Savina - An Actor Benchmark Suite
Enabling Empirical Evaluation of Actor Libraries

Shams Imam Vivek Sarkar
shams@rice.edu vsarkar@rice.edu
Department of Computer Science, Rice University, Houston, USA.

Abstract
This short paper introduces the Savina benchmark suite for actor-oriented programs. Our goal is to provide a standard benchmark suite that enables researchers and application developers to compare different actor implementations and identify those that deliver the best performance for a given use-case. The benchmarks in Savina are diverse, realistic, and represent compute (rather than I/O) intensive applications. They range from popular micro-benchmarks to classical concurrency problems to applications that demonstrate various styles of parallelism. Implementations of the benchmarks on various actor libraries will be made publicly available through an open source release. This will allow other developers and researchers to compare the performance of their actor libraries on these common set of benchmarks.

Categories and Subject Descriptors D.1 [Programming Techniques]: Concurrent Programming; F.2 [Analysis of Algorithms and Problem Complexity]: General

General Terms Concurrent Programming, Measurement, Performance

Keywords Actor Model, Benchmarks, Scala, performance, compare different actor implementations and identify those that deliver the best performance for a given use-case. The benchmarks in Savina are diverse, realistic, and represent compute (rather than I/O) intensive applications. They range from popular micro-benchmarks to classical concurrency problems to applications that demonstrate various styles of parallelism. Implementations of the benchmarks on various actor libraries will be made publicly available through an open source release. This will allow other developers and researchers to compare the performance of their actor libraries on these common set of benchmarks.

1. Introduction
Concurrent programs have become the norm with the proliferation of multicore processors. The Actor Model (AM) of concurrency [1] has recently gained popularity, in part due to the success achieved by its flagship language - Erlang. The AM is based on asynchronous message passing and offers a promising approach for developing reliable concurrent systems. With the success of Erlang in production settings, the AM has catapulted into a approach. While micro-benchmarks are useful, micro-benchmarks rarely reflect the behaviour of larger real-world applications. A standard benchmark suite that goes beyond micro-benchmarks and allows end users to compare different implementations and use the one that delivers the best performance for a given use-case is highly desired. Unfortunately, such a suite does not exist as yet for actor programming models.

This paper presents Savina, a benchmark suite for actor-oriented programs. In this work, we are interested in developing a standardized benchmark suite that represents various use-cases in actor-oriented programs and allows users to do an apples-to-apples comparison between different actor libraries. Such a suite provides implementers of high-performance actor libraries an understanding of what the various use-cases are. It simplifies the identification of the issues that need to be corrected from the benchmark results and allows them to optimize for it. The benchmarks in Savina are diverse, realistic, and represent compute intensive applications. They range from popular micro-benchmarks to classical concurrency problems to applications that demonstrate various styles of parallelism. Implementations of the benchmarks on various actor libraries will be made publicly available through an open source release. This will allow other developers and researchers to compare the performance of their actor libraries on these common set of benchmarks.

The paper is organized as follows: in Section 2 we give a brief description of the benchmarks in the Savina suite. Section 3 presents our initial experimental results for some of the benchmarks. Section 4 discusses related work and we summarize our conclusions and future work in Section 5.

2. Benchmarks
A benchmark suite for the evaluation of actor runtimes should be representative of multiple use-cases and portable to many systems. The use of actors is very diverse and a good benchmark suite should cover various important domains. The goal of this work is to define a benchmark suite, Savina, that can be used to compare the performance of actor-oriented libraries and languages. Savina benchmarks are designed to be easily ported across different actor libraries (Section 3). The Savina benchmark suite focuses on computationally intensive applications, and includes both numeric and non-numeric problems. Savina aims to identify a representative set of actor applications which display commonly used parallel patterns. It covers applications that include common concurrency patterns.
problems, graph and tree navigation, linear algebra, and stencil computations. The applications are compute intensive, and perform no I/O operations in their kernels.

Savina is designed to be extended with new benchmarks to allow the suite to evolve and address currently uncovered domains. In addition to comparing the performance of various runtimes, the benchmarks allow the comparison of code and other additional features supported by an implementation. The primary performance metric that is output by each benchmark code is elapsed time (in milliseconds) for running the kernel body. The source code of the suite will be available for the purpose of: a) verifying what is actually being tested, b) porting the benchmarks to other actor languages and runtimes, c) allowing comparison of solutions for syntax and elegance, and d) enabling analysis of benchmarks to further study performance, and the impact of different features in different actor libraries. In addition, the results from running the suite provides end users with additional information that allows them to choose actor libraries based on the benchmarks which closely fits their own applications.

A brief description of the Savina applications is shown in Table 1, which includes the name of the benchmark, the abbreviation used to refer to the benchmark, the parallel / concurrency pattern represented by the benchmark, and the source of the benchmark. The list includes many well-known micro-benchmarks that are already de facto standards in actor-oriented models, such as Ping Pong, Chameneos, and Thread Ring, as well as many others benchmarks. The benchmarks are divided into three categories: a) micro-benchmarks, b) concurrency benchmarks, and c) parallelism benchmarks.

Micro-benchmarks Micro-benchmarks are simple benchmarks that involve simple logic dedicated to test specific features of the actor runtimes. As seen in Table 1, there are 7 well-known programs used to analyze actor languages or libraries in Savina. These are designed to measure overheads in message delivery, message throughput, concurrent mailbox implementation, actor creation and destruction.

Concurrency benchmarks The AM being a model of concurrent computation is a natural fit for exploiting concurrency in computations. The second set of benchmarks in Savina have 8 programs and include Bounded-Buffer problem, Readers and Writers problem, and Dining-Philosophers problem among others. This set is
a first step away from micro-benchmarks and towards more realistic applications. It focuses on classical concurrency problems which involves correctly coordinating non-deterministic interactions among multiple actors.

**Parallelism benchmarks** Taking full advantage offered by a multicore machine requires the writing of parallel code. The final set of benchmarks include 13 programs and concentrates on parallelism. Parallelism in the applications is obtained by task decomposition to effectively utilize multicore. The decompositions need to be converted into an actor-style computation. The benchmarks include a wide variety of computations that display pipeline parallelism, phased computations, divide-and-conquer style parallelism, master-worker parallelism, and graph and tree navigation. The programs in this set are larger than the programs from the previous two sets and represents more realistic parallel computations.

We are unaware of any other comprehensive benchmark suite for actor frameworks like Savina, and have designed it to be extensible framework so that more benchmarks can be easily added in the future. The goal is to cover a wide variety of patterns in the benchmarks which will not only allow comparison of performance, but also programmability of the solutions based on features available in the actor frameworks being evaluated. We will encourage researchers to contribute optimized versions of Savina benchmarks for existing as well as new actor frameworks into a community repository. We envision that the optimized versions of each Savina benchmark program will evolve over time with increased community contribution.

3. Experimental Results

The actor libraries used for comparison in this paper all run on the JVM. The libraries are: Akka (AK) [38], Functional Java (FJ) [9], GPars (GP) [28], Habanero-Java Actors (HA) [13, 15], Jetlang (JL) [23], Jumi (JU) [7], Lift (LI) [37], Scala actors (SC) [10], and Scalaz (SZ) [12]. All actor implementations of each benchmark use the same algorithm and mainly involved renaming the parent class of the actors to switch from one implementation to the other. All implementations use the pattern matching construct to represent the message processing body (MPB) and hence share the same overheads for the MPB. Similarly, all actor solutions use the same data structures for the user-written code of the benchmarks. We did this to ensure a fair comparison of the internals of the different frameworks.

The benchmarks were run on a 12-core (two hex-cores) 2.8 GHz Intel Westmere SMP node with 48 GB of RAM per node (4 GB per core), running Red Hat Linux (RHEL 6.2). Each core has a 32 kB L1 cache and a 256 kB L2 cache. The software stack includes a Java Hotspot JDK 1.8.0, Akka 2.3.2, FunctionalJava 4.1, GPars 1.2.1, Habanero-Scala 1.0, Jetlang 0.2.12, Jumi 0.1.196, Lift 2.6-M4, Scala 2.11.0, and Scalaz 7.1.0-M6. We provide a data set configuration for each benchmark in our scripts which can be used to reproduce the results for the benchmarks on different machines. For benchmarking, it is typically desirable to exclude code executed during JVM startup and shutdown from one’s measurements. Each benchmark was configured to run using thirteen worker threads and used the same JVM configuration flags (-Xmx16384m -XX:+UseParallelGC -XX:+UseParallelOldGC -XX:-UseGCPauseLimit) and was run for twenty iterations in six separate JVM invocations. The arithmetic mean of the best fifty execution times (from the hundred and twenty iterations) are reported to minimize effects of JIT and GC overheads from the reported results. In the bar charts, the error bars represent one standard deviation of the fifty execution times. Execution time is measured using JDK’s `System.nanoTime()`. We have implemented 19 of the 28 benchmarks from Table 1 and present results of three benchmarks below.

### 3.1 Mailbox Contention (Chameneos)

![Figure 1](image-url)  
**Figure 1**: The Chameneos benchmark was run with 500 chameneos (actors) constantly arriving at a mall (another actor). There are multiples of millions meetings between chameneos orchestrated at the mall.

The Chameneos micro-benchmark, shown in Figure 1, measures the effects of contention on shared resources while processing messages. The original SC implementation was obtained from the public Scala SVN repository [11]. The benchmark involves all chameneos constantly sending messages to a mall actor that coordinates which two chameneos get to meet. Adding messages into the mall actor’s mailbox serves as a contention point and stress tests the concurrent mailbox implementation. The SZ version performs best with JL following closely. The next set of GP, HA, AK, and JU actors are slightly slower, but competitive. The SC version pays the penalty of generating exceptions to maintain control flow in its react construct. In general, all the implementations scale linearly with an increase in the input size.

### 3.2 Logistic Map Benchmark

![Figure 2](image-url)  
**Figure 2**: Results of the LogisticMap benchmark using 150 term actors and 150 ratio actors. Each term actor is responsible for computing 150000 terms. Average execution time (x-axis) reported in seconds.

We created the Logistic Map benchmark to measure the performance of actor implementations for the synchronous request-response pattern. It computes the Logistic Map [35] using a recurrence relation $x_{n+1} = rx_n(1 - x_n)$. In the benchmark there are three

---

1. one worker thread gets blocked waiting after initialization
classes of actors: a manager actor, a set of term actors, and a set of ratio actors. The ratio actors encapsulate the ratio \( r \) and know how to compute the next term given the current term \( x_n \). The term actors require a synchronous reply from the ratio actor before they update their value of \( x \) and, only then, process the next message from the master to compute the next term in the series. We use non-blocking solutions for all the actors frameworks, thread blocking solutions take much longer time to execute and do not provide a fair comparison as some solutions might be non-blocking. Our solution for the AK version uses a custom extension that allows individual unstashing of messages, by default the Akka library only allows unstashAll which introduces a lot more overhead. Figure 2 displays the results of this benchmark for the various actor implementations. GP, JL, SZ, and AK perform the best with HA, FJ, JU, and LI following in the next set. SC performs noticeably poorly.

3.3 Split-Join Benchmark (Filter Bank)

We use the Filter Bank streaming benchmark ported from the StreamIt [30] to quantify the performance of the join pattern. Filter Bank is used to perform multi-rate signal processing and consists of multiple pipeline branches. On each branch the pipeline involves multiple stages including multiple delay stages, multiple FIR filter stages, and sampling. Since Filter Bank represents a streaming pipeline, it can be implemented using actors. The Branches stage involves a split-join to combine the results of individual Bank stages. Supporting such a join requires maintaining a dictionary to track each sequence arriving from the different banks. The performance of the benchmark is affected by the rate at which the messages are delivered across actors and the scheduler that determines which actors are scheduled on the worker threads. Figure 3 compares the performance of the Filter Bank benchmark against the different actor library implementations. SZ, JL, LI, and AK perform best followed by HA and GP. JU performs noticeably poorly.

4. Related Work

Many actor benchmarks and benchmark suites have been designed and are currently being used for many different purposes, but none match our goals for a diverse set of compute intensive applications. Actor-oriented programs. The benchmarks in PARSEC [10] can be used to compare various implementations of actors. Finally, there have also been attempts to compare programming languages by defining a set of benchmarks. The Computer Language Benchmarks Game [8] captures a broad set of languages, it compares over 20 programming languages on a set of 13 micro-benchmarks.

Figure 3: Results of the Filter Bank benchmark results configured to use 8-way join branches. The input used 300,000 data items and 131,072 columns. Average execution time (x-axis) reported in seconds.

5. Summary

We’re excited to be introducing the Savina benchmark suite for actor-oriented programs. The benchmarks in Savina are diverse, realistic, and represent compute intensive applications. We encourage the community to submit open-source solutions to the benchmark for other actor libraries and languages. This will allow performance comparison across languages and also allow judging the elegance of the solutions.

We plan to add a few more applications into the next version of Savina. An important issue we are not addressing with the current release of the suite is inter-language comparisons. Future work will focus on examining a wider range of platforms and environments, and extending the benchmark suite to include codes which use more complex parallel algorithms. We will be making revisions on an ongoing basis in order to fix bugs or expand the scope of the benchmark suite.

References

S. Imam and V. Sarkar. Habanero-Java Library: a Java 8 Framework
http://www.scala-lang.org/node/54

S. Imam and V. Sarkar. Integrating Task Parallelism with Actors. In
Proceedings of the ACM international conference on Object Oriented

S. Imam and V. Sarkar. Habanero-Java Library: a Java 8 Framework
for Multicore Programming. In 11th International Conference on
the Principles and Practice of Programming on the Java Platform,

Imam, Shams and Sarkar, Vivek. Habanero-Scala: Async-Finish Pro-

D. Kafura. ACT++
D. Kafura. ACT++

W. Thies and S. Amarasinghe. An Empirical Characterization of
Stream Programs and Its Implications for Language and Compiler
Design. In Proceedings of the 19th International Conference on

A. Mason. The CountingActor benchmark.
http://www.

A. Mason. Theron performance.
http://www.

Microsoft Corporation. Asynchronous Agents Library.

W. Partain. The nofib Benchmark Suite of Haskell Programs.
In J. Launchbury and P. Sansom, editors, Functional Programming,

Rettig, Mike. retlang: Message based concurrency in .NET.
http://code.google.com/p/retlang/.

Rettig, Mike. jetlang: Message based concurrency for Java, 2014.
URL http://code.google.com/p/jetlang/.

J. Shirako, N. Vrvilo, E. G. Mercer, and V. Sarkar. Design, Verifica-
tion and Applications of a New Read-write Lock Algorithm. In
Proceedings of the Twenty-fourth Annual ACM Symposium on Par-
allelism in Algorithms and Architectures, SPAA ’12, pages 48–57.
10.1145/2312005.2312015.

J. Shun, G. E. Blelloch, J. T. Fineman, P. B. Gibbons, A. Kyrola,
H. V. Simhadri, and K. Tangwongsan. Brief Announcement: The
Problem Based Benchmark Suite. In Proceedings of the Twenty-
fourth Annual ACM Symposium on Parallelism in Algorithms and
URL http://doi.acm.org/10.1145/2312005.2312018.

J. Silillo. Stage: Exploring Erlang Style Concurrency in Ruby. In
Proceedings of the 1st international workshop on Multicore soft-
ware engineering, IWMSSE ’08, pages 33–40, New York, NY , USA,
10.1145/1370082.1370092. URL http://doi.acm.org/10.1145/
1370082.1370092.

M. Sulzmann, E. S. L. Lam, and P. V. Weert. Actors with Multi-headed
Message Receive Patterns. In Proceedings of the 10th International
Conference on Coordination Models and Languages, volume 5052 of
68264-3, 978-3-540-68264-6.

URL http://www.gpars.org/guide/.

The Scala Programming Language. pingpong.scala, 2012. URL

Thies and S. Amarasinghe. An Empirical Characterization of
Stream Programs and Its Implications for Language and Compiler
Design. In Proceedings of the 19th International Conference on

Wikipedia, The Free Encyclopedia. Cigarette smokers prob-


Wikipedia, The Free Encyclopedia. Sleeping barber prob-

URL http://liftweb.net/api/26/api/#net.liftweb.
actor.LiftActor.

Wyatt, D. Akka Concurrency - Building reliable software in a multi-
core world. Artima Incorporation, USA, 2013.

X. Zhao and N. Jamali. Load Balancing Non-uniform Parallel Compu-
tations. In Proceedings of the 2013 Workshop on Programming Based
on Actors, Agents, and Decentralized Control, AGERE’ ’13, pages
1145/2541329.2541337.

Submitted to AGERE’ 2014 5
2014/9/8