Layered Scheduling: Toward Better Real-Time Lingua Franca

Francesco Paladino, Erling Jellum, Efsane Soyer, and Edward A. Lee

***Abstract*—Lingua Franca is a programming paradigm that eases the development of distributed cyber-physical systems and ensures determinism. These systems are subject to stringent timing constraints, generally expressed as task deadlines, and meeting them requires real-time scheduling.**

This work presents a layered scheduling strategy for Lingua Franca for enhanced real-time performance that builds upon any priority-based operating system thread scheduler. The application designers need to specify *only* the application-specific deadlines, and the Lingua Franca runtime automatically converts them into appropriate priority values for the OS scheduler to obtain earliest deadline first scheduling.

***Index Terms*—Lingua Franca, Real-Time Scheduling, Hierarchical Scheduling, EDF, Priority**

I. Introduction

CYBER-PHYSICAL systems have grown in complexity over the years. On multi-core platforms and networked distributed systems, the exchange of messages and their timing massively affects the behavior of these systems, and a careful analysis of the temporal properties is more and more problematic.

Determinism and timing synchronization in the communication offered by new programming paradigms considerably help with the analysis of these systems. One example is Lingua Franca (LF) [1], a coordination language that simplifies the design and development of distributed applications. LF is based on reactive components called *reactors* that encode the logic of the application in any of several popular programming languages, including C/C++, Python, and Rust. Reactors communicate with each other, and LF automatically generates the code for the communication and coordination, thus helping dominate the complexity of modern cyber-physical systems. Reactors react to events that are ordered according to the abstraction of logical time, which enables determinism and synchronization even in distributed settings.

Cyber-physical systems often include periodic or sporadic tasks that read data from the environment, elaborate it, and perform an actuation back on the environment. Because of this interaction with the external world, timing matters. They can be expressed as deadlines, to guarantee that the actuation takes place within a specified amount of time that makes it effective. Missing deadlines might lead to catastrophic effects. Therefore, when designing cyber-physical systems using Lingua Franca, real-time scheduling becomes necessary.

One of the most popular real-time scheduling policies is earliest deadline first (EDF) [2], that privileges tasks having closer absolute deadline to the current time. When running in its multithreaded mode, LF does not have its own thread scheduler; it relies on the underlying OS and its scheduling policies to execute worker threads that execute reactions with ordering constraints that preserve LF semantics.

This paper proposes a layered real-time scheduling strategy that leverages the application-level deadlines specified by the designers. In detail, this layered scheduling approach:

- allows LF to configure and control the underlying OS thread scheduler;
- if the OS platform supports priority-based thread scheduling [2], enables EDF scheduling in LF; and
- hides the OS thread scheduling configuration (e.g., priority values) from the user, who needs to specify *only* the timing requirements of the application, i.e., periods and deadlines. The LF runtime will then automatically convert deadlines into appropriate priority values.

II. Scheduling reactions in Lingua Franca

A Lingua Franca [1] application is made of reactors connected together with ports and connections. Reactors are deterministic actors whose behavior is specified through reactions. Reactions are triggered by discrete events fired at specific time instants. The LF semantics builds upon the abstraction of *logical time* to give an order to the events. In the current version 0.8.0 of the LF runtime, reactions triggered at a given logical time t need to complete their execution before the runtime can process the events at a larger logical time. This *synchronization barrier* makes it easier to achieve deterministic behavior, in the sense that the results of a computation depend only on the data and the time stamps, not on accidents of scheduling. Deadlines can be assigned to reactions and represent timing requirements of the specific application. In the LF context, a deadline defines a physical time, relative to the logical time of the triggering event, by which the corresponding reaction must start executing once triggered.

In the LF scheduler that we build from, reactions that are triggered at a given logical time t are inserted into a *reaction queue*, waiting to be executed by *worker threads*. Worker threads execute one reaction at a time. They are spawned at initialization. The number of worker threads by default is the smaller of the number of cores of the platform and the maximum number of reactions that can execute in parallel.

The dispatch of reactions to worker threads is delegated to the *LF scheduler* and can follow one of two policies:

- first-in, first-out (FIFO): the reactions are served in the same order they are inserted in the reaction queue; or
- earliest deadline first (EDF): the reactions in the queue are served by ascending absolute deadline, i.e., the reactions with closer absolute deadline are served first.

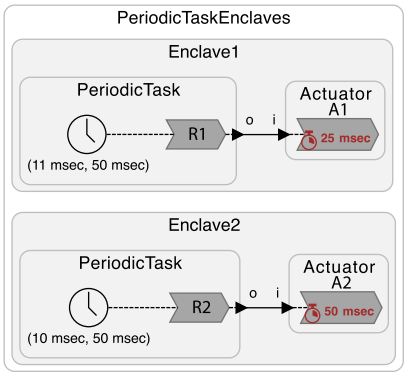

Fig. 1. Example of Lingua Franca real-time application.

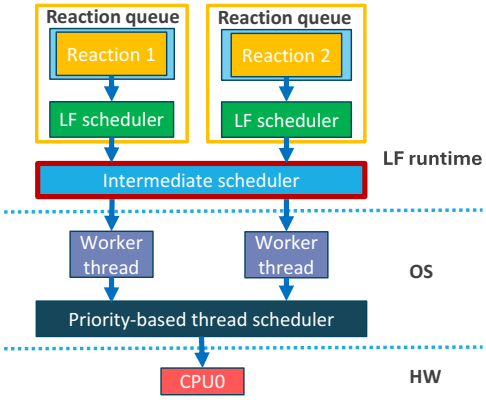

Fig. 2. The layered scheduling approach.

The dispatch policies become relevant when the number of triggered reactions in the reaction queue is greater than the number of available worker threads.

When the reactions triggered at a given logical time have a large execution time, the synchronization barrier might delay the execution of the reactions at a later logical time, because the runtime can advance the time and execute the latter only when all the former complete their execution. LF supports *federates* and (experimentally) *scheduling enclaves*. These are groups of reactors that can advance their logical time more independently of each other. Federates run in separate processes, while enclaves run within the same process. The synchronization barrier is used only with reactors within the same enclave or federate. The LF runtime treats each enclave or federate as a scheduling domain with its own reaction queue, LF scheduler, and set of worker threads. Enclaves and federates communicate with each other, and the runtime provides synchronization mechanisms between them that preserve deterministic semantics. In this paper, we use enclaves rather than federates, but our methods apply to both.

A. Problem Statement

Lingua Franca does not have its own thread scheduler. It relies on the OS scheduler of the platform it runs on to schedule the worker threads serving the reactions. Therefore, once the LF scheduler has dispatched the reactions in the reaction queue to the worker threads, LF has no control over when the worker threads execute those reactions.

Fig. 1 shows a simple test application with two reactors with periodic real-time tasks. Reactions R1 and R2 are triggered every 50ms and drive actuators with deadlines. The two periodic processes start at slightly different times (11ms and 10ms after start time, respectively). Actuator A1 has a deadline of 25ms, while A2 has a deadline of 50ms. Suppose that R1 takes 23ms to execute, while R2 executes for 10ms. In this case, LF cannot meet the deadlines with the synchronization barrier, and even using enclaves, may not meet the deadlines without the layered scheduling proposed here.

In LF, a deadline constrains the *start* time of a reaction. Reaction A1, for example, must begin executing no more than 25ms after a timer event that triggers R1. Hence, R1 has an implied *completion* deadline of 25ms, assuming negligible communication and scheduling overhead.

With the assumed execution times, in principle, it is feasible to meet the deadlines with a single processor core. The first invocation of R2 begins 10ms after the program starts. The first invocation of R1 is enabled (released) 11ms after the start of execution. If R2 is not preempted, however, it will finish execution at 20ms, at which time R1 will begin executing. R1 will complete execution at 43ms, which is past the deadline (11ms plus 25ms, or 36ms). If, on the other hand, R2 is preempted at 11ms, then both deadlines will be met.

In LF, the synchronization barrier prevents R1 from starting its execution at logical time 11ms until R2 has completed execution at logical time 10ms. To correct this, we wrap these two tasks in separate enclaves. Since there is no communication between the tasks, they can execute with separate worker threads and advance their logical time independently. Even then, however, without controlling the OS-level thread scheduler, there is no assurance that the deadlines will be met.

When executing the application of Fig. 1 on Linux, restricting it to run on only one core and selecting the LF EDF dispatch policy, Actuator A1 signals deadline violations.

Because the application contains two enclaves, each enclave has its own LF scheduler and set of worker threads. Therefore, the two reactions are mapped to two different worker threads. Then, the OS thread scheduler schedules them without any notion of timing constraints. The default Linux scheduler (before kernel version 6.6 [3]) is the completely fair scheduler (CFS) [4], which is not a real-time scheduler. LF has no control on the scheduling of the worker threads; even though the LF dispatch policy is set to EDF, the OS schedules with CFS, causing deadline violations. Real-time scheduling then requires LF be aware of the underlying OS thread scheduler.

III. Design of the Layered Scheduler

Because Lingua Franca does not have its own thread scheduler and relies on that of the OS, the scheduling of real-time applications requires the LF runtime to interact with and control the OS thread scheduler.

The approach we follow is to develop an intermediate layer of scheduling in the LF runtime, as shown in Fig. 2, bridging between the LF scheduler and the OS thread scheduler. Having this intermediate layer enables the separation of concerns

between the application-specific requirements and the OS management: the application designer is required to define *only* the timing requirements, i.e., reaction deadlines, while the LF runtime takes care of translating these requirements into an appropriate configuration of the OS thread scheduler. The burden of the OS scheduler settings, e.g., scheduling policy and thread priorities, is transferred to the intermediate layer of scheduling, which operates to meet the application-level timing requirements.

A. Layered scheduling on Linux

Lingua Franca real-time scheduling requires the threads be scheduled by the OS with a real-time policy. The real-time reaction scheduling policy in LF scheduler is EDF, meaning that when two or more reactions are triggered simultaneously and only one worker thread is available, the reaction with the earliest deadline will be dispatched to the worker. Without controlling the scheduling of the worker threads themselves, however, this policy does not achieve global EDF.

Not all general-purpose OSes and RTOSes provide an implementation of an EDF scheduler. Linux, from kernel version 3.14, supports `SCHED_DEADLINE` [5], which is a scheduling class implementing both EDF and a constant bandwidth server (CBS) algorithm [6]. However, this class requires a careful setting of the scheduling parameters for the CBS rules, and an estimate of the worst-case execution time of the tasks, known to be a challenging problem [7].

Given the absence of a suitable EDF reference implementation, we opted to configure the OS thread scheduler with a priority-based policy, which is the simplest and most popular policy for real-time applications. Priority-based schedulers are available in many commercial OSes [8], [9], in addition to Linux, enabling great portability of our approach. The intermediate scheduling layer assigns a priority value to the worker threads to realize a global EDF scheduler.

The scheduler described here was developed for Linux-based OSes using the `SCHED_FIFO` scheduling class. Linux threads under this policy are assigned a positive integer priority value in the range $[1, 99]$, where higher priority values mean greater urgency to be executed. On a hardware platform composed of m cores, this scheduler guarantees that at any time instant t, the m highest-priority threads among those ready for execution are running on the cores. To this end, the scheduler uses preemption to reclaim a core and assign it to a higher-priority task, if necessary.

The intermediate EDF scheduler, using the application-level deadlines defined by the designer, dynamically tunes the priority of the worker threads, which are then scheduled by the OS in such a way that the LF reactions are executed according to EDF. The priority assignment is therefore automatic and completely transparent to the user. Using our layered scheduling, the LF program in Fig. 1 meets all deadlines.

B. Priority assignment

The pseudo-code of Alg. 1 shows how each worker thread interacts with the layered scheduler to obtain a new reaction and set the priority value accordingly. At line 3, the worker calls the LF scheduler API to be assigned a new reaction to execute. At line 5, the intermediate scheduler is invoked to obtain a new priority value that is set at line 6. The new priority depends on the (possibly implied) deadline of the reaction assigned by the LF scheduler. These function calls are executed inside a critical section protected by a mutex because the scheduling decisions are global.

Algorithm 1 Thread body code for new reaction and priority

```
1: function WORKERTHREADBODY
2:     ...
3:     reac ← GETNEXTREACTION()                ▷ LF scheduler
4:     MUTEX_LOCK()
5:     prio ← GETEDFPRIORITY(reac)             ▷ Intermediate scheduler
6:     PTHREAD_SET_PRIORITY(prio)
7:     MUTEX_UNLOCK()
8:     ...
```

When the LF scheduler assigns a new reaction to a worker thread, the intermediate scheduler determines a priority value that the worker will have when it executes that reaction. This value depends on: (i) the deadline of the reaction the worker thread is about to execute; (ii) the deadline of the reactions currently running on the other worker threads.

To do so, the scheduler keeps track of the currently running worker threads in a global data structure containing the deadline of the reaction each is currently executing and the thread priority value. The data structure is sorted by descending absolute deadline. When a worker thread picks a new reaction, the intermediate scheduler creates a new element and inserts it to keep the data structure sorted. Then, it determines a new priority value depending on the position of the element in the data structure: the value shall be between the priority of the element on the left and that of the element on the right. Because priorities are integer numbers, if the left and right elements have consecutive priority values, the intermediate scheduler shifts the priority of some worker threads to make room for the new priority. The shift preserves the relative order of priorities. As soon as the worker thread completes the execution of its reaction, the corresponding element is removed from the data structure.

C. Mutex protocol

As shown in Alg. 1, the invocation of the intermediate scheduler and the setting of the priority are protected by a critical section, because the data structure containing the scheduling information is global across all worker threads. Line 6 might set a lower priority value than what the thread had when it called the scheduler. Therefore, it might be preempted by a higher-priority thread while still owning the mutex of the critical section. If this latter thread tries to acquire the same mutex, it cannot enter the critical section. A medium-priority thread can then preempt it, causing *indirect blocking time*. This problem is known as *priority inversion* [10].

Adopting a *priority inheritance protocol* [11] for the operations on the mutex addresses this issue. When a thread lowers its priority at line 6 and a higher-priority thread preempts it and tries to lock the same mutex, the preempted thread temporarily *inherits* the higher-priority thread's priority to get

back to execution until it exits the critical section. Immediately after releasing the lock, the lower-priority thread restores its nominal priority, and the higher-priority thread runs and enters the critical section. Medium-priority threads cannot preempt.

The `pthread` implementation of the mutexes on Linux supports priority inheritance with the use of the initialization flag `PTHREAD_PRIO_INHERIT`.

IV. Related work

Hierarchical scheduling has been a trending research topic in the real-time systems domain. Deng et al. [12] laid the basis for scheduling real-time applications along with non-real-time ones in open systems, enabling schedulability analysis for the real-time part. The proposed architecture is a two-layer scheduler: the underlying OS assigns a reservation to every application that is handled by a server, and the servers are scheduled with EDF. The real-time applications have their local fixed-priority scheduler. An extension to the above was proposed by Deng and Liu [13] to support aperiodic and sporadic tasks along with periodic ones, and a later work [14] uses a fixed-priority scheduler for the underlying OS.

Having scheduling reservations implies the definition of the server parameters, which might be a challenging problem. Lipari and Bini [15] define a methodology to derive the parameters of each reservation to guarantee schedulability of real-time tasks in a hierarchical scheduling setting with fixed-priority local scheduling.

Further improvements to the analysis of hierarchical scheduling were made. Lorente et al. [16] develop a schedulability analysis technique for components interacting through RPC calls. Lipari and Bini [17] define a framework to simplify the design of hierarchically scheduled open systems with the formalization of the timing requirements of the single applications as *interfaces*. The interfaces were then used for the schedulability analysis of the system.

Hierarchical scheduling fits well with mixed-criticality systems, where critical hard real-time tasks coexist with soft real-time ones, e.g., for infotainment. Schneider et al. [18] deal with deadline-critical and Quality-of-Control-critical tasks and design a two-layer scheduling scheme that assigns a priority value to tasks to guarantee their specific requirements.

Lipari et al. [19] and Inam et al. [20] propose the design of two frameworks for hierarchical scheduling for the real-time OSes SHaRK and FreeRTOS. Abeni et al. [21] design a hierarchical scheduler for virtual machines on a Linux host. The virtual machines are handled with the `kvm` hypervisor and scheduled by the host with the `SCHED_DEADLINE` policy to assign them a reservation. Each virtual machine uses a partitioned fixed-priority local scheduler.

Our work differs from prior hierarchical scheduling techniques because the higher level of the hierarchy is defined by the programming model of Lingua Franca, which is explicit about timing requirements, and the lower level only needs to provide a service to control the priority of worker threads.

V. Conclusion

We have introduced a promising layered scheduling strategy to obtain real-time scheduling of Lingua Franca programs, leveraging the underlying Linux thread scheduler. In the future, this approach may be ported to embedded boards with RTOSes that support priority-based scheduling and a priority inheritance protocol for resource management. Also, an extensive evaluation of the layered scheduling is planned to assess the overhead generated by the priority assignment algorithm.

References

[1] M. Lohstroh, C. Menard *et al.*, "Toward a lingua franca for deterministic concurrent systems," *ACM Trans. Embed. Comput. Syst.*, vol. 20, no. 4, may 2021. [Online]. Available: https://doi.org/10.1145/3448128

[2] C. L. Liu and J. W. Layland, "Scheduling algorithms for multiprogramming in a hard-real-time environment," *J. ACM*, vol. 20, no. 1, p. 46–61, jan 1973. [Online]. Available: https://doi.org/10.1145/321738.321743

[3] J. Corbet. (2023) An EEVDF CPU scheduler for Linux. [Online]. Available: https://lwn.net/Articles/925371/

[4] C. S. Pabla, "Completely fair scheduler," *Linux J.*, vol. 2009, no. 184, aug 2009.

[5] D. Faggioli, F. Checconi *et al.*, "An EDF scheduling class for the Linux kernel," in *11th Real-Time Linux Workshop (RTLWS)*, 2009.

[6] L. Abeni and G. Buttazzo, "Integrating multimedia applications in hard real-time systems," in *Proceedings 19th IEEE Real-Time Systems Symposium*, 1998, pp. 4–13.

[7] R. Wilhelm, J. Engblom *et al.*, "The worst-case execution-time problem—overview of methods and survey of tools," *ACM Trans. Embed. Comput. Syst.*, vol. 7, no. 3, may 2008. [Online]. Available: https://doi.org/10.1145/1347375.1347389

[8] Amazon Web Services, Inc. FreeRTOS - Market leading RTOS (Real Time Operating System) for embedded systems with Internet of Things extensions. [Online]. Available: https://www.freertos.org/

[9] Zephyr Project. Zephyr Project – A proven RTOS ecosystem, by developers, for developers. [Online]. Available: https://zephyrproject.org/

[10] B. W. Lampson and D. D. Redell, "Experience with processes and monitors in mesa," *Commun. ACM*, vol. 23, no. 2, p. 105–117, feb 1980. [Online]. Available: https://doi.org/10.1145/358818.358824

[11] L. Sha, R. Rajkumar, and J. Lehoczky, "Priority inheritance protocols: an approach to real-time synchronization," *IEEE Transactions on Computers*, vol. 39, no. 9, pp. 1175–1185, 1990.

[12] Z. Deng, J.-S. Liu, and J. Sun, "A scheme for scheduling hard real-time applications in open system environment," in *Proceedings Ninth Euromicro Workshop on Real Time Systems*, 1997, pp. 191–199.

[13] Z. Deng and J.-S. Liu, "Scheduling real-time applications in an open environment," in *Proceedings Real-Time Systems Symposium*, 1997, pp. 308–319.

[14] T.-W. Kuo and C.-H. Li, "A fixed-priority-driven open environment for real-time applications," in *Proceedings 20th IEEE Real-Time Systems Symposium*, 1999, pp. 256–267.

[15] G. Lipari and E. Bini, "A methodology for designing hierarchical scheduling systems," *J. Embedded Comput.*, vol. 1, no. 2, p. 257–269, apr 2005.

[16] J. Lorente, G. Lipari, and E. Bini, "A hierarchical scheduling model for component-based real-time systems," in *Proceedings 20th IEEE International Parallel & Distributed Processing Symposium*, 2006, pp. 8 pp.–.

[17] G. Lipari and E. Bini, "A framework for hierarchical scheduling on multiprocessors: From application requirements to run-time allocation," in *2010 31st IEEE Real-Time Systems Symposium*, 2010, pp. 249–258.

[18] R. Schneider, D. Goswami *et al.*, "Multi-layered scheduling of mixed-criticality cyber-physical systems," *Journal of Systems Architecture*, vol. 59, no. 10, Part D, pp. 1215–1230, 2013.

[19] G. Lipari, P. Gai *et al.*, "A hierarchical framework for component-based real-time systems," *Electronic Notes in Theoretical Computer Science*, vol. 116, pp. 253–266, 2005, proceedings of the International Workshop on Test and Analysis of Component Based Systems (TACoS 2004).

[20] R. Inam, J. Mäki-Turja *et al.*, "Hard real-time support for hierarchical scheduling in freertos," in *Euromicro Conference on Real-Time Systems*, 2011.

[21] L. Abeni, A. Biondi, and E. Bini, "Hierarchical scheduling of real-time tasks over linux-based virtual machines," *Journal of Systems and Software*, vol. 149, pp. 234–249, 2019.