Dealing with Time: Measuring Real-Time Capabilities of Lingua Franca

Efsane Soyer

Electrical Engineering and Computer Sciences
University of California, Berkeley

Technical Report No. UCB/EECS-2024-131
http://www2.eecs.berkeley.edu/Pubs/TechRpts/2024/EECS-2024-131.html

May 17, 2024

Dealing with Time: Measuring Real-Time Capabilities of Lingua Franca

by

Efsane Soyer

A thesis submitted in partial satisfaction of the

requirements for the degree of

Master of Science

in

Electrical Engineering and Computer Science

in the

Graduate Division

of the

University of California, Berkeley

Committee in charge:

Professor Edward A. Lee, Chair
Prabal Dutta

Spring 2024

Dealing with Time: Measuring Real-Time Capabilities of Lingua Franca

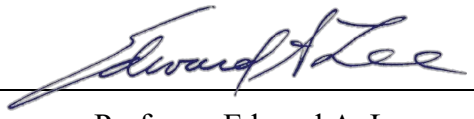
by Efsane Soyer

Research Project

Submitted to the Department of Electrical Engineering and Computer Sciences, University of California at Berkeley, in partial satisfaction of the requirements for the degree of **Master of Science, Plan II**.

Approval for the Report and Comprehensive Examination:

Committee:

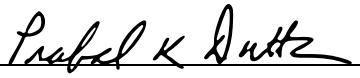


Professor Edward A. Lee
Research Advisor

May 16, 2024

(Date)

* * * * * * *



Professor Prabel Dutta
Second Reader

May 16, 2024

(Date)

Dealing with Time: Measuring Real-Time Capabilities of Lingua Franca

Abstract

Dealing with Time: Measuring Real-Time Capabilities of Lingua Franca

by

Efsane Soyer

Master of Science in Electrical Engineering and Computer Science

University of California, Berkeley

Professor Edward A. Lee, Chair

Precise timing, reproducibility, and concurrency play an important role in cyber-physical systems. Lingua Franca, or LF, is a reactor-based coordination language that can exploit parallelism while preserving determinism and exposes time-based semantics. These features make LF a suitable choice for real-time systems. By conducting experiments and analyzing the results, this study seeks to provide valuable insights into the real-time capabilities of Lingua Franca. We introduced two case studies: timer utilization and periodic tasks. The study on timer utilization investigated the relationship between timing behavior and task execution time, showing that LF can achieve high utilization rates of up to 95% on a Linux operating system while effectively keeping lags under 20 microseconds using the lag controller. The periodic tasks study explored various scheduling scenarios, examining how task orders, periods, and offsets affect meeting deadlines. This case highlighted the need for further efforts in implementing deadline monotonic and earliest deadline first schedulers. By leveraging these insights, we have developed future optimization strategies. The results of this thesis will kickstart the empirical timing analysis of Lingua Franca and offer valuable information for enhancing its real-time capabilities in the future.

Contents

List of Figures

List of Tables

Acknowledgments

The work in this report was supported in part by the National Science Foundation (NSF), award #CNS-2233769 (Consistency vs. Availability in Cyber-Physical Systems).

I would express my most sincere gratitude to my advisor, Professor Edward Lee for his mentorship over the years; he is a true source of inspiration. He is the perfect mentor giving enough guidance while providing the freedom to build my path.

I would like to thank Professor Prabal Dutta for agreeing to be my second reader and for his feedback and input on this thesis.

This project would not be possible without the contributions of my colleagues in the Lingua Franca group across multiple organizations and their work in the development of the Lingua Franca compiler, runtime, schedulers, and various features. I want to acknowledge Erling Rennemo Jellum's work on enclaves which was critical for parts of this thesis and specifically thank him for his reviews of the lag controller. I want to thank Francesco Paladino for his continuous help and feedback on this thesis and his contributions to the periodic tasks case study.

I would like to thank all my friends and my parents for their support and understanding throughout my academic journey in Berkeley.

Chapter 1

Introduction

Cyber-physical systems (CPS) have become indispensable in today's society, serving as the backbone for a wide array of crucial applications. They power sensor networks that capture and analyze data, facilitate the operation of autonomous vehicles navigating our roads, drive industrial automation processes, etc. Precise timing, reproducibility, and concurrency play an important role in CPS [1, 4]. The timing aspect is critical to cyber-physical systems' operations because any delay in component response can lead to serious consequences such as reduced system performance, safety hazards, or even complete failures with potentially catastrophic outcomes. Reproducibility is crucial for testing, debugging, and ensuring the reliability of the system's behavior. Furthermore, concurrency is essential to facilitate the simultaneous execution of multiple tasks and enhance system efficiency, particularly as their computational demand grows.

There is a need for reliable and accurate methods and tools to ensure runtime guarantees are met consistently. Lingua Franca (LF), a reactor-oriented coordination language serves as a promising solution to address these challenges through its deterministic concurrency model, explicit management of timing, and the ability to exploit parallelism [11]. The performance of Lingua Franca [11, 7] and the correctness of its deterministic properties [15, 7] are well studied in prior work. The purpose of this master's thesis is to measure the real-time capabilities of Lingua Franca, specifically its ability to provide precise timing predictability and limitations of its schedulers.

Measuring the real-time capabilities and conducting a timing analysis of a system is inherently challenging since the timing behavior of a system gets affected by every layer of abstraction from synchronous digital systems to models of computation. Let us take an LF program with one reaction that doesn't perform any computation and gets triggered by a timer with a 1-millisecond period. Upon measuring the delay between the actual trigger time and the expected trigger time on a Raspberry Pi 4 Model B, it was observed that there is an average delay of 80 microseconds, even reaching a maximum of 3.5 milliseconds. This non-constant delay can be speculatively caused by the Linux kernel, processor power management system, LF runtime, clock synchronization, interference with another process, etc. The differences between these causes are subtle and hard to measure independently. For

instance, if the processor is going into idle mode due to the lack of computation, this delay measurement can be caused by the wake-up routine overhead of the CPU, but if there is interference by another process or kernel, it might be due to scheduling and context switching overheads. Therefore, running timing analysis requires understanding the timing constraints at each level and quantifying the effects of various factors such as processor speed, kernel delays, task scheduling, etc.

This thesis aims to investigate and measure the real-time performance of Lingua Franca in the context of cyber-physical systems. By conducting experiments and analyzing the results, we aim to provide valuable insights into the suitability of Lingua Franca for real-time applications. Moreover, the thesis will explore potential optimizations and best practices for utilizing Lingua Franca in real-time systems to enhance its performance and reliability.

The rest of the paper is structured as follows. Chapter 2 provides a general overview of Lingua Franca and its runtime schedulers, as well as a primer on Linux schedulers. Chapter 3 talks about the earlier micro-benchmarks that build the baseline on the measurement strategy and platform configurations and introduces the lag controller optimization. Chapter 4 discussed two case studies that evaluate the real-time capabilities of Lingua Franca in different application scenarios. We describe the future work in chapter 5 and conclude in chapter 6.

Chapter 2

Background

2.1 Lingua Franca

Lingua Franca is a reactor-oriented, polyglot, coordination language designed for modeling, simulating, and implementing real-time and cyber-physical systems [8]. Reactors can be described as deterministic actors with discrete-event execution semantics and explicitly declared ports and connections [9]. LF provides a deterministic concurrency model to mainstream programming languages that allows them to automatically utilize opportunities to leverage parallelism. This enables developers to build concurrent systems that are efficient and can scale well to a large number of cores or nodes in a distributed system without worrying about uncaught race conditions. Lingua Franca aims to address the challenges of developing and maintaining complex real-time systems by providing a clear and efficient means of expressing timing constraints and coordinating the interactions between different components.

Lingua Franca assigns each event a logical timestamp that does not advance during the execution of the reaction body [11]. These timestamps create a logical timeline which is used to order events and ensure deterministic execution. When the event with the lowest logical time is getting processed, the Lingua Franca scheduler determines all the reactions that are triggered by the event. In the current implementation strategy, if multiple reactions within the same reactor are triggered by logically simultaneous events, i.e. the events that share the same logical timestamp, they will always be invoked deterministically in the order they are defined. Reactions can also invoke downstream reactions via port connections. Each layer on this reaction chain is assigned a level; any reactions with the same level can be executed in parallel safely since they do not depend on each other. The current scheduler implementation is designed such that only when all reactions in the current level are completed, the next level of reactions can be scheduled for execution. Reactions that can be safely executed in parallel can be executed by the LF threads, called workers. The runtime environment keeps a thread pool of workers and maps the ready-to-execute reactions to these workers. The number of workers in the system can be determined by the developer.

Lingua Franca was built on top of the synchronous languages model where reactions are executed logically instantaneously. However, in practice, this assumption can cause independent reactors to block the execution of one another. For instance, let's assume two reactors that both have 50 millisecond periods, but the second reactor has a 1 millisecond offset. In the current runtime, the reactor with the 1-millisecond offset can only be invoked when the first reactor finishes executing. If the execution time of the first reactor is more than 1 millisecond, it will introduce a delay on the second one. To divide scheduling domains and introduce the idea of "local" logical time, the experimental enclaves feature of Lingua Franca is used. Reactors that are in different enclaves can advance their logical time independently of one another and have their own schedulers.

Lingua Franca deadlines are specified as a bound on the reaction invocation, not completion. In other words, if a reaction has a deadline Δ then it must be invoked by the physical time PT such that $PT \leq LT + \Delta$ where LT represents the logical time tag of the input trigger [10].

2.2 Schedulers

Lingua Franca offers an interface for programming parallelism in a deterministic and time-based manner. In this section, we will explore the scheduling mechanisms that can be utilized to leverage this parallelism for real-time tasks. Reactors are scheduled over two layers of scheduling in the C target: the runtime scheduler and the underlying operating system scheduler, such that the runtime scheduler operates on a higher layer of abstraction than the OS scheduler. The runtime scheduler keeps track of all scheduled future events, controls the advancement of logical time, and invokes any triggered reactions in the order specified by the dependency graph with a focus on maximizing parallelism [11]. The constraints the runtime scheduler needs to follow to ensure determinism and how it interacts with the runtime environment are outside the scope of this thesis and are detailed by Lohstroh et al. [9, 10]. The subsection 2.2.1 will discuss two of the runtime schedulers available in Lingua Franca and their limitations when dealing with real-time tasks. The subsection 2.2.2 and 2.2.3 will talk about Linux-based schedulers that will be used in the rest of the thesis.

2.2.1 Lingua Franca Schedulers

The Lingua Franca runtime offers three different scheduling strategies: Non-Preemptive (NP), Global Earliest Deadline First (GEDF), and an experimental adaptive scheduler, details of which are out of scope for this thesis. Due to two layers of scheduling mechanisms, none of these schedulers are preemptive or control how threads move across cores [10]. NP is the default scheduler when the LF program does not include any deadlines; it can be simplified as a last in first out (LIFO) scheduler.

The GEDF scheduler is a limited version of an EDF scheduler [3] such that when the semantics of LF allow for concurrent execution of multiple ready reactions with the same

level at a particular logical time tag, this scheduler will prioritize running the reaction with the earliest deadline [5]. The GEDF scheduler can only prioritize one reaction over the other if the following conditions are met:

- The two reactions should be within the same enclave.
- The two reactions should be triggered simultaneously, or in other words, the two reactions should have the same logical time tag since the logical time cannot advance until all reactions with that logical time tag are already executed.
- The two reactions should be on the same level. In other words, the current GEDF implementation uses a distinct priority queue for each level such that level base scheduling is prioritized over the deadline.
- The two reactions should have different deadlines, which allows the scheduler to determine the order of execution based on the urgency of the reactions.
- The two reactions should compete for the same worker thread since otherwise both can be scheduled at the same time on different threads.

Section 4.2 highlights the use cases and limitations of the GEDF scheduler.

2.2.2 Linux Completely Fair Scheduler

The Linux Completely Fair Scheduler (CFS) is the default process scheduler in the Linux kernel [12]. It ensures fairness and efficiency in distributing CPU resources among processes. Ingo Molnar, the author of CFS, summarizes its design goals as: "CFS basically models an 'ideal precise multitasking CPU' on real hardware" [6]. An ideal precise multitasking CPU can be characterized as one that provides equal CPU power to all processes [13]. Notably, CFS, unlike Round Robin (RR), defines ideal fairness in terms of processor power and not time. This means that if there is only one process running, it will take 100% of the processor power; two processes share the processor power equally, each receiving 50%; and so on.

The basic idea behind CFS is tracking CPU time per thread and scheduling threads to match the average rate of execution. Considering the desire to give more CPU time to some processors than others, the Linux CFS uses a concept called "weight" to assign priorities to processes. Each process is assigned a weight value, which determines the proportion of CPU time it receives compared to other processes. The scheduler's decisions are based on virtual runtime, which refers to the amount of CPU time a process has consumed, adjusted based on its weight. The implementation details of this are outside the scope of this paper.

2.2.3 Linux Real-Time Scheduler

The Linux Real-Time Scheduler, also known as the SCHED_FIFO or SCHED_RR scheduler, is designed for real-time systems where timing behavior is crucial. It provides deterministic

scheduling and guarantees that higher-priority tasks will always preempt lower-priority tasks. These schedulers follow a priority-based approach, where tasks are assigned static priorities ranging from 1 (lowest) to 99 (highest). Processes scheduled under one of the real-time policies (SCHED_FIFO or SCHED_RR) have higher priority than processes scheduled under the default CFS scheduler, even the kernel threads.

The real-time priority threads run until they explicitly yield the CPU, either by sleeping or performing I/O operations, or are preempted by a higher priority thread. A thread that is scheduled by SCHED_RR can also be preempted by another equal-priority thread. These characteristics can easily cause starvation for non-real-time threads, and even the kernel. This might sound desirable from the perspective of the real-time system; however, the kernel is also responsible for handling essential tasks and maintaining the overall stability of the system. Therefore, Linux implements a real-time throttling mechanism to safeguard against starvation [14]. As a default, every 1 second, real-time threads are allowed to execute for at most 950 milliseconds. If they exceed this threshold, real-time threads are throttled and yield the CPUs to CFS regular threads.

Chapter 3

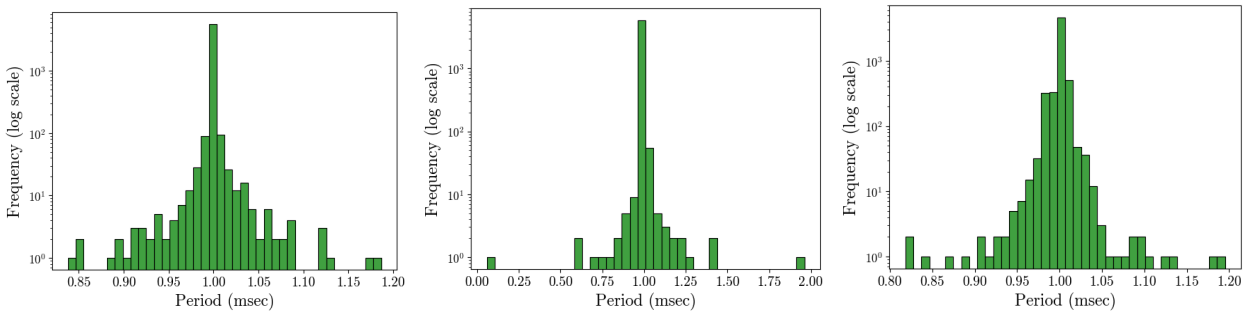
Micro-Benchmarks and Optimizations

In this chapter, we'll be using an LF program with a single timer-triggered reaction with an empty body. There is no additional stress introduced to the CPU, and efforts were made to minimize the impact of other processes during this benchmark, such as limiting the number of external processes, not using graphical interfaces, etc.

3.1 Measurement Strategy

Gathering timing behavior measurements with a logic analyzer is a frequently employed method in embedded systems. This method involves connecting the logic analyzer to the system under test and capturing timing signals through toggled GPIO pins. The Lingua Franca runtime has a built-in tracing mechanism by adding certain trace points throughout its codebase. These tracepoints allow developers to gather timing behavior measurements during the execution of Lingua Franca programs, providing valuable insights into the real-time capabilities of the program. These measurements provide insights into the logical and physical timestamps, microsteps, the source of the trigger, the reactor that was triggered, and a human-readable explanation of the event. One can also define their tracepoints within the Lingua Franca runtime to capture specific timing events of interest.

The LF tracer provides a flexible and customizable way to gather timing behavior measurements in Lingua Franca programs, allowing developers to gain a deeper understanding of the real-time capabilities of their systems compared to more traditional methods like logic analyzers. However, prior to utilizing the LF tracer as the primary measurement approach, it is necessary to demonstrate its precision and dependability in capturing timing behavior measurements. Figure 3.1 displays the period jitter histogram after executing an LF program with a single timer-triggered reaction containing only minimal code (any necessary GPIO toggling for the logic analyzer setup), described above, using SCHED_FIFO as the underlying Linux scheduler. The reason behind picking SCHED_FIFO will be explained in detail in Section 3.2. The timer is set to be triggered every 1 msec. Other than some outlier measurements, Figures 3.1a and 3.1b demonstrate that the tracer could be an alternative to



(a) Logic Analyzer (b) Default LF Tracer (c) Optimized LF Tracer

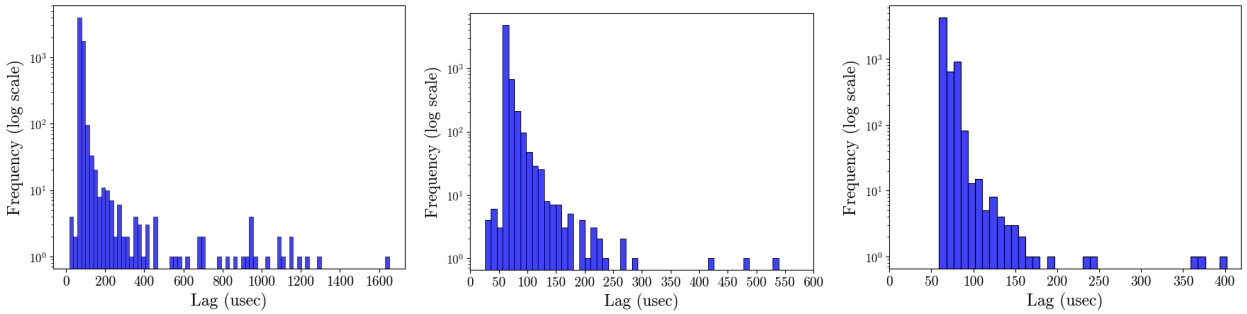
Figure 3.1: Logic analyzer measurements compared with LF Tracer

a logic analyzer. Upon investigating the reason behind the outlier points, we have concluded that the cause is the necessity to flush the buffer to memory or disk when the allocated TRACER_BUFFER gets filled. Many different optimization techniques can be employed here, but considering the user-friendliness of the generated dump, we chose to provide a configuration for the subset of the tracepoints the users will see in the generated file. In Figure 3.1c, one can see that when collecting only one tracepoint entry for the start of the reaction, such that the total number of tracepoints is less than the TRACER_BUFFER, the accuracy of the tracer is comparable to the logic analyzer. Therefore for the rest of the experiments, we will be using the optimized LF tracer as the primary measurement approach due to its flexibility and comparative accuracy in capturing timing behavior measurements.

3.2 Thread Policy and Core Isolation

One approach to dealing with the timing behavior in embedded systems is prioritizing the threads involved in time-sensitive tasks. By assigning higher priority to these threads, the operating system ensures that they are scheduled and executed promptly, minimizing any potential timing issues or OS-induced delays. This approach can help improve the accuracy and predictability of timing behavior, as higher-priority threads are given preferential treatment regarding CPU allocation and scheduling.

Figures 3.2a and 3.3a show the lag histogram of reaction start time and period jitter graphs respectively from the execution of the aforementioned LF program with a timer period of 1 msec on top of a “vanilla” Linux kernel, i.e. one that uses CFS as the scheduler with default thread policy setting, on a Raspberry Pi 4 Model B. Lag is defined as the time difference between the physical time and logical time, representing how delayed the physical time is compared to logical time. Figure 3.2a illustrates an average lag of 76 microseconds (usec), and 93.92% of the lags fall within the range of 58 to 89 usec. Figure 3.3a demonstrates a considerable jitter, with values being as far as 1.5 msec away from the period in the worst case since CFS is not optimized for real-time scheduling. However, the standard deviation



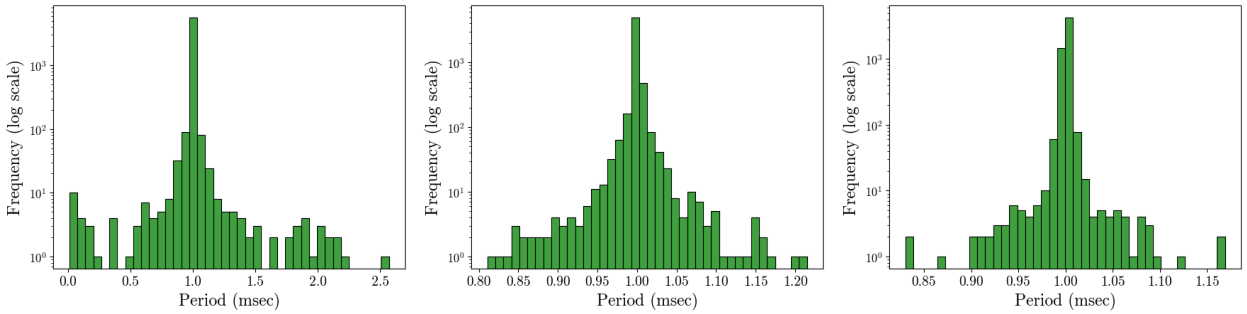
(a) Default thread policy (b) SCHED_FIFO (c) SCHED_FIFO and on isolated CPU

Figure 3.2: Lag histogram graphs from running the micro-benchmark for thread priority and isolation

of period jitter is still quite low such that 95% of the periods fall within 30 microseconds of 1 millisecond period.

Figures 3.2b and 3.3b show considerable improvement compared to Figures 3.2a and 3.3a respectively by scheduling under the real-time priority-based Linux scheduler `SCHED_FIFO` with priority 99. As described in section 2.2, `SCHED_FIFO` assigns respective real-time threads a higher priority than kernel threads up to a certain threshold. The worst-case lag drops to 538 usec from 1650 usec. Notably, a lag of 1650 causes the next trigger of the timer to be at least 650 usec late; with SCHED_FIFO, the subsequent reaction will never be delayed due to the lag introduced by the previous reaction. Moreover, 75% of the lags fall within the range of 58 to 65 usecs compared to only around 50% in the default thread policy setting. With SCHED_FIFO, 90% of the lags are less than 75 usec, and 95% of the lags are less than 84 usec. The worst-case period jitter also drops from 1.5 msec to 0.2 msec. With SCHED_FIFO, 97% of the periods fall within 30 microseconds and 94% of the periods are within 15 microseconds of 1 millisecond period.

Another optimization one can employ in the spirit of reducing OS interference is to isolate the core that the real-time code is executing on. This way all possible other threads are moved to other cores of the system. Figures 3.2c and 3.3c show additional improvement by also pinning the task onto a specific core and removing other tasks from that core by setting the `isolcpus` boot parameter. The worst case lag drops to 400 usec; with SCHED_FIFO alone, the number of lags exceeding 150 usecs is three times higher compared to the example with the isolated CPU pinning optimization. Here, an intriguing point to note is that there are no lags in Figure 3.2c that are less than 59 microseconds, whereas there are 14 instances where the lag falls below 59 when the `isolcpus` feature is not activated. We speculate the reason behind this is when the core is not shared with other threads (potentially kernel threads) and when there is no utilization of the period, it is more likely for the core to go idle or turn on power saving mode. Pinning the LF thread onto an isolated CPU while using SCHED_FIFO



(a) Default thread policy (b) SCHED_FIFO (c) SCHED_FIFO and on isolated CPU

Figure 3.3: Period jitter graphs from running the micro-benchmark for thread priority and isolation

shows a more significant improvement in period jitter with 98% of all periods falling within 10 microseconds of a 1 millisecond period, compared to running SCHED_FIFO alone, as demonstrated in Figure 3.3c.

OS interference is a significant factor that can affect the timing behavior of embedded systems. Any real-time system developer should employ techniques to mitigate this interference as much as possible. We argue that it's impossible to measure the timing impacts of the other parts of the systems with such significant OS impact. Therefore, the rest of the experiments will run using `SCHED_FIFO` with the worker thread pinned to an isolated core, unless otherwise specified.

3.3 Lag Correction

Even when the threads are assigned higher priority or pinned to isolated cores, there is an average lag of 68 usecs in the execution of time-sensitive tasks as demonstrated in Figure 3.2c. In order to address this issue, we have implemented an integral controller (I-controller) within the LF runtime to dynamically adjust the parameters of the control algorithm based on the measured lag. An I-controller is a component of a PID controller. A PID controller is a control loop feedback mechanism widely used in industrial control systems to manage processes and achieve desired setpoints. The way the output of a PID controller, which is equal to the control input to the system, is calculated is shown in Equation 3.1. The I-controller eliminates any error between the expected and measured outputs by continuously integrating the error signal over time and using the accumulated error to adjust the control input without using the proportional and derivative components.

$$output = K_p e(t) + K_i \int e(t)dt + K_d \frac{de}{dt} \tag{3.1}$$

This task is challenging because the system requirements diverge from those typically encountered in traditional PID controllers. In a traditional PID system, the control output is directly linked to the measured process value. For instance, the current temperature of the room can be a process value for a heating control system while the fuel you apply to the boiler can be the control output, and there is a bidirectional direct relationship between the error value (the difference between the desired and actual temperature) and the control output. However, in the case of controlling lag in a real-time system, there is no bidirectional relationship between the measured lag and the control output. One can base their control output on the measured lag, but there is no guarantee the measured lag next time will change based on the control output due to the inherent randomness of the OS or platform-induced lag.

The function we are interested in within the Lingua Franca runtime for the I-controller is `wait_until`. On a high level, the function takes in a physical time to wait until, `requested_time`, and calls the underlying sleep function to pause execution until that requested time is reached if the current time is less than the requested time. In the default LF runtime, no reaction executes before physical time passes its logical time since this behavior is useful for many real-time applications. Therefore this function should not return earlier than this requested time since that would cause the logical time to be behind the physical time.

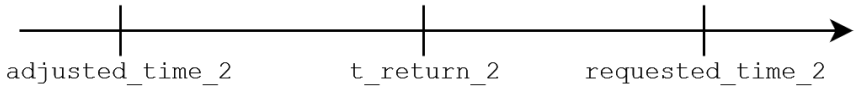
A traditional i-controller uses the integral component to continuously sum the error (the measured lag) over time and adjust the control output accordingly. This adjustment aims to minimize the overall lag and bring the system closer to the desired state. For this specific case, the critical question is the definition of the lag in the system. Let's start by defining some useful parameters:

- `requested_time` or t_{req}: the argument to the `wait_until` function, the actual desired time to wait until
- `control_value`: the integral total used to adjust the `requested_time` such that the lag between the physical time and the logical time is minimal when the function returns.
- `adjusted_time` or t_{adj}: `requested_time` - `control_value`; this is the value provided to the underlying sleep function by the `wait_until` function
- `t_return` or t': is the physical time measured when the underlying sleep function returns

At this point, one can propose two viable implementations for the lag controller: (1) using a running average of the measured lag between `t_return` and `adjusted_time`, or (2) using a running sum of the measured lag between `t_return` and `requested_time`. These two strategies for the lag controller can be evaluated and compared in terms of their effectiveness in minimizing the overall lag and bringing the system closer to the desired state. Figure 3.4 demonstrates two possible scenarios of timing behavior. In the first scenario shown in Figure 3.4a, the sleep function returns sometime after the requested time, resulting in a positive

lag. In the second scenario, shown in Figure 3.4b, the sleep function returns sometime before the requested time, causing the `wait_until` function to busy wait before returning. Although the scenario where sleep returns before the adjusted time is possible theoretically, the underlying sleep function never returns before the requested time. Therefore, we will not be considering that scenario.

(a) Scenario #1: sleep returns after requested time



(b) Scenario #2: sleep returns before requested time

Figure 3.4: Timing behavior scenarios with lag controller

In the scenario described in Figure 3.4a, both implementation options #1 and #2 would increase the `control_value` since the `control_value` was not large enough such that the requested time got overshot. Similarly, in the scenario described in Figure 3.4a, both implementation options #1 and #2 would decrease the `control_value` since the `requested_time` was too far in the future and the sleep function returned early. This is desired to reduce the amount of busy wait time and bring the system closer to the desired state.

We also want our control value to be "forgetting", i.e. the older measurements have less than or equal effect compared to recent measurements, but not more, since we do not want the control algorithm to be driven by the oldest measurements. Let's consider that during execution, the situation shown in Figure 3.4a is followed by the situation shown in Figure 3.4b. The equations below illustrate the adjustments in the `control_value` resulting from the application of the running average algorithm.

At timestep 0: $\texttt{control\_value} = 0$

At timestep 1:
$$\begin{aligned}\texttt{control\_value} &= \frac{0 + t'_1 - t_{adj_1}}{1} \\ &= t'_1 - t_{req_1} - \texttt{control\_value} = t'_1 - t_{req_1}\end{aligned}$$

At timestep 2:
$$\begin{aligned}\texttt{control\_value} &= \frac{t'_1 - t_{req_1} + t'_2 - t_{adj_2}}{2} = \frac{t'_1 - t_{req_1} + t'_2 - (t_{req_2} - \texttt{control\_value})}{2} \\ &= \frac{t'_1 - t_{req_1} + t'_2 - (t_{req_2} - t'_1 + t_{req_1})}{2} = \frac{2t'_1 - 2t_{req_1} + t'_2 - t_{req_2}}{2} \\ &= t'_1 - t_{req_1} + \frac{t'_2 - t_{req_2}}{2}\end{aligned}$$

The equation above illustrates that the running average algorithm determines the new control value by assigning greater significance to older measurements compared to newer ones. On the other hand, the running sum algorithm calculates the control value by simply adding up all the measured lags over time; therefore, all measurements have the same effect on the outcome as shown in the equations below. As such, the running sum algorithm was chosen.

At timestep 0: $\texttt{lag\_control} = 0$
At timestep 1: $\texttt{lag\_control} = t'_1 - t_{req_1}$
At timestep 2: $\texttt{lag\_control} = t'_1 - t_{req_1} + t'_2 - t_{req_2}$

Algorithm 1 shows the implementation of the i-controller in pseudocode. The control value is initialized to 0 as a static variable at the beginning of the function. If the current time has already passed the requested time, the function returns. Before adjusting, we constrain the control value to a minimum of zero because running sum implementation lags can be positive or negative. This is desirable because if there is a negative control value, the adjusted time given to the sleep function would be later than the requested time, introducing an inherent lag to the function. One can see the adjustment based on running sum implementation in lines 14 and 15 of the pseudocode shown in algorithm 1.

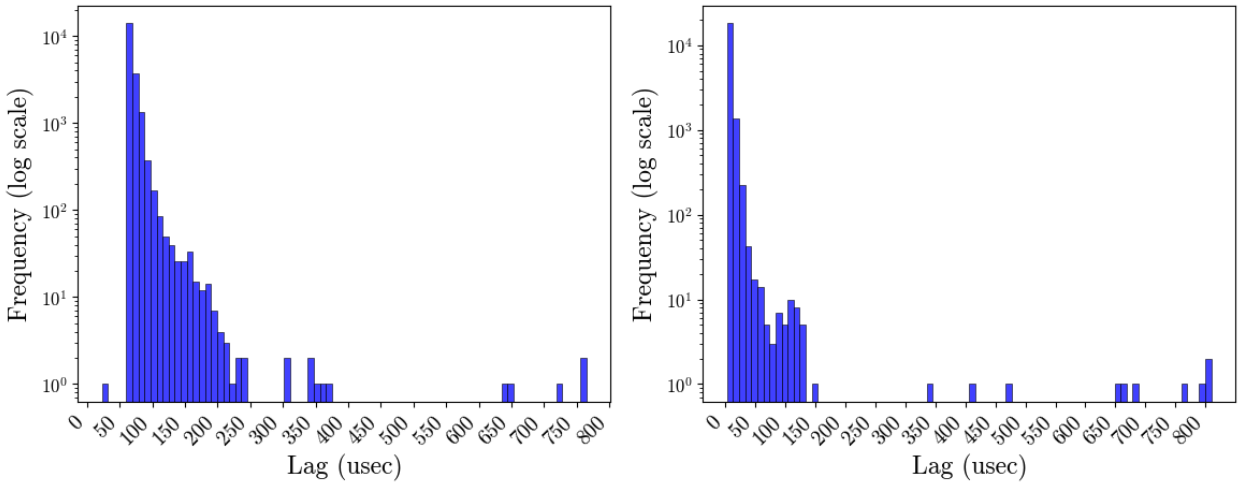
Algorithm 1 I-controller algorithm

```
1:  #define K_i                                   ▷ The integral gain or the multiplication factor
2:  procedure WAIT_UNTIL(requested_time)
3:      static control_value ← 0

4:      control_value = MAX(control_value, 0)
5:      adjusted_time ← requested_time − control_value

6:      if now > requested_time then
7:          return
8:      else if now > adjusted_time and now < requested_time then
9:          //Equal to checking: wait_duration < control_value
10:         control_value ← control_value − K_i × (now − adjusted_time)
11:         //busy wait and return
12:
13:     //call underlying sleep function which returns at some physical time t_return
14:     lag ← t_return − requested_time
15:     control_value ← control_value + K_i × lag

16:     if now < requested_time then
17:         //busy wait and return
```

Notably, line 10 also changes the control value. If the else if condition is true, such that if `wait_duration` < `control_value`, it does not make sense to invoke the underlying sleep function because this would result in a lag according to the control value. Thus, if this condition holds, busy waiting is preferred. To understand why it is necessary to modify the control value, think about a situation in which the control value becomes significantly high, such as due to an outlier lag. Without any updates for the control value, it's possible to keep busy waiting for the rest of the program without utilizing the underlying sleep schedule since the else-if condition will always hold. Constantly busy waiting is undesirable for several reasons, such as inefficient use of CPU cycles that could be utilized for other tasks, reduced energy efficiency, and the potential for the operating system to seize control later if the program does not yield by invoking the sleep function. This can result in greater delays in the program. To maintain the desired behavior of the program and prevent it from continuously busy waiting, it is necessary to reduce the control value by the amount it overshot the current physical time. The impacts of this decision will be further analyzed in section 4.1.

Ki Gain	Percentage of lags below 10 usec
2	47.56
1.5	75.34
1	82.58
0.5	76.12

Table 3.1: The effectiveness of different i-gain values



(a) SCHED_FIFO and isolated CPU

(b) With lag controller

Figure 3.5: Lag histogram graphs from running the micro-benchmark for lag controller

The key part of any PID controller is to tune the gains to achieve optimal control performance. Table 3.1 shows the percentage of lags that are less than 10 usec when using different gain values for the PID controller. The expected success result is shifting as many data points to 0 lag as possible. Although all of these gains shift the histogram towards zero lag, table 3.1 highlights that the gain value of 1 achieves the best performance. Figure 3.5 shows the resulting histograms of running the microbenchmark with and without the lag controller. Due to its significant impact, unless otherwise specified the rest of the experiments will run using the I-controller with the gain value of 1.

Chapter 4

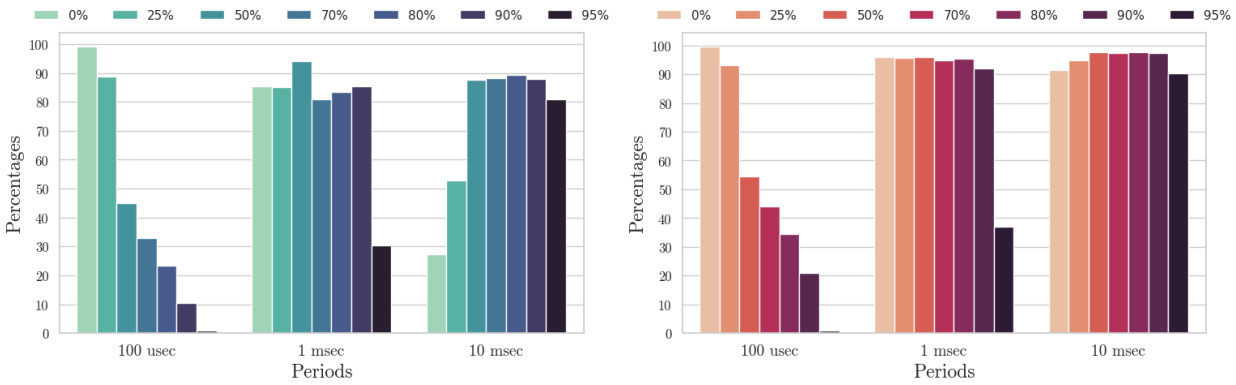
Case Studies

In this chapter, we discuss two case studies that simulate the behavior of different real-time systems in a simplified manner. These case studies will help us isolate the individual influence of independent variables on the system's timing behavior. Section 4.1 introduces the timer utilization case study, which explores how timing behavior correlates with the execution time of a task. Section 4.2 introduces the periodic tasks case study, which investigates the impact of parallelism on periodic tasks with varying periods, offsets, and deadlines.

4.1 Timer Utilization

To understand the timing behavior of a system, it is essential to consider the execution times of the tasks. In order to properly investigate execution times independently from other factors, we will use utilization, which is defined as the percentage of time that a task requires compared to the total available processing time, in this context. Utilization is a key metric in assessing the timing behavior of a system, as it represents the efficiency with which tasks are executed relative to the period. In essence, this case study tries to force the limits of the utilization and evaluate the resulting timing behavior so that developers can find suitable utilization rates based on their acceptable timing behavior and tolerance towards unpredictability.

We picked 100 usec, 1 msec, and 10 msec as the period values for this case study, representing varying time constraints. Periods of 1 msec and 10 msec are commonly used in various CPS applications as polling time while 100 usec represents a relatively tight period considering the average lag on Raspberry Pi 4 was 68 usecs with real-time scheduling and isolated CPUs. We ran each of these periods with utilization values of 0, 25, 50, 70, 80, 90, and 95%. Figures 4.1a and 4.1b show the percentage of lags that are less than 10 usec and 20 usec respectively for each period and utilization combination after running the initial experiment with the lag controller explained in the previous chapter. The measurements confirm the correctness of the lag controller as evidenced by the percentages of lags that are under 20 usec compared to the baseline case without a lag controller.



(a) The percentage of $<$ **10 usec** lags

(b) The percentage of $<$ **20 usec** lags

Figure 4.1: Bar graphs for the percentages of lags below a threshold for each period and utilization

There are four key notable points from the figures:

- The smallest lag that can be observed with a 95% utilization for a 100 usec period is 88.52 usecs. Figure 4.2 demonstrates how the lag value scales constantly within an example run. This behavior is different from other period values for the same utilization because of the tighter time constraint. Even in an ideal system, the wait time between periods is just 5 usecs; considering other overheads introduced by Lingua Franca, OS scheduling, or kernel-induced latencies, it is not surprising to observe higher lags since the system is constantly lagging behind.

- Figure 4.3 demonstrates that in the scenario with 95% utilization for 1 millisecond, there is a notable spike around the timestamp of 1.25 seconds, resulting in a delay of up to 50 milliseconds. We speculate that this lag is caused by real-time throttling As discussed in section 2.2.3, Linux's real-time throttling is a safety mechanism that allows real-time threads to execute for at most 950 milliseconds in every timeframe of 1 second. Due to high utilization, the kernel lacks sufficient runtime during the initial part of the run; therefore, it interrupts the real-time execution, i.e. the LF threads, to run the essential kernel tasks to prevent their starvation. Notably, we do not see this behavior for other period values. For the 100-microsecond period, in order to generate an equivalent number of tracepoints, each run only executes for 200 microseconds, while the 1-millisecond period runs for 2 seconds. As a result, with the 100-microsecond period, we do not reach the limit point. For the 10-millisecond period, the 95% utilization still needs to wait for 500 microseconds in the ideal scenario, which does not stress the kernel as much.

- The percentage of lags that are less than 10 microseconds are 27% and 53% for the 10 millisecond period with 0% and 25% utilization respectively. This is surprising because

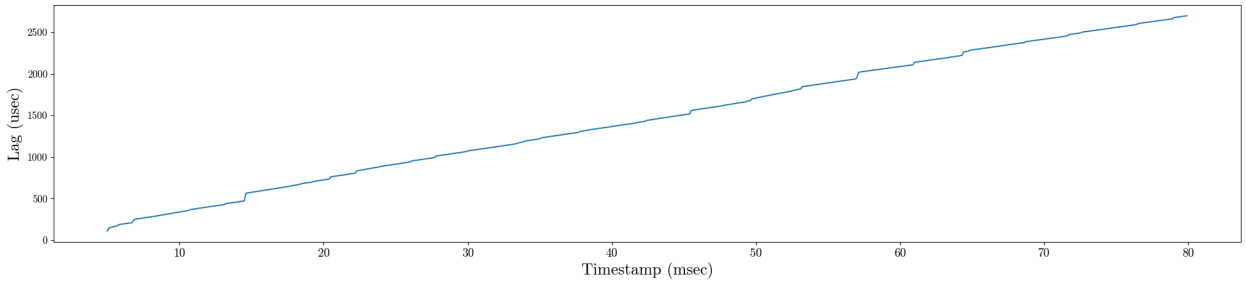


Figure 4.2: Lag vs. Time graph for 100 usec period 95% utilization

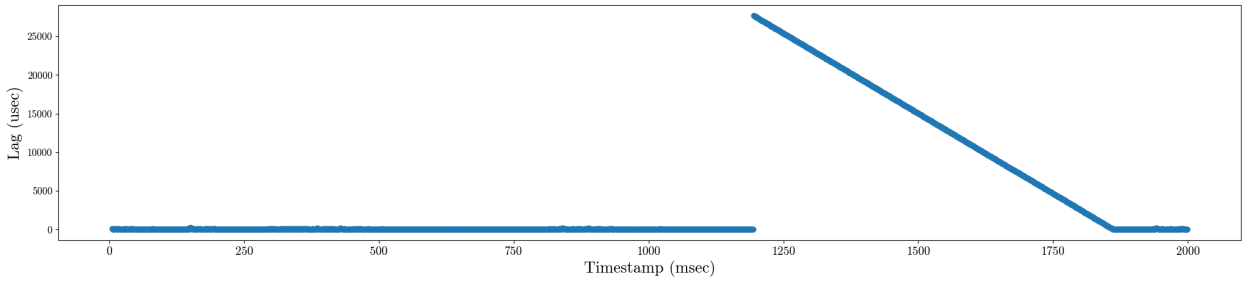
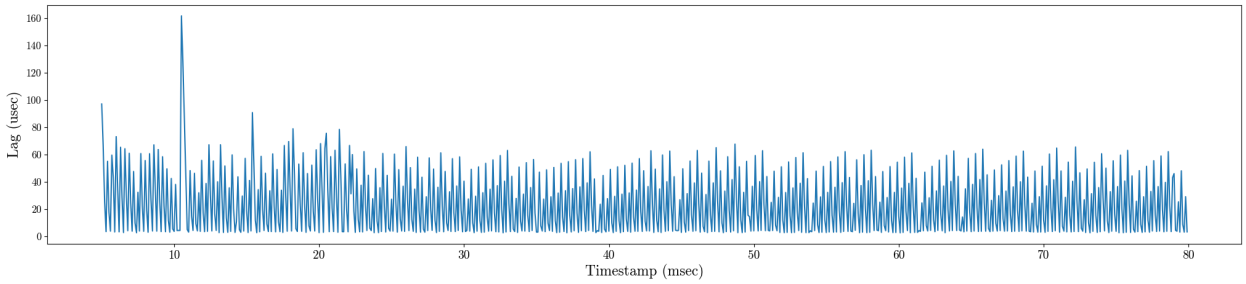


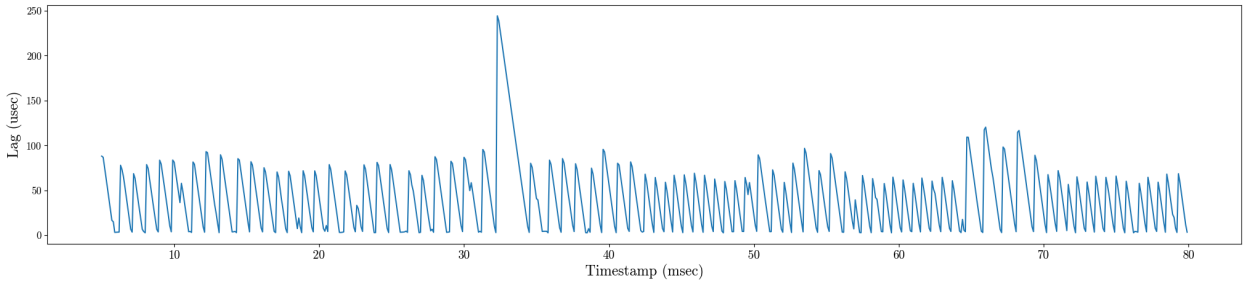
Figure 4.3: Lag vs. Time graph for 1 msec period 95% utilization

it's expected to see lower lags when the CPU is less stressed. It's important to highlight that for both such scenarios, the percentages of lags that are less than 20 microseconds are 92% and 95% respectively. This confirms that the lag controller is still effective, but shifted. This shift can be attributed to the variability of the OS-induced lag based on sleep time. Testing this assumption involved calculating the standard deviation of the delays without the lag controller at 0% utilization for periods of 100 microseconds, 1 millisecond, and 10 milliseconds, yielding values of 2.53, 8.48, and 13.37 respectively. This implies that as the sleeping duration increases, the variance of the lags increases, making it harder for the lag controller to correct the control value, thus causing lags to be shifted. The respective percentages of lags that are less than 10 microseconds, 99%, 85.35%, and 27.15%, confirm this correlation.

- Although it is acceptable to see a decrease in the percentage of lags below a certain threshold as the utilization increases, the reduction in percentage is quite substantial over a 100-microsecond period, even with relatively low levels of utilization, such as 50%. Figures 4.4a and 4.4b show non-converging oscillation in the lag vs time graphs for utilization values of 50% and 80%, with a period of 100 microseconds over 80 milliseconds. This behavior indicates a misbehavior in the system as it fails to reach a stable state.



(a) 100 usec period 50% utilization



(b) 100 usec period 80% utilization

Figure 4.4: Lag vs Time graphs showing oscillating behavior

This oscillation is caused by the control value adjustment in the else if condition of the lag controller, as shown in line 10 of algorithm 1. Since the control value is zero, when the wait_until function is initially called, the function calls the underlying sleep function. Upon waking up, the system measures the lag and updates the control value. In most cases, this value hovers around an average lag of 68 microseconds. When the function is called for a second time, it requests a wait duration of 20 microseconds in the ideal scenario for a period of 100 microseconds and an 80% utilization rate. As this duration is not long enough according to the current control value, it triggers the execution of the else-if case in the lag controller algorithm. Consequently, the control value is reduced to match the requested wait duration. This adjustment leads to calling the underlying sleep function with an incorrect control value during the next iteration, resulting in the increased lag. Therefore, this scenario results in a continuous adjustment of the control value and causes oscillation in the lag measurements. Due to the high utilization, which leads to wait duration being less than the average control value, the desired behavior is for it to consistently fall into the else-if scenario without adjusting the control value.

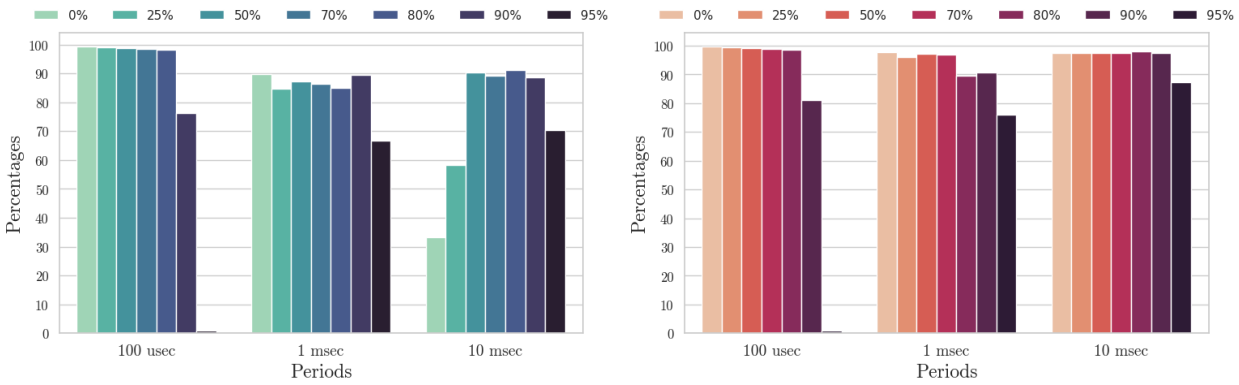
In order to prevent this oscillating behavior, it is important to highlight the two orthogonal goals of the system. The lag controller aims to minimize the amount of busy waiting in the system, especially the busy waiting caused by outlier lag measurements while ensuring that the system busy waits if the wait duration is less than the control value. In the new version of the lag controller, we will isolate these two goals from each other by not incorpo-

rating the outlier lags into the control value calculation, to begin with, therefore ensuring that the value used in the else-if case is safe to use without any adjustments to the control value. The question here is to identify what is classified as an outlier lag since this value is highly correlated with system and platform-level assumptions.

Algorithm 2 Updated I-controller algorithm

```
1:  #define K_i                                  ▷ The integral gain or the multiplication factor
2:  #define min_period                           ▷ The minimum timer period value for the application
3:  procedure WAIT_UNTIL(requested_time)
4:      static control_value ← 0

5:      control_value = MAX(control_value, 0)
6:      adjusted_time ← requested_time − control_value

7:      if now > requested_time then
8:          return
9:      else if now > adjusted_time and now < requested_time then
10:         //Equal to checking: wait_duration < control_value
11:         control_value ← control_value − K_i × (now − adjusted_time)    ▷ Deleted code
12:         //busy wait and return
13:
14:     //call underlying sleep function which returns at some physical time t_return
15:     lag ← t_return − requested_time
16:     if lag < min_period then
17:         control_value ← control_value + K_i × lag
18:
19:     if now < requested_time then
20:         //busy wait and return
```

Let us assume a system with a 1 millisecond timer period; this means the lowest frequency that this system will invoke the wait_until function is every 1 millisecond, assuming 0% utilization and overhead. If a measured lag value is larger than 1 millisecond, the control value will go over 1 millisecond, causing any future call to the wait_until function to busy wait regardless of the wait duration. Therefore any lag measurement exceeding 1 millisecond, the minimum period value for the system, can be identified as an outlier lag and should not incorporated into the control value. This solution alone addresses the oscillation problem highlighted above for timer period values that are larger than the underlying average lag of the system. A future optimization here could be defining this threshold based on the system and platform characteristics statically or dynamically. However, in this paper, considering the tradeoff between implementation complexity and the respective gain, we decided to go



(a) The percentage of $<$ **10 usec** lags

(b) The percentage of $<$ **20 usec** lags

Figure 4.5: Bar graphs of the percentages of lags below a threshold for each period and utilization with the updated lag controller

with the simpler solution. We posit that for a time-critical application to consider a lag controller, it must operate on a platform with an average delay smaller than the minimum period required by the application. Algorithm 2 shows the pseudocode for the updated I-controller algorithm. The additions compared to algorithm 1 are highlighted with yellow, and the deletions are highlighted with red.

Figures 4.5a and 4.5b show the percentage of lags that are less than 10 usec and 20 usec respectively for each period and utilization combination after running the experiment with the updated lag controller. Figure 4.5a shows that there is no significant percentage decrease in lags of less than 10 microseconds as the utilization increases when the period value is 100 microseconds. Although, notably, with 95% utilization of the 100-microsecond period, the percentage of lags that are less than 20 microseconds is still 0 due to consistent scaling delay that causes the logical time to never catch up with physical time. The aforementioned key points described for the original lag controller, aside from oscillations, also hold for the updated lag controller. It's also worth highlighting that the following period-utilization pairs can lead to nondeterministic real-time throttling:

- 1 msec period - 80% utilization
- 1 msec period - 90% utilization
- 10 msec period - 95% utilization

4.2 Periodic Tasks

Periodic tasks are one of the common design patterns in real-time systems. This case study on periodic tasks explores the impact of parallelism on tasks with varying periods, offsets, and deadlines. By analyzing these various scenarios and their schedulability with different scheduling policies, we also aim to explore how different layers of scheduling mechanisms, in either layer, change the timing behavior of the system.

As described in section 2.1, LF deadlines are defined based on the start of the reaction and not when it gets completed. However, an application might be interested in setting a deadline for the end of a reaction. We can have a workaround by defining a downstream reaction that is triggered by the completion of the task and has its own deadline, as shown in Figure 4.6. The deadlines in LF are inherited; therefore, the upstream reaction can still be aware of the downstream reaction's deadline when making a scheduling decision. This way, the reaction start deadline of the downstream reaction can serve as the reaction end deadline for the original reaction. The deadlines referred to in this section are defined based on the end of the reaction using this workaround unless otherwise specified.

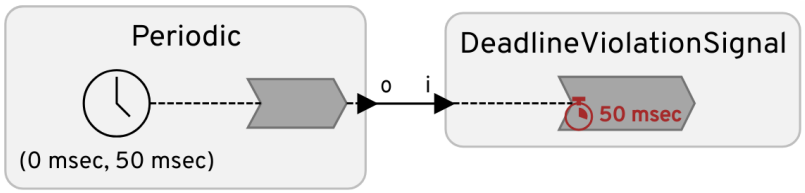
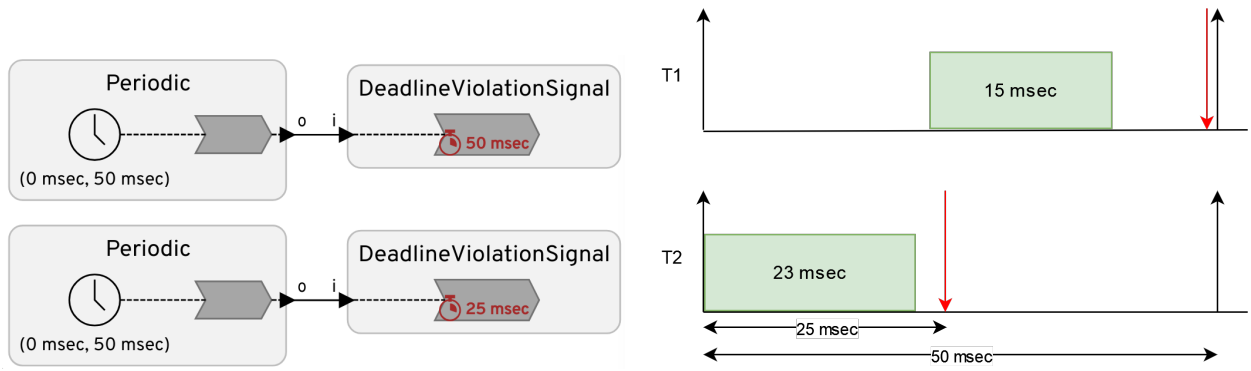


Figure 4.6: Implementing deadlines with downstream reactions

In the first scenario, whose diagram is demonstrated by Figure 4.7a, we consider two periodic tasks with periods of 50 msec. The first task has a deadline of 50 msec, representing the inherent deadline based on the period; the second task has a deadline of 25 msec, representing a relatively tight time constraint. The first task has an execution time of 15 msec, and the second task has an execution time of 23 msec. The only way to schedule these tasks without violating any deadlines is to execute the second task to completion before starting to execute the second task. Figure 4.7b demonstrates the ideal scheduling of these tasks to avoid deadline violations.

When these tasks are executed within the same enclave with the GEDF scheduler, the expected behavior is GEDF prioritizing the reactions based on the inherited deadlines, therefore executing the second task before the first one. However, in the current LF runtime, the level-based scheduling, described in section 2.1, is prioritized over deadlines. As a result, the scheduler executes all reactions in the first level, i.e. the `Periodic` reactors, before moving to the second level, i.e. the `DeadlineViolationSignal` reactors. This causes a deadline violation in the downstream reaction even when the upstream reaction completes on time because a lower priority reaction blocks the higher priority task due to its lower level. This indicates the current GEDF scheduler cannot prioritize inherited deadlines with concurrent



(a) The LF diagram representation

(b) The ideal scheduling

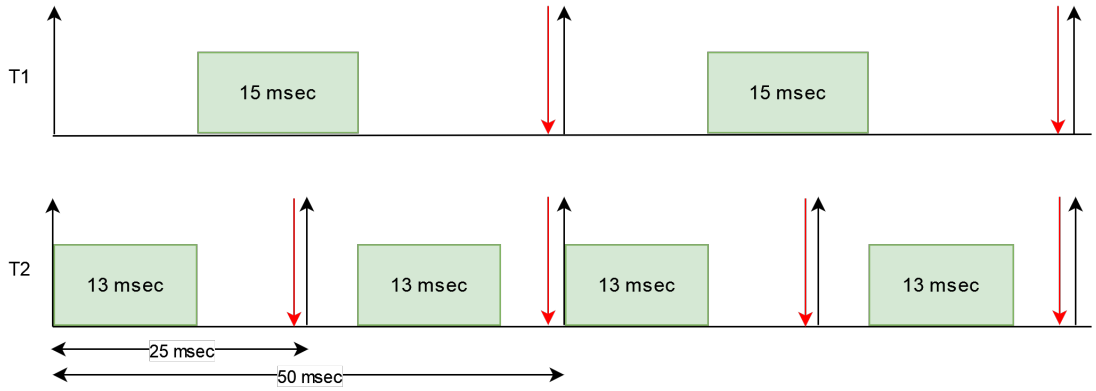
Figure 4.7: Periodic Tasks Scenario #1

reactions. This limitation triggered a discussion for a future change to extend the use cases of the GEDF scheduler by prioritizing deadlines over levels; the implementation of such a change is left as future work. Notably, if the deadlines were defined within the `Periodic` reactor directly, i.e. the `DeadlineViolationSignal` reactor did not exist, the GEDF scheduler would have prioritized the second task; therefore it would have been executed to completion before the first task, and both tasks would have met their respective start-of-reaction deadlines.

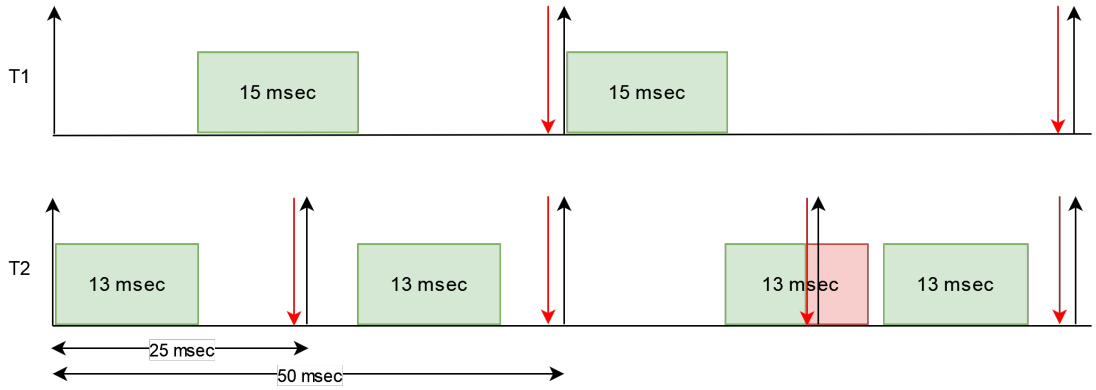
In the current LF runtime, in order to employ the end-of-reaction deadline strategy and make use of inherent deadlines in reaction chains, these scenarios need to be run in separate enclaves. In our experimentation, we executed these two reactions with only one worker thread per enclave and on only one core in total, so that we can limit the underlying resources and force interference. We first run the experiment with the GEDF scheduler, which is the default LF scheduler when the program has deadlines. Depending on the order in which the enclaves are defined in the main reactor, the reaction that starts executing changes, but in most cases, both of the reactions start executing before one of them completes. This behavior can be attributed to the "fairness" policy of CFS, the default OS-layer scheduling mechanism on Linux. Due to context switches, between reactions, and potentially with other kernel and user threads, CFS fails to meet most (if not all) deadlines. Additionally, it introduces non-determinism across runs on two key points: (1) The number of deadlines it violates across runs which is due to the number of context switches, length of interruptions by non-LF threads, etc., and (2) the point of context switches between reactions, i.e. at which point after the first reaction start executing, the program would context switch to the second thread. Based on this scenario, we can conclude CFS is not suitable for LF programs with tight deadlines.

A proposition for further examination would be to modify the underlying scheduler to SCHED_FIFO. Given that the GEDF scheduler only assigns priorities to tasks with earlier deadlines within the same enclave, this is not applicable in our case, causing the GEDF

scheduler to behave no differently than the NP scheduler. Therefore, even if we set the LF scheduler to be GEDF, the task that is defined earlier in the main reactor executes until completion before the other task gets triggered due to the FIFO nature of the scheduler. If this ordering is done correctly, such that the higher priority reaction is defined before the other, both tasks meet all of their deadlines. Notably, SCHED_FIFO provides two key features in this scenario: (1) it fulfills the requirement of the execution of the higher priority task until completion before the lower priority, therefore giving the developer the control to prioritize by task declaration, and (2) it provides a priority to both tasks over the kernel threads and other user processes, and reduces the outside interference. However, notably, it is *not* responsible for providing any prioritization between LF threads.



(a) The ideal scheduling



(b) Represented run with SCHED_FIFO

Figure 4.8: Periodic Tasks Scenario #2

In the second scenario, we consider two periodic tasks with differing periods, 50 msec and 25 msec respectively. Each task uses its period values as its deadlines. The tasks have execution times of 15 and 13 msecs respectively. Figure 4.8a shows the ideal schedule for these tasks that avoids deadline violations. Similar to scenario 1, the example is run on one

worker thread per enclave and pinned to one core. Figure 4.8b demonstrates a representation of a run of this scenario with SCHED_FIFO, highlighting that the third trigger of the second task violates its deadline. It is important to note that this diagram does not include any context switch time or kernel interruptions. For example, the second invocation of task 2 might not immediately start; however, we know that it meets its deadline, so the exact time it starts is not relevant to us at this time. Scenario 2 demonstrates that SCHED_FIFO is only effective when we want to decrease the interference from kernel threads, but the task order does not provide effective prioritization between the LF threads.

Let us consider why scenario 1 was schedulable with SCHED_FIFO while the second scenario is not. In scenario 1, SCHED_FIFO was able to schedule the tasks in the correct order if they were defined in the correct order since it simply executed them in the order they were defined until completion. This order was preserved across multiple periods since when the higher priority task completes its execution, it looks at its event queue for the next trigger in the future, and calls the wait_until function until that physical time. In a way, it schedules itself for the future, putting itself in the queue of the SCHED_FIFO before the lower priority task can. Therefore, each trigger respected the same task order, i.e. the priority. However, in scenario 2, the higher priority task has a lower period than the lower priority task. Therefore, when the first trigger of the second task finishes executing, it schedules the thread for the second trigger, i.e. puts itself immediately after the first trigger of the first task. Similarly, when the first task finishes executing, before yielding, it puts itself immediately after the second trigger of the second task. When the second trigger of the second task finishes executing, it schedules for the next trigger, which would be at a logical timestamp of 50 milliseconds. However, since the first task has already scheduled itself for its second trigger at the same logical time, and according to SCHED_FIFO, the priorities of these tasks are equal, at a logical time of 50 milliseconds, the first task is executed before the second one, causing a deadline violation.

This violation is fixable by introducing deadline monotonic priorities that are reflected in the underlying scheduler. Deadline monotonic (DM) scheduling is a static real-time scheduling algorithm that assigns priorities to tasks based on their deadlines [2]. The tasks with the shortest deadlines are given the highest priority, ensuring that tasks with shorter periods are always executed before tasks with longer periods.

Notably, if we get rid of the downstream reaction and directly define the deadline within the `Periodic` reactor, the GEDF scheduler would be able to meet the start-of-the-reaction deadlines in scenario 2 since the given execution times do not require preemption. However, if the first task had a longer execution time, like 25 msec, combined with the context switch and kernel interrupt times, we will see some deadline violations for the second task such that some subset of triggers for the second task could not get triggered within its respective period. This problem could be resolved by any preemptive scheduler based on deadlines, including the DM scheduler.

Figure 4.9 represents the ideal scheduling for scenario 3, which is the same as scenario 1 except the higher priority task gets triggered with a 1 millisecond offset. Due to this offset, the order in which tasks are defined cannot establish inherent priorities for SCHED_FIFO.

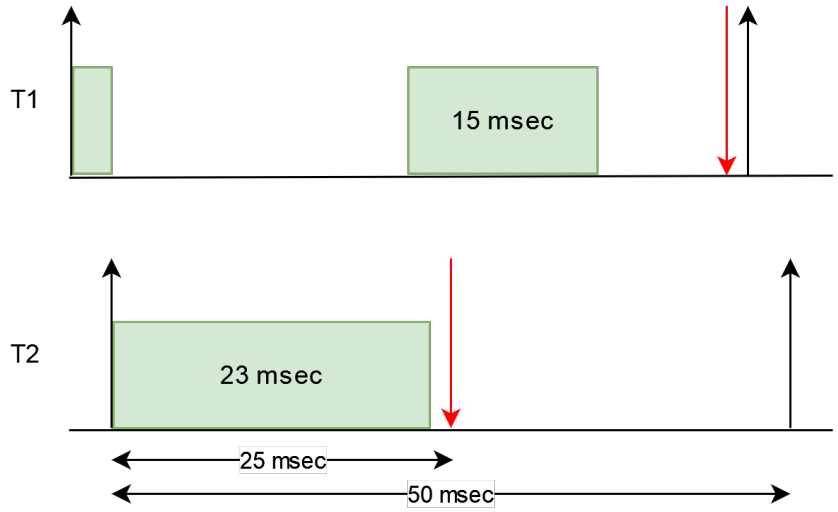
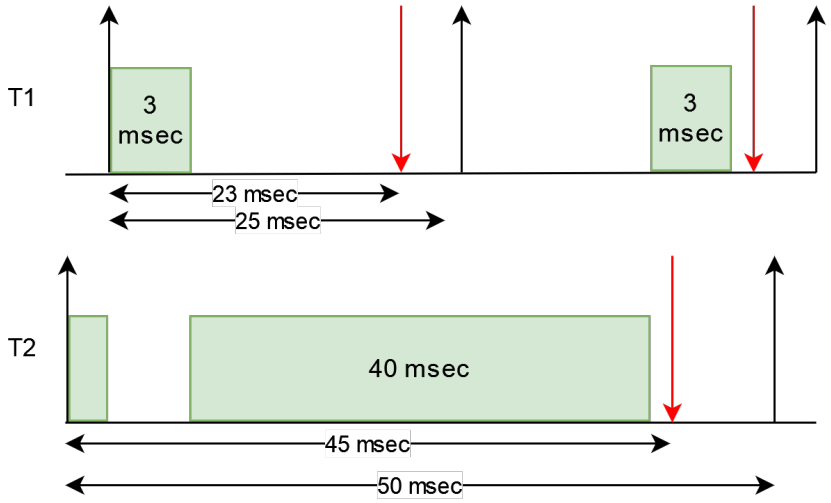


Figure 4.9: Periodic Tasks Scenario #3

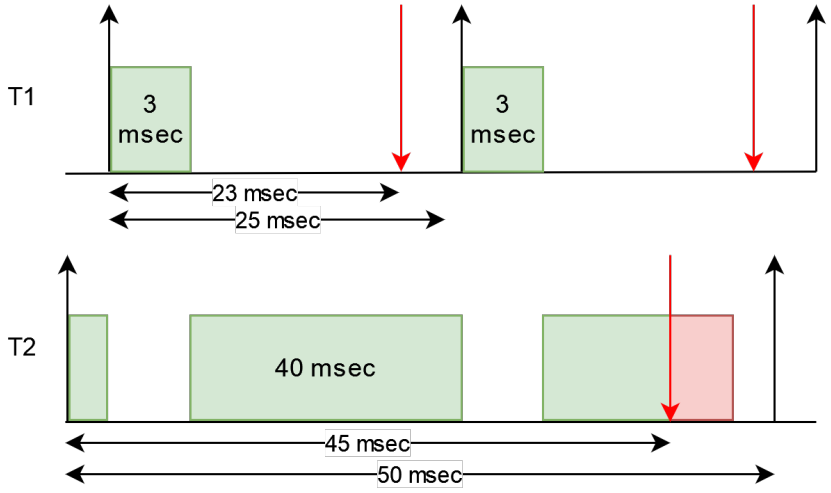
This leads to lower-priority tasks being triggered before higher-priority ones, resulting in deadline violations. If the offset value between the tasks drops below 200 microseconds, then the first trigger of the tasks can be triggered in the correct order such that the higher priority tasks get executed first, due to the internal delay of creating the other enclave environment in the Lingua Franca runtime environment. However, this doesn't extend to subsequent triggers. Similar to scenario 2, this scheduling problem can be resolved by a deadline monotonic scheduler, since when the second task with a shorter deadline gets triggered, it would get assigned a higher priority and preempt the lower priority task as intended. A scenario that merges scenarios 2 and 3 such that the higher priority task has a shorter period and has an offset, also can be scheduled by the deadline monotonic scheduler as long as the execution times are schedulable, i.e. the total execution time of the reactions can fit to the periods. Notably, scenario 3 is not schedulable with the GEDF scheduler even if we get rid of the downstream reaction, since the GEDF scheduler is limited to prioritizing reactions that have the same logical tag.

So far the higher priority thread has not changed dynamically during the execution of the program in any of the scenarios we have discussed. Figure 4.10a introduces the ideal schedule for such a scenario. In scenario 4, task 1 has a period of 25 milliseconds, a deadline of 23 milliseconds, and an execution time of 3 milliseconds while task 2 has a period of 50 milliseconds, a deadline of 45 milliseconds, and an execution time of 40 milliseconds. The first task has an offset of 1 millisecond. Figure 4.10b represents how these tasks would be scheduled with a deadline monotonic scheduler. Since the deadline monotonic scheduler assigns priorities statically based on the duration of the deadline, task 1 would always have a higher priority. However, due to dynamic execution times, the ideal schedule should keep executing task 2 until completion before switching to the second trigger of task 1. This ideal

behavior can be scheduled by the Earliest Deadline First (EDF) scheduler. EDF scheduling is another real-time scheduling algorithm that dynamically assigns priorities to tasks based on their remaining time until their deadline. This means that as tasks get closer to their deadlines, they are given higher priority. Since it's dynamic, when task 2 is closer to its deadline, it will be given higher priority than task 1 and will be executed until completion before switching to the second trigger of task 1.



(a) EDF scheduling



(b) Deadline Monotonic Scheduling

Figure 4.10: Periodic Tasks Scenario #4

The main difference between Deadline Monotonic and EDF schedulers lies in how they prioritize tasks. While DM scheduling prioritizes tasks based on their deadlines, EDF

scheduling prioritizes tasks based on their remaining time until their deadline. This difference can have implications for how tasks are scheduled and can impact the meeting of deadlines in real-time systems.

While DM scheduling is effective in ensuring that tasks with shorter deadlines are executed first, it may not be as flexible in adapting to changing task priorities during execution. EDF scheduling, with its dynamic prioritization based on remaining time until the deadline, can adapt to varying task execution times and prioritize tasks accordingly. DM scheduling's deterministic prioritization based on deadlines may be suitable for systems with fixed task priorities and predictable deadlines. In contrast, EDF scheduling's dynamic prioritization based on remaining time until the deadline may be more suitable for systems with varying task execution times and changing task priorities during execution. On the other hand, this dynamic scheduling approach makes EDF more costly and complex compared to DM. It usually requires additional resources and introduces greater intricacy into the system architecture. Overall, DM and EDF schedulers come with their tradeoffs, and it would be helpful to see both of them implemented within the Lingua Franca schedulers for developers to pick based on the needs of their applications.

Chapter 5

Future Work

Extending LF Schedulers: The periodic tasks case study revealed the need for re-evaluation of the LF schedulers and highlighted the limitations of the current GEDF scheduler. The efforts on implementing a DM and EDF scheduler could be extended to work with the underlying Linux real-time schedulers. Another goal could be improving these schedulers for non-Linux systems such as RTOS or bare-metal platforms. The future work will involve evaluating these schedulers under various time constraints, including soft and hard real-time as well as mixed-criticality systems.

Execution over different platforms: The hardware platform is directly linked to the timing behavior of the system. As a result, we propose carrying out this evaluation on the following platforms: (1) nRf 52 with Zephyr RTOS, (2) RP2040, and (3) the latest version of PRET machines, FlexPRET [16].

Incorporating more case studies: Future work could involve focusing on other variables that contribute to the timing behavior of Lingua Franca. We suggest the following case studies for further evaluation:

- Extending periodic tasks case study to work with federates or experimental feature hardware threads.
- Introducing startup reactions to periodic tasks case study with randomized execution times.
- Introducing concurrency to the timer utilization case study.
- Adding a control loop case study to simulate a timer-triggered sensor to an actuator model. This case study could include variations in polling and processing times, including dynamically changing processing times based on sensor input. This case study could also be extended to include deadlines for scheduling policies as well as concurrency with other tasks. Further work could focus on different execution distributions of the processor reactor, implementing either pipelining with multiple reactors or a scatter-gather.

Chapter 6

Conclusion

In this thesis, the focus was on investigating and analyzing the real-time capabilities of Lingua Franca, a deterministic reactor-based coordination language where time is a first-class citizen. We have shown the efficiency and usability of the LF tracer and introduced the design and implementation of a lag controller that reduces the vast majority of lags below 20 microseconds. We introduced two case studies: timer utilization and periodic tasks. The timer utilization study explored how timing behavior correlates with the execution time of a task and demonstrated that LF can reach high utilizations like 95% over a Linux operating system while keeping the majority of the lags under 20 microseconds using the lag controller. The periodic tasks study delved into different scheduling scenarios, considering the impact of task order, priority, and deadlines on meeting real-time constraints. This case study highlighted the impact of the current implementation practices and the need for additional efforts to implement deadline monotonic and earliest deadline first schedulers.

Bibliography

[1] Krste Asanovic et al. "The landscape of parallel computing research: A view from Berkeley". In: (2006).

[2] Neil C Audsley et al. "Hard real-time scheduling: The deadline-monotonic approach". In: *IFAC Proceedings Volumes* 24.2 (1991), pp. 127–132.

[3] Giorgio C Buttazzo. *Hard real-time computing systems: predictable scheduling algorithms and applications*. Vol. 24. Springer Science & Business Media, 2011.

[4] Edward A Lee. "Computing needs time". In: *Communications of the ACM* 52.5 (2009), pp. 70–79.

[5] *Lingua Franca Website and Documentation*. URL: `https://www.lf-lang.org`.

[6] *Linux Kernel CFS Scheduler*. URL: `https://docs.kernel.org/scheduler/sched-design-CFS.html`.

[7] Marten Lohstroh. "Reactors: A Deterministic Model of Concurrent Computation for Reactive Systems". PhD thesis. EECS Department, University of California, Berkeley, Dec. 2020. URL: `http://www2.eecs.berkeley.edu/Pubs/TechRpts/2020/EECS-2020-235.html`.

[8] Marten Lohstroh et al. "A language for deterministic coordination across multiple timelines". In: *2020 Forum for Specification and Design Languages (FDL)*. IEEE. 2020, pp. 1–8.

[9] Marten Lohstroh et al. "Reactors: A deterministic model for composable reactive systems". In: *Cyber Physical Systems. Model-Based Design: 9th International Workshop, CyPhy 2019, and 15th International Workshop, WESE 2019, New York City, NY, USA, October 17-18, 2019, Revised Selected Papers 9*. Springer. 2020, pp. 59–85.

[10] Marten Lohstroh et al. "Toward a lingua franca for deterministic concurrent systems". In: *ACM Transactions on Embedded Computing Systems (TECS)* 20.4 (2021), pp. 1–27.

[11] Christian Menard et al. "High-performance deterministic concurrency using lingua franca". In: *ACM Transactions on Architecture and Code Optimization* 20.4 (2023), pp. 1–29.

[12] I. Molnar. *Modular Scheduler Core and Completely Fair Scheduler [CFS]*. 2007. URL: https://lwn.net/Articles/230501/.

[13] Chandandeep Singh Pabla. *Completely Fair Scheduler*. URL: https://dl.acm.org/doi/fullHtml/10.5555/1594371.1594375.

[14] *Red Hat Documentation Real Time Throttling*. URL: https://access.redhat.com/documentation/en-us/red_hat_enterprise_linux_for_real_time/7/html/tuning_guide/real_time_throttling.

[15] Marcus Rossel et al. "Provable Determinism for Software in Cyber-Physical Systems". In: *15th International Conference on Verified Software: Theories, Tools, and Experiments (VSTTE)*. 2023, pp. 1–22.

[16] Michael Zimmer et al. "FlexPRET: A processor platform for mixed-criticality systems". In: *2014 IEEE 19th Real-Time and Embedded Technology and Applications Symposium (RTAS)*. IEEE. 2014, pp. 101–110.