono limitate, questo approccio può portare a un elevato consumo di risorse e a un rallentamento complessivo del sistema.
In un sistema operativo real-time lo scheduling delle task viene eseguito in base alla loro priorità, data dalla loro deadline o dal loro periodo, e quelle con priorità più alta devono essere esuguite prima delle altre per asscurarsi il corretto funzionamento del sistema.
Il nostro lavoro si concentra sull'implementazione di uno scheduler di eventi che supporta la schedulazione di chiamate asincrone in base alla priorità.
L'idea principale del nostro approccio è creare un esegutore che istanzia dei thread per l'esecuzione delle chiamate asincrone e le esegue prima in base alla loro priorità e poi seguendo un ordine FIFO tra quelle con la stessa priorità.
In questo modo evitiamo di creare nuovo spazio di memoria per ogni chiamata asincrona e garantiamo che il numero di task in esecuzione non sia superiore al numero di core della CPU. Come Risultato otteniamo un minore utilizzo della memoria e tempi di esecuzione più brevi, un maggiore determinismo e l'assenza di inversioni di priorità.

Parole chiave: sistemi operativi real-time, sistemi embedded, chiamate asincrone, algoritmi di scheduling

Contents

1 | Introduction

The goal of the thesis is to explore efficient implementations of asynchronous programming paradigms in C++ for embedded real-time operating systems.
Real-time systems have the goal of executing every task under some defined time constraint. This feature is crucial because the missing of a deadline can lead to catastrophic consequences. An example can be the pacemaker device, where a deadline miss can lead to the death of the wearer.
Real-time operating systems (RTOS) are designed to support real-time systems by providing deterministic scheduling policies, allowing the designer to statically determine the temporal behavior of the system. This is achieved by RTOS, with scheduling algorithms that translate the tasks deadline in a priority, and execute their processes based on that.
Working on a cyber-physical system, real-time systems are usually embedded systems, that are characterized by a scarse hardware resources, compared with a modern computer. This implies that techniques used in modern computers cannot be applied in embedded systems because they would necessitate more hardware resources.
Our work goes in this direction, working on asynchronous calls. Considering the implementation of the C++ standard library, for every asynchronous call, a thread is created to execute it and leading potentially to high resources consumptions if multiple calls are scheduled. The goals of our work are that asynchronous calls are executed in priority order, respecting the real-time constraints set upon their parent thread, assuring that a limited number of calls are in execution, and that the number of created threads remains low. We applied our research on Miosix, a real-time operating system that supports development with C/C++, multi-core scheduling, and full compatibility with the POSIX standard.
To achieve our goal, we implemented a scheduler that allocates predefined memory space to execute asynchronous calls and executes them in a loop. The number of schedulers in the system is equal to the number of CPU cores multiplied by the number of priority levels. In particular, for every priority level, as many schedulers are instantiated as the number of CPU cores, so that the operating system scheduler can manage the execution of the different schedulers based on their priority, giving precedence to the ones

with higher priority. and allow maximum parallelism between the scheduled asynchronous calls. Starting with this scheduler, we implemented two different approaches, one based on a standard threadpool, and the other based on the algorithm of lazy task creation.

1.1. Contributions

The thesis presents two implementations for using asynchronous calls in Miosix with the goal of reducing memory usage and improving execution speed with respect to the implementation of asynchronous calls of the C++ standard library. The implementations, working on a real-time operating system, support the scheduling of asynchronous calls with priority that is inherited from the thread that performs the asynchronous calls.
The work also has the goal of maintaining a similar interface of asynchronous calls in the C++ standard library, so that the utilizer can use them as if he is working with the standard asynchronous call interface.
Our approach result in a significant improvement of asynchronous calls in Miosix, expecially in the scenario where a thread perform a high number of asynchronous calls, where we have a faster execution time, and a drastic reduction of memory usage that is higly valued in a real-time operating system.

1.2. Thesis structure

The thesis is divided into four chapters: In the background chapter, are reported a brief introduction of real-time operating systems, an overview about Miosix, focusing on the relevant parts for our needs, and an explanation about how asynchronous function works in the standard approach. It also includes other techniques that were taken into consideration for reaching our purposes like compiler optimizations, and the programming style: continuation passing style.
Then, it includes three state of the art approaches that try to solve our problem or part of it. All three approaches can give a partial solution to our problem, because they cannot be applied entirely in our context.
In the design and implementation chapter, we start presenting the implementation of the priority event queue, which is the basis of our work. Then we explain how this approach is used for the implementation of the threadpool and lazy task creation solutions that we implemented for asynchronous calls. Lastly, we explain about a solution that we tried to implement, based on compiler code transformation, but it revealed non applicable to our context.
In the chapter tests and experimental results, we first show the functionality of the priority

event queue. Then we demonstrate that our approaches improve the execution time and memory usage compared with the asynchronous calls of the C++ standard library, both in Miosix and Linux, executing different test cases.

2 | Background

2.1. Real-time operating systems

Real-time systems [1] are computing systems that must react within precise time constraints to events in the environment. In these systems, correctness depends not only on producing the correct output, but also on producing it at the right time. A late response could be useless or even dangerous. Real-time operating systems (RTOS) are implemented to meet the requirements of real-time systems. They differ from general-purpose operating systems (GPOS) [2] where the performance is measured in terms of throughput, while in an RTOS, the main metric of concern is whether tasks can be completed within their predetermined deadline. The main benefits of RTOS lie in the predictability of the execution time of any given task, where developers can ensure that specific tasks are executed within a specific time frame. These systems are widely used in safety-critical applications such as drones, industrial machines, and medical devices, where the miss of a deadline can lead to catastrophic consequences.
Real-time systems are classified as hard [1], soft [3], and firm [4] systems. Hard real-time systems must guarantee that critical tasks are completed within a range of time, and missing a deadline can lead to a failure of the system. Examples are airbag systems, flight control systems, and pacemakers. Soft real-time systems provide some relaxation regarding the time limit, allowing a few tasks to miss their deadline, resulting only in performance degradation. Examples are video streaming and multimedia systems. Firm systems have strong deadlines, but missing a few of them does not imply a failure of the system, but only that the produced result is useless. Examples are telecommunication packets, switches, and radar systems.

2.1.1. Features

To achieve predictability, real-time operating systems must support some very important properties, including:

- Timelines: To support the execution of tasks within their deadline, the operating

system must provide a specific kernel mechanism for time management and for handling tasks with explicit timing constraints and different criticality.

- Predictability: The systems must be analyzable to predict the consequences of any scheduling decision. In safety critical applications, all timing requirements should be guaranteed offline, before putting the system in operation.
- Efficiency: Since most of real-time systems are embedded into small devices with severe constraints in terms of space, weight, energy, memory, and computational power, an efficient management of the available resources by the operating system is essential for achieving a desired performance.
- Robustness: Real-time systems must not collapse when they are facing peak-load conditions, so they must be designed to manage all anticipated load scenarios.
- Fault tolerance: Single hardware and software failures should not cause the crash of the system.
- Maintainability: The architecture of a real-time system should be designed according to a modular structure to ensure easy system modification.

2.1.2. Scheduling algorithms

The main classes of scheduling algorithms are the following:

- Preemptive / Non-preemptive [5]: In preemptive algorithms, the running tasks can be interrupted at any time to assign the processor to another task, while in non-preemptive algorithms, a task, once started is executed by the processor until completion
- Static / Dynamic [6]: In Static algorithms scheduling decisions are based on fixed parameters assigned to tasks before their activation, while in dynamic algorithms, parameters can change during system evolution.
- Offline / Online [7]: In offline algorithms the entire task set is known before the execution of the scheduling algorithm, while in online tasks can enter in the system at every moment and the scheduling algorithm must make scheduling decisions at runtime.
- Optimal / Heuristic [8]: In optimal algorithms scheduling is performed to minimize a given cost function, while in heuristic algorithms tends towards an optimal schedule but it is not guaranteed

In hard real-time applications that require highly predictable behavior, the feasibility of the schedule, must be done before task execution. In this way, if a task cannot be scheduled within its deadline, the systems must execute alternative actions to avoid catastrophic consequences.
In static real-time systems, where the task set is fixed and known a priori, all tasks can be precalculated offline, then at runtime, the dispatcher removes the tasks from the table and executes them. In dynamic real-time systems, where tasks can be created at runtime, the guarantee that they can be executed within their deadline must be done online every time a that a new task is scheduled.
The most used scheduling algorithms in real-time operating systems are rate monotonic [9], and EDF [10]. Rate monotonic is a static and preemptive scheduling algorithm. The priority is decided according to the period of the processes that are involved, where processes with a shorter period will have higher priority. Thus, if a process with the highest priority starts execution, it will preempt the other running processes. Earliest-deadline-first (EDF) is a dynamic scheduling algorithm where the higher priority is assigned to tasks with the earliest deadline, and if a new task with an earlier deadline is scheduled while another task is running, the scheduler preempts the currently running task and starts executing the new task.

2.2. Asynchronous calls in C++

`std::async()` is a C++ standard library function that allows executing a function in parallel with respect to the function that calls it [11]. It is used to exploit cpu parallelism, resulting in an improvement of the execution speed of the system. When called, `std::async()` returns a `std::future<T>` object, that can be used to get the return value computed by the called function with the `get()` method, when it finish its execution. The object `std::future<T>` represents a future of the return value with type `T` of the asynchronous function, and its `get()` method returns it, once the execution of the asynchronous function has completed. `std::async()` supports two scheduling policies:

- `launch::async` is the standard one and creates a separate thread to execute the function
- `launch::deferred` executes the function in the same thread after the `future.get()` call.

For our purposes, we consider only the default implementation, the `launch::async` one.

2.2.1. Low level implementation

When a `std::async()` call is performed, a new thread is created to execute it [12]. This will then be executed following the scheduling policy of the operating system. When it finishes its execution, the result is stored in a shared state, where the caller will take the result. the caller thread, after a `get()` operation will collect the result of the scheduled function if it has finished the execution, otherwise it will go in `wait()`, and it will be woken up when the scheduled function has finished. Synchronization between the caller thread and the called thread is handled by the shared state.

```
1  int sum(vector<int> v) {
2      int sum = 0;
3      for(int i = 0; i < v.size(); i++){
4          sum += v[i];
5      }
6      return sum;
7  }
8
9  void f(vector<int>& v) {
10     std::future<int> fut = std::async(sum, v);
11     // do something...
12     int sum = fut.get();
13 }
```

Listing 2.1: std::async() example

In the example shown in 2.1, a new thread is created for executing the function `sum`. `f()` and `sum()` are executed asynchronously, and possibly in parallel depending on the scheduler and other processes running in the system. Then, when the function f will perform the `get()` operation on the future, it will wait until the asynchronous function has terminated, and then save its return value in the integer variable sum.

2.3. Miosix

Miosix is an OS kernel designed to run on 32-bit micro-controllers that is under active development since 2008, with most of its kernel written in C++ [13].
Miosix provides support for C and C++ development, with a focus on C++, and it supports full C and C++ standard libraries, and standard POSIX thread API, including thread, mutexes, and condition variables.
Miosix is an RTOS suited for embedded systems, and each thread has a priority, given by

its deadline. Priority is represented as an integer that goes from `0` to `NUM_PRIORITIES - 1` where `0` is the lower priority and `NUM_PRIORITIES - 1` is the maximum. `NUM_PRIORITIES` can be set by the configuration file, and is set to 4 by default.
The Miosix Toolchain is a patched version of the GCC compiler, Newlib C library, and libstdc++ library. The patches to the compiler and standard libraries are required to provide thread safety, improve the speed of the pthread and C++11 thread APIs, and to reduce the code size and the RAM requirements.

2.3.1. Kernel Architecture

The central part of the kernel is the scheduler [14]. The scheduler is preemptive and manages both kernel threads and user space threads within processes. Both are represented by the Miosix Task Control Block, implemented by the `Thread` class. An overview of the architecture is shown in 2.1.

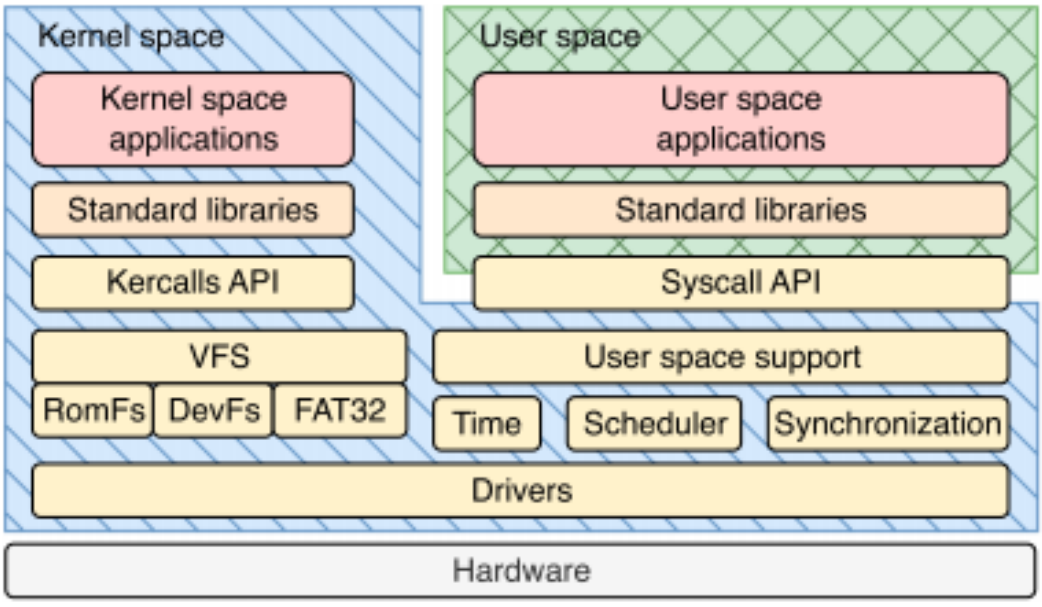

Figure 2.1: Architecture of the miosix kernel from [14]

Miosix supports three scheduling algorithms: priority, EDF, and I+Pi [15]. The scheduler policy must be defined at compile time, and the priority scheduler is the default one. It is implemented in a way where higher priority threads always have precedence over lower priority threads, and, if multiple threads with the same priority exists, they are

scheduled in a round robin way. A simplified algorithm for the priority scheduler is shown in 2.2.

```
// Full code in:
// kernel/scheduler/priority/priority_scheduler.cpp / .h
Thread *threadList[NUM_PRIOS]; //Array of circular lists
void scheduler() {
    for(int i=NUM_PRIOS-1;i>=0;i--) {
        if(threadList[i]!=nullptr) {
            Thread *chosen=threadList[i];
            threadList[i]=threadList[i]->next;
            if(chosen->isInKernelspace()) disableMPU();
            else enableMPUregions(chosen->proc.mpu);
            schedule(chosen);
            setNextPreemption(chosen);
            return;
        }
    }
}
```

Figure 2.2: Implementation of the scheduler from [14]

Miosix also supports multi-core scheduling in its unstable version, where threads can run in different cores, and they are scheduled as in the priority scheduler. It keeps in a global list with all the ready threads for each priority level, and when a core performs the interrupt `IRQrunScheduler(coreId)` it take from the non empty list with higher priority the thread in front and executes it. If a core does not find ready threads to run, it will run its idle thread. The interrupt that performs `IRQrunScheduler(coreId)` is periodic. This means that when a thread is scheduled to run on a core is not assured that it will finish its execution. If `IRQrunScheduler(coreId)` is called before its finish, it is stopped and scheduled at the back of the list of ready threads with its priority, leading to a round robin behavior if there are different threads scheduled with the same priority.

2.3.2. Event Queue

Miosix supports an event queue system, with the goal of scheduling functions, represented by the `std::function<void ()>` C++ structure, and executing them in a separate executor thread. It is implemented by the class `EventQueue` ad has the following interfaces:

- `void post(std::function<void ()> event)` Schedule the execution of an event passed has a `std::function<void ()>` in a FIFO order

- `void run()` Executes functions scheduled by the `post()` method in a FIFO order using an infinite loop. The running thread can `wait()` if no events are present in the queue, and it will be woken up when an event is scheduled. This function never returns, apart from when an exception is raised by the function being executed.
- `runOne()` If the list containing the events is not empty, it executes only the first one and then returns.

2.4. Compiler optimizations

During the compilation process, optimization techniques are applied to transform source code into more efficient machine code, aiming to enhance execution speed and reduce memory usage [16].

In the classical compiler design model, a compiler can be divided into three components:

- front-end: Parses the code into tokens, creates the abstract syntax tree (AST) [17] from those, performs semantic analysis, and converts the AST into an intermediate representation (IR).
- optimizer: Applies most of the optimizations, like dead code elimination, loop unrolling, or inline expansion
- back-end: Converts optimized IR into target-specific machine code with operations like instruction selection, register allocation, instruction scheduling, and target-specific optimization.

The three phases are represented in Figure 2.3.

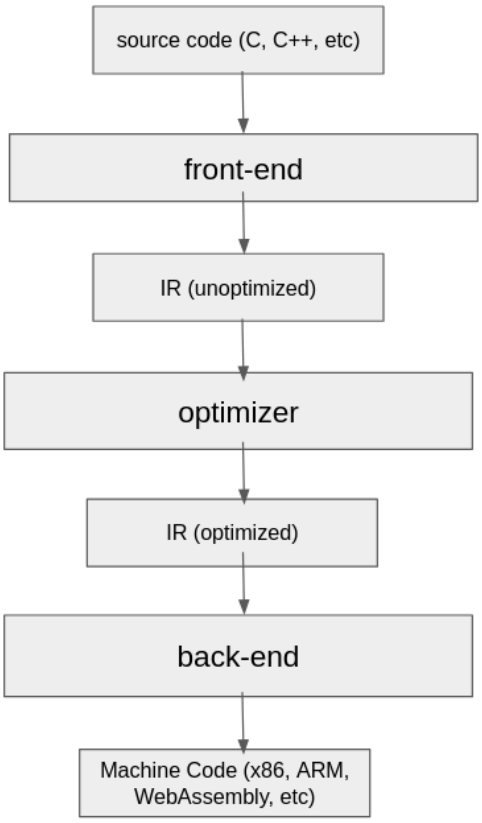

Figure 2.3: Representation of the 3 compiler optimization phases

2.4.1. LLVM

LLVM [18] is a compiler infrastructure, implementing the optimizer (middle-end) and back-end parts described above. LLVM works starting from the LLVM IR [19] produced by a front-end like CLang [20] for C/C++ or rustc [21] for Rust. LLVM IR can be described as a high-level assembly-like language that acts as an intermediate stage between source code and machine code, which can be optimized with a variety of transformations over multiple passes.

In the following example, we are going to see how a basic C++ code can be optimized by working on the LLVM IR.

Considering the code on Listing 2.2 that implements a basic function to add two integers:

```
1 int add(int a, int b) {
2     return a + b;
3 }
```

Listing 2.2: Basic add() function

This code, compiled without LLVM optimizations, will generate the IR presented on Listing 2.3:

```
define dso_local i32 @add(i32 %a, i32 %b) {
    entry:
    %retval = alloca i32, align 4
    %a.addr = alloca i32, align 4
    %b.addr = alloca i32, align 4
    store i32 %a, i32* %a.addr, align 4
    store i32 %b, i32* %b.addr, align 4
    %0 = load i32, i32* %a.addr, align 4
    %1 = load i32, i32* %b.addr, align 4
    %addtmp = add nsw i32 %0, %1
    store i32 %addtmp, i32* %retval, align 4
    %2 = load i32, i32* %retval, align 4
    ret i32 %2
}
```

Listing 2.3: LLVM IR add function

Without optimizations, everything goes through memory with operations alloca, load, and store.
In the code on Listing 2.4, we are going to see how the optimization will reflect in the optimized LLVM IR, removing data passage by memory, and executing everything on one line:

```
define dso_local i32 @add(i32 %a, i32 %b) {
    entry:
    %add = add nsw i32 %a, %b
    ret i32 %add
}
```

Listing 2.4: LLVM IR optimized add function

2.5. Continuation passing style

Continuation passing style (CPS) [22] is a programming style where functions do not return values in the usual way. A function, written in CPS, will receive as a parameter a continuation that will be executed when the function finishes its execution, and its return value will be passed as a parameter in the continuation.

Compared to a code written in the canonical style, where a function call is performed with a statement like `f(args)` and the caller function gets the return value of `f()`, and then continues its execution, in CPS the statement is modified in `f(args, cont_args, cont)` where `cont_args` is a reference to the local variables used by the caller function, and `cont` contains all the instruction performed in the continuation of the caller function. The called function `f()` when it finish its execution, it will call `cont` using its return value.
The code in Listing 2.5 shows an example of the implementation of the factorial algorithm in continuation passing style.

```
func factorial(n int, next func(int)) {
    if n == 0 {
        next(1)
    } else {
        factorial(n-1, func(k int) {
            next(n * k)
        })
    }
}
```

Listing 2.5: Pseudocode of CPS Factorial implementation

2.6. State of the art

2.6.1. Asynchronous programming in Rust

In this part, we are going to present how asynchronous programming can be used in Rust with a focus on how its scheduler is implemented from [23]. The scheduler can be of two different types: preemptive [24] and cooperative [25]. In a preemptive scheduler, each task is executed in a dedicated thread, and the scheduler manage the execution space of each tasks by hardware interrupts to switch the execution from one thread to another. This guarantee more fairness regarding the execution time of the tasks, but, as disadvantages, it has high memory usage, creating a thread for every task, and have expensive context switches when the scheduler interrupt a task to execute another one. The cooperative scheduler does not use hardware interrupt as the preemptive scheduler, but tasks themselves yield execution to allow other tasks to run on the cpu, saving their state for their restart when some resources will become available. This approach saves memory space by executing different tasks on the same thread.
Rust implements cooperative multitasking through the `Future` `rust_future_trait` trait

and `async/await`. A future represents a value that will become available in the future. Instead of blocking while waiting for the value, the execution can continue with other tasks until the future is ready. The `Future` implements the `poll()` method and specifies the value that the future has to return. The `poll()` output type is an enum `Poll` that can be `Ready(T)` with the value inside it, if the execution is finished, or `Pending` if the execution is not finished.
The executor is responsible for scheduling the different tasks. It maintains a queue of futures, it repeatedly pops a task from the future, pools it, and handles its results: if `Ready` the task is finished, and the future can be dequeued, if `Pending` it is scheduled again in the queue.

Tokio

Tokio [26] is an asynchronous runtime for Rust that implements the executor explained in the previous part. It is the most used runtime in the Rust ecosystem, and it provides different functionalities, including a task scheduler, an I/O driver for interacting with the operating system, asynchronous network interfaces, and synchronization primitives like mutexes and barriers. We are going to focus on the multithreaded scheduler.
The scheduler, in the beginning, creates a default number of threads, usually equal to the number of cores of the cpu. Each thread has a local queue of tasks that it has to execute. It enters a loop where it continuously pops tasks from the queue and executes them until it yields. When tasks are scheduled, they are sent to the queue of one of the schedulers, and usually they comes in batches, creating a different amount of workloads between the different threads. To solve this issue, Tokio implements the work-stealing technique [27]. When a thread does not have tasks to run on its queue, it steals tasks from other queues and executes them. If all queues are empty, it will go to sleep. If the queue is full and there is a thread that is stealing from the queue, it means that the run queue will be empty soon, and the given task is pushed to the global queue. The global queue is a linked list shared among all the threads, where the operations of insertion and deletion of tasks in the queue is guarded by a mutex. An example of work-stealing is represented in Figure 2.4.

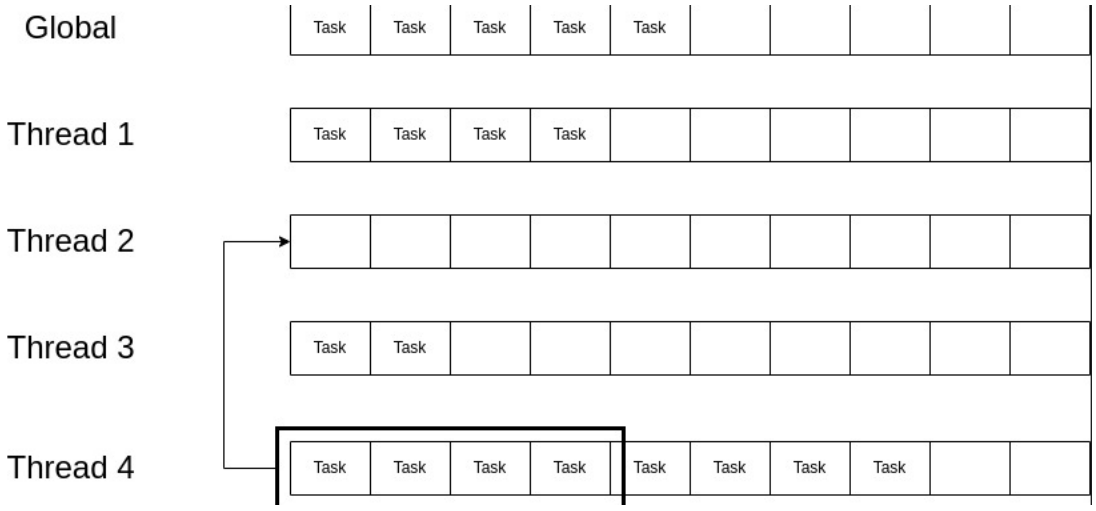

Figure 2.4: Example of work-stealing in Tokio from [23]

This approach is very efficient when there are a lot of tasks scheduled, and communication between threads is avoided by the use of one queue for every thread.
This approach could be implemented using coroutines [28] in C++. We did not go in that way because coroutines modify at compile time the functions, adding some lines, for example, for allowing their execution stop, and these modifications make them a little bit slower.

2.6.2. Lazy task creation

The work of Eric Mohr, David A. Kranz and Robert H. Halstead, Jr. [29] proposes the technique of *lazy task creation*, implemented on Mul-T [30], an efficient parallel implementation of Scheme. The main goal of this implementation is to create tasks retroactively, only when resources become available. The lazy task scheduling approach tries to solve the problem of task granularity, especially, when a lot of tasks are created. If we consider a normal parallel algorithm, it often ends up creating tasks of a finer grain than the implementation can exploit efficiently, due to their scheduling costs that become greater than the work they perform. In the Mul-T system, an asynchronous call can be executed with `(C (future X))` declaring that computation `X` may proceed in parallel with its parent continuation C. In a normal interpretation, a child task is created to execute X, while the parent task computes the continuation of C. One optimization to avoid creating tasks that are too fine-grained is load-based inlining, consisting in making a separate task to execute X if the system is not loaded, or execute X after the termination of C, alternatively. This strategy solves the problem of tasks granularity, but has some speed drawbacks, especially when the continuation of C after the future call is way longer than

the execution of X. A better approach is using the lazy task creation algorithm, which consists in starting evaluating X in the current thread, and save data so that its continuation C can be moved to a separate task when another processor become idle. With this execution tree, an abundance of potential fork points is avoided, maximizing the run-time task granularity, while preserving parallelism and achieving good load balancing.

An example where the lazy task creation approach solves the solution of the granularity problem is the algorithm to sum the leaf of a binary tree represented by the scheme code in Listing 2.6:

```
(define (sum-tree tree)
    (if (leaf? tree)
        (leaf-value tree)
        (+ (sum-tree (left tree))
            (sum-tree (right tree)))))
```

Listing 2.6: Recursive tree sum in Scheme from [29]

This code can be parallelized with two recursive calls to `sum-tree` that can proceed in parallel, represented with the implementation in 2.7:

```
(define (psum-tree tree)
    (if (leaf? tree)
        (leaf-value tree)
        (+ (future (psum-tree (left tree)))
            (psum-tree (right tree)))))
```

Listing 2.7: Parallel tree sum in Scheme from [29]

With eager task creation, this program would create 2^d tasks to sum a tree of depth d, in particular, the number of tree node handled by a task would be 2 on average. This execution will likely lead to poor performance. An example of this execution is shown in 2.5. Applying lazy task scheduling, it expands the tree breadth-first until all processors are busy, and then expands the tree depth-first within the task on each processor. It can be summarized with the BUSD(Breadth-first. Until Saturation, then Depth-first) notation. This approach has a balanced load, reducing task creation and increasing granularity. An example is shown in Figure 2.6.

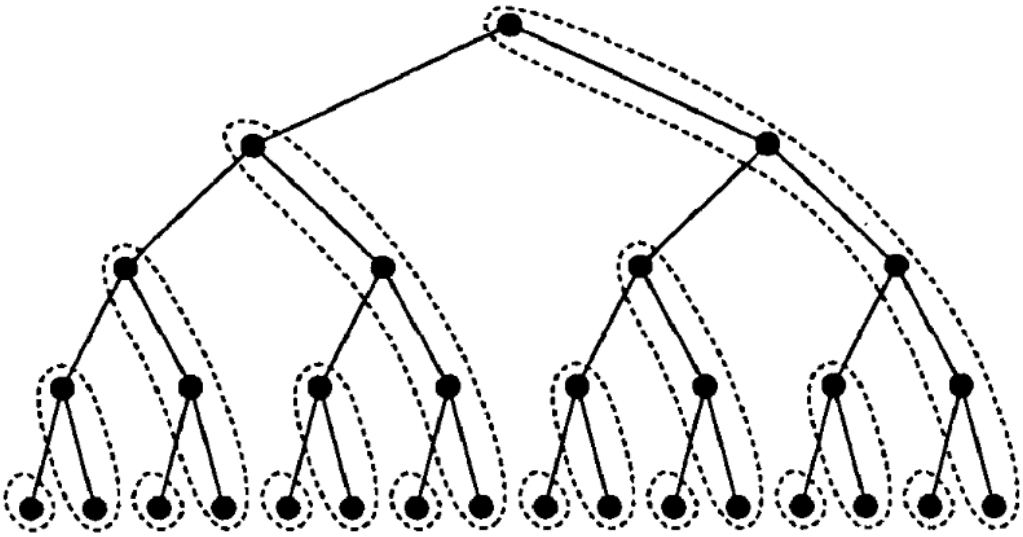

Figure 2.5: Tree representing the task graph by the processors that run them with eager task creation from [29]

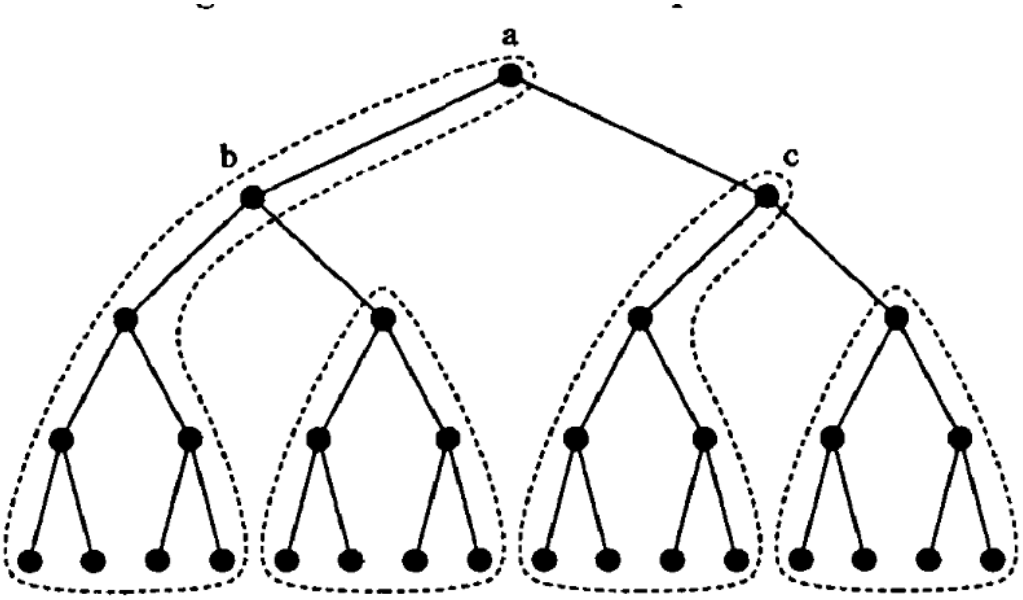

Figure 2.6: Tree representing the task graph by the processors that run them with lazy task creation from [29]

This approach is the base of lazy task creation, but it is implemented with a functional programming language, that made easy capturing the continuation of a program. Our

goal is to apply this approach in a standard C++ code.

2.6.3. Prioritized asynchronous calls

The authors Tomas A. Lopez and Nobuyuki Yamasaki [31] propose an implementation for asynchronous tasks based on the lazy task approach, implemented on the "Responsive Multithreaded Processor" (RMTP) [32].
The RMTP is a high-performance processor architecture for distributed embedded hard real-time operating systems. It offers a high set of unique hardware features, valued in the time-critical domain, that support the development of deterministic concurrent systems, including:

- **prioritized execution** where hardware threads have a priority between 0 and 255, and the hardware schedules them considering their priority and available resources.
- **hardware thread management** allows to create and destroy hardware threads, and also block and restart targeted threads
- **context cache** with memory on-chip used to quickly save, copy, or restore data to and from the register file, allowing to swap the contents of private registers of a thread on context switch, in a constant time without hitting main memory.

The system adopts a conditional sporadic DAG task model. Each task is represented as a directed acyclic graph (DAG) of sub-tasks with dependency relations [33]. The scheduler assumes a fixed task priority (FTP) or Fixed Job Priority (FJP) policy, therefore assuming that priorities will not change once a job is released.
Lazy task creation is applied to create threads gradually, delegating the creation of the subsequent thread to the child threads. Differently from eager forking, which creates all threads at once, lazy forking adapts the number of active threads to the available processing resources.
Lazy task creation, instead of spawning a thread after every asynchronous call, saves the continuation of the current program and schedules it for later execution, where continuations will be executed from a shared thread pool. This approach ensures that no more threads are created than there are available cores.
An example is shown in Figure 2.7.

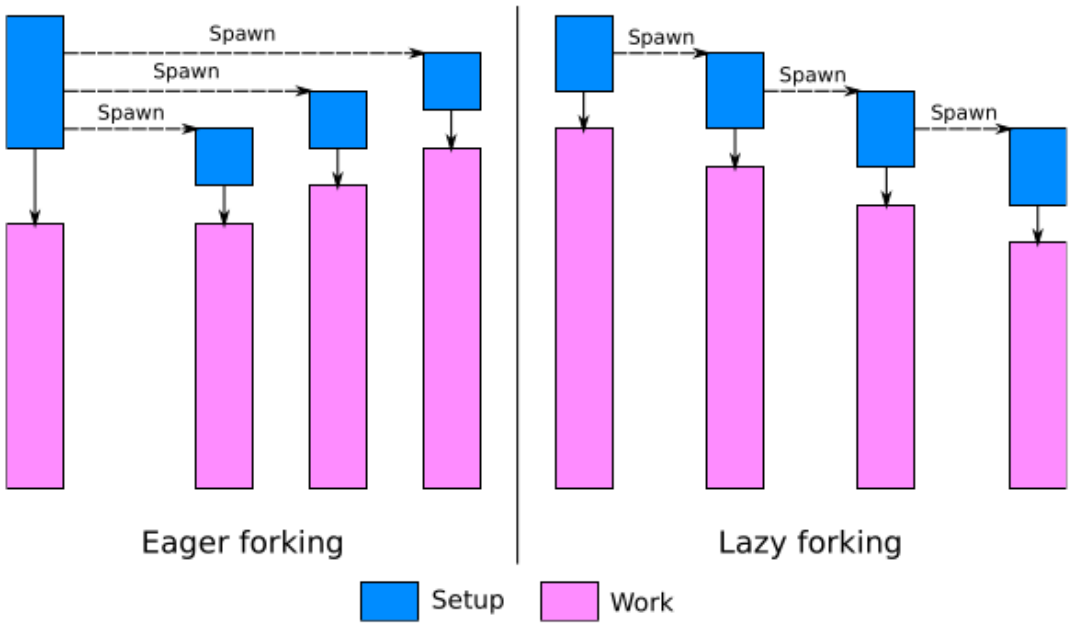

Figure 2.7: Lazy task creation delegates the fork procedure to the thread child form [31]

Backend architecture

Threads created by the system are directly mapped to RMT hardware threads. When asynchronous calls are performed, the current execution state is saved in the context cache with an atomic instruction. If the cache is full, the call will be performed as a standard C procedure call.

The RMT has simple scheduling policies, where the threads with higher priority will be the ones in execution. Rescheduling is necessary only when threads are created, destroyed, or new jobs are submitted.

The runtime is built on three main components: the context manager, the work pile, and a pair of latent threads that manage accesses to the work pile and perform preemptions. The context manager is the entity that manages the active contexts of the systems, and is the only interface between the code in execution and the backend runtime system. It is responsible for forking, post-termination cleanup, and the selection of the next thread that will replace it.

The data structure used to implement the algorithm of lazy task creation is a 2-level work pile shown in Figure 2.8. The first level is a priority-ordered linked list, supporting random insertion and in order deletion. List elements are called plates, each one containing a deque of local work deques, one per context manager. For every priority level, there is only one plate containing all threads associated with their DAG job.

Threads scheduled by asynchronous calls inherit the priority of the caller thread. Context managers starts scheduling threads from the plate with higher priority level, then it starts the thread execution and then remove it from the deque. They move to the next lower priority, adjust their priority with the one that they are currently working on, and schedule threads from the plate. This is repeated until the bottom of the pile is reached, when they suspend themselves. This approach result is a priority scheduler where each priority level has different threads executing in parallel. With this implementation, a group of workers is able to asynchronously schedule works on their own without the intervention of a centralized scheduler.
Lazy task creation is implemented, saving the continuation of a thread in the register file, and the top portion of the thread stack is copied. This memory copy is necessary to preserve the stack contents of the asynchronous calls, to avoid the child thread to overwrite it.

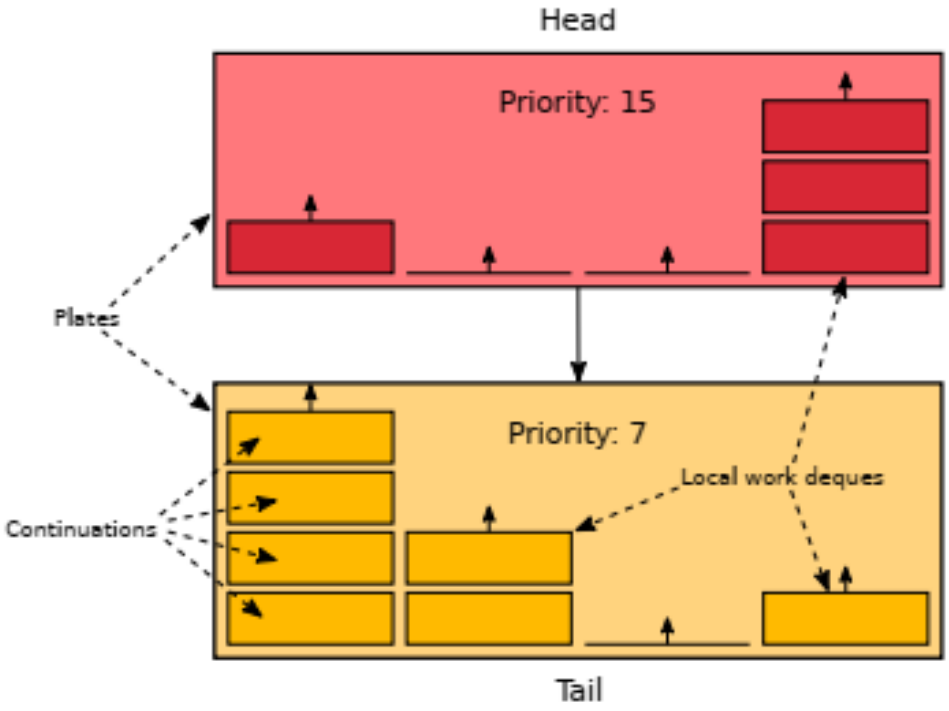

Figure 2.8: A work pile of two plates, hosting 2 DAG jobs of priority 15 and 7, executed on 4 context from [31]

We used this approach as the basis of our work, but it could not be applied in our context because it relies on hardware facilities for saving the execution state in a data structure.

2.6.4. Cilk Plus

Intel Cilk Plus [34] is an extension to the C and C++ languages to support data and task parallelism [35], and it is one of the best way to exploit multicore and vector processing. Cilk Plus uses different the keywords `cilk_spawn`, `cilk_spawn`, `cilk_sync` and `cilk_for` to specify parallelism. `cilk_spawn` permits a given function to execute asynchronously with the rest of the calling function, `cilk_sync` indicates that all the tasks spawned in the current function must finish before the execution continues. `cilk_for` followed by a for loop tells that the iterations of the loop can be executed in parallel. The code on Listing 2.8 is a simple program that computes the Fibonacci sequence exploiting maximum core parallelism with Cilk Plus:

```
#include <cilk/cilk.h>
#include <assert.h>

int fib(int n) {
    if (n < 2)
    return n;
    int a = cilk_spawn fib(n-1);
    int b = fib(n-2);
    cilk_sync;
    return a + b;
}

int main() {
    int result = fib(30);
    assert(result == 832040);
    return 0;
}
```

Listing 2.8: Fibonacci sequence computed with Cilk Plus from [35] for exploiting cpu parallelism

Considering how `cilk_spawn` works internally, [36] when called, it does not create a new thread to execute it, but it relies on worker threads, where each one has its own queue to take tasks. The worker threads use work stealing, if they do not have a task to executes on their queue, and if more tasks are scheduled in the queues than available resources, they will proceed sequentially. `cilk_sync`, when called, it does not cause any thread to stall, and the worker that performs the `cilk_sync` call will find other work to steal. Cilk Plus implements lazy task creation, where the scheduler performs parent

stealing, not children stealing. In particular, when a call like `int x = cilk_spawn f()` is called, and there are no free resources, the current worker will continue executing `f()`, and the task in execution is scheduled in the queue. We followed the idea of this approach for implementing the lazy task scheduling, but we could not applied it entirely because it relies on CPU mechanisms for scheduling, running, and stopping the task's execution, while we could rely on the operating system without hardware-specific support.

3 | Design and implementation

The goal of the implementation was to provide a back-end architecture based on an event queue that supports scheduling asynchronous calls with different priorities, and maintaining the same semantic of `std::async()`, where the scheduled function inherit the priority from the thread that calls it. We implemented two different versions, one based on a standard threadpool, and the other one based on lazy task creation [29]. The main difference between the two approaches is that, in the threadpool approach the caller thread will not be included in the event queue, and it will continue in parallel with the functions scheduled by the event queue, while in the other approach, the caller, if there are no more free cores, will stop its execution, let executes the scheduled call, and its restart will be scheduled in the event queue with all the other calls. The programmer can select the policy to use by choosing between two alternative C++ namespaces, namely `threadpool` and `lazy`. Asynchronous calls can be performed by `async(f, args...)` as per the standard C++ API, and then its result can be stored in the `future` class and obtained with the `get()` method. One difference from`std::async()` is that in our implementations the event queue needs to be instantiated, and it can be done by `start_async()`. In the next sections, we are going to explain the implementation steps needed to obtain a working system, starting with the implementation of the priority event queue, which forms the basis of the threadpool and lazy approaches, and then we are going to enter into details about how the two approaches have been implemented.

3.1. Priority event queue

The Priority Event Queue extends the event queue presented in 2.3.2 to support the scheduling of functions based on their priority.
Functions can be scheduled passing their priority as an argument by the method: `void post(std::function<void ()> event, Priority priority)`, or with the pre-existing Miosix event queue API interface `void post(std::function<void ()> event)`, where it get the thread priority through Miosix API and it calls the other function with the computed priority as an argument.

Functions are executed by methods `void run()` and `void runOne()` where the next function to execute is the one with higher priority, and scheduled before all the other functions that have its same priority.
The priority event queue has been implemented with two different approaches that both exploit multi-core system to optimize the execution.
Both approaches work under the assumption that the threads created by the priority event queue are the only ones running in the operating system. The goal of both approaches is to have at most C functions in execution, where C is the number of cores in the system.

3.1.1. Priority thread reuse scheduler

With this approach, when a function is scheduled, if there are free cores or its priority is greater than the one of any function in execution, it creates a thread to execute it, otherwise it is inserted in the waiting list of the functions with its priority. When a function terminates its execution, before destroying the thread, it checks if there are other functions to be scheduled, and if so, it executes the one with higher priority and in front of the waiting list, and the thread will change his priority with the one of the new function. This allow us to avoid the run time cost of the allocation of resources for another thread space. Considering $NUM\_PRIORITIES$ as the number of different priorities in the system, and C as the number of cores, we allow the execution of a maximum C functions at the same time, in particular, if a function with higher priority is scheduled, the Miosix scheduler will stop the execution of the function with lower priority and execute the one with higher priority.
A pseudocode of this approach is shown in Listing 3.1: .

```
1  void run(std::function<void ()> event){
2      std::function<void ()> f = event;
3      while (f) {
4          f();
5          f = nullptr;
6          for (int prio = NUM_PRIORITIES - 1; prio >= 0; prio--) {
7              if (!events[prio].empty()) {
8                  f = events[prio].front();
9                  events[prio].pop();
10                 break;
11             }
12         }
13     }
```

```
}

void post(std::function<void ()> event, Priority prio){
    if(free_core() || has_greater_priority(prio)){
        thread(run(event));
    }else{
        lock();
        events[getInt(prio)].push(event);
        unlock();
    }
}

void post(std::function<void ()> event){
    post(event, getPriority(event));
}
```

Listing 3.1: Pseudocode of the priority event queue run and post methods implementation with the priority thread resuse scheduler approach

3.1.2. Pre-allocated priority thread scheduler

In this approach, when the priority event queue is created, it spawns C thread for every priority level, resulting in having `C * NUM_PRIORITES` threads. Each thread executes the `void run()` method as it was implemented in the basic event queue, with the small difference that a different list and mutex are used for every priority level. All the `run()` methods will schedule events of their priority in a FIFO order, and the os scheduler will manage the execution of the threads based on their priority.
The previous approach optimizes the memory space of the system, while this approach optimizes the execution speed because it never creates threads, apart from when it is instantiated. In our implementation, we decide to value execution speed more than dynamic memory saving, and also creating a few threads, which will be in wait status for a long time, does not affect the overall system that much, and for this reason, we decide to use the second approach. A pseudocode of this approach is shown in Listing 3.2:

```
void post(std::function<void ()> event, Priority prio){
    lock();
    events[getInt(prio)].push_back(event);
    unlock();
}

```

```
7  void post(std::function<void ()> event){
8      post(event, getPriority(event));
9  }
10
11 void run(std::function<void ()> event){
12     int prio = getPriority();
13     std::function<void()> f;
14     while(!empty()){
15         lock();
16         while(events[prio].empty) cv.wait();
17         f = events[prio].front();
18         unlock();
19         f();
20     }
21 }
22
23 void startRun(){
24     for(int i=NUM_PRIORITIES;i>=0;i--){
25         for(int j=0;j<num_core;j++){
26             std::thread t(&PriorityEventQueue::run, this, i);
27             t.detach();
28         }
29     }
30 }
```

Listing 3.2: Pseudocode of the priority event queue startRun, run and post methods implementation with the pre-allocated priority thread scheduler approach

The implementation of `PriorityEventQueue` is available here:
`https://.com/LucaRomano2/miosix-kernel/tree/thesis/miosix/e20`

3.2. Threadpool

The threadpool implementation is pretty similar to the implementation of the priority event queue with some differences to suit for `async()` and `future<T>`.

A UML class diagram of this approach is shown in 3.1.

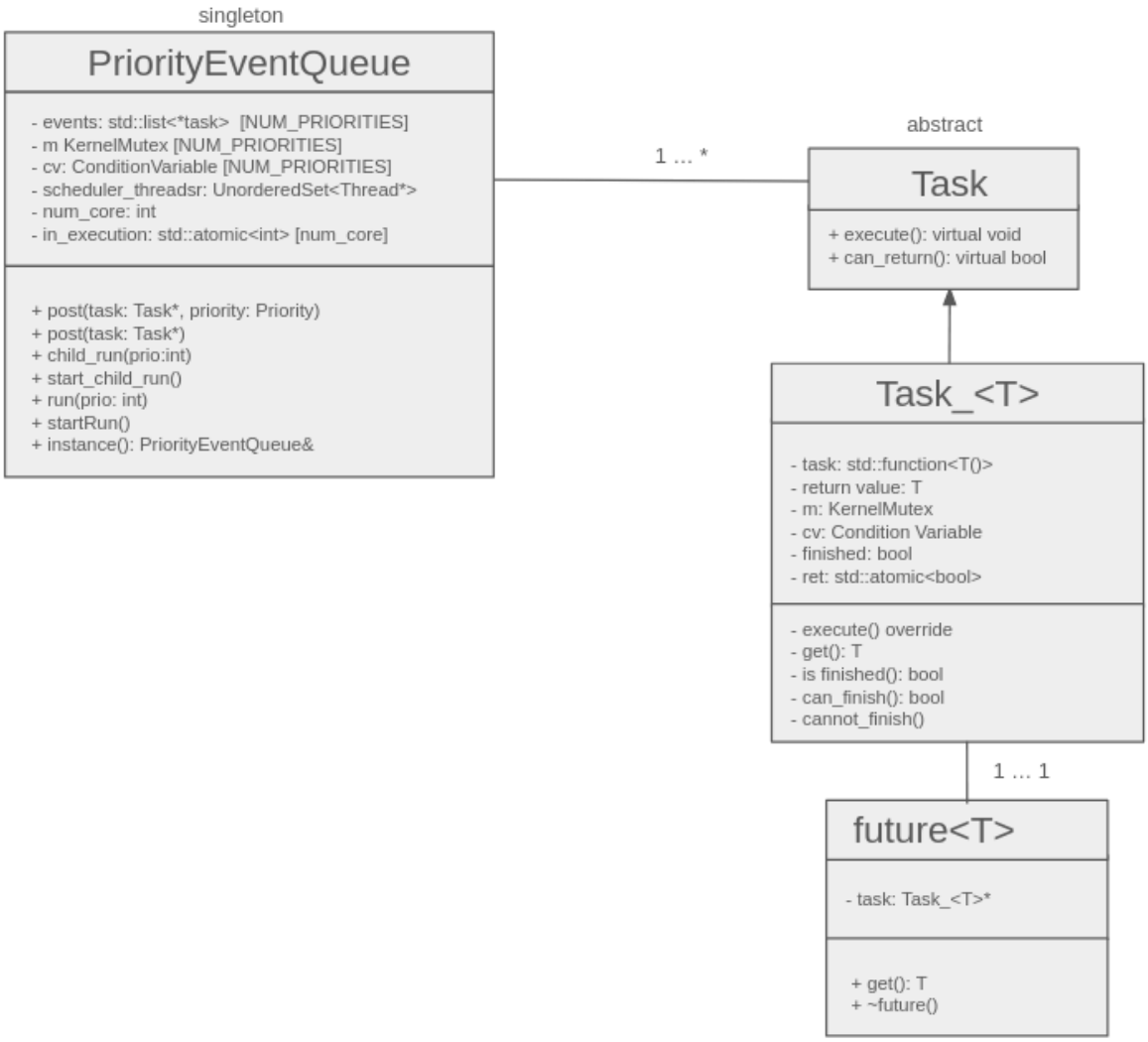

Figure 3.1: UML class diagram of the threadpool approach

The post method of `PriorityEventQueue`, instead of receiving a `std::function<void ()>`, receives a pointer to an instance of the `Task` class and it will also store pointers to instances of the `Task` class in the list that stores all of the scheduled asynchronous calls for each priority level. `Task` is an abstract class with the `execute()` method that must be overridden. `Task` is extended by the `Task_` class, where `T` is a template that is given by the return type of the function scheduled by `async()`. `Task_` has the attribute `task` that is the `std::function` with included also its arguments that it has to execute. Its attributes includes also a template variable `return_value` for storing the execution return value, and synchronization variables: mutex and condition variable, `Task_` has methods `execute()` that override the `execute()` method of `Task`, and it is used to runs its associated task and save the returned value, and `get()` that return the return value when it is ready.

`thrreadpool::async()` when called, returns an object of `threadpool::future`. `threadpool::future` has a private attribute that points to the object of `Task_` and it can be used by the `get()` method that will execute the `get()` call of its associated `Task_`. In the case when a task that is part of the scheduler, meaning that is executing in the `run()` of `PriorityEventQueue`, perform a `get()` operation, and the result of its associated task is not ready yet, it will create another thread to executes the `child_run()` function that will executes the other scheduled functions in the list, and then, the `run()` method were the call has been performed, goes in the `wait()` status. An atomic array is used to count how many schedulers are in execution. All the `child_run()` tasks have a condition for continuing in the executing loop that checks if the number of schedulers in execution is equal to or greater than `num_core`, and if it is, they will terminate their execution. This means that the `run()` method that went in `wait()` status by the `future::get()` call has been woken up because its task has finished the execution, and the `child_run()` method does not need to substitute one of the schedulers. The `threadpool::async(F&& f, Args&&... args)` function must be called passing as parameters the function that is planned to be executed asynchronously, and an arbitrary number of arguments, that are the ones that the function `f()` necessitate. Then, it allocates space on the heap for an instance of `Task_` passing as argument a packaged task that binds `f`, and `args...` in a `std::function` type. It calls the `post()` method of the `PriorityEventQueue` passing a pointer to a `Task` that points to the `Task_` just created, and it returns to the caller an object of `future`, which contains a pointer to the `Task_` instance that has just been scheduled in `PriorityEventQueue`. The implementation of `threadpool::async(F&& f, Args&&... args)` is shown on Listing 3.3.

```
1  template<class F, class... Args>
2  future<typename std::result_of<F(Args...)>::type> async(F&& f,
       Args&&... args)
3  {
4      typedef typename std::result_of<F(Args...)>::type return_type;
5      std::function<return_type()> t =
           std::bind(std::forward<F>(f), std::forward<Args>(args)...);
6      Task_<return_type>* task_ = new Task_<return_type>(t);
7      Task* task = task_;
8      PriorityEventQueue::instance().post(task);
9      return future<return_type>(task_);
10 }
```

Listing 3.3: Implementation of the async API function

The `PriorityEventQueue` will executes all the scheduled tasks calling their `execute()` method, which will also mark the task as executed, setting a boolean variable to true. When the caller function will perform the `get()` operation, if its task is marked as finished, it will get the returned value from `f()`. Otherwise, if the execution of its task is not finished yet, it will go in `wait()` status, and then, when the execution is finished, it will be woken up and get its returned value. `PriorityEventQueue` is implemented as a Singleton class, and its thread scheduler can be created by the call `start_async()` that, for every priority level, will create a number of threads that execute the `run()` method, equal to the number of cpu cores. The implementation of the `run()` method is similar to the one presented in Listing 3.1 with some differences to suit this context, and it is shown in Listing 3.4.

```
1  void PriorityEventQueue::run(int prio)
2  {
3      Thread* thread=Thread::getCurrentThread();
4      thread->setPriority(Priority(prio));
5      scheduler_threads.insert(thread);
6      Task* t;
7      for(;;)
8      {
9          Lock<KernelMutex> l(m[prio]);
10         while(events[prio].empty()) cv[prio].wait(l);
11         t = events[prio].front();
12         events[prio].pop_front();
13         Unlock<KernelMutex> u(l);
14         t->execute();
15         if(!t->can_return()) delete t;
16     }
17 }
```

Listing 3.4: Implementation of PriorityEventQue::run()

When the instance of `future` goes out of scope, if its task has finished the execution, it will deallocate the instance of `Task_`. If the execution has not finished yet, it means that the function will never return its result, but it needs to be executed anyway by the `run()` method `PriorityEventQueue`, so we cannot deallocate it. To avoid this, it marks its `Task_` with an atomic boolean variable to indicate that it cannot return its execution. When `PriorityEventQueue` finish to execute a task, it checks if the task cannot return, and if it is the case, it deallocates it. With this approach, we guarantee the functionality of the ram memory, avoiding problems of memory leaks, and accessing memory space that has been deallocated.

3.3. Lazy task scheduling

In this implementation, we apply the algorithm of lazy task creation on the priority event Queue. The overall implementation is similar to the one of `threadpool`, in particular, the `async()` function, and the `future` class are the same, and also the `run()` method is very similar, with a difference to handle the case where a function that is running on the scheduler schedules another asynchronous call, that will be explained later. A UML class diagram of this approach is shown in 3.2.

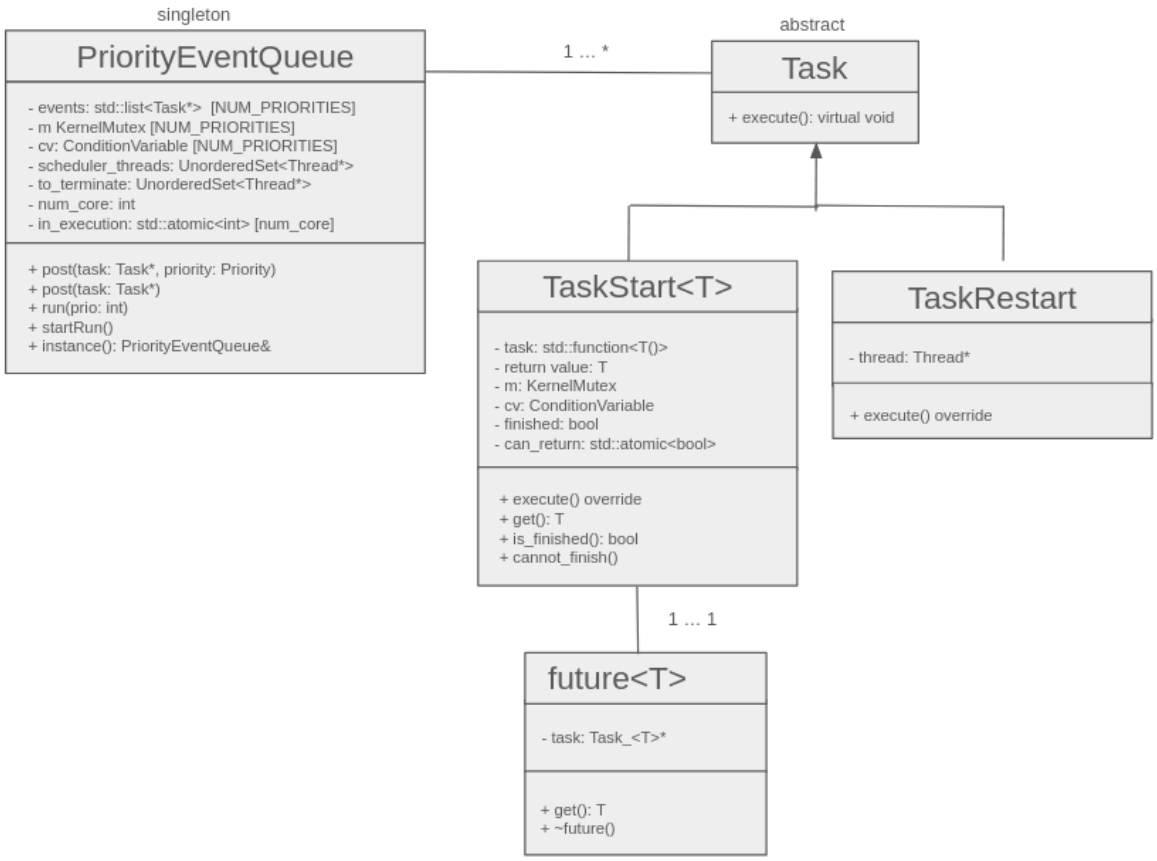

Figure 3.2: UML class diagram of the lazy task scheduling approach

The main differences between the two implementations are in the `post()` method of `PriorityEventQueue` and in how the `Task` class is extended. The `Task` class, to handle both the starting of the execution of a task and the restart of a task, is extended by the classes `TaskStart` and `TaskRestart`. Both classes overrides the `execute()` method of `Task` that will be called by the `PriorityEventQueue`. `TaskRun` will start the execution of its task as `Task_` of `threadpool`, while `TaskRestart` will receive a pointer to the thread where its task is in execution, and when `execute()` is called, it will wake up the thread.

The `post()` method of `PriorityEventQueue`, when called for scheduling a task, it checks if the number of scheduled task in `PriorityEventQue` with the current priority is lower or equal to the number of cores, and, if it is lower, it adds in the queue an instance of `TaskStart` with the task passed as argument, and it will notify the `PriorityEventQueue` that will be woken up and executes it. Otherwise, if the number of scheduled tasks with the current priority is equal to the number of cores, it applies the Lazy Task Creation algorithm, by inserting in the queue an instance of `TaskStart` with the posted task, and then an instance of `TaskRestart` passing as an argument a pointer to the current thread. After inserting the two tasks in the queue, it creates another thread that executes the `run()` method of `PriorityEventQueue`, and it will get from the list the task that has just been inserted. Then, it calls the `wait()` method, so that the current thread goes to sleep, and the execution will continue with maximum parallelism between the tasks scheduled by `PriorityEventQueue`, meaning that the number of active threads created by `PriorityEventQueue`, will be equal to the number of cpu cores, in this scenario. The implementation of the `post()` is shown in Listing 3.5.

```
1  void PriorityEventQueue::post(Task* task, Priority priority)
2  {
3      Lock<KernelMutex> l(m[priority.get()]);
4      Thread* current_thread=Thread::getCurrentThread();
5      if(in_execution[priority.get()].load() + 1 >= num_core)
6      {
7          Task* task_restart = new TaskRestart(current_thread);
8          if(scheduler_threads.count(current_thread))
9          {
10             std::thread t(&PriorityEventQueue::run, this,
                   priority.get());
11             t.detach();
12             in_execution[priority.get()]--;
13             to_terminate.insert(current_thread);
14         }
15         events[priority.get()].push_back(task);
16         events[priority.get()].push_back(task_restart);
17         cv[priority.get()].signal();
18         Unlock<KernelMutex> u(l);
19         Thread::wait();
20     }else{
21         events[priority.get()].push_back(task);
22         cv[priority.get()].signal();
```

```
23      }
24  }
```

Listing 3.5: Implementation of PriorityEventQue::post()

When a task scheduled by `PriorityEventQueue` schedules another call, and a new thread is created for it, it is marked to be deleted at the end of its task execution, because the created thread will continue to be part of the scheduler instead of it. For this, a thread safe unordered set has been implemented, which simply does the operation of an unordered set but covered by a lock, where scheduler threads are inserted that need to terminate their execution after the completion of their task execution. In a scenario where there is one thread that performs different asynchronous calls, we will always have maximum parallelism, because the main thread will be either in execution or scheduled as the last element in the list. The implementations of threadpool and async can be consulted here: `https://github.com/LucaRomano2/miosix-kernel/tree/thesis/miosix/e20/async`

3.4. Lazy task scheduling with LLVM transformations pass

In this part we present an approach that had the goal of applying the lazy task creation algorithm without creating a new thread for every asynchronous call. This approach resulted impossible to be applied on a standard C++ code for different reasons that we are going to explain later. We can formalize this approach, saying that, for a given function `f()` that performs an asynchronous call of the function `g()`, and naming `f'()` the continuation of `f()` after the asynchronous call, the goal was to schedule `f'()` in the `PriorityEventQueue`, and execute `g()` in the thread of `f()`. The pseudocode in 3.3 represents an example of that.

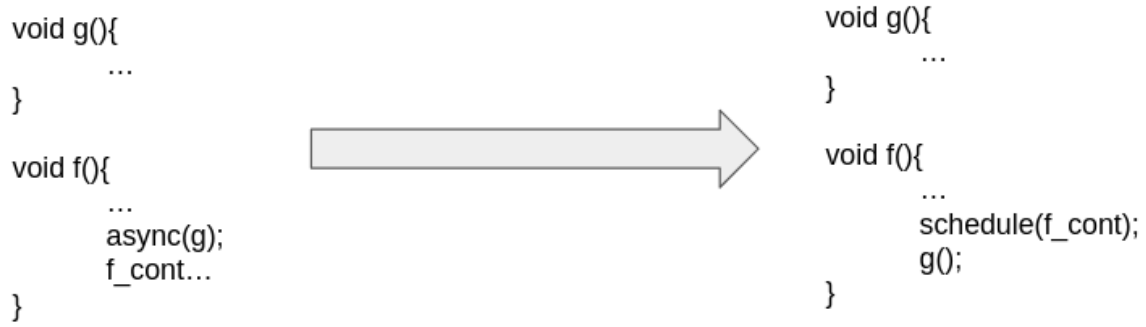

Figure 3.3: Pseudocode of the application of the lazy task creation in our context

We applied an LLVM transform pass to modify the source code in continuation passing style so that we can capture the continuation of the program after the line that perform the asynchronous call and scheduling it as a function in the `PriorityEventQueue`. The image 3.3 shows an example of how standard code is modified in a way that we can have the behavior described above.

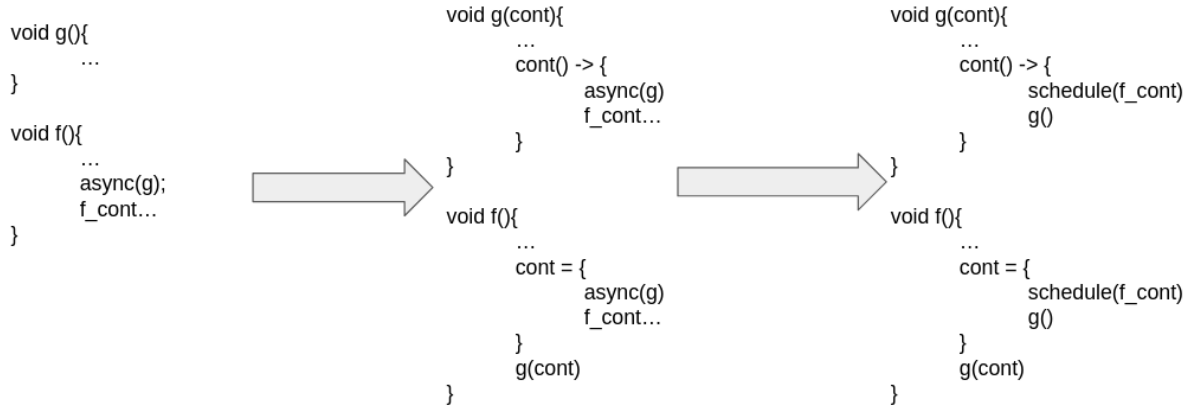

Figure 3.4: Transformation of a standard code in CPS, and then application of the lazy task creation

We reach a functioning implementation of that, but under very strict assumptions, including:

- Only `void` functions
- Only global variables
- Asynchronous calls are not performed inside loops or recursive functions

These assumptions are too restrictive, and this approach cannot be extended to every C++ program. For these reasons, we did not proceed with this implementation.

4 | Tests and experimental results

In this chapter, we are going to run different tests on our implementations: Firstly, we show that the priority event queue works as it is expected, giving precedence to call with higher priority. Then we are going to run tests cases to compare the execution time of our implementations of asynchronous calls by the threadpool and lazy approaches with the one of the standard library, in both Miosix and Linux. Lastly, we compare our implementations on two test cases taken from the Cilk Plus paper.
The tests on Miosix are executed on a Raspberry Pi Pico W with 2 ARM cores running at 200 MHz, 264 KB of RAM and 2 MB of flash.

4.1. Priority event queue

In this part, we run two test cases to demonstrate that the implementation of the priority event queue, presented in 3, works as expected, executing the asynchronous calls in parallel on two cores and in a FIFO order, and switching execution in the case when calls with higher priority are scheduled.

Considering the code on Listing 4.1 that simply count from `0` to `N - 1` and print also an `num` to identify the function:

```
1  #define N 5
2
3  void counterTask(int num)
4  {
5      iprintf("std::func%d_start\n", num);
6      for(int i = 0; i < N; i++){
7          iprintf("counter%d: %d\n", num, i);
8      }
9      iprintf("std::func%d_finished\n", num);
10 }
```

Listing 4.1: Implementation of counterTask function

Executing the code with the main function on Listing 4.2

```
int main()
{
    iprintf("__main__\n");
    PriorityEventQueue* q = new PriorityEventQueue();
    q->startRun();
    q->post([]() { counterTask(1); }, Priority(1));
    q->post([]() { counterTask(2); }, Priority(1));
    q->post([]() { counterTask(3); }, Priority(1));
}
```

Listing 4.2: main function for testing code on Listing 4.1

It gives the output reported on Listing 4.3.

```
__main__
std::func1_start
std::func2_start
counter1: 0
counter2: 0
counter1: 1
counter2: 1
counter1: 2
counter2: 2
counter1: 3
counter2: 3
counter1: 4
counter2: 4
std::func1_finished
std::func2_finished
std::func3_start
counter3: 0
counter3: 1
counter3: 2
counter3: 3
counter3: 4
std::func3_finished
```

Listing 4.3: Execution output of code on Listing 4.2

We can notice that calls with `num` equal to `0` and `1` are executed in parallel in the two

cores, then when one of them finishes, the call with `num` equal to `3` is scheduled.

Considering the same function on Listing 4.1 called by the code on Listing 4.4:

```
int main()
{
    iprintf("__main__\n");
    PriorityEventQueue* q = new PriorityEventQueue();
    q->startRun();
    q->post([]() { counterTask(1); }, Priority(1));
    q->post([]() { counterTask(2); }, Priority(1));
    q->post([]() { counterTask(3); }, Priority(2));
    q->post([]() { counterTask(4); }, Priority(3));
}
```

Listing 4.4: main function for testing code on Listing 4.1 with different priorities

it reports the output on Listing 4.5.

```
__main__
std::func4_start
std::func3_start
counter4: 0
counter3: 0
counter4: 1
counter3: 1
counter4: 2
counter3: 2
counter4: 3
counter3: 3
counter4: 4
counter3: 4
std::func4_finished
std::func3_finished
std::func1_start
std::func2_start
counter1: 0
counter2: 0
counter1: 1
counter2: 1
```

```
counter1: 2
counter2: 2
counter1: 3
counter2: 3
counter1: 4
counter2: 4
std::func1_finished
std::func2_finished
```

Listing 4.5: Result output of code on Listing 4.4

We can notice that even if calls with `num` equals to `3` and `4` are scheduled after the calls with `nums` equals to `1` and `2`, having higher priority they are executed before them and only after their finish, the two calls with lower priority can execute.

4.2. Execution time comparison on Miosix

In this part, we run some test cases on the implementation of `threadpool::async()` and `lazy::async()` compared with `std::async()`. We consider the implementation in Listing 4.6 that simulates a caller thread that performs different asynchronous calls and gets their results.

```
#define N 100
#define T 100

int f(int n)
{
    int j = 0;
    for(int i=0; i<N; i++){
        j = (j + i) % 1000000;
    }
    return j;
}

int main()
{
    iprintf("__main__\n");
    start_async();
    long long time_before, time_after;
    std::list<future<int>> v;
```

```
19      time_before = getTime();
20      for(int i = 0; i < T; i++){
21          v.push_back(async(f, i));
22      }
23      for(auto& li : v){
24          li.get();
25      }
26      time_after = getTime();
27      iprintf("%lld\n", time_after - time_before);
28  }
```

Listing 4.6: Test program for comparing std, threadpool and lazy approaches

The code is executed with different values of the variable `T` to see how they perform considering different numbers of asynchronous calls.
Execution time is reported in nanoseconds.
The result are reported in Table 4.1 and Figure 4.1

T	std	threadpool	lazy
5	737938	333938	510000
25	2423063	1813417	1920355
50	4439583	3645834	3759709
100	9003563	7455542	7324480
150	crash	11248042	10975459
200	crash	15093792	14505313

Table 4.1: Execution time (ns) comparison between std, threadpool and lazy on Miosix

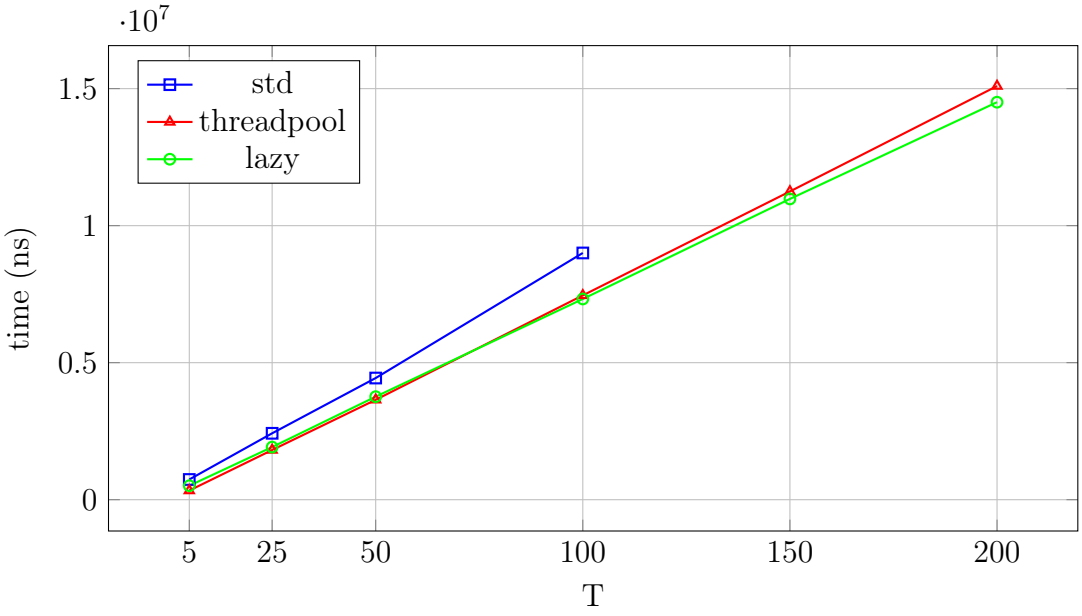

Figure 4.1: Execution time (ns) comparison between std, threadpool and lazy on Miosix)

4.3. Execution time comparison on Linux

In this part, we run the tests on Listing 4.6 with the threadpool and lazy task creation implementations in the Linux environment in computer with the following hardware: CPU AMD Ryzen 5 3500U with Radeon Vega Mobile Gfx, RAM 8 GB, OS Ubuntu 24.04.2 LTS. The implementations of `threadpool::async()`, `lazy::async()`, and the code on Listing 4.6 are slightly modified to suit a standard C++ code, without the Miosix API.

In this test, we use the `N` variable of Listing 4.6 equal to `1000` because resources are more powerful in this case. Execution time is reported in milliseconds. The results are reported in Table 4.2 and Figure 4.2

T	std	threadpool	lazy
5000	399	11	11
10000	768	23	17
15000	1160	40	26
20000	1528	62	38

Table 4.2: Execution time (ms) comparison between std, threadpool and lazy on Linux

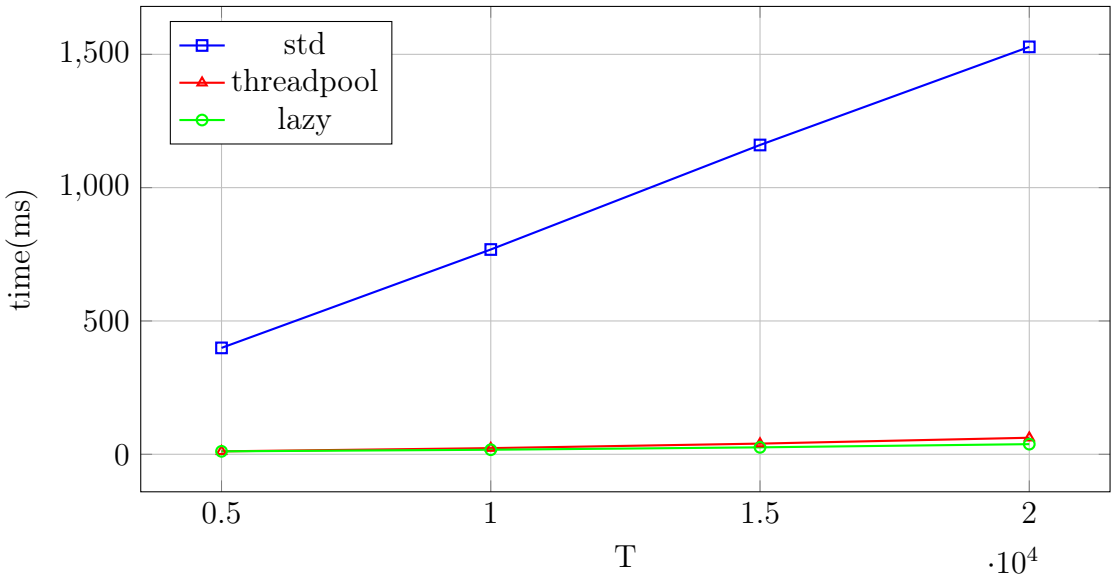

Figure 4.2: Execution time (ms) comparison between std, threadpool and lazy on Linux

4.4. Memory usage

In the tests on Listing 4.6 the runtime memory usage of the threadpool and lazy approaches is equal to `O(NUM_PRIORITIES * NUM_CORE * THREAD_SPACE)`. while with the standard implementation is equal to `O(T * THREAD_SPACE)`.
Considering that `NUM_PRIORITIES` equal to `4` and `NUM_CORE` equal to `2`, we have a significant improvement in the memory usage with respect to the standard approach that, in these cases is proportional to `T` that is the number of scheduled asynchronous calls, while with our implementations it is a low constant value.

4.5. Cilk Plus benchmark on Miosix

In these test we run two benchmark taken from [34] with the goal of testing our implementation on a more realistic scenario. Both implementations have been converted from the C++ code using cilk plus notation to the use of `async()`.
The code in Listing 4.7 implements an optimization of the reassembly phase of EFFT using cilk plus.

```
int scatter_index[numofbins];
find_scatter_index(scatter_index,numofsplits);
cilk_for(int j = 0; j < numofbins; j++)
    for (long i = 0; i < binsize; i++)
        tempArray[scatter_index[j]*binsize+i] = dataArray[i*
           numofbins+j];
```

Listing 4.7: EFFT scatter optimized implementation with cilk plus from [34]

The code in Listing 4.8 is a translation of the code in Listing 4.7 using `async()`.

```
int compute_temp(int j){
    for (long i = 0; i < binsize; i++) {
        tempArray[scatter_index[j]*binsize + i] = dataArray[i*
           numofbins + j];
    }
    return 0;
}

for (int j = 0; j < numofbins; j++){
    v.push_back(lazy::async(compute_temp, j));
}

for(auto& li : v){
    li.get();
}
```

Listing 4.8: Transformation of the code on Listing 4.7 with async()

Considering the values:

```
N = 4096
numofsplits = 3
numofbins = 8
```

```
binsize = 512
```

The execution times computed by the two approaches compared to the standard library are the following:

- std: 983396 ns
- threadpool: 526521 ns
- lazy: 499771 ns

The code in Listing 4.9 implements an optimization of the scatter phase of EFFT using cilk plus.

```
void BasicReassemble(float* evens, float* odds,
float* target, const long int size) {
    const float trigconst = -3.14159265359f/size;
    target[0]=evens[0]+odds[0];
    target[1]=evens[0]-odds[0];
    target[size ]=evens[1];
    target[size+1L]=-odds[1];

    for(long int k = 1L; k < size/2L; k++) {
        float cosk = cosf(k*trigconst);
        float sink = sinf(k*trigconst);
        target[2*k]=evens[2*k]+odds[2*k]*cosk-odds[2*k+1]*sink;
        target[2*k+1]=evens[2*k+1]+odds[2*k]*sink+odds[2*k+1]*cosk
           ;
        target[2*size-2*k]=evens[2*k]-odds[2*k]*cosk+odds[2*k+1]*
           sink;
        target[2*size-2*k+1]=-evens[2*k+1]+odds[2*k]*sink+odds[2*k
           +1]*cosk;
    }
}
```

Listing 4.9: EFFT reassembly optimized implementation with cilk plus from [34]

The code in Listing 4.10 translate the code in Listing 4.9 using `async()`.

```
int compute_target(long kk){
    float coslist[kTILE] __attribute__((aligned(64)));
    float sinlist[kTILE] __attribute__((aligned(64)));
    for(int i = 0; i < kTILE; i++){
```

```
5           coslist[i] = cosf((kk + i) * trigconst);
6           sinlist[i] = sinf((kk + i) * trigconst);
7       }
8       for(long k = kk; k < kk + kTILE && k < size_/2; k++){
9           target[2*k]   = evens[2*k] + odds[2*k]*coslist[k-kk] -
               odds[2*k+1]*sinlist[k-kk];
10          target[2*k+1] = evens[2*k+1] + odds[2*k]*sinlist[k-kk] +
               odds[2*k+1]*coslist[k-kk];
11      }
12      return 0;
13  }
14
15  void basic_reassemble(){
16      start_async();
17      for(long i = 0; i < 2*size_; i++){
18          evens[i] = i * 1.0f;
19          odds[i]  = (2*size_ - i) * 1.0f;
20      }
21      list<lazy::future<int>> v;
22      for(long kk = kTILE; kk < size_/2; kk += kTILE){
23          v.push_back(lazy::async(compute_target, kk));
24      }
25      for(auto& li : v){
26          li.get();
27      }
28  }
```

Listing 4.10: Transformation of the code on Listing 4.9 with async()

Considering the values:

```
long size = 16;
float trigconst = -3.14159265359f / size
const long kTILE = 4
```

The execution time computed by the two approaches compared to the standard library are the following:

- std: 738292 ns
- threadpool: 497876 ns

- lazy: 516416 ns

5 | Conclusion and future work

Our work result in the implementation of an event queue that support scheduling of asynchronous calls based on priority. This work has been applied on Miosix, a real-time operating systems that supports the development of C++ code. The priority event queue has been applied to improve asynchronous calls with respect to the implementation of the C++ standard library. We implemented two versions: threadpool and lazy, which both result in a significant improvement in the execution speed and memory usage of asynchronous calls. These implementations have also been applied to a Linux environment, resulting in a more significant improvement in the execution speed.
Our work optimizes scenarios where a thread performs a high number of asynchronous calls, and in a scenario where threads continue to perform asynchronous calls in a recursive way, our works behaves as the implementations of the C++ standard library, creating a thread for every call. The future work will be trying to optimize the lazy task creation approach in a way where recursive asynchronous calls are executed without creating new threads, resulting in an improvement also in this scenario.

Bibliography

[1] Giorgio Buttazzo. *Hard Real-Time Computing Systems: Predictable Scheduling Algorithms and Applications*. Springer, Cham, Switzerland, 4 edition, 2024.

[2] Carlos Garre, Domenico Mundo, Marco Gubitosa, and Alessandro Toso. Performance comparison of real-time and general-purpose operating systems in parallel physical simulation with high computational cost. volume 1, 04 2014. doi: 10.4271/2014-01-0200.

[3] Changpeng Fan. Mmoss: Soft real-time operating system support in a multimedia communication subsystem. *Annual Review in Automatic Programming*, 18:127–131, 1994. ISSN 0066-4138. doi: https://doi.org/10.1016/0066-4138(94)90022-1. URL `https://www.sciencedirect.com/science/article/pii/0066413894900221`.

[4] B. Srinivasan, S. Pather, R. Hill, F. Ansari, and D. Niehaus. A firm real-time system implementation using commercial off-the-shelf hardware and free software. In *Proceedings. Fourth IEEE Real-Time Technology and Applications Symposium (Cat. No.98TB100245)*, pages 112–119, 1998. doi: 10.1109/RTTAS.1998.683194.

[5] Laurent George, Nicolas Rivierre, and Marco Spuri. Preemptive and Non-Preemptive Real-Time UniProcessor Scheduling. Research Report RR-2966, INRIA, 1996. URL `https://inria.hal.science/inria-00073732`. Projet REFLECS.

[6] C.E. Love and H.F. Jordan. An investigation of static versus dynamic scheduling. In *[1990] Proceedings. The 17th Annual International Symposium on Computer Architecture*, pages 192–201, 1990. doi: 10.1109/ISCA.1990.134525.

[7] Weirong Wang, Aloysius K. Mok, and Gerhard Fohler. Pre-scheduling: Integrating offline and online scheduling techniques. In Rajeev Alur and Insup Lee, editors, *Embedded Software*, pages 356–372, Berlin, Heidelberg, 2003. Springer Berlin Heidelberg. ISBN 978-3-540-45212-6.

[8] Nikos Andrikos and Luciano Lavagno. Optimal and heuristic scheduling algorithms for asynchronous high-level synthesis. In *2011 17th IEEE International Symposium*

on Asynchronous Circuits and Systems, pages 13–21, 2011. doi: 10.1109/ASYNC.2011.18.

[9] Alan A. Bertossi and Andrea Fusiello. Rate-monotonic scheduling for hard-real-time systems. *European Journal of Operational Research*, 96(3):429–443, 1997. ISSN 0377-2217. doi: https://doi.org/10.1016/S0377-2217(97)83306-1. URL `https://www.sciencedirect.com/science/article/pii/S0377221797833061`.

[10] Mario Günzel, Kuan-Hsun Chen, and Jian-Jia Chen. Edf-like scheduling for self-suspending real-time tasks. *CoRR*, abs/2111.09725, 2021. URL `https://arxiv.org/abs/2111.09725`.

[11] cppreference.com. std::async – cppreference.com. `https://en.cppreference.com/w/cpp/thread/async.html`.

[12] Eric Niebler. P2470r0: Async execution in standard c++ — the standard executors effort. `https://www.open-std.org/jtc1/sc22/wg21/docs/papers/2021/p2470r0.pdf`, 2021. ISO/IEC JTC1/SC22/WG21 C++ Standards Committee Paper; accessed 2026-02-23.

[13] Miosix: os kernel for embedded ystems. `https://miosix.org/`, 2025.

[14] Federico Terraneo and Daniele Cattaneo. Fluid kernels: Seamlessly conquering the embedded computing continuum. *IEEE Transactions on Computers*, 74(12):4050–4064, 2025. doi: 10.1109/TC.2025.3605745.

[15] Martina Maggio, Federico Terraneo, and Alberto Leva. Task scheduling: A control-theoretical viewpoint for a general and flexible solution. *ACM Trans. Embed. Comput. Syst.*, 13(4), March 2014. ISSN 1539-9087. doi: 10.1145/2560015. URL `https://doi.org/10.1145/2560015`.

[16] Rahim Mammadli, Marija Selakovic, Felix Wolf, and Michael Pradel. Learning to make compiler optimizations more effective. *CoRR*, abs/2102.13514, 2021. URL `https://arxiv.org/abs/2102.13514`.

[17] Weisong Sun, Chunrong Fang, Yun Miao, Yudu You, Mengzhe Yuan, Yuchen Chen, Quanjun Zhang, An Guo, Xiang Chen, Yang Liu, and Zhenyu Chen. Abstract syntax tree for programming language understanding and representation: How far are we?, 2023. URL `https://arxiv.org/abs/2312.00413`.

[18] C. Lattner and V. Adve. Ll: a compilation framework for lifelong program analysis & transformation. In *International Symposium on Code Generation and Optimization, 2004. CGO 2004.*, pages 75–86, 2004. doi: 10.1109/CGO.2004.1281665.

[19] David Chisnall. Modern intermediate representations (ir). https://llvm.org/devmtg/2017-06/1-Davis-Chisnall-LLVM-2017.pdf, 2017. Presented at the LLVM Developers Meeting / Summer School, Paris, France.

[20] The LLVM Project. Clang: a c language family frontend for llvm. https://clang.llvm.org/, 2026. Accessed: 2026-02-26.

[21] The Rust Project Developers. What is rustc? https://doc.rust-lang.org/rustc/what-is-rustc.html, 2026. Accessed: 2026-02-26.

[22] Jeremy Gibbons. Continuation-passing style, defunctionalization, accumulations, and associativity. *The Art, Science, and Engineering of Programming*, 6(2), November 2021. ISSN 2473-7321. doi: 10.22152/programming-journal.org/2022/6/7. URL http://dx.doi.org/10.22152/programming-journal.org/2022/6/7.

[23] Arash Sal Moslehian. An introduction to asynchronous programming in rust and a high-level overview of tokio's architecture. https://moslehian.com/posts/2023/1-intro-async-rust-tokio/.

[24] Klaus Jansen and Lorant Porkolab. Preemptive scheduling on dedicated processors: Applications of fractional graph coloring. In Mogens Nielsen and Branislav Rovan, editors, *Mathematical Foundations of Computer Science 2000*, pages 446–455, Berlin, Heidelberg, 2000. Springer Berlin Heidelberg. ISBN 978-3-540-44612-5.

[25] Burak Gökgür, Selin Özpeynirci, and Mutlu İpek Tanıl. Cooperative scheduling and subcontracting strategies for products with yield decay: A mixed-integer programming approach. *Expert Systems with Applications*, 300:130226, 2026. ISSN 0957-4174. doi: https://doi.org/10.1016/j.eswa.2025.130226. URL https://www.sciencedirect.com/science/article/pii/S0957417425038412.

[26] Tokio Project. Tokio — an asynchronous rust runtime. https://tokio.rs/, 2026. Accessed: 2026-02-27.

[27] Robert D. Blumofe and Charles E. Leiserson. Scheduling multithreaded computations by work stealing. *J. ACM*, 46(5):720–748, September 1999. ISSN 0004-5411. doi: 10.1145/324133.324234. URL https://doi.org/10.1145/324133.324234.

[28] cppreference.com. Coroutines (c++20). https://en.cppreference.com/w/cpp/language/coroutines, 2026. Accessed: 2026-02-27.

[29] E. Mohr, D. A. Kranz, and R. H. Halstead. Lazy task creation: A technique for increasing the granularity of parallel programs. *IEEE Trans. Parallel Distrib. Syst.*,

2(3):264–280, July 1991. ISSN 1045-9219. doi: 10.1109/71.86103. URL https://doi.org/10.1109/71.86103.

[30] D. A. Kranz, R. H. Halstead, and E. Mohr. Mul-t: a high-performance parallel lisp. In *Proceedings of the ACM SIGPLAN 1989 Conference on Programming Language Design and Implementation*, PLDI '89, page 81–90, New York, NY, USA, 1989. Association for Computing Machinery. ISBN 089791306X. doi: 10.1145/73141.74825. URL https://doi.org/10.1145/73141.74825.

[31] Tomas A. Lopez and Nobuyuki Yamasaki. Prioritized asynchronous calls for parallel processing on responsive multithreaded processor. In *2022 Tenth International Symposium on Computing and Networking (CANDAR)*, pages 46–55, 2022. doi: 10.1109/CANDAR57322.2022.00014.

[32] Kazutoshi Suito, Rikuhei Ueda, Kei Fujii, Takuma Kogo, Hiroki Matsutani, and Nobuyuki Yamasaki. The dependable responsive multithreaded processor for distributed real-time systems. *IEEE Micro*, 32(6):52–61, 2012. doi: 10.1109/MM.2012.88.

[33] Sanjoy Baruah, Vincenzo Bonifaci, and Alberto Marchetti-Spaccamela. The global edf scheduling of systems of conditional sporadic dag tasks. In *2015 27th Euromicro Conference on Real-Time Systems*, pages 222–231, 2015. doi: 10.1109/ECRTS.2015.27.

[34] Ryo Asai and Andrey Vladimirov. Intel cilk plus for complex parallel algorithms: "enormous fast fourier transform" (EFFT) library. *CoRR*, abs/1409.5757, 2014. URL http://arxiv.org/abs/1409.5757.

[35] Intel Corporation. Intel cilk plus. https://cilkplus.github.io/.

[36] John Mellor-Crummey. Shared-memory parallel programming with cilk plus. https://www.clear.rice.edu/comp422/lecture-notes/comp422-534-2020-Lecture4-CilkPlus.pdf.

List of Figures

List of Tables

Acknowledgements

A huge thanks to prof. Federico Terraneo and Tomas Antonio Lopez that helped me a lot for implementing and completing the thesis.
A special thanks to my family and my friends that supported me through all these years.