POLITECNICO
MILANO 1863

SCUOLA DI INGEGNERIA INDUSTRIALE
E DELL'INFORMAZIONE

Executive Summary of the Thesis

Priority aware asynchronous programming for embedded real-time systems

Laurea Magistrale in Computer Science and Engineering - Ingegneria Informatica

Author: Luca Romanò

Advisor: Prof. Federico Terraneo

Co-advisor: Tomas Antonio Lopez

Academic year: 2025-2026

---

1. Introduction

The goal of this work is to provide efficient implementations of asynchronous calls in C++ targeting real-time operating systems suited for embedded systems, which are characterized by low resources. We applied our work on Miosix, a real-time operating system that supports C++ development, and also most of its kernel is written in C++. Our work result in two distinct implementation that can be used with almost the same interface of `std::async()` and they are optimized in terms of average execution time and average memory usage. The issue with the original implementation of `std::async()` is that it creates a new thread for every call. Being in a context with scarce resources, this can lead to high resource consumption, especially if multiple asynchronous calls are scheduled sequentially. Our work has the main goal of do not create a new thread for every asynchronous call and execute all the calls in an executor that have predefined threads that execute the calls. This led to a faster execution time and lower memory usage. Then, working on a real-time operating system, our work supports scheduling asynchronous calls by their priority that they inherit from the called thread.

2. Background

2.1. Real-time operating systems

Real-time operating systems [2] 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 to the system. Real-time operating systems (RTOS) are implemented to meet the requirements of real-time systems. They differ from general-purpose operating systems (GPOS), 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. A typical approach in Real-times operating systems to meet the deadline require-

ments is to translating the tasks deadline or period in a priority, where shorter tasks will have higher priority, and their scheduling algorithms schedules tasks based on that.

2.2. Miosix

Miosix [1] 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++. 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`. The central part of the kernel is the scheduler [6]. 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. Miosix supports three scheduling algorithms: priority, EDF, and I+Pi [4]. 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. Miosix also supports multicore scheduling in its unstable version, where threads can run in different cores. The multicore scheduler works like the priority scheduler, with a global list that stores all of the active threads for each priority level, and each core, when the periodic interrupt occurs, execute the thread with higher priority in front of its list. If a thread does not finish before the next scheduling interrupt, it is scheduled again in the back of the list with his priority, leading to a round robin scheduling, if multiple active threads with the same priority are presents.

2.3. Asynchronous calls in C++

`std::async()` [3] is a C++ standard library function that allows executing a function in parallel with respect to the function that calls it. 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. When a `std::async()` call is performed, a new thread is created to execute it. This will then be executed following the scheduling policy of the operating system. When it finishes it 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.

2.4. Lazy task creation

The work of [5] proposes the technique of *lazy task creation*, implemented on Mul-T, an efficient parallel implementation of Scheme. The main goal of this implementation is to create tasks retroactively, only when resources become available. The 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. 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.

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 semantics of `std::async()`, where the

scheduled function inherits the priority from the thread that calls it.
We implemented two different versions, one based on a standard threadpool approach, and the other one based on the lazy task creation algorithm. 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 the scheduled call execute, and then will schedule the resumption 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 their result can be stored in the `future<T>` 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 parts, 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. The code can be consulted at: `https://github.com/LucaRomano2/miosix-kernel/tree/thesis/miosix/e20`

3.1. Priority event queue

The priority event queue is implemented to support the scheduling of asynchronous calls with an associated priority and running them in a pre-existing executor.
The priority event queue, when is instantiated, spawns C thread for every priority level, resulting in having `C * NUM_PRIORITES` threads where C is the number of CPU cores in the system. Each thread executes a `void run()` method that takes the function to execute by the list of scheduled calls with its priority and executes them in a FIFO order, and the Miosix scheduler will manage the execution of the threads based on their priority. A pseudo-code of the main methods of the `PriorityEventQueue`is shown in Listing 1:

```
1  void post(std::function<void ()> event,
       Priority prio){
2      lock();
3      events[getInt(prio)].push_back(event);
4      unlock();
5  }
6
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 1: Pseudocode of the priority event queue startRun, run and post methods implementation

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` with some differences to support mainly the `get()` method of the future class. A class `Task`, extended by the class `Task_<T>` to support saving in a list calls with different return types, is used instead of the `std::function<void ()>` in the priority event queue and the `run()` method will call the `execute()` method of `Task`, that is implemented in `Task_<T>` to execute the calls that have been taken from the front of the list as in the priority event queue implementation, and store its result. The `get()` method of `threadpool::future<T>` will call the `get()` method of its associated `Task_` that will check if the execution of the task is finished, and if is the case it return the return value computed by the task execution,

otherwise it will go in `wait()` status and then it will be woken up when the execution is finished, and it will return the returned value. In the case when an asynchronous call perform another asynchronous call, and then when it perform the `get()` operation, it needs to `wait()` and we would lose execution time because a thread scheduler would go in `wait()` status. To avoid this, when a thread that belong to the scheduler perform a `get()` operation on another `future` and it needs to go in `wait()`, a substitute scheduler will be created to replace it, and when the main thread will be woken up, the thread that is replacing it will finish the execution of one task and then it will terminate. When the `threadpool::future<T>` goes out of scope, it check if the execution of its task has been finished by the scheduler, and if it the case it deallocate space on the heap of it associated `Task_<T>`, otherwise it marks the task indicating that it cannot return its value, and it will delete it when its execution is finished in the priority event queue scheduler. With this approach, we avoid the problem of memory leaks.

3.3. Lazy task scheduling

The implementation of `lazy::async()` and `lazy::future<T>` follow the same approach of the threadpool implementation with some differences, expecially for handling the implementation of the lazy task creation approach. `Task` is extended in `TaskStart<T>` and `TaskRestat`. `TaskStart<T>` works like `Task_<T>` in the threadpool implementation, with a `execute()` method that runs the task and store its result, and the `get()` method that return the computed result, when available. `TaskRestat` takes as argument a pointer to the thread that has been put in `wait()` status and it has the `execute()` method that wake up the the thread passed as argument. The `run()` method of the priority event queue takes from the event queue list a pointer to `Task`, that can be `TaskStart<T>` or `TaskRestart`, and call the `execute()` method that both override from `Task`. The post method, to apply the lazy task creation algorithm, when called, if the number of executing asynchronous calls is lower than the number of cpu cores it simply push in the list the task with a pointer to an instance of `TaskStart<T>`, otherwise if the number of executing calls is equal to the number of cores it firstly push in the list a pointer to `TaskStart<T>`, then it push `TaskRestart` passing the current thread as argument, and then the current thread goes to sleep. 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.

4. Tests and experimental results

In this part 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 Miosix. 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.

5. Priority event queue

In this part, we run two test cases to demonstrate that the implementation of `PriorityEventQueue`, works as expected, executing the asynchronous calls in parallel on two cores in a FIFO order, and switching execution in the case when calls with higher priority are scheduled. Considering the code on Listing 2 that simply count from `0` to `N - 1` and print also an `num` to identify the function:

```
#define N 5

void counterTask(int num){
    iprintf("std::func%d_start\n", num);
    for(int i = 0; i < N; i++){
        iprintf("counter%d: %d\n", num, i);
    }
    iprintf("std::func%d_finished\n", num);
}
```

Listing 2: Implementation of counterTask()

Executing the code with the main function on Listing 3

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

```
    q->post([]() { counterTask(2); },
    Priority(1));
    q->post([]() { counterTask(3); },
    Priority(1));
}
```

Listing 3: Main function to test the code on Listing 2

It gives the output reported on Listing 4.

```
__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: Execution output of the code on Listing 3

Where 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 its execution, the call with `num` equal to `3` will start its execution. Considering the same function on Listing 3 called by Listing 5.

```
int main(){
    iprintf("__main__\n");
    auto* 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 5: Main function to the the code on Listing 2 with different priority levels

It reports the output on Listing 6:

```
__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 6: Execution output of the code on Listing 5

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 the are executed before them and only after their finish, the two calls with lower priority can execute.

6. Execution time comparison

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 7 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();
    Thread::sleep(5000);
    long long time_before, time_after;
    std::list<future<int>> v;
    time_before = getTime();
    for(int i = 0; i < T; i++){
        v.push_back(async(f, i));
    }
    for(auto& li : v){
        li.get();
    }
    time_after = getTime();
    iprintf("%lld\n", time_after - time_before);
}
```

Listing 7: 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 number of asynchronous calls.
Execution time is reported in nanoseconds.
The result are reported in Table 1 and Figure 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 1: Execution time (ns) comparison between std, threadpool and lazy on Miosix

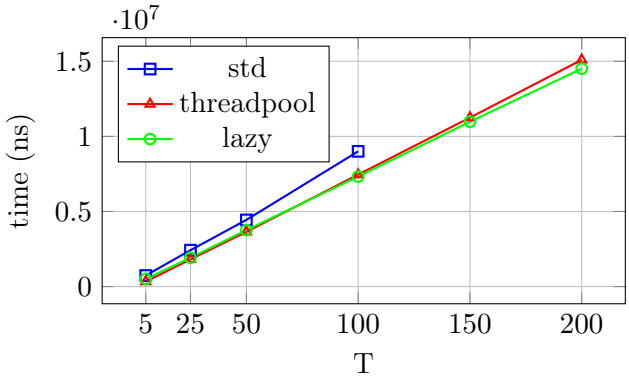

Figure 1: Execution time (ns) comparison between std, threadpool and lazy on Miosix)

7. Conclusion

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 system that supports the development of C++ code. The priority event queue has been applied to improve asynchronous calls, considering the implementation of the C++ standard library. We implemented two versions: threadpool and lazy, which both result in an improvement in the average execution speed and average memory usage of asynchronous calls.

8. 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.

References

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

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

[3] cppreference.com. std::async – cppreference.com. `https://en.cppreference.com/w/cpp/thread/async.html`, 2024.

[4] 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.

[5] 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.

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