Asynchronous Snapshots of Actor Systems for Latency-Sensitive Applications

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

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

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

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

Abstract

The actor model is popular for many types of server applications. Efficient snapshotting of applications is crucial in the deployment of pre-initialized applications or moving running applications to different machines, e.g for debugging purposes. A key issue is that snapshotting blocks all other operations. In modern latency-sensitive applications, stopping the application to persist its state needs to be avoided, because users may not tolerate the increased request latency.

In order to minimize the impact of snapshotting on request latency, our approach persists the application's state asynchronously by capturing partial heaps, completing snapshots step by step. Additionally, our solution is transparent and supports arbitrary object graphs.

We prototyped our snapshotting approach on top of the Truffle/Graal platform and evaluated it with the Savina benchmarks and the Acme Air microservice application. When performing a snapshot every thousand Acme Air requests, the number of slow requests (0.007% of all requests) with latency above 100ms increases by 5.43%. Our Savina microbenchmark results detail how different utilization patterns impact snapshotting cost.

To the best of our knowledge, this is the first system that enables asynchronous snapshotting of actor applications, i.e. without stop-the-world synchronization, and thereby minimizes the impact on latency. We thus believe it enables new deployment and debugging options for actor systems.

CCS Concepts • **Software and its engineering** → **Software testing and debugging**; • **Theory of computation** → *Concurrency*.

Keywords Actors, Snapshots, Micro services, Latency

ACM Reference Format:
Dominik Aumayr, Stefan Marr, Elisa Gonzalez Boix, and Hanspeter Mössenböck. 2019. Asynchronous Snapshots of Actor Systems for Latency-Sensitive Applications. In *Proceedings of the 16th ACM SIGPLAN International Conference on Managed Programming Languages and Runtimes (MPLR '19), October 21–22, 2019, Athens, Greece.* ACM, New York, NY, USA, 15 pages. https://doi.org/10.1145/3357390.3361019

1 Introduction

Snapshotting persists a program's state so that the program can be restored and continued later. Programming environments such as Lisp and Smalltalk use snapshotting to create images of the system's state. These images allow developers to deploy a pre-configured system into production or continue development at a previous state. Snapshots can also facilitate time-traveling and record & replay debugging [4, 5], by restoring a program execution to an earlier point in time to investigate events that lead to the occurrence of a bug. Finally, snapshots enable quick crash recovery and moving a program's execution to a different machine.

This paper focuses on snapshotting support for actor-based applications. Popular implementations of the actor model such as Akka,[1] Pony [12], Erlang [2], Elixir [25], and Orleans [9] are used to build complex responsive applications. We present a novel technique to snapshot such responsive actor-based applications avoiding stop-the-world pauses.

Creating a snapshot requires the program state to be persisted. In Lisp, Smalltalk, and other systems [4], this is done with heap dumps integrated with the garbage collector (GC). This, however, requires virtual machine (VM) support and usually *stops the world* to create a consistent snapshot, making the program unresponsive. This is problematic for applications that aim to respond consistently with low latency.

Snapshotting is also common in high-performance computing [8, 11, 14, 15, 20] to address distributed failures. These approaches provide inspiration but in this work we do not

MPLR '19, October 21–22, 2019, Athens, Greece
© 2019 Copyright held by the owner/author(s). Publication rights licensed to ACM.
This is the author's version of the work. It is posted here for your personal use. Not for redistribution. The definitive Version of Record was published in *Proceedings of the 16th ACM SIGPLAN International Conference on Managed Programming Languages and Runtimes (MPLR '19), October 21–22, 2019, Athens, Greece*, https://doi.org/10.1145/3357390.3361019.

[1] *Akka Website*, https://akka.io/

address such failures and focus on non-distributed actor applications, simplifying the problem significantly.

In this paper, we present an efficient approach for transparent asynchronous snapshots of non-distributed actor-based systems. It is implemented on top of an unmodified Java VM and does not rely on GC integration. By snapshotting the state of each actor individually, we avoid stop-the-world synchronization that blocks the entire application. We applied our approach to communicating event loop (CEL) actors [22] in SOMns. SOMns is an implementation of Newspeak [7] built on the Truffle framework and the Graal just-in-time compiler [28]. We evaluated the performance of our approach with the SOMns implementation. On the Savina benchmark suite [18], we measure the run-time overhead and memory impact of frequent snapshot creation. On the modern web-application Acme Air [26], we measure the effect of snapshot creation on request latency, to ensure that user experience remains acceptable, i.e. additional latency is below 500ms [1].

The main contribution of this paper is a novel snapshotting approach for non-distributed actor programs that minimizes latency by avoiding stop-the-world synchronization and persisting program state asynchronously. Furthermore, our approach does not require changes to the VM nor GC integration. Our evaluation shows that for the Acme Air experiment snapshotting increases the number of slow requests with latency over 100ms by 5.43% while the maximum latency is unchanged.

2 Background and Requirements for Asynchronous Actor Snapshots

This section provides the background on actor concurrency to show the challenges of designing an asynchronous snapshot mechanism for actor systems. We also briefly discuss SOMns, a programming language with communicating event loop actors, for which we implemented our approach.

2.1 Communicating Event Loops (CELs)

Originally, the actor model was proposed by Hewitt et al. [17]. Since then it has been used as an inspiration for many derived variants [13]. In this work, we focus on programs written in the communicating event loop (CEL) variant, which was first described for the programming language E [22]. This variant has all the characteristics typically associated with actor models: isolation of state and message passing. Additionally, it provides non-blocking promises as a high-level abstraction for returning results of asynchronous computations. CELs were also adopted by languages such as AmbientTalk [27] and Newspeak [7], and correspond to the event loops in widely used systems such as JavaScript and Node.js.

The structure and main components of CELs are shown in fig. 1. Each actor contains a set of objects that are isolated from those of other actors, a mailbox for receiving messages, and an event loop. The event loop of an actor perpetually takes messages from the mailbox in the order of their arrival and processes them one-by-one. Each message specifies a receiver, i.e. an object of the actor which understands the message, and executes the corresponding method defined in the object. The processing of a message is an atomic operation with regard to other messages on the same actor and defines a so-called *turn*.

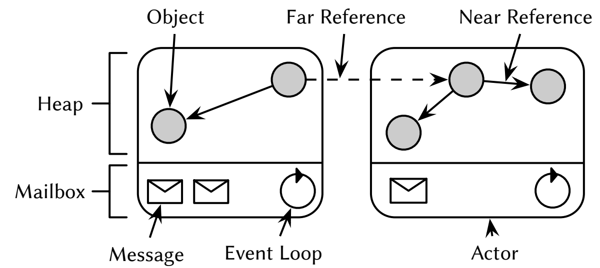


Figure 1. Main components of communicating event loops.

To guarantee state isolation, each actor can only access its own objects directly. Objects in other actors are accessed via *far references*, which restrict interactions to asynchronous messages. Objects given as parameters in messages are by default passed by far reference to ensure isolation of state.

Asynchronous message sends immediately return a *promise* (also known as *future*). The promise is a placeholder for the result of the message send. The receiver of the message promises to provide the result at a later time. When the result becomes available the promise is *resolved*. Promises are objects themselves and consequently can receive messages. A promise handles messages differently from other objects: instead of executing a method, received messages are accumulated in the promise while it is not resolved. When a promise with stored messages is resolved, all the messages are forwarded to the final value of the promise. It is also possible to resolve a promise with another promise (i.e. *promise chaining*). In this case, the resolved promise is registered as a dependent in the other promise. The promise is fully resolved when the other promise is resolved, and only then forwards stored messages. Similar to passing objects by far reference when they cross actor boundaries, promise chaining is used when promises are passed to other actors. For the receiver of the message, a new promise is created that is resolved with the original.

***SOM*ns** We implement our snapshotting approach for actor programs in SOMns, an implementation of Newspeak [7] built on the Truffle framework and the Graal just-in-time compiler [28]. SOMns implements a CEL actor model (cf. section 2.1) and is designed for shared-memory multicore systems. Shared memory is used for optimizations where applicable. For instance, far references are placeholder objects that contain direct pointers to objects in other actors. SOMns does not expose or modify the underlying GC and

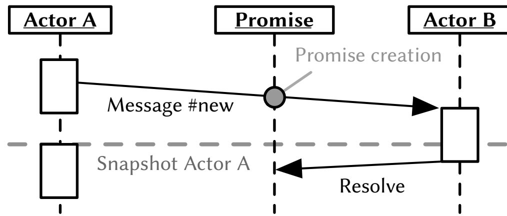


Figure 2. Race condition between serializing `Promise` as part of `Actor A`, and its resolution by `Actor B`. In an optimized implementation, the promise value may be lost.

VM platform. As such, it can be used on top of a stock JVM with the JVM compiler interface (JVMCI).

2.2 Asynchronous Snapshotting for Actor Systems

As described in the introduction, our goal is to create transparent asynchronous snapshots for actor programs with isolated state. In the following, we describe the issues that have to be considered in the design of a snapshotting solution for CELs: (1) serialization of far references, (2) lost promise resolution, and (3) serialization of messages. While the serialization of far references and lost promise resolution are specific to the CEL model, the serialization of messages is required for any actor model variant.

Issue 1: Serialization of Far References As mentioned before, objects are passed by far reference to maintain state isolation, making all intra-actor communication asynchronous. However, serializing objects by naively traversing the object graph of an actor may reach state that belongs to another actor via far references. Since the other actor may concurrently change its state, snapshotting needs to account for it. Specifically, there are two issues to tackle: (1) Data races may occur when the owner of an object reached via far reference is concurrently processing a message and changes its state, when a snapshot is recorded. This can lead to inconsistent data in the snapshot. (2) Would the traversal not stop in the referenced actors, but continue with the far references present in those as well, the entire program state may be potentially serialized at once. This would increase latency, and thus, defeat the purpose of asynchronous snapshots.

Issue 2: Lost Promise Resolutions Recall from section 2.1 that promises are used to asynchronously return results of message sends. Different implementation strategies can be used to resolve promises. The classic approach is to resolve them with an explicit asynchronous message that is handled as any other message, avoiding race conditions. Alternatively, to minimize the number of messages sent, promises can also be resolved directly using locking at the implementation level, e.g. in SOMns. This conceptually violates the state isolation between actors, as one actor can change the state of a promise owned by another actor. But, in practice, this data race cannot be observed by a program, because the state (and value) of a promise can only be accessed asynchronously by sending a message. However, a snapshotting approach that is applicable to optimized actor systems, such as SOMns, needs to take special care with promise resolutions as this data race could lead to inconsistent or incomplete snapshots.

Figure 2 shows a scenario where the resolution of a promise is lost, because the snapshot takes place after the actor's state was serialized. This particular scheduling serializes an unresolved `Promise` object which is owned by `Actor A`. The promise is then resolved with an object belonging to `Actor B`, but the snapshots do not reflect that promise resolution, because there are no promise resolution messages that could be captured. Hence, we refer to such a scenario as a *lost promise resolution*. Its effect is that after snapshot restoration, the promise is unresolved, and since the resolving message was not part of the snapshot, it remains unresolved indefinitely. Any messages sent to the `Promise` object after the snapshot is restored would accumulate and never be delivered.

Issue 3: Serialization of Messages To successfully restore a snapshot, it has to contain the messages that were about to be executed by an actor, i.e., the mailbox contents. In the context of asynchronous non-blocking snapshots this is a challenge as actors finish processing their current messages before snapshotting. For some time both snapshotted and un-snapshotted actors may coexist. Snapshotted actors may receive additional messages from un-snapshotted actors, and vice-versa. A snapshotting approach therefore has to be able to recognize messages that were sent from un-snapshotted actors to snapshotted actors and add them to the snapshot.

3 Snapshotting Actor Systems

We now present our snapshotting approach, which is designed for actor systems with isolated state and atomic message execution. The main idea of our approach is to perform snapshotting asynchronously to avoid stop-the-world pauses and minimize application latency. Furthermore, the snapshotting mechanism is designed to be transparent to programs, i.e., can be used without annotating code to specify state that needs to be captured and to support arbitrary object graphs. Since the actor model ensures that objects owned by an actor remain unchanged when the actor is not processing a message, our approach captures the state of actors before they start processing messages after a snapshot was triggered. In addition to the actor's state, we persist all unprocessed messages that originate from before the snapshot. When restoring a snapshot, the recorded messages are used to restore an actor's mailbox in the correct order. In this section we detail how to capture snapshots and section 4 provides details on restoring snapshots.

3.1 Capturing Snapshots

To minimize latency when snapshotting an application, we designed an asynchronous snapshotting mechanism that is based on capturing partial state for each actor (cf. section 3.3). Capturing partial state is possible due to the isolation of state and atomic processing of messages in actor programs. To ensure completeness of the snapshots, we capture the following components of an actor's state: (1) state directly owned by an actor, (2) messages and near-referenced objects reachable from them (cf. section 3.3), and (3) objects that are far-referenced from other actors (cf. section 3.4).

Snapshotting can be triggered asynchronously either explicitly by user code or automatically. Automatic snapshots can be created for example at regular time intervals or when used alongside record & replay (cf. section 6) after the trace size reaches a defined limit. For simplicity our approach creates only one snapshot at a time, i.e. snapshot creation has to be completed before another snapshot can be initiated.

Before an actor starts processing the first message after a snapshot was triggered, we persist its directly owned state. As stated in *issue 3* (cf. section 2.2), we also need to identify messages that were sent before the snapshot request and to serialize them whenever they are about to be processed by an actor. We solve this issue by dividing the program execution into phases. Each time a snapshot is triggered, a new phase begins and a global phase counter is incremented. When a message is sent, we attach the phase number of the sending actor to it. Hence, finding out if a message needs to be captured is a simple numeric comparison. All messages that are about to be processed and whose send phase does not match the global phase have to be serialized.

Note that actors only change phases at the beginning of a turn, i.e., when they start processing a message. Because the processing of messages is atomic, actors that are busy executing a message cannot immediately observe a change of the global phase counter and remain in the phase in which they started processing their current message until it is completed. This means that such actors may send messages with phase numbers smaller than the current global phase.

Figure 3 illustrates a program execution with snapshots. It shows that different actors may be in different phases, *Actor 1* is the last actor to finish processing of its current message and to advance to *Phase 1*. Any messages sent by *Actor 1* after the snapshot but before finishing the message are part of *Phase 0* and have to be captured by the receiving actor. A snapshot is said to be complete when there are no more actors in the previous phase, i.e., no more messages have to be captured. As long as actors are in the previous phase, additional messages may need to be captured by actors that already advanced. We assume that actors finish processing messages and do not remain in one phase indefinitely. Otherwise, only state which is far referenced by other actors will be part of the snapshot.

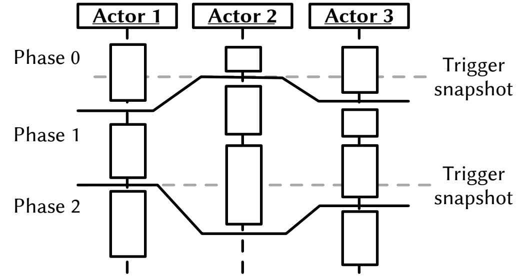


Figure 3. Asynchronous snapshots divide a program execution into different phases. Since actors only switch phases when they are not executing a message, phases can overlap.

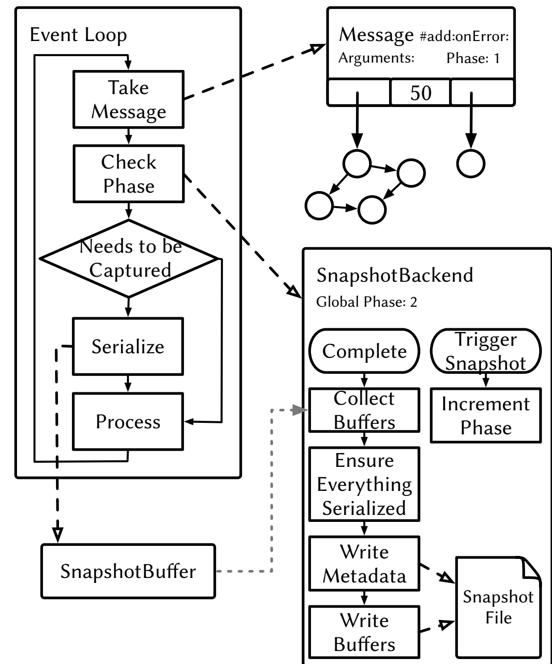


Figure 4. Architecture of our solution showing how the components connect to produce snapshots.

As messages are assigned phase numbers when they are originally sent, we have to update the phase numbers of messages when they are forwarded to the result of a promise resolution. Hence, these messages will not be captured for the current snapshot, but are reproduced in the restored execution when the promises containing them are resolved.

3.2 System Architecture

We now describe the architecture of our snapshotting approach which is shown in fig. 4. When an event loop is being snapshotted, it will still take a message out of its mailbox but then the event loop needs to perform additional steps before processing a message. In the first step, the event loop checks the phase of the message against the global phase in the

snapshot backend and determines if the message needs to be serialized. If that is the case, the message is then serialized into a snapshot buffer. Snapshot buffers store the data as byte arrays in memory.

As an optimization, in SOMns the actors are scheduled flexibly on a thread pool. This can be used to reduce the number of buffers needed. Instead of assigning a snapshot buffer to each actor, buffers can be assigned to the processing threads. Each buffer is used by only one actor at a time, but may contain data of different actors. As actors can be executed on different processing threads, their snapshot data can be spread across multiple buffers.

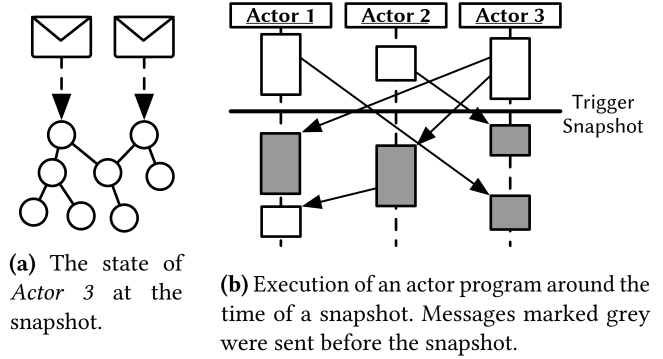
The snapshot backend is responsible for triggering snapshots, maintaining metadata, collecting the snapshot buffers from the threads, and writing the snapshot files. Snapshots are written to disk by a separate thread once they are complete. We detect if snapshots are complete by queuing a special task in the thread pool used for actors. After the task is executed, there are no more actors with messages that need to be captured in the thread pool's queue. Currently executing actors may still contain messages from the previous phase. Hence, we have to wait for each thread to finish processing it's current task, i.e. actor, before the snapshot can be persisted to disk. At this point, there may still be some left-over deferred serializations (c.f. section 3.4) that need to be handled before the snapshot can be persisted. We have bookkeeping in place that tells us which actors have unfinished deferred serializations, and proceed by scheduling all those actors for execution so that they may serialize missing objects. When all of them are done, we check if this caused some new deferred serializations, and if yes repeat the scheduling and checking. Only after there are no more messages to capture and all deferred serializations have been handled, we can persist the snapshot to disk.

If the root cause of an error is before the latest snapshot, earlier snapshots have to be used. Otherwise, only the effects can be reproduced and observed.

Compared to a thread-based concurrency model, we rely on CEL actor turns to terminate. This means, we assume that turns do not contain infinite loops. Infinite loops are generally considered a bug in CEL actor programs, since they impact latency and performance more generally. In a thread-based system, such an infinite loop could continuously add elements to a data structure, which would make it difficult to persist the data structure.

3.3 Capturing Partial Heaps

The actor state, i.e., the object graph that can be reached from different messages in the mailbox of an actor is typically partial. For some actors, the whole object heap can be reached via a single message, while for others state may consist of multiple disjoint or partially-connected object graphs. The latter is depicted in fig. 5a. To minimize the impact on message latency, we capture only partial state per message



(a) The state of *Actor 3* at the snapshot.

(b) Execution of an actor program around the time of a snapshot. Messages marked grey were sent before the snapshot.

Figure 5. Example where partial heaps can be snapshot.

if possible. The idea is to first persist the state a message depends on and to process that message afterwards. This is done incrementally for subsequent messages, i.e. before processing another message, we capture the state that was not persisted before. Splitting the serialization enables actors to make progress and be more responsive, as they gradually complete the snapshot. State that is only reachable from far references is also captured as we explain in section 3.4.

In the example in fig. 5b, two messages (marked grey) were sent to *Actor 3* before a snapshot. As these messages are processed after the snapshot, they need to be captured. The state of *Actor 3* that is reachable from these messages can be seen in fig. 5a. Different parts of the actor's state can be reached from each of them, but they also have a common subgraph. This means that the state of the actor does not have to be captured all at once, instead we can capture the state incrementally message by message, each message completing the snapshot with the reachable state that was not captured before. This incremental capturing of the state can increase the responsiveness of actors after a snapshot was triggered, but only if there is an overlap in the object graph reachable from the messages, as in fig. 5a.

3.4 Far References – Deferring Serialization

As described in the problem statement (section 2.2), far references connect the object graphs of different actors. To ensure correctness, i.e. prevent data races, far-referenced objects have to be serialized by their owner. We call this *deferred serialization*. As our solution is asynchronous, we cannot wait for another actor to serialize a far-referenced object and return its location. In addition to the far-referenced object, the owner is handed a buffer plus offset where the reference information has to be filled in after serialization. If the object was already serialized, it is not added to the queue of deferred serializations. Instead the reference information is written to the buffer immediately. Before an actor starts processing a message it handles all the serializations that were deferred to it. As actors can be idle for longer periods of time, we have to consider the possibility that an actor has

deferred serializations when a snapshot is written to disk. To avoid having missing pieces in our snapshot we keep track of actors with deferred serializations. We force those actors to perform the deferred serializations until the snapshot is complete, i.e. there are no more deferred serializations. Hence, even actors that were not active between triggering a snapshot and writing it to disk are captured and can be used after restoring the snapshot. We achieve forced serialization by scheduling all the actors with deferred serializations on the thread pool we use for execution, even if they do not have any messages to process. This avoids data races that could happen if the forced serialization was, for example, done in the snapshot writer thread.

3.5 Promises

Promises require special treatment when they are serialized as they can contain an arbitrary number of messages and references to other promises (cf. section 2.1). The referenced promises typically belong to a different actor. We therefore defer the serialization of those promises to the respective owners as explained above (cf. section 3.4).

Promises are local to their owner. When sent to another actor, a new promise is created for the receiver and chained with the original one (cf. section 2.1). Thus, all messages that a promise receives were sent by its owner, i.e. there is no race regarding the messages contained in a promise.

Note that when serializing a promise, it can be either a promise that was already resolved, or an unresolved one. The messages referenced by an unresolved promise can be serialized like any other object since they are owned by the actor that owns the corresponding promise. Messages related to resolved promises were already sent to another actor and are instead captured by the receiver if necessary.

As discussed in section 2.2, we need to handle promise resolutions that race with the snapshotting, which otherwise could get lost. This could occur when a promise is serialized as unresolved, but is then resolved by an actor in a previous phase, which would not be reflected in the snapshot. We detect such racy resolutions by comparing the resolving actor's phase number with the global one. If they do not match, the resolution is captured separately in the snapshot.

3.6 Snapshot Format

An important part of serialization is to use a format that can be efficiently recorded and read, while preserving data in the correct order. We now briefly detail the snapshot format used and relevant correctness concerns.

Listing 1 shows a representation of our binary snapshot format. It consists of two kinds of data. First, it has a section of metadata with a registry of the snapshot messages, offsets of the snapshot buffers within the snapshot file, and data to reconstruct promise resolutions that are otherwise lost. The rest of the format consists of interconnected heaps that represent the serialized object graphs.

We now detail the different components of the metadata. To ensure correct execution when loading a snapshot, we need to restore the snapshotted messages to the mailboxes of their respective actor in the original order. As such, for each snapshot message, the `MessageRegistry` contains the location where the message object is stored in the snapshot. To ensure that the messages are restored correctly, the registry also contains the id of the actor, and an ordering number.

Promise resolutions are also part of the metadata and can be found as `Resolutions`. This part links unresolved promises to a result and identifies the actor that performed the resolution in the original execution as well as whether it was successful or erroneous. Section 3.5 details why we need to capture promise resolutions separately.

The `ClassEnclosures` are used to support nested classes. This is necessary to support the Newspeak semantics [7] but would apply to other languages such as Java, too. It contains a mapping of class ids to the object that enclosed the class in the original execution.

The final piece of metadata is the `HeapMap`; it is a mapping of buffer ids to offsets in the snapshot file. `HeapMap` is used to decode our reference representations. Each reference consists of a 16-bit buffer id, and a 48-bit offset. Referenced objects can be found in the snapshot file by getting the start location of the containing buffer and then adding the offset.

```
Snapshot = MetaData Heaps.
MetaData = MessageRegistry Resolutions
ClassEnclosures HeapMap.
MessageRegistry = msgCnt {actorId msgNo
msgLocation}.
ClassEnlosures = cnt {classId outerObject}.
Resolutions = cnt {resolver result actor
state}.
HeapMap = nHeaps {heapId offset}.
Heaps = {Object}.
Object = classId ObjectData.
ObjectData = Message Promise Array ...
```

Listing 1. EBNF grammar for our binary snapshot format.

4 Restoring Snapshots

Restoring snapshots boils down to deserializing all captured objects, recreating actors, and initializing the mailboxes with the snapshot messages in the right order. The first step in the deserialization is to parse the `MetaData` section of the snapshot, which contains the storage locations of the messages and the start locations of the different heaps in the snapshot file. Afterwards, the messages can be deserialized using the `MessageLocations`. The messages are then put into the mailboxes of the respective actors in the same order they are in the snapshot, which ensures they are processed in the original execution order. For simplicity, actors are not

allowed to process messages until after all messages are deserialized and in a mailbox. Otherwise the actors would start sending each other messages and altering their state while we are still restoring the program state.

Note that a snapshot does not contain a list of actors. Instead, actors are created implicitly when their ids are encountered during snapshot parsing.

4.1 Deserializing Messages and Object Graphs

We now provide further details on the deserialization. Messages, like any other object in the snapshot, are encoded as seen in listing 1, and their entry starts with a `classId`. The deserialization of an object starts by looking up the class with that id. Previously unknown classes are loaded lazily, i.e., the first time they are encountered during deserialization. The looked-up class allows us to deserialize its instances transparently, similar to serialization during snapshot-creation, and transitively deserializes any other referenced objects.

As different messages may reference the same object, and the object graphs may have circles, we keep track of which objects were already deserialized. In the case of cyclic object graphs, we omit the cyclic reference and fix it later when both objects are available. Our implementation uses a map of an object's snapshot address to its instance. We can therefore check if an object was already deserialized and can maintain object identity by using the map entry instead of deserializing it again. For handling cycles, we install a placeholder in the map before deserialization of an object. When during deserialization a reference is encountered for which a placeholder exists, instead of deserializing it and causing an infinite loop, we leave the reference uninitialized and add some fixup information to the placeholder. This can be, for example, a tuple of the object and the field that needs to be fixed. Finally, when a placeholder is replaced by the actual object, all the fixups that accumulated are performed, i.e., references to the object are initialized.

5 Evaluation

This section evaluates the performance of our snapshotting approach. Since SOMns is a research language, we first compare its performance to other language implementations to provide sufficient context. We measure the run-time overhead and memory impact of snapshotting using the Savina [18] benchmark suite. We assess the impact on request latency with the Acme Air microservice application, Following Arapakis et al. [1], snapshots should not increase request latency by more than 500ms, otherwise the delay will be noticed by users. In short, we will assess SOMns' baseline performance, the snapshot overhead on microbenchmarks, and their impact on a microservice application.

5.1 Methodology

SOMns relies on dynamic compilation to reach its peak performance. Thus, to account for the VM's warmup behavior [6], we executed each of the Savina and Are We Fast Yet [21] benchmarks for 1000 iterations within a single process. Manual inspection of the complete run-time plots indicates that the performance of the benchmarks stabilizes after 100 iterations (cf. Appendix A for plots showing these first 100 iterations). We thus discarded the first 100 iterations to discount warmup and be more representative for longer running applications.

For Acme Air, we use JMeter [16] 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 8 threads for making requests and executes a mix of about 2 million randomly generated requests based on the predefined workload pattern. After inspecting the latency plots, we discarded the first 100,000 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, Java 8.171 and Graal version 1.05. Acme Air experiments were 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 Mojave (10.14.3), Java 8.151 and Graal version 0.41.

5.2 Baseline Performance of SOMns

To show that the baseline performance of SOMns is competitive with other language implementations, we evaluate the sequential performance of SOMns, Node.js, and Java with the Are We Fast Yet benchmarks. The results shown in fig. 6 indicate that SOMns's performance is at the level of Node.js, a similar dynamic language, but is not as fast as Java.

To show that the CEL implementation of SOMns has a performance that is similar to other actor implementations, we used the Savina benchmark suite. This benchmark suite was originally designed for impure actor languages with shared memory, such as Akka, Jetlang, and Scalaz. Hence, some of the benchmarks rely on shared memory, which is not supported in SOMns. As a consequence, only 18 out of 28 benchmarks could be ported to SOMns. Our results from running this subset of the Savina benchmark suite are shown in fig. 7 and suggest that the actor model of SOMns reaches performance comparable to other JVM-based implementations, despite its lower sequential performance.

Considering SOMns overall performance, we argue that it is a suitable platform for the discussed research, and allows us to draw conclusions about performance that are applicable to other state-of-the-art language implementations.

5.3 Savina: Worst Case Cost of Creating Snapshots

To assess our snapshotting approach for a range of actor programs, we compare the warmed-up execution time of

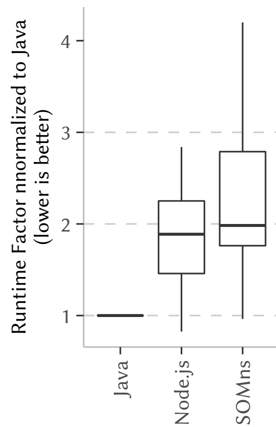


Figure 6. Boxplot comparing the performance of SOMns other languages, showing SOMns performs similar to Node.js.

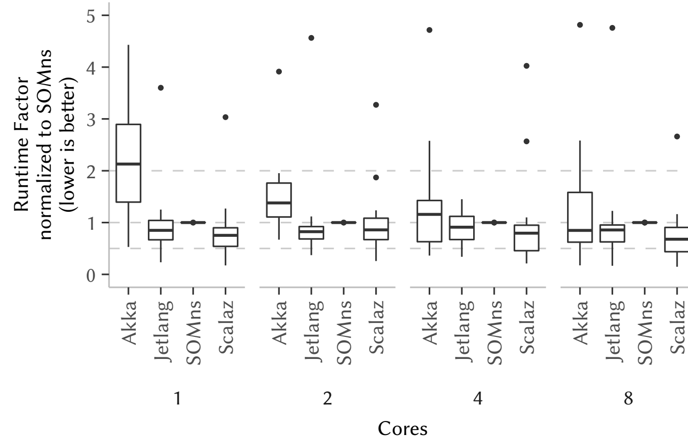


Figure 7. Boxplot comparing the performance of Savina benchmarks in different actor languages for different numbers of cores. It shows that the performance of SOMns is comparable to other actor implementations.

benchmark iterations with and without snapshotting using the Savina benchmarks.

The overhead of running a program with snapshots enabled depends on the number of snapshots, the number of snapshot messages, and the object graph. Production systems run for a long time and trigger snapshots infrequently. As a result, the overall overhead of creating snapshots is distributed over a larger time frame and is averaged out. Our Savina benchmarks, on the other hand, have a short run time. The overhead of infrequent snapshots might get lost in the noise, while selecting high snapshot frequencies automatically increases the measured overhead. To compare benchmark iterations it is important that the number of snapshots per iteration is constant for each benchmark. Hence, we decided to do trigger a snapshot every second benchmark iteration, which can be considered a worst case scenario. In the Savina benchmarks, workload is often generated by sending a large number of messages (up to hundred thousands) in a loop. If a snapshot is triggered after the generation of the workload was started, an equally large number of messages has to be captured due to their phase number. This means that depending on the benchmark, a large share of the overall messages has to be snapshot, increasing overhead. This is especially noticeable for non-computational-intensive benchmarks with short run time, such as *Counting*, *ForkJoinActorCreation* and *Chameneos*, which have large numbers of messages. Hence, the Savina benchmark suite presents more of a worst-case scenario for our snapshotting implementation. However, it does not represent real-world latency-sensitive applications for which we use the Acme Air application in section 5.4.

Performance differences between benchmarks can be explained by their different object graphs, snapshot sizes, and the number of messages and actors. Figure 8 shows that for

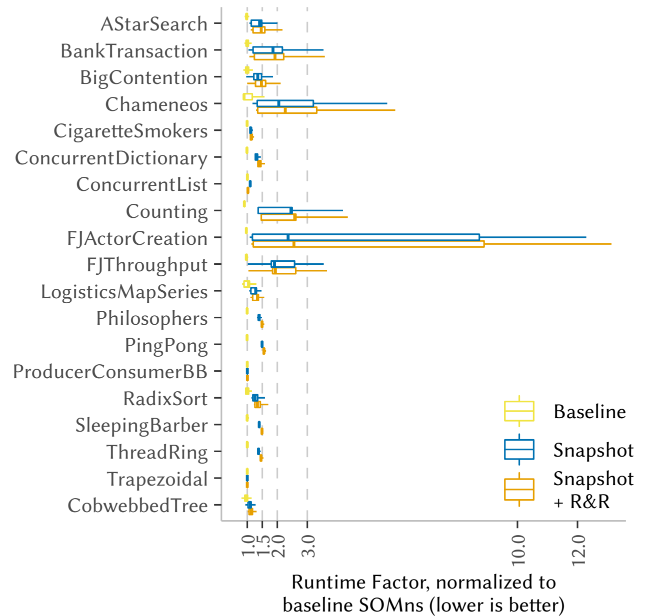


Figure 8. Boxplot comparing the run-time performance of Savina benchmarks when snapshotting, as well as when snapshotting is combined with record & replay. Snapshotting performance varies depending on the benchmark, but is for most of the microbenchmarks in the 1x to 1.5x range.

snapshotting Savina benchmarks, the run time normalized to the mean of baseline iterations is generally in the 1x to 1.5x area, with a few outliers up to 5x. For the more computationally intensive benchmarks of the Savina suite, such as *TrapezoidalApproximation*, overhead can be as low as 0.04%.

Memory Impact of Snapshotting Savina Benchmarks We also evaluated memory metrics using the Savina benchmark suite as additional data structures and objects have an impact on memory usage and GC behavior of an application. We captured the metrics between benchmark iterations to analyze their behavior over all iterations. Due to our snapshotting, we expected to find that the application has more and larger temporary objects, larger heap, and more time spent on GC. Figure 9 shows that the number of collected bytes is higher, and fulfills our expectations on temporary objects. We can observe that over the course of 1000 iterations the number of collected bytes increases roughly linear for both configurations. However, for snapshotting the inclination is higher, causing the number of collected bytes to diverge. Like runtime overhead, the rate of divergence depends on the individual benchmark and ranges from close to zero (e.g. *SleepingBarber*) to more than double (e.g. *FJActorCreation*). For more memory metrics we refer to appendix A.

Performance of Snapshots Combined with Record & Replay Finally, we used the Savina benchmarks to assess how record & replay interacts with snapshotting. In previous work, we evaluate the performance of record & replay [3]. Since snapshotting allows us to limit the size of recorded traces, the two techniques should work well together.

As expected, adding record & replay comes at a higher overall performance cost than snapshotting alone. Figure 8 shows that overhead increases only minimally. The average overhead (geometric mean) increases from 49.35% with only snapshotting to 54.88% with additional record & replay.

5.4 Acme Air: Impact on User Experience

User experience is a critical factor for the success of responsive server applications. Unresponsive websites or interfaces can discourage their use. The latency between request and response is a key component for the user experience. We use Acme Air, a web application simulating the booking system of a fictive airline, to show the impact of our snapshots on the latency of web requests. Additionally, we aim to show that the maximum impact is still acceptable. According to Arapakis et al. [1], occasional delays of up to 500ms are barely noticed by users. Thus, the impact of our snapshots should be below this threshold.

For this experiment, we decided to trigger a snapshot once every 1000 requests. Every incoming request is represented by a message in the actor system and may be serialized to be part of the snapshot. Because Acme Air's implementation uses a cache for flights and airports to reduce database access, the snapshots reach a size of up to 5MB for the last snapshots.

Figure 10 shows the results of this experiment. In particular, it shows the latency profile of the different requests. We use a logarithmic scale for the y-axis, so individual requests with high latency, for example due to GC, are still visible. Latency spikes, i.e. individual requests with high latency did not increase dramatically compared to baseline, and are distributed across the different requests. As the different request types are not equally common, there are some differences in the distribution of spikes.

With snapshotting enabled, the number of requests with latency above 100ms increases by 5.43%. These requests make up only 0.007% of all 20 million requests.

When looking at the slowest requests over 200ms, we observe that the number of such requests increases from 250 to 385 requests with snapshotting enabled. We conclude that the latency distribution is similar to the unmodified benchmark execution, but there is a shift towards higher latency, which we attribute to the frequent snapshot creation.

Figure 11 shows the average latency of the different request types. The effect of snapshotting is most noticeable in *QueryFlight* requests, where the average latency with snapshots is up to 2.28% higher than the average latency of the baseline. Hence, snapshotting has a small impact on the average latency of requests, while the frequenzy and latency of the slowest requests, which we attribute to GC, remains similar to the original execution. We conclude that the additional latency for individual requests is below 500ms, therefore our performance goal is reached and the user experience for Acme Air is not significantly impacted.

6 Discussion

In our snapshotting approach, we capture a transitive closure of all objects that are reachable from active actors and messages processed after the snapshot. However, external resources also need to be considered for snapshots. SOMns supports extension modules that can keep state, such as references to objects and actors, which might be unreachable and therefore not part of any snapshot. Since there is no generally correct strategy, we leave it up to the implementers of such modules to ensure unreachable state is persisted. For instance, for our HTTP server as well as time-out actors, we register their roots explicitly with the snapshotting mechanism, so that they are recorded correctly in the snapshot.

Integration with Record & Replay Our snapshotting approach is integrated with the SOMns record & replay implementation [3]. Both components can be used individually or together. When combined, the snapshot backend notifies the record & replay that a new snapshot is created. Any recorded trace data afterwards is part of a new trace file relative to the new snapshot. Old trace files can be deleted to free-up disk space if needed. This allows record & replay to support long-running applications by forgoing the ability to reproduce the execution before a snapshot. While our snapshotting approach captures state, messages, and their order, it is not a record & replay solution on its own. It ensures that the captured messages are restored in order, but does not enforce any specific interleaving with un-captured messages. As all actors remain suspended until the entire state is restored

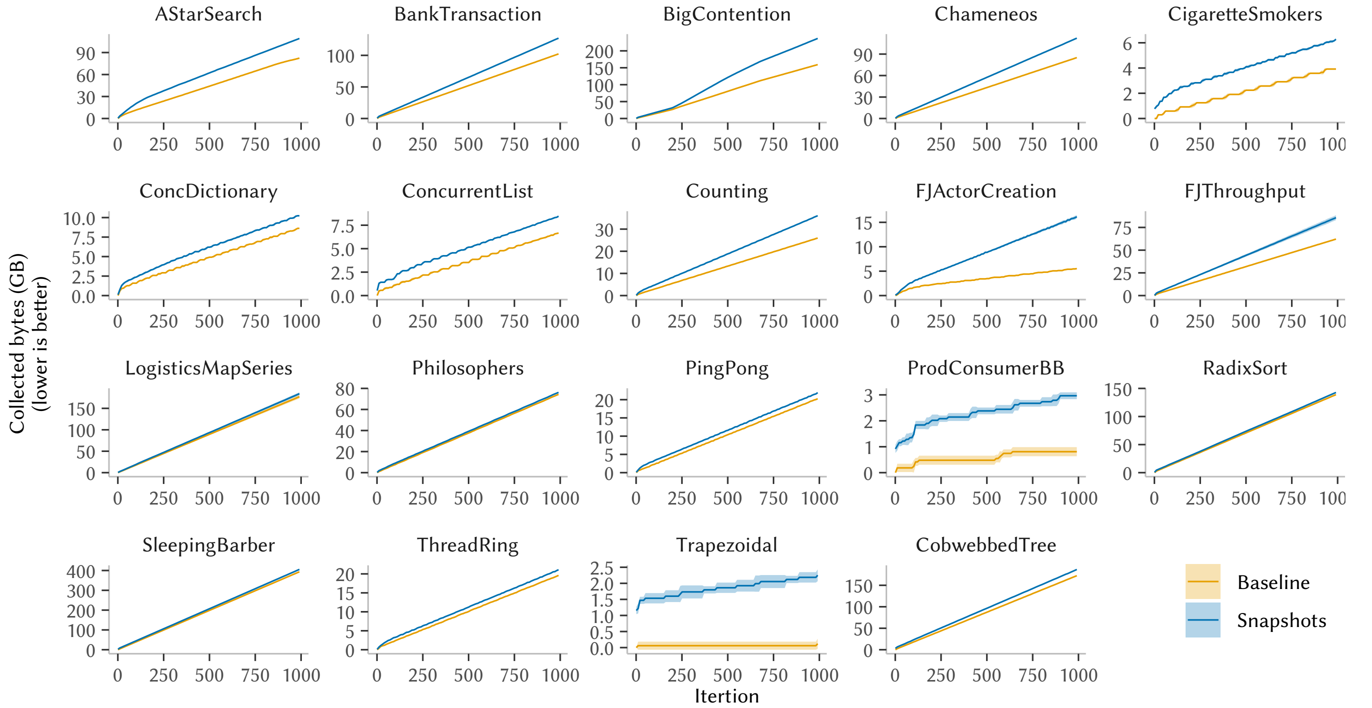


Figure 9. Plot of total collected bytes (GB) over 1000 iterations for the Savina benchmarks, with and without snapshotting. Most benchmarks indicate that the collected bytes increase by a small constant factor for each iteration, which is within expectations.

from a snapshot, captured messages are not interleaved with un-captured messages, which limits the number of observable behaviors. In combination with record & replay, one can however ensure that the order of all messages is identical to the previously recorded one.

Applicability to other systems Our approach for asynchronous snapshotting of actor programs should be generally applicable to actor systems with isolated state and atomic message processing in FIFO order. We integrated our prototype directly with the language implementation on top of Truffle/Graal. However, this is not a requirement, and instrumentation techniques, such as Java bytecode transformation, can be used as an alternative to achieve similar results.

7 Related Work

In this section we discuss related work to provide context for our contributions. We compare our work to two types of approaches that aim to solve similar issues: snapshotting solutions for distributed systems, and back in time debugging.

7.1 Checkpointing in Distributed Systems

Checkpointing in distributed systems has been explored extensively. According to Elnozahy et al. [14], the two main approaches are coordinated and uncoordinated checkpoints.

In uncoordinated checkpointing, the distributed entities perform checkpoints independent from each other. When the state of one of those entities is restored to a checkpoint, e.g. due to a failure, other entities may be forced to rollback to one of their snapshots to keep the systems state consistent. As the snapshots are uncoordinated, rollback may be propagated through the system. In the worst case, rollback is performed until the initial state is reached. To avoid this effect in distributed systems, coordinated checkpointing was developed by Chandy and Lamport [11]. The original coordinated approach is based on persisting a processes state and then propagating marker messages when a checkpoint is created. A process then records incoming messages for each channel, until it receives a marker back, and adds them to the state to counteract inconsistencies. In the following, we further discuss variants of coordinated checkpointing.

Blocking vs. NonBlocking Buntinas et al. [8] implemented and compared blocking and non-blocking variants of coordinated checkpointing based on the algorithm by Chandy and Lamport [11]. In the blocking variant, after checkpointing and propagating markers to all neighbors, a process waits for all neighbors to send back a marker before execution continues. Hence, blocking coordinated checkpoints prevent the system from making progress until all

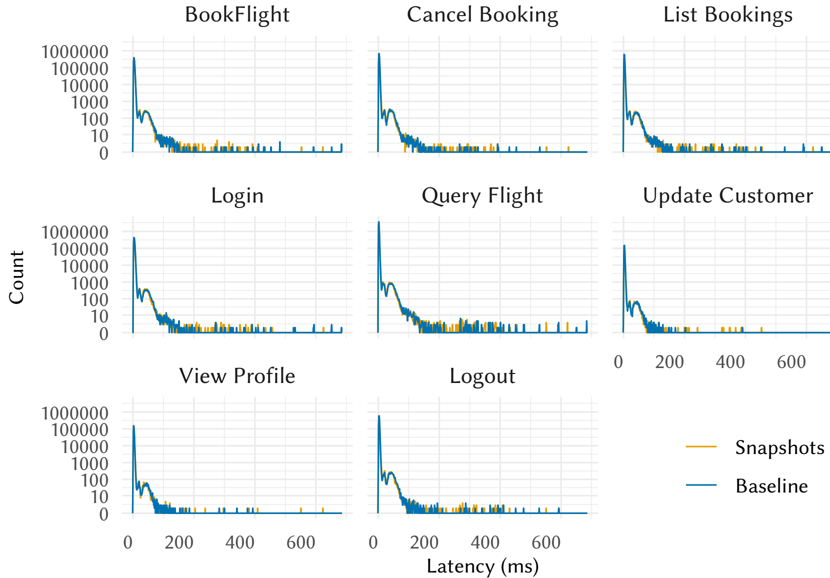


Figure 10. Distribution of the total number of requests over different latencies for the Acme Air benchmark with snapshots every 1000 requests. The values with snapshotting enabled are minimally shifted. The number of requests over 100ms increases by 5.43%. Though, these requests make up only 0.007% of all 20 million requests and thus seem acceptable.

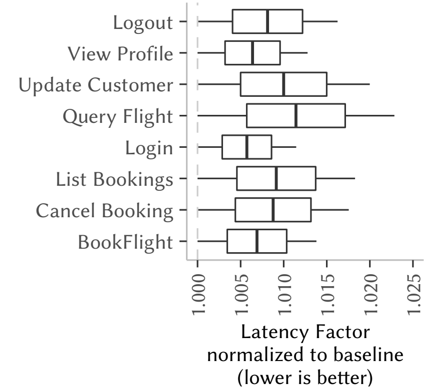


Figure 11. Latency of Acme Air requests with snapshotting, normalized to baseline. Overall, latency increases by 1.66% (geometric mean).

processes have performed a checkpoint, and consequently has a high latency. The non-blocking variant, on the other hand, assumes that the communication channels are guaranteed to be FIFO, and allows execution to continue after propagating markers, while recording messages on incoming channels until markers are received back. Their implementation optimized capturing a process' state by forking, and having the clone persist its state. This benefits non-blocking checkpoints the most, as execution can continue almost instantaneous.

Our snapshotting solution is non-blocking as we allow for messages to be processed before the entire program state is persisted. Furthermore, we do not require synchronization between actors before they capture their state. As we target non-distributed applications, coordination is not an issue. We instead use a global phase number (cf. section 3.1), that actors check before they process a message. Hence, capturing state is done lazily when an actor processes a message.

Checkpoint Optimizations Disk-less checkpointing [24] avoids costly disk access by keeping checkpoints in memory. or sending them over the network. The checkpoint can be encoded and split into small chunks, that can then be sent to a checkpoint processor, which is responsible for recovering the state of a failed process. Only the state of the failed process needs to be decoded, other processes go back to the checkpoint they already have in their memory. In contrast, our solution keeps snapshots in memory until they are complete, and then writes them to disk.

Another optimization for distributed systems is incremental checkpointing [23], where full checkpoints are created infrequently, with incremental checkpoints of changed memory pages captured in-between. The incremental checkpoints are smaller and can be created faster than full snapshots, reducing network usage when transmitting checkpoints. Restoring incremental checkpoints requires the last full checkpoint and all incremental checkpoints up to the selected point to be processed. For our approach to perform incremental checkpoints we would need write barriers that update an object's entry in the snapshot. However, because our solution does partial heap snapshots, we still keep latency similar to incremental approaches.

Coordinated Checkpointing with Relaxed Synchronization Losada et al. [20] explored coordinated checkpoint/restart that enables rollback of individual processes without having to rollback others. To achieve such a rollback, they replay messages between restarted and non-restarted processes. To replay messages, a message logging protocol is used to capture non-deterministic events and messages after a process checkpoints. When a process is rolled back, the events in the log are replayed to bring it to a state consistent

with the other processes. The message logging introduces additional overhead but avoids rollback of all processes to the last consistent state, which compensates the overhead with faster recovery. Our approach only captures messages that are part of the programs state, but this can be augmented with lightweight record & replay to deterministically replay events after the snapshot. However, as our solution is non-distributed, record & replay does not need to record message contents as reproducing message order is sufficient.

Asynchronous Local Checkpointing for Actors To the best of our knowledge, the only checkpointing approach for actors is for SALSA [19]. Compared to our approach, it is not transparent and it is integrated with a programming model called *transactor*. The programming model allows for some transactional behavior including rolling back actors, which is implemented using local checkpoints. Since we focus on snapshotting, we do not offer this kind of functionality.

Asynchronous Barrier Snapshotting Asynchronous Barrier Snapshotting (ABS) [10] is an approach for Apache Flink, that propagates snapshot barriers through a program. After a process' input receives a barrier, the input is blocked until a barrier was received on all inputs, which also causes the state of a process to be captured. To be able to handle communication cycles, ABS relies on static analysis to identify back-edges. Similar to our approach, after a process receives a barrier it does not process inputs until its state is captured. A big difference to our solution is in ABS's blocking behavior. Our solution does not require actors to wait for others before they can snapshot and continue processing messages.

7.2 Back-in-Time Debugging

Back-in-time debugging relies on snapshots of the program state at certain intervals, and offers time travel by replaying execution from the checkpoint before the target time. We now compare our approach to back-in-time debugging solutions for actor-based systems.

Jardis Jardis [5] 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. When our snapshotting is combined with record & replay, it provides a similar functionality for a CEL system without relying on GC piggybacking or a modified VM. In contrast to Jardis, our system is designed for multiple event loops and needs to ensure correct snapshots for each event loop.

Event Sourcing Event sourcing is a technique for actors, where all state changes are logged incrementally. It has been used to create snapshots of actor programs [15]. Their retrospective snapshots are based on processing an event sourcing log and aggregating the individual state changes up to a certain point in time into one independent snapshot. This gives them the ability to extract arbitrary snapshots by doing post-processing. While our solution is designed to take snapshots infrequently during program execution, it can be combined with SOMns ordering-based replay [3] to achieve similar results. By restoring a snapshot and replaying the program execution based on the compact trace, we can reproduce the state at any point after the snapshot without having to log all state changes in the original execution.

8 Conclusion and Future Work

The actor model has become popular for implementation of responsive server applications. Unfortunately, many snapshotting approaches are blocking, and cause applications to become unresponsive for the duration of snapshot creation.

In this paper, we presented a novel approach for creating asynchronous snapshots of actor programs transparently and without VM modifications or GC integration. Our approach uses the isolation of state of the actor model to reduce snapshot latency by capturing partial heaps and allowing actors to make progress before all their state is persisted.

We evaluated the impact of our snapshotting approach on application latency with the Acme Air benchmark. Our results show that with frequent snapshotting enabled, the latency of most requests remains below 100ms. Only 0.007% of 20 million requests had a latency over 100ms. While the number of such requests did increase by 5.43%, their total is still small. We conclude that our approach does not negatively impact the user experience of such a web service.

Future Work As future work, we plan to apply our approach to JavaScript with a wider range of benchmarks. This would also enable a direct comparison with Jardis.

Time Travel Debugging. While our snapshotting can already be combined with record & replay, further research is needed to enable time travel debugging. For example, restoring a snapshot currently requires a fresh start of the program. This is a problem as time travel debugging requires frequent snapshot restoration, which means, we need to be able to replace the whole application state within a VM.

Application Redeployment. To be practical for moving applications between machines, our approach has to be enhanced so that required resources are also moved. This could be achieved by bundling the source code of all loaded classes with the snapshot and tracking all used external resources so they can be made available in the new environment.

A Appendix

In this section, we present supplemental warmup and memory metrics for the evaluation of our snapshotting approach

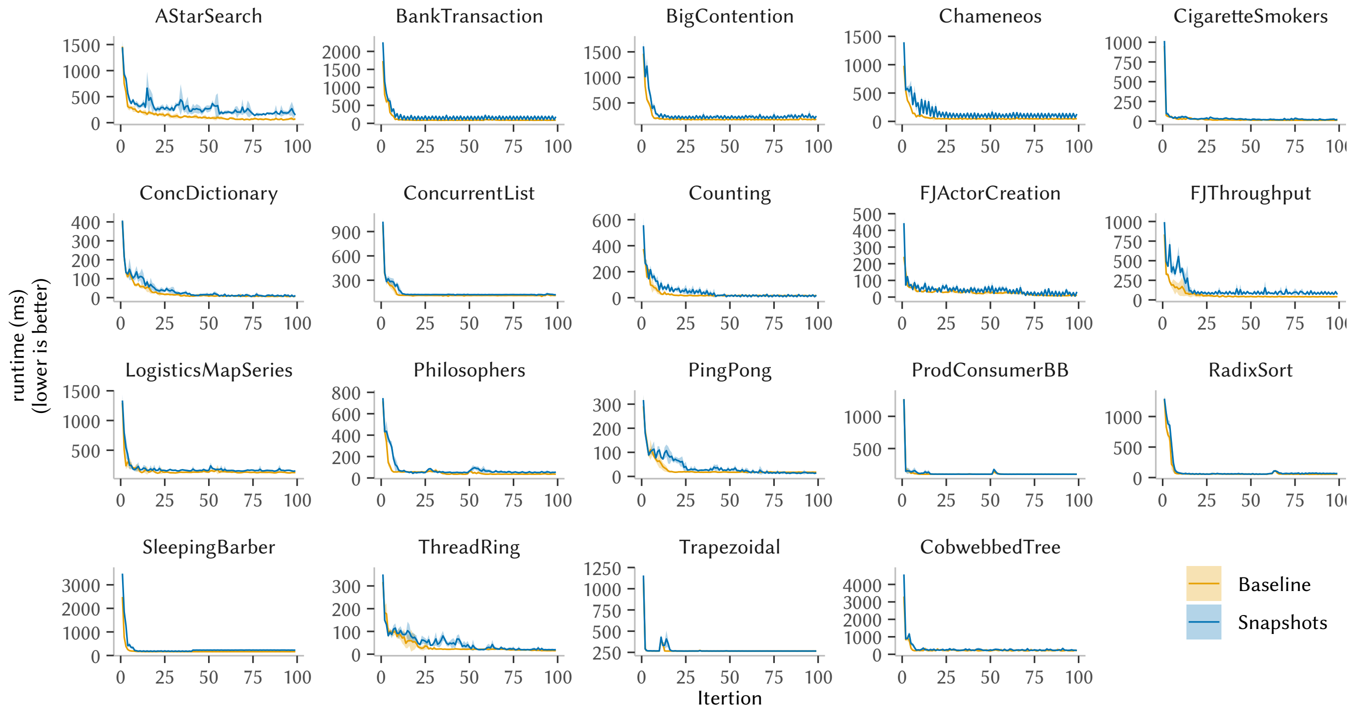


Figure 12. Warmup behaviour of Savina benchmarks with and without snapshotting.

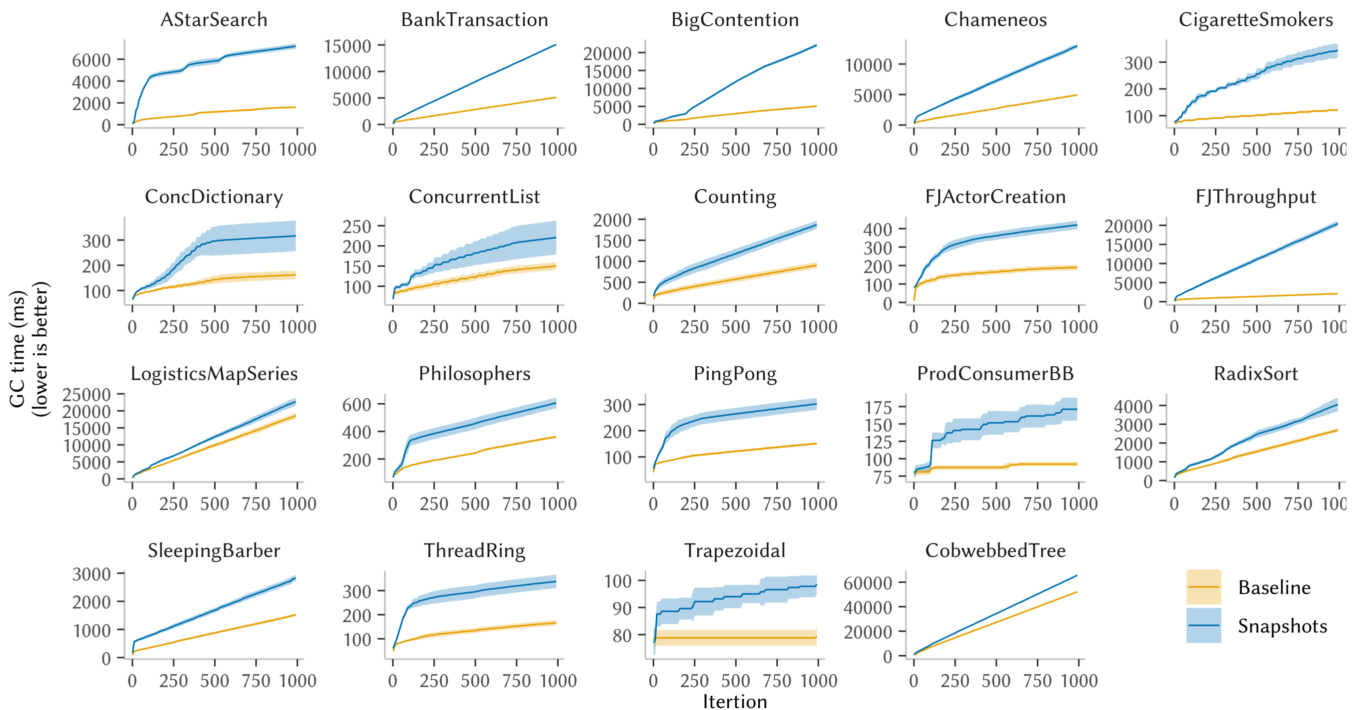


Figure 13. Development of total time spent on GC over 1000 iterations of Savina benchmarks, with and without snapshotting.

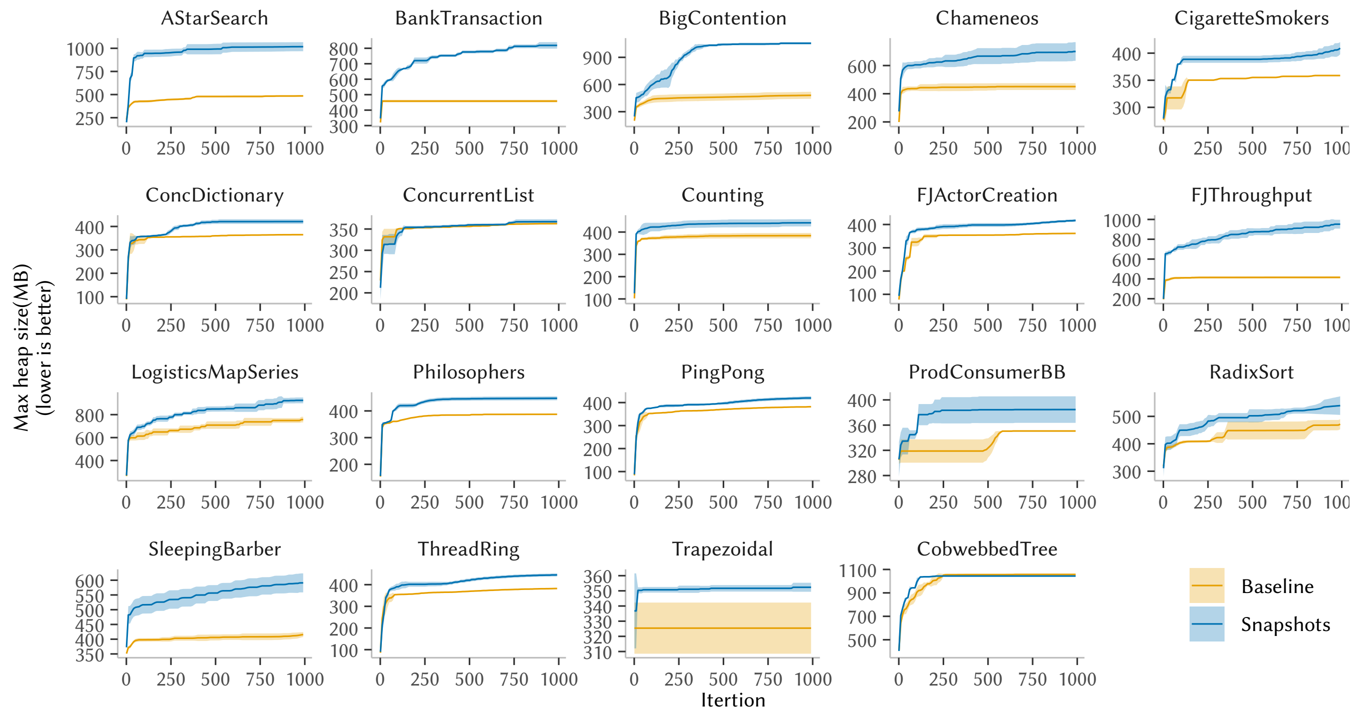


Figure 14. Development of the max heap size for Savina benchmarks with and without snapshotting.

in the Savina benchmark suite. Figure 12 shows the development of the absolute run time of the benchmarks over the first 100 iterations, which we discarded to account for warmup. The regular spikes visible in benchmarks such as *Chamenos* are caused by the creation of snapshots every second iteration.

Figure 13 shows how the GC time of the different benchmarks develops, like the number of collected bytes and the run time overhead. We observe that GC time varies greatly between the different benchmarks. For some benchmarks the GC time itself contributes significantly to the measured overhead. *ForkJoinThroughput*, for example, with an average execution time of 40.36ms per iteration has a GC time overhead of 19.22ms per iteration. As the average overhead of the benchmark is 2.25x, GC itself is responsible for 39.82% of the overhead.

Figure 14 shows how the maximum heap size stabilizes over the course of 1000 iterations. As before, there is a high variation between the different benchmarks. While benchmarks such as *ConcurrentList* do not need additional memory when snapshotting, benchmarks like *AStarSearch* need more than double the heap size and use all of the available 1GB memory.

Acknowledgments

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

References

[1] Ioannis Arapakis, Xiao Bai, and B Barla Cambazoglu. 2014. Impact of response latency on user behavior in web search. In *Proceedings of the 37th international ACM SIGIR conference on Research & development in information retrieval*. ACM, 103–112.

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

[3] Dominik Aumayr, Stefan Marr, Clément Béra, Elisa Gonzalez Boix, and Hanspeter Mössenböck. 2018. Efficient and Deterministic Record & Replay for Actor Languages. In *Proceedings of the 15th International Conference on Managed Languages & Runtimes (ManLang '18)*. ACM, Article 15, 14 pages. https://doi.org/10.1145/3237009.3237015

[4] Earl T. Barr and Mark Marron. 2014. Tardis: Affordable Time-travel Debugging in Managed Runtimes. In *Proceedings of the 2014 ACM International Conference on Object Oriented Programming Systems Languages & Applications (OOPSLA '14)*. ACM, New York, NY, USA, 67–82. https://doi.org/10.1145/2660193.2660209

[5] 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

[6] 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

[7] 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 (Lecture Notes in Computer*

Science), Vol. 6183. Springer, 405–428. https://doi.org/10.1007/978-3-642-14107-2_20

[8] Darius Buntinas, Camille Coti, Thomas Herault, Pierre Lemarinier, Laurence Pilard, Ala Rezmerita, Eric Rodriguez, and Franck Cappello. 2008. Blocking vs. non-blocking coordinated checkpointing for large-scale fault tolerant MPI Protocols. *Future Generation Computer Systems* 24, 1 (2008), 73 – 84. https://doi.org/10.1016/j.future.2007.02.002

[9] 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, Article 16, 14 pages. https://doi.org/10.1145/2038916.2038932

[10] Paris Carbone, Gyula Fóra, Stephan Ewen, Seif Haridi, and Kostas Tzoumas. 2015. Lightweight asynchronous snapshots for distributed dataflows. *arXiv preprint arXiv:1506.08603* (2015).

[11] K. Mani Chandy and Leslie Lamport. 1985. Distributed Snapshots: Determining Global States of Distributed Systems. *ACM Trans. Comput. Syst.* 3, 1 (Feb. 1985), 63–75. https://doi.org/10.1145/214451.214456

[12] 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

[13] 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

[14] E. N. (Mootaz) Elnozahy, Lorenzo Alvisi, Yi-Min Wang, and David B. Johnson. 2002. A Survey of Rollback-recovery Protocols in Message-passing Systems. *ACM Comput. Surv.* 34, 3 (Sept. 2002), 375–408. https://doi.org/10.1145/568522.568525

[15] Benjamin Erb, Dominik Meißner, Gerhard Habiger, Jakob Pietron, and Frank Kargl. 2017. Consistent retrospective snapshots in distributed event-sourced systems. In *2017 International Conference on Networked Systems (NetSys)*. IEEE, 1–8.

[16] Emily Halili. 2008. *Apache JMeter*. Packt Publishing.

[17] 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, 235–245.

[18] 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

[19] Phillip Kuang, John Field, and Carlos A. Varela. 2014. Fault Tolerant Distributed Computing Using Asynchronous Local Checkpointing. In *Proceedings of the 4th International Workshop on Programming based on Actors Agents & Decentralized Control (AGERE! '14)*. 81–93. https://doi.org/10.1145/2687357.2687364

[20] Nuria Losada, George Bosilca, Aurélien Bouteiller, Patricia González, and María J. Martín. 2019. Local rollback for resilient MPI applications with application-level checkpointing and message logging. *Future Generation Computer Systems* 91 (2019), 450 – 464. https://doi.org/10.1016/j.future.2018.09.041

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

[22] 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, 195–229.

[23] N. Naksinehaboon, Y. Liu, C. Leangsuksun, R. Nassar, M. Paun, and S. L. Scott. 2008. Reliability-Aware Approach: An Incremental Checkpoint/Restart Model in HPC Environments. In *2008 Eighth IEEE International Symposium on Cluster Computing and the Grid (CCGRID)*. 783–788. https://doi.org/10.1109/CCGRID.2008.109

[24] J. S. Plank, , and M. A. Puening. 1998. Diskless checkpointing. *IEEE Transactions on Parallel and Distributed Systems* 9, 10 (Oct 1998), 972–986. https://doi.org/10.1109/71.730527

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

[26] 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

[27] 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

[28] 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