Beyond the Threaded Programming Model on Real-Time Operating Systems

Erling Rennemo Jellum
NTNU, Trondheim, Norway

Shaokai Lin
University of California at Berkeley, CA, USA

Peter Donovan
University of California at Berkeley, CA, USA

Efsane Soyer
University of California at Berkeley, CA, USA

Fuzail Shakir
University of California at Berkeley, CA, USA

Torleiv Bryne
NTNU, Trondheim, Norway

Milica Orlandic
NTNU, Trondheim, Norway

Marten Lohstroh
University of California at Berkeley, CA, USA

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

Abstract

The use of a real-time operating system (RTOS) raises the abstraction level for embedded systems design when compared to traditional bare-metal programming, resulting in simpler and more reusable application code. Modern RTOSes for resource-constrained platforms, like Zephyr and FreeRTOS, also offer threading support, but this kind of shared memory concurrency is a poor fit for expressing the reactive and interactive behaviors that are common in embedded systems. To address this, alternative concurrency models like the actor model or communicating sequential processes have been proposed. While those alternatives enable reactive design patterns, they fail to deliver determinism and do not address timing. This makes it difficult to verify that implemented behavior is as intended and impossible to specify timing constraints in a portable way. This makes it hard to create reusable library components out of common embedded design patterns, forcing developers to keep reinventing the wheel for each application and each platform. In this paper, we introduce the embedded target of Lingua Franca (LF) as a means to move beyond the threaded programming model provided by RTOSes and improve the state of the art in embedded programming. LF is based on the reactor model of computation, which is reactive, deterministic, and timed, providing a means to express concurrency and timing in a platform-independent way. We compare the performance of LF versus threaded C code – both running on Zephyr – in terms of response time, timing precision, throughput, and memory footprint.

2012 ACM Subject Classification Computer systems organization → Embedded software

Keywords and phrases Real time, concurrency, reactors, Lingua Franca, RTOS

Digital Object Identifier 10.4230/OASIcs.NG-RES.2023.3

Supplementary Material *Software (Source Code)*: `https://github.com/lf-lang/lingua-franca/tree/v0.4.0`

Funding This work was supported by the Research Council of Norway (RCN) grant number 309328, Multi-Sensor Data Timing, Synchronization and Fusion for Intelligent Robots, grant number 327538, the center of excellence NTNU AMOS, grant number 223254, and the iCyPhy Research Center at UC Berkeley.

Acknowledgements We want to thank Matthew Chorlian, Arthur Deng, Abanob Bostouros and Christian Menard for their work on porting the Savina benchmark to the Lingua Franca C target, as well as Hannes Klein and Felix Wittwer for their work on the original port of the Savina benchmark to the Lingua Franca C++ target. We would also like to acknowledge Soroush Bateni and Alexander Schulz-Rosengarten for their work on the Lingua Franca C runtime.

1 Introduction

Microcontrollers are ubiquitous in embedded applications, such as consumer electronics, industrial automation, robotics, and automotive systems. Such applications are concurrent, event-driven, and in many cases, time-sensitive. Traditionally, these systems have been programmed in C, based on a limited shared-memory concurrency model offered by interrupts. However, such programs have long been known to be notoriously difficult to get right [24, 5]. This has contributed to serious bugs and vulnerabilities, sometimes with catastrophic results [11, 10, 14].

Real-time operating systems (RTOSes) offer a richer version of the shared-memory concurrency model through *threads*. Threads enable the user to decompose their application into multiple concurrent activities. The RTOS offers low-level primitives for controlling the scheduling, synchronization, timing, and communication between threads. Just like interrupts, the threaded programming model admits nondeterminism [12] and leaves the error-prone job of synchronizing the threads' accesses to shared variables to the programmer [25]. A concurrency primitive that popular operating systems like Zephyr [4] provide as an alternative to threads is *callbacks*, which are invoked from interrupt service routines [20]. Callbacks do not address the problem of nondeterminism either, but they do create the problem of control flow inversion [5], which makes programs much harder to understand [18]. The RTOS introduces a notion of time, which is useful when designing time-aware applications. However, the timing is too imprecise for many applications. The RTOS will use regular interrupts, called *systicks*, to measure time and provide timing abstractions. The interval between the systicks is user-configurable but typically in the order of milliseconds. For many applications, this level of accuracy is not sufficient [9]. Developers can resort to implementing the timed behavior through callbacks, but this will expose them to the aforementioned problems.

In this paper, we propose Lingua Franca (LF) [16] as a means to enable intuitive concurrent, reactive, and time-aware programming of microcontrollers. LF offers lightweight concurrent objects called reactors, which can be composed in a network. Reactors do not share state but communicate through message passing via named ports. Deterministic concurrency is ensured on the basis of the topology of the reactor network. LF has a semantic notion of time that enables specification of arbitrary timing behavior of a program. Being a coordination language, not a general-purpose programming language, LF lets the programmer express computation in a target language of their choice. The Zephyr support implemented and evaluated in this paper is based on the C target. We compare the implementation of typical patterns in embedded system software using the timed reactor-based concurrency model against achieving the same kinds of behavior with threads.

The remainder of the paper is organized as follows. Section 2 briefly introduces the reactor-oriented paradigm followed by LF and its suitability for embedded system design. Section 3 takes a look at related work. Section 4 introduces the Zephyr support layer we developed for LF. In Section 5 we compare the performance of LF on Zephyr versus threaded C code running directly on Zephyr. Conclusions are provided in Section 6.

2 Embedded Programming in Lingua Franca

In the reactor-oriented programming paradigm followed by LF, concurrency is exposed by the programmer through the composition of stateful components called *reactors* that encapsulate event-triggered *reactions*, leaving the task of concurrency management to the LF compiler and runtime. Programmers need not deal with locks, semaphores, or other low-level synchronization mechanisms. Any two simultaneously triggered reactions that do not have any data dependencies between them may execute in arbitrary order (or in parallel, on a multicore system) without execution order affecting the effects of those reactions. It is the runtime system, not the programmer, that is in charge of ensuring that data dependencies are observed. This is what makes LF programs deterministic by default.

When events are not logically simultaneous, the LF runtime guarantees that they will appear to be processed in order according to their positions, called *tags*, on a totally ordered logical timeline. Through its correspondence to physical time, as measured by the system clock, this logical timeline also determines the program's real-time behavior. Events with well-defined logical times are produced by hooking up reactors to *timers* that produce events periodically, or by scheduling events in reaction code via *actions*. LF's timing constructs can be parameterized to let the runtime control the offsets, periods, and spacing between events. Logical delays and deadlines can be used to specify timing constraints and to handle possible timing violations.

LF separates a declarative specification of a program's coordination fabric from the program's imperative code, which is contained in reaction bodies and written in a target language of choice. The LF toolchain capitalizes on this by providing automatically synthesized interactive diagrams in the IDE. In the diagrams, reactors are shown as rounded boxes. Small black triangles denote ports, and reactions are represented by chevrons. Reactions can be triggered by events originating from upstream ports or actions, connected via dashed lines. There are also distinguished startup and shutdown triggers that, respectively, have events at the very first and very last time instant of program execution. Dashed lines also connect a reaction to its *effects*, which are downstream ports or actions that it can produce events on. Events produced on ports are always instantaneous; those produced on actions are always scheduled to occur at a future time.

To express modal behaviors, where the software has to switch between different modes of operation, LF also has syntax for grouping a reactor's elements in mutually exclusive regions called *modes*. A modal reactor has a single initial mode and can have several other modes. A transition from one mode to another is initiated in reaction code and requires the reaction to specify the target mode among its effects. A mode can be entered via a reset transition (the default) or a history transition. The former initializes all timers and resets any state marked to be reset; the latter resumes operation where it left off when the mode was last exited. Modes and transitions are rendered visually in the diagrams as highlighted areas with arcs between them. The initial mode is marked by a thicker border.

The LF runtime can be compared to an RTOS kernel. Like a kernel, it supports message-passing and context-switching between concurrently executing components. The message-passing is done by setting up direct pointers from input ports to the output ports they are connected to on. Ports carry immutable data by default which enables zero-copy communication. There are two types of context-switches performed by the LF runtime. The first, referred to as *reaction-switching*, involves scheduling a new reaction at the same tag a the previously executed reaction. The second type of context-switch requires advancing the logical time of the program. This is referred to as *logical time advancement*. It might include putting the system into low-power sleep to wait for the physical time to reach that particular value.

2.1 Embedded design patterns

In the following sections we will show how three common design patterns in embedded systems can be expressed easily using Lingua Franca.

```
target C;

reactor ButtonDebounce {
    preamble {=
        void * button_press_action; // Pointer to physical action
        void gpio_pin_isr() {
            lf_schedule(button_press_action, 0);
        }
    =}

    physical action button_press(0, 10 msec)

    reaction(startup) -> button_press {=
        // Setup GPIO pin and register ISR
        ...
        // Make physical action available externally
        button_press_action = button_press;
    =}

    reaction (button_press) {=
        // Handle
    =}
}
```

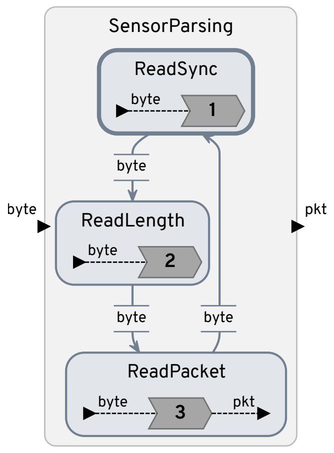
Figure 1 Button debouncing in Lingua Franca.

2.1.1 Button debouncing

When a button is pressed, it may generate multiple spurious open and close transitions. Debouncing is the process of filtering out transitions that are so close in time that they likely are not due to an actual button press or release. LF's timed semantics makes such patterns simple to express. To capture asynchronous events, like button presses, and pin them to LF's logical timeline, a *physical* action is used, which can be scheduled from an external thread or ISR. The tag of such an event is derived not from the runtime's current logical time (as an event scheduled on a *logical* action would), but from the systems's physical time. An action can be assigned a minimum spacing, which will make the scheduler reject events scheduled too close to one another. Fig. 1 shows how this is expressed in the C target of Lingua Franca. The physical action *button_press* has a minimum spacing of 10 ms, and the runtime system handles the debouncing.

2.1.2 Sensor parsing

Consider the common application of parsing an incoming packet from an external sensor connected through a digital communication interface. Figs. 2b and 2a show the implementation and diagram of a generic packet parser reactor in LF. Parsing a stream of bytes is conveniently expressed as a state machine. LF has native support for encoding state machines through modal reactors. The *SensorParsing* reactor has an input port for incoming bytes and an output port for the parsed packet. How the reactor responds to data on its input port depends on its current mode. In the initial mode, *ReadSync*, the incoming bytes are compared against the known sync ID of the packet. Upon finding the sync bytes, the reactor performs a mode change to *ReadLength*, where it reads the length field of the packet. When the length has been decoded, the reactor moves to the *ReadPacket* mode, reads the packet, and outputs the result on the *pkt* output port.



(a) Diagram.

```
reactor SensorParser {
  input byte:char
  output pkt:sensor_pkt_t;

  initial mode ReadSync {
    reaction(byte) -> ReadLength {=
      // Search for sync-byte ...
      lf_set_mode(ReadLength);
    =}
  }

  mode ReadLength {
    reaction(byte) -> ReadPacket {=
      // Read length field ...
      lf_set_mode(ReadPacket);
    =}
  }

  mode ReadPacket {
    reaction(byte) -> pkt {=
      // Read packet into p ...
      lf_set(pkt, p);
    =}
  }
}
```

(b) Source code.

Figure 2 A sensor packet parser in Lingua Franca.

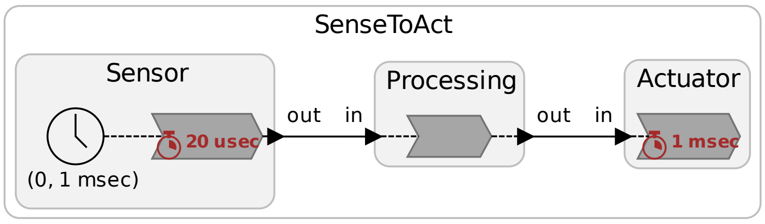


Figure 3 Tight control loop application in Lingua Franca.

2.1.3 Control loops

Another common design pattern is that of tight control loops in, for example, robotics and control. Consider the LF program depicted Fig. 3. It consists of a *Sensor* reactor that reads from a sensor every 1 ms. The sensor values are passed downstream to the *Processing* reactor, which implements a control algorithm. Finally, the control set points calculated by *Processing* are applied to the actuators in the *Actuator* reactor.

Notice that there are two deadlines. First, the *Sensor* reaction is associated with a 20 us deadline. This is a way of expressing a bound on how much jitter we accept in our sensor readings. Likewise, the *Actuator* reactor has a deadline of 1 ms. This corresponds with the period of the control loop and expresses that the overhead starting from obtaining a reading from *Sensor* up to right before invoking *Actuator* should be less than 1 ms. The timed semantics of LF provides an intuitive way to express such control loops and their timing requirements.

3 Related Work

There is a rich body of work exploring ways of raising the abstraction level for embedded software design. SynchronVM [20] is a virtual machine running on top of Zephyr OS or ChibiOS targeting microcontrollers. The underlying concurrency model is based on message passing and resembles communicating sequential processes (CSP) [6]. In SynchronVM one can specify a time window in which a synchronous message passing should occur. The time window informs the scheduling of the processes but does not have a well defined semantics. The underlying model of computation, CSP, is fundamentally nondeterministic as it allows processes to wait for events on multiple channels and synchronize with the first available.

MicroPython [19] is a project bringing the interpreted and garbage-collected high-level language Python to microcontrollers. The MicroPython interpreter supports REPL (Read-Evaluate-Print-Loop) interaction, which greatly simplifies prototyping. However, MicroPython requires FLASH memory on the order of 256KB, and its garbage collection and dynamic typing make it unsuited for real-time and safety-critical applications.

Medusa [2] is a concurrent programming language based on the Actor model [1]. Medusa is an extension of Python and runs an Owl Python runtime for microcontrollers. The Owl runtime has a code size of 38KB, and each Actor requires 100B of RAM. In Medusa, the hardware is also modeled as actors, and the peripherals communicate with the rest of the system through message passing performed by the interrupt service routines. Medusa does not have a semantic notion of time, and the underlying actor model is nondeterministic.

xC [22] is a C-based programming language designed for the Xcore [17] CPU architecture from XMOS. xC also has CSP as its underlying model, but it targets a CPU architecture with hardware support for processes and rendezvous. As such, there is no runtime providing a message-passing API. xC has language primitives representing hardware timers, which provide sleeping and waiting. The Xcore architecture is timing-predictable, and the tools support worst-case execution time analysis.

4 Design and Implementation

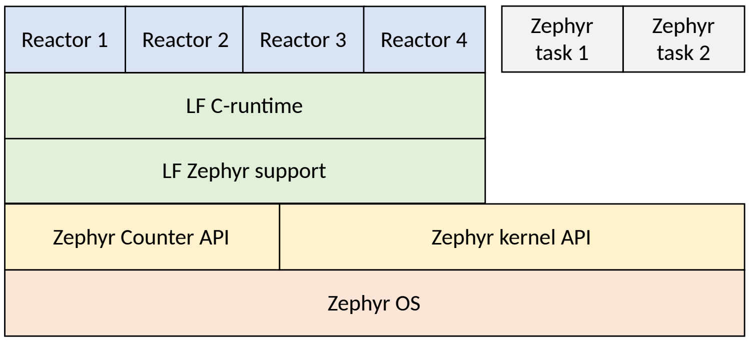


Figure 4 Lingua France runtime stack.

Fig. 4 shows the software stack supporting LF on embedded systems. The Zephyr OS [4] forms the foundation of this stack. This was chosen over a bare-metal target for multiple reasons. First of all, the APIs of the Zephyr kernel and drivers are platform independent, and, at the time of writing, Zephyr supports more than 400 boards. Zephyr also comes with many useful drivers for communication protocols like Bluetooth Low Energy (BLE), Ethernet, USB, and CAN buses. This makes Zephyr an attractive framework even for single-threaded

Table 1 LF code and data size (B is bytes).

	Runtime	Reactor	Reaction	Timer	Action	Mode	Port (In/Out)	Event
Code size	9.8KB(1KB)	96B	62B	88B	174B	36B	32B/16B	–
Data size	266B(170B)	72B	80B	60B	72B	28B	104B/40B	32B

applications. Using Zephyr also enables simple integration of LF programs with existing libraries like TCP/IP stacks or clock synchronization implementations. These can simply execute in separate threads and be scheduled by the Zephyr kernel. Lastly, for running LF on multicore platforms, a threading library is needed in order to make use of all the cores. Zephyr implements a POSIX-compliant threading library.

In this paper we focus on the single-threaded LF C runtime. Like any LF target port, this one must implement a few key functions to (1) enter and leave critical sections; (2) sleep for a duration or until interrupted asynchronously; and (3) read a monotonically increasing physical clock. A critical section is created by disabling all interrupts; this functionality is exposed by the kernel API. Sleeping can be achieved in various ways, but the Zephyr kernel provides the function *k_sleep* which puts the calling thread to sleep for a number of clock ticks. A physical clock can be read using *k_cycle_get*_32. However, on many platforms, this clock, which underlies the kernel services, is a low-frequency 32KHz clock. This might not be sufficient for high-precision control or highly dynamic robots [9]. Most platforms will also have high-frequency clocks which can drive hardware timers. Such hardware timers can be used through the platform-independent Counter API. The Counter API is not part of the kernel API and might not be implemented for all platforms.

For multi-threaded support, a full POSIX-like implementation must be provided. Zephyr does implement the POSIX API, but again the granularity of the timing services is too coarse. Here, we use the Counter API again to provide high-precision sleeps.

For platforms that do not have hardware timers or do not implement the Counter API, the Zephyr kernel's timing primitives are used. This drastically reduces precision, but it enables testing and running LF programs on QEMU emulations [3].

5 Evaluation

The evaluation was done on a NRF52832 development kit from Nordic Semiconductor [21]. It has an Arm Cortex M4 microcontroller with 64KB RAM and 512KB flash clocked at 64MHz. Benchmarks were compiled with the west build tool version 0.14.0 using Zephyr version 3.2.0.

5.1 Memory footprint

Memory footprint is a critical property for most microcontroller applications. In the following we provide a detailed evaluation of both the flash and RAM footprints of LF.

5.1.1 Code size

The top row of Table 1 shows the code size of different LF components. The runtime itself consumes less than 10KB and can be further reduced by 3KB if the program has no modal reactors. The 1KB in the parenthesis is the implementation of the Zephyr Counter driver. The Zephyr kernel adds around 19KB which makes a total of less than 30KB. For comparison, Medusa [2] requires a total of 38KB, Espruino [23] and MicroPython [19] requires 100KB-200KB and SynchronVM [20] requires 32KB.

```
target C;

reactor Unbounded {
    logical action a;
    logical action b
    state i:int(0)

    reaction(a,b) -> a,b {=
        lf_schedule(a,0);
        lf_schedule(b,++self->i);
    =}
}
```

(a) LF program with unbounded event queue.

```
target C;

reactor Bounded {
    timer t1(0, 10 msec)
    timer t2(0, 50 msec)
    reaction(t1) {= =}
    reaction(t2) {= =}
    reaction(t1,t2) {= =}
}
```

(b) LF program with bounded event queue.

Figure 5 Memory requirements for LF programs.

A minimal reactor with just a single reaction triggered at startup only requires 96B. Adding reactions, timers and modes all add less than 100B of overhead. The user code in the reaction bodies, as well as the SDK components they depend on, will add to this.

As we will see next, the limiting factor for platforms like the NRF52 is not the size of the program memory but rather the size of RAM.

5.1.2 Data size

The bottom row of Table 1 shows the RAM footprint of LF. It consists of three parts. First, there is the constant size of the runtime, which only amounts to 265B plus 170B for the Counter implementation. Then there is the memory footprint for each LF component like reactions, actions and ports. This is known at compile-time but is currently being allocated dynamically in the initialization routine. Last, we have the events being communicated between the reactors over the ports through the event queue. These events are allocated as needed at run-time. When an event has been consumed, it is recycled to avoid memory fragmentation. Due to the expressiveness of the Reactor model of computation, there is, in general, no upper bound to the size of the event queue. There exist syntactically valid pathological LF programs for which the event queue can grow without bound. For instance, consider the reactor *Unbounded* in Fig. 5a. Each time the action a triggers, two events will be added to the event queue. The first triggers a at the same tag, while the second triggers b at a future tag. Only events triggering a will ever be handled and the events triggering b will accumulate. Such a program is said to exhibit a *stuttering Zeno* condition [13].

However, there also exist classes of “safe” LF programs for which upper bounds on the event queue can be computed. Consider for instance the reactor *Bounded* in Fig. 5b. It is a simple reactor with two periodic timers and three reactions triggered by them. The size of the event queue for this reactor will never exceed two. At any time during execution, there is one event per timer on the event queue. In fact, any LF program that uses only timers to drive logical time forward will have a statically computable bound on the size of its event queue.

5.2 Timing precision

One of the advantages of using LF for embedded applications is its timed semantics. An implementation of LF has good timing precision if the physical time at which a reaction is executed is close to the logical time at which it was scheduled to execute.

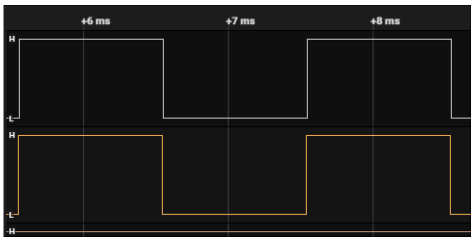


Figure 6 Logical analyzer dump.

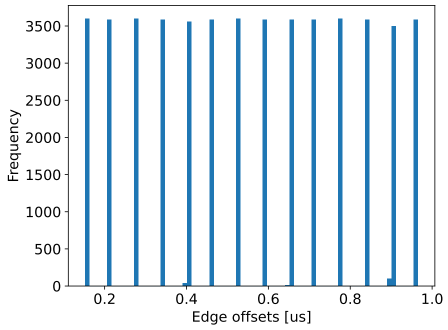


Figure 7 Actuator driving precision.

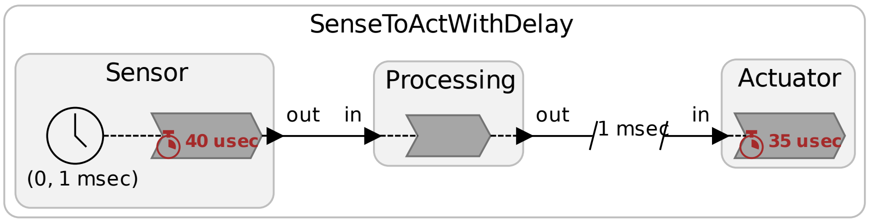


Figure 8 Tight control loop with after-delay.

5.2.1 Square-wave generator

To test the timing precision, the NRF52 was configured to generate a 1 KHz square wave on an output pin using hardware timers. We created a simple LF program to generate the same signal using a timer and a reaction that toggles a GPIO pin.

Fig. 6 shows the two resulting square waves as measured by a Salaea Logic Pro logic analyzer. Fig. 7 shows a histogram of the offset between the LF square wave and the hardware generated square wave. Clearly, the LF runtime can deliver precise and repeatable timing. All observed offsets were less than 1 μs. The multi-modality of the distribution is due to inaccuracies associated with waking up from sleep. This could be improved by targeting a lower-level timer API; however, such an approach would not be platform-agnostic. An important observation is that this timing precision can be affected by independent long-running reactions due to barrier synchronization at each time tag. We are working on new scheduling strategies and language primitives to improve timing precision.

5.2.2 Tight control loops with load

Consider again the SenseToAct example from Fig. 3 in Section 2.1.3. An interesting performance metric is the load a system can sustain while still meeting its deadlines. By introducing busy-waits in the reaction bodies, different workloads can be emulated. Our experiments show that the runtime overhead for SenseToAct is only 70 μs. This leaves 93% of the 1 ms period for actual computation.

An important observation is that the semantics of an LF deadline only puts an *upper bound* on the lag of a reaction invokation. In fact, for Fig. 3 there is a full 1 ms interval in which the *Actuator* reaction can legally be invoked. To introduce a *lower* bound on the invokation time we can use logical delays. In Fig. 8 we have introduced a 1 ms logical

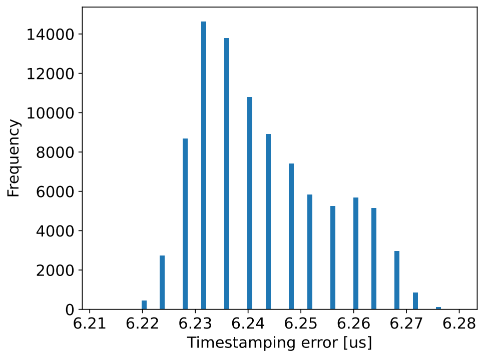


Figure 9 Physical action precision.

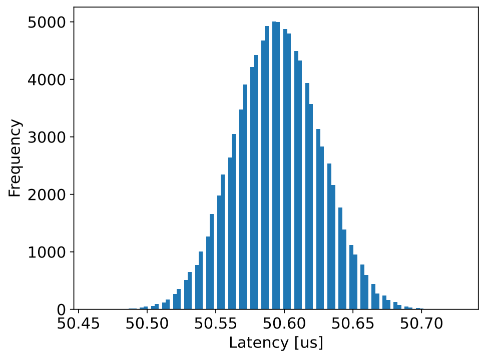


Figure 10 Physical action latency.

delay on the connection between *Processing* and *Actuator*. This means that any output produced by *Processing* will be visible at *Actuator* one *logical* ms later. This, together with the deadline, specifies an interval of 35 μs in which it is acceptable to invoke the *Actuator* reaction. This is a portable way of explicitly specifying bounds on jitter. Notice also that this will coincide with the next evaluation of *Sensor*, so we have, in effect, made a pipeline.

With this configuration, we can execute *Processing* for 885 μs, without missing any deadlines. This yields a utilization of 88.5%. Notice that both the utilization is lower and the deadlines are higher. The reason for this is that the logical delay on the connection is syntactic sugar for a reactor consisting of two reactions and a logical action, which introduces some additional overhead [15].

The execution of this program without any deadline misses means that the runtime claims that timing is not being violated. To verify this, we instrumented hardware timers to generate events on GPIO pins and compared with events generated from within the reaction bodies of *Sensor* and *Actuator*.

The results show offsets of around 28 μs and 35 μs for sensing and actuating, respectively. This is well below their deadlines. *Sensor* is lagging more than *Actuator*, this is expected as their reactions are logically simultaneous. On single-core platforms, simultaneous reactions are ordered based on earliest deadline first scheduling.

5.2.3 Asynchronous events

The ability to react to asynchronous events in a timely manner is crucial for many embedded systems. For instance, a sensor might signal that it has made a new sample by creating a transition on a pin. There are two important timing properties of a reaction to such an event. (1) The precision with which the system *timestamps* the occurrence of the event and (2) *latency* from occurrence to being handled by the system.

Fig. 9 shows the latency between the invokation of an interrupt handler that schedules a physical action and LF's timestamping of the resulting event. The low jitter indicates that a constant offset of around 6.22 μs could be subtracted from the timestamp associated with the physical action.

Fig. 10 shows the latency from asynchronously scheduling a physical action until a reaction triggered by the physical action is invoked. The latency is normally distributed with a mean of 50 μs.

The reported overheads are in addition to the interrupt latency of the underlying platform.

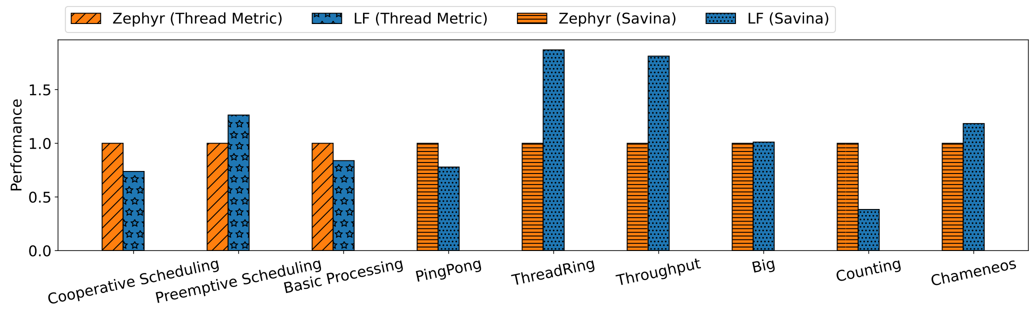


Figure 11 Results of Thread-Metric and Savina benchmark suites for Lingua Franca and a threaded implementation targeting Zephyr kernel API.

5.3 Performance

To assess the general performance of LF on microcontrollers, we run a relevant subset of the Thread-Metric and Savina benchmarks against a baseline implementation using Zephyr's thread APIs. At a fundamental level, LF's scheduling and context-switching overhead is compared to Zephyr's overhead of message-passing and context-switching. This is evaluated for multiple different program topologies and workloads. In the following, we will use the term *actor* to refer to a concurrent component. In the threaded baseline, it corresponds to a thread, and in the LF implementation, it corresponds to a reactor.

5.3.1 Thread-Metric

Thread-Metric [8] is a benchmark for measuring performance of RTOSes. It consists of seven benchmarks that measure overheads associated with different RTOS services like message passing, synchronization, scheduling, interrupt latency and memory allocation. LF does not expose low-level APIs for interrupts, memory allocation and message buffers to the user. In this evaluation, we have thus excluded Message Processing, Synchronization Processing, Interrupt Processing, and Memory Allocation.

The left part of Fig. 11 shows the results of the remaining Thread-Metric benchmarks normalized to the threaded baseline. Performance is measured as the number of iterations through the program during a fixed period. In Cooperative Scheduling, LF achieves 74% of the performance of Zephyr threads.

We measured the cost of a Zephyr thread yield to be 8 μs while LF's reaction-switching and logical time advancement take 4 μs and 29 μs respectively. This means that the LF implementation would need eight low-overhead reaction switches to make up for the relatively expensive time advancement. Cooperative Scheduling only consists of five actors and thus renders LF's performance worse than the baseline.

In the Preemptive Scheduling benchmark, LF outperforms threads by 26%. The reason is the long chain of preemptions occurring when the lowest priority thread executes first. In LF this is not a problem as there is no priority-based scheduling, rather, the topology of the reactor network imposes well-defined restrictions on scheduling. In LF, the preemption is translated into reaction-switching followed by a logical time advancement in each iteration. The threaded baseline must perform twice as many context-switches as reaction-switches in the LF version, due to the priority-based preemption.

Basic Processing only involves a single actor, in a while loop, performing a long-running computation before incrementing a counter. LF is outperformed as it performs logical time advancement after each iteration.

5.3.2 Savina

The Savina benchmark suite is an actor-oriented benchmark suite [7]. The Savina benchmarks focus on compute-intensive rather than IO-intensive applications and put emphasis on message-passing overhead. We evaluate LF on a subset of the Savina benchmarks called the micro benchmarks. The right portion of Fig. 11 shows the result normalized to the Zephyr baseline.

PingPong only consists of two actors sending a message back and forth. The Zephyr implementation uses a message queue [26], which incurs around 8 μs overhead for each communication. In LF the same communication only takes around 6 μs. However, LF advances logical time after each round of back and forth. This takes almost 30μs. In other words, LF performs better when there are long chains of reactions between each logical time advancement. Counting suffers from the same problem. While PingPong has three reactions per logical tag, Counting only has two. Moreover, the threaded implementation of Counting allows for two iterations per context-switch where PingPong requires a context-switch between each ping and pong. This is because we are using a message queue of size 1 which can buffer one element and thus enable two elements communicated per context-switch.

Throughput and ThreadRing are examples at the other extreme. In Throughput, a single producer sends a message to 60 consumers, each performing some floating-point arithmetic on the received data. In LF, this constitutes a chain of 60 reactions per logical time advancement. The Zephyr baseline performs a normal context-switch for each communication. ThreadRing is similar only there is a a chain of consumers rather than the one-to-many topology of Throughput. Again the low reaction-switching overhead of LF enables a 87% and 81% improvement over Zephyr, respectively.

Big is a many-to-many benchmark where each actor sends a ping to a random actor and waits for a response. Due to limitations in RAM we only use 10 actors. This constitutes a chain of reactions just long enough to make up for the expensive logical time advancement.

Chameneos measures how contention on a shared resource affects the performance. In this benchmark, a large number of creatures called chameneos send messages to a single mall, which pairs up two chameneos for a meeting. The Zephyr threading API and the LF runtime both allow the mall to process messages in batches without context-switching between each message. This lets both implementations attain good performance because the mall processes a large volume of messages in comparison to the other actors in the program.

6 Conclusion

We have introduced the embedded target of Lingua Franca, which brings deterministic concurrency and portable specification of timing behaviors to resource-constrained platforms like microcontrollers. The embedded target is built on top of the popular Zephyr RTOS. The runtime is evaluated in terms of memory footprint, timing precision and general performance against baselines using Zephyr's shared memory concurrency primitives directly. We show that the LF runtime adds little overhead and in several cases outperforms the baseline.

References

1 Gul Agha and Carl Hewitt. *Actors: A Conceptual Foundation for Concurrent Object-Oriented Programming*, pages 49–74. MIT Press, Cambridge, MA, USA, 1987.

2 Thomas W. Barr and Scott Rixner. Medusa: Managing concurrency and communication in embedded systems. In *Proceedings of the 2014 USENIX Conference on USENIX Annual Technical Conference*, USENIX ATC'14, pages 439–450, USA, 2014. USENIX Association.

3 Fabrice Bellard. Qemu, a fast and portable dynamic translator. In *USENIX annual technical conference, FREENIX Track*, volume 41, pages 10–5555. Califor-nia, USA, 2005.

4 The Linux Foundation. Zephyr RTOS. `https://zephyrproject.org/`, 2022.

5 Philipp Haller and Martin Odersky. Event-Based Programming Without Inversion of Control. In David E. Lightfoot and Clemens Szyperski, editors, *Modular Programming Languages*, Lecture Notes in Computer Science, pages 4–22. Springer, 2006. `doi:10.1007/11860990_2`.

6 C. A. R. Hoare. Communicating sequential processes. *Commun. ACM*, 21(8):666–677, August 1978. `doi:10.1145/359576.359585`.

7 Shams M. Imam and Vivek Sarkar. Savina – An actor benchmark suite: Enabling empirical evaluation of actor libraries. In *Proceedings of the 4th International Workshop on Programming Based on Actors Agents & Decentralized Control*, pages 67–80, 2014.

8 iSotEE. Thread Metric Test Suite. `https://github.com/iSotEE/thread_metric_test_suite`, 2020.

9 Erling Rennemo Jellum, Torleiv Håland Bryne, Tor Arne Johansen, and Milica Orlandíc. The syncline model–analyzing the impact of time synchronization in sensor fusion. *arXiv preprint*, 2022. `arXiv:2209.01136`.

10 Phil Koopman. A case study of Toyota unintended acceleration and software safety. *Presentation. Sept*, 2014.

11 Karl Koscher, Alexei Czeskis, Franziska Roesner, Shwetak Patel, Tadayoshi Kohno, Stephen Checkoway, Damon McCoy, Brian Kantor, Danny Anderson, Hovav Shacham, and Stefan Savage. Experimental security analysis of a modern automobile. In *2010 IEEE Symposium on Security and Privacy*, pages 447–462, 2010. `doi:10.1109/SP.2010.34`.

12 Edward A. Lee. The problem with threads. *Computer*, 39(5):33–42, 2006.

13 Edward A. Lee and Sanjit A. Seshia. *Introduction to Embedded Systems - A Cyber-Physical Systems Approach*. MIT Press, Cambridge, MA, USA, second edition, 2017. URL: `http://LeeSeshia.org`.

14 Nancy G. Leveson and Clark S. Turner. An investigation of the Therac-25 accidents. *Computer*, 26(7):18–41, 1993.

15 Hendrik Marten Frank Lohstroh. *Reactors: A deterministic model of concurrent computation for reactive systems*. University of California, Berkeley, 2020.

16 Marten Lohstroh, Christian Menard, Soroush Bateni, and Edward A. Lee. Toward a lingua franca for deterministic concurrent systems. *ACM Trans. Embed. Comput. Syst.*, 20(4), May 2021. `doi:10.1145/3448128`.

17 David May. The XMOS architecture and XS1 chips. *IEEE Micro*, 32(6):28–37, 2012.

18 Tommi Mikkonen and Antero Taivalsaari. Web applications – spaghetti code for the 21st century. In *2008 Sixth international conference on software engineering research, management and applications*, pages 319–328. IEEE, 2008.

19 George Robotics. MicroPython – Python for microcontrollers. `micropython.org`, 2022.

20 Abhiroop Sarkar, Bo Joel Svensson, and Mary Sheeran. Synchron – an API and runtime for embedded systems, 2022. `doi:10.48550/arXiv.2205.03262`.

21 Nordic Semiconductor. nRF52 DK. `https://www.nordicsemi.com/Products/Development-hardware/nRF52-DK`, 2022.

22 Douglas Watt. *Programming XC on XMOS devices*. XMOS Limited, 2009.

23 Gordon F. Williams. *Making Things Smart: Easy Embedded JavaScript Programming for Making Everyday Objects into Intelligent Machines*. Maker Media, Inc., 2017.

24 W. Wulf and Mary Shaw. Global variable considered harmful. *SIGPLAN Not.*, 8(2):28–34, February 1973. `doi:10.1145/953353.953355`.

25 Weiwei Xiong, Soyeon Park, Jiaqi Zhang, Yuanyuan Zhou, and Zhiqiang Ma. Ad hoc synchronization considered harmful. In *Proceedings of the 9th USENIX Conference on Operating Systems Design and Implementation*, OSDI'10, pages 163–176, USA, 2010. USENIX Association.

26 Zephyr. Message Queues. `https://docs.zephyrproject.org/3.2.0/kernel/services/data_passing/message_queues.html`, 2022.