Automating timing enclaves for reactive programs in Lingua Franca

Julian Robledo
Chair for Compiler Construction
TU Dresden
Dresden, Germany
julian.robledo@tu-dresden.de

Jeronimo Castrillon
Chair for Compiler Construction
TU Dresden
Dresden, Germany
jeronimo.castrillon@tu-dresden.de

***Abstract*—Lingua Franca (LF) is a coordination language for cyber-physical systems that ensures deterministic execution through the reactor model. While LF's conservative scheduler guarantees correctness, it can hinder performance due to synchronization barriers that limit parallelism. Timing enclaves, a recent extension to LF, mitigate this by partitioning programs into independently scheduled regions with decoupled logical timelines. However, determining optimal enclave configurations is non-trivial and not intuitive to programmers, as it requires balancing parallelism, synchronization overhead, and system constraints. In this paper, we present a framework for Design Space Exploration (DSE) of timing enclaves in LF programs. Our approach automatically analyzes the program's topology, identifies valid enclave partitions, and uses heuristics to explore trade-offs between determinism and latency. We evaluate the effectiveness of our method on synthetic benchmarks and a real-world 4G baseband processing application. Results show that our DSE framework can significantly improve performance while preserving deterministic semantics, highlighting its practical value for complex real-time systems.**

***Index Terms*—DSE, Reactors, Lingua Franca.**

I. Introduction

Cyber-Physical Systems (CPS) are increasingly transforming the way we interact with the physical world by tightly integrating computational intelligence with physical processes. They play a critical role across a wide range of industries, including automotive systems, industrial automation, healthcare, and telecommunications. As these systems continue to evolve rapidly, they introduce complex challenges in concurrency, safety, and scalability. Unlike purely digital systems, the physical elements of CPS impose strict timing constraints. However, most software abstractions used in CPS components do not account for time explicitly. Since computations take time and interact with the physical world, an appropriate Model of Computation (MoC) for CPS must incorporate a notion of time.

The reactor model [1] emerged as a promising programming paradigm to address these challenges. It is a MoC for reactive systems that provides a structured approach to managing time-sensitive interactions. It follows a discrete-event paradigm, where concurrently executing entities called reactors, communicate through timestamped messages. This allows developers to express temporal logic explicitly while maintaining deterministic execution, even in highly concurrent environments. Event handling is governed by a partially ordered set of timestamps. Lingua Franca (LF) is a coordination language that supports the reactor model, offering both a compiler and a runtime engine [2]. LF has demonstrated good performance on various benchmarks, proving it can execute a range of applications deterministically without compromising efficiency [3]. However, these benchmarks tend to feature regular structures and uniform execution times, which may not reflect the performance in more complex, real-world scenarios where timing and determinism are not enforced.

Designing runtime strategies for such models involves two main challenges: effectively leveraging their inherent parallelism for execution on multi-core processors, and guaranteeing predictable timing for real-time task execution. The LF runtime processes events in timestamp order, allowing parallel execution only when events are logically simultaneous and there are no dependencies between them. Dependencies are mapped as a graph and resolved at startup via topological sorting. While effective, this introduces implicit timing barriers that constrain parallel execution and may lead to priority inversion, where high-priority tasks are delayed by unrelated, lower-priority ones [4].

The work in [5] solved this problem by introducing a mechanism that divides reactors programs into independent timelines called *timing enclaves*, each managed by its own scheduler. This reduces the number of implicit timing barriers and boosts parallelism. The proposed approach implements coordination strategies for concurrent execution while maintaining determinism across these enclaves. However, deriving strategies to find the optimal number of timing enclaves and the distribution of the reactors among those enclaves is not a trivial task. For evaluation, authors in [5] leveraged prior knowledge about a signal processing application to propose a set of partitioning via timing enclaves, which were manually implemented.

In this paper, we propose a methodology to automatically explore the design space of reactor programs with timing enclaves, using heuristics to identify solutions that meet specific constraints or optimization goals. For this, we extend the LF compiler to recognize design patterns that are common to several signal processing and streaming applications, and generate multiple reactor programs for evaluation in order to

find the best suited solution. The experimental results show that the number of missed deadlines of a complex real-time use case, as a 4G baseband base station, can be reduced by 52% with our methodology, compared to the baseline application without timing enclaves.

The remainder of the paper is organized as follows. Section II introduces the reactor model and the timing enclaves mechanism. Section III describes our framework, the used heuristics and its limitations. Section IV analyses our experimental results. Section V discusses related work and Section VI discusses conclusions and future work.

II. Background

This section provides a summary of the reactor model, focusing mainly on the aspects relevant to the problems we tackle. For a more comprehensive explanation of the model, refer to the work in [6].

A. The reactor model and its scheduler

Although most programming languages do not include time in their syntax, for time-sensitive systems, time should be an intrinsic property of the model, rather than a performance metric. Therefore, the reactor model uses the concept of logical time [7]. Logical time is an abstract notion of time that is used to order events in a deterministic and consistent way, regardless of how fast or slow the physical system or processor is running. It does not correspond directly to a wall clock or real (physical) time. In the reactor model, logical time allows to reason about when things happen in a program in relation to other events, rather than based on the actual physical time when the code is executed. Logical time is usually represented as a timestamp [8], also called tag $g = (t, m)$. The component t is a numerical value representing elapsed time, while the microstep m is a sub-order used to sequence multiple events that happen at the same time t.

Reactors are the fundamental building block in the reactor model. They are self-contained, concurrent, and reactive units that process events according to a specified timeline. Reactors have input and output ports through which they receive and send events. An event is a timestamped, optionally data-carrying signal that triggers reactions within reactors. Reactions specify the behavior of the reactor, they are mutually exclusive within a single reactor, meaning that they cannot run concurrently. The reactor model defines a strict partial order over events, ensuring that even in concurrent systems, events are processed in a consistent and predictable manner.

Building a scheduler for a reactor model poses multiple challenges. The scheduler is responsible for managing event execution in timestamp order, exploiting concurrency when possible, and advancing logical time based on pending events. A process with tag g may only be executed when the physical time is greater than the timestamp of g. Moreover, the scheduler ensures that all reactions scheduled for the current logical time complete before progressing to the next tag. This results in an implicit tag barrier. If a reaction with high computational load takes significantly longer to execute, it may finish at a physical time beyond the logical time of the upcoming tag. In such situations, we say that logical time is lagging behind physical time.

Exploiting concurrency of simultaneous reactors, without dependency between them, is a desired but non-trivial feature of a scheduler in the reactor model. The goal is to run as many reactions in parallel as possible without breaking determinism. Lingua Franca, a programming language that implements the reactor model, attempts to solve this by using a dependency graph. At startup, the scheduler derives an acyclic precedence graph (APG) by looking into the ports and connections of the reactor program. The LF assigns levels to reactions within the APG, permitting parallel execution only among reactions at the same level. However, this results in a very conservative approach since the dynamic traversal of the graph can be computationally costly, specially for large-scale applications. Moreover, in the case of a system with multiple signal processing chains that do not interact at every step (e.g., pipelined channels in signal processing), when the execution time of the reactions is data dependent, and there is jitter between consecutive calls to the same reaction, it might affect the execution of other unrelated chains. The level-based approach creates gaps in the schedule that could be avoided for optimized performance. This is also known as the level barrier.

B. Timing enclaves

The tag barrier prevents reactions triggered at different tags from executing in parallel and scheduling timestamped events in LF is handled by a level-based runtime that limits the amount of parallelism. The work in [5] introduces *timing enclaves* in LF as a solution for better multi-core utilization. Timing enclaves are partitions of a reactor program into independent scheduling domains, each with its own logical timeline and scheduler. The key idea is to relax the global synchronization barriers of the reactor model as such barriers become local to each enclave. Coordination mechanisms are also introduced for safe interaction between enclaves, ensuring determinism even across logical time domains.

This method, if used correctly, addresses the limitations introduced by the synchronization barriers in an efficient manner. Figure 1 illustrates a simple reactor program. It shows a directed graph representing the structure of the program, where each node corresponds to a reactor containing a single reaction. The topology consists of a source node connected to two independent pipelines, each composed of two sequential tasks, followed by a final sink node. Two alternative enclave strategies are visualized: the first one highlights groups of reactors in gray to indicate a horizontal partitioning, where each pipeline forms its own enclave, while the second one uses blue to mark a vertical partitioning strategy, where stages across pipelines are grouped together. The source and sink nodes are embedded in their own enclave for each of the strategies.

Figure 2 shows the impact of these two enclave strategies on execution performance. Let us assume that the source node,

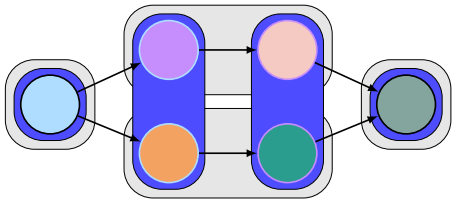
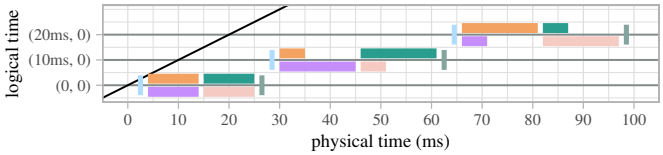
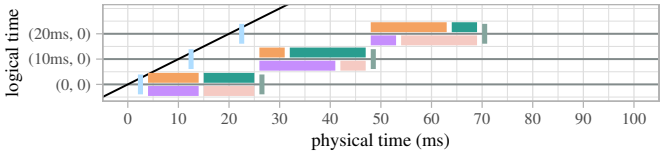


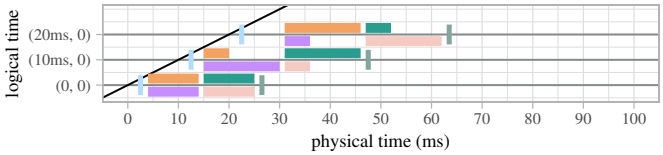
Fig. 1: A directed graph representing a reactor program, every node corresponds to a reactor containing a single reaction. It also highlights two possible partition strategies, one is shown in blue and the other one in gray.



(a) Timing diagram for the program in Fig 1 without enclaves.



(b) Timing diagram for the program in Fig 1 with the enclaves in gray.



(c) Timing diagram for the program in Fig 1 with the enclaves in blue.

Fig. 2: timing diagrams for different enclave configurations.

triggers the two pipelines every 10 ms, and that the makespan is data dependent, i.e. different calls to the same reaction might result in different execution times. The figure shows a timing diagram with logical time in the y-axis and the physical time in the x-axis. The colors of the bars correspond to the execution time of the reactor with the same color in Figure 1. These figures are conceptual and do not include the potential overhead introduced by coordination mechanisms between enclaves. The Figure 2a shows the baseline performance without timing enclaves, where all reactions are synchronized at each logical step, limiting parallelism. In Figure 2b, each pipeline is placed in its own enclave, allowing concurrent progression across pipelines. Figure 2c, groups the stages of both pipelines into shared enclaves, which can reduce synchronization overhead at specific stages. These comparisons highlight how the choice of enclave configuration can influence system performance for a specific set of inputs.

While timing enclaves enable concurrent execution by decoupling logical timelines, their effectiveness heavily depends on how the program is partitioned. A poorly chosen enclave structure may introduce unnecessary coordination costs or limit the available parallelism, ultimately negating the performance benefits. Moreover, designing efficient partition strategies often requiring expert knowledge of the system's structure, timing behavior, and the semantics of the reactor model.

In LF, enclaves are specified by annotating reactors in the source code. Due to the composable and hierarchical nature of reactors, annotating a reactor as an enclave implicitly includes all of its contained (nested) reactors within that enclave. This means that even a seemingly simple annotation can have far-reaching effects, potentially influencing large portions of the application's structure and execution. Moreover, This design choice has significant implications: to create an enclave that encapsulates two or more interconnected reactors, the user must first create a third, higher-level reactor that wraps them, and then apply the enclave annotation at this new level. Such transformations are not only tedious and error-prone but also obscure the program's original modular design, making manual enclave definition especially challenging for large systems.

Furthermore, the impact of enclave placement on runtime behavior is not always intuitive. Naive strategies, such as annotating every reactor as its own enclave, quickly become counterproductive. While this might expose maximum parallelism in theory, in practice it leads to the creation of many independent schedulers, each with its own overhead. This excessive fragmentation can increase coordination costs and introduce inefficiencies, ultimately degrading performance rather than improving it.

In this paper, we argue that the responsibility of finding effective enclave structures should not fall on the user; instead, there should be automated tools and frameworks that support this task. From the compiler's perspective, supporting timing enclaves requires sophisticated analysis: identifying candidate groupings of reactors, reasoning about their dataflow and timing dependencies, and generating valid program transformations that preserve the semantics of the original application. Therefore, DSE becomes essential as it provides a systematic approach to evaluate multiple enclave configurations, guided by heuristics, to identify those that best meet performance and timing requirements without compromising correctness. By automating this process, DSE empowers developers to fully exploit the potential of timing enclaves, especially in real-time and resource-constrained systems.

III. Design Space Exploration for Timing Enclaves

To better understand the impact of timing enclaves and identify effective enclave configurations, we developed a DSE framework for programs written in the LF coordination language. The primary goal of this framework is to automatically

explore partitioning strategies that balance parallelism, determinism, and scheduling efficiency, without relying on manual tuning from the user. The framework was developed as an extension for the LF compiler, integrating directly into the compilation workflow to analyze the program structure.

A. LF compiler extension

We primarily focus on systems with topologies similar to signal processing systems. They are an ideal domain for evaluating our framework because of their well-defined computational patterns, stringent timing requirements, and frequent use in real-time and embedded applications. These systems often exhibit structured, acyclic dataflows, such as pipelines, filter chains, and parallel processing stages, that align well with the hierarchical and modular design of Lingua Franca programs. Moreover, signal processing workloads are typically data dependent, and involve variable latency and throughput demands, making them a suitable testbed for stressing the synchronization barriers of the reactor model as they display both, pipeline and data parallelism.

To support automated enclave generation, we extended the LF compiler with a new compilation parameter `--generate-enclaves`. This flag triggers the DSE process for timing enclaves during compilation. Since signal processing systems often consist of many identical chains of reactors executed in parallel, our methodology requires that the input LF program contains only a single instance of such a pipeline. The desired number of parallel replicas can be specified using an additional parameter, `--num-replicas`. When both parameters are set, the modified compiler analyzes the structure and dependencies of the input program to identify feasible enclave configurations. Therefore, an LF that uses our extension to generate 5 replicas of given pipeline, can be compiled as: `lfc my_program.lf --generate-enclaves --num-replicas=5`. For each valid configuration, the compiler automatically generates appropriate wrapper reactors to define enclaves and updates the connection topology accordingly. This relieves the user from manually constructing hierarchical compositions and guarantees syntactic correctness.

While our framework is designed to automate the generation and evaluation of enclave configurations, it also offers flexibility for expert users. Developers with a deep understanding of their program's topology and timing behavior may choose to bypass the automated heuristics and instead provide custom enclave configurations directly. This can be done by additionally using a straightforward list-based syntax `--enclave-list {[reactor1, reactor2], [reactor3]}`. In this format, each square bracket defines a single enclave and contains the names of the reactors assigned to it. This manual approach enables fine-grained control over the program's partitioning strategy, allowing users to encode domain-specific knowledge. Moreover, by accepting enclave configurations as a structured list, our toolchain provides an interface that can be easily connected to an auto-tuner, to iteratively propose improved enclave partitions. By combining this manual capability with the automated exploration tools, our framework caters to both novice users and experienced system designers.

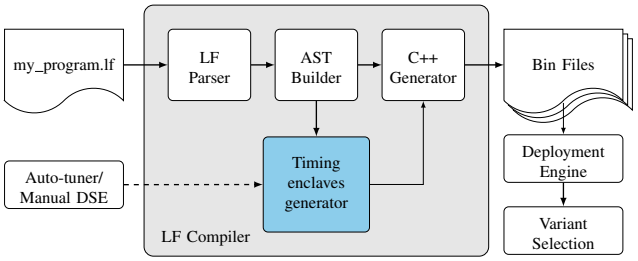


Fig. 3: Overview of our toolflow.

B. DSE for timing enclaves

Our DSE framework employs a heuristic based on a the Algorithm U described in [9], which generates all k-subsets of a given array, where a k-subset of a set X is a partition of all the elements in X into k non-empty subsets. In this way it is possible to get the set partitions of the reactor components into a fixed number of blocks.

These partitions represent potential timing enclave configurations. However, not all partitions are suitable: each enclave must form a connected subgraph to ensure proper communication and scheduling semantics. To enforce this, we use the Spielman's connectivity test [10], which efficiently verifies the connectivity of each candidate block. Spielman's approach to graph connectivity centers around the use of spectral graph theory, where the Laplacian matrix of a graph encodes key connectivity information. It allows fast approximations of connectivity and other structural properties in very large graphs. Only partitions where every block corresponds to a connected subgraph are retained. The valid partitions are then generalized across all pipeline replicas in the original program to construct full-system enclave configurations. For each configuration, the framework generates the corresponding LF code, enabling subsequent evaluation through deployment and analysis.

Figure 3 shows an overview of our toolflow. An LF program is processed by our extended LF compiler. Based on our heuristics and manual DSE, the compiler explores multiple partitioning strategies and generates a corresponding reactors program for each valid configuration, that is individually compiled into a binary executable. These binaries are subsequently passed to a deployment engine, which evaluates them in the target environment. We then proceed to select the variant that best aligns with a given optimization goal, such as minimizing latency.

While this approach relies on relatively simple heuristics and structural analysis, it removes the burden from developers and enables consistent, syntactically valid transformations. It also serves as a foundation for more sophisticated exploration techniques. To encourage broader use and further innovation, we open-source our tool[1].

[1]https://github.com/tud-ccc/lingua-franca-enclaves

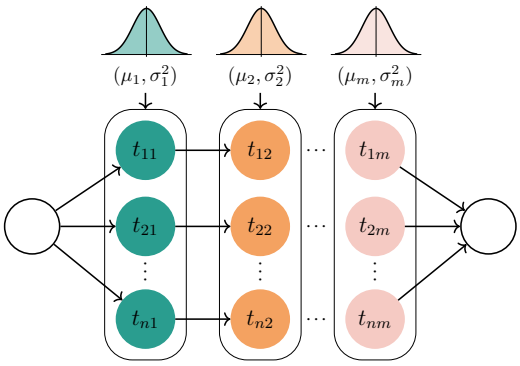


Fig. 4: Synthetic benchmark structure.

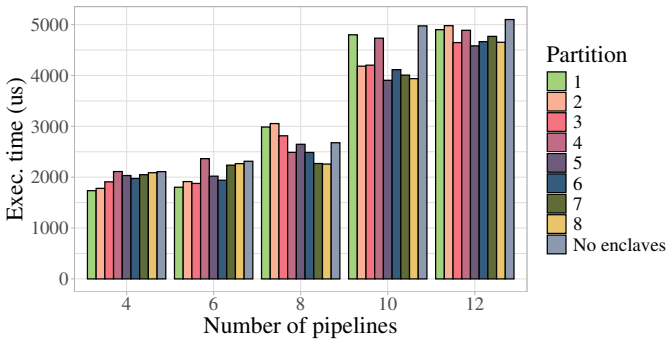


Fig. 5: Performance with homogeneous task execution time.

IV. Execution analysis

To assess the effectiveness and practicality of our DSE framework for timing enclaves in Lingua Franca, we conduct a series of experiments on both synthetic and real-world benchmarks. Our evaluation focuses on measuring the impact of different enclave configurations on key runtime metrics such as execution time and scheduler efficiency. We analyze how well the framework identifies valid and beneficial enclave structures, and to what extent it reduces scheduling bottlenecks caused by tag and level barriers. This section presents the experimental methodology, benchmarks, and performance results, followed by a discussion of insights and limitations. All experiments were conducted on a machine equipped with a multi-core Intel(R) Core™ i7-6700, a 4-core/8-thread processor working at a base frequency of 3.40GHz. We use a single workstation to ensure consistent performance measurements.

A. Synthetic benchmark generation

To systematically evaluate our framework, we construct a suite of synthetic programs that mimic common concurrency and communication patterns found in signal processing pipelines, filter banks, and multi-stage computational graphs. These benchmarks allow fine-grained control over parameters such as pipeline depth, width, and execution time. Figure 4 illustrates the structure of the synthetic benchmarks. Each of them consists of multiple replicas of a configurable processing pipeline, where the pipeline depth is defined by the parameter m, and the number of parallel pipelines is given by the parameter n. Each node in the graph represents a reactor performing a single task.

The benchmarks are synthetic as the computation carried out by the reactions does not have any meaning, and they are intended to simulate workload during a specified execution time. To simulate realistic execution behavior, each type of task (i.e., each position in the pipeline) across all pipelines is assigned execution times drawn from a normal distribution specific to that task type. This allows us to control both the average execution time and its variability. This setup enables us to isolate and analyze the impact of structural and timing variability on enclave partitioning strategies.

We generated versions with both homogeneous and heterogeneous reaction execution times. For the homogeneous workload scenario, we simply set the variance of all distributions to zero, resulting in identical execution times for all corresponding tasks across pipelines. On the other hand, heterogeneous variants draw costs from the normal distribution, emulating load imbalance scenarios. Additionally, we vary the number of pipeline replicas to assess how the framework scales with increasing parallelism and program size. These synthetic programs serve as controlled test cases to evaluate the correctness of enclave detection, the effectiveness of partition pruning heuristics, and the runtime performance improvements achieved through timing enclave insertion.

Figure 5 shows the average execution time of LF programs composed of multiple parallel pipelines of reactors, each with a fixed depth of four sequential tasks. Since all reactions in the same stage across all pipelines have the same execution time, the only source of variability is the number of pipelines. For a pipeline with length 4, our heuristics identify 8 different enclave configurations. For the cases with 4 and 6 pipelines, there are some performance differences across the various enclave configurations, but almost all of them outperform the baseline implementation without enclaves. With 8 pipelines, the system starts to exhibit more distinct performance differences across enclave configurations. However, most enclave configurations still perform better than the baseline. When increasing the number of pipelines to 10 and 12, a performance bottleneck emerged in the baseline (non-enclave) version. Without enclaves, all pipelines must execute the same reaction phase in lockstep; that is, each pipeline must complete a given reaction before any can proceed to the next one due to the level barrier. When the number of pipelines exceeds the number of cores, at least one pipeline must wait for CPU resources to become available. This forces all other pipelines to idle at synchronization points until the slowest one completes its reaction. Enclaves mitigate this problem by decoupling execution timelines, allowing pipelines to progress independently and enabling better scheduling across available cores. As a result, in the 10 and 12 pipelines scenarios, all enclave configurations outperformed the baseline by improving core utilization and reducing idle time caused by global synchronization barriers.

Figure 6 shows the average execution time for the same

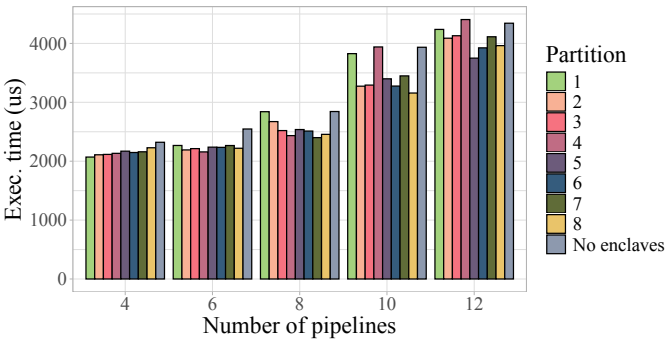


Fig. 6: Performance with heterogeneous task execution time.

programs as in Figure 5, but varying also the execution time of individual tasks. The execution times of individual reactions were assigned randomly. This variability introduces an additional layer of complexity, as synchronization barriers are no longer stressed uniformly across different pipeline configurations. The figure displays a similar trend to Figure 5. With 4 and 6 pipelines, the execution times are relatively close across all partitioning strategies and the baseline without enclaves.

The configurations with 8, 10, and 12 pipelines, however, showed greater variability in results. In this context, certain enclave configurations were more effective at alleviating contention and minimizing idle time. While not all enclave strategies yielded improvements, most of them outperformed the baseline version. These results indicate that even a simple DSE approach can identify beneficial partitioning strategies that significantly improve performance in scenarios with limited resources and irregular execution profiles.

These results underscore a key insight: identifying effective partitioning strategies for timing enclaves is not always intuitive. Factors such as workload distribution, core availability, and synchronization patterns significantly influence whether a given enclave configuration improves or degrades performance. Even in seemingly simple scenarios, subtle differences in execution times or system load can lead to substantial changes in the behavior of the scheduler and the effectiveness of parallelism. This variability highlights the importance of automated tools, as manual tuning would be both time-consuming and error-prone in the face of such complex interactions.

B. 4G use case

To complement our synthetic benchmarks, we evaluate the Design Space Exploration framework on a real-world use case: a 4G baseband signal processing system. This application represents a highly relevant and complex scenario with strict real-time requirements and substantial computational variability. The system is structured around multiple parallel instances of a signal processing pipeline that includes stages such as FFT/IFFT, modulation, demodulation, filtering, and encoding. Each of these stages is modeled as a reactor in Lingua Franca. For this, we use the benchmark presented in [5] for a high traffic scenario. The input traces for this traffic scenario were obtained by following the methodology in [11].

While the overall pipeline topology remains consistent across instances, execution times vary significantly across different calls due to dynamic factors such as the amount of data contained in the input signal and the modulation scheme used to encode that signal. Moreover, the amount of parallelism that can be leveraged is influenced by both static and dynamic factors. For instance, a new batch of input signals, also called requests, is received at every millisecond, this is called a subframe. The number of requests in a subframe defines the number of pipelines instances to be called. On the other hand, the number of antennas in a base station, which defines the degree of parallelism of some reactors, is a fixed hardware parameter. This variability poses a challenge for traditional scheduling approaches and makes the use of timing enclaves particularly compelling. Since the application processes a wide range of input signals and call scenarios, the primary metric for evaluation is not latency alone but the number of missed deadlines, during execution across many inputs, i.e. when a call to a given pipeline fails to process an input signal within a predefined deadline.

In this use case, we evaluated 128 different enclave configurations generated by our framework. Figure 7 presents the enumerated enclave variants from best to worst according to their performance measured in terms of missed deadlines. We used a high traffic scenario, which contains 10.000 subframes with high signals variability. The best-performing configuration reduced the number of missed deadlines by 12% compared to the worst-performing one. Moreover, compared to the baseline version without enclaves, the number of missed deadlines is reduced by 52%, highlighting the substantial gains possible through effective partitioning.

V. Related work

Models of computation such as Synchronous Data Flow (SDF) [12] and Kahn Process Networks (KPNs) [13] have long been foundational in expressing concurrency in signal processing and similar applications. These paradigms allow deterministic behavior and efficiently exploit both pipeline and data-level parallelism. Applications are typically represented as directed graphs, where computation is encapsulated in nodes and communication flows along edges. In KPNs, computation units are modeled as persistent processes that communicate through blocking FIFO channels. In contrast, SDF imposes more rigid semantics by requiring that a fixed number of data tokens be available for a node to fire and precisely defining how many tokens are produced. Tools such as PREESM [14], MAPS [15], Mocasin [16], and Sesame [17] provide robust support for mapping such models onto heterogeneous architectures, and hybrid scheduling strategies can dynamically optimize execution at runtime However, these models face two significant limitations in the context of modern telecommunication workloads. First, they lack a notion of time, which limits their ability to express and reason about real-time constraints of independent computations. Second, they

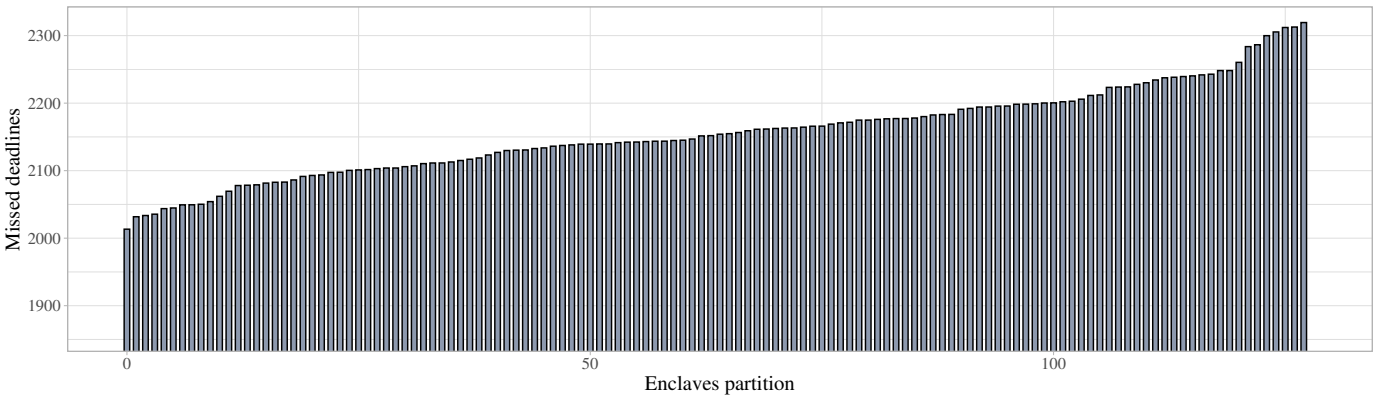


Fig. 7: Performance of timing enclaves for a 4G base station use-case.

struggle to model dynamic and reactive behaviors, such as responding to asynchronous inputs or adapting to changing system conditions.

The recent contribution in [5] introduced the concept of timing enclaves within the Lingua Franca coordination language to improve parallelism while maintaining deterministic execution in reactive systems. This work demonstrates how partitioning programs into independent enclaves, each with its own logical timeline, can mitigate the limitations imposed by global tag and level barriers in the reactor model. This approach has shown significant promise for applications with acyclic structures by enabling concurrent execution of logically independent components. However, this methodology has key limitations. It lacks automated heuristics or guidance for identifying optimal enclave structures, placing the burden of manual partitioning on the developer. These gaps highlight the need for DSE tools and automated analysis techniques that can systematically discover viable enclave configurations. Additionally, the method does not generalize well to programs containing cyclic dependencies, where determining valid timing enclaves becomes significantly more complex due to potential feedback loops and scheduling constraints. However, this problem is yet to be addressed.

VI. Conclusions and future work

In this paper, we presented a design space exploration framework for timing enclaves in the Lingua Franca coordination language, aimed at improving the parallel execution of cyber-physical systems modeled using the reactor model. By leveraging heuristics for systematic partitioning, our framework automatically generates valid enclave configurations, relieving users from the complex task of manual enclave design.

Our evaluation on both synthetic signal processing pipelines and a real-world 4G baseband processing use case demonstrated the framework's ability to uncover enclave structures that enhance runtime performance by reducing implicit timing barriers and improving parallelism. Particularly in scenarios with variable execution times and large-scale parallelism, the automatic exploration significantly decreased latency and deadline misses compared to baseline and manually designed configurations.

Future work will focus on extending the framework to better handle cyclic dependencies within programs, integrating sophisticated heuristics for exploring the design space for performance and energy consumption like in [18], [19], and exploring adaptive enclave reconfiguration at runtime. We believe this approach provides a valuable tool for developers targeting predictable and efficient execution in Lingua Franca.

Acknowledgment

This work was funded in part by the German Federal Ministry of Research, Technology and Space of Germany (BMFTR) 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, Í. Í. Romeo, A. Goens, P. Derler, J. Castrillon, E. A. Lee, and A. Sangiovanni-Vincentelli, "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), pp. 59–85, Springer, 2020.

[2] M. Lohstroh, S. Bateni, C. Menard, A. Schulz-Rosengarten, J. Castrillon, and E. A. Lee, "Deterministic coordination across multiple timelines," *ACM Transactions on Embedded Computing Systems (TECS)*, Oct. 2023.

[3] C. Menard, M. Lohstroh, S. Bateni, M. Chorlian, A. Deng, P. Donovan, C. Fournier, S. Lin, F. Suchert, T. Tanneberger, H. Kim, J. Castrillon, and E. A. Lee, "High-performance deterministic concurrency using lingua franca," *ACM Transactions on Architecture and Code Optimization (TACO)*, vol. 20, pp. 1–29, Aug. 2023.

[4] C. Menard, *Deterministic Reactive Programming for Cyber-physical Systems*. PhD thesis, June 2024.

[5] J. Robledo, C. Menard, E. Jellum, E. A. Lee, and J. Castrillon, "Timing enclaves for performance in lingua franca," in *2024 Forum for Specification and Design Languages (FDL)*, pp. 1–9, Sept. 2024.

[6] M. Lohstroh, *Reactors: A Deterministic Model of Concurrent Computation for Reactive Systems*. PhD thesis, EECS Department, University of California, Berkeley, Dec 2020.

[7] M. Lohstroh, E. A. Lee, S. A. Edwards, and D. Broman, "Logical time for reactive software," in *Proceedings of Cyber-Physical Systems and Internet of Things Week 2023*, CPS-IoT Week '23, (New York, NY, USA), p. 313–318, Association for Computing Machinery, 2023.

[8] M. Lohstroh, C. Menard, S. Bateni, and E. A. Lee, "Toward a Lingua Franca for deterministic concurrent systems," *ACM Trans. Embed. Comput. Syst.*, vol. 20, may 2021.

[9] D. E. Knuth, *The Art of Computer Programming, Volume 4, Fascicle 3: Generating All Combinations and Partitions*. Addison-Wesley, 2005. Algorithm U.

[10] D. A. Spielman and S.-H. Teng, "Nearly-linear time algorithms for graph partitioning, graph sparsification, and solving linear systems," in *Proceedings of the Thirty-Sixth Annual ACM Symposium on Theory of Computing*, STOC '04, (New York, NY, USA), p. 81–90, Association for Computing Machinery, 2004.

[11] 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)*, pp. 381–386, 2022.

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

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

[14] M. Pelcat, K. Desnos, J. Heulot, C. Guy, J. F. Nezan, and S. Aridhi, "PREESM: A dataflow-based rapid prototyping framework for simplifying multicore DSP programming," in *2014 6th European Embedded Design in Education and Research Conference (EDERC)*, pp. 36–40, 09 2014.

[15] J. Castrillon, R. Leupers, and G. Ascheid, "MAPS: Mapping concurrent dataflow applications to heterogeneous MPSoCs," *IEEE Transactions on Industrial Informatics*, vol. 9, pp. 527–545, Feb. 2013.

[16] C. Menard, A. Goens, G. Hempel, R. Khasanov, J. Robledo, F. Teweleitt, and J. Castrillon, "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*, DroneSE and RAPIDO '21, (New York, NY, USA), p. 66–73, Association for Computing Machinery, 2021.

[17] C. Erbas, A. D. Pimentel, M. Thompson, and S. Polstra, "A framework for system-level modeling and simulation of embedded systems architectures," *EURASIP Journal on Embedded Systems*, vol. 2007, pp. 1–11, 2007.

[18] R. Khasanov and J. Castrillon, "Energy-efficient runtime resource management for adaptable multi-application mapping," in *Proceedings of the 2020 Design, Automation and Test in Europe Conference (DATE)*, DATE '20, pp. 909–914, IEEE, Mar. 2020.

[19] R. Khasanov, M. Dietrich, and J. Castrillon, "Flexible spatio-temporal energy-efficient runtime management," in *29th Asia and South Pacific Design Automation Conference (ASP-DAC'24)*, pp. 777–784, IEEE, Jan. 2024.