Timing enclaves for performance in Lingua Franca

Julian Robledo[§]
TU Dresden
Dresden, Germany
julian.robledo@tu-dresden.de

Christian Menard[§]
TU Dresden
Dresden, Germany
christian.menard@tu-dresden.de

Erling Jellum
NTNU
Trondheim, Norway
erling.jellum@gmail.com

Edward A. Lee
UC Berkeley
Berkeley, USA
eal@berkeley.edu

Jeronimo Castrillon
TU Dresden
Dresden, Germany
jeronimo.castrillon@tu-dresden.de

***Abstract*—The reactor model is a model of computation for concurrent systems that includes semantics for time to guarantee deterministic execution of events. However, the guarantee of determinism comes at the price of raising the complexity of building a runtime scheduling algorithm that efficiently exploit parallelism of real time systems. In this paper we propose a methodology called "timing enclaves" for partitioning of reactor programs written using Lingua Franca, a novel coordination language that implements the reactor model. Timing enclaves decouple the timeline of an application to use multiple schedulers that allow parallel computation while preserving determinism. We evaluate our approach on a baseband processing benchmark, a complex use case with a high degree of parallelism and real-time constraints. We show that our approach has performance comparable to a prior asynchronous and nondeterministic implementation while ensuring determinism.**

***Index Terms*—Baseband processing, models of computation, reactors.**

I. Introduction

Cyber physical systems (CPS) integrate modern computing and network technologies with physical subsystems to enable automotive applications, industrial automation, and smart medicine. They pose a number of challenges regarding concurrency, safety, and scalability. The physical component of a CPS system imposes timing constraints not present in pure information processing systems. The cyber component uses programming abstractions that typically do not include timing. However, since computation takes time, and it interacts with the physical component, an effective model of computation (MoC) for CPS should include a notion of time.

The reactor model [1] is a reactive MoC for CPS that provides timing semantics and deterministic concurrency. It is based on discrete-event systems where concurrent objects called reactors communicate via timestamped events. The semantics define a partial order for handling events. Developing runtime strategies for such systems presents two key challenges: how to efficiently exploit the inherent parallelism for multi-core execution, and how to ensure time-predictable execution of real-time tasks. The Lingua Franca (LF) coordination language is a recent framework that implements the reactor model [2]. It provides a compiler and a runtime system. The current LF runtime handles all events in timestamp order, where only logically simultaneous events can be executed in parallel, and only if there are no dependencies between them. Dependencies form a graph, and to simplify the dependency analysis at runtime, a topological sort is performed at startup. This approach, while efficient, inserts implicit timing barriers that limit the amount of exploitable parallelism. Moreover, timing barriers can introduce priority inversion because a high-priority task may wait for an unrelated low-priority task handling an earlier event.

LF has been tested on various benchmarks, showing that a wide range of applications can be implemented deterministically without compromising performance [3]. However, the benchmarks have a regular structure and homogeneous execution times in relation to input data. Performance on real applications may suffer compared to direct implementations that ignore timing and determinism, as we will show.

In this paper we extend LF with *timing enclaves*, a mechanism that partitions LF programs into decoupled timelines. Each partition is endowed with its own scheduler, reducing the amount of implicit timing barriers and thus improving parallelism. To achieve this, we propose coordination mechanisms between timing enclaves for concurrent execution that preserves determinism.

To understand how enclaves can improve the performance of reactor programs, we first use synthetic examples that abstract from common concurrency patterns. We then test the technique on an open-source implementation of 4G baseband processing system [4] that we port to LF. This is a real-life use case with stringent real-time constraints that allows us to evaluate strategies for using enclaves in a complex application.

The rest of the paper is structured as follows. Section II introduces the reactor model and discusses prior runtime management mechanisms. Section III shows limitations of the current solutions and describes timing enclaves as a method to exploit more parallelism in LF programs. Section IV introduces the 4G baseband processing benchmark. We evaluate our methodology in Section V followed by a discussion of related work in Section VI. Finally, conclusions and future work are discussed in Section VII.

II. Background

This section provides an overview of the reactor model and presents illustrative examples. We only focus on the

[§]The authors contributed equally to this paper.

979-8-3315-0457-1/24/$31.00 ©2024 IEEE

model elements that are relevant to the problems that we address. A more complete description of the model is given by Lohstroh [5].

A. The reactor model

The reactor model is an MoC where concurrent components called reactors contain the following elements: reactions, ports, connections, state variables, logical actions and physical actions. A reaction defines a behavior that is triggered by events. Reactions belonging to the same reactor are mutually exclusive; they cannot be executed in parallel. Reactors can also contain other reactors, which allows composing programs in a hierarchical way. Logical actions are triggers produced by reactions that schedule future events. Physical actions are similar, but they are scheduled asynchronously from a context outside the reactor program. The ports and connections allow a reactor to send or receive events to or from another reactor.

Fig. 1 shows a reactor program with four reactors B, C, D, and a top-level reactor A. Logical actions are represented as triangles with "L" and physical actions with "P". The clock symbol represents a timer. A timer, like the one in reactor B, is a trigger that produces events with a fixed period. The circle and diamond symbols denote a startup and shutdown triggers, respectively. These trigger reactions that are called when the the program starts and terminates, respectively. The dark grey chevron symbols represent reactions, which contain imperative code in a target language (C, C++, Python, Rust, or TypeScript, currently). The plain arrows represent port to port connections while the dashed arrows are used to connect all ports and actions that a reaction may have an effect on and vice versa.

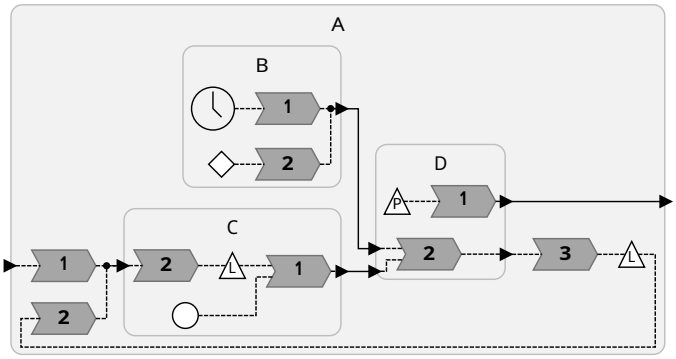

Fig. 1: An example reactor program.

Therefore, the reactor model uses the concept of logical time that is used to order events on a logical timeline [6]. The reactor model assigns timestamps that impose a well-defined order on events. Lingua Franca uses generalized superdense representation of time [7]. Logical time is denoted by a tag $g = (t, m)$, where t is a time value, an integer representing elapsed nanoseconds. The microstep m imposes an order on events with the same timestamp. Tags with same time value but different microstep are not considered logically simultaneous. In the reactor model logical time does not increase during execution of a reaction. Physical time, on the other hand, does elapse. Logical actions allow to schedule events with a delay $d \in \mathbb{Z}^+$ relative to the current tag $g = (t, m)$; the tag of the new event is defined by the tag delay function $\mathcal{D}$, as follows:

$$\mathcal{D}(t, m, d) = \begin{cases} (t, m+1) & \text{for } d = 0 \\ (t+d, 0) & \text{for } d > 0\,. \end{cases}$$

Ports and connections impose ordering constraints on execution of logically simultaneous reactions. The constraints form an acyclic precedence graph (APG). Actions do not imply a dependency because scheduling an action creates a future event with a tag strictly greater than the current tag. Fig. 2 shows the APG for the reactor program in Fig. 1. Solid arrows represent connections while dashed arrows represent the priority order for reactions within the same reactor.

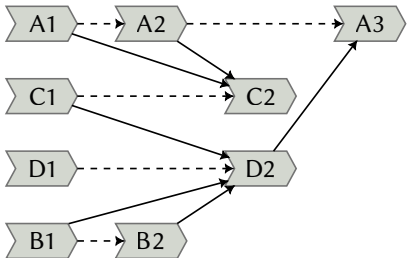

Fig. 2: Dependency graph of the example reactor program in Fig. 1.

A scheduler for a reactor program only processes an event with tag g if the current physical time is greater than the timestamp of g. This induces an implicit physical time barrier. The prior scheduler makes sure that all reactions in the current logical time finish execution before advancing the logical time to the next tag. However, it is possible that a computationally heavy reaction finishes at a physical time that is much greater than the logical time of the next tag. In this case, we say that the logical time lags behind the physical tag.

Reactions may be annotated with a deadline and a deadline handler. When a reaction is invoked, the runtime compares the current physical time with the timestamp of the current tag. If the difference is equal or greater to the annotated deadline, the deadline handler is called instead of the reaction body. This allows to easily detect and handle deadline violations.

B. Lingua Franca and its scheduler

The Lingua Franca (LF) coordination language implements the reactor model. In LF, the execution of a reactor program is managed by a runtime scheduler. The scheduler keeps track of scheduled future events, controls advancement of logical time, and invokes any triggered reaction.

To ensure determinism, the current LF runtime implementation uses a conservative approach that assigns levels to reactions in the APG and allows parallel execution only for reactions with the same level. An alternative would be to use the APG at runtime to identify when a reaction is ready for execution. The scheduler could walk the APG checking whether all dependent reactions are either not triggered or have completed executing. For large applications, however, traversing the graph can be computationally expensive. The conservative approach assigns a level to each reaction, where the level is the length of the longest path from any root of the graph to the reaction. For the example, in Fig. 2, the reactions A1, C1, D1 and B1 will be at level 1, reactions A2, C2, D2

and B2 at level 2, and A3 at level 3. If two reactions have the same level, they cannot have any dependencies on each other and can safely be executed in parallel.

III. Timing Enclaves

In this section, we use synthetic examples to show that it is possible to find better solutions that efficiently exploit parallelism without incurring a large overhead.

A. Problem analysis: The tag barrier

Consider the program in Fig. 3. In this program, new data is generated in the source actor every 10 ms and passed to a pipeline of reactors. Fig. 4 shows the resulting APG for this program. Given the connection between reactors, every reactor is assigned a different level, which means that the execution is strictly sequential.

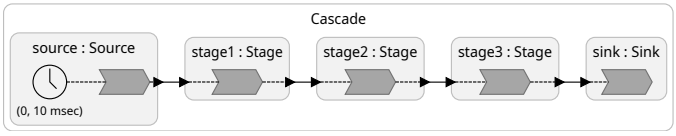

Fig. 3: A simple pipeline of reactors.

Fig. 4: APG for the simple pipeline of reactors.

Fig. 5 shows a timing diagram for the application in Fig. 3, with the physical time in the x-axis and logical time in the y-axis. Because logical time does not elapse as a reaction executes, reactions are logically instantaneous. The diagonal line represents the physical time barrier imposed by the model, where events are only scheduled if the physical time is equal or greater than the logical time. Ideally, it should be possible to exploit pipeline parallelism. However, the scheduler only advances to the next tag once all reactions at the current tag have been executed. In contrast, Fig. 6 shows an execution that exploits available parallelism.

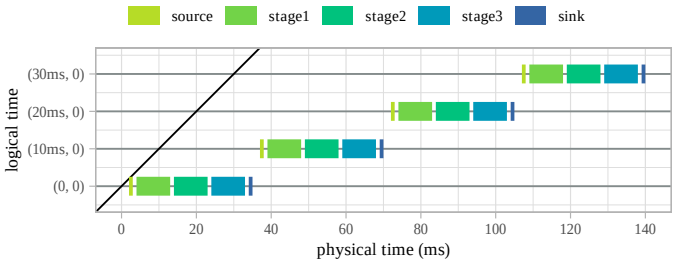

Fig. 5: Timing diagram for a simple pipeline of reactors with LF scheduler.

One possible solution to increase the parallelism of the example in Fig. 3 is to leverage LF's timing semantics to add logical delays between the stages as shown in Fig. 7. Adding a delay to a connection schedules a future event with a given offset. For example, the reaction in `stage1` will be scheduled with an offset of 10 ms in logical time after the tag of reaction in `source`. This solution breaks the dependency of reactions in the APG, leading to all reactions sharing the same level. In this way all reactions can execute in parallel. However, this approach requires that delays between stages are chosen carefully to always match the production rate, which is not always easy and impossible at times.

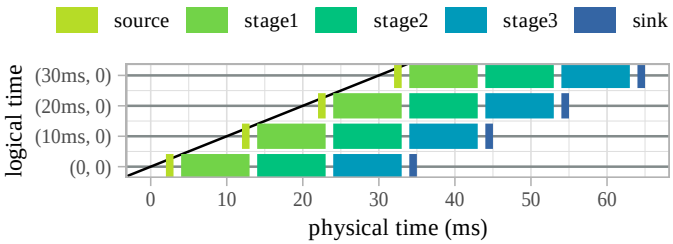

Fig. 6: Ideal timing diagram for a simple pipeline of reactors.

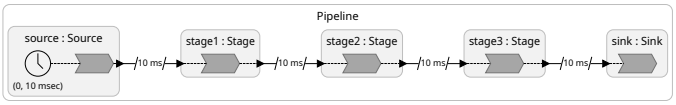

Fig. 7: A simple pipeline with delays in between stages.

B. Problem analysis: The level barrier

Consider now the program in Fig. 8. Here, the source reactor distributes the generated data to two parallel computing paths. The upper path, (`stage1` and `stage3`), does not have any data dependency with the lower path, (`stage2` and `stage4`), and thus the paths could execute in parallel. Due to the level-based scheduling approach, the runtime would schedule `stage1` in parallel to `stage2` because they have the same level. However, the scheduler will proceed to execute `stage3` and `stage4` in parallel only after *both* `stage1` and `stage2` have finished the execution. This works well as long as the stages have a similar execution times and the entire execution can keep up with the physical time barrier.

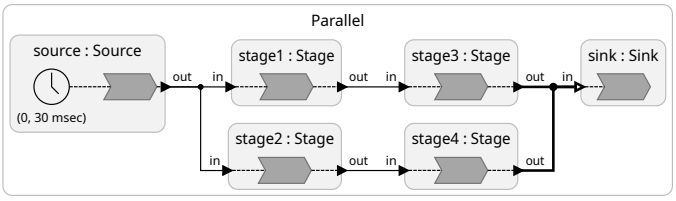

Fig. 8: An LF program with two parallel compute paths.

In Fig. 9 we see how variability in execution times creates gaps in the schedule. The example shows that, with variable execution times, the level-based scheduler may lead to interference between reactions that do not depend on each other, and the logical time might lag behind the physical time as a consequence. Fig. 10 shows an ideal timing diagram for the same program without scheduling gaps. Also here we could consider adding delays between stages, in a similar way to the example in Fig. 7. However, it would be inefficient because it imposes a static delay to a variable execution time.

A possible solution would be to replace the level-based scheduler and the tag barrier with more elaborated topological

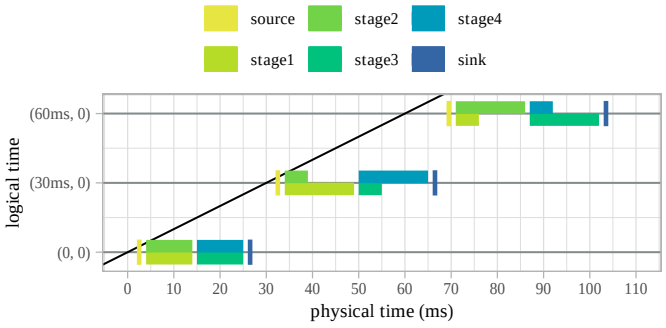

Fig. 9: Timing diagram for the program in Fig. 8.

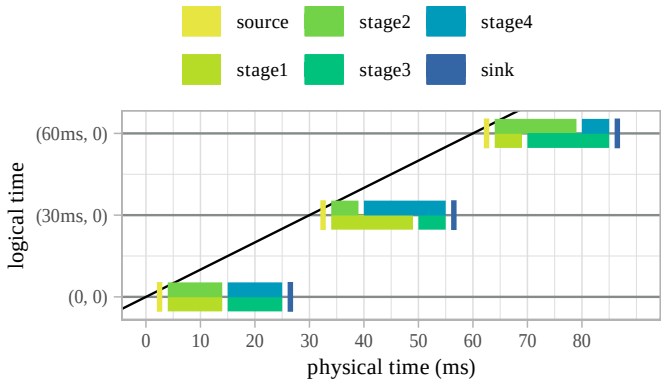

Fig. 10: Ideal timing diagram for the program in Fig. 8.

sorting algorithms that are able to execute reactors with different tags in parallel and are capable of identifying reactions that are ready for execution without relying on levels. However, deriving such solutions is challenging [5]. On the other hand, having a-priori knowledge of the application might be useful to propose tailor-made strategies to traverse the APG efficiently. However, this would add an extra burden to programmers as this requires time debugging and a deep understanding of the application. Moreover, the resulting strategy would be useful for a family of problems and not for the general case.

C. Timing enclaves

Rather than devising new scheduling and graph traversal algorithms, we propose to partition the program and endow each partition with a scheduler using the tag and level barrier approach. We call these partitions *timing enclaves* because their timelines are decoupled; i.e., each enclave keeps track of its own logical time. This decoupling makes it possible to execute more reactions in parallel, because timing barriers become local to each enclave.

Partitioning is a common practice across programming models. In the particular case of the reactor model, care must be taken to ensure that a partitioned LF program preserves the semantics of the original one. In particular, mechanisms are needed to retain a time-deterministic execution by implementing signaling between timing enclaves. This signaling coordinates the advancement of logical time in the different enclaves.

We have explained how the physical time barrier is used to control the advancement of logical time. In the LF runtime,

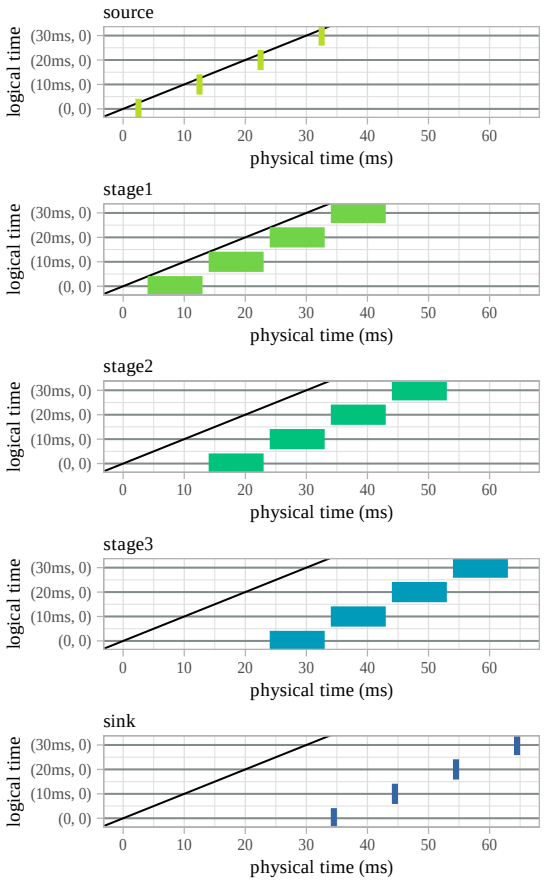

Fig. 11: Timing diagram for the example in Fig. 3 when using enclaves.

the scheduler has to wait for all reactions at a current tag to finish execution before advancing the logical time. We follow a similar approach to synchronize across timing enclaves by introducing a logical time barrier for entire enclaves. More precisely, a downstream enclave waits for upstream enclaves to release a tag before locally processing the reactions at that tag. By releasing a tag, an upstream enclave signals that it will only produce new events with a later tag. To implement this, before processing a tag, the scheduler of a timing enclave will block, awaiting a tag release from all external inputs, then process the tag locally, and finally release the tag on all outgoing connections. Fig. 11 shows the decoupled timing diagrams of the pipeline shown in Fig. 3 when using enclaves.

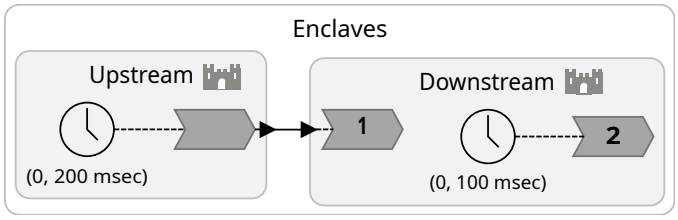

Fig. 12: An example program with two enclaves.

While the methodology described above enforces a deterministic execution, it may still lead to a significant lag on downstream enclaves. Consider the example in Fig. 12, which

contains two reactors. The castle symbol indicates that both reactors are enclaves. Reactors Upstream and Downstream have a timer that triggers at intervals of 200 ms and 100 ms, respectively. Every time the upstream enclave processes a reaction, it notifies the outgoing connection to release the tag on the logical time barrier, which in turn is acquired by the downstream enclave. Fig. 13 shows the timing diagram for this example. The figure includes vertical lines to mark the physical times at which Downstream acquires a tag and Upstream releases a tag. (These vertical lines carry no meaning on the axis of logical time.) The figure also shows the physical time at which the reactions are triggered. `Downstream 1` and `Downstream 2` refer to the execution of the reactions 1 and 2 of the Downstream enclave, respectively. As can be seen, the reaction triggered by Downstream's timer is executed with a significant lag at tags $(100ms, 0)$ and $(300ms, 0)$. This is because the upstream enclave only releases a tag after it has completed processing it. Thus, the downstream enclave does not receive any notifications for tags for which there is no event scheduled in the upstream enclave. Because Upstream has no event scheduled at $(100ms, 0)$, Downstream has to wait until it receives the release for the tag $(200ms, 0)$ before it can process the tag $(100ms, 0)$.

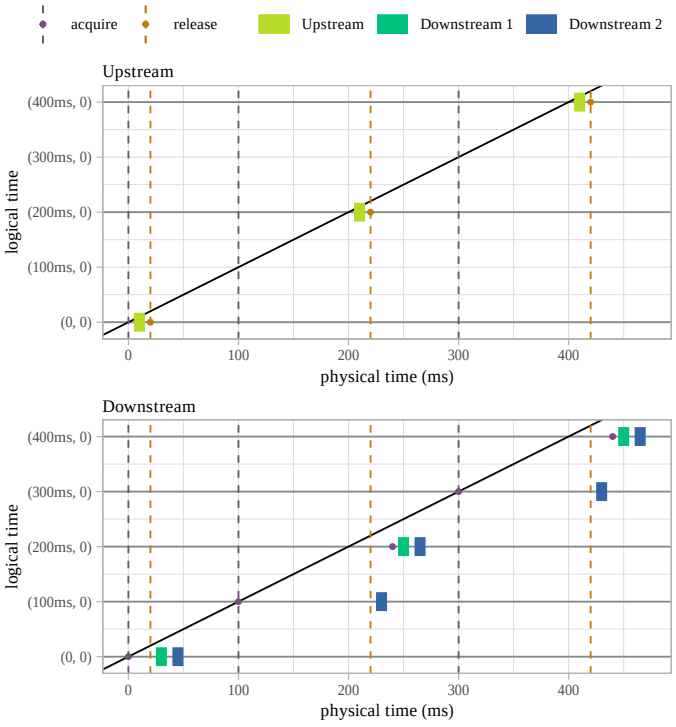

Fig. 13: Timing diagram for the example in Fig. 12 using a naive coordination scheme.

To avoid downstream enclaves waiting indefinitely for a release at a given tag, the downstream enclave notifies the upstream enclave of its request to acquire a certain tag. In this case, the upstream enclave inserts an empty event in its local event queue if there is not already an event scheduled for this tag. An empty event denotes an entry in the event queue at a specific tag without any associated reactions. Upstream will acquire the tag from all its connections and, since the event is empty and no reactions are triggered, immediately release the tag. This protocol ensures both determinism and timely execution. Fig. 14 shows the timing diagram of the previous example with the described methodology.

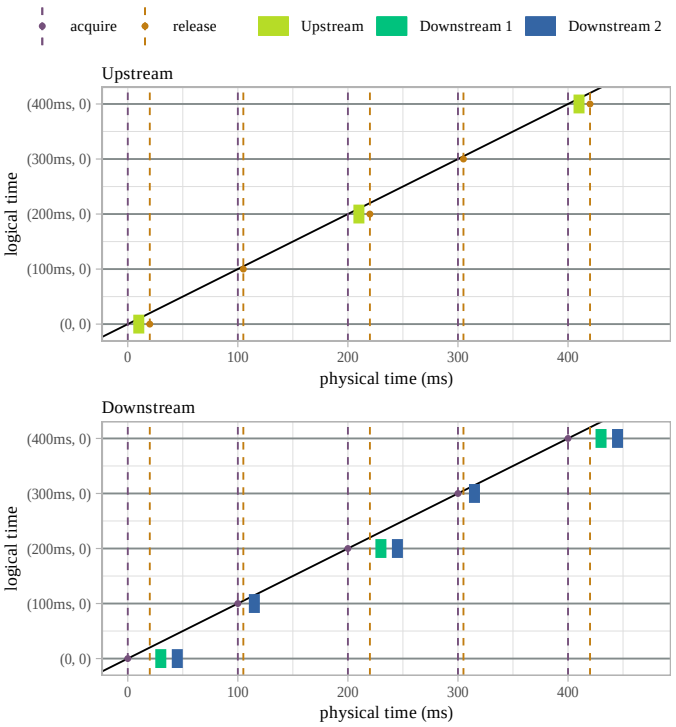

Fig. 14: Timing diagram for the example in Fig. 12 using the complete coordination scheme with empty upstream events.

The presented approach is useful to overcome the constraints imposed by the tag barrier and level barrier for a large set of use cases, especially for programs with an acyclic structure. Programs with cycles, however, remain problematic. Let us assume an application with 2 enclaves, A and B, interconnected in a zero-delay cycle; i.e. there is a connection from a A to B, and one from B to A, both without delays. In this case, as A and B have separate schedulers, A will try to acquire the startup tag from B, which in turn would acquire the same tag for A. The execution immediately deadlocks. A solution is to break the zero-delay cycle by adding a logical delay d on the connection from B to A. This avoids the deadlock, but suffers from poor performance and changes the logic of the program. If A wants to handle an event at tag t it must acquire the tag $t - d$ at B. But for B to release tag $t - d$ it must first acquire it at A, which in turn must acquire $t - 2d$ from B. This goes on until they reach the previous completed tag. Future work will focus on extending the current methodology to make the program aware of the presence of cycles and develop strategies to handle them. Fortunately, there are many applications without cycles, including the use case studied next, that can benefit from timing enclaves.

IV. Use case: Baseband processing

As discussed in Section I, we use baseband processing in 4G/5G networks as a strong use case to demonstrate our timing enclaves and how different partitions impact performance. This

case study contributes a software-based implementation, as required by the current trend towards virtual radio access networks (VRAN) [8]. Moreover, our model-based approach responds to challenges identified by Wittig et al. [9] to cope with the highly time-varying workloads and stringent time-constraints of 4G/5G networks. In this section, we explain the main relevant features of the use case and briefly discuss our implementation in LF.

A. 4/5G in a nutshell

Baseband processing for modern cellular networks is characterized by complex signal processing algorithms. The latest standards add on top highly heterogeneous traffic, including enhanced mobile broadband (eMBB), ultra-reliable and low-latency communication (URLLC), and massive machine-type communication (mMTC) [10].

In uplink communication, a base station allocates a frequency band to every user equipment (UE). Frequency is always allocated in blocks of a basic resource unit known as a physical resource block (PRB). Given the spectrum flexibility of 5G, The allocated frequency band might change according to the user needs. Every millisecond a new set of UE requests is received at the base station in a *subframe*.The multiple-input and multiple-output (MIMO) [11] technique is used to increase the data rate. It allows the UEs to transmit independent streams of information, also known as layers, to increase the channel capacity. Moreover, depending on the signal quality, the UEs might apply different modulation schemes to improve the spectral efficiency. Baseband processing is therefore a demanding task where the base station has to recover multiple independent streams of data coming from different UEs and apply different decoding algorithms to each of them.

Additionally, according to the current standard, the deadline for processing a subframe is 2.5 milliseconds [12].If the subframe processing time exceeds this threshold, it will be discarded, resulting in packet loss. Such loss compromises the ability to offer reliable quality of service that is essential for latency-critical applications.

B. Reference implementation

We use the open-source LTE PHY benchmark [4] as a reference for our work. The implementation provides a realistic model of a base station that exposes the parallelism of the algorithms and captures the dynamic behavior of mobile workloads. Fig. 15 shows the cascade of signal processing kernels included in the benchmark. The PHY benchmark is modular, such that kernels can be replaced to investigate new algorithms and optimization techniques. The execution of concurrent kernels is parallelized using a customizable number of worker threads and a work-stealing runtime [13]. The model can be parallelized over the number of users and, for the highlighted tasks in Fig. 15, within a single user. The degree of exploitable parallelism depends on a set of dynamic and static parameters like the number of antennas in the base station, which is a fixed number in every base station, and the number of layers sent by a UE through MIMO, which depends on the specific UE device.

The execution of the LTE PHY benchmark proceeds as follows: At every subframe (i.e. every 1 millisecond), a new

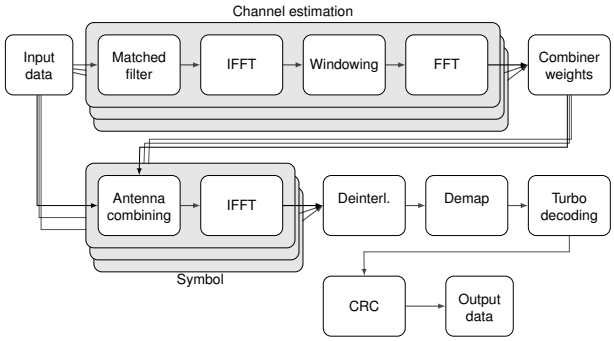

Fig. 15: Block diagram of the LTE PHY benchmark baseband receiver for a single UE request.

set of randomly parameterized UE requests is generated to emulate the incoming workload. The requests are organized in a user queue. Idle worker threads look in the user queue and take the next UE to be processed. Every time a UE is dequeued, the respective thread fills a local task queue with the processing kernels while respecting the pipeline order. If the user queue is empty, the idle threads will look into the progress of the other threads and will steal tasks that can be executed in parallel. The complexity of real-time requirements as well as the heterogeneity of the workloads, makes this use case well-suited to stress our timing enclaves design.

C. The LF implementation

We ported the PHY benchmark to LF, as shown in Fig. 16. Our LF implementation has a customizable number of parallel reactors called user managers. Every user manager contains a chain of reactors, where every reactor represents a kernel of the PHY benchmark. Additionally, we include a workload dispatcher reactor that simply takes the input data and distribute it to a set of parallel kernels. The workload manager generates new user requests every 1 ms and sends them to the user managers in a round-robin fashion. Although, the original PHY benchmark is written in C language, we used the C++ target of LF, as it is designed for efficiently exploiting parallel hardware [3]. We compile the kernels of the benchmark with a C compiler and link them from the C++ code.

V. Execution analysis

We have described timing enclaves as a mechanism for partitioning reactive systems. However, finding the best partition for a program is a nontrivial task, especially considering the complexity of the baseband use case. In this section, we leverage our knowledge of the baseband processing use case and propose multiple partition strategies that help expose the trade-off space exposed by timing enclaves.

The pipeline parallelism of the LTE benchmark would lead to problems similar to the example in Fig. 8, where parallel pipelines will be stalled due to long execution time in one of the reactions, even if they do not have data dependencies. Notice that given the large number of reactors, it is not possible to explore the whole search space. Instead, we propose a set of intuitive solutions that can give us a hint of the potential gain of a given partitioning strategy.

- `enclaves1`: The first proposed solution is to decouple the timeline between pipelines and make every pipeline

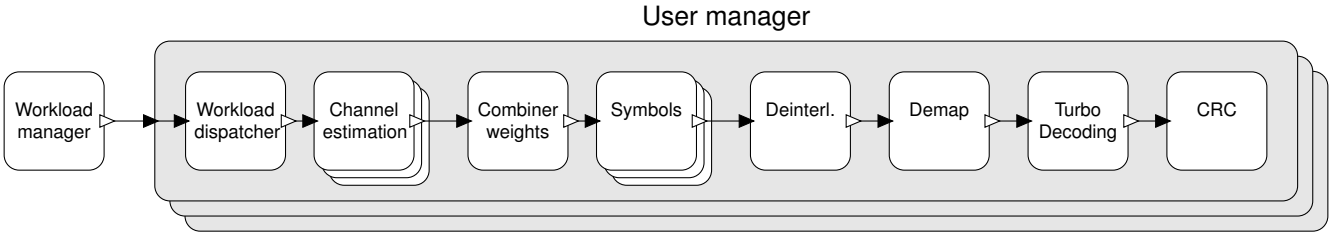

Fig. 16: Reactor implementation of the PHY benchmark.

(or user manager) an enclave. Notice that we fixed the number of user managers to 10, which corresponds to the maximum number of user requests per subframe according to the LTE standard.

- `enclaves2`: Implementation where one enclave is shared by every pair of user managers. This version contains half of the enclaves compared to `enclaves1`, but every enclave handles double the workload. The idea is to evaluate whether the overhead introduced by the coordination mechanism between enclaves is significant compared to the potential gain of timeline decoupling.
- `enclaves3`: As every user manager contains eight different types of kernels, some of them with multiple parallel instances. This variant breaks every pipeline into eight enclaves, with every kernel of the same type in one enclave.
- `enclaves4`: We leverage some prior knowledge about the application and identify the computationally heaviest kernels. We define an implementation where we split every pipeline into 4 stages in such a way that the execution time on each of the stages is similar. The first stage includes the workload dispatcher and channel estimation reactors. It is followed by a stage with combiner weights, which is the most intense kernel, as the only reactor. The third enclave includes all the parallel symbols reactors. Finally, we combine deinterleave, demap, turbo decoding, and CRC reactors into a single enclave.
- `enclaves5`: We generate a set of enclaves that are shared across multiple user managers. All kernels of the same type for all user managers are grouped in the same enclave, i.e. all workload dispatchers will be placed in the same enclave.
- `enclaves6`: Finally, given that there is also parallelism within the pipelines, we evaluate a version where every reactor is an enclave. By declaring every reactor as an enclave, we maximize the decoupling of components because reactors operate independently from each other. This leads to the highest amount of potential parallelism.

Table I summarizes the proposed implementations with their corresponding number of enclaves. The table also includes the original LTE PHY benchmark, `PHY`, and the implementation in LF without enclaves, `reactors`. Both of them can be seen as if they have one enclave because they have a single timeline.

For the evaluation, we generate workloads (traces of subframes) according to realistic traffic profiles. Concretely, we use a profile from a dense urban area during a day, using the methodology in [14].We generate 10,000 subframes, equivalent to 10 seconds, of traffic traces for low, medium, and high traffic, which correspond to the lowest, mid, and highest traffic hours, respectively. A realistic traffic scenario presents a heterogeneous workload where some of the subframes are empty, while other subframes might contain a heavy load of multiple users, each of them characterized by a different set of baseband parameters like the number of PRBs and modulation scheme. For evaluation, we use an Intel(R) Core(TM) i7-6700 CPU with 4 cores and 8 total treads, at 3.40GHz working frequency.

TABLE I: Number of enclaves for each implementation.

Version	Number of enclaves
PHY	1
enclaves1	10
enclaves2	5
enclaves3	80
enclaves4	40
enclaves5	8
enclaves6	460
reactors	1

Fig. 17 shows the number of missed deadlines for the different implementations for the described traffic scenarios. The plot shows that `enclaves6` gives the worst performance out of all implemented versions, including the version without enclaves. Making every reactor an enclave comes at the price of adding significant overhead for coordinating the individual reactors. `enclaves1` gives the best performance out of all LF implementations and it performs also better then `PHY` for a low-traffic scenario. `PHY` on the other hand, presents a lower number of missed deadlines for medium and high traffic.

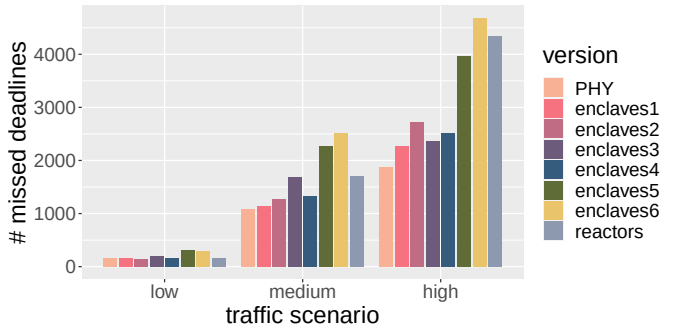

Fig. 17: Data traffic generated in a base station during a day.

Finally, we leverage the reactor model's timed semantics to add deadline management. By adding deadline management, we can drop jobs early that do not meet the

deadline, which frees up resources for the other jobs. For this, we take the implementation with reactors with the best performance (`enclaves1`) and generate a new version with deadline management `enclaves1_deadline`. Fig. 18 shows the number of missed deadlines for `PHY`, `enclaves1`, `enclaves1_deadline`, and `reactors`. The `enclaves1_deadline` version presents a lower number of missed deadlines for low and medium-traffic scenarios.

This evaluation shows how enclaves can help bridge the performance gap between a non-deterministic implementation (`PHY`) and a deterministic one (`reactors`) for a challenging, time-varying workload. It also exposes the trade-off space opened up by different timing enclaves. Exploring this automatically is an interesting avenue for future work.

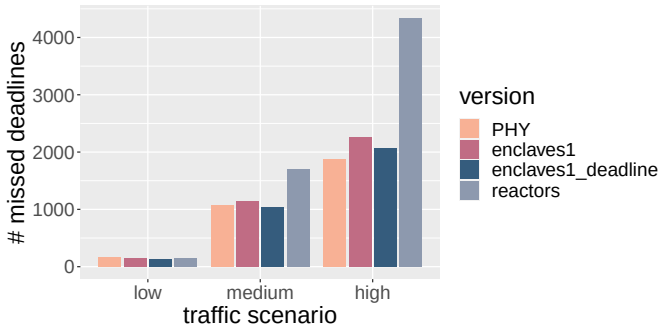

Fig. 18: Data traffic generated in a base station during a day.

VI. Related work

Dataflow MoCs like synchronous data flow (SDF) [15] or Kahn process networks (KPNs) [16] have been used successfully to express concurrent computation in telecommunication applications. Both models ensure a deterministic execution while exploiting pipeline and data parallelism. Design space exploration tools such as PREESM [17], MAPS [18], Mocasin [19] and Sesame [20] allow to efficiently map the application to heterogeneous hardware. With respect to the requirements of modern telecommunication workloads, SDF and KPN have two major limitations. First, they don't provide a timed semantics that allows to reason about the real-time requirements of different independent computations in the system. Second, they cannot easily model dynamic and reactive behavior, such as reactions to spontaneous inputs and adaptations of the application.

The lack of dynamicity in SDF and KPN is commonly addressed by switching to a more permissive model like Hewitt actors [21], general asynchronous message passing paradigms, or general task models. However, if we use such an asynchronous model, we lose the determinism guarantee and still lack a timed semantics [22]. The reactor model promises to close the gap between static deterministic models and more dynamic approaches [1] (see Section II-A). While reactor programs are also represented as a static graph, the model also defines *mutations* that allow modifications of the graph during execution.

The logical execution time (LET) model, which can be traced back to the time-triggered language Giotto [23], is a related approach that is gaining popularity in the automotive sector and was recently included in the AUTOSAR standard [24]. In LET, a logical execution time is assigned to all tasks. If a task finishes its computations earlier than its logical execution time, it will delay its outputs. This enables a deterministic composition of dependent tasks. System-level LET (SL-LET) [25] is a recent extension to LET, introducing Time Zones and interconnecting LET, enabling the modeling of distributed systems. In LF, LET can be modeled using delayed connections. Timing enclaves represent an efficient way of utilizing the parallelism exposed by LET, enabling both multicore and real-time scheduling of LET programs.

Authors in [26] propose an approach for coordination across multiple timelines through federated execution. This work targets distributed execution of reactors by using two schemes, one centralized and one decentralized. The centralized system has a runtime infrastructure (RTI) process responsible for coordinating the advancement of logical time of the nodes, also called federates, each of which executes in its own process. To ensure correctness, all messages exchanged between federates must pass through the RTI. The decentralized system avoids this bottleneck and allows for peer-to-peer communication between the federates. However, it is based on Ptides and requires known bounds on the apparent latency between the federates to deliver determinism. Timing enclaves resemble federates, but they execute within a single process using peer-to-peer communication and coordination. Using the logical time barriers, they achieve determinism without any assumptions on apparent latency. In future work, these techniques may be combined to coordinate cycles of timing enclaves.

VII. Conclusions and future work

We have presented timing enclaves, a mechanism for partitioning a discrete-event system into multiple enclaves with decoupled timelines. Our approach exposes more parallelism compared to prior work, both within a logical tag and across tags. Timing enclaves significantly outperform prior scheduling approaches on a realistic 4G/5G workload. The results are comparable to nondeterministic programming models like the original task-based model used in the PHY benchmark. Future work will focus on extending the presented work to consider better mechanisms for analyzing LF programs and prompting users with useful feedback that allows them to make educated decisions on how to partition the program. Additionally, we plan to develop tools that automatically explore the design space of possible solutions and utilize heuristics to identify solutions that are near optimal regarding certain constraints. This is particularly interesting in the context of heterogeneous systems in the compute continuum [27], [28].

Acknowledgment

This work was funded in part by the German Federal Ministry of Education and Research (BMBF) through the project "E4C" (16ME0426K) and the programme of "Souverän. Digital. Vernetzt." joint project 6G-life (16KISK001K). This work also received funding from the EU Horizon Europe Programme under grant agreement No 101135183 (MYRTUS). Views and opinions expressed are however those of the author(s) only and do not necessarily reflect those of the European Union. Neither the European Union nor the granting authority can be held responsible for them.

REFERENCES

[1] M. Lohstroh *et al.*, "Reactors: A deterministic model for composable reactive systems," in *Cyber Physical Systems. Model-Based Design*, R. Chamberlain, M. Edin Grimheden, and W. Taha, Eds., Cham: Springer, 2020, pp. 59–85.

[2] M. Lohstroh *et al.*, "Deterministic coordination across multiple timelines," *ACM Transactions on Embedded Computing Systems (TECS)*, Oct. 2023, ISSN: 1539-9087.

[3] C. Menard *et al.*, "High-performance deterministic concurrency using lingua franca," *ACM Transactions on Architecture and Code Optimization (TACO)*, vol. 20, no. 4, pp. 1–29, Aug. 2023, ISSN: 1544-3566.

[4] M. Själander *et al.*, "An lte uplink receiver phy benchmark and subframe-based power management," in *2012 IEEE International Symposium on Performance Analysis of Systems & Software*, 2012, pp. 25–34.

[5] M. Lohstroh, "Reactors: A deterministic model of concurrent computation for reactive systems," Ph.D. dissertation, EECS Department, University of California, Berkeley, 2020.

[6] M. Lohstroh *et al.*, "Logical time for reactive software," in *Proceedings of Cyber-Physical Systems and Internet of Things Week 2023*, ser. CPS-IoT Week '23, San Antonio, TX, USA: Association for Computing Machinery, 2023, 313–318, ISBN: 9798400700491.

[7] M. Lohstroh *et al.*, "Toward a Lingua Franca for deterministic concurrent systems," *ACM Trans. Embed. Comput. Syst.*, vol. 20, no. 4, 2021.

[8] X. Wang *et al.*, "Virtualized cloud radio access network for 5g transport," *IEEE Communications Magazine*, vol. 55, no. 9, pp. 202–209, 2017.

[9] R. Wittig *et al.*, "Modem design in the era of 5G and beyond: The need for a formal approach," in *Proceedings of the 27th International Conference on Telecommunications (ICT)*, Virtual. Bali, Indonesia, 2020, pp. 1–5.

[10] P. Popovski *et al.*, "5G wireless network slicing for embb, urllc, and mmtc: A communication-theoretic view," *IEEE Access*, vol. 6, pp. 55 765–55 779, 2018.

[11] O. Elijah *et al.*, "A comprehensive survey of pilot contamination in massive MIMO—5G system," *IEEE Communications Surveys & Tutorials*, 2016.

[12] V. Venkataramani *et al.*, "SPECTRUM: A software-defined predictable many-core architecture for LTE/5G baseband processing," *ACM Trans. Embed. Comput. Syst.*, vol. 19, no. 5, 2020, ISSN: 1539-9087.

[13] R. D. Blumofe and C. E. Leiserson, "Scheduling multithreaded computations by work stealing," *J. ACM*, vol. 46, no. 5, 720–748, 1999, ISSN: 0004-5411.

[14] J. Robledo and J. Castrillon, "Parameterizable mobile workloads for adaptable base station optimizations," in *2022 IEEE 15th International Symposium on Embedded Multicore/Many-core Systems-on-Chip (MCSoC)*, 2022, pp. 381–386.

[15] E. Lee and D. Messerschmitt, "Synchronous data flow," *Proceedings of the IEEE*, vol. 75, no. 9, pp. 1235–1245, 1987.

[16] G. Kahn, "The semantics of a simple language for parallel programming," *Information processing*, vol. 74, pp. 471–475, 1974.

[17] M. Pelcat *et al.*, "PREESM: A dataflow-based rapid prototyping framework for simplifying multicore DSP programming," in *2014 6th European Embedded Design in Education and Research Conference (EDERC)*, Sep. 2014, pp. 36–40.

[18] J. Castrillon, R. Leupers, and G. Ascheid, "MAPS: Mapping concurrent dataflow applications to heterogeneous MPSoCs," *IEEE Transactions on Industrial Informatics*, vol. 9, no. 1, pp. 527–545, Feb. 2013, ISSN: 1551-3203.

[19] C. Menard *et al.*, "Mocasin—rapid prototyping of rapid prototyping tools: A framework for exploring new approaches in mapping software to heterogeneous multi-cores," in *Proceedings of the 2021 Drone Systems Engineering and Rapid Simulation and Performance Evaluation: Methods and Tools Proceedings*, ser. DroneSE and RAPIDO '21, Budapest, Hungary: Association for Computing Machinery, 2021, 66–73, ISBN: 9781450389525.

[20] C. Erbas *et al.*, "A framework for system-level modeling and simulation of embedded systems architectures," *EURASIP Journal on Embedded Systems*, vol. 2007, pp. 1–11, 2007.

[21] C. Hewitt, P. Bishop, and R. Steiger, "Session 8 formalisms for artificial intelligence a universal modular actor formalism for artificial intelligence," in *Advance Papers of the Conference*, Stanford Research Institute Menlo Park, CA, vol. 3, 1973, p. 235.

[22] M. Lohstroh and E. A. Lee, "Deterministic actors," in *2019 Forum for Specification and Design Languages (FDL)*, 2019.

[23] T. A. Henzinger, B. Horowitz, and C. M. Kirsch, "Giotto: A time-triggered language for embedded programming," in *EMSOFT 2001*, vol. LNCS 2211, Springer-Verlag, 2001, pp. 166–184.

[24] AUTOSAR, "Specification of timing extensions," *AUTOSAR CP Release 4.4.0*, Oct. 2018.

[25] K.-B. Gemlau *et al.*, "System-level logical execution time: Augmenting the logical execution time paradigm for distributed real-time automotive software," *ACM Trans. Cyber-Phys. Syst.*, vol. 5, no. 2, 2021, ISSN: 2378-962X.

[26] S. Bateni *et al.*, "Risk and mitigation of nondeterminism in distributed cyber-physical systems," in *Proceedings of the 21st ACM-IEEE International Conference on Formal Methods and Models for System Design*, ser. MEMOCODE '23, Hamburg, Germany: Association for Computing Machinery, 2023, 1–11, ISBN: 9798400703188.

[27] C. Pilato *et al.*, "EVEREST: A design environment for extreme-scale big data analytics on heterogeneous platforms," in *Proceedings of the 2021 Design, Automation and Test in Europe Conference (DATE)*, ser. DATE'21, Virtual Conference, Feb. 2021, pp. 1320–1325.

[28] F. Palumbo *et al.*, "MYRTUS: Multi-layer 360 dynamic orchestration and interoperable design environment for compute-continuum systems," in *Proceedings of the 21st ACM International Conference on Computing Frontiers (CF'24)*, ser. CF '24, Ischia, Italy: Association for Computing Machinery (ACM), May 2024, 6pp, ISBN: 979-8-4007-0492-5/24/05.