# Programming Time-Predictable Processors with Lingua Franca

**Magnus Mæhlum**
Norwegian University of Science and Technology, Trondheim, Norway

**Erling Rennemo Jellum**
University of California, Berkeley, CA, USA

**Shaokai Lin**
University of California, Berkeley, CA, USA

**Marten Lohstroh**
University of California, Berkeley, CA, USA

**Martin Schoeberl**
Technical University of Denmark, Denmark

**Sverre Hendseth**
Norwegian University of Science and Technology, Trondheim, Norway

**Edward A. Lee**
University of California, Berkeley, CA, USA

## Abstract

Precision-timed (PRET) machines are an alternative to modern processors that provide precise control over the timing of software execution. This paper describes a platform for developing predictable real-time embedded systems that pair PRET machines with Lingua Franca (LF), a recent reactor-based coordination language with temporal semantics. Specifically, we port LF to FlexPRET, a PRET machine with flexible hardware thread scheduling. We evaluate single-threaded LF with a tight control loop style application on four embedded platforms, including the FlexPRET. The results reveal the underlying platform's timing variability and how LF plus FlexPRET can remedy this timing variability. Finally, we compare single-threaded to multithreaded LF, again concerning timing. The four embedded platforms used are FlexPRET (bare-metal), RP2040 (bare-metal), nRF52 (with Zephyr), and Raspberry Pi 3b+ (with Linux). Our results indicate that FlexPRET with LF is attractive when precise timing is essential.

**2012 ACM Subject Classification** Computer systems organization → Embedded and cyber-physical systems; Computing methodologies → Distributed programming languages

**Keywords and phrases** Real-time systems, time-predictable architecture, embedded system, coordination language

**Digital Object Identifier** 10.4230/OASIcs.NG-RES.2025.1

**Supplementary Material**

*Software (Source Code)*: `https://github.com/magnmaeh/programming-pret-machines` [21]
archived at `swh:1:dir:8aaaf4d001e8c25c7126b4aafa70a56c9b91725c`

## 1 Introduction

Modern processors implement out-of-order execution, dynamic branch prediction, multi-level caching, and deep pipelines to improve average-case performance. These techniques, particularly when coupled with threaded execution and interrupts, hurt timing predictability [3, 15, 22, 25]. In real-time systems, correct program behavior is determined not only by logical function but also by its timing. Failing to reach just a single deadline can have severe consequences [10]. As such, the concern is not about average-case performance but rather worst-case execution time (WCET).

Lee et al. [13] found that engineers typically resort to some combination of the following to ensure correct execution of real-time software: (1) over-provisioning of processor capabilities, (2) using old technologies, (3) WCET analysis, and (4) real-time operating systems (RTOSes).





These techniques make it possible to develop real-time embedded systems, but they should not be necessary. The problem lies in the many layers of abstraction that discard timing semantics: instruction set architectures (ISAs), programming languages, RTOSes, and networks [3]. Precision timed (PRET) machines, and Lingua Franca (LF) are recent efforts to solve these problems.

PRET machines are a family of processors that provide cycle-accurate timers, a predictable memory hierarchy, and an interleaved pipeline [3, 13, 17]. They address the growing issue of unpredictability because timing semantics are no longer abstracted away at the ISA level. There are many members of the PRET machine family [2, 4, 7, 8, 18, 23], of which FlexPRET is a recent addition [34]. It is designed for mixed-criticality systems, providing guarantees of scheduling hard real-time tasks and flexibility for non-critical tasks. Specifically, hard real-time tasks can be temporally isolated and partially spatially isolated. Furthermore, FlexPRET's ISA contains custom timing instructions that maintain nanosecond precision.

PRET machines have been around since 2007 [3], but one major issue remains today: How are they best programmed? It is non-trivial to answer because (1) PRET machines only achieve competitive throughput when multiple hardware threads are interleaved in the pipeline [13] and (2) the programming should leverage the timing semantics available in the ISA. Previously, PRET machines have been programmed with various extensions to C [24, 31, 32]. Instead, we propose and evaluate the reactor model of computation with the polyglot coordination language Lingua Franca.

The paper is organized as follows. Section 2 provides the background on FlexPRET and LF. Section 3 presents the FlexPRET firmware extension enabling the execution of LF programs. In Section 4, we evaluate the timing characteristics of an LF program with four embedded platforms. Section 5 discusses related work. Section 6 concludes the paper.

# 2 Background

## 2.1 FlexPRET

FlexPRET is an open-source RISC-V processor with a 5-stage in-order pipeline [35]. It implements fine-grained multithreading and has an always-taken branch predictor. Scratchpad memories provide constant memory access latency as opposed to traditional cache hierarchies. It extends the RISC-V ISA with custom timing instructions to enable the developing of time-sensitive applications.

FlexPRET fetches instructions from different threads during each cycle. The instructions are interleaved in the pipeline with instructions from other *hardware threads*. Each hardware thread has its program counter, register file, and other control registers. A hardware scheduler manages which thread to run at each clock cycle. With multiple threads interleaved in the pipeline, the spacing between dependent instructions increases, which reduces the number of pipeline bubbles. This increases the overall throughput.

FlexPRET's hardware threads can be configured as a hard real-time thread or a soft real-time thread. Hard real-time threads have temporal isolation, meaning they can guarantee a fraction of all processor cycles independent of the execution of other threads. When a hard real-time thread finishes its task, it may enter a sleeping state, which frees up its cycles to be used by soft real-time threads. In a multicore system, the user might assign execution of the main application to core 0 and interrupt handling to core 1 to provide temporal isolation for core 0. FlexPRET can achieve the same by assigning the main application to hardware thread 0 and interrupt handling to hardware thread 1.

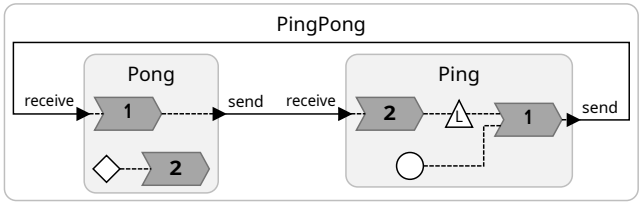



**Figure 1** An auto-generated diagram representing an example program.

## 2.2 Lingua Franca

Lingua Franca (LF) is a polyglot coordination language introducing determinism to concurrent and distributed systems [19] by using actors exchanging messages with time stamps [20]. Figure 1 shows an auto-generated diagram representing a LF program. In LF, a program comprises stateful components called *reactors* represented as rounded rectangles in the diagram. In Figure 1 there are three reactors: the `Ping` reactor, the `Pong` reactor, and the top level `main` reactor. In this LF program, the `Ping` reactor sends integers to the `Pong` reactor, which echos them back.

Reactors encapsulate event-triggered *reactions* represented as dark grey chevrons in the diagram. The reactions encapsulate a sequential program called the reaction body, written in the target language. A reaction is associated with a set of *triggers* and *effects*. Triggers include input ports of the parent reactor, represented as black triangles; actions, represented as white triangles; timers, represented as clocks; and the two builtin triggers *startup* and *shutdown*, represented by a white circle and a white diamond, respectively. Triggers can be typed, meaning that their events also carry a payload. E.g., the input port `receive` of `Ping` has the type `int`. In the diagram, a dashed line between a reaction and a trigger or an effect indicates a dependency.

A reaction is invoked whenever there is an event at any of its triggers, and it might produce events that affect any of its effects, which include output ports and actions. In LF, it is the responsibility of the runtime to execute the reactions such that all of their dependencies are met, and determinism is assured. This relieves the programmer from handling low-level synchronization primitives such as locks, semaphores, and condition variables. Reactors communicate through `connections`, drawn as solid lines connecting input and output ports. An input port can only be driven by a single upstream output port.

LF has a semantic notion of logical time. Each event has a tag that denotes the logical time it should be handled. The logical time of an event imposes a lower bound on the physical time at which it is handled. A deadline can be associated with a reaction to impose an upper bound. It specifies the maximum allowed discrepancy between the logical and the physical time at which the reaction is invoked. A violation will trigger the deadline miss handler supplied by the programmer.

LF introduces determinism to concurrent and distributed software, a property rarely found in these systems [10, 12, 14]. Given an initial state and a given input data to an LF program, LF imposes a partial order on the triggering reactions to ensure data determinism. Computed data will not depend on scheduling decisions nor on execution times (unless deadlines are violated). We refer to this as *logical* determinism. PRET machines have *temporal* determinism. Given its input data, a program executed on a PRET machine defines a unique timing behavior. When PRET machines are programmed with LF, we can expand the model of computation to encompass temporal *and* logical determinism. As such, the correctness of the system is determined not only by the ordering of outputs but also by their timing.

```
target C;
reactor Sensor {
  output out: int
  physical action a(0 msec, 10 msec):int
  reaction(a) -> out {=
    int filtered = filter(a->value);
    lf_set(out, filtered);
  =}
}
reactor Actuator {
  input in: int
  reaction(in) {=
    // Handle filtered sensor message.
  =} deadline(5 msec) {=
    // Handle deadline violation.
  =}
}
main reactor {
  sensor = new Sensor()
  act = new Actuator()
  sensor.out -> act.in
}
```

**Listing 1** A LF program reaction to asynchronous events with a deadline.

LF is particularly suited for the programming of PRET machines because:

**(1)** It has a deterministic semantics establishing well-defined ordering constraints on processing events. As such, we move away from programming PRET machines with threads.

**(2)** It simplifies concurrent development and exploits its knowledge of independent reactors to execute them in parallel. This increases the overall throughput of a PRET machine.

**(3)** It has a semantic notion of time, which lets the developer specify time-sensitive tasks, enforce deadlines, and handle deadline misses. This can directly map to semantics in a PRET machine's ISA, which yields efficient implementation.

**(4)** It is based on a scalable programming model that can efficiently be executed on multicore and distributed systems. For future work, we intend to extend LF support to InterPRET [8], a multicore PRET machine based on FlexPRET.

# 3 Lingua Franca for FlexPRET

In this section, we present FlexPRET's LF runtime support. We begin by motivating the use of reactor-oriented programming for PRET machines before delving into the details of supporting single-threaded and multithreaded LF programs.

## 3.1 Motivation

Our overarching goal is to develop predictable real-time systems. PRET machines are time-predictable on the microarchitectural level and, as such, lay the foundation for our goal. Most real-time systems will have concurrent tasks, each performing different functions with different timing requirements. This is also matched by the PRET machines' microarchitectural support for concurrency through hardware threads. However, if there is any interdependency

```
reaction(a) -> out {=
    int filtered = a->value;
    fp_interrupt_on_expire(lf_time_logical() + MSEC(5), timeout);
    filtered = filter(a->value);
    fp_interrupt_on_expire_cancel();
timeout:
    lf_set(out, filtered);
=}
```

**Listing 2** An LF program that reacts to asynchronous events with a deadline.

among the different concurrent tasks executing on the various threads, ensuring deterministic execution becomes challenging. Typically, programmers rely on mutexes, semaphores, and condition variables to orchestrate such tasks, all of which are error-prone techniques.

LF addresses these issues and provides deterministic coordination of concurrent real-time tasks. A network of reactors will denote, at compile time, a well-defined ordering of reaction executions. However, to deliver real-time computations in LF, we must be able to provide an upper bound on the execution times of reactions.

Consider the LF program shown in Listing 1. It consists of two reactors. `Sensor` has a physical action `a` that is triggered by an asynchronous event with a `min_delay` of 0 and a `min_spacing` of 10 ms. The events are filtered and passed to `Actuator` that reacts with a 5 ms deadline. If this is a hard real-time system, we must guarantee that the `Actuator` does not miss any deadlines. By running this program on a FlexPRET, it is possible to derive a tight bound on the WCET of the reaction in `Sensor`. Suppose we can compute a WCET of less than 5 ms (neglecting the runtime overhead). In that case, we can statically guarantee the latency from handling the asynchronous input until triggering the `Actuator` reaction. However, this alone is insufficient; we might miss deadlines if the asynchronous events arrive too frequently. In LF, this is handled by the `min_spacing` argument on the physical action, which creates a lower bound on the time between events. The LF runtime handles events that violate this bound according to some policy (the events are either deferred, dropped, or replaced). In the case of Listing 1, we are guaranteed not to receive more than one asynchronous event per 10 ms. Not all reaction bodies will allow for deriving a sufficiently small WCET; for some reaction bodies, no WCET can be derived. With FlexPRET, we can easily handle this using the `interrupt_on_expire` instruction combined with `setjmp` and `longjmp`. The reaction of `Sensor` can be rewritten as shown in Listing 2. Now, the reaction is guaranteed to terminate within 5 ms even if the execution time of the function `filter` is greater than 5 ms. This programming style can be dangerous because it can leave a program in an inconsistent state, and any updates to shared global variables should be done with interrupts disabled.

## 3.2 Single-Threaded Lingua Franca Support

Figure 2 shows an overview of the software stack executing LF programs on the FlexPRET platform. At the top of the stack is the *LF application* written by the user. It builds on both the *LF runtime* and the *FlexPRET library*, as the user may opt-in to use functionality from the FlexPRET libraries directly. The LF runtime exposes an application programming interface (API) to the LF application with functions for reading and writing from ports, reading the current logical and physical time, and scheduling new events.

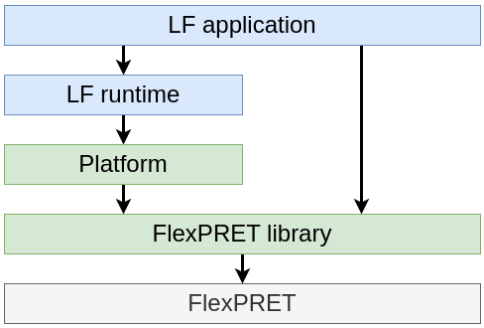



**Figure 2** LF's software stack for FlexPRET.

The LF runtime itself is built in terms of a generalized API implemented in *Platform*, with functions and type definitions. Examples are `void lf_initialize_clock(void)` or `int lf_critical_section_enter(void)`. Adding support for a platform means implementing these functions for the particular platform. Since FlexPRET has native support for timing functionality, many implementations directly map to its custom instructions, which yields efficient implementations.

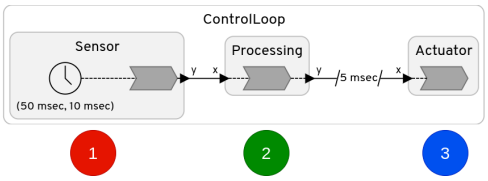
The LF C runtime can run in single-threaded or multithreaded modes. In the single-threaded mode, the LF runtime executes on a single FlexPRET hardware thread. The other hardware threads can be used for different tasks, such as interrupt handling. The two are temporally isolated by running the LF application on one hardware thread and interrupt handling on another. This protects the timing characteristics of the LF application. Assuming they do not share memory, they are isolated, as if running on two isolated processors. Another possibility is to run an entirely different program, possibly written in LF.

## 3.3 Multithreaded Lingua Franca Support

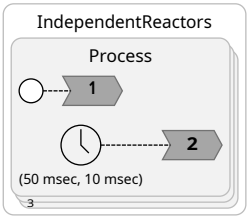
In the multithreaded mode, the LF runtime coordinates multiple workers, each executing within a thread. FlexPRET exposes *hardware* threads to the LF scheduler instead of the usual *software* threads. Context switching between hardware threads is round-robin every clock cycle, unlike software threads that typically have a millisecond granularity. This gives the single-core FlexPRET the appearance of executing several threads *in parallel*, like a multicore processor. Multiple FlexPRET hardware threads can control time-sensitive applications involving multiple actuators simultaneously.

However, FlexPRET cannot support more than $N$ worker threads, where $N$ is the number of hardware threads available on the core ($N$ is 8 in current implementations). Software threads and a scheduler can be built on top of one or more hardware threads to increase the number of threads, but this defeats the purpose of the PRET machine. Fortunately, LF delivers data determinism for any number of worker of threads.

Supporting multithreaded LF on FlexPRET requires additional function implementations, primarily related to running and synchronizing threads. FlexPRET implements all thread synchronization through a single hardware lock (a single bit wide). Acquiring and releasing the lock are atomic operations, and only a single thread may hold the lock at any time. Because of this, the lock can compose any number of operations into an atomic one. We use the single hardware lock to implement other thread synchronization primitives, such as mutexes, semaphores, and conditional variables. Because the critical sections using the hardware lock are very short, we implement it as a spinlock.




**(a)** A tight control loop benchmark. The colors are legends for the upcoming histograms.




**(b)** Independent processing benchmark. There are no data dependencies among the reactors.

**Figure 3** The LF benchmark programs we use in the evaluation.

We found that FlexPRET's hardware lock mechanism can become a bottleneck. The multithreaded LF runtime relies on many synchronization primitives, which may cause a lot of congestion. Multithreaded LF will be more suitable once FlexPRET adds better hardware support for thread synchronization, e.g., like in Strøm et al. [30]

# 4 Evaluation

The evaluation consists of two parts. In Section 4.1-4.2, we implement a tight control loop in LF in single-threaded mode and benchmark its timing characteristics. We compare the results across four different embedded platforms. A complete description of the setup can be found on Github.[1] In Section 4.3, we compare the single-threaded and multithreaded LF runtime for FlexPRET with other commonly used embedded platforms.

## 4.1 Benchmark Description

Control systems often run with a fixed period and consist of (1) the sampling of sensors, (2) processing and sensor fusion, and (3) driving of actuators. Variations in the period between two invocations of the same stage are known as jitter. Jitter can harm control system performance concerning actuation (i.e., output jitter) and sampling (i.e., input jitter).

In the first experiment, we evaluate the timing characteristics of a tight control loop shown in Figure 3a. We sample the time when the Sensor reactor transmits a measurement to the Processing stage (1). Next, we sample the time to run the Processing stage (2). Like (1), we sample when the Actuator receives a command from the Processing stage (3).

The stages in the control loop are simulated. The Sensor samples a normal distribution and transmits it to the Processing stage. In general, the execution time of control algorithms may depend on the sensor data, e.g., if the control algorithm runs an optimization. To simulate this behavior, we implement the Processing stage as `for (volatile int i = 0; i < x; i++);` where $x$ is the measurement from the sensor. This introduces jitter to the Processing stage. We then leverage LF's *delayed connections* to decouple the connection between Processing and Actuator, which adds a logical delay to the data to remove the jitter from the earlier stages, as in the logical execution time (LET) principle [5, 6].

We evaluate the control loop on four different embedded platforms. Table 1 gives some key parameters for each platform. FlexPRET is synthesized for a Zedboard Zynq-7000 Field Programmable Gate-Array. The RP2040 has a default clock frequency of 133 MHz, but is

[1] `https://github.com/magnmaeh/programming-pret-machines`

**Table 1** An overview of all four platforms evaluated.

| Platform | #cores@freq | Abstraction layer | Processor(s) |
|---|---|---|---|
| FlexPRET | 1@100MHz | Bare-metal | RISC-V custom |
| RP2040 | 1@64MHz | Bare-metal | ARM Cortex-M0+ |
| nRF52dk_nRF52832 | 1@64MHz | Zephyr RTOS [9] | ARM Cortex-M4F |
| Raspberry Pi 3b+ | 4@1.4GHz | Raspberry Pi OS (formerly Raspbian) | ARM Cortex-A53 |

reduced to 64 MHz to make it comparable to nRF52. The Raspberry Pi 3b+ (RPi) operating system has a default configuration and runs some services in the background. We have made no effort to optimize RPi for real-time applications; this is outside the scope of this paper.

To evaluate the robustness of each platform, we trigger interrupts while the benchmark runs. We distinguish between *periodic* and *sporadic* interrupts. We transmit the periodic interrupts with a fixed period of 10.5 ms, which is selected not to match the period of the tight control loop, which is 10 ms. The periodic interrupts could emulate, e.g., a sensor interrupt. In addition, we transmit interrupts sporadically and in bursts, which simulate a network connection. The periodic and sporadic interrupts have an interrupt handler, which performs a fixed amount of computation. Depending on which platform is used, this might disrupt the timing characteristics of the benchmark. All platforms receive identical interrupt signals, and throughout the benchmark, we transmit approximately 1000 periodic interrupts and 100 sporadic interrupts.

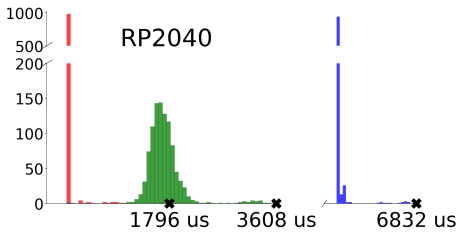
The physical setup is shown in the code repository.[2] After completing the benchmark, a computer uploads executables to the target platform and receives timestamps. We use a Digilent Analog Discovery as a waveform generator. It transmits periodic and sporadic interrupt signals after receiving a trigger signal from the platform. The computer predetermined the exact waveforms, making it possible to transmit identical waveforms to all platforms.

## 4.2 Benchmark Results

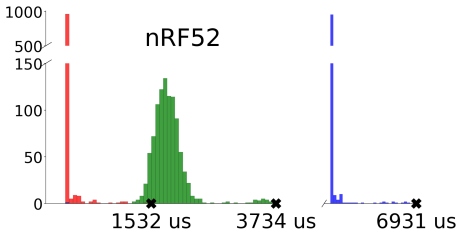
Figure 4 shows the results for all platforms. The RP2040 and nRF52 are comparable platforms and yield similar results. They both have long tails behind the most recurring execution times from interrupt handling. In Figures 4a and 4b, we mark three points on the x-axis, each corresponding to the worst observed execution time of each stage in the control loop. E.g. for the RP2040, the worst observed execution time for the Sensor stage is 1796 us and 3608 us for the Processing stage. However, when no interrupts occur, the Sensor and Actuator stages execute within 100 us. Therefore, interrupts significantly impact the execution time of the reactions. As expected, we conclude that the RP2040 and nRF52 platforms are susceptible to jitter from interrupt handling.

As seen in Figure 4c, the RPi does not have long tails like the RP2040 and nRF52 but instead has a few rare outlier points. The RPi has four processor cores and runs interrupt handlers on a core different from the one running the application. Therefore, the timing characteristics of the benchmark are unaffected by interrupts. The outlier points instead come from the operating system (OS), such as running background services or handling page faults. The programmer cannot control such OS issues without changing the underlying OS, and it is difficult to determine how they impact the timing characteristics of the application. In Figure 4c, the worst observed execution time for the Sensor stage is 3037 us, almost 3000 us

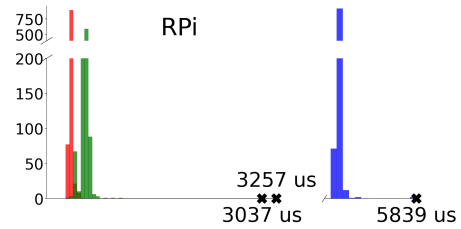
[2] `https://github.com/magnmaeh/programming-pret-machines`



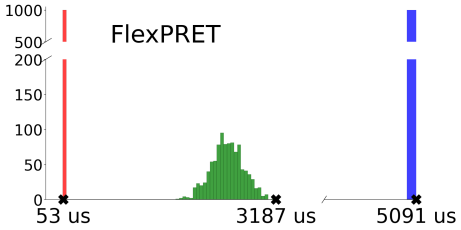

**(a)** RP2040 has a long "tail" behind the most recurring execution times stemming from interrupt handling.




**(b)** nRF52 has a long "tail" behind the most recurring execution times stemming from interrupt handling.




**(c)** RPi has some jitter in all stages. It is unaffected by interrupts, but operating system features introduce jitter.



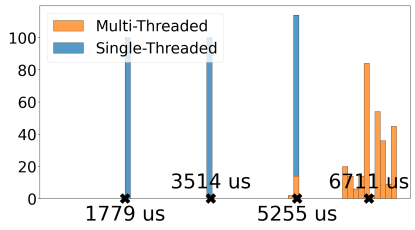

**(d)** FlexPRET has no jitter in the Sensor and Actuator stages. Interrupts introduce no jitter. However, it spends a relatively long time inside the Processing stage.

**Figure 4** The results from running the tight control loop benchmark in Figure 3a on the different platforms. The x-axis is broken from the maximum time measured in the Processing stage until approximately 5000 $\mu$s. The width of the bars has no meaning; they can appear wider because we 'zoom' into certain intervals on the x-axis.

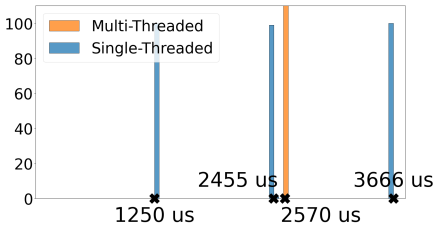
more than the average case. However, the outlier in the Actuator stage is much less severe, only adding approximately 800 us to the average case. It is difficult to determine whether we will observe even worse execution times. We conclude that the RPi is mainly immune from the jitter introduced by interrupt handling, but the underlying platform (including OS) is inherently unpredictable.

Figure 4d shows the results for FlexPRET. FlexPRET has temporal isolation between the benchmark application and the interrupt handlers, much like the RPi running the interrupt handlers on idle processor cores. Unlike the RPi, however, the FlexPRET is programmed in bare-metal and does not suffer from OS timing issues. On the other hand, the FlexPRET spends a relatively long time inside the Processing stage. This is a combination of a relatively low clock frequency of 100 MHz and using multiple hardware threads. The hardware thread running the benchmark is configured as a hard real-time thread with a schedule granting it 1/4 of all processor cycles, resulting in effectively 25 MHz. Aside from the relatively slow execution, FlexPRET has no jitter in the Sensor and Actuator stages. The jitter in the Processing stage is inserted by design, and it does not propagate to the Actuator stage due to a delayed connection from LF.

Two takeaways from Figures 4a-4d: First, LF applications on FlexPRET run with negligible jitter compared to the other platforms. On the FlexPRET, the execution time of the Sensor stage is exactly 47.280 us *every* iteration of the control loop, except for the first iteration due to initialization of the LF runtime. In contrast, the RP2040 and nRF52 have moderate jitter in these stages when no interrupts trigger and extreme jitter when they do. Second, despite the jitter stemming from the underlying platform or data-dependent processing, the LF's delayed connection successfully filters out the jitter from the Sensor and Processing stages.




**(a)** The results for nRF52.




**(b)** The results for FlexPRET. The height of the multithreaded bar is 300.

**Figure 5** The results from running the independent reactors benchmark from Figure 3a in unthreaded and threaded mode for FlexPRET and nRF52.

## 4.3 Multithreaded Lingua Franca

To compare LF in single-threaded and multithreaded mode running on the FlexPRET, we implement a simple LF application consisting of three identical reactors as shown in Figure 3b. Each reactor contains a single reaction that performs a long, constant-time computation. In LF, the three reactions will execute logically simultaneously. However, the actual physical execution timeline will depend on the platform. We execute this program in multithreaded and single-threaded modes on the FlexPRET and nRF52 with Zephyr.

Figure 5a shows the results for nRF52. The x-axis displays the time. With the single-threaded LF runtime, the three reactions execute sequentially. The reactions finish at different physical times: 1779 us, 3514 us, and 5255 us. In addition, the LF runtime may execute the reactions in any order. This means a given reactor might finish at very different physical times when executed twice, e.g., at 1779 us one iteration and at 5255 us another iteration.

Using the multithreaded LF runtime on the nRF52 adds much overhead. We configure Zephyr's scheduler to be preemptive with a time slice of 1 ms. This means any running thread must yield execution to other threads after 1 ms. We also configure LF to use three worker threads. From Figure 5a we can see that using the multithreaded runtime adds quite a lot of overhead; this is a combination of overhead from LF multithreaded runtime, overhead from Zephyr's scheduling algorithm running every 1 ms, and the coarse granularity of the system clock that triggers the scheduler. However, there is less variation in the physical times the reactions finish.

Next, we consider the results for FlexPRET in Figure 5b. Similar to the nRF52, the single-threaded LF runtime executes the reactions sequentially. The FlexPRET finishes execution quicker than the nRF52 because its clock frequency is 100 MHz instead of 64 MHz, and the FlexPRET's hardware thread executing the LF application is given all clock cycles. (In the previous benchmark, the LF application was given only 1/4 of all clock cycles, essentially running it at 25 MHz.) Similar to the nRF52, sequential execution of the reactions means they finish at widely different physical times.

Changing to the multithreaded LF runtime has two consequences. First, all reactions finish execution at approximately the same time. They appear to finish simultaneously in the plot, but in reality, the execution time is distributed around 2570 us with a standard deviation of 7.8 us. The small amount of jitter probably comes from FlexPRET's hardware synchronization primitives. Second, the total time to execute all three reactions decreases. As established in Section 2.1, interleaving multiple hardware threads in the pipeline increases FlexPRET's overall throughput. The effect is strengthened by the aforementioned long computation being implemented as `for (volatile int i = 0; i < 10000; i++);`. This compiles into frequent load and branch instructions, the latter being the culprit of pipeline bubbles in single-threaded execution.

## 5 Related Work

MultiPRET [7] is a multicore version of FlexPRET using shared memory for inter-core communication. The authors provide runtime support for the synchronous language ForeC [33].

XCore [23] is a commercial multicore PRET architecture developed by XMOS. It is also based on a barrel register file with fine-grained multithreading of hardware threads. Unlike MultiPRET, XCore does inter-core communication using a credit-based network-on-chip. The XCores can be programmed with a C-like language called xC, which has semantics based on Communicating Sequential Processes (CSP) [31].

InterPRET [8] is another multicore PRET machine based on FlexPRET. Like the XCore, it is based on a network-on-chip for inter-core communication. Unlike XCore, however, InterPRET is based on a time-triggered, time-predictable network-on-chip [28]. This opens the door for end-to-end time-predictable computations on a many-core processor. InterPRET is currently programmed in C, but we intend to leverage the FlexPRET support for LF to program multicore PRET machines in LF.

Patmos is, like the PRET architecture, a processor designed for real-time systems [29]. For example, the caches are optimized for time-predictability. Like InterPRET, several Patmos processors are combined with a network-on-chip to build the T-CREST multicore architecture [26]. Patmos supports the single-threaded LF runtime [11], including WCET analysis of LF reactions [27]. We are currently developing the multithreaded LF runtime for the multicore T-CREST architecture.

The logical execution time (LET) model, which can be traced back to the time-triggered language Giotto [6], is a related approach that is gaining popularity in the automotive sector and was recently included in the AUTOSAR standard [1]. System-level LET [5] is a recent extension to LET, introducing Time Zones and interconnecting LET, enabling the modeling of distributed systems using LET. The time-predictability of PRET machines makes them an ideal fit for LET-style programming. We focus on LF since it can be considered a generalization of the LET model [16] and provides compilation and runtime infrastructure for both modeling and implementing LET systems.

## 6 Conclusion

This paper presents a new platform for real-time systems based on the reactor-oriented programming of PRET machines. We present runtime support for executing Lingua Franca programs on FlexPRET. We evaluate LF programs on FlexPRET and several other commonly used embedded platforms. Regarding timing, LF on FlexPRET outperforms the other platforms in single-threaded and multithreaded execution. We believe this opens up an interesting design space and can be the basis for promising future work.

## References


1 AUTOSAR. Specification of timing extensions. *AUTOSAR CP Release 4.4.0*, October 2018.

2 M. Delvai, W. Huber, P. Puschner, and A. Steininger. Processor support for temporal predictability - the SPEAR design example. In *15th Euromicro Conference on Real-Time Systems, 2003. Proceedings.*, pages 169–176, 2003. doi:10.1109/EMRTS.2003.1212740.

3 Stephen Edwards and Edward A. Lee. The case for the precision timed (pret) machine. Technical Report UCB/EECS-2006-149, EECS Department, University of California, Berkeley, November 2006. URL: http://www2.eecs.berkeley.edu/Pubs/TechRpts/2006/EECS-2006-149.html.

4 Stephen A. Edwards, Sungjun Kim, Edward A. Lee, Isaac Liu, Hiren D. Patel, and Martin Schoeberl. A disruptive computer design idea: Architectures with repeatable timing. In *Proceedings of IEEE International Conference on Computer Design (ICCD 2009)*, Lake Tahoe, CA, October 2009. IEEE.

5 Kai-Björn Gemlau, Leonie Köhler, Rolf Ernst, and Sophie Quinton. System-level logical execution time: Augmenting the logical execution time paradigm for distributed real-time automotive software. *ACM Trans. Cyber-Phys. Syst.*, 5(2), January 2021. doi:10.1145/3381847.

6 T. A. Henzinger, B. Horowitz, and C. M. Kirsch. Giotto: A time-triggered language for embedded programming. In *EMSOFT 2001*, volume LNCS 2211, pages 166–184. Springer-Verlag, 2001.

7 Nicolas Hili, Alain Girault, and Eric Jenn. Worst-case reaction time optimization on deterministic multi-core architectures with synchronous languages. In *2019 IEEE 25th International Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA)*, pages 1–11, 2019. doi:10.1109/RTCSA.2019.8864570.

8 Erling R. Jellum, Shaokai Lin, Peter Donovan, Chadlia Jerad, Edward Wang, Marten Lohstroh, Edward A. Lee, and Martin Schoeberl. Interpret: A time-predictable multicore processor. In *Proceedings of Cyber-Physical Systems and Internet of Things Week 2023*, CPS-IoT Week '23, pages 331–336, New York, NY, USA, 2023. Association for Computing Machinery. doi:10.1145/3576914.3587497.

9 Erling Rennemo Jellum, Shaokai Lin, Peter Donovan, Efsane Soyer, Fuzail Shakir, Torleiv Bryne, Milica Orlandic, Marten Lohstroh, and Edward A. Lee. Beyond the Threaded Programming Model on Real-Time Operating Systems. In Federico Terraneo and Daniele Cattaneo, editors, *Fourth Workshop on Next Generation Real-Time Embedded Systems (NG-RES 2023)*, volume 108 of *Open Access Series in Informatics (OASIcs)*, pages 3:1–3:13, Dagstuhl, Germany, 2023. Schloss Dagstuhl – Leibniz-Zentrum für Informatik. doi:10.4230/OASIcs.NG-RES.2023.3.

10 Mike Jones. What really happened on Mars Rover Pathfinder. Technical report, Microsoft, 1997.

11 Ehsan Khodadad, Luca Pezzarossa, and Martin Schoeberl. Towards lingua franca on the patmos processor. In *Proceedings of the 2024 IEEE 27th International Symposium on Real-Time Distributed Computing (ISORC)*, 2024.

12 Phil Koopman. A case study of toyota unintended acceleration and software safety. Technical report, ECE Department, Carnegie Mellon University, 2014.

13 Edward Lee, Jan Reineke, and Michael Zimmer. Abstract PRET machines. In *2017 IEEE Real-Time Systems Symposium (RTSS)*, pages 1–11, 2017. doi:10.1109/RTSS.2017.00041.

14 Edward A. Lee. The problem with threads. *IEEEC*, 39(5):33–42, 2006. doi:10.1109/MC.2006.180.

15 Edward A. Lee. Computing needs time. *Communications of the ACM*, 2009.

16 Edward A. Lee and Marten Lohstroh. *Generalizing Logical Execution Time*, pages 160–181. Springer Nature Switzerland, Cham, 2022. doi:10.1007/978-3-031-22337-2_8.

17 Isaac Liu. *Precision Timed Machines*. PhD thesis, EECS Department, University of California, Berkeley, May 2012. URL: http://www2.eecs.berkeley.edu/Pubs/TechRpts/2012/EECS-2012-113.html.

18 Isaac Liu, Jan Reineke, David Broman, Michael Zimmer, and Edward A. Lee. A PRET microarchitecture implementation with repeatable timing and competitive performance. In *Proceedings of the 30th IEEE International Conference on Computer Design (ICCD 2012)*, October 2012. URL: http://chess.eecs.berkeley.edu/pubs/919.html.

19 Marten Lohstroh, Christian Menard, Soroush Bateni, and Edward A. Lee. Toward a lingua franca for deterministic concurrent systems. *ACM Transactions on Embedded Computing Systems*, 2021.

**20** Marten Lohstroh, Martin Schoeberl, Andrés Goens, Armin Wasicek, Christopher Gill, Marjan Sirjani, and Edward A. Lee. Actors revisited for time-critical systems. In *Proceedings of the 56th Annual Design Automation Conference 2019*, DAC '19, pages 152:1–152:4, New York, NY, USA, June 2019. ACM. doi:10.1145/3316781.3323469.

**21** Magnus Mæhlum. Programming PRET machines. Software, swhId: swh:1:dir:8aaaf4d001e8c25c7126b4aafa70a56c9b91725c (visited on 2025-03-17). URL: https://github.com/magnmaeh/programming-pret-machines, doi:10.4230/artifacts.23000.

**22** Goncalo Martins, Dave Lacey, Allistair Moses, Matthew J. Rutherford, and Kimon P. Valavanis. A case for i/o response benchmarking of microprocessors. In *IECON 2012 - 38th Annual Conference on IEEE Industrial Electronics Society*, pages 3018–3023, 2012. doi:10.1109/IECON.2012.6389416.

**23** David May. *The XMOS XS1 Architecture*, 2009.

**24** Saranya Natarajan and David Broman. Timed c: An extension to the c programming language for real-time systems. In *2018 IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, pages 227–239, 2018. doi:10.1109/RTAS.2018.00031.

**25** Martin Schoeberl. Time-predictable computer architecture. *EURASIP Journal on Embedded Systems*, vol. 2009, Article ID 758480:17 pages, 2009. doi:10.1155/2009/758480.

**26** Martin Schoeberl, Sahar Abbaspour, Benny Akesson, Neil Audsley, Raffaele Capasso, Jamie Garside, Kees Goossens, Sven Goossens, Scott Hansen, Reinhold Heckmann, Stefan Hepp, Benedikt Huber, Alexander Jordan, Evangelia Kasapaki, Jens Knoop, Yonghui Li, Daniel Prokesch, Wolfgang Puffitsch, Peter Puschner, André Rocha, Cláudio Silva, Jens Sparsø, and Alessandro Tocchi. T-CREST: Time-predictable multi-core architecture for embedded systems. *Journal of Systems Architecture*, 61(9):449–471, 2015. doi:10.1016/j.sysarc.2015.04.002.

**27** Martin Schoeberl, Ehsan Khodadad, Shaokai Lin, Emad Jacob Maroun, Luca Pezzarossa, and Edward A. Lee. Invited Paper: Worst-Case Execution Time Analysis of Lingua Franca Applications. In Thomas Carle, editor, *22nd International Workshop on Worst-Case Execution Time Analysis (WCET 2024)*, volume 121 of *Open Access Series in Informatics (OASIcs)*, pages 4:1–4:13, Dagstuhl, Germany, 2024. Schloss Dagstuhl – Leibniz-Zentrum für Informatik. doi:10.4230/OASIcs.WCET.2024.4.

**28** Martin Schoeberl, Luca Pezzarossa, and Jens Sparsø. S4noc: a minimalistic network-on-chip for real-time multicores. In *Proceedings of the 12th International Workshop on Network on Chip Architectures*, pages 8:1–8:6. ACM, October 2019. doi:10.1145/3356045.3360714.

**29** Martin Schoeberl, Wolfgang Puffitsch, Stefan Hepp, Benedikt Huber, and Daniel Prokesch. Patmos: A time-predictable microprocessor. *Real-Time Systems*, 54(2):389–423, April 2018. doi:10.1007/s11241-018-9300-4.

**30** Tórur Biskopstø Strøm, Jens Sparsø, and Martin Schoeberl. Hardlock: Real-time multicore locking. *Journal of Systems Architecture*, 97:467–476, 2019. doi:10.1016/j.sysarc.2019.02.003.

**31** Douglas Watt. *Programming XC on XMOS devices*, 2009.

**32** Eugene Yip, Alain Girault, Partha S. Roop, and Morteza Biglari-Abhari. The forec synchronous deterministic parallel programming language for multicores. In *2016 IEEE 10th International Symposium on Embedded Multicore/Many-core Systems-on-Chip (MCSOC)*, pages 297–304, 2016. doi:10.1109/MCSoC.2016.13.

**33** Eugene Yip, Alain Girault, Partha S. Roop, and Morteza Biglari-Abhari. Synchronous deterministic parallel programming for multi-cores with ForeC. *ACM Transactions on Programming Languages and Systems*, 2023.

**34** Michael Zimmer. *Predictable Processors for Mixed-Criticality Systems and Precision-Timed I/O*. PhD thesis, EECS Department, University of California, Berkeley, August 2015. URL: http://www2.eecs.berkeley.edu/Pubs/TechRpts/2015/EECS-2015-181.html.

**35** Michael Zimmer, David Broman, Chris Shaver, and Edward A. Lee. FlexPRET: A processor platform for mixed-criticality systems. *2014 IEEE 19th Real-Time and Embedded Technology and Applications Symposium (RTAS)*, 2014. doi:10.1109/RTAS.2014.6925994.