Efficient and Deterministic Record & Replay for Actor Languages

Dominik Aumayr
Johannes Kepler University
Linz, Austria
dominik.aumayr@jku.at

Stefan Marr
University of Kent
Canterbury, United Kingdom
s.marr@kent.ac.uk

Clément Béra
Vrije Universiteit Brussel
Brussel, Belgium
clement.bera@vub.be

Elisa Gonzalez Boix
Vrije Universiteit Brussel
Brussel, Belgium
egonzale@vub.be

Hanspeter Mössenböck
Johannes Kepler University
Linz, Austria
hanspeter.moessenboeck@jku.at

ABSTRACT

With the ubiquity of parallel commodity hardware, developers turn to high-level concurrency models such as the actor model to lower the complexity of concurrent software. However, debugging concurrent software is hard, especially for concurrency models with a limited set of supporting tools. Such tools often deal only with the underlying threads and locks, which obscures the view on e.g. actors and messages and thereby introduces additional complexity.

To improve on this situation, we present a low-overhead record & replay approach for actor languages. It allows one to debug concurrency issues deterministically based on a previously recorded trace. Our evaluation shows that the average run-time overhead for tracing on benchmarks from the Savina suite is 10% (min. 0%, max. 20%). For Acme-Air, a modern web application, we see a maximum increase of 1% in latency for HTTP requests and about 1.4 MB/s of trace data. These results are a first step towards deterministic replay debugging of actor systems in production.

CCS CONCEPTS

• **Computing methodologies** → **Concurrent programming languages**; • **Software and its engineering** → *Software testing and debugging*;

KEYWORDS

Concurrency, Debugging, Determinism, Actors, Tracing, Replay

ACM Reference Format:
Dominik Aumayr, Stefan Marr, Clément Béra, Elisa Gonzalez Boix, and Hanspeter Mössenböck. 2018. Efficient and Deterministic Record & Replay for Actor Languages. In *15th International Conference on Managed Languages & Runtimes (ManLang'18), September 12–14, 2018, Linz, Austria.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3237009.3237015

1 INTRODUCTION

Debugging concurrent systems is hard, because they can be non-deterministic, and so can be the bugs one tries to fix. The main challenge with these so called *Heisenbugs* [13], is that they may manifest rarely and may disappear during debugging, which makes them hard to reproduce and to fix.

McDowell and Helmbold [27] distinguish two broad categories of debuggers for finding and fixing bugs: traditional breakpoint-based debuggers and event-based debuggers. Event-based debuggers see a program execution as a series of events and abstract away implementation details. Commonly such event traces are used for post-mortem analyses. However, they can also be used to reproduce program execution, which is known as *record & replay*. With record & replay it is possible to repeat a recorded execution arbitrarily often. Therefore, once a program execution with a manifested bug was recorded, the bug can be reproduced reliably. This makes such bugs easier to locate even though many executions may need to be recorded to capture the bug.

Record & replay has been investigated in the past [8] for thread-based programs or message-passing systems, at least since the 1980s [10]. However, debugging support for high-level concurrency models such as the actor model has not yet received as much attention [40]. As a result, there is a lack of appropriate tools, which poses a maintenance challenge for complex systems. This is problematic because popular implementations of the actor model, such as Akka[1], Pony [9], Erlang [1], Elixir [38], Orleans [7], and Node.js[2], are used to build increasingly complex server applications.

Debugging support for the actor model so far focused either on breakpoint-based debuggers with support for actor-specific inspection, stepping operations, breakpoints, asynchronous stack traces, and visualizations [4, 25], or it focused on postmortem debugging, e.g. Causeway [37], where a program's execution is analyzed after it crashed. While specialized debuggers provide us with the ability to inspect the execution of actor programs, they do not tackle non-determinism. However, to the best of our knowledge, existing record & replay approaches for actor-based systems focus either on single event loop environments [2, 6] or have not yet considered the performance requirements for server applications [35].

In this paper, we present an *efficient* approach for recording & replaying concurrent actor-based systems. By tracing and reproducing the ordering of messages, recording of application data can be limited to I/O operations. To minimize the run-time overhead, we

ManLang'18, September 12–14, 2018, Linz, Austria

[1] *Akka website*, https://akka.io/
[2] *Node.js website*, https://nodejs.org/

determine a small set of events needed to replay actor applications. We prototype our approach on an implementation of communicating event loop actors [28] in SOMns. SOMns is an implementation of Newspeak [5] on top of the Truffle framework and the Graal just-in-time compiler [43]. Furthermore, we provide support for recording additional detailed information during replay executions, which can be used in the Kómpos debugger [25] for visualizations or post-mortem analyses.

We evaluate our approach with SOMns. Using the Savina micro-benchmark suite [17], we measure the tracing run-time overhead and the trace growth rate for each benchmark. On the Acme-Air web application [41], we measure the latency with and without tracing, and the total trace size recorded.

The contributions of our approach are:

(1) Deterministic replay of actor applications using high-level messaging abstractions,
(2) Capture of non-deterministic data to deal with external inputs,
(3) Scalability to a high number of actors and messages.

2 TOWARDS EFFICIENT DETERMINISTIC REPLAY FOR ACTOR LANGUAGES

In deterministic programs, the result of an execution depends only on its input. Thus, reproducing an execution is straightforward, provided the environment is the same. In practice, it is often necessary to debug a program multiple times before the root cause of a bug is found. This approach to debugging is called *cyclic debugging* [27]. As convenient as cyclic debugging is, it requires bugs to be reproducible reliably. This makes it unsuitable for non-deterministic programs, where the occurrence of a bug may depend on a rare scheduling of messages.

As mentioned before, record & replay [8] enables deterministic re-execution of a previously recorded program execution, and thereby enables cyclic debugging also for non-deterministic programs. During the initial execution, such approaches record a program trace, which is then used during replay to guide the execution and reproduce, for instance, input from external sources and scheduling decisions, and thereby eliminate all non-determinism.

Record & replay for parallel and concurrent programs has been studied before, but a majority of the previous work focused on shared memory concurrency and MPI-like message passing [8]. Recent work focused either on single event loops or did not consider performance [2, 6, 35]. Thus, none of the approaches that we are aware of support efficient deterministic record & replay for modern actor-based applications.

The remainder of this section considers the practical requirements for an efficient deterministic record & replay system. Furthermore, it provides the necessary background on actor-based concurrency and considers the limitations of record & replay systems.

2.1 Practical Requirements for Record & Replay

Since modern actor systems such as Akka, Pony, Erlang, Elixir, Orleans, and Node.js are widely used for server applications, we aim at making it practical to record the execution of such applications. In such an environment, bugs might occur rarely and could be triggered by specific user interactions only. We assume that development happens on commodity hardware, so that the issues can be reproduced and debugged on a developer's laptop or workstation.

Based on this scenario, we consider two main concerns. First, the recording should have minimal run-time overhead to minimize the effect on possible Heisenbugs. Second, the amount of recorded data should be small enough to fit either into memory or on a commodity storage. For comparison, Barr et al. [2] reported a maximal tracing overhead of 2% for their single event loop Node.js system and 4-8 seconds of benchmark execution. The produced trace data is less than 9 MB. Burg et al. [6] report 1-3% run-time overhead and in the worst case 65%. Their benchmarks execute for up to 26 seconds and produce up to 700 KB of traces.

To make our system practical, we aim to achieve a similarly small run-time overhead while tracing multiple event loops. However, in parallel actor applications, we need to account for much higher degrees of non-determinism. This means that the run-time overhead is likely larger. Additionally, run-time overhead can scale with the tracing workload, for instance, message intensive programs may have a higher overhead than computationally intensive ones. Thus, our goal is:

> **Goal 1**
> *The run-time overhead of tracing for server applications should be in the 0% to 3% range. Worst-case run-time overhead, e.g. for message intensive programs, should be below 25%.*

Since we aim at supporting long-running actor-based server applications, the reported trace sizes do not directly compare to our scenario. Furthermore, they are based on single event loops, which have much lower event rates. Since we assume that some bugs might be induced by user interactions, we want to support executions of multiple minutes and perhaps up to half an hour. Considering that contemporary laptops have about 500 GB of storage, this would mean an execution should produce no more than about 250 MB/s of trace data. Therefore, our second goal is:

> **Goal 2**
> *Recording should produce well below 250 MB/s of trace data.*

2.2 Communicating Event Loop Actors

This section provides the background on actor-based concurrency to detail the challenges of designing an efficient record & replay mechanism for actor languages, and our contributions.

The actor model of concurrency was first proposed by Hewitt et al. [15]. By now, diverse variations have emerged [11]. We focus on the communicating event loops (CEL) variant pioneered by the language E [28]. The CEL model exhibits all relevant characteristics of actor models and combines event loops with high-level abstractions, like non-blocking promises, which represent a challenge for deterministic replay, as we detail below. This class of actor models has been later adopted by languages such as AmbientTalk [42] and Newspeak [5], and also corresponds to the asynchronous programming model of JavaScript and Node.js [39].

The general structure of CEL is shown in fig. 1. Each actor is a container of objects isolated from the others, a mailbox, and an

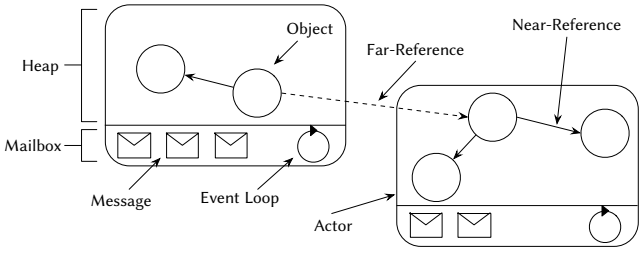

Figure 1: Overview of CEL model. Each actor consists of a thread of execution (an event loop), a heap with regular objects, and a mailbox. An event loop processes incoming messages in a serial order from its mailbox. An actor can directly access and mutate objects it owns. All communication with objects owned by other actors happens asynchronously via far references.

event loop. The event loop processes messages from its mailbox one-by-one in order of arrival. When a message is processed, the receiver object is identified and the method corresponding to the message is invoked. The processing of one message by an actor defines a *turn*. Since actors have isolated state and messages are handled atomically with respect to other messages, the non-determinism of the system is restricted to the order in which messages are processed.

To maintain the state of each actor isolated from the other actors, each actor only has direct access to the objects it owns. Communication with objects owned by other actors happens via *far references*. Far references do not support synchronous method invocation nor direct access to fields of objects. Instead, they can only receive asynchronous message sends, which are forwarded to the mailbox of the actor owning the object. Objects passed as arguments in asynchronous message sends are parameter-passed either by far reference, or by (deep) copy.

An asynchronous message send immediately returns a *promise* (also know as a future). A promise is a placeholder object for the result that is to be computed. Once the return value is computed, it is accessible through the promise, which is then said to be *resolved* with the value. The promise itself is an object, which can receive asynchronous messages. Those messages are accumulated within the promise and forwarded to the result value once it is available.

Other actor variants have different semantics for message reception and whether they support (non-blocking) promises. Note, however, that the queuing on non-blocking promises introduces additional non-determinism compared to other actor variants. Thus, they are the most challenging variant for deterministic replay.

2.3 Record & Replay for Actors

As mentioned before, record & replay has been investigated before [8]. Ronsse et al. [31] categorizes such approaches into content-based and ordering-based replay based on what type of data is recorded. We now describe their characteristics and applicability to actor-based concurrency.

Content-based Replay. Content-based replay is based on recording the results of all operations that observe non-determinism, and returning the recorded results during replay. In the context of shared memory concurrency, this means that all reads from memory accessed by other threads need to be captured. A representative example of such an approach is BugNet [29].

In the context of actor-based concurrency, it is necessary to record all kinds of events received by actors. To the best of our knowledge, there exist only three approaches providing record & replay for actor-based concurrency: Jardis [2], Dolos [6] and Actoverse [35]. They can be categorized as content-based replay. Actoverse provides record & replay for Akka programs and records messages exchanged by actors including message contents. Dolos does record & replay for JavaScript applications running in a browser, and Jardis for both the browser and Node.js. Both Dolos and Jardis capture all non-deterministic interactions within a single event loop, i.e. interactions with JavaScript/Node.js APIs.

Ordering-based replay. Ordering-based replay (also known as control-based replay) focuses on the order in which non-deterministic events occur. The key idea is that by reproducing the control-flow of an execution, the data is implicitly reproduced as well. This means that only data needed to reproduce the control-flow has to be recorded, producing smaller traces in the process. An early implementation of ordering-based replay is Instant replay [20], which maintains version numbers for shared memory variables. However, ordering-based replay does not work when a program has non-deterministic inputs. For such programs, ordering-based replay can be used for internal non-determinism, combined with content-based replay for non-deterministic inputs.

In actor-based concurrency, since the non-determinism of the system is restricted to the order in which messages are processed, it is only necessary to reproduce the message processing order of an actor. Ordering-based replay has not been explored for actor-based concurrency, but there is work for message passing interface (MPI) libraries. MPL* [18] is an ordering-based record & replay debugger for MPI communication. MPL* records the sequence of message origins (senders). This is enough information to reproduce the ordering of messages for MPI communication, since messages from the same source are race-free, i.e., they arrive in the order they were sent in. Another ordering-based record & replay approach for MPI is Reconstruction of Lamport Timestamps (ROLT) [32]. Like MPL*, ROLT assumes messages from the same source being race-free. It then uses Lamport clocks in all actors, and records when a clock update is larger than one time step. These "jumps" are caused by communication with actors that have a different clock value, which synchronizes the Lamport clocks. In replay, the sending of a message is delayed until all messages with smaller timestamps have been sent.

2.4 Problem Statement

The existing record & replay approaches discussed above leave three issues that need to be solved for actor-based concurrency.

Issue 1: Deterministic replay of high-level messaging abstractions. Existing record & replay approaches typically only record the sequence of messages to reproduce the message order. MPL* for example only records message senders, while Actoverse records

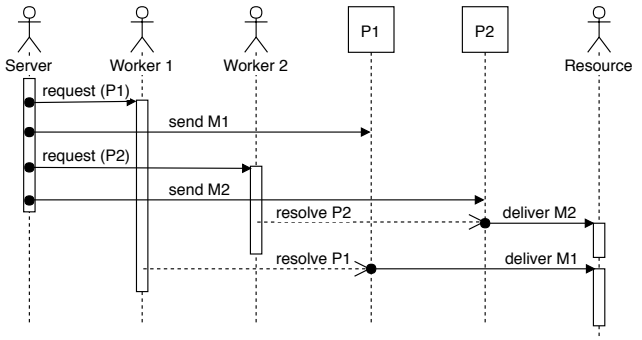

Figure 2: Promise issue with the MPL* approach, message `M2` is able to overtake message `M1`.

message contents as well. Unfortunately, message sender and content are not enough to reproduce the original message ordering in the presence of high-level messaging abstractions such as promises.

Figure 2 gives an example of a scenario where replay using only message sender information would not suffice for an actor-based language, because it is not eliminating all non-determinism. The `Sever` actor creates two promises `P1` and `P2` and then sends a `request` message with promise `P1` as an argument to the `Worker1` actor, and a message `M1` to promise `P1`. This is repeated with `Worker2`, `P2` and `M2`. In our example, `Worker2` resolves `P2` to `Resource`, causing the message `M2` stored in `P2` to be delivered to it. Later, `Worker 1` also resolves its promise (`P1`) to the same `Resource`, and message `M1` is delivered. Despite being sent first, `M1` is processed after `M2`, as in our scenario the processing order depends on which promise is resolved first. This makes the message ordering non-deterministic when there is a race on which promise is resolved first.

In short, MPL* and Actorverse cannot reliably replay a program similar to the scenario of fig. 2, as `Server` is the sender of both messages, and replay cannot distinguish between `M1` and `M2`.

Issue 2: Recording non-deterministic input. As stated before, pure ordering-based replay cannot deal with non-determinism caused by external inputs. Ordering-based replay variants devised for MPI programs can deal with one source of non-determinism: messages exchanged between processes. In particular, MPL* does not trace non-deterministic contents of messages and as such, it does not support replay of I/O operations.

On the other hand, content-based replay variants devised for JavaScript's event loop concurrency can deal with non-determinism caused by external input. Jardis [2] is able to trace systems calls and I/O. Dolos [6] captures all I/O, user input, and other sources of non-determinism, such as timers for JavaScript programs. However, both Jardis and Dolos only support a single event loop.

It is thus an open issue to support both types of non-determinism for actor-based concurrency: message non-determinism (MPL*) and non-deterministic interactions within a turn (Jardis, Dolos).

Issue 3: Scale. With content-based replay, the trace contains enough information to make replay of individual actors in isolation possible. This can be useful when the origin of a bug has been narrowed down to a few actors, the behavior of which can then be examined in detail without being distracted by the rest of the system. However, the set of problematic actors is usually unknown beforehand, rendering the approach often impractical, as it does not offer deterministic replay of all the actors in a system. We also expect high overhead for content-based replay both in execution time and memory footprint since more events need to be recorded, for example, messages exchanged between actors.

Ordering-based replay approaches proposed in the context of message passing libraries (MPL) seem better suited for actors. To the best of our knowledge, there is no existing performance comparison between the two flavors, MPL* and MPL-ROLT. However, MPL-ROLT suffers from scalability issues when applied to large-scale systems, since it needs to update the clock of a message sender, when the receiver's clock is greater.

This back-propagation of clocks works in the context of MPI, where mandatory ACK can be used. Also, the sender requires synchronization of its mailbox to avoid clock updates from received messages while waiting for the ACK response. Blocking the mailbox while sending a message may be problematic given the larger number of actors and messages found in actor programs.

Even though MPL replay approaches provide a starting point for replaying actor-based concurrent programs, they assume a coarse-grained granularity of processes and sparse use of message-based communication. In contrast, actors are very lightweight and are commonly used on a very fine-grained level, comparable to objects. As such, a large number of actors can be created per VM. Not only does this imply that the traffic generated by messages is higher than in MPL programs, but also that tracing needs to be optimized for events such as actor creation, messages, and I/O.

3 DETERMINISTIC REPLAY FOR ACTORS

The following sections present our solution to the non-determinism of high-level messaging abstractions, input from external sources, and the scale and granularity of actor systems.

The effects of high-level messaging abstractions, such as promises, are replayed by recording and using additional information, which is discussed in the remainder of this section.

To handle non-deterministic input, we propose a design that distinguishes between synchronous and asynchronous inputs to fit well with the actor model (section 4). Finally, to handle fine-grained actor systems, we use a compact trace format that can be recorded with a low run-time overhead and generates traces with manageable sizes (section 5).

3.1 High-level Architecture

To achieve deterministic replay, we record the necessary information to replicate the message execution order of an execution precisely. To this end, we record actor creation, the message processing order, and external non-determinism, i.e., input data.

As mentioned previously, there is a wide range of different actor systems [11]. However, some actor systems use similar implementation strategies to gain efficiency. While they are not a precondition for our approach, they can influence the efficiency of tracing. One common optimization used by many actor runtimes is that actors are scheduled on threads arbitrarily, possibly using a thread pool. This means actors are not bound to a specific thread.

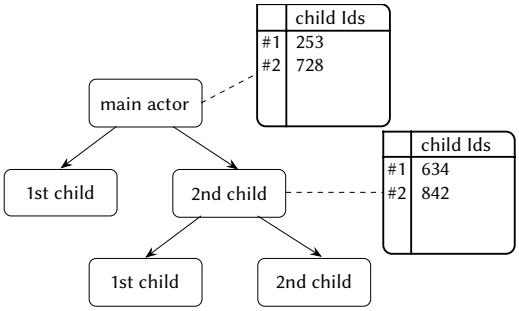

Figure 3: Actor family tree in replay. Actors know which Ids to assign a new child actor.

Another common optimization is that message are processed in batches to avoid making the actor mailbox a synchronization bottleneck. Thus, a thread that executes the actor can take the actor's mailbox, replace it with an empty one, and then starts executing the messages in the mailbox without having to synchronize again.

In section 5, we utilize this property to avoid redundancy in subtraces that correspond to a batch of messages.

To minimize the perturbation introduced by tracing, we decouple the event recording from the writing to a file. While it is possible to store data actor-local, doing so causes memory overhead to scale with the number of actors, which is problematic for fine-grained actor-based concurrency. Consequently, each thread that executes actors uses thread-local buffers to store the recorded events. One buffer records the generic events. The other buffer records external data. When the buffers are full, they are handed to a thread that writes them to a file (cf. section 6.2). The recording itself is also optimized as discussed in section 5 and section 6.1. The resulting trace file can then be used to replay the whole execution within a new process.

3.2 Identifying Actors

For recorded events, we need to know on which actor they happened. For this purpose, each actor is assigned a unique integer Id (`ActorId`). To correctly assign traced data to actors during replay, our technique has to reproduce the assignment of actor Ids. To this end, we consider the actors that an actor spawned to be its children. We record actor creation in our trace, so that we can determine the Ids of an actor's children. Using the creation order, we can reassign Ids correctly in replay.

The main actor, which is created when the program starts, is always assigned the same Id. We can therefore identify it and use it as a basis for identifying all its child actors. For each actor, we keep track of how many children it created so far. When a new actor is created during replay, we use the actor family tree shown in fig. 3 to look up the Id that has to be assigned to the new actor.

3.3 Messages & Promise Messages

For replaying normal messages, we have to record the Ids of their senders just as MPL* does. However, as shown by fig. 2 and discussed in issue 1 of section 2.4, this is insufficient to replay high-level messaging abstractions such as promises. We solve the issue by recording the actor that resolved the promise, i.e., caused a so-called promise message to be delivered.

With this additional information we are able to distinguish messages that would otherwise appear identical, as for instance in a MPL* replay. In the example of fig. 2, we now know which worker is responsible for which message, and can therefore ensure that they are processed in the same order as in the original execution.

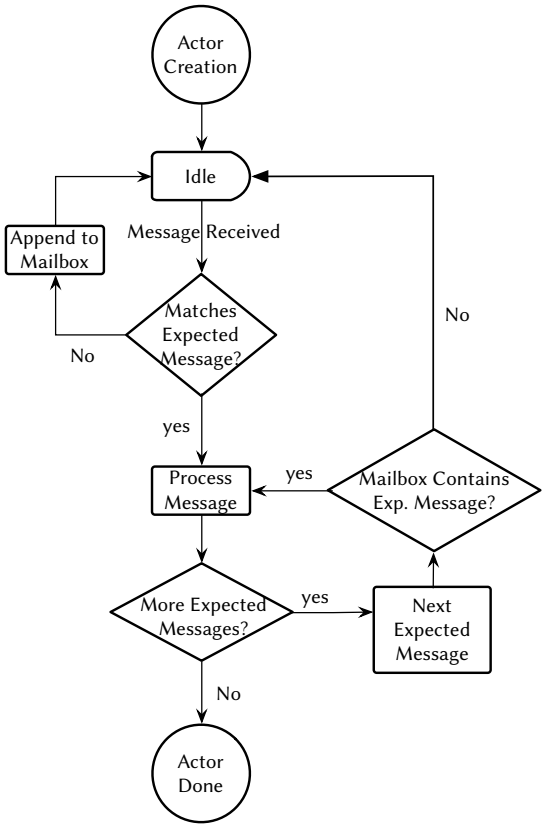

Figure 4: Behavior of actors during replay. To reproduce the message order, actors check if the message type, sender, and (for promise messages) resolver of a message matches the expected message. Only a matching message is executed, and mismatches delayed until they match.

3.4 Replay

When a program is started in replay mode, the interpreter loads the trace file and starts executing the program. Instead of relying on the normal actor implementation, it uses an implementation specifically adapted to replay the trace exactly.

During replay, each actor holds a queue of recorded information that represents the message order to reproduce. We call the head of this queue the *expected message*. The expected message is either a normal message or a promise message. To be processed, a received message needs to match this type. For normal messages, the received message also needs to have the same sender Id. Similarly, for a promise message the received message needs to have the same sender and resolver Ids as the expected message.

Figure 4 shows how actors behave in replay executions. The way an actor handles an incoming message depends on whether it is currently idle or processing a message. An idle actor will check if the received message has the sender and possibly resolver Id of the expected message. If it does, the new message will be processed right away. Otherwise, the message is appended to the mailbox. When an actor is busy and receives a message, the message is simply appended to the mailbox. When a busy actor finishes processing a message, it will peek at the next expected message in the queue, and then iterate through the mailbox in search for a matching message. If a match is found, the message is processed, otherwise the actor becomes idle and stops processing messages until a matching message is received.

4 CAPTURING EXTERNAL NON-DETERMINISM

Most programs interact with their environment, the effects of which can be non-deterministic. For instance, in an HTTP server that receives requests and reacts to them, the request order determines the program behavior. Another example for external non-determinism are system calls to get the current time. Hence, capturing such inputs is essential for deterministic replay.

We distinguish two ways non-determinsim is introduced by such interactions: system calls and asynchronous data sources. System calls are interactions with the environment that directly return a result, such as getting the current system time, or checking whether a file exists. Asynchronous data sources are more complex and introduce non-determinism through an arbitrary number of messages that are pushed as result of a non-deterministic event. For example, an incoming HTTP request can cause a message to be sent to an actor.

During recording, all interactions with the environment are performed and the data needed to return results or send messages is recorded. Each operation's data is assigned an Id that is used to reference it, and is written to a data file.

To enable tracing with minimal run-time overhead and storage use, we leave the decision what and how to record to the implementers of data sources. Hence, the tracing mechanism for external data is general enough to be used for a wide range of use cases.

4.1 System Calls

The system call approach targets synchronous interactions with non-deterministic results, which are recorded. All system calls are expected to be implemented as basic operations in the interpreter and are executed synchronously without sending a message. This means that they happen as part of a turn.

Each system call needs to be carefully considered for tracing, to prevent external data from leaking into the program uncontrolled. Critical objects on which the system calls operate (e.g. a file handle) need to be wrapped, and have to be completely opaque. Otherwise the program can access external data that is not replayed. This means that all operations that involve the wrapped object are either system calls or only access fields of the wrapper.

As a result of the tracing, we get an ordered sequence of system calls for each actor as well as the data that came from each of these calls. By reproducing the order of events for an actor, we also reproduce the order of performed system calls. Hence, the result of the n-th system call by an actor is referenced by the n-th system call event in the trace.

```
1  public boolean pathExists(String path) {
2    if (REPLAY) {
3      return getSystemcallBoolean();
4    }
5    boolean result = Files.exists(path);
6    if (TRACING) {
7      recordSystemCallBoolean(result);
8    }
9    return result;
10 }
```

Figure 5: Simplified example for the implementation of a path exists system call.

When an actor performs a system call in replay, the `DataId` of the queues head is used to get the recorded data. The system call then processes that data, instead of interacting with the environment, and thus returns the same result as in the original execution.

The implementation of system calls is straightforward. Figure 5 is a simple example for a system call that checks whether a path represented by a string exists in the file system. In the Java implementation of the system call, we insert two if clauses, the first one (lines 2-4) is placed before the existence is actually checked. During replay, it will get the result from the original execution and return it immediately, bypassing the rest of the method. The existence check is performed in line 6 and the result is stored in a variable `result`. Finally, the second if clause (lines 6-8) is responsible for recording the result when tracing is enabled. The infrastructure adds a system call event to the trace and records the result in a separate data trace.

Our design focuses on the reproduction of the returned result, but it is general enough to allow reproduction of other effects a system call may have on the program. For instance, a system call that also resolves a promise in addition to returning a value. In this case the recorded data has to contain both the result and the value used for promise resolution.

4.2 Asynchronous Data Source

Input data that is not handled with system calls is generally considered to come from some asynchronous data source. In an actor system, this means the external data source is typically represented by an actor itself and data is propagated in the system by sending messages or resolving promises. Thus, an actor wraps the data source and makes it available to other actors via messages.

Through this wrapping, the deterministic replay can rely in part on the mechanisms for handling messages and promises. However, we need to augment them to record the data from the external source when it becomes accessible to the application. These messages and promise messages that are sent as result of external events are marked as *external messages*. For example, a message sent to an actor that is triggered by an incoming HTTP request will be marked as external and will contain the data of the request. In the trace, these messages are marked as external as well and contain the data Id to identify the recorded data during replay. They also

contain a marker to identify the type of event for a data source. This is necessary because each data source may have multiple events of different kinds. The data itself is stored in a file separate from the traced events (cf. section 4.4 and section 5).

When an actor expects an external message during replay, it will not wait for a message, but instead simulate the external data source. Thus, it reads the recorded data associated with the sending actor and the data Id in the trace. With this information, the replay can resolve promises and send messages with the same arguments as during recording.

4.3 Combining Asynchronous Data Sources and System Calls to Record Used Data Only

Depending on the application that is to be recorded & replayed, it can be beneficial to avoid recording all external data and instead only record the data that influenced the application, i.e., was used by it. To this end, we can combine our notion of asynchronous data sources and system calls. We detail this idea using our example of an HTTP server, where an application might only inspect the headers sent by a client, but might not need the whole body of the request. Figure 6 gives an overview of how the system is structured to deal with such a scenario.

The HTTP server is considered an external data source and is thus represented by its own actor. Application actors can register to handle incoming HTTP requests on certain request paths, which is a pattern common to many web frameworks. The server handles the incoming HTTP requests, and then delegates them to the registered actor by sending a message.

A request itself can be modeled as an object, with which an application can interact, for instance, to read the header or to respond with a reply to the HTTP client.

To minimize the data that needs to be recorded, we model our data source for the HTTP server so that it creates a HTTP request object only with the minimal amount of data. The HTTP headers and body are only going to be recorded when they are accessed. Therefore, during recording, the initial incoming HTTP request only leads to the recording of the kind of HTTP request that was made, e.g., a get or post request, and the request path. This information is needed to identify the callback handlers that an application actor registered on the HTTP data source.

As detailed in section 4.2, triggering the callback handler is done via an external message. Thus, the recorded message contains the `DataId`, which references the kind of HTTP request made and its path. During replay, when the HTTP server is created, its actor has the same `ActorId` as in the recorded execution, and the same callbacks are registered by the same actors in the same order. When the application actor expects to receive the external message, it looks up the data source (HTTP server) based on the sender's `ActorId`, and requests the simulated event. Thus, the data source recreates the HTTP request object based on the `DataId`. When an application accesses for instance the HTTP headers and body at a later point, we handle these as system calls. Thus, during recording the header data is written into the trace, and read from the trace file during replay.

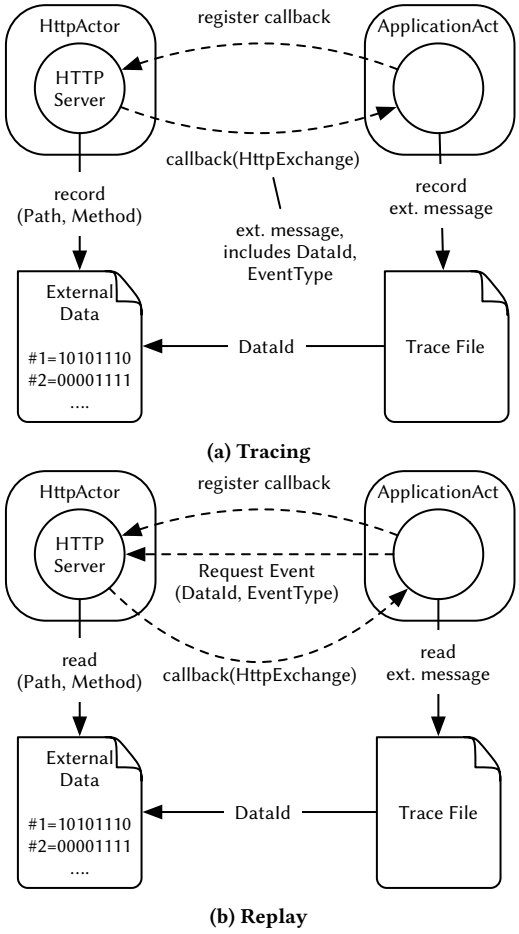

Figure 6: Data flows of an HTTP server during tracing and replay. Information about an incoming request is recorded in the trace, this event is reproduced in replay on request of the ApplicationActor.

4.4 Format for External Data

External data that is recorded for external events and system calls is stored in a separate trace file using a binary format. The file has a simple structure of consecutive entries with variable length. Each entry starts with a 4-byte `ActorId` for the origin of the entry. It is followed by the 4-byte `DataId`, which is referenced by the trace entries for external messages and system calls. The length of the payload is encoded also with 4-byte field. The combination of `ActorId` and `DataId` allows it to identify a specific entry globally.

5 COMPACT TRACING

To encode trace events, we use a binary format that can be recorded without introducing prohibitive run-time overhead. As mentioned in section 3.1, we also need to account for actors being executed on different threads over time. Both aspects are detailed below.

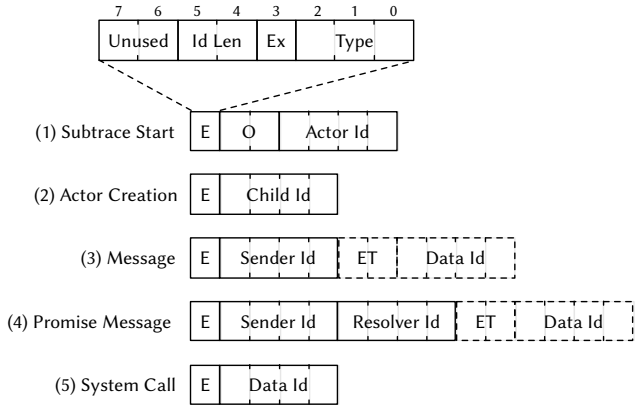

Figure 7: Sketch of the encoding of trace entries with 4-byte Ids. The `EventType` and `DataId` fields of messages and promise messages are only needed when they are marked as external in the header.

5.1 Subtraces

Since actors can be scheduled on different threads over time, and we use thread-local buffers to record events, we need to keep track of the actor that performed the events. To avoid having to record the actor for each event, we start a new *substrace* when an actor starts executing on a thread. Similarly, when a buffer becomes full, a new subtrace is started.

To minimize run-time overhead, we use thread-local buffers that are swapped only when they are full. This however means that an actor could execute on one thread, and then on another, and the buffer of the second thread could be written to the file before the first one. Thus, we need to explicitly keep track of an ordering of subtraces. For this reason, actors maintain a counter for the subtraces. We record it as a 2-byte Id as part of the start of subtraces. For well-behaved actor programs, the buffers are written in regular intervals and 2-byte Ids provide sufficient safety even with overflows to restore the original order.

5.2 Trace Format

Our compact binary trace format uses a one-byte header to encode the details of a trace entry, and then encodes entry-specific fields. The bits in the `E` event header encode the type of the entry, whether a message is marked as external, and the number of bytes that are used for Ids. Figure 7 visualizes the encoding.

By encoding the Ids with flexible length, we can reduce the trace size significantly (cf. section 6.1 and section 7). Ideally, it means that an Id smaller than 256 can be encoded in a single byte, one smaller 65536 in two bytes, and so on.

As discussed in the previous section, we need to record the start of subtraces, actor creation, messages, promise messages, and system calls. Their specific fields are as follows:

(1) Subtrace Start. A subtrace start indicates the beginning of each subtrace to associate all events within it with the given `ActorId`. `O` is the 2-byte ordering Id to restore the correct order of subtraces before replaying them.

(2) Actor Creation. The actor creation entries correspond to when a child actor is spawned. It includes the Id of the new actor (`Child Id`) so that we can construct the parent-child tree of actors for replay and reassign Ids to each actor.

(3) Message & (4) Promise Message. Message entries correspond to the messages processed by the actor of a subtrace. The `SenderId` identifies the actor that sent the message. Promise messages also include the `ResolverId` to identify the actor that resolved the promise.

External messages are marked by the `Ex` bit in the event header and record `EventType` (ET in fig. 7) and `DataId`. The `EventType` identifies the kind of external event, e.g., an HTTP request. It is used to distinguish different kind of events from the same source. The 4-byte `DataId` references the data for the external event.

(5) System Call. System call entries record the `DataId` to identify the data. Note that the order of trace entries is in most cases sufficient to recreate a mapping during replay. Identifiers are only introduced for cases where the ordering is insufficient.

6 IMPLEMENTATION

We implemented our record & replay solution for communicating event-loop actors in SOMns. SOMns is written in Java as a self-optimizing abstract syntax tree (AST) interpreter [44] using the Truffle framework and Graal just-in-time compiler [43]. This allows us to integrate record & replay directly into the language implementation. The tracing is added as nodes that specialize themselves (i.e. optimize) based on the inputs they encounter. This means, the tracing is compiled together with the application code and executes as highly optimized native code, which reduces run-time overhead. Our implementation optimizes recording of Ids and delegates the writing of trace data to a background thread, which we detail below.

6.1 Optimized Recording of Ids

As seen in section 5.2, identifiers (Ids) are the main payload for trace entries. Thus, efficient recording of Ids is crucial for performance. To minimize the trace size, we decided to encode them in smaller sizes if possible. However, in a naive implementation this would increase the run-time overhead significantly, because for each Id we would need to check how to encode it resulting in complex control flow possibly limiting compiler optimizations.

With the use of self-optimizing nodes, we can avoid much of the complexity of writing Ids. A program location that for instance spawns an actor can thus specialize to the value range of Ids it has seen. To minimize the overhead, a node specializes to the value range that fits all previously seen Ids. Thus, if only an Id 34 has been recorded, the node specializes to check that the Id matches the 1-byte Id range and to write it. If Ids 34 and 100,000 has been seen, the node specializes to check that the Id can be stored in 3 bytes and writes it. In case an Id is encountered that does not fit into the given number of bytes, the node replaces itself with a version that can write longer Ids. This will also invalidate the compiled code, and eventually result in optimized code being compiled.

While this approach does not achieve the smallest possible trace size, it reduces the run-time overhead. We evaluate the effectiveness of our optimization and its effect on the performance in section 7.

6.2 Buffer Management

For our tracing of regular events, we use the following thread-local buffer approach as described by Lengauer et al. [21]. By using thread-local buffers, we avoid synchronization for every traced event. Buffers that are not currently used by a thread are stored in two queues, one containing full, and the other containing empty buffers. When a thread's trace buffer does not have enough space for another entry, it is appended to the full queue, and the thread takes its new buffer from the empty queue. The full queue is processed by a background thread that writes the trace to a file.

For external data, we use separate buffers and a separate queue. As external data can be of any size, we allocate buffers on demand and discard them when they are no longer needed.

The *writer thread* that persists the trace also processes the queue for external data. Once a buffer is written, it is added to the queue of empty buffers for trace data or discarded for external data.

To avoid slowing down application threads with serialization and conversion operations, they are done by the writer thread. The application threads hand over the data without copying whenever it is safe to do so. For instance, for our HTTP server data source, this is possible because most data is represented as immutable strings. Data sources that use complex objects use serializers that are handed over to the writer thread together with the data. This makes it possible to persist also complex data on the writer thread.

7 EVALUATION

This section evaluates the run-time performance and trace sizes of our implementation in SOMns using the Savina benchmark suite for actors [17], and Acme-Air as an example for a web application. We also use the *Are We Fast Yet* benchmarks to provide a baseline for the SOMns performance [24].

7.1 Methodology

As SOMns uses dynamic compilation, we need to account for the VM's warmup behavior [3]. The Savina and Are We Fast Yet benchmarks run for 1000 iterations within the same VM process using ReBench [23]. Since we are interested in the peak-performance of an application with longer run times, we discount warmup. We do this by inspecting the run-time plots for all benchmarks, which indicates that the benchmarks stabilize after 100 iterations.

For Acme-Air, we use JMeter [14] to produce a predefined workload of HTTP requests. The workload was defined by the Node.js version of Acme-Air. JMeter is configured to use two threads to send a mix of ca. 42 million randomly generated requests based on the predefined workload pattern. After inspecting the latency plots, we discarded the first 250,000 requests to exclude warmup.

The Savina and Are We Fast Yet benchmarks were executed on a machine with two quad-core Intel Xeons E5520, 2.26 GHz with 8 GB RAM, Ubuntu Linux with kernel 4.4, and Java 8.171. Acme-Air was executed on a machine with a four-core Intel Core i7-4770HQ CPU, 2.2 GHz, with 16 GB RAM, a 256 GB SSD, macOS High Sierra (10.13.3), and Java 8.161. In both cases, we used Graal version 0.41.

In section 2.1, we defined the performance goals of a tracing run-time overhead of less than 25% for message intensive programs, i.e., microbenchmarks such as from the Savina benchmark suite. Savina falls into this category as many of the benchmarks perform little computation, for instance, in the counting benchmark 200,000 messages are sent to an actor who increments a counter. Furthermore, we aimed for a tracing mechanism that produces under 250 MB/s of trace data to be practical on today's developer machines. For larger systems, which are not dominated by message sends, we aim for run-time overhead that is in the range of 0-3%.

To assess whether we reach these goals, we measure the overhead of tracing on the benchmarks while restricting the actor system to use a single thread. This is necessary to measure the actual tracing overhead. Since some benchmarks are highly concurrent, running on multiple threads can give misleading results. One issue is that some of these benchmarks have very high contention and any overhead in the sequential execution can result in a speedup in the parallel execution, because it reduces contention and the number of retries of failed attempts.

7.2 Baseline Performance of SOMns

To show that SOMns reaches a competitive baseline performance, and is a solid foundation for our research, we first compared it to Java, Node.js, and Scala.

The sequential performance of SOMns, as measured with the Are We Fast Yet benchmarks, is shown in fig. 8. While SOMns is not as fast as Java, it reaches the same level of performance as Node.js, which is used in production environments. This indicates that the sequential baseline performance of SOMns is competitive with similar dynamic languages.

To ensure that SOMns' actors are suitable for this work, we compare its actor performance with other actor implementations, based on the Savina benchmark suite.

Unfortunately, the benchmarks are designed for impure actor systems, such as Akka, Jetlang, and Scalaz. This means, some of the benchmarks rely on shared memory. Thus, we had to restrict our experiments to a subset of 18 benchmarks from the total of 28, as the other ones could not be ported to SOMns, because it does not support shared memory between actors.

The results of our experiments with Savina benchmarks are shown in fig. 9 and indicate that SOMns reaches the performance of other widely used actor systems on the JVM. Hence, it is a suitable foundation for this research.

7.3 Tracing Savina

Figure 10 shows the run-time overhead of tracing. It includes the results for recording full Ids, i.e. all Ids are recorded with 4 bytes, and the optimized version where the Ids are encoded with fewer bytes if possible (cf. section 6.1). The average overhead for tracing with full Ids is 9% (min. 0%, max. 18%). As seen in table 1, the benchmarks produce up to 109 MB/s of data. Applying our optimization for recording small Ids, the average overhead for tracing is only minimally higher with 10% (min. 0%, max. 20%). Furthermore, the maximal data rate goes down to 59 MB/s.

With these results, we fulfill our goal of having less than 25% overhead for programs with high message rates, and to produce less than 250 MB/s of trace data.

As seen from table 1, effectiveness of using *small Ids* depends on the benchmark. *TrapezoidalApproximation* for instance, has an insignificant reduction in trace size. Other benchmarks, such as

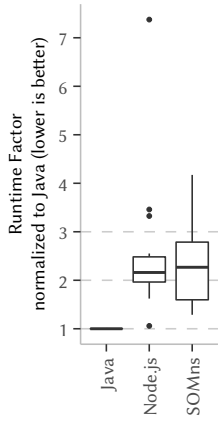

Figure 8: Performance comparison with other languages. SOMns performs similar to Node.js.

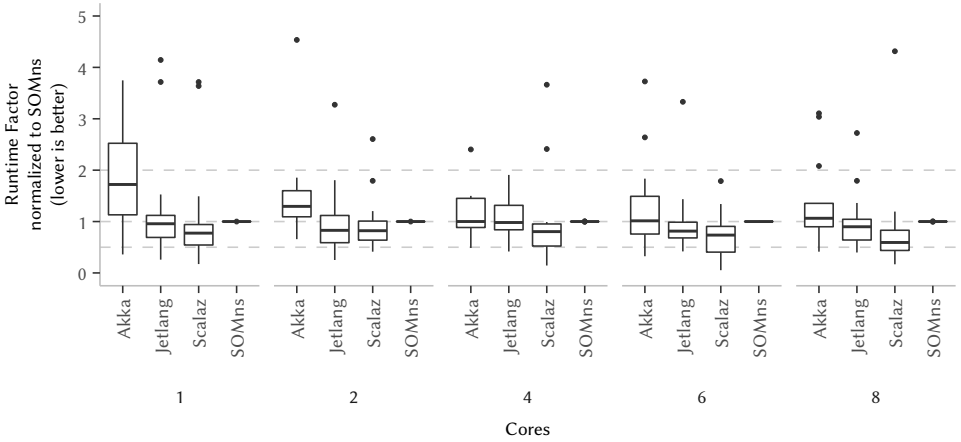

Figure 9: Performance of Savina benchmarks in different actor languages for different numbers of Cores.

Counting and *RadixSort*, have a near halved data-rate. Due to the minimal performance impact, we consider using small Ids beneficial.

Figure 10: Performance of traced executions of the Savina benchmarks using a single thread. Results are normalized to the untraced execution.

	Harmonic Mean MB/s	
	Small Ids	Full Ids
BankTransaction	19.11	23.87
BigContention	10.38	12.14
Chameneos	52.02	53.46
CigaretteSmokers	30.72	31.50
ConcurrentDictionary	0.02	0.03
ConcurrentSortedLinkedList	0.00	4.38
Counting	58.84	108.58
ForkJoinActorCreation	18.56	34.23
ForkJoinThroughput	7.96	25.44
LogisticMapSeries	30.64	49.00
Philosophers	44.52	71.26
PingPong	39.22	68.18
ProducerConsumerBoundedBuffer	5.29	0.02
RadixSort	36.29	66.44
SleepingBarber	55.88	70.42
ThreadRing	50.00	76.91
TrapezoidalApproximation	3.17	3.21
UnbalancedCobwebbedTree	19.60	20.02

Table 1: Trace production per second over 1000 benchmark iterations.

7.4 Tracing Acme-Air

Acme-Air is a benchmark representing a server application implemented with micro-services [41]. It models the booking system of a fictional Airline. Acme-Air is available on GitHub[3] for Java and Node.js. The JavaScript version of Acme-Air served as the basis for our SOMns port. We stayed true to the original design, in which a single event loop is used to process requests. Instead of using a stand-alone database, we used an embedded Derby[4] database. The database was reset and loaded with data before each benchmark to factor out its potential influence on the results.

JMeter measures the latency for each request it makes. Since the highest resolution is 1 ms, some results are rounded to 0 ms. The predefined workload uses different frequencies for the different possible requests seen in fig. 11. For instance, Query Flight is the most common one representing 46.73% of all requests.

[3] *Acme-Air repository*, https://github.com/acmeair/

[4] *Apache Derby*, https://db.apache.org/derby/index.html

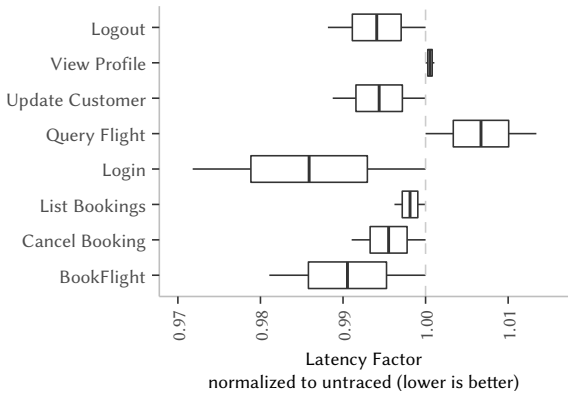

Figure 11: Latency of different request types in traced executions, normalized to untraced executions.

Figure 11 shows the effect of tracing on the latency of different requests. Tracing *small Ids* results in a maximum overhead of 1% (geomean. -1%, min. -3%). We consider average negative overhead, i.e. speedup, an artifact of dynamic compilation. The compiler heuristics can trigger slightly differently, which leads to differences in the applied optimizations, native code layout, and caching effects.

The trace size for a benchmark execution is 993 MB in total, with an observed data-rate of 1.4 MB/s. External data is responsible for the majority of the trace size (88%, 872 MB), while tracing of internal events accounts for 121 MB. Variations in trace size between different benchmark executions were negligible, i.e. below 1 MB.

The Acme-Air results suggest that real-word applications tend to have lower data-rates and overhead than most Savina benchmarks. The maximal overhead of 1% suggests, however, that tracing larger application has likely minimal impact and is thus practical, meets our goals, and is comparable to systems that are not optimized for the fine granularity and high message rate of actors.

8 RELATED WORK

This section discusses related work in addition to the record & replay approaches analyzed in section 2.3. We compare our solution to work that employs similar tracing techniques: record & refine, back-in-time debugging, shared memory, and profiling.

8.1 Record and Refinement

As explained in section 2, record & replay approaches record a program trace, which is then used during replay to guide execution and reproduce non-deterministic behavior (such as input from external sources and scheduling of messages) in a deterministic way. Such deterministic replay can then also provide access to more detailed information after the original execution finished. Felgentreff et al. [12] define this process as *record and refine*.

Record & refine enables low-overhead postmortem debugging. Thus, during recording, only the minimum necessary data to reproduce the desired parts of a program execution is recorded, i.e., to avoid non-determinism during replay. All additional data, for instance to aid debugging, can be obtained during replay execution. We apply the same idea to SOMns. During recording, only the minimal amount of information is retained and during replay, all features of the Kómpos debugger are supported.

8.2 Back-in-Time Debugging

Unlike record & replay, back-in-time debugging takes snapshots of the program state at certain intervals, and they offer time travel by replaying execution from the checkpoint before the target time.

Jardis. Jardis [2] provides both time-travel debugging and replay functionality for JavaScripts event loop concurrency. It combines tracing of I/O and system calls with regular heap snapshots of the event loop. It keeps snapshots of the last few seconds, allowing Jardis to go back as far as the oldest snapshot, and discard trace data from before that point. While this keeps the size of traces and snapshots small, it limits debugging to the last seconds before a bug occurs. This may be a problem as the distance between root cause and actual issue is typically large in concurrent programs [30].

Jardis reports a run-time overhead of $\leqslant$2% for compressed trace sizes of below 9 MB for 2-4 second runs. For Acme-Air, our approach has a data rate (1.4 MB/s) lower than the one of Jardis. As such, our impact on the performance of the benchmark is competitive.

Actoverse. Actoverse [35] also provides both time-travel debugging and record & replay for Akka actors. Unlike Jardis or our solution, Actoverse is implemented as a library and uses annotations to mark fields to be recorded. A snapshot of those fields is saved when sending and after processing messages. The order of messages and snapshots is determined with Lamport clocks to avoid a global clock. While performance is not reported, the memory usage is indicated with about 5 MB for 10,000 messages. Our ordering-based approach requires only about 2-15 byte per message.

CauDEr. CauDEr [19] is a reversible debugger for Erlang. It is able to undo actions and step backwards in the execution by relying on reversible execution semantics. CauDEr currently only addresses a subset of Erlang and focuses on the semantic aspects of reversible execution for debugging. Therefore, it can help in correctness considerations, but does not focus on enabling the debugging of larger systems as our work does.

8.3 Shared Memory

There is a lot of work on record & replay in the context of shared memory. Generally, shared memory record & replay reproduces the order of synchronization operations and accesses to shared memory. The used techniques are very diverse, as is their impacts on run-time performance, which can range from negligible overhead [22, 26] to 35x overhead [20]. Castor [26] can use transactional memory to improve its performance. iReplayer [22] records explicit synchronization, regularly creates snapshots of program state, and provides in-situ replay, i.e. within the same process. LEAP [16] uses static analysis to determine what is shared between threads. Unfortunately, static approaches can introduce synchronization on all operations with a shared field. For the actor model, this synchronization corresponds to one global lock for all mailboxes. In the actor model synchronization on mailboxes and promises are essential. They correspond roughly to the tracing in SOMns. Hence, shared-memory approaches conceptually have to record at least as many events as our actor tracing. However, the actor model

ensures that there are no races on shared memory, which would need to be traced. For pure actor models as in SOMns, shared memory approaches are therefor likely having the same or additional overhead. For impure actor models, it seems beneficial to find ways to combine actor and shared memory record & replay techniques.

8.4 Profiling

We now discuss related work in the context of profiling for actor-based concurrency.

Profiling of Akka Actors. Rosà et al. [33, 34] profile the utilization and communication of Akka actors based on platform-independent and portable metrics. An application collects profiling information in memory. On termination, it generates a trace file that can be analyzed to determine performance bottlenecks. It tracks various details including message counts and executed bytecodes. To attribute this information precisely, it maintains a shadow stack. In contrast to this, SOMns records the ordering of messages, their processing, and any external input. Since offline analysis is not a direct goal, SOMns does not need a shadow stack, but could provide such information during replay execution. For replaying, however, we need to record the events instead of just counting them. Since Rosà et al. [33] aimed for platform-independent and portable metrics, run-time performance was not a major concern. They observed a run-time overhead of about 1.56x (min. 0.93x, max. 2.08x).[5] However, these numbers include instrumentation overhead and do not directly compare to the overhead of long-running applications, which is probably much lower.

Large-scale Tracing. Lightbend Telemetry[6] offers a commercial tool for capturing metrics of Akka systems. The provided actor metrics are based on counters, rates, and times. For example, it records mailbox sizes, the processing rate of messages, and how much time messages spent in the mailbox. The run-time overhead can be finely adjusted by selecting the elements that are to be traced, and possibly a sampling granularity. This seems to be a standard approach for such systems and is also used by tracing systems based on the OpenTracing standard[7] or Google's Dapper [36]. However, since we want to eliminate all non-determinism, doing selective or sample-based tracing is not an option.

9 CONCLUSION AND FUTURE WORK

To better handle the complexity of concurrency and avoid dealing with threads and locks, developers embrace high-level concurrency models such as actors. Unfortunately, actor systems usually have a limited number of supporting tools, making them hard to debug. In this paper, we presented an efficient record & replay approach for actor languages letting the programmer debug non-deterministic concurrency issues by replaying a recorded trace.

Our approach is able to replay high-level messaging abstractions such as promises by recording extra information. Non-deterministic inputs are recorded and replayed deterministically. In addition, our approach scales to a high number of actors and exchanged messages through its low execution time overhead and its compact trace.

We evaluated the performance of our approach with the Savina benchmark suite, the average tracing run-time overhead is 10% (min. 0%, max. 20%). In the case of the modern web application Acme-Air, our approach showed a maximum increase in latency of 1% and about 1.4 MB/s of trace data.

Applicability to Actor Models. We argue that our approach is general enough to be applied to all forms of message processing (continuous/blocking and consecutive/interleaved). The main reason is that we record the order in which messages are processed. Thus, our approach is independent of any variation in selecting which message may be executed next and all selections, blocking, or interleaving already happened. Hence, all those mechanisms determining the message order in the original execution do not have to be reproduced, as we already have the final ordering, which is replayed in a re-execution. A requirement for our approach is, however, that actors are isolated and shared memory is not allowed because we do not track races on shared memory.

Future work: Long Running Applications. Although our approach is able to scale up in term of the number of actors and exchanged messages, it is currently not suitable for applications that run for extended periods of time. The trace recorded by our approach keeps growing as the program runs, at some point the trace will become too large for the disk. Besides the problem of growing traces, there is also the practical issue of replaying such a program. Replay of a program that has been running for such a long period of time will take a similar (or higher) amount of time.

One solution is to create snapshots of the programs state at regular intervals. Each time a snapshot is created, previous trace data can be discarded. Replay can then start at the last snapshot before a failure, and allows developers to investigate the cause. To minimize traces further, we could apply simple compression too.

Future work: Replay Performance. Currently, our replay implementation parses the entire trace on startup. This comes with a high memory overhead, and causes scalability issues with the employed data structures. Replay scalability can be improved by parsing the trace on-the-go. By dividing the parsing effort across the replay execution, startup time and memory overhead can be reduced.

Future work: Partial Replay. Partial replay of an execution can enable debugging and testing techniques, such as regression tests and exploration of different interleavings. The biggest challenge for partial replay is external non-determinism when switching from replay to free execution.

ACKNOWLEDGMENTS

This research is funded by a collaboration grant of the Austrian Science Fund (FWF) and the Research Foundation Flanders (FWO Belgium) as project I2491-N31 and G004816N, respectively.

REFERENCES

[1] Joe Armstrong, Robert Virding, Claes Wikstrom, and Mike Williams. 1996. *Concurrent Programming in Erlang* (2 ed.). Prentice Hall PTR.

[2] Earl T Barr, Mark Marron, Ed Maurer, Dan Moseley, and Gaurav Seth. 2016. Time-travel debugging for JavaScript/Node.js. In *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE 2016)*. ACM, 1003–1007. https://doi.org/10.1145/2950290.2983933

[5] Results unpublished, from private communication.

[6] *Telemetry*, Lightbend, access date: 2018-05-03, https://developer.lightbend.com/docs/cinnamon/2.5.x/instrumentations/akka/akka.html

[7] *OpenTracing.io*, access date: 2018-05-03, http://opentracing.io/

[3] Edd Barrett, Carl Friedrich Bolz-Tereick, Rebecca Killick, Sarah Mount, and Laurence Tratt. 2017. Virtual Machine Warmup Blows Hot and Cold. *Proc. ACM Program. Lang.* 1, OOPSLA, Article 52 (Oct. 2017), 27 pages. https://doi.org/10.1145/3133876

[4] Elisa Gonzalez Boix, Carlos Noguera, Tom Van Cutsem, Wolfgang De Meuter, and Theo D'Hondt. 2011. Reme-d: A reflective epidemic message-oriented debugger for ambient-oriented applications. In *Proceedings of the 2011 ACM Symposium on Applied Computing*. ACM, 1275–1281.

[5] Gilad Bracha, Peter von der Ahé, Vassili Bykov, Yaron Kashai, William Maddox, and Eliot Miranda. 2010. Modules as Objects in Newspeak. In *ECOOP 2010 – Object-Oriented Programming*, Theo D'Hondt (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 405–428.

[6] Brian Burg, Richard Bailey, Andrew J. Ko, and Michael D. Ernst. 2013. Interactive Record/Replay for Web Application Debugging. In *Proceedings of the 26th Annual ACM Symposium on User Interface Software and Technology (UIST'13)*. ACM, 473–484. https://doi.org/10.1145/2501988.2502050

[7] Sergey Bykov, Alan Geller, Gabriel Kliot, James R. Larus, Ravi Pandya, and Jorgen Thelin. 2011. Orleans: Cloud Computing for Everyone. In *Proceedings of the 2Nd ACM Symposium on Cloud Computing (SOCC '11)*. ACM, New York, NY, USA, Article 16, 14 pages. https://doi.org/10.1145/2038916.2038932

[8] Yunji Chen, Shijin Zhang, Qi Guo, Ling Li, Ruiyang Wu, and Tianshi Chen. 2015. Deterministic Replay: A Survey. *ACM Comput. Surv.* 48, 2, Article 17 (Sept. 2015), 47 pages. https://doi.org/10.1145/2790077

[9] Sylvan Clebsch, Sophia Drossopoulou, Sebastian Blessing, and Andy McNeil. 2015. Deny Capabilities for Safe, Fast Actors. In *Proceedings of the 5th International Workshop on Programming Based on Actors, Agents, and Decentralized Control (AGERE! 2015)*. ACM, New York, NY, USA, 1–12. https://doi.org/10.1145/2824815.2824816

[10] Ronald Curtis and Larry D. Wittie. 1982. BUGNET: A debugging system for parallel programming environments. In *Proceedings of the 3rd International Conference on Distributed Computing Systems (ICDCS'82)*. IEEE Computer Society, 394–400.

[11] Joeri De Koster, Tom Van Cutsem, and Wolfgang De Meuter. 2016. 43 Years of Actors: A Taxonomy of Actor Models and Their Key Properties. In *Proceedings of the 6th International Workshop on Programming Based on Actors, Agents, and Decentralized Control (AGERE 2016)*. ACM, New York, NY, USA, 31–40. https://doi.org/10.1145/3001886.3001890

[12] Tim Felgentreff, Michael Perscheid, and Robert Hirschfeld. 2017. Implementing record and refinement for debugging timing-dependent communication. *Science of Computer Programming* 134 (2017), 4–18. https://doi.org/10.1016/j.scico.2015.11.006

[13] Jim Gray. 1986. Why do computers stop and what can be done about it?. In *Symposium on reliability in distributed software and database systems*. Los Angeles, CA, USA, 3–12.

[14] Emily H Halili. 2008. *Apache JMeter: A practical beginner's guide to automated testing and performance measurement for your websites*. Packt Publishing Ltd.

[15] Carl Hewitt, Peter Bishop, and Richard Steiger. 1973. A Universal Modular ACTOR Formalism for Artificial Intelligence. In *IJCAI'73: Proceedings of the 3rd International Joint Conference on Artificial Intelligence*. Morgan Kaufmann.

[16] Jeff Huang, Peng Liu, and Charles Zhang. 2010. LEAP: Lightweight Deterministic Multi-processor Replay of Concurrent Java Programs. In *Proceedings of the Eighteenth ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE '10)*. ACM, New York, NY, USA, 207–216. https://doi.org/10.1145/1882291.1882323

[17] 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)*. ACM, 67–80. https://doi.org/10.1145/2687357.2687368

[18] Jacques Chassin de Kergommeaux, Michiel Ronsse, and Koenraad De Bosschere. 1999. MPL*: Efficient Record/Play of Nondeterministic Features of Message Passing Libraries. In *Proceedings of the 6th European PVM/MPI Users' Group Meeting on Recent Advances in Parallel Virtual Machine and Message Passing Interface*. Springer, London, UK, 141–148.

[19] Ivan Lanese, Naoki Nishida, Adrián Palacios, and Germán Vidal. 2018. CauDEr: A Causal-Consistent Reversible Debugger for Erlang. In *Functional and Logic Programming (FLOPS'18)*, Vol. 10818. Springer, 247–263. https://doi.org/10.1007/978-3-319-90686-7_16

[20] Thomas J LeBlanc and John M Mellor-Crummey. 1987. Debugging parallel programs with instant replay. *IEEE Trans. Comput.* 4 (1987), 471–482.

[21] Philipp Lengauer, Verena Bitto, and Hanspeter Mössenböck. 2015. Accurate and Efficient Object Tracing for Java Applications. In *Proceedings of the 6th ACM/SPEC International Conference on Performance Engineering (ICPE '15)*. ACM, New York, NY, USA, 51–62. https://doi.org/10.1145/2668930.2688037

[22] Hongyu Liu, Sam Silvestro, Wei Wang, Chen Tian, and Tongping Liu. 2018. iReplayer: In-situ and Identical Record-and-replay for Multithreaded Applications. In *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI 2018)*. ACM, New York, NY, USA, 344–358. https://doi.org/10.1145/3192366.3192380

[23] Stefan Marr. 2018. ReBench: Execute and Document Benchmarks Reproducibly. (August 2018). https://doi.org/10.5281/zenodo.1311762 Version 1.0.

[24] Stefan Marr, Benoit Daloze, and Hanspeter Mössenböck. 2016. Cross-Language Compiler Benchmarking—Are We Fast Yet?. In *Proceedings of the 12th ACM SIGPLAN International Symposium on Dynamic Languages (DLS'16)*, Vol. 52. ACM, 120–131. https://doi.org/10.1145/2989225.2989232

[25] Stefan Marr, Carmen Torres Lopez, Dominik Aumayr, Elisa Gonzalez Boix, and Hanspeter Mössenböck. 2017. A Concurrency-Agnostic Protocol for Multi-Paradigm Concurrent Debugging Tools. In *Proceedings of the 13th ACM SIGPLAN International Symposium on Dynamic Languages (DLS'17)*. ACM. https://doi.org/10.1145/3133841.3133842

[26] Ali José Mashtizadeh, Tal Garfinkel, David Terei, David Mazieres, and Mendel Rosenblum. 2017. Towards Practical Default-On Multi-Core Record/Replay. In *Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'17)*. ACM, 693–708. https://doi.org/10.1145/3037697.3037751

[27] Charles E. McDowell and David P. Helmbold. 1989. Debugging Concurrent Programs. *ACM Comput. Surv.* 21, 4 (Dec. 1989), 593–622. https://doi.org/10.1145/76894.76897

[28] Mark S. Miller, E. Dean Tribble, and Jonathan Shapiro. 2005. Concurrency Among Strangers: Programming in E As Plan Coordination. In *Proceedings of the 1st International Conference on Trustworthy Global Computing (TGC'05)*. Springer.

[29] Satish Narayanasamy, Gilles Pokam, and Brad Calder. 2005. BugNet: Continuously Recording Program Execution for Deterministic Replay Debugging. *SIGARCH Comput. Archit. News* 33, 2 (May 2005), 284–295. https://doi.org/10.1145/1080695.1069994

[30] Michael Perscheid, Benjamin Siegmund, Marcel Taeumel, and Robert Hirschfeld. 2016. Studying the advancement in debugging practice of professional software developers. *Software Quality Journal* 25, 1 (2016), 83–110. http://dblp.uni-trier.de/db/journals/sqj/sqj25.html#PerscheidSTH17

[31] Michiel Ronsse, Koen De Bosschere, and Jacques Chassin de Kergommeaux. 2000. Execution replay and debugging. In *Proceedings of the Fourth International Workshop on Automated Debugging (AADebug)*.

[32] M. A. Ronsse and D. A. Kranzlmuller. 1998. RoltMP-replay of Lamport timestamps for message passing systems. In *Parallel and Distributed Processing, 1998. PDP '98. Proceedings of the Sixth Euromicro Workshop on*. 87–93. https://doi.org/10.1109/EMPDP.1998.647184

[33] Andrea Rosà, Lydia Y. Chen, and Walter Binder. 2016. Actor Profiling in Virtual Execution Environments. In *Proceedings of the 2016 ACM SIGPLAN International Conference on Generative Programming: Concepts and Experiences (GPCE'16)*. ACM, 36–46. https://doi.org/10.1145/2993236.2993241

[34] Andrea Rosà, Lydia Y. Chen, and Walter Binder. 2016. Profiling Actor Utilization and Communication in Akka. In *Proceedings of the 15th International Workshop on Erlang (Erlang 2016)*. ACM, 24–32. https://doi.org/10.1145/2975969.2975972

[35] 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*. ACM, 50–57.

[36] Benjamin H. Sigelman, Luiz André Barroso, Mike Burrows, Pat Stephenson, Manoj Plakal, Donald Beaver, Saul Jaspan, and Chandan Shanbhag. 2010. *Dapper, a Large-Scale Distributed Systems Tracing Infrastructure*. Technical report. Technical report, Google, Inc. https://research.google.com/archive/papers/dapper-2010-1.pdf

[37] Terry Stanley, Tyler Close, and Mark S Miller. 2009. Causeway: A message-oriented distributed debugger. *Technical Report of HP, HPL-2009-78* (2009).

[38] Dave Thomas. 2014. *Programming Elixir: Functional , Concurrent , Pragmatic , Fun* (1st ed.). Pragmatic Bookshelf.

[39] Stefan Tilkov and Steve Vinoski. 2010. Node.js: Using JavaScript to Build High-Performance Network Programs. *IEEE Internet Computing* 14, 6 (Nov 2010), 80–83. https://doi.org/10.1109/MIC.2010.145

[40] Carmen Torres Lopez, Stefan Marr, Hanspeter Mössenböck, and Elisa Gonzalez Boix. 2016. Towards Advanced Debugging Support for Actor Languages: Studying Concurrency Bugs in Actor-based Programs. (30 Oct. 2016), 5 pages.

[41] Takanori Ueda, Takuya Nakaike, and Moriyoshi Ohara. 2016. Workload Characterization for Microservices. In *2016 IEEE International Symposium on Workload Characterization (IISWC'16)*. IEEE, 85–94. https://doi.org/10.1109/IISWC.2016.7581269

[42] Tom Van Cutsem. 2012. AmbientTalk: Modern Actors for Modern Networks. In *Proceedings of the 14th Workshop on Formal Techniques for Java-like Programs (FTfJP '12)*. ACM, 2–2. https://doi.org/10.1145/2318202.2318204

[43] Thomas Würthinger, Christian Wimmer, Christian Humer, Andreas Wöß, Lukas Stadler, Chris Seaton, Gilles Duboscq, Doug Simon, and Matthias Grimmer. 2017. Practical Partial Evaluation for High-performance Dynamic Language Runtimes. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'17)*. ACM, 662–676. https://doi.org/10.1145/3062341.3062381

[44] Thomas Würthinger, Andreas Wöß, Lukas Stadler, Gilles Duboscq, Doug Simon, and Christian Wimmer. 2012. Self-Optimizing AST Interpreters. In *Proceedings of the 8th Dynamic Languages Symposium (DLS'12)*. 73–82. https://doi.org/10.1145/2384577.2384587