High-Performance Deterministic Concurrency using Lingua Franca

CHRISTIAN MENARD, TU Dresden, Germany
MARTEN LOHSTROH, UC Berkeley, USA
SOROUSH BATENI, UC Berkely, USA
MATTHEW CHORLIAN, UC Berkeley, USA
ARTHUR DENG, UC Berkeley, USA
PETER DONOVAN, UC Berkeley, USA
CLÉMENT FOURNIER, TU Dresden, Germany
SHAOKAI LIN, UC Berkeley, USA
FELIX SUCHERT, TU Dresden, Germany
TASSILO TANNEBERGER, TU Dresden, Germany
HOKEUN KIM, Arizona State University, USA
JERONIMO CASTRILLON, TU Dresden, Germany
EDWARD A. LEE, UC Berkeley, USA

Actor frameworks and similar reactive programming techniques are widely used for building concurrent systems. They promise to be efficient and scale well to a large number of cores or nodes in a distributed system. However, they also expose programmers to nondeterminism, which often makes implementations hard to understand, debug, and test. The recently proposed reactor model is a promising alternative that enables deterministic concurrency. In this paper, we present an efficient, parallel implementation of reactors and demonstrate that the determinacy of reactors does not imply a loss in performance. To show this, we evaluate Lingua Franca (LF), a reactor-oriented coordination language. LF equips mainstream programming languages with a deterministic concurrency model that automatically takes advantage of opportunities to exploit parallelism. Our implementation of the Savina benchmark suite demonstrates that, in terms of execution time, the runtime performance of LF programs even exceeds popular and highly optimized actor frameworks. We compare against Akka and CAF, which LF outperforms by $1.86x$ and $1.42x$, respectively.

Additional Key Words and Phrases: coordination, concurrency, determinism, performance

1 INTRODUCTION

Theoreticians working on programming language semantics have long understood the value of determinism as well as the expressive power of nondeterminism in programming languages. In practice, however, today, nondeterminism creeps into programming languages and frameworks not to benefit from its expressiveness, but rather because of a widespread perception that it is needed to get good performance on parallel hardware. In this paper, we show experimentally that a wide range of reactive applications can be implemented deterministically without sacrificing performance. We do this

Authors' addresses: Christian Menard, christian.menard@tu-dresden.de, TU Dresden, Germany; Marten Lohstroh, marten@berkeley.edu, UC Berkeley, USA; Soroush Bateni, soroush@berkely.edu, UC Berkely, USA; Matthew Chorlian, mattchorlian@berkeley.edu, UC Berkeley, USA; Arthur Deng, langxing.deng@berkeley.edu, UC Berkeley, USA; Peter Donovan, peterdonovan@berkeley.edu, UC Berkeley, USA; Clément Fournier, clement.fournier@tu-dresden.de, TU Dresden, Germany; Shaokai Lin, shaokai@berkeley.edu, UC Berkeley, USA; Felix Suchert, felix.suchert@tu-dresden.de, TU Dresden, Germany; Tassilo Tanneberger, tassilo.tanneberger@tu-dresden.de, TU Dresden, Germany; Hokeun Kim, hokeun@asu.edu, Arizona State University, USA; Jeronimo Castrillon, jeronimo.castrillon@tu-dresden.de, TU Dresden, Germany; Edward A. Lee, eal@berkeley.edu, UC Berkeley, USA.

2023. Manuscript submitted to ACM

by focusing on actor frameworks, which have proved popular and successful in many very demanding applications, but admit nondeterminism that is often not actually needed by their applications.

Exploiting parallel hardware such as multicore machines to improve performance is only possible when programs expose concurrency. Common abstractions for concurrency include threads [24], remote procedure calls [72], publish-subscribe [34], service-oriented architectures [75], and actors [1, 40]. Each of these models has its own merits, but they all introduce nondeterminism: situations where, for a given state and input, the behavior of a program is not uniquely defined. While nondeterminism can be useful in some applications, most programming tasks benefit from more repeatable behavior. Deterministic programs are easier to understand, debug, and test (for each test vector, there is one known-good response). For nondeterministic programs, problematic behaviors might be harder to discover because they may only occupy a small fraction of the state space [44]. And reproducing failures can be extremely hard [50, 56, 70] because they might occur only when the system is under a specific amount of load [82].

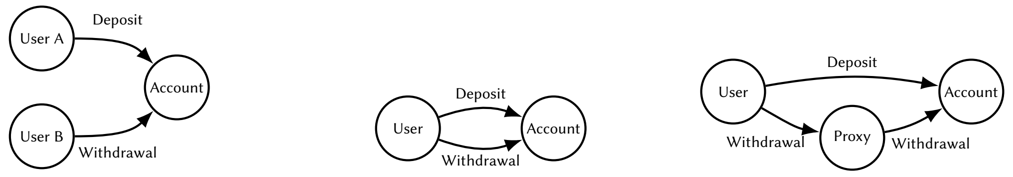
Determinism is a subtle concept [49]. Here, we focus on a particular form of determinism for programs, where a program is deterministic if, given the same inputs, it always produces the same outputs. This definition does not require that operations be performed in a particular order, and therefore is not at odds with concurrency and parallel execution. It is possible, but often not easy, to achieve this form of determinism even when using nondeterministic abstractions such as threads, actors, and asynchronous remote procedure calls. For simple enough programs, such as a chain of actors, if communication is reliable, then execution will be deterministic. Some of the benchmarks we compare against in this paper are deterministic in this way. As we will show, however, even slightly more complex communication structures result in nondeterminism that can be difficult to correct.

In this paper, we evaluate a language-based coordination that preserves determinism by default and only admits nondeterminism when explicitly introduced by the programmer. The coordination language Lingua Franca (LF) [61], which is based on a concurrent model of computation called reactors [60, 58], achieves this by analyzing program structure and ensuring that data dependencies are observed correctly at runtime. An LF program defines reactive software components called "reactors" and provides operators to compose them hierarchically (through containment) and bilaterally (via connections). Because the language supports both deterministic and nondeterministic concurrency, it provides a fertile ground for exploring the impact of determinism on performance.

The semantics of the deterministic subset of LF can be thought of as a deterministic variant of actors [1, 40, 59]. We show in this paper that it delivers performance comparable to popular nondeterministic realizations of actors on parallel hardware, like Akka [79] and CAF [21]. Similar to Akka and CAF, LF orchestrates the execution of code written in conventional programming languages. However, unlike those frameworks, LF is polyglot. It currently supports C, C++, Python, TypeScript, and Rust. This paper focuses on the runtime performance of the C++ target, which, as a core contribution of this paper, has been optimized to efficiently exploit concurrency on parallel hardware. Earlier work [61] has only reported preliminary performance indications of LF based on its C target, which is predominantly aimed at running on embedded systems.

At the core of LF's concurrency model is a logical model of time that gives a clear notion of simultaneity and avoids deadlocks using dependency analysis based on causality interfaces [53]. It is this timed semantics that enables efficient deterministic concurrency in LF. However, the benchmarks we compare against were created to evaluate actor frameworks, which have no temporal semantics. None of the benchmarks take advantage of the time-related features of LF; the temporal semantics is only used to deliver determinism.

Since the execution of LF programs requires a dependency analysis, the precise structure of the program needs to be known at startup. Modifying the program structure during execution is currently not possible. Therefore, the



(a) Deposit and Withdrawal sent by different users. (b) Deposit and Withdrawal sent by same user. (c) Withdrawal sent via a proxy.

Fig. 1. Example actor programs that may expose nondeterministic behavior.

performance comparison in this paper is limited to actor programs that can be expressed statically. We argue that this is true for most applications, especially those that benefit from LF's semantics. For instance, out of the 32 programs in the Savina benchmark suite [41], only 8 require dynamic actor creation. While the underlying reactor model defines so-called *mutations* for runtime adaptations, LF does not implement them yet. A full discussion of mutations in LF including a performance comparison with dynamic actor creation remains for future work.

Contributions. We show that the reactor-oriented paradigm as implemented in Lingua Franca enables efficient exploitation of parallel hardware without relinquishing determinism. For this, we explain the mechanisms through which LF programs expose concurrency; we present a language extension that allows for the definition of scalable programs; and we introduce an optimized C++ runtime for LF that enables efficient parallel execution. We further present an extensive evaluation based on the Savina benchmark suite [41], showing that our LF runtime outperforms Akka and CAF by $1.86x$ and $1.42x$, respectively.

Outline. We first motivate our work (Section 2) and then introduce LF (Section 3). We go into detail about the concurrency in LF, discuss a syntax extension for scalable connection patterns and introduce our optimized C++ runtime (Section 4). Next, we report benchmark results (Section 5), discuss related work (Section 6), and conclude (Section 7).

2 MOTIVATION

The actor model is widely accepted and deployed in production for its promise to allow programmers to easily express concurrency, provide high execution performance, and scale well to large datasets and complex applications. Moreover, in contrast to thread-based programs, actor semantics prevents *low-level* data races. However, like most message-passing paradigms, actors expose the programmer to nondeterminism in the form of *high-level* data races [92], a problem that becomes considerably challenging to manage as the complexity of a program grows.

Consider the simple example in Fig. 1a. The `Account` actor manages the balance of a bank account that two users interact with. User A sends a deposit message increasing the account's balance and User B sends a withdrawal message decreasing the account's balance. If we assume that the balance is initialized to 0 and the account only grants a withdrawal if the resulting balance is not negative, then there are two possible behaviors. If A's message is processed first, the withdrawal is granted to B. If B's message is processed first, the withdrawal is denied. The actor model assigns no meaning to the ordering of messages. Therefore, there is no well-defined correct behavior for this example.

The reader may object that for an application like that of Fig. 1a, the order of transactions is intrinsically nondeterministic, and any additional nondeterminism introduced by the software framework is inconsequential. However, if we focus on testability, we see that even identical inputs can yield different results, making testing more difficult. If we focus on consistency, the problem that different observers of the same events may see different behaviors becomes problematic. In databases, it is common to assign time stamps to external inputs and to then treat those timestamps as a semantic property of the inputs and define the behavior of the database relative to those time stamps. We adopt this perspective in this paper, and rely on the definition of determinism given by Lee [49]: "determinism is a property of models, not of physical realizations," and "A model is deterministic if given all the *inputs* that are provided to the model, the model defines exactly one possible *behavior*." If we define "inputs" in Fig. 1a to be time-stamped user queries and "behavior" to be the sequence of actions taken by the Account, then it is reasonable to demand determinism.

Consider Fig. 1b, which has only one user. Even if this one user first sends a deposit and then a withdrawal message, the actor model does not guarantee that the receiving actor sees and processes the incoming messages in this order. While some actor frameworks, e.g., Akka and Erlang, guarantee in-order message delivery, others, e.g., AmbientTalk [93], expressly do not. Yet, even if the framework guarantees point-to-point in-order message delivery, this property is not transitive. If we add a `Proxy`, as shown in Fig. 1c, then we cannot make any assumptions about the order in which `Account` receives messages. This example further illustrates that composing actors can have unexpected side effects.

Consequently, implementing solutions to practical concurrency problems with actors can be challenging. Even seemingly simple concurrency problems like the one discussed above require high programming discipline, and solutions are typically difficult to maintain and tend to lack modularity. In addition, the inherent nondeterminism of actor frameworks makes it hard to verify such solutions. Erroneous behavior might only occur in a fraction of executions, and thus integration tests cannot reliably detect such "Heisenbugs" [70].

In a recent study, Bagherzadeh et al. [4] analyzed bugs in Akka programs that were discussed on StackOverflow or GitHub and determined that 14.6% of the bugs are caused by races. This makes high-level races the second most common cause of bugs in Akka programs after errors in the program logic. In a similar study of 12 actor-based production systems, Hedden and Zhao [39] determined that 3.2% of the reported bugs were caused by bad message ordering, 4.8% of bugs were caused by incorrect coordination mechanisms, 4.8% were caused by erroneous coordination at shutdown, and 2.4% of bugs were caused by erroneous coordination at startup. Note that these numbers only cover *known* bugs in their studied projects and, as noted by the authors, the majority of the reported message ordering bugs belonged to the Gatling project because it already incorporated a debugging tool called Bita [86] that is designed to detect such bugs. We suspect that there are more undetected bugs in projects that do not use specialized debugging tools.

The actor community has addressed the inherent nondeterminism of actors and the resulting bugs by introducing better tools for analyzing and debugging actor programs. This includes TransPDOR [87], Bita [86], Actoverse [84], iDeA [63], CauDEr [46], and Multiverse debugging [91]. While these are valuable solutions, we argue that a programming model for expressing concurrent programs should provide deterministic semantics by default and allow the programmer to introduce nondeterminism only where it is desired and understood to do no harm. In such cases, the aforementioned tools for nondeterministic behavior can still be utilized to debug the implementation.

There are a number of ways to achieve deterministic concurrency, including Kahn process networks [42, 43], many flavors of dataflow models [27, 71, 51], physically asynchronous, logically synchronous models [83], synchronous-reactive languages [8, 31], and discrete-event systems [96, 18, 52, 30]. Lohstroh et al. [61] compare the reactor model to each of these, showing that it has many of their best features and fewer of their pitfalls. Lingua Franca builds on this reactor model because it is more expressive than some of the alternatives (e.g., Kahn networks) and is stylistically close to actors, which have proven effective in practice. In this paper, we show that the resulting determinism does not incur a performance penalty, but on the contrary, helps to achieve improved performance in most cases.

```
1  target Cpp
2  reactor User(offset: time(0), val: float(0)) {
3    timer t(offset)
4    output req: float
5    reaction (t) -> req {= req.set(val); =}
6  }
7  reactor Account {
8    state balance: float(0.0)
9    input reqA: float
10   input reqB: float
11   reaction (reqA) {= apply(*reqA.get()); =}
12   reaction (reqB) {= apply(*reqB.get()); =}
13   method apply(val: float) {=
14     if (balance + val > 0) {
15       balance += val; std::cout << "Accepted\n";
16     } else { std::cout << "Denied\n"; }
17   =}
18 }
```

```
20 main reactor {
21   account = new Account()
22   userA = new User(offset=1 sec, val=20)
23   userB = new User(offset=2 sec, val=-10)
24   userA.req -> account.reqA
25   userB.req -> account.reqB
26 }
```

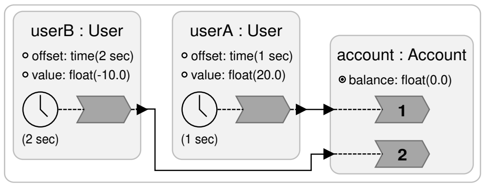


Fig. 2. LF implementation of the actor program shown in Fig. 1a. The diagram on the right is automatically generated by the LF IDE.

3 INTRODUCTION TO LINGUA FRANCA

Lingua Franca (LF) builds on the relatively new reactor-oriented programming paradigm. Intuitively, we can describe reactors as deterministic actors with a discrete event execution semantics and explicitly declared ports and connections. A logical timeline is used to order events and ensure a deterministic execution. As a polyglot language, LF incorporates code in a target programming language to implement the logic of each component. LF itself is only concerned with the coordination aspect of a program. In this section, we introduce the core concepts of reactors and LF. For a more detailed introduction to LF's concepts and syntax, the interested reader may refer to Lohstroh et al. [61].

3.1 LF by Example

Fig. 2 shows an LF implementation of the deposit/withdrawal example in Fig. 1a. The diagram shown on the right uses a graphical syntax to visualize the LF program. It is automatically synthesized from the source code by the LF IDEs [94]. The first line of the program is a **target** declaration, which specifies the target language (C++ in this case). The program further specifies three **reactor** classes: `Account`, `User`, and an anonymous **main** reactor. The main reactor serves as an entry point for LF programs and is instantiated automatically at runtime. Reactor classes in LF are in many ways analogous to classes in object-oriented languages. In particular, reactor classes encapsulate state, methods and other components, offer a form of inheritance, can be generic, and are parameterized at instantiation.

The `User` reactor class (lines 2 to 6) is parameterized by an offset and a value of types `time` and `float`. **time** is the only built-in type of LF and represents a time value. All other types are given in the target language. The **timer** declared on line 3 will trigger once after the given offset. Note that timers may additionally specify a period to trigger the timer repeatedly. The **output** port `req` declared on line 4 is used to send events with an associated `float` value to other reactors, indicating a deposit or withdrawal request.

In LF, all computation is performed in reactive code segments called **reactions** that are implemented in the target language. In the diagram, reactions are represented by dark gray chevrons. All reactions must explicitly declare their triggers, other dependencies and potential effects.In line 5, the `User` reactor declares a reaction that is triggered by the timer `t` and that may produce an event on the output port `req`. The reaction body is given in C++ code and sets the `req` output based on the value that the class is parameterized with at instantiation.

The `Account` reactor is defined on line 7. It has a **state** variable `balance` of type `float`, two input ports `reqA` and `reqB`, one reaction for each input, and a **method** named `apply`. State variables and methods in LF are equivalent to protected member variables and methods in object-oriented languages. Methods are useful for sharing code within a reactor, but they cannot be invoked by other reactors. Triggering functionality in other reactors is only possible by emitting events via ports on connections, which can subsequently trigger a reaction. In addition to methods, LF also provides **preambles** which can be used to define shared functions and types, and to insert target language imports. Preambles live in a global scope and cannot access reactor members.

The reactions on lines 11 and 12 are triggered by the `reqA` or `reqB` ports and attempt to apply the requested change to the balance. Reactions can access any methods, parameters, or state variables declared by the local reactor. Both reactions retrieve the value associated with the triggering event on the respective port and call the method `apply`. In the C++ target, the additional dereference operator (`*`) is required as all values are wrapped by a smart pointer for fine-grained access control and safe memory management. The `apply` method defined on lines 13-17 implements the account's business logic. If the resulting balance is non-negative, it modifies the balance accordingly and prints "Accepted". Otherwise, it prints "Denied".[1]

Note that we implemented `account` using two separate ports and reactions for the sake of simplicity. The reader might notice that the separated reactions duplicate logic and are not a practical solution, in particular if there are many users. We choose this representation to keep our exposition simple. In Sec. 4.2, we will introduce a syntax that enables a more compact implementation of `Account`.

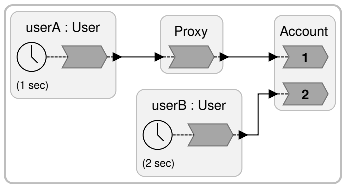
The main reactor assembles the program. It creates a single instance of `Account` (line 21), two instances of `User` (lines 22, 23), and connects the outputs of the user instances to the inputs of the account instance (lines 24, 25) using the -> operator. The `userA` is parameterized with an offset of 1 second and a value of 20 and `userB` is parameterized with an offset of 2 seconds and a value of -10. When executed, the program will wait for 1 second before triggering the timer of the `userA` reactor and invoking the reaction on line 5. The event produced by this reaction will trigger the reaction on line 11 which is invoked immediately after the first reaction completes. Two seconds after program startup, `userB` will react and subsequently trigger the reaction on line 12.

In this example, the deposit event (+20.0) occurs earlier than the withdrawal event (-10.0), and hence our execution semantics ensures that the account processes the deposit event before the withdrawal event, meaning the balance will not become negative. In a more realistic implementation, the two users would generate events sporadically and have their reactions triggered not by a timer but a physical action (see Section 3.3). However, using a timer greatly simplifies our exposition as we only have to consider a single *logical timeline* along which events are ordered. Moreover, such timers can be used to create regression tests that validate program execution with specific input timings.

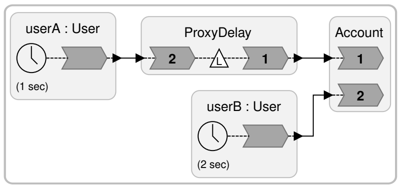
Note that even when the two events occur logically simultaneously, meaning that both reactions in the `account` reactor are triggered at the same logical time, the resulting program will be deterministic. All reactions at the same logical time are executed according to a well-defined precedence relation. In particular, any reactions within the same reactor are mutually exclusive and executed following the lexical declaration order of the reactions in LF code. This order is also reflected by the numbers displayed on the reactions in the diagram in Fig. 2. More details on the precedence relation of reactions are given in Section 4.1. Since reactions and connections are logically instantaneous, the execution order is also preserved if any proxies are inserted, as is shown in Fig. 3a.

To deliberately change the order in which events occur, a logical delay can be introduced in the program using a **logical action**, as shown in Fig. 3b and the corresponding code in Fig. 3c. In the diagram, actions are denoted by small white triangles. In contrast to ports, which allow relaying events logically instantaneously from one reactor to

[1]This implementation of `account` is oversimplified to keep our exposition concise. A more realistic implementation of `account` would interact with a database reactor and send feedback to the users to indicate it the transaction was successful.



(a) Adding a proxy reactor.



(b) Adding a proxy introducing a logical delay.

```
1  target Cpp
2  import User, Account from "Example.lf"
3
4  reactor ProxyDelay<T>(delay: time(0)) {
5    input in: T
6    output out: T
7    logical action a: T
8    reaction(a) -> out {= out.set(a.get()); =}
9    reaction(in) -> a {= a.schedule(in.get(), delay); =}
10 }
11 main reactor {
12   account = new Account()
13   userA = new User(offset=1 sec, val=20)
14   userB = new User(offset=2 sec, val=-10)
15   delay = new ProxyDelay<float>(delay=2 sec)
16   userA.req -> delay.in
17   delay.out -> account.reqA
18   userB.req -> account.reqB
19 }
```

(c) LF code of the program shown in (b).

Fig. 3. Modifications of the example shown in Fig. 3a

another, logical actions provide a mechanism for scheduling new events at a later (logical) time. Upon receiving an input, reaction 2 of the `ProxyDelay` reactor is triggered, which schedules its logical action with a configurable delay. This creates a new event which, when processed, triggers reaction 1 of the `ProxyDelay` reactor, which retrieves the original value and forwards it to its output port.

The reactor class `ProxyDelay` (line 4) has a type parameter `T` that denotes the type of the values of its input, output, and logical action. Using a runtime API function called `schedule`, the reaction on line 9 schedules a future event (on logical action `a`) with the value of the triggering input event and a given delay.[2] The reaction on line 8 is triggered by `a` and simply forwards the value of the triggering event to the output port.

On line 15, the delay reactor is instantiated with a delay of 2 seconds and using the type `float`. The other reactors are instantiated from the definitions given in Fig. 2, which are imported on line 2. Due to the additional delay of 2 seconds, the deposit message from `userA` will only be processed after the withdrawal message from `userB`, causing B's request to be denied. Since delaying messages is a common problem, LF provides a dedicated syntax for it. Instead of manually inserting a delay reactor, we can use an **after** delay. For this, we remove line 15 and replace lines 16 and 17 with `userA.req -> account.reqA after 2 sec`.

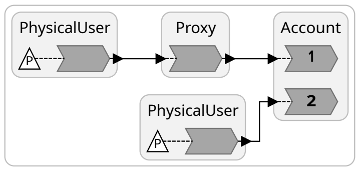
It is important to note that all of the discussed examples are deterministic, regardless of the physical execution times of reactions, as all events are unambiguously ordered along a single logical timeline. The physical timing of the events, on the other hand, will be approximate. The contribution of this paper is to show that such determinism does not necessarily reduce performance and is also useful for applications that have no need for explicit timing.

3.2 Logical and physical time

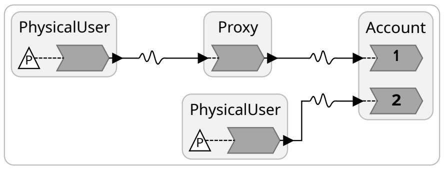
All events have an associated **tag**. Tags are ordered along a logical timeline and can be thought of as a timestamp. Timers automatically schedule events at regular intervals relative to a start tag that is determined at startup. Reactions may use logical actions to schedule future events with a given delay relative to the current tag.

In time-sensitive applications, tags are not purely used for logical ordering but also relate to physical time. By default, the runtime only processes the events associated with a certain tag once the current physical time T is greater than the

[2]The smart pointer obtained with `get` on lines 8 and 9 is not dereferenced to obtain the actual value. Multiple overloads exist for `set` and `schedule` and they accept both references to values as well as smart pointers. Thus the smart pointer obtained with `get` can be passed directly to `set` or `schedule`.



(a) Asynchronous input with physical actions.



(b) Nondeterministic implementation using physical connections.

```
1  target Cpp
2  import Account, Proxy from "Example.lf"
3
4  reactor PhysicalUser {
5    physical action a: float
6    output req: float
7    reaction (a) -> req {= req.set(a.get()); =}
8  }
9
10 main reactor {
11   account = new Account()
12   userA = new PhysicalUser()
13   userB = new PhysicalUser()
14   proxy = new Proxy()
15   userA.req -> proxy.in
16   proxy.out -> account.reqA
17   userB.req -> account.reqB
18 }
```

(c) LF code of the program shown in (a).

Fig. 4. Implementation of the account example with asynchronous input via physical actions (a and c) and a fully nondeterminisitc variation that uses physical connections (b).

time value of the tag t ($T > t$). We say that logical time "chases" physical time. The relationship between physical and logical time in the reactor model gives logical delays a useful semantics and also permits the formulation of deadlines. This timed semantics is particularly useful for software that operates in cyber-physical systems. For a more in-depth discussion of LF's timed semantics, the interested reader may refer to [61].

If an application has no need for any physical time properties, the concurrence of physical and logical time can be turned off; in this case, the tags are used only to preserve determinism, not to control timing. Moreover, LF programmers are not required to explicitly control the timing aspects of their programs. Delays can simply be omitted, for instance when scheduling an action, in which case the runtime will use the next available tag. In consequence, also untimed general-purpose programs can benefit from the deterministic concurrency enabled by LF's timed semantics.

3.3 Asynchrony and deliberate nondeterminism

In the examples discussed above in Section 3.1, we have hard-coded the order in which the users send requests by using timers and, thus, assigned fixed tags to the request events. While using a predefined order is useful for testing and for demonstration, reactor programs that are deployed in practice need to be able to handle sporadic asynchronous inputs in order to be useful. Concretely, in our account example we do need to handle asynchronous events that are created when users initiate withdrawal or deposit requests.

The reactor model distinguishes logical actions and *physical* ones. While a logical action is always scheduled synchronously with a delay relative to the current tag, a **physical action** may be scheduled from asynchronous contexts. Its event is assigned a tag based on the current reading of physical time. Physical actions are the key mechanism for handling sporadic inputs originating from physical processes and for introducing deliberate nondeterminism.

The assignment of tags to physical actions is nondeterministic in the sense that it is not defined by the program. However, once those tags are assigned, for example, to deposit or withdrawal requests by a user, the processing of the events is deterministic and occurs in tag order. Hence, the tags assigned to externally initiated events are considered as part of the *input*, and given this input, the program remains deterministic. This approach draws a clear perimeter around the deterministic and therefore testable program logic while allowing it to interact with sporadic external inputs.

Fig. 4a and 4c show an implementation of our account example that uses physical actions to handle sporadic user requests. The physical action on line 5 may be scheduled from asynchronous processes outside of the LF program. It has type `float` which allows it to carry the value of the deposit or withdrawal request initiated by the user. The reaction on line 5 is triggered by the physical action and it forwards the value of the action to the output port. The `Account` and `Proxy` reactors remain unchanged. Consequently, they implement the same testable behavior as in our earlier examples. We only exchanged the event sources from predefined timers to physical actions to allow sporadic input. Concretely, if `userA` sends a deposit message at tag g_A and `userB` sends a withdrawal message at tag g_B with $g_B > g_A$, then the semantics of LF guarantees that the response of the account is identical to the response in a test case that uses the same tag ordering (i.e. the behavior is identical to the program in Fig. 3a).

Physical actions can also be used within the program itself, for example, to nondeterministically assign a new tag to a message received from another reactor. In this usage, physical actions provide a means for deliberately introducing actor-like nondeterminism into a program. For example, the program shown in Fig. 4b reproduces the nondeterministic behavior of the actor program shown in Fig. 1c. It is created by replacing the logical connection operator -> with the physical connection operator ~> on lines 15 to 17. Physical connections behave similar to after delays, but instead of inserting a delay, they insert a physical action to create events with tags based on the current physical time. Thus, in Fig. 4b the deposit and withdrawal messages are tagged nondeterministically in the order in which they arrive at the account. Consequently, the account processes the messages in the order of arrival.

3.4 The scope of Lingua Franca

LF was originally designed for modeling applications in the context of cyber-physical systems. The deterministic concurrency delivered by LF in combination with its timed semantics makes it particular attractive for time-sensitive and safety-critical applications [65, 61].

However, as we demonstrate in this paper, reactors and LF are not limited to cyber-physical systems. Lingua Franca is very expressive and can model a wide range of concurrent applications. Applications like the simple account example from Fig. 4a do not require explicit modeling of time, but they benefit from deterministic concurrency nonetheless. The interaction of multiple components within the system becomes testable and reproducible, as the system delivers the same response when provided with the same inputs, no matter if it is deployed in production or in a test environment.

Similar to the actor model, Lingua Franca is not limited to a particular domain, but is a general-purpose coordination language. We demonstrate this in our evaluation by porting a wide range of actor programs to LF. In fact, LF can express any actor program that does not require dynamic actor creation. In future work, we plan to introduce mutations in LF to also support the dynamic creation of reactors so that LF becomes as expressive as the actor model. However, even with this limitation, LF is strictly more general than other known solutions for deterministic concurrency. Most prominently, Kahn process networks [42, 43] and various dataflow models [27, 11, 47] are known to deliver high performance for streaming applications, but they have limited expressivity as they cannot easily model reactions to sporadic events. In this paper, we demonstrate that LF can express the reactive behavior of actors while guaranteeing a deterministic execution and delivering high performance that even exceeds actors in some benchmarks.

4 EFFICIENT DETERMINISTIC CONCURRENCY

LF programs are deterministic by default. This property is inherited from the reactor model that LF implements. Lohstroh et al. [61] explain why reactors behave deterministically. Their argument can be adapted to the concrete context of the Lingua Franca language, but this is beyond the scope of this paper. Reactors are also concurrent, and, as we show in this paper, the exposed concurrency is sufficient for the runtime system to effectively exploit multi-core hardware to where it matches or exceeds the performance of fundamentally asynchronous and nondeterministic actor frameworks.

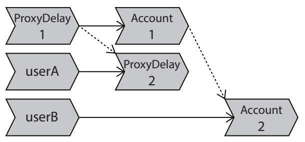


Fig. 5. Reaction graph for the program in Fig. 3b.

In this section, we show how LF exposes concurrency, introduce a syntax extension for writing scalable LF programs, and describe in more depth how our C++ runtime is implemented and how it efficiently utilizes parallel hardware.

4.1 Parallelism

The use of statically declared ports and connections as well as the declarations of reaction dependencies, distinguish reactors from more dynamic models like actors or other asynchronous message-passing frameworks where communication is purely based on addresses. While the fixed topology of reactor programs is less flexible and limits runtime adaptation, it also provides two key advantages. First, it achieves a separation of concerns between the functionality of components and their composition. Second, it makes explicit at the interface level which dependencies exist between components. As a consequence, a dependency graph can be derived for any composition of reactors.

The dependency graph is an **acyclic precedence graph (APG)** that organizes all reactions into a partial order that captures all scheduling constraints that must be observed to ensure that the execution of a reactor program yields deterministic results. Because this graph is valid irrespective of the contents of the code that executes when reactions are triggered, reactions can be treated as a black box. It is this property that enables the polyglot nature of LF and exposes the concurrency in the application.

Fig. 5 shows the dependency graph for the program given in Fig. 3b. The solid arrows represent dependencies that arise because one reaction (possibly) sends data to the other via ports and connections. The dashed arrows represent dependencies that arise because the two reactions belong to the same reactor. Analogous to the behaviors of actors, reactions of the same reactor are mutually exclusive. The execution order is well-defined and given by the lexical declaration order of the reactions in LF code. This order is also indicated by the numbers in the reaction labels in Fig. 3b. Note that actions do not create dependencies and are thus not represented in the APG. This is because actions are always scheduled in the future with a tag greater than the current tag. Since the runtime ensures that events are processed in tag order (cf. Section 4.3), the dependency between scheduling an event and reacting to it is not represented in the APG.

The dependency graph precisely defines in which order reactions need to be executed. Independent reactions may be executed in parallel without breaking determinism. For instance, the APG in Fig. 5 tells us that reaction 1 of `ProxyDelay` and the reactions of `userA` and `userB` can all execute in parallel. Note that the dependency graph is required to be acyclic as any cycle would violate causality. The LF compiler ensures that a valid program has an acyclic dependency graph. Any dependency cycles in LF programs can be resolved by introducing a logical action, and thus a logical delay, to break one of the dependencies and moving part of the computation to a later tag.

4.2 Scalable Connection Patterns

Explicitly listing all individual reactor instances, ports and connections in LF code may become tedious for larger programs. For example, consider again the program from Fig. 2. To scale it to four users, we would need to explicitly list two more ports in the account reactor, and add two more named user instances and connections to the main reactor.

⟨*input*⟩ ::= ‘input’ ⟨*width*⟩? ID (‘:’ ⟨*type*⟩)?

⟨*output*⟩ ::= ‘output’ ⟨*width*⟩? ID (‘:’ ⟨*type*⟩)?

⟨*instance*⟩ ::= ID ‘=’ ‘new’ ⟨*width*⟩? ID ‘(’ ⟨*arg-list*⟩? ‘)’

⟨*width*⟩ ::= ‘[’ (INT | ID | ⟨*code*⟩) ‘]’

⟨*connection*⟩ ::= ⟨*port-refs*⟩ ‘->’ ⟨*port-refs*⟩
| ‘(’ ⟨*port-refs*⟩ ‘)+’ ‘->’ ⟨*port-refs*⟩

⟨*port-refs*⟩ ::= ⟨*port-ref*⟩ (‘,’ ⟨*port-ref*⟩)*

⟨*port-ref*⟩ ::= (ID ‘.’)? ID | ‘interleaved(’ (ID ‘.’)? ID ‘)’

Fig. 6. LINGUA FRANCA syntax extension for expressing banks, multiports, and connections over multiple port references.

```
1  target Cpp
2  reactor Account(num_users: size_t(4)) {
3    state balance: float (0.0)
4    input[num_users] req: float
5    reaction (req) {=
6      for (size_t i{0}; i < num_users; i++) {
7        if (req[i].is_present()) { apply(i, *req[i].get()); }
8      }
9    =}
10   method apply(idx: size_t, val: float) {=
11     std::cout << "Process request " << val << "...";
12     if (balance + val >= 0) {
13       balance += val;
14       std::cout << " Accepted \n";
15     } else { std::cout << " Denied \n"; }
16   =}
17 }
```

```
22 reactor User(bank_index: size_t(0)) {
23   output req : float
24   reaction (startup) -> req {=
25     req.set(15.0 - bank_index * 10.0);
26   =}
27 }
28 main reactor(num_users: size_t(4)) {
29   account = new Account (num_users=num_users)
30   users = new[num_users] User()
31   users.req -> account.req
32 }
```

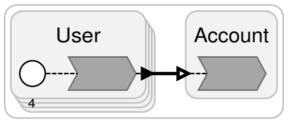


Fig. 7. Modified version of the LF program in Fig. 2 using banks and multiports.

This explicit listing of all ports, connections and instances is not only cumbersome for the programmer, it also means that the LF code needs to be adjusted and recompiled whenever the problem size changes.

To address this problem, this section introduces a syntax extension that allows to create multiple ports or reactor instances at once. Further, we introduce an overloading of LF's connection operator to create multiple connections at once. This mechanism allows realizing various complex connection patterns in a single line of code and in a parameterizable way, allowing LF programs to transparently scale to a given problem size without recompilation. This is a key enabler for implementing the programs of the Savina benchmark suite which we use in our evaluation.

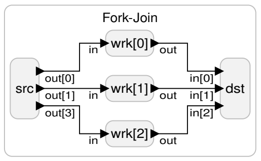
4.2.1 Syntax extension. Concretely, we extend the **input**, **output**, and the **new** keyword to accept an optional width specification in brackets. This creates an array of ports or an array of reactor instances. We call such an array of ports a **multiport** and an array of reactor instances a **bank**. We further extend the connection operator, such that multiple ports may be listed on either side of the operator in a comma-separated list. Finally, we introduce a **broadcast** modifier `(...)+` and the **interleaved** modifier which provide more control over how the listed ports are connected. Fig. 6 lists all the modified syntax rules. We explain the newly introduced concepts by example in the following.

First, let us consider the program in Fig. 7, which is a scalable modification of our initial example in Fig. 2. The account reactor defines a multiport input (line 4) with a width that is given by the `num_users` parameter. The reaction on line 5 triggers if any of the individual ports in the multiport carry an event. If multiple ports carry an event at the same tag, then the reaction is only triggered and executed once at this tag. Since we do not know which port actually carries an event, the reaction body iterates over all ports, checks if a value is present, and then calls `apply` for each present request. This small modification allows the account reactor to interact with an arbitrary number of users and truly separates the business logic as implemented in the account's reaction from its usage in the system.

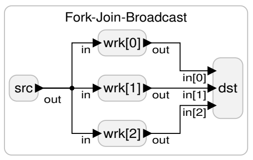
Instead of creating individual users, the main reactor instantiates a bank of user reactors on line 30. The width of the bank is again given by the parameter `num_users`. By default, this will create four instances of `User`. The `bank_index` parameter (line 30) is set automatically to the index of the instance within the bank. This allows for the state or the

```
1  target Cpp
2  reactor Src(w: int(3)) {
3    output[w] out: int
4  }
5  reactor Worker {
6    input in: int
7    output out: int
8  }
12 reactor Sink(w: int(3)) {
13   input[w] in: int
14 }
15 main reactor(w: int(3)) {
16   src = new Src(w = w)
17   dst = new Sink(w = w)
18   wrk = new[w] Worker()
19   src.out -> wrk.in
20   wrk.out -> dst.in }
```

(a) LF code for implementing a fork-join pattern



(b) Diagram of the program in (a)



(c) Variant with broadcast

Fig. 8. A simple fork-join program in LF.

behavior of a reactor to depend on its position within a bank. Here, each user sends a request at startup with a value that is calculated from the `bank_index`.

The connection operator on line 31 connects the request outputs of all the users to the multiport input of the account. Thereby, it connects the output of the nth user to the nth port in the input multiport of the account. This pattern implements a many-to-one communication. Note that the number of users can be adjusted arbitrarily. Since `num_users` is a parameter to the main reactor, the LF code generator will also add it to the program's command line parameters, which allows overwriting the default parameter without recompilation.

In case the number of ports on the left-hand and right-hand side of the connection operator do not match, some ports remain unconnected. Let n and m denote the number of ports on the left and right side, resepectively, then only the first $min(n, m)$ ports are connected on either side. In this case, we also issue a warning message.

4.2.2 Connection Patterns. The syntax extension introduced in this section is relatively simple, but powerful enough to cover many communication patterns. In the following, we show how a selection of common patterns can be conveniently expressed in LF. Note that we omit all reactions and other implementation details for brevity and solely focus on the connection patterns. The presented patterns are used extensively in our benchmark implementations in Section 5.

Fork-Join. Fig. 8a gives an example program implementing a fork-join pattern, which combines one-to-many and many-to-one communication. The program defines a `Src`, a `Worker`, and a `Sink` reactor. `Src` and `Sink` both define a multiport output or input of width `w`. `Worker` only uses each a single input and output port, but is instantiated in a bank of width `W` on line 18. The two connection statements in the main reactor (line 19, 20) establish `w` connections each, one for each pair of multiport and bank instance. The resulting connection pattern is illustrated in Fig. 8b for `w=3`.

In this example, the source reactor produces three separate values to be sent to the worker. We provide a modifier `(...)+` which allows to instead broadcast a single value to all workers. Configuring the source reactor to use a single output (by setting `w=1` in line 16) and changing line 19 to `(src.out)+ -> wrk.in` creates the pattern shown in Fig. 8c.

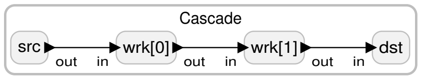
In either variant, the reactions of each worker may execute in parallel to the reactions of all other workers.

Cascade Composition. Our proposed syntax can also conveniently express cascade composition, as illustrated by the program in Fig. 9a. The connection operator sequences all ports listed on the left- and right-hand side, and connects the nth port on the left-hand side to the nth port on the right-hand side. By offsetting the left-hand side of the connection statement in line 5 with a single source port and appending the sink port to the right-hand side, we can effectively arrange the connections to form the cascade shown in Fig. 9b.

Fully Connected. The connection operator also connects multiports within banks. In this case, the operator will implicitly unfold all port instances on both sides of the connection to form a flat list of ports. The unfolding happens

```
main reactor(n: int(2)) {
  src = new Src(w = 1)
  dst = new Sink(w = 1)
  wrk = new[n] Worker()
  src.out, wrk.out -> wrk.in, dst.in
}
```

(a) LF code for implementing a cascade

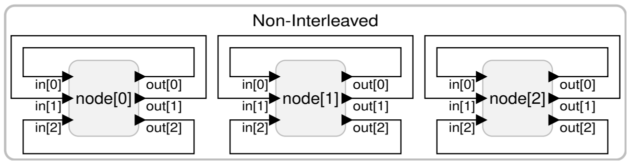


(b) Diagram of the program in (a)

Fig. 9. A simple cascade in LF.

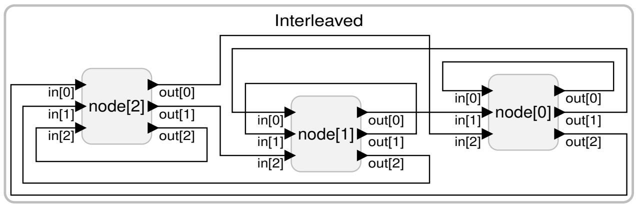
```
reactor Node(w: int(3)) {
  input[w] in: int
  output[w] out: int
}
main reactor(w: int(3)) {
  node = new[w] Node(w=w)
  node.out -> node.in
}
```

(a) Connecting multiports within a bank



(b) Diagram for the program in (a)

```
main reactor(w: int(3)) {
  node = new[w] Node(w=w)
  node.out -> interleaved(node.in)
}
```



(c) Using `interleaved` for fully connected nodes

(d) Diagram for the program in (c)

Fig. 10. Connecting multiports within banks to create fully connected networks.

such that we first list all ports of the first bank instance, then all ports of the second instance, and so on. Consider the program in Fig. 10a. This will create the pattern shown in Fig. 10b which is not very useful. Using the `interleaved` modifier on either side of the connection, we can modify the unfolding strategy to first list all first port instances within all bank instances, then the second port instances within all bank instances, and so on. The program in Fig. 10c creates the fully connected pattern shown in Fig. 10d. This allows each node to send and receive messages to or from all other nodes. Thereby, the n-th output or input port corresponds to the n-th instance of the node.

4.3 Runtime Implementation

In the previous subsections, we have discussed how Lingua Franca exposes parallelism and how we can express various connection patterns in a scalable way. In this subsection we discuss how this parallelism can be exploited efficiently during execution. The execution of each LF program is governed by a runtime. Most importantly, the runtime includes a scheduler which keeps track of all scheduled future events, controls the advancement of logical time, and invokes any triggered reactions in the order specified by the dependency graph while aiming to exploit as much parallelism as possible. Lohstroh et al. have already sketched a simple scheduling algorithm for reactor programs [60]. In this section, we present a C++ implementation of this scheduling algorithm that aims at exploiting parallelism while keeping synchronization overhead to a minimum and avoiding contention on shared resources.

Fig. 11 gives a high-level overview of the scheduling mechanism as defined in [60] and as used in our runtime. The scheduler keeps track of future events in the *event queue* and processes them strictly in tag order. When processing an event, the scheduler first determines all reactions that are triggered by the event and stores them in the *reaction queue*. Any reactions in the reaction queue for which all dependencies are met (as indicated by the APG) are forwarded to

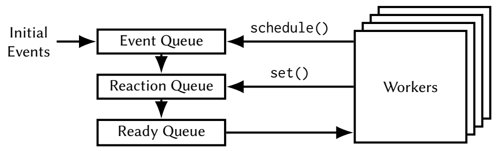


Fig. 11. Scheduling mechanism in the LF runtime.

the ready queue and then picked up for execution by the worker threads. If the executed reactions trigger any further reactions by setting ports, those reactions are inserted in the reaction queue. If a reaction schedules future events via an action, these new events are inserted into the event queue. Note that the scheduler always waits until all reactions at the current tag are processed before advancing to the next tag and triggering new reactions.

The most important task of the scheduler is to decide when any given reaction should be moved from the reaction queue to the ready queue. As the APG precisely defines the ordering constraints of reactions, reaction scheduling is closely related to DAG-based scheduling strategies [45, 2]. However, the APG is not equivalent to a task graph as it may contain reactions that do not need to be executed. Most often only a fraction of the reactions is triggered at a particular tag. Moreover, we do not know in advance precisely which reactions will be triggered for a given tag, as reactions may or may not send messages via their declared ports. In consequence an optimal schedule cannot be computed in advance.

To decide whether a given triggered reaction is ready for execution, we need to check if it has a dependency on any other reaction that is triggered or currently executing. To avoid traversing the APG at runtime, we utilize a simple heuristic. Concretely, we assign a *level* (top level as defined in [45]) to each reaction. Any reactions with the same level do not depend on each other and hence can be executed in parallel. Our scheduler then processes reactions going from one level to the next. Once all reactions within a level are processed, all triggered reactions in the next level are moved to the ready queue. This approach avoids the need for analyzing the APG during execution, but also falls short on exploiting all opportunities for parallel execution. For instance, this approach does not execute reaction 2 of `ProxyDelay` in parallel with reaction 2 of `Account`. Nonetheless, our evaluation shows that this strategy is sufficient to efficiently exploit parallelism in most cases. Given the extensive research on DAG-scheduling, we are confident that we future work can apply more complex strategies to account for the missed opportunities for exploiting parallelism.

Another limitation of our scheduling approach is that the scheduler only considers reactions that are triggered at the same tag. In particular, this may hinder exploiting pipeline parallelism in programs that do not use logical delays to create pipeline stages. However, this limitation can be overcome by using a federated execution strategy [61, 7].

The scheduling mechanism described above is fundamentally different from typical actor implementations. Since actor programs are nondeterministic and thus the workload cannot be predicted, the runtime needs to make ad-hoc decisions to distribute the workload. The predominant solution is work stealing [16, 95], which is also the default scheduling mechanism of Akka and CAF. The main advantage of work stealing is that it avoids a centralized scheduler and minimizes the synchronization points between workers. As long as they have sufficient work, workers can operate independently. The work stealing approach, however, does not work for a reactor runtime, as deciding which reactions are ready to process requires more knowledge about the system's state. While we require a central scheduler, we can leverage knowledge about the program to optimize the execution.

Since the reactor model is based on discrete events, our scheduling algorithm is closely related to the mechanisms used in discrete event simulators such as SystemC [73, 13] or gem5 [12, 62]. However, parallelizing the execution in such simulators is commonly hard as the dependencies and the precise interactions of components are not known in advance. While multiple works exist that allow a multithreaded execution of discrete event simulations [81, 22, 80], they often require manual partitioning and it remains challenging to deliver high performance for general applications. Therefore, both SystemC and gem5 simulations are single threaded by default. By leveraging the properties of the reactor model, we created a novel discrete event scheduler that precisely understands which reactions can be executed in parallel and delivers an efficient execution.

4.4 Optimizations

While the scheduling algorithm sketched in [60] and discussed in the previous subsection is relatively straightforward to implement, further optimizations where needed to achieve competitive performance. In the following, we detail the most important optimizations that we use in our C++ runtime.

Coordinating worker threads. In Fig 11 we conceptually distinguished the scheduler from the workers. In an actual implementation, however, using a central scheduler and separate worker threads introduces several synchronization points. The scheduler needs to send work to the workers, and the workers need to notify the scheduler when they finished. Instead, in our implementation, a thread that runs out of work tries to become the scheduler and moves ready reactions to the ready queue or advance logical time to the next tag if all reactions have been processed. Only one worker thread can become the scheduler and all other workers that run out of work will go to sleep until they are woken up again by the scheduler. This is guarded by an atomic flag.

At any time we know we know the precise number of reactions that may execute in parallel. Thus we can use a counting semaphore to wake up precisely as many workers as needed. By limiting the number of active workers, we avoid unnecessary contention on the ready queue in situations where there are more workers than ready reactions.

Lock-free data structures. The three queues and other data structures that are required for bookkeeping (e.g., a list of all set ports) are shared across workers. Using mutexes for synchronization proved to be inefficient due to high contention on the shared resources, especially when many parallel reactions set ports or schedule actions. Instead, we utilize lock-free data structures where possible. For instance, the ready queue is implemented as a fixed-size buffer paired with an atomic counter. Since we know precisely how many reactions can at most run in parallel (i.e. the maximum number of reactions in the APG that have the same level), we can fix the size of the queue. Every time new reactions are moved to the reaction queue, the atomic counter is set to the number of reactions in the queue. Each time a worker tries to execute a reaction it atomically decrements the counter. If the result is negative, then the queue is empty. Otherwise, the result provides the index within the buffer to read from. We further exploit knowledge about the execution of reactor programs. For instance, the scheduler advances logical time only once all reactions have been processed. This operation is safe without additional synchronization, as all of the workers are waiting for new reactions.

Sparse multiports. Reactors that use a multiport input to interact with multiple other reactors that may send messages individually (such as in the example from Fig. 7), need to identify which ports actually have a present value. Let n denote the width of the multiport and p the number of present ports. If the multiport width is large and communication is sparse ($p \ll n$), then iterating over all ports and checking for presence individually is inefficient ($O(n)$). Therefore, our C++ runtime internally uses a lock-free buffer to keep track of the ports that are actually set and exposes an API function called `present_indices_unsorted` that obtains the indices of only set ports. Using this function, iterating over all present ports has complexity $O(p)$. Note, however, that the port indices can have an arbitrary order if the ports are written by parallel reactions. If a fixed order is required, `present_indices_sorted` can be used to obtain a sorted list of indices. The sorting has complexity $O(p \cdot \log(p))$.

5 PERFORMANCE EVALUATION

The actor model is widely accepted for programming large concurrent applications, and implementations such as the C++ Actor Framework (CAF) [21] and Akka [79] are known to be fast and efficient in utilizing a larger number of threads. Compared to actors, LF imposes various restrictions that amount to a model of computation in which fewer behaviors are allowed. In this section, we show that these restrictions do not necessarily introduce overhead or higher execution times. In fact, LF is considerably faster for many benchmarks.

5.1 Methodology

Our evaluation is based on the Savina benchmark suite [41] for actor languages and frameworks. While this suite has several issues, as Blessing et al. discuss in [14], Savina covers a wide range of patterns and, to the best of our knowledge, is the most comprehensive benchmark suite for actor frameworks that has been published. The Savina suite includes Akka implementations of all benchmarks. CAF implementations of most Savina benchmarks are also available.[3]

We ported 22 of the 30 Savina benchmarks to the C++ target of LF. Due to the fundamental differences between the actor and reactor model, the process of porting benchmarks is not always straightforward. We aimed at closely resembling the original workloads and considered the intention behind the individual benchmarks. We will present more details about our implementations of selected benchmarks in the next section.

We did not implement the benchmarks Fork Join (actor creation), Fibonacci, Quicksort, Bitonic Sort, Sieve of Eratosthenes, Unbalanced Cobwebbed Tree, Online Facility Location, and Successive Over-Relaxation as they require the capability to dynamically create actors. In the reactor model, this can be achieved with mutations that may modify the reactor topology [60, 58]. However, mutations are not yet fully implemented in LF, and a discussion of language-level constructs for supporting mutations is beyond the scope of this paper. Although the precise cost of performing mutations is currently unknown to us, this cost will mostly depend on how efficiently the APG can be modified. Since the APG remains static in between mutations, we expect no difference in performance for the execution of reactions, and hence the results discussed her yield measurements that will be useful even when mutations are eventually supported.

We further omit the A*-Search and Logistic Map Series benchmarks from our presentation. The A*-Search implementation in the Savina suite suffers from a severe race condition that results in wildly varying execution times [14]. Logistic Map Series is omitted as the Akka implementation violates actor semantics and requires explicit synchronization [14]. For this reason, the CAF implementation needs to use a blocking call, which makes it slower than the other implementations by at least two orders of magnitude. Since this is not a problem of CAF, but rather a problem in the benchmark design, we omit Logistic Map Series to avoid skewing the analysis.

All measurements were performed on a workstation with an Intel Core i9-10900K processor (10 cores, 20 hardware threads) with 32 GiB DDR4-2933 RAM running Ubuntu 22.04 and using CAF version 17.6 and Akka version 2.6.17. Following the methodology of Savina, measurements exclude initialization and cleanup. Each measurement comprises 32 iterations. The first two iterations are excluded from our analysis and are used to warm up.

5.2 Benchmark implementation in LF

Table 1 provides an overview of all the Savina benchmarks that we have ported and included in our discussion. The table also lists various key characteristics of our implementations. The middle section displays characteristics about the size of each program such as the total number of reactors, reactions, actions, ports, and connections. The right section

[3]https://github.com/woelke/savina

Table 1. Characteristics of the Savina benchmarks implemented in LF. The middle part denotes static information about the size of the program and the right part denotes runtime information about the execution of the program.

ID	cat.	benchmark	reactors	reactions	actions	ports	connections	processed tags	processed reactions	set ports	scheduled actions	time per reaction [ns]
1	micro	Ping Pong	4	8	3	8	4	1,000,004	3,000,005	2,000,002	1,000,010	74
2		Thread Ring	103	109	3	406	203	1,004	101,008	100,004	1,208	71
3		Counting Actor	4	11	3	12	6	1,000,005	2,000,010	1,000,005	1,000,011	77
4		Fork Join (throughput)	63	67	3	65	62	10,004	610,006	10,002	10,128	26
7		Chameneos	103	208	3	20,404	10,202	4,005	408,209	800,402	4,209	169
8		Big	123	487	122	57,964	29,042	20,004	6,472,034	4,800,122	2,400,248	221
9	concurrency	Concurrent Dictionary	24	30	3	146	83	10,004	220,028	400,023	10,050	217
10		Concurrent Sorted Linked List	24	31	4	145	83	8,005	176,029	320,022	8,051	43,323
12		Dining Philosophers	23	71	3	224	122	20,004	460,019	1,000,002	20,048	123
13		Sleeping Barber	2,005	8,017	5	24,014	12,008	4,004	18,008	14,002	8,012	1,265
14		Cigarette Smokers	203	208	4	404	202	1,005	2,007	1,002	1,409	2,692
16		Bank Transaction	1,003	3,007	3	2,004,004	1,002,002	79	78,905	101,002	2,083	464
11	parallelism	Producer Consumer (bounded)	83	209	42	484	282	1,005	122,128	160,082	40,208	12,325
17		All-Pairs Shortest Path	45	115	38	873	798	304	21,643	10,839	10,892	45,270
19		N Queens First K Solutions	23	30	4	124	62	256	5,472	9,130	300	46,150
20		Recursive Matrix Multiplication	23	50	4	105	62	37	651	661	81	1,268,569
22		Radix Sort	64	248	63	186	123	899,962	7,000,100	6,100,002	900,104	86
23		Filter Bank	54	141	3	254	150	34,821	1,073,278	809,063	34,927	2,075
28		Trapezoidal Approximation	103	108	3	404	202	5	108	202	209	6,050,557
29		Precise Pi Computation	23	30	4	84	42	213	4,584	8,322	257	22,172

shows details about the benchmark execution such as the number of tags (or events) that were processed, the number of executed reactions, and how often ports were set and actions scheduled. Finally, the average time per executed reaction gives an estimate of the size of the workload implemented in each reaction.

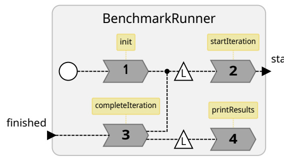
As indicated in Table 1, the Savina benchmarks are divided in three categories: *micro*, *concurrency* and *parallelism.*[4]. The micro benchmarks focus on stressing various mechanisms in the runtime scheduler to expose overheads in the runtime. The concurrency benchmarks have a similar goal, but they put more focus on the concurrent operation of (re)actors and also require synchronization mechanisms to solve the particular problem. Since the micro and concurrency benchmarks are designed to mostly stress the runtime, the workload implemented in each (re)actor is relatively small (with the exception of Concurrent Sorted Linked List). The benchmarks in the parallelism category are mostly designed to test the capability to exploit parallel hardware efficiently and hence the workload implemented by each (re)actor is more significant (with the exception of Radix Sort).

The interested reader may find the full LF implementation of all our benchmarks on GitHub.[5] In the remainder of this section, we discuss implementation details for selected benchmarks that we consider representative.

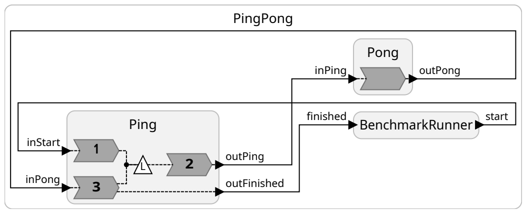
The execution of all benchmarks in the original Savina suite is governed by an actor called `BenchmarkRunner`. The benchmark runner is responsible for initiating a benchmark run and for measuring the time until each benchmark run completes. This enables performing measurements in repeated iterations while keeping caches (and the JVM in case of Akka) warm. We adopt this mechanism in our LF implementations and created the `BenchmarkRunner` reactor shown in Fig. 12a. The runner has two ports which are used to initiate a benchmark run (`start`) and for receiving feedback from the actual benchmark when it completed its computation (`finished`).

In our LF benchmarks, we created a reactor for each actor in the original implementation and a connection for each message that can be send between actors. For instance, Fig. 12b shows our implementation of the Ping Pong benchmark. The benchmark consists of two (re)actors `Ping` and `Pong` that send each other messages back and forth. When the `Ping` reactor in our implementation receives a message on the `inStart` port, it schedules a new event using its internal action. The reaction triggered by this action then sends the first ping message. `Pong` reacts to this message by sending a

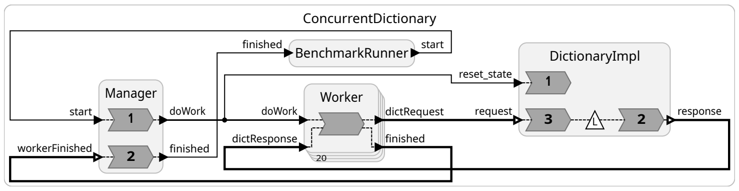
[4]The original Savina suite lists Producer Consumer as a concurrency benchmark, but we find it fits better to the group of parallelism benchmarks.
[5]https://github.com/lf-lang/benchmarks-lingua-franca



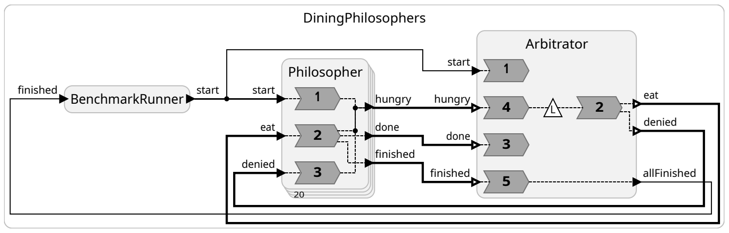
(a) Benchmark Runner



(b) Ping Pong



(c) Concurrent Dictionary



(d) Dining Philosophers

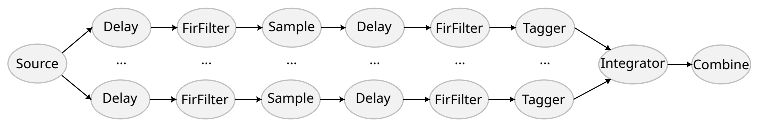
Fig. 12. Lingua Franca implementations of the benchmark runner and selected benchmarks.

pong message back to `Ping`, which in turn reacts by scheduling a new event on the internal action to repeat the process. Once all 1,000,000 ping and pong messages have been sent, the `Ping` reactor does not schedule a new event, but instead notifies the benchmark runner to indicate that the benchmark execution is completed.

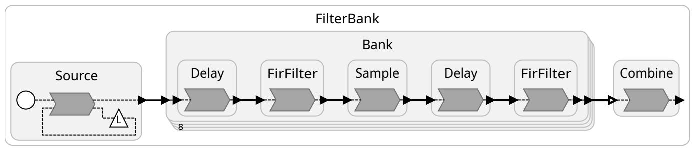
Note the use of the logical action to break the dependency cycle between `Ping` and `Pong`. If we would merge reactions 2 and 3 of `Ping` to send another ping message right when receiving a pong message, there would be a causality loop. We break the loop by scheduling a new event and sending the ping message at the next tag. All of the Savina benchmarks have a direct feedback loop and, thus, we carefully inserted logical actions where needed to break dependency cycles.

The concurrency benchmarks are particularly interesting as we can utilize LF's semantics to implement them efficiently. The Concurrent Dictionary benchmark, for instance, consists of a `Dictionary` (re)actor that receives read or write requests from 20 `Worker` (re)actors. The dictionary processes each request and sends a reply back to the workers. Fig. 12c shows our LF implementation. It instantiates a bank of workers that communicate with the dictionary via multiports. The workers operate concurrently, and each invocation of the worker reaction is logically simultaneous to the other workers. In consequence, the dictionary will receive multiple logically simultaneous requests from the workers. This notion of logical simultaneity allows us to effectively batch-process all the requests received at a single tag in a single reaction. The dictionary reaction iterates over all present `request` and processes the requests sequentially.

In an actor implementation of the Concurrent Dictionary benchmark, however, the dictionary could only process individual requests as there is no notion of simultaneity. Thus the runtime needs to invoke the actor behavior repeatedly, which adds additional overhead. Moreover, the particular order in which requests are processed in an actor



(a) Actor implementation



(b) Reactor implementation in LF

Fig. 13. Comparison between the reactor and the actor implementation of the Filter Bank benchmark.

implementation is nondeterministic. Since the workers send interleaving read and write requests, they may observe different responses depending on the order in which requests are processed. LF's notion of logical time establishes a deterministic ordering between messages, and allows to observe all the present inputs at a given tag at once.

We can make a similar observation for the Dining Philosophers benchmark. Our implementation in Fig. 12d uses an arbitrator reactor and a bank of 20 philosopher reactors. The philosophers *think* and *eat* concurrently. In order to start eating, philosophers send a `hungry` message to the arbitrator which replies with `eat` or `denied`. When a philosopher finishes eating, they indicate this with a `done` message. When the request to eat was denied, the philosopher will send a new `hungry` message. While the philosophers operate concurrently, the arbitrator can process all logical simultaneous `hungry` requests in one batch. In this concrete benchmark, this has the additional advantage that the arbitrator always knows which philosophers are hungry at a particular tag and can therefore find a fair strategy to grant the resources to the philosophers. In an actor implementation, the arbitrator can only decide on the basis of individual messages, which makes it much harder to find a fair solution. The original Savina implementation "solves" this simply by having each philosopher send another `hungry` message immediately after receiving `denied`. This increases the chance of each philosopher to eat eventually, but it also adds a significant amount of unnecessary messages. In our measurements, the philosophers sent about 10 million `hungry` messages in the Akka implementation, whereas our LF implementation used about 200,000 `hungry` messages. Of course it would be possible to implement other, more elaborate, arbitration strategies with actors, but compared to the Lingua Franca solution, this would always come with additional cost in terms of code size and also overhead for additional messages.

The LF implementation of Dining Philosophers could even be further simplified. Since there is no delay between sending an `eat` message in reaction 2 of the arbitrator, eating in reaction 2 of the philosopher, and processing the `done` message in reaction 3 of the arbitrator, all three steps are logical simultaneous. Since our scheduler first completes processing all reactions at the current tag before moving to the next tag (cf. Section 4.3), the `done` message is redundant. When the arbitrator is invoked to make a decision at a particular tag, we know that all philosophers must have completed eating at the previous tag. However, we decided to keep the `done` message to avoid deviating too much from the original benchmark implementation. This also allows for alternative implementations of the philosopher reactor, which might use a delay internally and send `done` messages at a later tag.

The advantage of LF's synchronous semantics also becomes evident in the Filter Bank benchmark shown in Fig. 13. It applies a cascade of filters to 8 parallel channels in a data stream. The output of each filter bank is then combined into a single stream. The combine operation is applied on the *n*th output message of each bank. This is trivial in LF, as the output messages are logically synchronous. The actor implementation, however, requires an additional protocol to

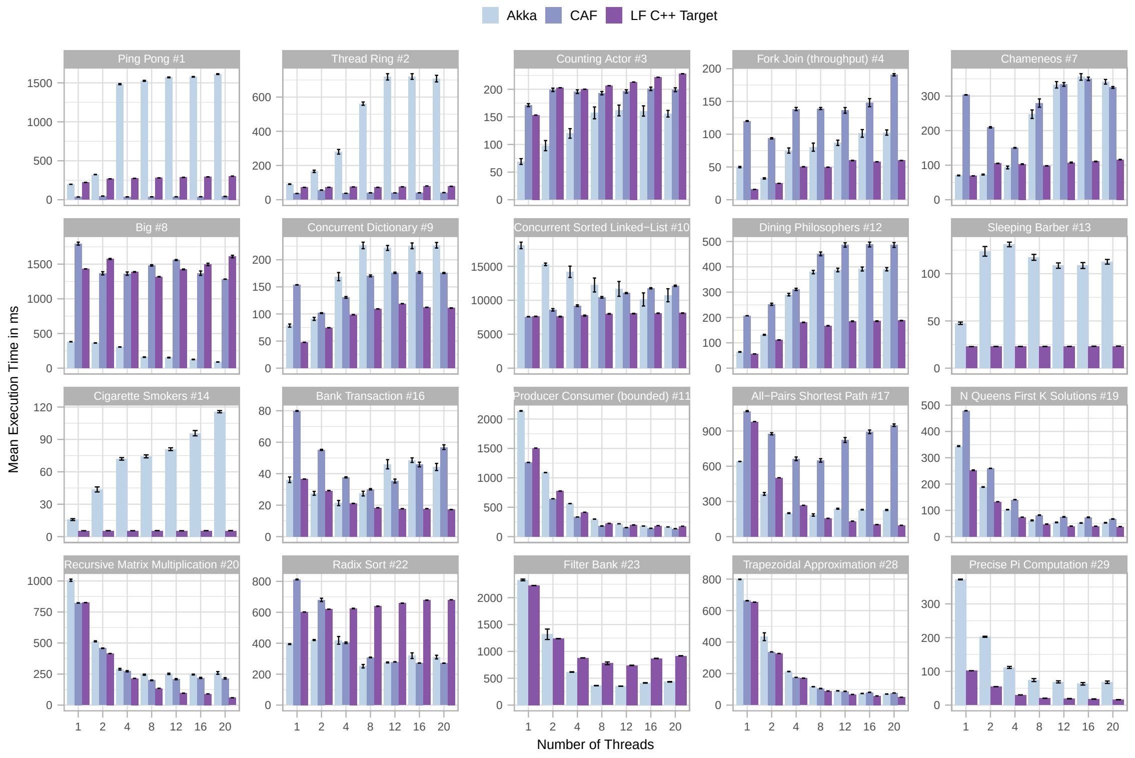


Fig. 14. Mean execution times and 99% confidence intervals for various Savina benchmarks implemented in LF, CAF, and Akka, measured for a varying number of worker threads. The numbers prefixed with # are benchmark IDs as listed in [41].

explicitly synchronize the outputs of the asynchronously operating banks. The original Savina implementation utilizes an additional `Tagger` actor that annotates the output messages of each bank with a tag indicating the ID of the bank. A so-called `Integrator` actor buffers the tagged messages from all banks. Once it receives a complete set of messages from all banks, it forwards them as one message to the `Combine` actor. As this synchronization mechanism is fully redundant in LF, we have removed it from our implementation of the benchmark.

5.3 Measurement results and discussion

Fig. 14 reports measured results for all supported benchmarks obtained with Akka, CAF, and the C++ target of LF. The plots show the mean execution times (including 99% confidence intervals) for a varying number of worker threads for each of the benchmarks. Not all benchmarks are implemented in CAF and hence it is missing in some plots.

The first six plots in Fig. 14 belong to the group of micro benchmarks in the Savina suite. Overall, our C++ runtime shows comparable performance to Akka and CAF. In Ping Pong and Thread Ring, our implementation is considerably faster than Akka but is still outperformed by CAF. For Counting Actor and Big, Akka performs better and the LF performance is slightly behind CAF. In Fork Join and Chameneos, the LF implementation outperforms both Akka and CAF, especially when using a larger number of worker threads.

The next six plots (Concurrent Dictionary to Bank Transaction) belong to the group of concurrency benchmarks. LF significantly outperforms CAF and Akka in all the concurrency benchmarks (especially for a high number of worker threads). This highlights how concurrent behavior is expressed naturally in LF and can be executed efficiently. As discussed in the previous subsection, we can exploit the well-defined notion of logical simultaneity in LF to execute independent reactions in parallel, and batch-process simultaneous messages from multiple reactors in a single reaction. Moreover, no explicit synchronization is needed. In the actor benchmarks, explicit synchronization (e.g., by sending acknowledge messages), or through blocking calls add additional overhead.

The remaining plots belong to the group of parallelism benchmarks in the Savina suite. Radix Sort and Filter Bank are affected by inefficiency in our scheduler, as discussed in Section 4.3. In these particular benchmarks, our simple algorithm leads to a non-optimal execution as some reactions are executed later than they could. We will revise this algorithm in future work. However, the remaining parallelism benchmarks highlight that LF can efficiently implement parallel algorithms. Our LF implementations are on par with Akka and CAF and scale well with more threads.

On average, LF outperforms both Akka and CAF. For 20 threads, the C++ runtime achieves a speedup of $1.85x$ over Akka and a $1.42x$ speedup over CAF. These speedups were calculated using the geometric mean over the speedups of individual benchmarks. We conclude that LF can compete with and even outperform modern and highly optimized actor frameworks such as Akka and CAF. Particularly with workloads that require synchronization, LF significantly outperforms actor implementations. LF is as efficient as the actor frameworks in exploiting parallelism and scales well to a larger thread count. In summary, the deterministic concurrency provided by LF does not hinder performance but enables more efficient implementations. This is possible in part because the scheduler has insights into the program structure, and explicit synchronization can be avoided in LF, as opposed to many of Savina actor benchmarks.

The performance comparison between C++ and Scala (Akka) needs to be taken with care, as other factors such as different library implementations and the behavior of the JVM may influence performance. For instance, the large discrepancy between Akka and our implementation in the Pi Precision benchmark is explained by a less efficient representation of large numbers in Scala/Java. However, the other benchmarks of the Savina suite do not depend on external libraries and are designed to be more portable between languages. Also note that over all benchmarks CAF only achieves an average speedup of $1.09x$ over Akka for 20 threads and is outperformed in 9 out of 16 benchmarks. For single-threaded execution, Akka outperforms CAF in 10 benchmarks and achieves an average speedup of 1.33x. This indicates that the implemented Scala workloads are comparable to the C++ implementations. Even considering a potential skew due to the JVM, our results clearly show that LF can compete with state-of-the-art actor frameworks.

To better understand the impact of the optimizations discussed in Section 4.3, Fig. 15 also shows the speedup of our runtime for 20 worker threads compared to a less optimized runtime. This baseline is an older version of our runtime that is optimized in the sense that we used code profiling to identify obvious bottlenecks and eliminated them using common code optimization techniques, but that does not include the optimizations discussed in this paper. The average overall speedup (geometric mean) achieved by our optimizations is $2.18x$. In particular, Big and Bank Transaction significantly benefit from our optimization for sparse communication patterns. The concurrency benchmarks (e.g., Concurrent Dictionary and Dining Philosophers), are mostly improved by reducing the contention on shared resources. However, not all benchmarks benefit from our optimizations. The reduced performance in Ping Pong and Counting Actor shows that optimizing for efficient parallel execution also comes at a cost for simple sequential programs.

6 RELATED WORK

LF is closely related to the languages and frameworks evolved around Hewitt's actor model [1, 40], including Akka [79], CAF [21], Ray [69], Erlang [3], Rebeca [85], P [28] and Pony [23]. Also reactive programming techniques [5], as used in frameworks like ReactiveX [64] and Reactors.IO [77] but also in language-level constructs like event loops [90], are closely related to LF. While actors and reactive programming provide good resiliency and scalability, this comes at

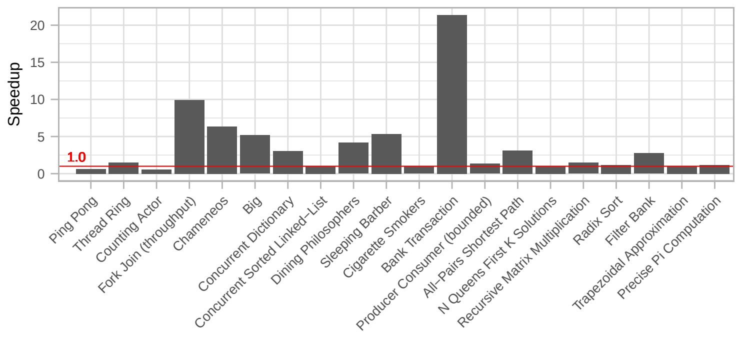


Fig. 15. Speedup achieved by our optimized C++ runtime for 20 worker threads compared to an unoptimized version.

the cost of nondeterminism, which makes programs notoriously hard to test and debug [6, 91]. Even more problems arise if languages, frameworks, and libraries do not enforce the underlying model and invite the programmer to break its semantics [88]. Pony addresses the later problem by leveraging a strong type system similar to Rust to prevent data races at compile time. Rebeca provides a formalism and model checking techniques for analyzing and verifying actor networks. While this can improve confidence in a correct implementation, the programmer is still responsible for finding this correct implementation. P goes a step further in that it also has an efficient runtime system and a compiler that generates correct-by-construction code with reasonable performance.

Blessing et al. propose a strategy that maps actor communication to a tree topology in order to guarantee a causal ordering of messages [15]. In a similar approach, Sang et al. utilize a DAG topology to achieve serializability in the processing of events. Orleans [17] is also based on an actor-like model and provides guarantees on atomicity on transactions. Finally, Reactors as defined by Field et al. [35] is a model that is closely related to actors but that supports both synchronous and asynchronous communication and also provides atomicity guarantees. All these strategies are most useful in distributed scenarios, in particular in presence of network failures. In this paper, however, we focus on the execution of a single host. Moreover, the determinism guarantees that LF makes are stronger. Nonetheless, such techniques are highly relevant to LF and could be deployed for ensuring fault-tolerant execution in distributed LF programs. We believe that LF can provides a more general solution, as the programmer can explicitly trade consistency for availability in distributed contexts [54], and hence the solution can be adjusted to the concrete application requirements.

Dataflow models [27, 11, 47] and process networks [42, 48] provide deterministic concurrency by creating statically connected networks of actors with deterministic semantics. Many tools and languages for modeling, analyzing and compiling dataflow models have evolved over time [20] including StreamIt [89], Sesame [76], OmpSs [29], PREESM [74] and MAPS [55, 19]. Similarly to LF, dataflow models enable improved static analysis and optimization [37], but in contrast to LF they also limit the application's flexibility and capability to react to sporadic events. For this reason dataflow is typically used for long running streaming applications, but not in scenarios where an application needs to react to sporadic input from the environment.

Since dataflow-based languages and tools also rely on statically declared connections, solutions comparable to the LF syntax extension that we propose in this paper have been created. StreamIt is particularly close to LF in spirit, due to its philosophy of using hierarchy for assembling larger programs out of smaller components. However, in connections cannot be drawn freely and programmers need to rely on pre-defined library nodes to implement certain patterns. The concept of multiports is also found in Ptolemy II [78], but it only defines a graphical syntax. Some tools use a more implicit approach. For instance, PREESM analysis the production and consumption rates of nodes and automatically decides to use multiple data-parallel instances of the same node. Ohua [33] has taken this to the extreme by generating a complete optimized dataflow graph implicitly from a sequential description of the program. Such implicit solutions, however, are challenging for LF as it pairs the dependency information defined in the graph with a clear timed semantics.

Deterministic concurrency is also found in synchronous languages such as Esterel [10], Lustre [38], and SIGNAL [9] as well as in Functional Reactive Programming (FRP) languages, like Fran [32], FrTime [25], and Elm [26]. However, these languages are challenging to use for general-purpose programming as they require pure functions and there is a lack of widely-applicable libraries. Only recently, side effects have been considered in a formal semantics for Esterel [36] and distributed dataflow [68]. In Lingua Franca, arbitrary code can be embedded in reactions and we can benefit from the available libraries for popular general-purpose languages.

Another interesting approach is taken by deterministic multithreading libraries such as DThreads [57] or Consequence [66], which enforce a total order for concurrent store operations. Recent work has made considerable progress in avoiding the bottlenecks of conventional DTM techniques [67]. However, we argue that threads are not a convenient concurrency model for the reasons outlined in [50]. Moreover, threads do not allow for transparent distributed execution as is possible with (re)actors.

The work presented in this paper is also closely related to research on efficient execution of programs on parallel hardware in general. However, due to the unique semantics of reactors, existing techniques cannot be easily applied to LF. In future work, we will aim at relaxing some of the constraints currently imposed in the runtime to allow sections of the program to execute more independently, which will likely give room to apply well known techniques such as work stealing [16] to efficiently balance the workload.

7 CONCLUSION

Unlike actors and related models for asynchronous concurrency, LF enforces determinism by default, and features asynchronous behavior only when introduced deliberately. Our evaluation, based on LF's C++ target, shows that this deterministic model does not impede performance. On the contrary, we achieve an average speedup of 1.85x over Akka and 1.42x over CAF. With LF, we manage to combine reproducible (and testable) behavior with good performance. Yet, our relatively simple scheduling strategy likely still leaves room for significant improvement. We leave it as future work to explore more advanced scheduling algorithms capable of exploiting more parallelism at runtime. We also aim to furnish full runtime support for mutations and implement the remaining Savina benchmarks that require them to evaluate the performance impact of mutations. Finally, we note that our implementation of the Savina benchmark suite is not only useful for comparing LF to actor-oriented frameworks; it also demonstrates that LF, which is still in its infancy, is already suitable for solving practical problems.

ACKNOWLEDGMENTS

We thank the anonymous reviewers of ACM TACO, PLDI, CGO and CC for the valuable feedback and suggestions they provided on earlier versions of the manuscript.

The work in this paper was supported in part by the German Federal Ministry of Education and Research (BMBF) as part of the Software Campus (01IS12051) and the program "Souverän. Digital. Vernetzt.", joint project 6G-life (16KISK001K). This work was also supported in part by the National Science Foundation (NSF), award #CNS-1836601 (Reconciling Safety with the Internet), the iCyPhy Research Center (Industrial Cyber-Physical Systems), supported by Denso, Ford, Siemens, and Toyota, and the National Research Foundation (NRF) of Korea (No. NRF-2022R1F1A1065201).

REFERENCES

[1] Gul A. Agha et al. 1997. A foundation for actor computation. *Journal of Functional Programming*, 7, 1, 1–72.

[2] 2018. *A review of dynamic scheduling algorithms for homogeneous and heterogeneous systems.* (2018), 73–83. ISBN: 978-981-10-8533-8.

[3] Joe Armstrong et al. 1996. *Concurrent programming in Erlang.* (Second ed.). Prentice Hall. ISBN: ISBN 0-13-508301-X.

[4] Mehdi Bagherzadeh et al. 2020. Actor concurrency bugs: a comprehensive study on symptoms, root causes, api usages, and differences. In *Proc. ACM Program. Lang.*

[5] Engineer Bainomugisha et al. 2013. A survey on reactive programming. *ACM Comput. Surv.*, 45, 4, (2013), 52:1–52:34.

[6] Herman Banken et al. 2018. Debugging data flows in reactive programs. In *2018 IEEE/ACM 40th International Conference on Software Engineering (ICSE).* IEEE, 752–763.

[7] Soroush Bateni et al. 2022. Xronos: predictable coordination for safety-critical distributed embedded systems. (2022).

[8] Albert Benveniste and Gérard Berry. 1991. The synchronous approach to reactive and real-time systems. *Proceedings of the IEEE*, 79, 9, 1270–1282.

[9] Albert Benveniste and Paul Le Guernic. 1990. Hybrid dynamical systems theory and the SIGNAL language. *IEEE Tr. on Automatic Control*, 35, 5, 525–546.

[10] Gérard Berry and Georges Gonthier. 1992. The Esterel synchronous programming language: design, semantics, implementation. *Science of Computer Programming*, 19, 2.

[11] G. Bilsen et al. 1994. Static scheduling of multi-rate and cyclo-static dsp applications. In *Workshop on VLSI Signal Processing.* IEEE Press.

[12] Nathan Binkert et al. 2011. The gem5 simulator. *ACM SIGARCH computer architecture news*, 39, 2, 1–7.

[13] David C. Black et al. 2010. *SystemC: From the ground up, Second Edition.* Springer. ISBN: 978-0-387-69957-8.

[14] Sebastian Blessing et al. 2019. Run, actor, run: towards cross-actor language benchmarking. In *Proceedings of the 9th ACM SIGPLAN International Workshop on Programming Based on Actors, Agents, and Decentralized Control* (AGERE 2019). Association for Computing Machinery, Athens, Greece, 41–50. ISBN: 9781450369824.

[15] Sebastian Blessing et al. 2017. Tree topologies for causal message delivery. In *Proceedings of the 7th ACM SIGPLAN International Workshop on Programming Based on Actors, Agents, and Decentralized Control* (AGERE 2017). Association for Computing Machinery, Vancouver, BC, Canada, 1–10. ISBN: 9781450355162.

[16] R.D. Blumofe and C.E. Leiserson. 1994. Scheduling multithreaded computations by work stealing. In *35th Annual Symposium on Foundations of Computer Science.*

[17] Sergey Bykov et al. 2011. Orleans: cloud computing for everyone. In *Proceedings of the 2nd ACM Symposium on Cloud Computing* (SOCC '11). Association for Computing Machinery, Cascais, Portugal. ISBN: 9781450309769.

[18] C. G. Cassandras. 1993. *Discrete Event Systems, Modeling and Performance Analysis.* Irwin.

[19] Jeronimo Castrillon and Rainer Leupers. 2014. *Programming Heterogeneous MPSoCs: Tool Flows to Close the Software Productivity Gap.* Springer, 258. ISBN: 978-3-319-00675-8.

[20] Anupam Chattopadhyay et al., (Ed.) 2022. *Dataflow models of computation for programming heterogeneous multicores. Handbook of Computer Architecture.* Springer Nature Singapore, (2022), 1–40. ISBN: 978-981-15-6401-7.

[21] Dominik Charousset et al. 2016. Revisiting Actor Programming in C++. *Computer Languages, Systems & Structures*, 45, (2016), 105–131.

[22] Moo-Kyoung Chung et al. 2014. Simparallel: a high performance parallel systemc simulator using hierarchical multi-threading. In *2014 IEEE International Symposium on Circuits and Systems (ISCAS)*, 1472–1475.

[23] Sylvan Clebsch et al. 2017. Orca: gc and type system co-design for actor languages. *Proc. ACM Program. Lang.*, 1, OOPSLA, (2017).

[24] Eric C Cooper and Richard P Draves. 1988. C threads. Tech. rep. CMU-CS-88-154.

[25] Gregory H Cooper and Shriram Krishnamurthi. 2006. Embedding dynamic dataflow in a call-by-value language. In *European Symposium on Programming.* Springer.

[26] Evan Czaplicki and Stephen N Chong. 2013. Asynchronous functional reactive programming for GUIs. In *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation-PLDI'13.* ACM Press.

[27] Jack B. Dennis. 1974. First Version Data Flow Procedure Language. Report MAC TM61. MIT Laboratory for Computer Science.

[28] Ankush Desai et al. 2012. P: Safe Asynchronous Event-Driven Programming. Report. Microsoft Research, (2012).

[29] Alejandro Duran et al. 2011. Ompss: a proposal for programming heterogeneous multi-core architectures. *Parallel processing letters*, 21, 02, 173–193.

[30] Stephen Edwards and John Hui. 2020. The sparse synchronous model. In *Forum for Specification and Design Languages (FDL), Kiel, Germany, Sep. 15–17.* IEEE, 1–8.

[31] Stephen A. Edwards and Edward A. Lee. 2003. The semantics and execution of a synchronous block-diagram language. *Science of Computer Programming*, 48, 1, 21–42.

[32] Conal Elliott and Paul Hudak. 1997. Functional reactive animation. In *ACM SIGPLAN Notices* number 8. Vol. 32. ACM, 263–273.

[33] Sebastian Ertel et al. 2018. Supporting fine-grained dataflow parallelism in big data systems. In *Proceedings of the 9th International Workshop on Programming Models and Applications for Multicores and Manycores*, 41–50.

[34] Patrick Th. Eugster et al. 2003. The many faces of publish/subscribe. *ACM Computing Surveys*, 35, 2, 114–131.

[35] John Field et al. 2009. Reactors: a data-oriented synchronous/asynchronous programming model for distributed applications. *Theoretical Computer Science*, 410, 2, 168–201.

[36] Spencer P Florence et al. 2019. A calculus for esterel: if can, can. if no can, no can. *Proceedings of the ACM on Programming Languages*, 3, POPL, 1–29.

[37] Marc Geilen et al. 2005. Minimising buffer requirements of synchronous dataflow graphs with model checking. In *Design Automation Conference (DAC).* ACM, 819–824.

[38] Nicholas Halbwachs et al. 1991. The synchronous data flow programming language lustre. *Proceedings of the IEEE*, 79, 9, 1305–1320.

[39] Brandon Hedden and Xinghui Zhao. 2018. A comprehensive study on bugs in actor systems. In *Proceedings of the 47th International Conference on Parallel Processing* (ICPP 2018). Association for Computing Machinery, Eugene, OR, USA. ISBN: 9781450365109.

[40] Carl Hewitt. 2010. Actor model of computation: scalable robust information systems. *arXiv preprint arXiv:1008.1459.*

[41] 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* (AGERE! '14). Association for Computing Machinery, Portland, Oregon, USA, 67–80. ISBN: 9781450321891.

[42] Gilles Kahn. 1974. The semantics of a simple language for parallel programming. In *Proc. of the IFIP Congress 74.* North-Holland Publishing Co., 471–475.

[43] Gilles Kahn and D. B. MacQueen. 1977. Coroutines and networks of parallel processes. In *Information Processing.* B. Gilchrist, (Ed.) North-Holland Publishing Co., 993–998.

[44] Dieter Kranzlmüller and Martin Schulz. 2002. Notes on nondeterminism in message passing programs. In *Recent Advances in Parallel Virtual Machine and Message Passing Interface.* Dieter Kranzlmüller et al., (Eds.) Springer Berlin Heidelberg, Berlin, Heidelberg, 357–367. ISBN: 978-3-540-45825-8.

[45] Yu-Kwong Kwok and Ishfaq Ahmad. 1999. Static scheduling algorithms for allocating directed task graphs to multiprocessors. *ACM Comput. Surv.*, 31, 4, (1999), 406–471.

[46] Ivan Lanese et al. 2018. Cauder: a causal-consistent reversible debugger for erlang. In *Functional and Logic Programming.* John P. Gallagher and Martin Sulzmann, (Eds.) Springer International Publishing, Cham, 247–263. ISBN: 978-3-319-90686-7.

[47] E. A. Lee and D. G. Messerschmitt. 1987. Synchronous data flow. *Proceedings of the IEEE*, 75, 9, 1235–1245.

[48] E. A. Lee and T. M. Parks. 1995. Dataflow process networks. *Proceedings of the IEEE*, 83, 5, 773–801.

[49] Edward A. Lee. 2021. Determinism. *ACM Transactions on Embedded Computing Systems (TECS)*, 20, 5, (2021), 1–34.

[50] Edward A. Lee. 2006. The problem with threads. *Computer*, 39, 5, 33–42.

[51] Gérard Huet et al., (Eds.) 2009. *The semantics of dataflow with firing. From Semantics to Computer Science: Essays in memory of Gilles Kahn.* Cambridge University Press.

[52] Edward A. Lee and Haiyang Zheng. 2007. Leveraging synchronous language principles for heterogeneous modeling and design of embedded systems. In *EMSOFT*. ACM.

[53] Edward A. Lee et al. 2005. Causality interfaces and compositional causality analysis. In *Foundations of Interface Technologies (FIT), Satellite to CONCUR*.

[54] Edward A. Lee et al. 2021. Quantifying and generalizing the CAP theorem. *CoRR*, abs/2109.07771. arXiv: 2109.07771.

[55] Rainer Leupers and Jeronimo Castrillon. 2010. MPSoC programming using the MAPS compiler. In *Proceedings of the 2010 Asia and South Pacific Design Automation Conference* (ASPDAC '10). IEEE Press, Taipei, Taiwan, 897–902.

[56] Bozhen Liu et al. 2021. When threads meet events: efficient and precise static race detection with origins. In *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation*. Association for Computing Machinery, New York, NY, USA.

[57] Tongping Liu et al. 2011. Dthreads: efficient deterministic multithreading. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles* (SOSP '11). Association for Computing Machinery, Cascais, Portugal, 327–336. ISBN: 9781450309776.

[58] Marten Lohstroh. 2020. *Reactors: A Deterministic Model of Concurrent Computation for Reactive Systems*. Ph.D. Dissertation. EECS Department, UC Berkeley, (2020).

[59] Marten Lohstroh and Edward A. Lee. 2019. Deterministic actors. In *Forum on Specification and Design Languages (FDL)*. (2019).

[60] Marten Lohstroh et al. 2019. Reactors: a deterministic model for composable reactive systems. In *8th International Workshop on Model-Based Design of Cyber Physical Systems (CyPhy'19)*. Vol. LNCS 11971. Springer-Verlag, 27.

[61] Marten Lohstroh et al. 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.

[62] Jason Lowe-Power et al. 2020. The gem5 simulator: version 20.0+. *CoRR*, abs/2007.03152. arXiv: 2007.03152.

[63] Aman Shankar Mathur et al. 2018. Idea: an immersive debugger for actors. In *Proceedings of the 17th ACM SIGPLAN International Workshop on Erlang* (Erlang 2018). Association for Computing Machinery, St. Louis, MO, USA, 1–12. ISBN: 9781450358248.

[64] Erik Meijer. 2010. Reactive extensions (rx): curing your asynchronous programming blues. In *ACM SIGPLAN Commercial Users of Functional Programming* (CUFP '10). ACM, Baltimore, Maryland, 11:1–11:1. ISBN: 978-1-4503-0516-7.

[65] Christian Menard et al. 2020. Achieving determinism in Adaptive AUTOSAR. In *Proceedings of the Design, Automation and Test in Europe Conference (DATE)*.

[66] Timothy Merrifield et al. 2015. High-performance determinism with total store order consistency. In *Proceedings of the Tenth European Conference on Computer Systems* (EuroSys '15). Association for Computing Machinery, Bordeaux, France. ISBN: 9781450332385.

[67] Timothy Merrifield et al. 2019. Lazy determinism for faster deterministic multithreading. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems* (ASPLOS '19). Association for Computing Machinery, Providence, RI, USA, 879–891. ISBN: 9781450362405.

[68] Ragnar Mogk et al. 2019. A fault-tolerant programming model for distributed interactive applications. *Proceedings of the ACM on Programming Languages*, OOPSLA, 1–29.

[69] Philipp Moritz et al. 2017. Ray: A distributed framework for emerging AI applications. *CoRR*, abs/1712.05889. arXiv: 1712.05889.

[70] Madanlal Musuvathi et al. 2008. Finding and reproducing heisenbugs in concurrent programs. In *OSDI* number 2008. Vol. 8.

[71] Walid A. Najjar et al. 1999. Advances in the dataflow computational model. *Parallel Computing*, 25, 13-14, (1999), 1907–1929.

[72] Bruce Jay Nelson. 1981. *Remote Procedure Call*. Ph.D. Dissertation. USA.

[73] Preeti Ranjan Panda. 2001. Systemc: a modeling platform supporting multiple design abstractions. In *Proceedings of the 14th International Symposium on Systems Synthesis* (ISSS '01). Association for Computing Machinery, Montréal, P.Q., Canada, 75–80. ISBN: 1581134185.

[74] Maxime Pelcat et al. 2014. Preesm: a dataflow-based rapid prototyping framework for simplifying multicore dsp programming. In *2014 6th European Embedded Design in Education and Research Conference (EDERC)*, 36–40.

[75] R. Perrey and M. Lycett. 2003. Service-oriented architecture. In *2003 Symposium on Applications and the Internet Workshops*. IEEE, 116–119.

[76] Andy Pimentel et al. 2006. A systematic approach to exploring embedded system architectures at multiple abstraction levels. *IEEE Transactions on Computers*, 55, 2, 99–112.

[77] Aleksandar Prokopec. 2018. Pluggable scheduling for the reactor programming model. In *Programming with Actors: State-of-the-Art and Research Perspectives*. Alessandro Ricci and Philipp Haller, (Eds.) Springer International Publishing, 125–154. ISBN: 978-3-030-00302-9.

[78] Claudius Ptolemaeus. 2012. *System Design, Modeling, and Simulation Using Ptolemy II*. Ptolemy.org, Berkeley, CA, USA.

[79] Raymond Roestenburg et al. 2016. *Akka In Action*. Manning Publications Co. ISBN: 1617291013.

[80] Tim Schmidt et al. 2017. Exploiting thread and data level parallelism for ultimate parallel systemc simulation. In *Proceedings of the 54th Annual Design Automation Conference 2017* (DAC '17). Association for Computing Machinery, Austin, TX, USA. ISBN: 9781450349277.

[81] Christoph Schumacher et al. 2010. ParSC: synchronous parallel SystemC simulation on multi-core host architectures. In *Proceedings of the 8th International Conference on Hardware/Software Codesign and System Synthesis* (CODES/ISSS '10). ACM, 241–246. ISBN: 9781605589053.

[82] Koushik Sen. 2008. Race directed random testing of concurrent programs. *SIGPLAN Not.*, 43, 6, (2008), 11–21.

[83] Lui Sha et al. 2009. PALS: Physically Asynchronous Logically Synchronous Systems. Technical Report. Univ. of Illinois.

[84] Kazuhiro Shibanai and Takuo Watanabe. 2017. Actoverse: a reversible debugger for actors. In *Proceedings of the 7th ACM SIGPLAN International Workshop on Programming Based on Actors, Agents, and Decentralized Control* (AGERE 2017). Association for Computing Machinery, Vancouver, BC, Canada, 50–57. ISBN: 9781450355162.

[85] Marjan Sirjani et al. 2004. Modeling and verification of reactive systems using rebeca. *Fundam. Inform.*, 63, 4, 385–410.

[86] Samira Tasharofi et al. 2013. Bita: coverage-guided, automatic testing of actor programs. In *28th International Conference on Automated Software Engineering (ASE)*, 114–124.

[87] Samira Tasharofi et al. 2012. Transdpor: a novel dynamic partial-order reduction technique for testing actor programs. In *Formal Techniques for Distributed Systems*. Holger Giese and Grigore Rosu, (Eds.) Springer Berlin Heidelberg, Berlin, Heidelberg, 219–234. ISBN: 978-3-642-30793-5.

[88] Samira Tasharofi et al. 2013. Why do scala developers mix the actor model with other concurrency models? In *European Conference on Object-Oriented Programming*.

[89] William Thies et al. 2002. Streamit: a language for streaming applications. In vol. 2304. (2002). ISBN: 978-3-540-43369-9.

[90] Stefan Tilkov and Steve Vinoski. 2010. Node. js: using javascript to build high-performance network programs. *IEEE Internet Computing*, 14, 6, 80–83.

[91] Carmen Torres Lopez et al. 2019. Multiverse Debugging: Non-Deterministic Debugging for Non-Deterministic Programs (Brave New Idea Paper). In *33rd European Conference on Object-Oriented Programming (ECOOP 2019)*, 27:1–27:30.

[92] Alessandro Ricci and Philipp Haller, (Eds.) 2018. *Programming with actors: state-of-the-art and research perspectives*. Springer International Publishing, Cham. Chap. A Study of Concurrency Bugs and Advanced Development Support for Actor-based Programs, 155–185.

[93] Tom Van Cutsem et al. 2014. Ambienttalk: programming responsive mobile peer-to-peer applications with actors. *Computer Languages, Systems & Structures*, 40.

[94] Reinhard von Hanxleden et al. 2022. Pragmatics twelve years later: a report on lingua franca. In *11th International Symposium on Leveraging Applications of Formal Methods*.

[95] Jixiang Yang and Qingbi He. 2018. Scheduling parallel computations by work stealing: a survey. *Int. J. Parallel Program.*, 46, 2, (2018), 173–197.

[96] Bernard Zeigler. 1976. *Theory of Modeling and Simulation*. Wiley Interscience, New York.