Quasi-Static Scheduling for Deterministic Timed Concurrent Models on Multi-Core Hardware

SHAOKAI LIN, University of California, Berkeley, USA
ERLING JELLUM*, University of California, Berkeley, USA
MIRCO THEILE*, TUM School of Engineering and Design, Technical University of Munich, Germany
TASSILO TANNEBERGER*, TU Dresden, Germany
BINQI SUN*, TUM School of Engineering and Design, Technical University of Munich, Germany
CHADLIA JERAD, University of Manouba, Tunisia
YIMO XU, University of California, Berkeley, USA
GUANGYU FENG, University of California, Berkeley, USA
MAGNUS MÆHLUM, Norwegian University of Science and Technology, Norway
JIAN-JIA CHEN, Technical University of Dortmund, Germany
MARTIN SCHOEBERL, Technical University of Denmark, Denmark
LINH THI XUAN PHAN, University of Pennsylvania, USA
JERONIMO CASTRILLON, TU Dresden, Germany
SANJIT A. SESHIA, University of California, Berkeley, USA
EDWARD A. LEE, University of California, Berkeley, USA

To design performant, expressive, and reliable cyber-physical systems (CPSs), researchers extensively perform quasi-static scheduling for concurrent models of computation (MoCs) on multi-core hardware. However, these quasi-static scheduling approaches are developed independently for their corresponding MoCs, despite commonality in the approaches. To help generalize the use of quasi-static scheduling to new and emerging MoCs, this paper proposes a *unified* approach for a class of deterministic timed concurrent models (DTCMs), including prominent models such as synchronous dataflow (SDF), Boolean-controlled dataflow (BDF), scenario-aware dataflow (SADF), and Logical Execution Time (LET). In contrast to scheduling techniques tailored exclusively to specific MoCs, our unified approach leverages a common *intermediate* formalism called state space finite automata (SSFA), bridging the gap between high-level MoCs and executable schedules. Once identified as DTCMs, new MoCs can directly adopt SSFA-based scheduling, significantly easing adoption.

*These authors contributed equally to this research.

Authors' Contact Information: Shaokai Lin, University of California, Berkeley, Berkeley, USA; Erling Jellum, University of California, Berkeley, Berkeley, USA; Mirco Theile, TUM School of Engineering and Design, Technical University of Munich, Munich, Germany; Tassilo Tanneberger, TU Dresden, Dresden, Germany; Binqi Sun, TUM School of Engineering and Design, Technical University of Munich, Munich, Germany; Chadlia Jerad, University of Manouba, Manouba, Tunisia; Yimo Xu, University of California, Berkeley, Berkeley, USA; Guangyu Feng, University of California, Berkeley, Berkeley, USA; Magnus Mæhlum, Norwegian University of Science and Technology, Trondheim, Norway; Jian-Jia Chen, Technical University of Dortmund, Dortmund, Germany; Martin Schoeberl, Technical University of Denmark, Lyngby, Denmark; Linh Thi Xuan Phan, University of Pennsylvania, Philadelphia, USA; Jeronimo Castrillon, TU Dresden, Dresden, Germany; Sanjit A. Seshia, University of California, Berkeley, Berkeley, USA; Edward A. Lee, University of California, Berkeley, Berkeley, USA.

EMSOFT 2025, Taipei, Taiwan

ACM ISBN 978-x-xxxx-xxxx-x/YYYY/MM
https://doi.org/XXXXXXX.XXXXXXX

We show that quasi-static schedules facilitated by SSFA are provably free from timing anomalies and enable straightforward worst-case makespan analysis. We demonstrate the approach using the reactor model—an emerging discrete-event MoC—programmed using the Lingua Franca (LF) language. Experiments show that quasi-statically scheduled LF programs exhibit lower runtime overhead compared to the dynamically scheduled LF programs, and that the analyzable worst-case makespans enable compile-time deadline checking.

CCS Concepts: • **Computer systems organization** → **Real-time systems**; **Embedded systems**; • **Theory of computation** → **Timed and hybrid models**.

Additional Key Words and Phrases: Quasi-Static Scheduling, Concurrency, DAG Scheduling, Predictability

ACM Reference Format:
Shaokai Lin, Erling Jellum, Mirco Theile, Tassilo Tanneberger, Binqi Sun, Chadlia Jerad, Yimo Xu, Guangyu Feng, Magnus Mæhlum, Jian-Jia Chen, Martin Schoeberl, Linh Thi Xuan Phan, Jeronimo Castrillon, Sanjit A. Seshia, and Edward A. Lee. 2025. Quasi-Static Scheduling for Deterministic Timed Concurrent Models on Multi-Core Hardware. In *Proceedings of EMSOFT (EMSOFT 2025).* ACM, New York, NY, USA, 25 pages. https://doi.org/XXXXXXX.XXXXXXX

1 Introduction

Modern cyber-physical systems (CPSs) aim to strike a balance between expressiveness, performance, and reliability. As use cases and operating environments become increasingly complex—for example, driving an autonomous vehicle in a densely populated city or flying the Ingenuity Mars Helicopter autonomously over complex Martian terrains—CPS designers aim to give the systems expressive behaviors to handle such complexity. However, certain use cases that demand rich behaviors pose stringent requirements on performance.

Despite steady progress, using multi-core architectures in CPSs remains challenging. For example, predicting timing properties on these architectures, such as a sequential program's worst-case execution time (WCET), still is an active research area [30]. Furthermore, concurrent programming is known to be notoriously difficult, especially using threads [21]—still a mainstream programming model today. To balance the tradeoffs between expressiveness, performance, and reliability, researchers have extensively studied concurrent models of computation (MoCs) and their formal properties. In particular, models that can either fully or partially derive schedules at compile-time receive continued attention, due to their attractive properties such as compile-time schedule validation and low runtime overhead. The technique of making most scheduling decisions at compile-time and only some absolutely necessary decisions at runtime is known as *quasi-static scheduling* [22]. Prominent models amenable to deriving schedules at compile-time include synchronous dataflow (SDF) [24], Boolean-controlled dataflow (BDF) [2], scenario-aware dataflow (SADF) [39], and logical execution time (LET) [19], just to name a few. In this work, we treat fully static scheduling of some models as a subset of quasi-static scheduling, which is the focus of this paper.

Challenges. For the aforementioned models, existing solutions on quasi-static scheduling show commonalities. However, these techniques appear isolated, and no prior work has looked at them through a unified lens. This fragmentation makes it difficult to understand their subtle differences, complicating systematic comparison and evaluation. When new MoCs emerge, it is unclear whether existing proven techniques can readily apply. Additionally, fragmented methods also lead to repeated, isolated implementations, hindering code reuse.

Problem Definition. In light of these challenges, we define the following research problems:

PD1 **No unified framework:** Existing quasi-static scheduling methods remain fragmented, with no unified framework available to systematically compare or integrate these techniques.

PD2 Undefined applicability: It remains unclear which exact classes of MoCs are suitable for unified quasi-static scheduling, complicating the adoption of existing methods into emerging MoCs.

PD3 Lack of demonstration: Even if a unified scheduling framework could theoretically apply, existing work does not provide concrete guidelines or examples demonstrating how such integration could occur in practice, particularly for newly emerging MoCs.

PD4 Uncertain effectiveness: The effectiveness and practical benefits of unified quasi-static scheduling approaches have not been demonstrated through systematic evaluation using concrete examples of emerging MoCs.

Our Solution. In this paper, we present the first *unified* approach to quasi-static scheduling for concurrent MoCs. Our approach generalizes lessons learned from SDF, BDF, SADF, and LET into a common scheduling framework based on a new intermediate formalism called state space finite automata (SSFA). This unified SSFA-based approach provides a frame of reference for comparing existing quasi-static scheduling techniques and helps identify a class of models to which the approach is applicable. We use the reactor model [28] as a concrete use case of our proposed scheduling approach and demonstrate an implementation in the reactor model's code generator—the Lingua Franca [27] compiler. Our source code and evaluation artifacts are publicly available online. [1]

Concretely, this paper presents the following key contributions:

(1) We present the first unified approach for quasi-static scheduling based on an intermediate formalism called state space finite automata (SSFA) (Sec. 4.2), which bridges the gap between high-level MoCs and low-level executable schedules (Sec. 2).
(2) We further define deterministic, timed, concurrent models (DTCMs), a class of MoCs on which SSFA-based scheduling applies. We discuss key properties of our proposed technique, including anomaly-free timing (without requiring WCETs) and analyzable worst-case makespan (Sec. 4).
(3) We concretely demonstrate how our unified SSFA-based scheduling approach applies to the reactor model and integrate it directly into its existing toolchain—the Lingua Franca (LF) compiler (Sec. 5).
(4) We evaluate the effectiveness of our approach by measuring the performance of quasi-statically scheduled LF programs against that of the default, dynamically scheduled LF programs using a set of benchmarks written in a synchronous subset of LF. Our results demonstrate that quasi-statically scheduled LF programs reduce runtime overhead, achieving an average improvement of 9 times (Sec. 6.1). In addition, we evaluate SSFA's analyzable worst-case makespan through a case study on a satellite attitude controller (Sec. 6.2).

2 Background and Related Work

2.1 Concurrent Models of Computation

A model of computation (MoC) is a mathematical abstraction of a computing device [16]. An MoC provides a formal framework either for designing or for reasoning about a computing device. For example, an MoC, such as Communicating Sequential Processes [12], can be used to design a signal processing algorithm, which together with a compiler and a runtime can result in an executable. However, the same MoC can also be used to model the set of all possible behaviors of an algorithm and to formally verify statements about the algorithm. In this work, we are concerned with the former, i.e. using MoC as a mechanism to design a computation, and in particular we present a methodology for producing quasi-static schedules from various MoCs.

[1]Paper artifacts: https://github.com/icyphy/emsoft25

MoCs are distinguished by their mechanisms for modeling concurrency and time. Some MoCs, such as the Finite State Machine and the Turing Machine do not include primitives for modeling concurrent processes and are known as Sequential MoCs. Concurrent MoCs are increasingly popular because the end of Dennard scaling has spawned the age of multi-cores, and recently, even low-power microcontrollers, such as the ESP32, have adopted multi-core architectures. Concurrent MoCs are naturally better suited for mapping onto parallel hardware.

In this work we will discuss the generation of executable schedules from various forms of Dataflow Process Networks [1], where computation is expressed as a set of connected actors with firing rules; Logical Execution Time [19], a task model with precedence constraints among tasks and where each task has a logical execution time; and reactors [28], where computation is expressed as a set of event-triggered actors with discrete-event semantics.

2.2 Quasi-Static Scheduling for Concurrent MoCs

In the literature of concurrent MoCs, the ability to derive schedules at compile-time from MoC semantics has long been a key topic of interest. Lee and Messerschmitt [24] propose the synchronous dataflow (SDF) model, where each concurrent actor specifies the number of tokens received from a channel to trigger a firing, as well as the number of tokens sent into a channel after a firing. By explicitly giving token production and consumption rates, SDF enables efficient static scheduling, which begins by solving a balance equation representing token distributions in the system [23]. Intuitively, each iteration of the static schedule must return the distribution of tokens to the starting state, effectively completing a hyperperiod.

SDF enables fully static compile-time scheduling, at the cost of rigidity in its behavior. To enable more dynamicity without entirely sacrificing analyzability, Buck and Lee [2] introduce the Boolean-controlled dataflow (BDF) model, which provides `SWITCH` and `SELECT` actors that accept Boolean tokens that route token flow and modulate token production rate at runtime. Given the dynamic nature of BDF, only certain BDF systems can yield compile-time schedules, subject to constraints such as observability of the Boolean token stream that controls token routing [3]. A hyperperiod is similarly obtained from solving a (now symbolic) balance equation.

Another MoC that aims to extend SDF with dynamism is scenario-aware dataflow (SADF) [6]. SADF models a system using multiple SDF graphs with nondeterministic transitions between them at the end of an SDF graph's iteration. To implement SADF, Kampenhout et al. [41] borrows the `SWITCH` and `SELECT` actors to perform scenario change and proposes using a rolling static-order schedule, which at runtime concatenates the static schedule of the next scenario (which itself is an SDF graph) to a rolling schedule as soon as the next scenario is determined. This approach exemplifies quasi-static scheduling since each scenario's schedule is determined at compile-time, while the switching between scenarios is determined at runtime.

Dataflow models are typically suitable for streaming applications such as digital signal processing. For cyber-physical systems, which benefit from the ease of specifying precise timing behavior, the Logical Execution Time (LET) model [7, 19] has gained traction. The LET model defines deterministic system behavior by specifying that tasks appear to execute instantaneously at logical time boundaries, while their actual computation is scheduled and completed within predefined time windows. LET is known to be amenable to static scheduling [11] based on the hyperperiod of the LET tasks [5] and precedence constraints [13].

While there has been work on surveying and comparing MoCs–see, e.g., a great recent survey from Roumage et al. [34], to our best knowledge, our work provides the first unified framework on quasi-static scheduling.

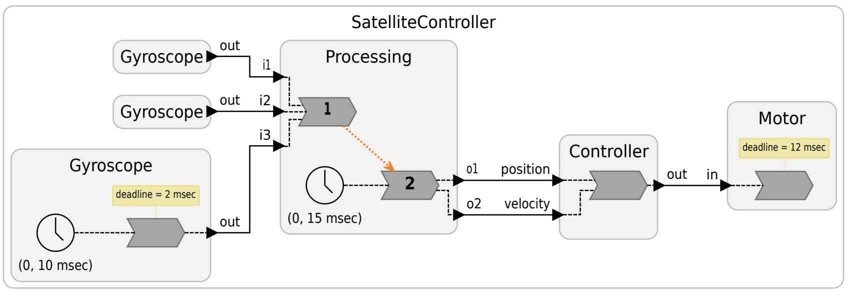


Fig. 1. A satellite attitude control system

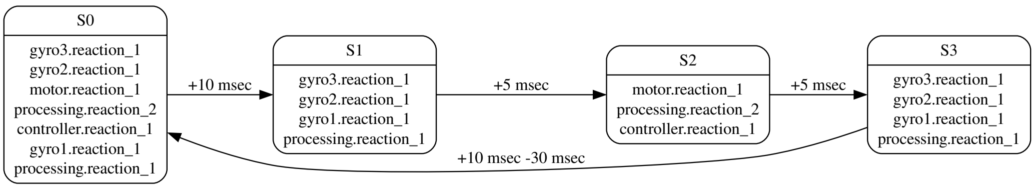


Fig. 2. Hyperperiod of the satellite attitude controller in Fig. 1

3 Running Example

Fig. 1 shows the block diagram of a satellite attitude control system. The three `Gyroscope` actors each sample a physical, or simulated, gyroscope sensor and are triggered every tenth millisecond. The `Processing` actor combines a sample from each of the three gyroscopes into an average. The `Processing` actor also produces a position and velocity estimate every 15th millisecond, which is fed into the `Controller` actor implementing a PID controller. Finally, the control output is forwarded to the `Motor` actor, controlling a physical (or simulated) stepper motor attached to a reaction wheel.

Attitude control of a satellite is clearly a real-time problem and the timeliness of the interaction between software and hardware is paramount. The program has two annotated deadlines. First, the `Gyroscope` actors have a 2-millisecond deadline on the reaction sampling of the sensors. This means that the code sampling the sensors should execute within 2 milliseconds from the time that the timer triggers. Second, the `Motor` actor has a 12-millisecond deadline on the reaction driving the stepper motor, meaning that this reaction must be invoked within 12 milliseconds of the triggering of the 15-millisecond timer in the `Processing` actor. This periodic program can conveniently be expressed in a DTCM, such as SDF, BDF, SADF, or LET.

4 Quasi-Static Scheduling based on SSFA

In Sec. 2.2, prior quasi-static scheduling work reveals an important pattern: generating schedules at compile-time typically requires finding the system's hyperperiod, or multiple hyperperiods if the system has dynamic behaviors (e.g., BDF or SADF). A hyperperiod encodes a sequence of system state transitions in which the system returns to its starting state at the completion of the sequence. For example, Fig. 2 shows the hyperperiod of the running example. Here, a state is defined by an

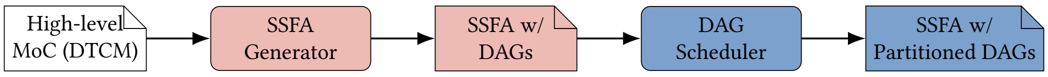


Fig. 3. General quasi-static scheduling workflow based on SSFA. The colors, red and blue, refer to two main stages in the workflow: SSFA generation (Sec. 4.3) and DAG scheduling (Sec. 4.4).

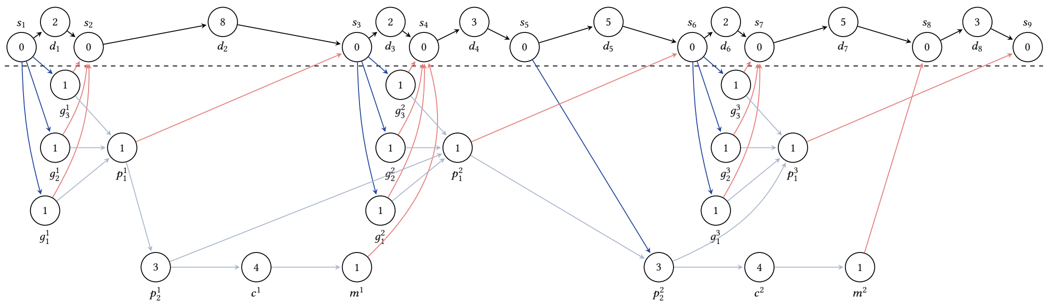


Fig. 4. DAG representation after conversion from the hyperperiod in Fig. 2, showing release edges in blue, deadline edges in red, and data dependency edges in gray, with the virtual path above the dashed line. The node names are the initials of the component names. The numbers in the nodes are the WCET bounds of each task, and the nodes above the dashed line are the nodes on the virtual path. Sync nodes are identified by s, marking the release and deadline of tasks; dummy nodes are identified by d, marking time intervals between sync nodes. Details of virtual nodes are introduced in Sec. 4.1.

ID and a set of task invocations, and a state transition is represented by time advancements, a representation common for time-triggered models like LET. We note that if the running example is explicitly specified in a dataflow model, e.g., SDF, a hyperperiod similar to Fig. 2 can still be identified by techniques such as Ghamarian et al. [8].

Fig. 3 shows our generalized quasi-static scheduling methodology. The input to the methodology is a system specified in a concurrent MoC, such as the MoCs mentioned in Sec. 2.2. From the system specification, a state space finite automaton (SSFA) is constructed from the semantics of the MoC. Intuitively, an SSFA is an FSM in which each node has an associated directed acyclic graph (DAG) representing task executions in a hyperperiod. The FSM further specifies guarded transitions that can be taken at the end of a DAG's execution.

In this section, we define DAGs and SSFA, and discuss how to generate both of them from a class of MoCs called deterministic timed concurrent models (DTCMs).

4.1 Directed Acyclic Graphs (DAGs)

DAGs are often used in real-time applications, such as automotive and avionics, to model real-time computing tasks and their precedence constraints. This subsection introduces basic concepts and notations for real-time DAG task modeling.

Example. Fig. 4 shows the DAG generated from the system hyperperiod in Fig. 2.

Task Model. A DAG task set is characterized by $(\mathcal{G}, P) = DAG \in DAGs$, in which $\mathcal{G}$ is a graph that defines the set of tasks and their precedences, and P denotes the period of the DAG task set. The graph $\mathcal{G}$ consists of $(\mathcal{V}, \mathcal{E})$, where $\mathcal{V} = \{v_i\}$ is a set of n nodes representing n tasks, and $\mathcal{E} = \{e_{ij}\}$ is a set of directed edges representing the precedence relation between the tasks. For any two nodes v_i and v_j connected by a directed edge e_{ij}, v_j can start execution only if v_i has finished its execution. Given the edge e_{ij}, v_i is the predecessor of v_j. Each task v_i is a non-preemptable sequential computing workload, with its WCET denoted as C_i.

Node-level Timing Attributes. The timing constraints of each node can be defined through four attributes: earliest starting time (EST), earliest finishing time (EFT), latest starting time (LST), and latest finishing time (LFT). The EST is the earliest time a node can start executing, equaling the maximum of its predecessors' EFTs. Similarly, the LFT defines the latest time a node can finish its execution to meet the deadline, equal to the minimum of its successors' LST. Note that EST, EFT, LST, and LFT are defined based on the precedence relations defined by $\mathcal{E}$ by *assuming that the execution time of a task is equal to its WCET.* If v_i has no precedessor, then $t_i^{\mathrm{EST}} = 0$. If v_i has no successor, then $t_i^{\mathrm{LFT}} = P$. Furthermore, we have

$$\begin{cases} t_i^{\mathrm{EST}} = \max\{t_j^{\mathrm{EFT}} \mid e_{ji} \in \mathcal{E}\}, \\ t_i^{\mathrm{LFT}} = \min\{t_j^{\mathrm{LST}} \mid e_{ij} \in \mathcal{E}\}, \\ t_i^{\mathrm{EFT}} = t_i^{\mathrm{EST}} + C_i, \\ t_i^{\mathrm{LST}} = t_i^{\mathrm{LFT}} - C_i. \end{cases} \tag{1}$$

These timing attributes can be computed iteratively using a fixed-point approach, following the classic ASAP (as soon as possible) and ALAP (as late as possible) analysis [33]. Specifically, all nodes are initially assigned $t_i^{\mathrm{EST}} = 0$ and $t_i^{\mathrm{LFT}} = P$, where P is the period of the DAG task set. At each iteration, values are updated according to the equations above until convergence, i.e., when no values change between successive iterations. Since the critical path of the DAG contains at most n nodes, convergence is guaranteed within n iterations [43]. The overall time complexity of this computation is $O(n^3)$.

Virtual Path. When mapping a multi-rate taskset to a single-rate DAG, job-level timing constraints need to be expressed. For example, the second reaction of the gyroscope 1, g_1^2, should only be able to start in the second gyroscope period, i.e., after 10 ms. Additionally, that gyroscope reaction has a deadline of 2 ms from its start, i.e., at 12 ms. To enforce these timing constraints, we borrow the concept of *virtual nodes* from [43]. There are two types of virtual nodes, *dummy nodes* with a predefined execution time C_{d_j} for dummy node d_j and *sync nodes* with zero execution time. A *virtual path* (the top row in Fig. 4) can be constructed using the virtual nodes, alternating S sync and $S-1$ dummy nodes. The total execution time of the virtual path is equal to the period of the DAG task set, i.e., $\sum_{j=1}^{S-1} C_{d_j} = P$. Through the defined execution time of the dummy nodes, it follows that for each sync node s_l,

$$t_{s_l}^{\mathrm{EST}} = t_{s_l}^{\mathrm{EFT}} = t_{s_l}^{\mathrm{LST}} = t_{s_l}^{\mathrm{LFT}} = \sum_{i=1}^{l-1} C_{d_i} = P - \sum_{j=l}^{S-1} C_{d_j}. \tag{2}$$

The nodes in the virtual path can be designed such that all unique offset and deadline values are represented by a sync node. Adding an edge from the appropriate sync node s_l to a task v_k, or vice-versa, respectively, results in the relation

$$O_k = t_{s_l}^{\mathrm{EFT}} \le t_k^{\mathrm{EST}} \quad \text{or} \quad t_k^{\mathrm{LFT}} \le t_{s_l}^{\mathrm{LST}} = D_k, \tag{3}$$

which enforces the timing constraints within the DAG task set.

For the offset of the second reaction of gyroscope 1, g_1^2, the sync node s_3 is created, which is preceded by two dummy nodes with a total execution time of 10 ms. By adding a precedence constraint from s_3 to g_1^2, that gyroscope reaction cannot start before 10 ms, enforcing its offset. Similarly, for its deadline constraint, sync node s_4 is created, which is succeeded by multiple dummy nodes with a total execution time of 18 ms. By adding a precedence constraint from g_1^2 to s_4, the gyroscope reaction needs to finish at the latest 18 ms before the hyperperiod, i.e., before 12 ms.

Trivial Schedulability. A DAG task $(\mathcal{G}, P)$ is defined to be *trivially schedulable* on M processors, if its makespan is shorter or equal to its period, and if its width is less or equal to the number of available processors [40]. Sun et al. [40] proved that a trivially schedulable DAG task is schedulable under any work-conserving executor (Lemma IV.3 and Lemma IV.4 in [40]). Additionally, they showed that any schedulable DAG task can be transformed into a trivially schedulable DAG task by adding edges to it. This transformation leads to the formulation of the *Edge Generation Scheduler* (EGS), in which edges are added iteratively to a DAG until it is trivially schedulable.

4.2 State Space Finite Automata (SSFA)

An SSFA is an extended finite state machine designed specifically for representing the behavior of concurrent MoCs. Formally, an SSFA is defined as a tuple:

$$SSFA = (M, V, D, T, G, m_0),$$

where:

- M is a finite set of modes,
- V is a finite set of variables, with cross product of their domains represented as $\mathcal{V} = \prod_{v \in V} \text{domain}(v) = \text{domain}(v_1) \times \text{domain}(v_2) \times \cdots \times \text{domain}(v_n)$,
- $D : M \rightarrow DAGs$ is a function assigning a directed acyclic graph (DAG) to each mode $m \in M$,
- $T \subseteq M \times M$ is a set of transitions between modes,
- $G : T \times \mathcal{V} \rightarrow \mathbb{B}$ is a guard function mapping the transition and variables to a boolean condition,
- $m_0 \in M$ is the initial mode.

The execution semantics of an SSFA are as follows:

(1) The SSFA begins execution in the initial mode m_0.
(2) When the SSFA enters a mode $m \in M$, it executes the DAG specified by $D(m)$ to completion.
(3) Upon completion of the DAG execution, the guard conditions associated with the outgoing transitions of the current mode are evaluated using G.
(4) If a guard condition evaluates to true, the corresponding transition is taken, and the SSFA moves to the next mode. If several guards are true simultaneously, one of the enabled transitions between modes is taken nondeterministicly.

The behavior of the running example can be represented by a trivial SSFA with a single mode containing the DAG shown in Fig. 4 with a transition back to itself that has a default guard *true*. In Sec. 4.3, we show how the SSFA for dynamic models like BDF and SADF can have more modes with non-trivial guards. We further show a non-trivial SSFA in Fig. 11 in Sec. 5.2.

4.3 Mapping High-Level MoCs to SSFA

Existing Models to SSFA. We now discuss how to map the MoCs surveyed in Sec. 2.2 to SSFA.

SDF. Since SDF defines a unique recurring behavior, the resulting SSFA therefore has only a single mode with a trivial self-transition. The DAG within the SSFA mode can be obtained by running the algorithm in Appendix I of Sih [38].

BDF. Buck [3] notes that generating compile-time schedules requires the knowledge of exact values of the emitted Boolean tokens, and an "acyclic precedence graph" (DAG) can be constructed for each case of Boolean token values. To generate a corresponding SSFA, each possible case of Boolean token values produces an SSFA mode, with the corresponding DAG constructed from a resulting SDF graph after filling in the concrete case of Boolean values. The modes have all-to-all transitions so that any mode can be switched to at runtime based on the upcoming Boolean tokens.

SADF. Each scenario becomes an SSFA mode. The SDF graph within each scenario is similarly transformed into a DAG based on Sih [38]. Transitions between SADF scenarios become SSFA mode transitions, and transition guards in SSFA are derived from the SADF model.

LET. The LET tasks are by design periodic, and the schedule repeats after each iteration of the hyperperiod of all task periods. This behavior results in a unique SSFA mode. A DAG can be constructed from the time and data dependencies specified among the LET tasks.

From lessons learned from existing MoCs, we summarize a list of observations these MoCs satisfy in order to map to SSFA.

(1) The MoC defines a finite set of execution phases.
(2) A phase often contains a hyperperiod, i.e., a unique partial order of task firings that eventually returns the system state to a starting state.[2]
(3) The MoC defines transitions, either deterministic or non-deterministic, between phases.

SDF and LET define a unique execution mode given the static nature of their task firings, while SADF defines a mode for each scenario. BDF defines a mode for a unique value set of its Boolean control tokens. The execution within a mode is fully deterministic, while mode switching depends on runtime conditions and can be non-deterministic, as SADF shows.

Deterministic Timed Concurrent Models (DTCMs). From the above requirements, we identify a class of MoCs, named Deterministic Timed Concurrent Models (DTCMs), where the proposed SSFA-based scheduling approach applies. A DTCM's execution can be represented as a finite set of *phases* P. Each phase $p \in P$ is a finite sequence of states $(s_i, s_{i+1}, ..., s_{i+n}) \in S^n$. The exact definition of a *state* $s \in S$ should be provided by the DTCM, and S denotes the set of all possible system states defined by the DTCM. Within each phase p, the evolution of system states is described by a *deterministic* state-transition relation $T_p \subseteq S \times S$, where each transition $(s_i, s_j) \in T_p$ takes the system from a state s_i to a unique successor state s_j. For example, SDF typically defines the state to be a distribution of tokens in the channels. In Sec. 5, we show another example of state definition for the reactor model. Transitions between phases are also defined by the DTCM, and they can be nondeterministic. In SADF, nondeterministic transitions between scenarios exemplify nondeterministic phase transitions [6].

Although there are no general rules for identifying phases for a DTCM's execution, a classical approach involves explicit state space exploration [2, 8, 26] to identify *hypercycles*. A hyperperiod h is a sequence of states $h = (s_i, s_{i+1}, ..., s_{i+n})$ such that the next state s_{i+n+1} returns to s_i, assuming no phase transition occurs. And we use the prefix "hyper-" to emphasize that, similar to the notion of hyperperiod—the least common multiple of the periods of all periodic tasks, we aim to find a periodic sequence of states considering *all* the periods of periodic tasks. To identify a hyperperiod, the DTCM defines a predicate $\mathcal{H} : S \times S \longrightarrow \mathbb{B}$ that determines whether the second state has returned to the first state, completing a hyperperiod. Then it unrolls from a starting state s_0 until a minimal hyperperiod $h = (s_i, s_{i+1}, ..., s_j)$ is found such that $\mathcal{H}(s_i, s_{j+1}) = \textit{true}$ for some $j \geq i \geq 0$. This process yields a sequence of states forming a lasso structure. The leading transient states can thus be categorized into one phase $p_0 = (s_0, s_1, ..., s_{i-1})$, and the remaining hyperperiod into another phase $p_1 = (s_i, s_{i+1}, ..., s_j)$. For certain DTCMs, more efficient state space exploration techniques exist, such as solving SDF's balance equations [24]. In general, the DTCM should supply a technique to obtain the minimal hyperperiod based on its semantics.

Once the phase set P is obtained, the DTCM should provide a method to convert the task firings involved in each $p \in P$ to a DAG, as described earlier in Sec. 4.1. A specific example for the reactor model is provided in Sec. 5. Once a DAG for a phase p is constructed, a corresponding SSFA mode can be produced—this process is repeated for each phase $p \in P$. Lastly, transitions between SSFA modes can be added based on guarded transitions defined by the DTCM semantics.

[2]We will later see that a sequence of non-recurring states can also form a phase, as in the case of reactors in Sec. 5.1.

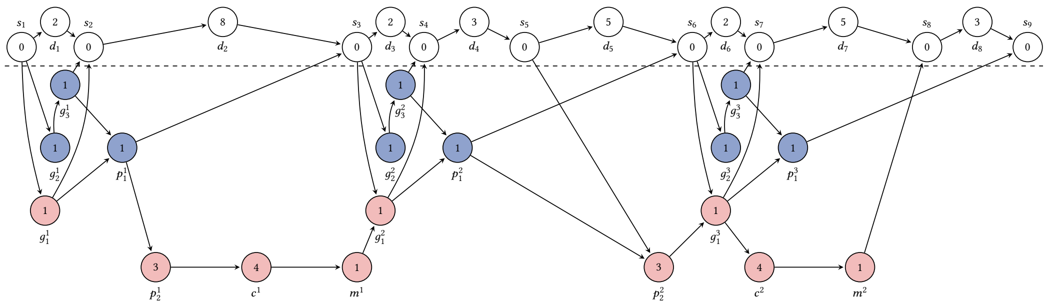


Fig. 5. Sample output from a DAG scheduler, a transitive reduction of the DAG in Fig. 4 with width $w = 2 + 1$, with +1 for the virtual path. The colors of the nodes show a valid partitioning to 2 available cores.

4.4 Scheduling SSFA by Graph Partitioning

To schedule the SSFA, a partitioned DAG scheduler is invoked on each DAG contained in the SSFA states. The DAG scheduler takes an unpartitioned DAG as input, partitions the DAG based on the number of processor cores available, and outputs graph partitions, which is a spatial mapping between tasks and execution platforms with timing and inter-task dependencies. Given an unpartitioned DAG, a DAG scheduler can designate nodes to any partition it deems fit. The scheduler can modify edges if the output partitioned DAG satisfies all dependencies in the input unpartitioned DAG. In addition, the DAG scheduler must *sequentialize* each partition by adding edges such that there exists a path in the output DAG passing through all nodes within the same partition. The resulting DAGs partitioned by this type of DAG schedulers are *trivially schedulable*, as defined by Sun et al. [40] and discussed in Sec. 4.1. A trivially schedulable DAG can easily be partitioned using paths that cover the DAG, as detailed in Algorithm 1 of [40], with an example shown in Fig. 5.

We note that, in real-time scheduling literature, there is a large body of prior work on partitioned scheduling [42]. And in principle, any DAG partitioning algorithm could be plugged into this step, as long as there is post-processing that guarantees each partition is sequentialized. To execute a partitioned DAG, there is prior work as well, e.g., Huang et al. [14] and Henzinger et al. [10]. In our demonstration of the reactor model in Sec. 5, we use a strategy similar to Henzinger et al. [10].

After scheduling all DAGs in the SSFA, we have now generated the quasi-static schedules for the DTCM. Compared to prior work's static or quasi-static schedules (such as SDF's acyclic precedence graphs [23] and LET's table of possible release timing [9]), the SSFA formalism is operationally equivalent, in the sense that task dependencies and task-to-core mappings are preserved. However, the SSFA-based representation differs in structure and expressiveness: instead of representing a schedule as a flat precedence graph (as in SDF) or a time-triggered release table (as in LET), SSFA encodes scheduling decisions within the modes of an extended finite-state machine, with a DAG in every mode and transitions guarded by logical conditions that define when the system evolves to the next mode.

4.5 Formal Properties

In this section, we discuss formal properties provided by our SSFA-based scheduling approach.

Anomaly-Free Timing. A timing anomaly in real-time systems refers to a situation where a local shorter execution time (like a cache hit) leads to an unexpected global effect of an increase in the overall execution time [20, 29]. Timing anomalies can occur with the dynamic scheduling of tasks (reactions) on multiple execution units. We now show that our proposed SSFA-based quasi-static scheduling guarantees that no timing anomalies will occur on multi-core processors. It is important to note that the monotonous property holds *without* requiring to know WCET bounds.

Example. Fig. 6 shows two scenarios that can cause scheduling anomalies under work-conserving scheduling for non-trivially schedulable DAGs. The first row shows a DAG task and a corresponding schedule using a work-conserving scheduler. The DAG on the right is modified to be trivially schedulable by adding two edges, yielding the same schedule as on the left. In the second row, we consider a scenario where the execution time of v_2 is decreased by 0.2 ms, causing it to finish earlier than v_1. As a result, v_5 can be started, delaying the execution of v_4 and v_6, leading to a makespan increase of 0.8 ms. This scenario is the typical scheduling anomaly example where the decrease of an individual execution time leads to an increase in the overall makespan. On the right, in the trivially schedulable DAG, the execution order is not altered since v_5 needs to wait for v_3 to finish, and the processor stalls until v_4 is ready. Thus, the makespan does not increase but decreases by 0.1 ms instead.

The third row shows that the same reordering in the work-conserving schedule can be triggered by increasing an individual execution time, causing a larger increase in the overall makespan. By increasing the execution time of v_1 by 0.2 ms, the same reordering occurs, causing an increase of the makespan of 1 ms. For the trivially schedulable DAG, the increase in individual execution time also does not lead to reordering, and the makespan only increases by 0.1 ms.

Formal definition. Consider a DAG task set $\mathcal{G}(\mathcal{V}, \mathcal{E})$ with estimated execution times $\mathbf{C} \in \mathbb{R}^{|\mathcal{V}|}$ associated with each node $v_i \in \mathcal{V}$ and a scheduling algorithm yielding an expected makespan M if all nodes execute with their respective C_i. Suppose a node executes with a different computation time $\bar{C_i} = C_i + \Delta_i$. A scheduling algorithm is *monotonous* if the makespan, denoted as M, resulting from the actual computation time satisfies $M(\bar{\mathbf{C}}) \leq M(\mathbf{C}) + \max\{\Delta_i, 0\}$.

Theorem 4.1 (Monotonicity of trivially schedulable DAG tasks). *When executing a trivially schedulable DAG task in a work-conserving manner, the monotonicity property is guaranteed.*

Proof. The makespan of a trivially schedulable DAG is equal to the length of its longest path, i.e., $M(\mathbf{C}) = \max_{p\in\mathcal{P}} L(p, \mathbf{C})$. The length of a path $p \in \mathcal{P}$ under execution times $\mathbf{C}$ is $L(p, \mathbf{C}) = \sum_{v_i\in p} C_i$. Under the new execution times $\bar{\mathbf{C}}$, the lengths of the paths in the DAG change as follows:

$$L(p, \bar{\mathbf{C}}) = \begin{cases} L(p, \mathbf{C}) + \Delta_i, & \text{if } v_i \in p, \\ L(p, \mathbf{C}), & \text{otherwise.} \end{cases} \tag{4}$$

Therefore, $M(\bar{\mathbf{C}}) \leq \max\{M(\mathbf{C}) + \Delta_i, M(\mathbf{C})\} = M(\mathbf{C}) + \max\{\Delta_i, 0\}$. □

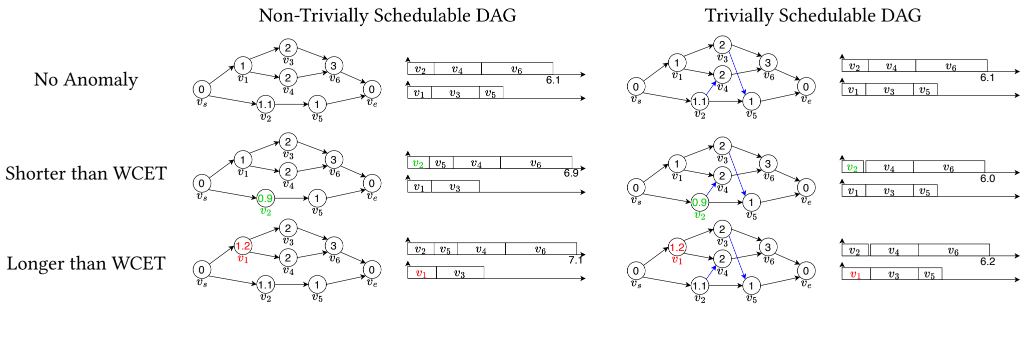


Fig. 6. Scheduling anomaly example.

Corollary 4.1.1. *If multiple nodes, denoted as N, alter their execution times, the resulting makespan is bounded by* $M(\bar{\mathbf{C}}) \leq M(\mathbf{C}) + \sum_{v_i \in N} \max\{\Delta_i, 0\}$.

Corollary 4.1.2. *Since any partitioned schedule with predefined sequentialization on each processor can be expressed as a trivially schedulable DAG (see Theorem IV.5, necessity part in [40]), any work-conserving partitioned scheduling algorithm is monotonous.*

Monotonicity implies two properties: (i) If the execution time of the nodes is less than their estimated execution time, the makespan does not increase. This property is often called *sustainability* [4] and guarantees the absence of timing anomalies. (ii) If the nodes exceed their execution times by some delta, the increase of the overall makespan is bounded by the sum of the deltas. This property is helpful for schedulers to decide on the action taken when deadlines are exceeded.

Analyzable Worst-Case Makespan. A partitioned, trivially schedulable DAG produced by the proposed workflow yields a simple procedure to check the worst-case makespan up to some node or for the entire hyperperiod.

Definition 4.1 (Partial makespan up to node *n*)**.** The partial makespan up to a node *n*, denoted as $\bar{L}(n)$, is defined recursively as

$$\bar{L}(n) = \begin{cases} wcet(n) & \text{if } U(n) = \varnothing, \\ \max_{u \in U(n)} L(u) + wcet(n) & \text{otherwise.} \end{cases}$$

where $wcet : \mathcal{V} \longrightarrow \mathbb{T}$ returns the WCET of a DAG node within a time domain $\mathbb{T}$ and $U : \mathcal{V} \longrightarrow \mathcal{P}(\mathcal{V})$ maps a DAG node *n* to a set of its immediately upstream DAG nodes.

Example 1. *To validate the deadline of* 12*ms of the first invocation of the motor reaction in Fig. 1, we calculate the partial makespan up to* m^1 *in Fig. 5, i.e.,* $\bar{L}(m^1)$.

$$\begin{aligned}
\bar{L}(m^1) &= \bar{L}(c^1) + wcet(m^1) \\
&= \bar{L}(p_2^1) + wcet(c^1) + wcet(m^1) \\
&= \bar{L}(p_1^1) + wcet(p_2^1) + wcet(c^1) + wcet(m^1) \\
&= \max(\bar{L}(g_3^1), \bar{L}(g_1^1)) + wcet(p_1^1) + wcet(p_2^1) + wcet(c^1) + wcet(m^1) \\
&= \max(wcet(g_2^1) + wcet(g_3^1), wcet(g_1^1)) + wcet(p_1^1) + wcet(p_2^1) + wcet(c^1) + wcet(m^1) \\
&= \max(1+1, 1) + 1 + 3 + 4 + 1 = 11 \leq 12
\end{aligned}$$

Therefore, the deadline of 12*ms of the first invocation of the motor reaction can be guaranteed to hold.*

So far we have introduced a generic methodology to generate quasi-static schedules for a class of DTCMs and their formal properties. In the subsequent section, we demonstrate applying the methodology to a specific new DTCM—the reactor model—and show how to implement the methodology in concrete tooling.

5 Applying Methodology to the Reactor Model

In this section, we demonstrate an application of the general SSFA-based scheduling methodology to an emerging MoC—the reactor model [28], which has a front-end coordination language called Lingua Franca (LF) [27]. The reactor model in LF decouples the passage of physical time from logical time, except when interactions with the physical world are involved (through timers, sensors, and actuators). In part, because LF allows sporadic events, the scheduling of LF programs is carried out dynamically at runtime. While the ideal, logical timing behavior is specified in the LF code, alignment with physical time is executed in a best-effort way, and there are no guarantees about

```
       Model ::= TargetDecl Preamble ∗  Reactor+
  TargetDecl ::= target id
     Reactor ::= reactor id? Params? { ReactorBody ∗ }
      Params ::= ( Parameter ( , Parameter) ∗ )
 ReactorBody ::= Preamble | StateVar | Input | Output | Timer
               | Reaction | Instantiation | Connection
    StateVar ::= state id (: Type)? (= Expression)?
       Timer ::= timer id ( Expression( , Expression)? )
  Connection ::= VarRef -> VarRef (after Expression)?
Instantiation ::= id = new id ( Parameter ( , Parameter) ∗ )
     Literal ::= true | false | INT | FLOAT
    Reaction ::= reaction ( TriggerList? ) (-> EffectList)? Code

  Expression ::= Literal | Time | Code
 TriggerList ::= TriggerRef ( , TriggerRef)∗
  EffectList ::= VarRef ( , VarRef)∗
   Parameter ::= id : Type = Expression
       Input ::= input id (: Type)?
      Output ::= output id (: Type)?
        Type ::= time | id | Code
  TriggerRef ::= BuiltinTrigger | VarRef
      VarRef ::= id (. id)∗
    Preamble ::= Code
        Code ::= {= id* =}
```

Fig. 8. Syntax of the synchronous subset of LF.

the absence of timing anomalies on multicores or deadline compliance at the MoC level, since they are the concerns of LF's underlying scheduling policies. We refer the reader to Lohstroh et al. [27] for further details on the relationship between logical time and physical time. The running example in Fig. 1 is an automatically generated diagram from the LF toolchain.

We now apply our SSFA-based quasi-static scheduling technique to a *synchronous* subset of LF, with its syntax shown in Fig. 8. We mainly exclude two LF language features: actions and physical connections, effectively only allowing timer-driven tasks and event-driven tasks from logical connections. Fig. 7 shows the LF compiler embedding our SSFA-based approach.

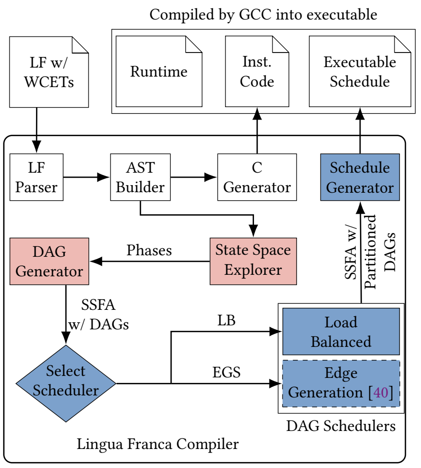
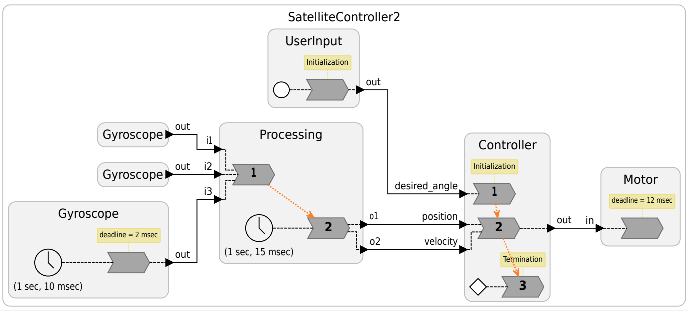


Fig. 7. LF compiler integrating the proposed approach in Fig. 3. Colored boxes are our work. Dashed lines denote external tools for which we build interfaces. Like Fig. 3, the red pipeline highlights SSFA generation, and the blue pipeline highlights DAG scheduling.

The user first provides an LF program with *optional* WCET annotations, each of which indicates the WCET of a reaction on a certain execution platform. WCETs, if specified, help guide the DAG schedulers to generate better schedules. In practice, this may not be known or the estimate may be inaccurate or pessimistic. Our procedure will still generate correct schedules, though timing optimality cannot be assured. Moreover, as we show in Sec. 4.5, the schedules are free from timing anomalies, in that if an execution time is less than the WCET estimate, the makespan is never worse.

The LF compiler then parses the program and builds an abstract syntax tree (AST). Once an AST is built, the compiler invokes a C code generator for generating program-specific instrumentation code, including custom C structs for reactor definitions in the specific program, user-specified reaction bodies written in C, memory



```
target C {
  workers: 2
}
reactor Gyroscope {
  timer t(1 s, 10 ms)
  output out: IntArr3
  @wcet("1 ms")
  reaction(t) -> out {=...=}
}
reactor Processing {
  input  i1: IntArr3
  input  i2: IntArr3
  input  i3: IntArr3
  output o1: IntArr3
  output o2: IntArr3
  timer t(1 s, 15 ms)
  @wcet("1 ms")
  reaction(i1,i2,i3) {=...=}
  @wcet("3 ms")
  reaction(t)
    -> o1, o2 {=...=}
}
reactor UserInput {
  output out: IntArr3
  reaction(startup)
    -> out {=...=}
}
reactor Controller {
  input position: IntArr3
  input velocity: IntArr3
  input desired_angle:IntArr3
  output out: IntArr3
  reaction(desired_angle)
    {= ... =}
  @wcet("4 ms")
  reaction(position,velocity)
    -> out {=...=}
  reaction(shutdown) {=...=}
}
reactor Motor {
  input in:IntArr3
  @wcet("1 ms")
  reaction(in) {=...=}
}
main reactor {

  gyro1      = new Gyroscope()
  gyro2      = new Gyroscope()
  gyro3      = new Gyroscope()
  userInput  = new UserInput()
  processing= new Processing()
  controller= new Controller()
  motor      = new Motor()

  gyro1.out -> processing.i1
  gyro2.out -> processing.i2
  gyro3.out -> processing.i3
  processing.o1 ->
    controller.position
  processing.o2 ->
    controller.velocity
  userInput.out ->
    controller.desired_angle
  controller.out -> motor.in

}
```

Fig. 9. Satellite attitude controller with user input and termination, written in the synchronous subset of LF. The C code inside reactions is omitted for conciseness.

allocation and deallocation functions, etc. These steps are captured in the white boxes of Fig. 7 and are not part of this work.

Our contribution begins with a *state space explorer*, which explores all *phase* of the LF program (the concept of phase is introduced in Sec. 4.3) and conditions for phase transitions. Then, a DAG generator transforms each hyperperiod to a corresponding DAG. Combined with the conditions of phase transitions, the red pipeline in Fig. 7 yields a fully specified SSFA. We now discuss SSFA generation in detail.

5.1 Generating SSFA from Reactors and Scheduling

To generate an SSFA from an LF program, the state space explorer identifies three execution phases of LF: *initialization*, *periodic*, and *shutdown*. The initialization phase contains a non-repeating sequence of state space nodes starting from the initial logical timestamp. The periodic phase contains a sequence of state space nodes that forms a hyperperiod. The shutdown phase contains a sequence of nodes that follows after the periodic phase until the end of execution.

To help identify a hyperperiod for reactors, we first define the notion of a system *state*, as required by DTCM in Sec. 4.3. The definition is similar to the one given by Lin et al. [26].

Definition 5.1 (Reactor System State). The system state s of an LF program is a tuple $s = (id^s, t^s, N, E)$, where $id^s \in ID$ is an identifier of the state, $t^s \in \mathcal{T}$ is a timestamp in a time domain, $N \subseteq \mathcal{N}$ is a set of reactions, and $E \subseteq \mathcal{E}$ is a set of pending events, where each event is $e = (id^e, t^e) \in E$ such that $id^e \in ID$ is an event identifier and $t^e \in \mathcal{T}$ is a timestamp.

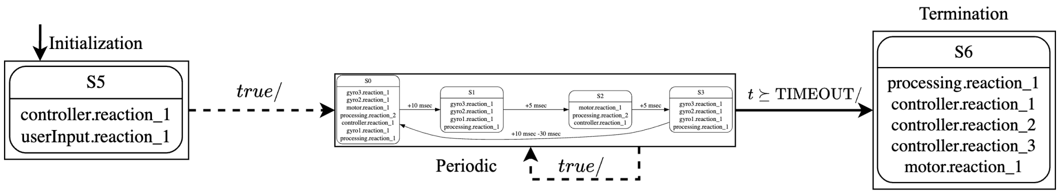


Fig. 10. State space exploration identifies phases of the updated satellite controller (Fig. 9) with guarded transitions. The full-size periodic phase is shown in Fig. 2.

Notation. Subseuqently, when referring to a particular element of s in a state sequence $(s_0, ..., s_n)$, we use the *subscript* to denote an element from a particular state in the sequence, e.g., N_0 is the element N of the state tuple s_0.

We now define the $\mathcal{H}$ function for reactors, as suggested in Sec. 4.3, to enable checking whether two states complete a hyperperiod.

Definition 5.2 (Hyperperiod Checking Function $\mathcal{H}$). A sequence of states $(s_0, ..., s_n)$ is said to form a hyperperiod, if the subsequent state s_{n+1} satisfies the following :

(1) $N_0 = N_{n+1}$. This means that the same sets of reactions should be invoked;
(2) $\forall(id_0^e, t_0^e) = e_0 \in E_0.[\forall(id_{n+1}^e, t_{n+1}^e) = e_{n+1} \in E_{n+1}.[id_0^e = id_{n+1}^e \land t_0^e - t_0^s = t_{n+1}^e - t_{n+1}^s]]$. This means that for all pending events in E_0 and E_{n+1}, not only their event identifiers need to be the same, but their relative time offsets to the current timestamps should also be the same.

If so, $\mathcal{H}(s_0, s_{n+1}) = true$, and $H(s_0, s_{n+1}) = false$ otherwise.

The LF state space explorer implements Algorithm 1 from [26], which constructs the initialization and periodic phase by unrolling a simulated execution from the initial startup events until either:

(1) there are no more pending events, i.e. $E = \varnothing$,
(2) or a hyperperiod is found based on $\mathcal{H}$ defined in Def. 5.2.

When this procedure is completed, the sequence of states simulated typically forms a lasso shape. The explorer factors out the nodes leading up to the loop, i.e., the stem of the lasso, into the initialization phase, and the loop nodes into the periodic phase.

To identify a hyperperiod for the shutdown phase, the explorer begins a new simulation run, assuming the shutdown triggers and all input ports are present in case some connection delivers an event at the same time as the shutdown time.

To show the above state exploration in action, in Fig. 9, we augment the running example with a `UserInput` reactor, which sends a desired satellite orientation to the controller when the LF program starts up (the white circle inside `UserInput` represents a `STARTUP` trigger). In addition, the `Controller` reactor has a third reaction triggered by a `SHUTDOWN` trigger, representing a termination procedure when the system shuts down. Fig. 10 now shows the three phases identified by the state space explorer with conditions for switching between phases. To construct a proper SSFA from this representation, a DAG needs to be constructed from each hyperperiod, a task performed by the DAG generator, which we discuss next.

Converting Reactor Phases to DAGs. For each phase of the LF program, a DAG is constructed based on Algorithm 1. The DAG generator traverses each state in a phase, generates a sync node (introduced in Sec. 4.1) from its logical timestamp, and generates a task node for each reaction invocation in the state.

Edges are drawn between task nodes and sync nodes based on the reactor semantics, including:

Algorithm 1 Generate a DAG from an LF Phase

```
1: procedure GenerateDag($phase$)
2:     $dag = (V, E) \leftarrow \{\emptyset, \emptyset\}$                          ▷ Initialize an empty DAG.
3:     $state \leftarrow phase.head$           ▷ Initialize a running variable to the head of the phase.
4:     while True do                                   ▷ Iterate over all states in the phase.
5:         BreakIfAllVisited($state$, $phase$)
6:         $time \leftarrow$ GetLogicalTime($state$)          ▷ Get the logical time of the current state.
7:         $syncNode \leftarrow$ CreateSyncNode($time$)
8:         $V \leftarrow V \cup \{syncNode\}$
9:         $reactions \leftarrow$ GetReactions($state$)
10:        $reactionNodes \leftarrow$ CreateTaskNodes($reactions$)
11:        $V \leftarrow V \cup reactionNodes$
12:        $E \leftarrow E \cup$ CreateSyncToTaskEdges($syncNode$, $reactionNodes$)
13:        $E \leftarrow E \cup$ CreateDataDependencyEdges($reactions$) ▷ Priorities, triggers, barrier sync.
14:        $state \leftarrow$ GetNextStateFromPhase($phase$, $state$)
15:    end while
16:    $V \leftarrow V \cup$ CreateFinalSyncNode($phase$, $dag$)
17:    $E \leftarrow E \cup$ CreateEdgesForUnconnectedTaskNodes($dag$)
18:    for all $reaction \in$ getReactionNodesWithDeadlines() do
19:        $deadlineNode$ = CreateDeadlineSyncNode($reaction$)
20:        $V \leftarrow V \cup \{deadlineNode\}$
21:        $E \leftarrow E \cup$ CreateTaskToDeadlineEdge($reaction$, $deadlineNode$)
22:    end for
23:    AddAndConnectDummyNodesBetweenSyncs($dag$)
24:    return $dag$
25: end procedure
```

(1) Release edges from sync nodes marking release times to released task nodes;
(2) Deadline edges that connect task nodes to sync nodes that mark their completion deadlines;
(3) Data dependency edges between nodes having data dependency constraints: (i) one reaction's output triggers a downstream reaction (e.g., in Fig. 4, the edge from p_2^1 to c^1); (ii) reaction priorities (e.g., the edge from p_1^1 to p_2^1); (iii) barrier synchronization within a reactor (e.g., the edge from p_2^1 to p_1^2, meaning that reaction 2 at a previous timestamp has to complete before releasing reaction 1 at a new timestamp).

The DAG generator produces additional sync nodes marking deadline on task completion. Lastly, it adds dummy nodes between sync nodes and use additional edges to connect all sync nodes to form the virtual path. For example, the DAG in Fig. 4 is generated based on the rules above from Fig. 2. More concretely, Fig. 2's `gyro3.reaction_1` in state $S0$ has a release time of $t = 0ms$ and a deadline of $t = 2ms$ (deadline not shown in Fig. 2), producing sync nodes s_1, s_2, and a dummy d_1 in Fig. 4. The resulting unscheduled SSFA after generating DAGs for all LF phases is shown in Fig. 11.

Scheduling each DAG by Partitioning. In our extended LF compiler, we support two different types of DAG schedulers: A load-balanced scheduler (LB) and an edge generation scheduler (EGS), by Sun et al. [40]. The load-balanced scheduler takes the generated DAG as its input and produces graph partitions that aim to assign each processor core the same amount of work measured in execution times. On the other hand, in case of the edge generation scheduler, instead of distributing workloads fairly across workers, the focus is on satisfying timing constraints. Given a DAG task set with a deadline, which can be the end of a hyperperiod, EGS checks whether the DAG task set is *trivially schedulable* on the given number of processors. If it is not trivially schedulable by construction, EGS adds edges to make it trivially schedulable. Once all DAGs in the SSFA is partitioned, the SSFA is considered scheduled and is ready to be compiled into an executable.

Scalability. We gauge at the scalability of our proposed approach through the lens of time complexity, which varies at different stages of our approach. First, during compile-time state space exploration, the explored state space for LF forms the shape of a lasso, which consists of a *stem* (the initial path leading into the loop) and a *loop* (the repeating cycle). In this case, the execution time of state space exploration scales linearly with the size of the stem and the loop, which are defined by the structure of the LF program. Next, when generating DAGs from explored phases (such as Fig. 10), the time complexity is $O(n)$, where n is the number of states in the explored phases, since Algorithm 1 linearly traverses through all states in a phase. Then, when partitioning the generated DAGs, the time complexity depends on the specific DAG partitioning algorithms, which [42] has a good overview of. The benefit of quasi-static scheduling is that the scheduling cost is paid at design-time. Once the executable schedule is generated (which we discuss in the next section), if the system has no dynamic behavior at runtime, the system operates with constant-time overhead: every core simply follows its precomputed schedule, yielding $O(1)$ per scheduling decision. A global dynamic scheduler, such as EDF, must account for all enabled tasks, which incurs $O(n)$ complexity for n concurrently enabled tasks.

5.2 Compiling Scheduled SSFA to Executable Schedules

In this section, we detail how a scheduled SSFA is further compiled into executable schedules.

Intermediate Representation (IR). From the partitioned DAGs, the schedule generator in Fig. 7 generates executable schedules, which are composed from an intermediate representation (IR) we design to direct how a CPU core should execute a DAG partition. The approach of using an IR for implementing real-time scheduling is pioneered by Henzinger et al. [10], which we take inspiration from. Table 1 shows the full IR for composing executable schedules. The IR combines elements from RISC-V [44], timing instructions from PRET Machines [17, 25, 45], and the reactor model's task abstraction. It includes standard control-flow and arithmetic instructions (`ADD`, `ADDI`, `BEQ`, `BGE`, `BLT`, `BNE`, `JAL`, `JALR`) for manipulating auxiliary state and managing schedule logic. Timing instructions (`DU`, `WU`, `WLT`) make timing a semantic aspect of execution, enabling real-time alignment and inter-worker synchronization. Reactor-inspired instructions include `EXE`, which invokes C functions (e.g., reaction bodies), and `STP`, which terminates execution.

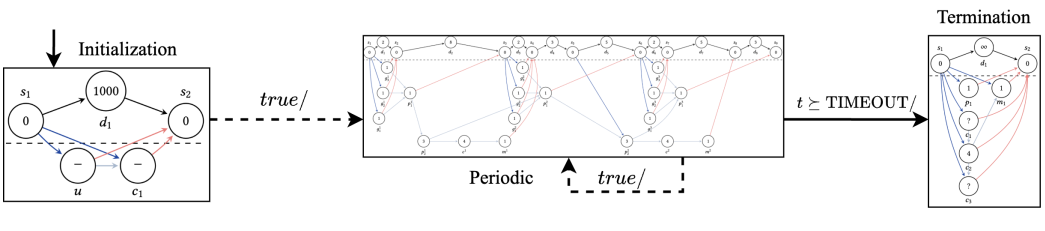


Fig. 11. SSFA of the updated satellite controller generated from Fig. 10. In the Initialization mode, Task nodes without WCETs are labeled "-". In the Termination mode, dummy nodes of "∞" indicate that tasks do not have timing constraints on completion.

Table 1. Intermediate Representation for Composing Executable Schedules

Instruction	**Description**
Basic instructions	*Inspired by RISC-V [44]*
`ADD op1, op2, op3`	Add to an integer variable (op2) by an integer variable (op3) and store to a destination variable (op1).
`ADDI op1, op2, op3`	Add to an integer variable (op2) by an immediate (op3) and store to a destination variable (op1).
`BEQ op1, op2, op3`	Take the branch (op3) if the op1 variable value is equal to the op2 variable value.
`BGE op1, op2, op3`	Take the branch (op3) if the op1 variable value is greater than or equal to the op2 variable value.
`BLT op1, op2, op3`	Take the branch (op3) if the op1 variable value is less than the op2 variable value.
`BNE op1, op2, op3`	Take the branch (op3) if the op1 variable value is not equal to the op2 variable value.
`JAL op1 op2`	Store the return address to op1 and jump to a label (op2).
`JALR op1, op2, op3`	Store the return address to op1 and jump to a base address (op2) + an immediate offset (op3).
Timing instructions	*Inspired by PRET Machines [17, 25, 45]*
`DU op1, op2`	Delay until the physical clock reaches a base timepoint (op1) plus an offset (op2).
`WLT op1, op2`	Wait until a variable value (op1) to be less than a desired value (op2).
`WU op1, op2`	Wait until a variable value (op1) to be greater than or equal to a desired value (op2).
Task instructions	*Inspired by the reactor model [28]*
`EXE op1, op2`	Execute a function (op1) with argument (op2).
`STP`	Stop the execution.

Auxiliary Variables. The semantics of quasi-static schedules rely on both worker-specific and global auxiliary variables. Each worker maintains a `counter` to track its progress, a `return_addr` for storing return addresses during jumps, and two temporary registers `temp0` and `temp1`. Global variables include `start_time` (the start timestamp), `timeout` (the final timestamp), and `time_offset` (the start time of the current SSFA state). To advance `time_offset`, the scheduler uses a global `offset_inc` variable to store per-state increments. For inter-worker synchronization at hyperperiod boundaries, each worker has a binary semaphore `binary_sema`. Two global constants, `zero` and `one`, hold the values 0 and 1, respectively.

Synchronization. In our execution strategy, we support two types of synchronization: (i) *local synchronization* between a pair of workers, and (ii) *global synchronization* across all workers. Local synchronization is achieved by letting a `WU` block on another worker's `counter` variable. On the other hand, global synchronization across all workers is performed at the completion of an SSFA state and is achieved by inserting a *synchronization code block* into each worker's executable schedule. Worker 0 first blocks until all other workers enter the code block and wait. Once all other workers start waiting, worker 0 increments the global `time_offset`, resets all workers' counters, advances all reactors' timestamps, clears all reactors' output ports, and finishes by signaling other workers to unblock.

Instruction Generation. Given a partitioned DAG, the instruction generation algorithm traverses the graph in topological order, emitting instructions for each node. For reaction nodes, it generates synchronization instructions (`WU`) to ensure correct inter-worker dependencies, timestamp advancement instructions (`ADVI`, `DU`) if the logical time progresses, and conditional branching instructions (`BEQ`, `JAL`) to test for triggers. If deadlines are declared, a guard sequence is emitted before executing the reaction body (`EXE`). After execution, connection management helpers and counter updates (`ADDI`) are inserted. For terminal synchronization nodes, the algorithm finalizes connection helpers and inserts synchronization and delay instructions across workers to prepare for the next logical tag. The generated instruction streams are compiled into binary programs; each core interprets its stream at runtime, coordinating to preserve the deterministic behavior of the original LF program.

Assumptions for Makespan Analysis. In Sec. 4.5, we introduced a simple procedure to analyze worst-case makespan for generic DAGs. To apply this procedure in the context of LF, the WCET of a task node (e.g., the WCET of $1ms$ for node g_1^1 in Fig. 5) needs to include: (i) the reaction body's WCET, (ii) the sum of the generated IR instructions' WCETs. For example, if a DAG node n produces instructions `EXE;ADDI`, where `EXE` launches the reaction body and `ADDI` performs bookkeeping using the `counter` variable, then the total WCET of node n is $wcet_n = \textsc{wcet}(reaction) + \textsc{wcet}(\texttt{EXE}) + \textsc{wcet}(\texttt{ADDI})$. The inter-core communication cost is accounted for in the WCETs of instructions `WU` and `WLT`. Their values can be derived from the microarchitecture of the multi-core processor. The additional wait time due to task dependencies is captured by the max operator in analysis (as seen in Example 1).

6 Evaluation

In this section, we evaluate the performance of executing quasi-statically scheduled LF programs and compare the execution overhead with that from using the dynamic scheduler of LF. Furthermore, we demonstrate the analyzable worst-case makespan property in Sec. 4.5 using a case study on a satellite attitude control system.

6.1 Performance

We evaluate the performance of the quasi- static scheduler by evaluating a set of LF benchmarks on a Raspberry Pi 4 Model B running QNX (SDP 8.0) real-time operating system. By performance, we refer to the runtime overhead, quantified as *lags* in the context of LF. Specifically, lags are measured as the difference between a reaction's actual physical firing time and its intended logical firing time. This metric applies to both the dynamic and the quasi-static scheduler. We compare the quasi-static schedulers' performance against that of the default dynamic scheduler. The eight benchmarks are selected from [26, 32] with a focus on concurrent actor design patterns [15], which can be found in modern cyber-physical systems. `LongShort` is an exception, as it is not drawn from the aforementioned citations but is instead introduced in this paper. This benchmark employs a design pattern that combines long- and short-running reactions, each executed at a frequency equal to the reaction's WCET. We annotate each reaction's WCET using an `@wcet` attribute (as seen in Fig. 9) for each benchmark program, and we collect performance data using LF's tracing utility. Each event contains an event type, a logical timestamp, and a physical timestamp. In this experiment, we only collect events for the starting points of reactions. This choice ensures that the tracing overhead stays uniform for all three schedulers under test.

Table 2 shows an overview of the performance results. Compared to the dynamic scheduler, both static schedulers have smaller average, maximum lag, and standard deviation on all benchmarks, except for the `ADASModel` when scheduled with EGS. This outlier is due to edge generation's scheduling strategy involving adding edges to the DAG, which could delay the releasing of tasks in exchange for schedulability guarantees. We note that the goal of using EGS is not to minimize lags, but to use its analysis capability to ensure that a DAG is schedulable. This makes larger lags from EGS tolerable.

Using the performance data reported in the "Average (μs)" column of Table 2, we compute the Performance Improvement Ratio (PIR), shown in the "PIR" column of the same table. The PIR quantifies the relative performance of the baseline techniques LB and EG compared to DY, and is calculated as the ratio of the average execution time of DY to that of LB ($\frac{\text{DY}}{\text{LB}}$) and EG ($\frac{\text{DY}}{\text{EG}}$), respectively. A higher PIR value indicates a greater performance improvement of the baseline

Table 2. Average, maximum, and standard deviation of the benchmark performance (lag) using the dynamic scheduler (DY), the static Load Balanced scheduler (LB), and the static Edge Generation scheduler (EGS).

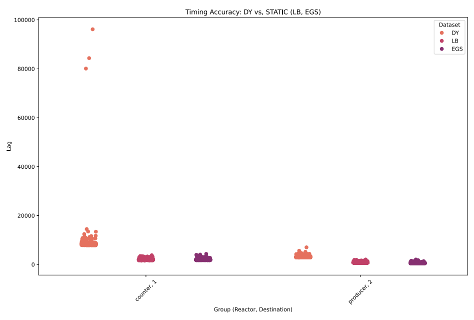
		Average (μs)			Maximum (μs)			Standard Deviation (μs)			PIR	
Program	LoC (LF)	DY	LB	EG	DY	LB	EG	DY	LB	EG	DY/LB	DY/EG
`Philosophers` [32]	314	86.1	2.99	2.53	1.37e+04	24.8	24	841	1.34	1.21	28.8	34.0
`PingPong` [32]	124	20.3	1.27	1.15	1.38e+04	6.43	5.8	377	0.758	0.677	16.0	17.7
`Throughput` [32]	166	5.21	2.43	2.53	10.1	6.33	7.43	1.38	1.19	1.25	2.1	2.1
`ThreadRing` [32]	217	6.95	2.29	2.02	13.1	5.89	5.15	2.39	1.32	1.17	3.0	3.4
`LongShort` (by us)	31	1.5e+06	77.2	78.7	2.92e+06	1.92e+03	1.95e+03	8.11e+05	379	386	19430.1	19059.7
`CoopSchedule` [18]	54	8.15	2.12	2.46	12.9	4.52	5.98	2.87	0.955	1.17	3.8	3.3
`Counting` [32]	179	5.81	1.41	1.27	98.5	7.76	9.2	3.3	0.592	0.703	4.1	4.6
`ADASModel` [26]	91	5.87	1.07	2.15e+03	22.2	3.35	1.2e+04	3.13	0.791	3.57e+03	5.5	0.0

technique over DY. Again, LB and EG consistently perform better than DY across various programs, with significant speedups. However, the `LongShort` and `ADASModel` programs are notable outliers, with `LongShort` showing extreme values due to its high execution time in DY, and `ADASModel` displaying a dramatic drop in performance when comparing DY to EG. We compute the average PIR of LB and EG across all benchmarks while excluding `LongShort` (because it appears overly favorable to our approach). The resulting average PIR values indicate that LB and EG improve performance by factors of 9.0 and 9.3, respectively, compared to DY.

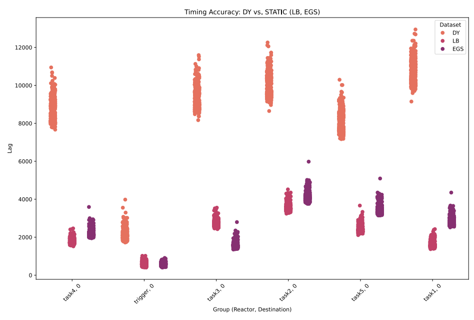
Fig. 12 shows strip plots of lags during the periodic phase of each program. For most reactions, LB and EGS have smaller lags than the dynamic scheduler. The occasional outliers shown in the strip plots can be due to several reasons, including interruptions, from the underlying OS and the flushing of trace buffers to disk.

Notably, for the `LongShort` benchmark, LB and EGS significantly outperforms the dynamic scheduler. This performance gain can be explained by the fact that the quasi-static schedulers keep track of the current logical time for *each* reactor, meaning that every reactor can advance its logical time independently as long as the core LF semantics is respected, i.e. each reactor processes events in timestamp order. On the other hand, the dynamic scheduler uses a single variable to track logical time for *all* reactors, ensuring that all reactors advance time together. The `LongShort` benchmark has a design pattern that combines infrequent, long-running reactions with frequent, short-running reactions. The dynamic scheduler struggles because it performs a *global barrier synchronization* at the end of each timestamp. The barrier makes reactions that are to occur at the next timestamp wait on the long-running reactions, even if there are no more actual data dependencies to fulfill. In the quasi-static schedulers, the capability to advance time individually for each reactor effectively prevents waiting unnecessarily, resulting in significantly reduced lags.

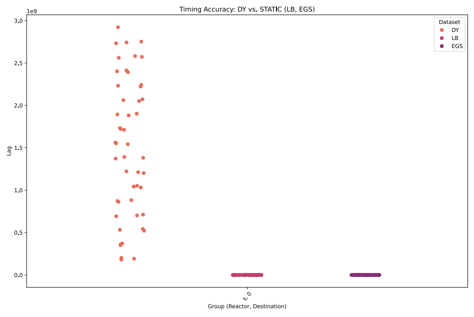
Limitations and Trade-offs. While our LF quasi-static schedulers seem to be quite performant in terms of low overhead, we note that quasi-static scheduling has its limitations. One of them is lack of flexibility. Since the schedule is determined at compile-time, the quasi-static schedulers require tight WCET labels and properly partitioned DAGs to generate high-quality schedules with high CPU utilization. Although this does not affect the correctness of the LF semantics delivered, overly loose WCET labels or poorly paritioned DAGs (in case of EGS, this could come from poorly added edges) could keep certain core unnecessarily idle. In addition, the execution times of tasks should not vary wildly within the WCET bound. If a task finishes too early, the nature of pre-determined schedules does not automatically keep the free core busy. On the other hand, dynamic scheduling is known for flexibility, which can keep the CPU utilization high without any knowing WCETs, at the cost of having scheduling overhead at runtime. This highlights the trade-off between predictability vs. utilization/runtime overhead. What we observe in this work is that more predictability results in less utilization and less runtime overhead, a trade-off the system designer needs to be mindful of.



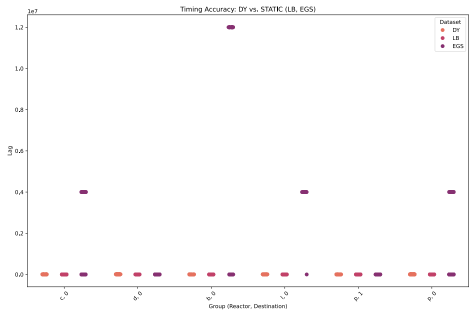
(a) Counting



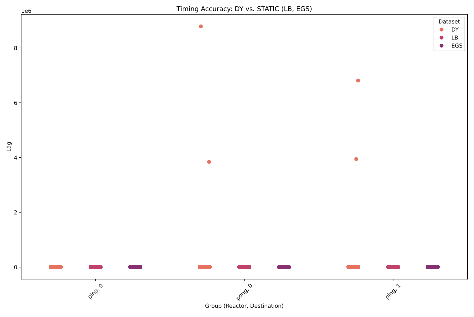
(b) CoopSchedule



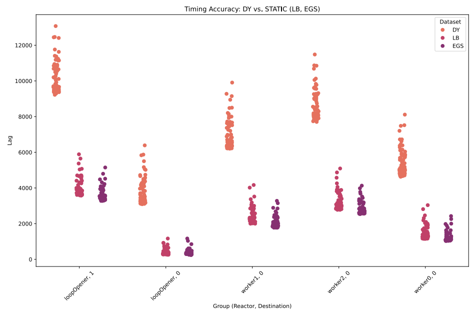
(c) LongShort



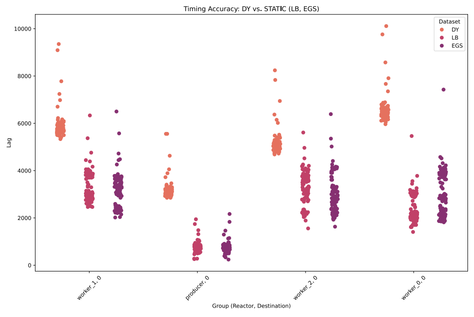
(d) ADASModel



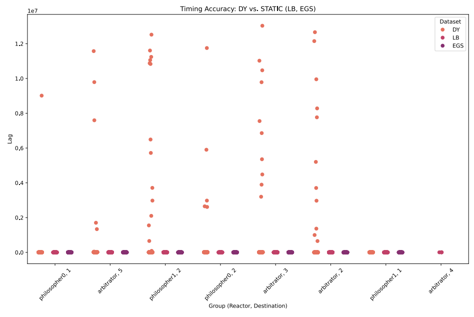
(e) PingPong



(f) ThreadRing



(g) Throughput



(h) Philosophers

Fig. 12. Timing accuracy results showing runtime overhead, measured on QNX (SDP 8.0) running on a Raspberry Pi 4. The X-axis represents reaction names, with each reaction displaying three distributions: DY, LB, and EGS. The Y-axis indicates lag in nanoseconds.

6.2 Case Study: Satellite Attitude Control System

We return to the satellite attitude control system from Fig. 1 and demonstrate the use of our analyzable worst-case makespan property in Sec. 4.5 to perform compile-time validation of LF deadlines. The experiment is done on FlexPRET [45], a time-predictable processor, and the WCET analysis tool Platin [31] to obtain verifiable hard real-time guarantees.

Real-time requirements. To assure the stability of this control system, we need to minimize both the input jitter and the end-to-end latency of the system. We encode the requirements on the input jitter by specifying a release deadline of 20 µs for the sensor sampling reactions. This means that using the logical timestamp of the Gyroscope reactions as sensor timestamps, at most, will be 20 µs off. The requirement on the end-to-end latency is encoded by the release deadline of 300 µs on the reaction of the Motor reactor.

Determining WCET Bounds. In LF, the business logic expressed in the reactions is often short and linear code sequences with bounded loops. They are, therefore, analyzable by existing WCET analysis tools including static tools such as Platin [36] and hybrid tools such as GameTime [37]. We extend the work of determining WCET bounds of LF reactions, originally started with LF on Patmos [35]. To perform WCET analysis with Platin, we first compile the program with the RISC-V version of patmos-clang, which generates a Platin Metainfo File that Platin consumes. We compute WCET bounds for all reaction bodies.

Table 3. WCET bounds for the reactions in the satellite attitude controller computed by Platin. The subscript denotes the reaction number within the reactor.

Reaction	WCET (µs)
Gyro_1	21
SensorFusion_1	61
Controller_2	172
Motor_1	1

This currently leaves a few small sections of code unanalyzed, such as the setup code of the reactions and the wake-up overhead of the platform. We performed measurements on those functions, showing that the wake-up execution time is in the order of microseconds, and the setup code execution times are in the sub-microsecond range, which we overapproximate with a constant number in our timing analysis. In future work, we plan to get these parts of the runtime amenable to WCET analysis, instead of using measurement.

Table 3 shows the WCET bounds computed by Platin for the reactions part of the periodic phase of the program, when targeting a FlexPRET processor clocked at 100 MHz.

Scheduling. The reactions are annotated with their WCET bounds, allowing the EGS scheduler to produce a partitioned schedule. EGS rejects a program if the input DAGs are not schedulable. The LF compiler takes the partitioned DAG and produces a quasi-static schedule. A subtlety is that the EGS scheduler does not consider runtime overhead when scheduling the DAG; this means that for certain high-utilization programs with very tight WCET bounds, the resulting schedule might not be feasible if it results in too much coordination overhead. The LF compiler thus performs a final deadline satisfaction check, including the runtime overhead of the executable schedules as computed by Platin.

Evaluation. The resulting cross-compiled program targets a 100 MHz FlexPRET processor supporting four fine-grain multithreading hard real-time threads, representing logical execution cores. A cycle-accurate emulation executes 1000 iterations of the program with simulated sensor inputs, and the execution times and completion times of each reaction are recorded. The results are shown in Fig. 13. The figure shows the measured execution time and the release time of each reaction in the periodic phase, represented as histograms. For the execution time, the WCET computed by Platin is shown by a red vertical line. For the release times, the LF release deadline, if applicable, is also shown by a red vertical line.

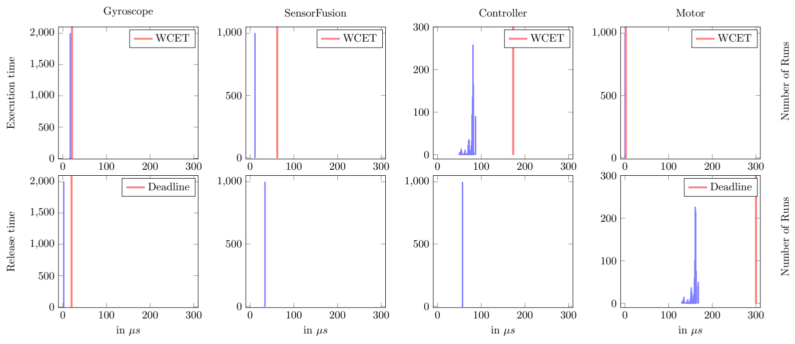


Fig. 13. Measured release times and execution times of the reactions from Fig. 9 on FlexPRET.

We observe that the measured execution times fall within the WCET bounds computed by Platin. The WCET bounds are, in most cases, significantly higher than the measured execution times; this is partly because FlexPRET does not have a hardware multiplier, so all multiplications and divisions are done in software. Such software implementations have data-dependent execution times. We also observe that the Gyroscope, SensorFusion, and Motor reactions show virtually no jitter in the execution times. Only the PID controller introduces variable, data-dependent execution times.

As guaranteed by our methodology, all reactions meet their release deadlines. The release time of the Gyroscope reactions is at 17 μs with zero variability, and the release time on the Motor reaction is met with a comfortable margin.

7 Conclusion

In this paper, we present the first unified quasi-static scheduling approach based on an intermediate formalism called state space finite automata (SSFA), bridging the gap between high-level MoCs and low-level executable schedules. We show that our unified scheduling approach applies to existing models, including SDF, BDF, SADF, and LET, and we identify a class of deterministic timed concurrent models (DTCMs) to which the SSFA-based methodology is applicable. We demonstrate the application of the proposed approach to an emerging MoC—the reactor model. Our evaluation confirms a successful application of the SSFA-based quasi-static scheduling approach to reactors from its reduced average runtime overhead (by 9 times compared to the default dynamic scheduler), and the LF compiler's newly added support for compile-time deadline validation, benefiting from the properties of quasi-static scheduling. Moving forward, we plan to evaluate our approach quantitatively using larger-scale benchmarks as well as applying our unified SSFA-based scheduling framework to more emerging models of computation.

Acknowledgment

The work in this paper was supported in part by the National Science Foundation (NSF), award #CNS-2233769 (Consistency vs. Availability in Cyber-Physical Systems), by Intel, and by the iCyPhy Research Center (Industrial Cyber-Physical Systems). This work was funded in part by the German Federal Ministry of Research, Technology and Space of Germany (BMFTR) in the programme "Souverän. Digital. Vernetzt." joint project 6G-life (16KISK001K), and by the German Research Council (DFG) through the InterMCore project (505744711). This work is also part of a project (PropRT) that has received funding from the European Research Council (ERC) under the European Union's Horizon 2020 research and innovation programme (grant agreement No. 865170). 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. Mirco Theile and Binqi Sun were supported by the Chair for Cyber-Physical Systems in Production Engineering at TUM. Linh Thi Xuan Phan was supported by NSF CNS-2111688 and CCF-2326606. The authors thank Marten Lohstroh, Christian Menard, and the anonymous reviewers for providing helpful feedback. Shaokai Lin thanks Zitao Fang and Yang Huang for collaborating on a CS267 project at UC Berkeley that inspired this work.

References

[1] 2014. *System Design, Modeling, and Simulation using Ptolemy II.*

[2] J.T. Buck and E.A. Lee. 1993. Scheduling dynamic dataflow graphs with bounded memory using the token flow model. In *1993 IEEE International Conference on Acoustics, Speech, and Signal Processing*, Vol. 1. 429–432 vol.1.

[3] Joseph T. Buck. 1993. *Scheduling Dynamic Dataflow Graphs with Bounded Memory Using the Token Flow Model.* Ph. D. Dissertation. EECS Department, University of California, Berkeley.

[4] Alan Burns and Sanjoy Baruah. 2008. Sustainability in real-time scheduling. *Journal of Computing Science and Engineering* 2, 1 (2008), 74–97.

[5] Emilia Farcas and Wolfgang Pree. 2007. Hyperperiod bus scheduling and optimizations for TDL components. In *2007 IEEE Conference on Emerging Technologies and Factory Automation (EFTA 2007).* 1262–1269.

[6] Marc C. W. Geilen, Mladen Skelin, J. Reinier van Kampenhout, Hadi Alizadeh Ara, Twan Basten, Sander Stuijk, and Kees G. W. Goossens. 2020. *Scenarios in Dataflow Modeling and Analysis.* 145–180.

[7] Kai-Björn Gemlau, Leonie Köhler, Rolf Ernst, and Sophie Quinton. 2021. System-level Logical Execution Time: Augmenting the Logical Execution Time Paradigm for Distributed Real-time Automotive Software. *ACM Transactions on Cyber-Physical Systems* 5, 2 (2021), 1–27.

[8] A.H. Ghamarian, M.C.W. Geilen, S. Stuijk, T. Basten, B.D. Theelen, M.R. Mousavi, A.J.M. Moonen, and M.J.G. Bekooij. 2006. Throughput Analysis of Synchronous Data Flow Graphs. In *Sixth International Conference on Application of Concurrency to System Design (ACSD'06).* 25–36.

[9] T. A. Henzinger, B. Horowitz, and C. M. Kirsch. 2001. Giotto: A Time-Triggered Language for Embedded Programming. In *EMSOFT 2001*, Vol. LNCS 2211. 166–184.

[10] Thomas A. Henzinger and Christoph M. Kirsch. 2002. The embedded machine: Predictable, portable realtime code. In *International Conference on Programming Language Design and Implementation (PLDI).* 315–326.

[11] Thomas A. Henzinger, Christoph M. Kirsch, and Slobodan Matic. 2003. Schedule-Carrying Code. In *Embedded Software.* 241–256.

[12] C. A. R. Hoare. 1972. Towards a Theory of Parallel Programming. In *Operating Systems Techniques*, Vol. 9. 61–71.

[13] Benjamin Horowitz. 2003. *Single-mode, single-processor Giotto scheduling.*

[14] Tsung-Wei Huang, Dian-Lun Lin, Chun-Xun Lin, and Yibo Lin. 2022. Taskflow: A Lightweight Parallel and Heterogeneous Task Graph Computing System. *IEEE Transactions on Parallel and Distributed Systems* 33, 6 (2022), 1303–1320.

[15] Shams M. Imam and Vivek Sarkar. 2014. Savina - An Actor Benchmark Suite: Enabling Empirical Evaluation of Actor Libraries. In *Proceedings of the 4th International Workshop on Programming Based on Actors Agents & Decentralized Control* (Portland, Oregon, USA). 67–80.

[16] Axel Jantsch. 2003. *Modeling Embedded Systems and SoCs - Concurrency and Time in Models of Computation.*

[17] Erling R. Jellum, Shaokai Lin, Peter Donovan, Chadlia Jerad, Edward Wang, Marten Lohstroh, Edward A. Lee, and Martin Schoeberl. 2023. InterPRET: A Time-Predictable Multicore Processor. In *Proceedings of Cyber-Physical Systems and Internet of Things Week 2023* (San Antonio, TX, USA). 331–336.

[18] Erling Rennemo Jellum, Shaokai Lin, Peter Donovan, Efsane Soyer, Fuzail Shakir, Torleiv Bryne, Milica Orlandic, Marten Lohstroh, and Edward A. Lee. 2023. Beyond the Threaded Programming Model on Real-Time Operating Systems. In *Fourth Workshop on Next Generation Real-Time Embedded Systems (NG-RES 2023)*, Vol. 108. 3:1–3:13.

[19] Christoph M Kirsch and Ana Sokolova. 2012. The logical execution time paradigm. *Advances in Real-Time Systems* (2012), 103–120.

[20] R. L. Graham. 1969. Bounds on Multiprocessing Timing Anomalies. *SIAM J. Appl. Math.* 17, 2 (1969), 416–429.

[21] E.A. Lee. 2006. The problem with threads. *Computer* 39, 5 (2006), 33–42.

[22] Edward A Lee. 1988. Recurrences, iteration, and conditionals in statically scheduled block diagram languages. *VLSI Signal Processing* 3 (1988), 330–340.

[23] Edward Ashford Lee and David G. Messerschmitt. 1987. Static Scheduling of Synchronous Data Flow Programs for Digital Signal Processing. *IEEE Trans. Comput.* C-36, 1 (1987), 24–35.

[24] Edward A. Lee and David G. Messerschmitt. 1987. Synchronous data flow. *Proc. IEEE* 75, 9 (1987), 1235–1245.

[25] Ben Lickly, Isaac Liu, Sungjun Kim, Hiren D. Patel, Stephen A. Edwards, and Edward A. Lee. 2008. Predictable Programming on a Precision Timed Architecture. In *Proceedings of the 2008 International Conference on Compilers, Architectures and Synthesis for Embedded Systems* (Atlanta, GA, USA). 137–146.

[26] Shaokai Lin, Yatin A. Manerkar, Marten Lohstroh, Elizabeth Polgreen, Sheng-Jung Yu, Chadlia Jerad, Edward A. Lee, and Sanjit A. Seshia. 2023. Towards Building Verifiable CPS Using Lingua Franca. *ACM Trans. Embed. Comput. Syst.* 22, 5s (2023).

[27] Marten Lohstroh, Christian Menard, Soroush Bateni, and Edward A. Lee. 2021. Toward a Lingua Franca for Deterministic Concurrent Systems. *ACM Transactions on Embedded Computing Systems (TECS), Special Issue on FDL'19* 20, 4 (2021), Article 36.

[28] Marten Lohstroh, Íñigo Íncer Romero, Andrés Goens, Patricia Derler, Jeronimo Castrillon, Edward A. Lee, and Alberto Sangiovanni-Vincentelli. 2020. Reactors: A Deterministic Model for Composable Reactive Systems. In *Cyber Physical Systems. Model-Based Design – Proceedings of the 9th Workshop on Design, Modeling and Evaluation of Cyber Physical Systems (CyPhy 2019) and the Workshop on Embedded and Cyber-Physical Systems Education (WESE 2019)* (New York City, NY, USA). 59–85.

[29] Thomas Lundqvist and Per Stenström. 1999. Timing Anomalies in Dynamically Scheduled Microprocessors. In *Proceedings of the 20th IEEE Real-Time Systems Symposium.* 12–21.

[30] Claire Maiza, Hamza Rihani, Juan M. Rivas, Joël Goossens, Sebastian Altmeyer, and Robert I. Davis. 2019. A Survey of Timing Verification Techniques for Multi-Core Real-Time Systems. *ACM Comput. Surv.* 52, 3 (2019).

[31] Emad Jacob Maroun, Eva Dengler, Stefan Dietrich, Chistian Hepp, Benedikt Herzog, Henriette Huber, Jens Knoop, Daniel Prokesch, Peter Puschner, Phillip Raffeck, Martin Schoeberl, Simon Schuster, and Peter Wägemann. 2024. The Platin Multi-Target Worst-Case Analysis Tool. In *22th International Workshop on Worst-Case Execution Time Analysis (WCET 2024).*

[32] Christian Menard, Marten Lohstroh, Soroush Bateni, Matthew Chorlian, Arthur Deng, Peter Donovan, Clément Fournier, Shaokai Lin, Felix Suchert, Tassilo Tanneberger, Hokeun Kim, Jeronimo Castrillon, and Edward A. Lee. 2023. High-performance Deterministic Concurrency Using Lingua Franca. *ACM Trans. Archit. Code Optim.* 20, 4 (2023).

[33] Giovanni De Micheli. 1994. *Synthesis and optimization of digital circuits.*

[34] Guillaume Roumage, Selma Azaiez, Cyril Faure, and Stéphane Louise. 2025. An Extended Survey and a Comparison Framework for Dataflow Models of Computation and Communication. *arXiv preprint arXiv:2501.07273* (2025).

[35] Martin Schoeberl, Ehsan Khodadad, Shaokai Lin, Emad Jacob Maroun, Luca Pezzarossa, and Edward A. Lee. 2024. Invited Paper: Worst-Case Execution Time Analysis of Lingua Franca Applications. In *22nd International Workshop on Worst-Case Execution Time Analysis (WCET 2024)*, Vol. 121. 4:1–4:13.

[36] Martin Schoeberl, Wolfgang Puffitsch, Stefan Hepp, Benedikt Huber, and Daniel Prokesch. 2018. Patmos: a time-predictable microprocessor. *Real-Time Syst.* 54, 2 (2018), 389–423.

[37] Sanjit A. Seshia and Alexander Rakhlin. 2012. Quantitative Analysis of Systems Using Game-Theoretic Learning. *ACM Transactions on Embedded Computing Systems (TECS)* 11, S2 (2012), 55:1–55:27.

[38] Gilbert C. Sih. 1991. *Multiprocessor Scheduling to Account for Interprocessor Communication.* Ph. D. Dissertation. EECS Department, University of California, Berkeley.

[39] Sander Stuijk, Marc Geilen, Bart Theelen, and Twan Basten. 2011. Scenario-aware dataflow: Modeling, analysis and implementation of dynamic applications. In *2011 International Conference on Embedded Computer Systems: Architectures, Modeling and Simulation.* 404–411.

[40] Binqi Sun, Mirco Theile, Ziyuan Qin, Daniele Bernardini, Debayan Roy, Andrea Bastoni, and Marco Caccamo. 2024. Edge Generation Scheduling for DAG Tasks Using Deep Reinforcement Learning. *IEEE Trans. Comput.* 73, 4 (2024), 1034–1047.

[41] Reinier van Kampenhout, Sander Stuijk, and Kees Goossens. 2017. Programming and analysing scenario-aware dataflow on a multi-processor platform. In *Design, Automation & Test in Europe Conference & Exhibition (DATE), 2017.* 876–881.

[42] Micaela Verucchi, Ignacio Sañudo Olmedo, and Marko Bertogna. 2023. A survey on real-time dag scheduling, revisiting the global-partitioned infinity war. *Real-Time Systems* 59, 3 (2023), 479–530.

[43] Micaela Verucchi, Mirco Theile, Marco Caccamo, and Marko Bertogna. 2020. Latency-aware generation of single-rate DAGs from multi-rate task sets. In *IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS).* 226–238.

[44] Andrew Waterman, Yunsup Lee, David Patterson, Krste Asanovic, Volume I User level Isa, Andrew Waterman, Yunsup Lee, and David Patterson. 2014. The RISC-V instruction set manual. *Volume I: User-Level ISA', version* 2 (2014).

[45] Michael Zimmer, David Broman, Chris Shaver, and Edward A. Lee. 2014. FlexPRET: A processor platform for mixed-criticality systems. In *2014 IEEE 19th Real-Time and Embedded Technology and Applications Symposium (RTAS).* 101–110.

Received 30 March 2025; revised 15 June 2025; accepted 13 July 2025