SALSA Lite: A Hash-Based Actor Runtime for Efficient Local Concurrency

Travis Desell[1] and Carlos A. Varela[2]

[1] Department of Computer Science, University of North Dakota
tdesell@cs.und.edu
[2] Department of Computer Science, Rensselaer Polytechnic Institute
cvarela@cs.rpi.edu

Abstract. As modern computer processors continue becoming more parallel, the actor model plays an increasingly important role in helping develop correct concurrent systems. In this paper, we consider efficient runtime strategies for non-distributed actor programming languages. While the focus is on a non-distributed implementation, it serves as a platform for a future efficient distributed implementation. Actors extend the object model by combining state and behavior with a thread of control, which can significantly simplify concurrent programming. Further, with asynchronous communication, no shared memory, and the fact an actor only processes one message at a time, it is possible to easily implement transparent distributed message passing and actor mobility. This paper discusses *SALSA Lite*, a completely re-designed actor runtime system engineered to maximize performance. The new runtime consists of a highly optimized core for lightweight actor creation, message passing, and message processing, which is used to implement more advanced coordination constructs. This new runtime is novel in two ways. First, by default the runtime automatically maps the lightweight actors to threads, allowing the number of threads used by a program to be specified at runtime transparently, without any changes to the code. Further, language constructs allow programmers to have first class control over how actors are mapped to threads (creating new threads if needed). Second, the runtime directly maps actor garbage collection to object garbage collection, allowing non-distributed SALSA programs to use Java's garbage collection "for free". This runtime is shown to have comparable or better performance for basic actor constructs (message passing and actor creation) than other popular actor languages: Erlang, Scala, and Kilim.

Keywords: Concurrent Programming, Actor Model, Actor Languages, Fairness, State Encapsulation, SALSA, Erlang, Kilim, Scala

1 Introduction

Actors model concurrency in open distributed systems [1, 2]. They are independent, concurrent entities that communicate by exchanging messages. Each

actor encapsulates a state with a logical thread of control which manipulates it. Communication between actors is purely asynchronous. The actor model assumes guaranteed message delivery and fair scheduling of computation. Actors only process information in reaction to messages. While processing a message, an actor can carry out any of three basic operations: alter its state, create new actors, or send messages to other actors (see Figure 1). Actors are therefore inherently independent, concurrent and autonomous which enables efficiency in parallel execution [3] and facilitates mobility [4, 5].

This paper describes the development of a new runtime for SALSA called SALSA Lite. This runtime was designed to perform the basics of actor based computation, simple message passing and actor creation, as efficiently as possible (see Section 4). Then the rest of SALSA's advanced message passing constructs, remote message passing, remote actor creation, and actor mobility are built using this optimized core. The strategy is to separate the overhead of distributed communication, universal naming, mobile computation, and distributed garbage collection, so that SALSA programs that run locally on multi-core processors do not have to pay the performance price for distribution and mobility.

Because encapsulation and fairness are guaranteed by the language semantics, it was possible to create a highly efficient and simple runtime to execute lightweight actors. The runtime itself is also based on the actor model, and is similar in structure to E's *vats* [7, 8]. It assigns lightweight actors to *stages*, each of which have a single thread and combined mailbox for every assigned actor. A stage processes messages from its mailbox sequentially on its assigned actors using their message handlers, and actors send messages to other actors by placing them in the target actor's stage. In this runtime, the stage is essentially a heavyweight actor, and lightweight actors can be implemented as simple objects.

Using this approach, it is also possible to provide first class stage support, allowing SALSA developers to specify what stage actors are assigned to, and dynamically create new stages as needed. Furthermore, the stage runtime maps actor garbage collection to object garbage collection, allowing the use of Java's garbage collection without additional overhead for non-distributed non-mobile SALSA programs. Results show that for benchmarks testing actor creation and message passing, SALSA Lite is two times faster than Kilim, two to ten times faster than Scala, and over an order of magnitude faster than Erlang. Additionally, SALSA Lite does this while providing actor garbage collection and ensuring state encapsulation, in contrast to Kilim and Scala.

2 Related Actor Languages and Frameworks

This section describes three commonly used languages with actor semantics: Erlang, Kilim, and Scala.

2.1 Erlang

Erlang is a functional programming language which allows concurrency via processes that use the actor model [9]. Erlang's scheduler accomplishes fairness by

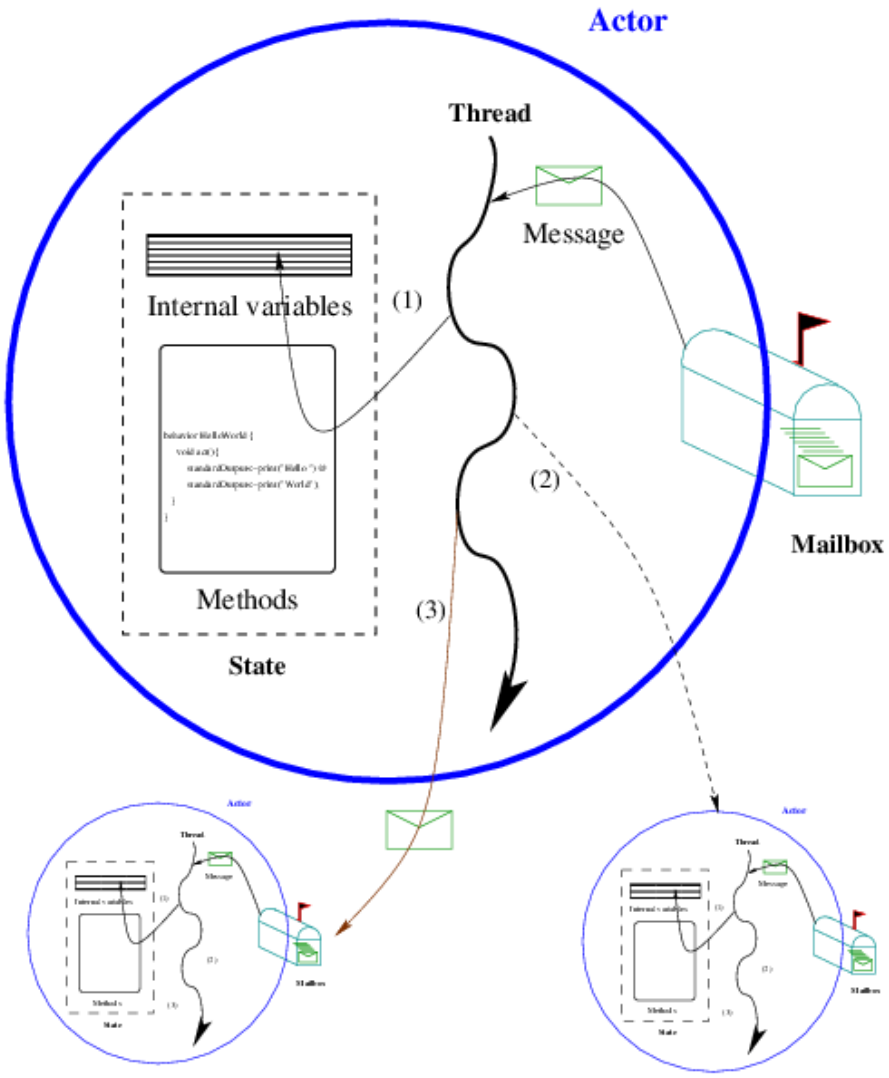

Fig. 1. Actors are reactive entities. In response to a message, an actor can (1) change its internal state, (2) create new actors, and/or (3) send messages to other actors (image from [6]).

counting the number of reductions, or function calls used by a process. When a process has made 1,000 reductions, it is paused and Erlang starts execution of a different process. This allows Erlang to scale to a large number of processes using a fixed number of actual processes without violating fairness.

The state of a process in Erlang can only be updated as the result of message passing. This coupled with safe message passing ensures state encapsulation. Safe message passing in Erlang is guaranteed by *single assignment*. Single assignment enforces that a value can be bound to a variable exactly once. As a consequence, all variables are immutable after their initial assignment. Since there is no way to directly update the state of other processes and variables passed in messages are immutable, there is no way to share mutable memory between Erlang processes.

2.2 Kilim

Kilim is an actor based message passing framework for Java. It uses byte-code transformation to convert specified Java objects into actors. Kilim actors use ultra-lightweight threads as well as safe, zero-copy message passing based on a type system [10]. Kilim's *weaver* transforms methods with a `@pausable` qualifier into continuation message passing. The resulting actor threads and continuation messages enable very fast context-switching via lightweight threads. Kilim uses a *linear ownership* type system to ensure that a message can have at most one owner at any time, which helps developers guarantee safe message passing.

However, Kilim requires users to explicitly copy Java objects when they are sent in messages, so it violates state encapsulation as actors can pass references to Java objects and access the same memory concurrently. Kilim actors can also be constructed with references to shared objects and access the mailbox and state of other actors directly. While having a reference to another actors mailbox allows actors to "send" messages, it also lets an actor "steal" messages from other actors' mailboxes. It also does not guarantee fair scheduling, as synchronous object method invocation and infinite loops may block an actor and the thread executing that and potentially other actors indefinitely, preventing it from processing further messages.

2.3 Scala

Scala provides a library `scala.actors`, heavily inspired by Erlang, to support the actor model. It supports synchronous and asynchronous message passing and fair scheduling by unifying threads and events [11]. However, it allows shared memory and synchronous execution of methods on other actors. While, this can be desirable in some programs, it can also result in a violation of actor semantics as well as data inconsistencies. Similarly, objects passed within messages may be accessed by multiple actors simultaneously leading to a loss of state encapsulation. Allowing synchronous message passing can also cause deadlocks [12, 13].

Scala actors can be either heavyweight, with each actor using its own thread, or event-based, using a thread pool to provide fairness. It is possible to combine

event-based and heavyweight actors in Scala. For event-based actors, Scala can use a single thread scheduler or a thread pool. Scala's thread pool scheduler will add a new thread to its thread pool if all worker threads are blocked due to long-running operations. This can be much more efficient than heavyweight actors, as the number of threads can typically remain constant if the worker threads are not continuously blocked. However, this implementation can still fail if enough actors are created that block worker threads, as the thread pool can run out of resources when the JVM cannot create any new threads.

3 The SALSA Lite Runtime

The SALSA Lite runtime was developed to execute the common case fast with the least amount of overhead. Message processing is accomplished via Java code, so in terms of the actor model the two most important common cases to execute fast are message sending and actor creation. Further, these need to be implemented in a way to protect state encapsulation and guarantee safe message passing.[3]

Figure 2 shows the runtime environment used by SALSA Lite. As the actor model provides a simple and efficient way to develop concurrent and distributed programs, the SALSA Lite runtime practices what we preach. It uses heavyweight actors (called *stages*) to simulate the execution of many concurrent lightweight actors in parallel, as with a heavyweight actor, each stage has its own mailbox and thread of control. Because of this, SALSA Lite actors are implemented as simple Java objects, consisting only of their state (object fields) and a reference to the stage they are *performing* or executing on, so other actors can easily send messages to them by placing those messages in their respective stage.

A drawback of this implementation is that if any message has an unbounded processing time, e.g., it enters an infinite loop or calls a blocking method invocation on an object like reading from a socket, the other actors on the same stage may starve. Currently, the solution to this problem is by creating an actor with its own stage, as described in Section 3.1, if it could potentially execute a message with unbounded execution time. This approach is also used by other performance focused actor implementaitons, such as `libcppa` [14].

Other implementations, which utilize thread pools (such as Scala) can also fall prey to this problem – if all threads in the threadpool are in an infinite loop or call a blocking method which never unblocks, actors waiting to process messages outside the thread pool will starve. Thread pools can also potentially cause significant performance overhead and can potentially cause the JVM to run out of resources when they cannot create new threads. Thread pools were examined for SALSA Lite, however they resulted in significantly worse performance. Further,

[3] *State encapsulation* refers to the inability to modify an actor's internal state other than indirectly by sending it messages. *Safe message passing* refers to the inability to missuse the message passing system in order to share memory and thereby break state encapsulation, e.g., by sending a reference to a mutable object in a message.

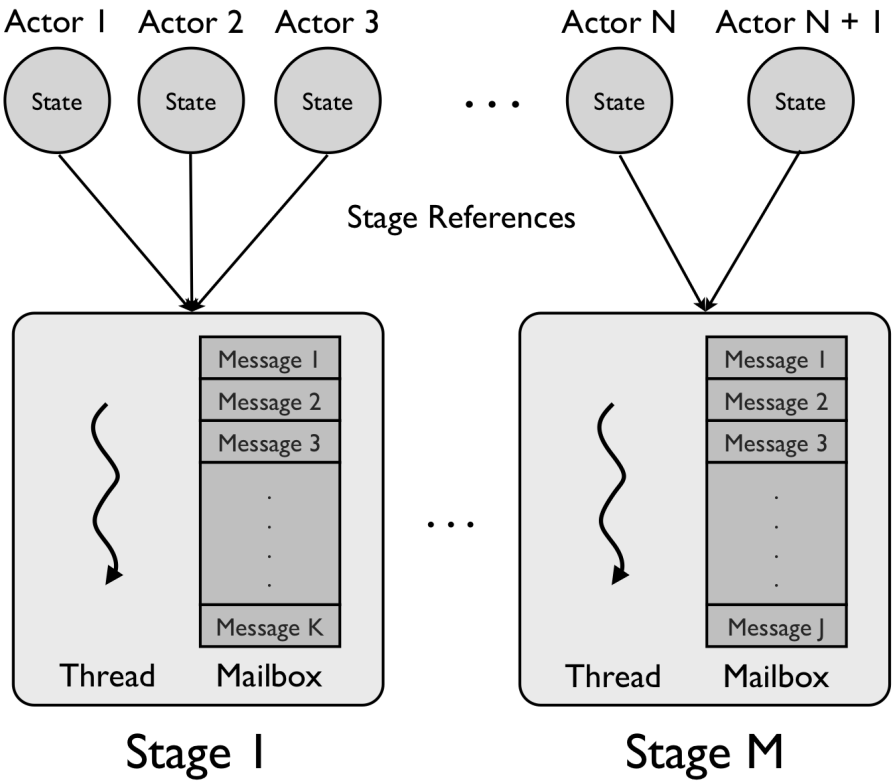

Fig. 2. The SALSA Lite runtime environment. Heavyweight actors called *stages* are used to process messages on multiple lightweight actors, simulating their concurrent execution. A stage will repeatedly get the first message from its mailbox and process that message on the message's target actor. Every actor is assigned to a stage. A Message sent to an actor is placed at the end of its assigned stage's mailbox.

they require each actor to also have some data structure to store their own individual mailbox, increasing memory requirements.

Examining methods for automatically quarantining actors with unbounded message processing behavior to their own stages, or other methods for efficiently ensuring fairness at the runtime level remains an area of future research.

3.1 Actor Creation

```
1: //create on a default stage
2: MyActor a = new MyActor();

3: //create b on a's stage
4: MyActor b = new MyActor() on (a);

5: //create c on stage 3
6: MyActor c = new MyActor() on (3);

7: //create d on its own new stage
8: MyActor d = new MyActor()
9:     on (StageService.getNewStage());
```

Fig. 3. SALSA Lite has first class support for which stage (or thread) an actor runs on. An actor can either use SALSA's default scheduling, run on the same stage as another actor, its own new stage, or a stage specified by an identifier.

Figure 3 gives an example of creating actors at different stages. The initial number of stages used can be specified at runtime, and these stages are identified 0 through $N-1$ where N is the number of stages. First class stage support can be used to create an actor at the same stage as another actor, a stage specified by its identifier, or its own new stage.

If an actor is created without specifying a target stage, the SALSA runtime uses a hash function to determine which stage it runs on. Currently, this is done using the actor's hash value (which is inherited from Java's default object hash value). This hash value was chosen over other strategies (such as generating a random number) for efficiency, as the hash values serve as random numbers and are already calculated by the Java runtime as part of object creation so there is no need to do additional calculation.

The stage the actor is placed on is the actor's hash value modulo the number of stages specified at runtime. This makes for an efficient way to distribute actors over stages in a generally balanced and random way. Hashing actors to stages is particularly interesting as a research question, as it provides transparent parallelism of SALSA Lite programs, allowing the number of stages to be specified at runtime, independent of the application's code. Further, it makes it possi-

ble to examine different hashing functions with respect to their load balancing capabilities and performance.

This implementation also allows SALSA programs to intermingle lightweight and heavyweight actors without any additional overhead, as a heavyweight actor is simply an actor running at a stage without any other actors. Furthermore, actors which communicate frequently can be assigned to the same stage so they do not have to pay the price of context switching when passing messages, which can result in significant performance as shown by the ThreadRing and Chameneos-Redux benchmarks in Sections 4.1 and 4.2, respectively. In this way SALSA actors have location *translucency*: a developer can simply specify an initial number of stages and have the SALSA runtime determine what stage actors will be assigned to, or the developer can have first class control over the number of stages used, what actors are assigned to them, and can even change the number of stages dynamically.

3.2 State Encapsulation

State encapsulation, asynchronous communication, and fairness are the main semantic concerns in actor languages. As stated by Karmani et al., "Without enforcing encapsulation, the Actor model of programming is effectively reduced to guidance for taming multi-threaded programming on shared memory machines" [15]. Asynchronous communication is critical in preventing deadlocks and to facilitate the execution of concurrent systems on distributed environments. Finally, fair scheduling ensures the correctness of an actor system composed of several existing systems [16]. Without state encapsulation, asynchronous communication, and fairness, it is not possible to guarantee the correct execution of an actor-oriented program.

Many current actor system implementations use a language or framework that combines both object-oriented and actor-oriented programming [15], such as the ActorArchitecture [17], the Actor Foundry [18], JavAct [19], Jetlang [20], Kilim [10] and Scala [11]. However, the combination of objects, threads, and actors can lead to inconsistencies in the actor model implementation. For example, if an actor passes a reference to an object to another actor within a message, both actors can then access the memory of that object concurrently which can lead to race conditions, deadlock, or memory inconsistency; nullifying many of the benefits of the actor model. Some approaches, such as Kilim's, have attempted to address this issue by zero-copy isolation types [10], while others simply allow these inconsistencies. Erlang monitors the call stack and suspends actor processing, yielding to others if an actor takes too long to process a message [9], and Scala uses a thread pool which will spawn new threads if message processing becomes blocked [11]. In summary, it is very difficult to guarantee state encapsulation and deadlock freedom; and often complicated run time solutions are necessary to ensure fairness.

SALSA Lite guarantees state encapsulation during the compilation process. The SALSA lite compiler generates Java objects for each actor, which have all their state fields and methods flagged as private. The compiler generates two

methods which take a message object and invoke the corresponding method or constructor on the actor, and these can only be invoked by that actor's controlling stage. Further, as SALSA Lite allows the use of Java objects, and the underlying implementation of actors and their references are objects, to prevent programmer confusion the compiler explicitly does not allow for methods to be invoked on actor references, and generates appropriate error messages. This guarantees state encapsulation of all actors.

3.3 Safe Message Passing

When a message is sent to an actor, it is placed in the mailbox of that actor's stage. Stages process messages in the same first-in, first-out manner as actors, except the messages are invoked on the target actor instead of the stage. As each actor is only assigned to a single stage, multiple messages will not be processed by an actor at the same time. Because this runtime is based on the actor model as well, there are very few synchronization points. The `LinkedList` of a stage's mailbox must be synchronized such that the thread will wait for new messages to be placed in the mailbox if it is empty, and incoming messages must be added to the mailbox one at a time. Inter-stage fairness follows from Java thread execution fairness. While the current implementation uses synchronization around the use Java's `LinkedList` class for a mailbox, performance may potentially be further improved by using lock-free data structures [14, 21–24], which will be investigated as future work.

When messages are sent, they must not allow direct access to the state of the actor sending the message, otherwise this would violate state encapsulation and distributed memory. One way to enforce this is by doing a deep copy on every argument passed in a message from one actor to another. However, this is not particularly efficient. Further, when an actor sends a message to another actor, the arguments of that message can either be references to other actors (whose state and references to objects and other actors do not need be copied) or objects, which do need to be copied. For a simple example, an argument to a message may be an `ArrayList` of actors. The `ArrayList` should be copied, but the actors (as well as the objects and actors referenced by those actors) it contains should not.

The SALSA Lite compiler uses static type checking and static method resolution which enable us to implement fast and safe message passing. In Java, primitives and immutable objects are passed by copy, while mutable objects are passed by reference. In order to sucessfully implement safe and efficient message passing in SALSA, primitives, immutable objects *and* mutable objects need to be passed by copy, while actors should be passed by reference.

Previous SALSA implementations use Java's default serialization interface, which would copy the entire message over a socket connection, which is not particularly efficient. In SALSA Lite, each stage only processes one message at a time, which allows the use of highly efficient and unsynchronized fast byte array input and output streams for the deep copy [25]. Further, as the SALSA lite compiler has static type checking, it can selectively copy only the message arguments

which require it (mutable objects), by wrapping those particular arguments in a *deep copy* call.

Actors were implemented as Java objects extending a simple `Actor` class. SALSA disallows direct access to any fields within an actor, and these objects to allow message passing from other actors. These references are essentially immutable, so it is safe to share them between actors and objects. State encapsulation is enforced in SALSA utilizing the `writeReplace` and `readResolve` methods of Java's `java.io.Serializable` interface. The SALSA compiler provides a `writeReplace` and `readResolve` for each actor. When an actor is to be serialized, Java will call the `writeReplace` method and instead serialize the object returned by that method. When that object is read, its `readResolve` method will be called and the result of that method used as the unserialized object. This is used to hijack the serialization process of actors, preventing them from being copied. The `writeReplace` places the written actor into a hash table (using a hash function which generates unique values, separate from Java's default implementation which potentially has collisions), and returns an object with the hash value for that actor. The `readResolve` method takes the hash value of the serialized object, looks up the actor in the hash table and returns that actor. This approach also has further benefits in that it allows actor references to be tracked when actors are serialized to remote locations via migration in distributed applications.

3.4 Garbage Collection

Erlang provides garbage collection via a mark-and-sweep algorithm [26]. The actor implementation on the Kilim and Scala languages do not provide garbage collection at all. Using the description of actor liveness and garbage presented by Kafura et al. [27], an actor is garbage if:

- it is not a root actor.
- it cannot potentially send a message to a root actor.
- it cannot potentially receive a message from a root actor.

We can define an *unblocked* actor as an actor that is either processing a message or has messages waiting for it in its mailbox. A *potentially unblocked* actor is an actor that another unblocked or potentially unblocked actor has a reference to (and thus messages could be sent to it). Because of this, an actor is garbage if it is not potentially unblocked [28, 29].

All SALSA actors have static references to standard output, standard input, and standard error (via Java), so they all have references to root actors and objects. Therefore, in SALSA there cannot be active garbage, or garbage actors that repeatedly send messages to each other, since if an actor is processing messages, it can potentially send messages to root actors. Detecting live (non-garbage) actors is therefore reduced to reachability from potentially unblocked actors, also called *pseudo-roots* [28, 29].

In Java garbage collection, objects are collected if they are unreachable by a non-system thread. In SALSA Lite, the only non-system threads are the threads

used by stages. In non-distributed programs, unblocked actors are always reachable by a stage thread, as the stage will have either a reference to the actor as it is processing a message, or a reference to a message in its mailbox which has a reference to that actor. As unblocked actors are always reachable by a stage thread, potentially unblocked actors are as well, because there will be a chain of references through other unblocked and potentially unblocked actors to every potentially unblocked actor. The only references to actors are in messages or in the state of an actor. If an actor is garbage, it is unreachable by any stage thread as there will be no messages to it in its stage mailbox and no unblocked or potentially unblocked actors will have a reference to it. Therefore it will be reclaimed by Java's garbage collector.

Because of this, the stage based runtime presented automatically maps local actor garbage collection to object garbage collection, allowing SALSA Lite to use Java's garbage collection to reclaim non-distributed garbage actors without additional overhead.

4 Performance Benchmarks

This section compares the performance of Erlang, Kilim, SALSA, and Scala with three different benchmarks. The ThreadRing benchmark measures the performance of message passing between concurrent entities (in this case, actors). Chameneos-Redux measures not only the performance of message passing, but also the fairness of scheduling for the concurrent entities. FibonacciTree measures the performance of message passing and actor creation, as well as the memory usage of many concurrent actors.

All experiments were run in a 2.93 GHz Intel Core 2 Duo MacBook Pro with 4 GB 1067 MHz DDR3 RAM, running Mac OS X 10.6.2. The Java version used was 1.6.0_17. The mean runtime for 25 experiments was used for all performance figures, and includes start up and shut down time (they were not run repeatedly within a JVM). The implementations of ThreadRing and Chameneos-Redux used by Java, Scala and Erlang were taken from the best performing versions at the *Computer Language Benchmarks Game*[4]. Scala 2.7.7, Erlang R14A, and Kilim 0.6 were used to perform the tests. The FibonacciTree has been used by SALSA in the past to test its performance, however it is not as well known as ThreadRing and Chameneos-Redux, so implementations were made for Erlang, Kilim and Scala using the same message passing strategy used by SALSA. Because of this it should be noted that there may be better performing implementations for Erlang, Kilim and Scala if written by an expert in those languages. However, the SALSA benchmarks were also programmed as typical programmers (e.g., see Appendix A and B for the SALSA code for the Fibonacci and ThreadRing benchmarks), and were not extensively optimized either.

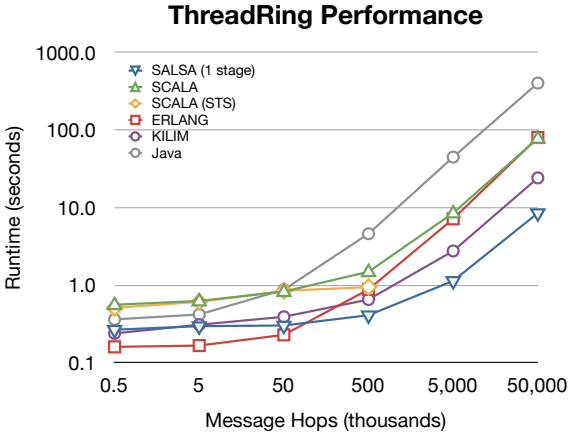

Fig. 4. The performance of Java, Kilim, Erlang, SALSA, and Scala for the ThreadRing benchmark. SALSA used a single stage runtime, while Scala used a single thread scheduler (STS) and its typical thread pool runtime. Kilim had indistinguishable results for a single and double thread scheduler. The Java implementation used standard Java threads and the `java.util.concurrent.locks.LockSupport` class for a locking mechanism.

4.1 ThreadRing

The specification for the ThreadRing benchmark states that 503 concurrent entities should be created and linked either explicitly or implicitly in a ring. Following this, a token should be passed around the ring N times. The ThreadRing benchmark provides a good measurement of the time to pass messages between actors. It also provides an interesting mechanism to examine the cost of context switching between actors (or threads), as only one is active at any given time while it is passing the token. Because of this, lightweight threading implementations which do not require context switching can provide significant speedup over heavyweight implementations.

Figure 4 compares the performance of Java, single stage SALSA, Kilim, Scala with a single thread scheduler (STS), typical Scala with a thread pool, and Erlang as the number of times the token was passed (message hops) was increased from 500 to 50,000,000. The runtime did not change much between 500 and 50,000 message hops, as the majority of this time was the startup cost of the runtime environment. While the startup cost of Erlang was the lowest, the performance overhead of message passing increased the fastest of the actor languages. Single stage SALSA had the fastest performance for message passing, and from 500,000 to 50,000,000 message hops had the lowest runtime. Single thread Scala had very fast message passing, however above 500,000 message hops it suffered

[4] http://benchmarksgame.alioth.debian.org/

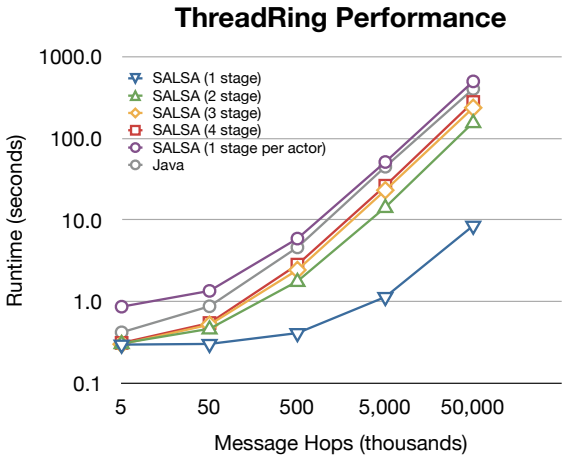

Fig. 5. The performance of SALSA using one to four stages, and to Java for the ThreadRing benchmark. This illustrates the high cost of thread context switching for this benchmark.

from stack overflow and could not complete the benchmark, because the message passing strategy used involved recursion and method invocation. Kilim had a similar startup time to SALSA, however message passing was not as fast. The Java ThreadRing had the worst performance, due to its traditional heavyweight thread usage. It should be noted that the runtime in the figure is a logarithmic scale, and that single stage SALSA had extremely fast message passing; for 50,000,000 message hops, SALSA was three times faster than Kilim, and an order of magnitude faster than Erlang and Scala, and almost two orders of magnitude faster than Java.

As the SALSA runtime allows the number of actors in the system to be independent from the number of stages, or threads, used; Figure 5 shows the runtime of the ThreadRing benchmark using one to four stages, and a heavyweight version with one stage per actor. With multiple stages, the high cost of context switching becomes apparent, as the more stages there are, the more threads the message must hop through, causing more context switching. This is further demonstrated as the heavyweight SALSA ThreadRing performance matches the Java performance with some overhead (approximately 23%). As only there is only one message being passed at a time around the ring, when it is passed between actors on different stages, it will not continue to be passed until the context switches to that other stage. This also illustrates the benefit of having first class control over what stage processes what actor. In a SALSA application, actors which communicate frequently can be assigned to the same stage and thus not have to pay the cost of context switching when message passing. This is not possible using a thread pool based runtime, as is done in Scala and Erlang.

4.2 Chameneos-Redux

The Chameneos-Redux benchmark not only tests the speed of message passing, but also the fairness of concurrency scheduling. Two runs are done, one with an odd number of creatures (three) and another with an even number of creatures (ten). For each run, the creatures repeatedly go to a meeting place and meet (or wait to meet) another creature. Each creature has a color and upon meeting another creature both change their color to the complement of the creature they met. This tests the performance of message passing as there are many messages between the chameneos creatures and the meeting place. Additionally, it tests fairness of concurrency scheduling as with an unfair scheduler, some creatures will meet more than others.

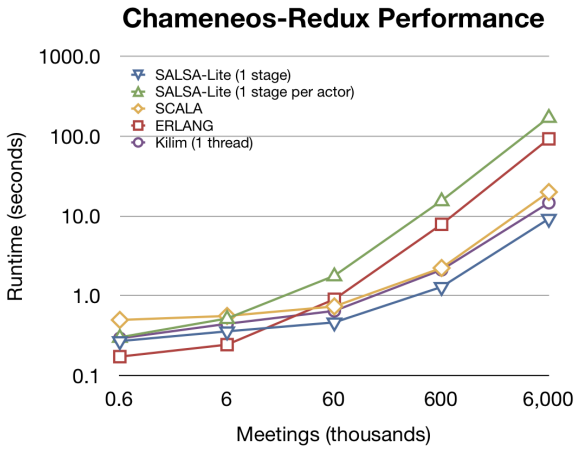

Fig. 6. The performance of Erlang, Kilim, SALSA, and Scala for the Chameneos-Redux benchmark. SALSA used both a lightweight runtime with a single stage and a heavy weight runtime which assigned each creature to its own stage. Only the thread pool version of Scala is shown as the benchmark had errors with a single thread scheduler. Kilim had indistinguishable results for both a single and double thread scheduler.

Figure 6 compares the performance of Erlang, Killim, Scala and SALSA for the Chameneos-Redux benchmark, as the number of meetings was increased from 600 to 6,000,000. SALSA used a single stage runtime and a heavyweight runtime with one actor per stage. As with ThreadRing, the single thread scheduler in Scala had runtime errors due to stack overflow, so only the thread pool version of Scala is given. Again, Erlang was the quickest to start up, however its message passing was slower than both Scala and the single thread SALSA runtime. As the heavyweight SALSA Chameneos-Redux required context switching for each message passed between the creatures and the meeting place, its performance was

very poor. As with the ThreadRing benchmark, single stage SALSA had the best runtime after startup costs became insignificant, being 1.75x faster than Kilim, 2.5x faster than Scala, and ten times faster than Erlang for 6,000,000 meetings.

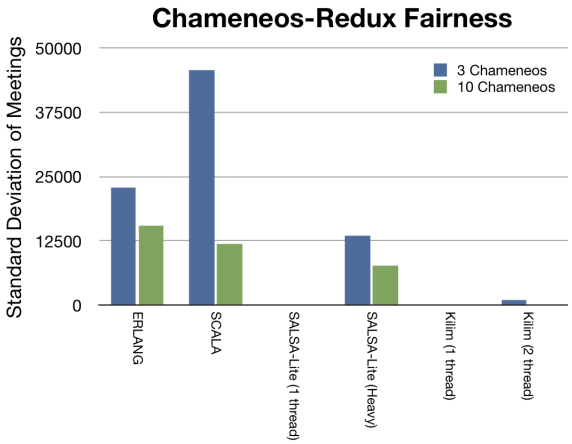

Fig. 7. The fairness of scheduling in Erlang, Kilim, SALSA and Scala. SALSA used both a single stage runtime and a multi-stage runtime that assigned each chameneos creature (and the meeting place) to its own stage. Kilim used both a single and double thread scheduler. The standard deviation between the meetings of creatures is shown, so a lower standard deviation is more fair concurrency. Single stage SALSA and single thread Kilim were perfectly fair with a standard deviation of 0.

Not only does the Chameneos-Redux benchmark test the speed of message passing between concurrent entities, it also provides a measure of the fairness of concurrency. With perfectly fair scheduling, each chameneos creature should have the same number of meetings. Figure 7 shows the standard deviation between the meetings of each chameneos creature for Chameneos-Redux with 6,000,000 meetings, run ten times for each language and runtime; for both the run with three creatures and the run with ten creatures. A lower standard deviation meant scheduling was more fair, as there was less difference in the number of times the creatures met. Both single stage SALSA and single thread Kilim were perfectly fair by processing messages in a first-in, first-out manner. Double thread Kilim had almost perfectly fair scheduling for an even number of creatures (10), and was almost perfect for an odd number (3). While a heavyweight Chameneos-Redux implementation in SALSA had the worst runtime, it had the next best fairness as it relied on Java's thread scheduling. Erlang and Scala had different fairness depending on the number of creatures. Erlang had better fairness for three creatures while Scala had better fairness with ten.

4.3 FibonacciTree

The last benchmark tested was a concurrent Fibonacci tree. This benchmark calculates the Fibonacci number using concurrent actors. A Fibonacci actor computes the Fibonacci number N by creating two child Fibonacci actors with the Fibonacci numbers $N-1$ and $N-2$, which create their own children and so on. If a Fibonacci actor is created with $N = 0$ it returns 0 to its creator, and if it is created with $N \leq 2$ it returns 1 to its creator. This benchmark not only tests the speed of message passing, but also the speed of creation of new actors. As this benchmark generates many actors, memory usage can be quite high. Because of this, Kilim, SALSA and Scala used the `-Xmx` and `-Xms` flags of the Java Virtual Machine to set the initial and maximum heap size to 2000MB, as the cost of allocating new memory has a significant effect on the runtime of the benchmark.

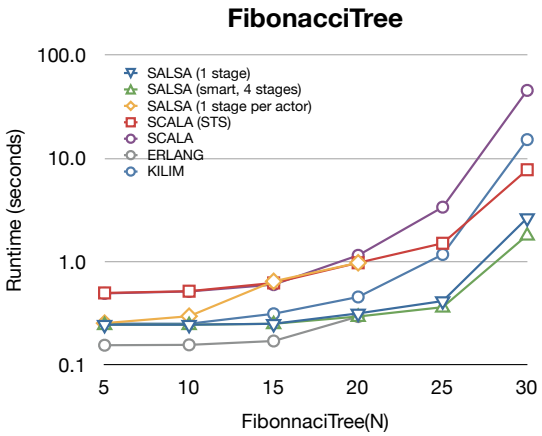

Fig. 8. The performance of the FibonacciTree for SALSA with a single stage, one stage per actor, and a smart implementation that placed subtrees across 4 stages, single thread scheduler and thread pool Scala, Kilim and Erlang. Both Erlang and heavyweight SALSA ran out of memory after FibonacciTree(20).

Figure 8 shows the performance of the FibonacciTree benchmark for SALSA with a single stage, one actor per stage, and a smart implementation that distributes subtrees across 4 stages, Kilim, Scala with a thread pool and a single thread scheduler, and Erlang. Both heavyweight SALSA and Erlang (which also uses a heavyweight actor implementation) failed after FibonacciTree(20), as no more resources were available to create new threads or processes. For FibonacciTree with $N > 20$, the smart implementation using four stages in SALSA had the best performance. This implementation shows the benefit of using first

class stage support to split the FibonacciTree into closely sized subtrees[5] and assigning each of these to its own stage to be processed in parallel, improving performance by 44% over single stage SALSA (1.8 seconds to 2.6 seconds). Kilim was initially faster than Scala due to its faster startup time, however the single thread scheduler for Scala had the next best performance for larger Fibonacci numbers. Using the thread pool runtime of Scala had the worst performance. For smaller Fibonacci numbers, Erlang had the best performance until it ran out of resources, due to its fast startup time.

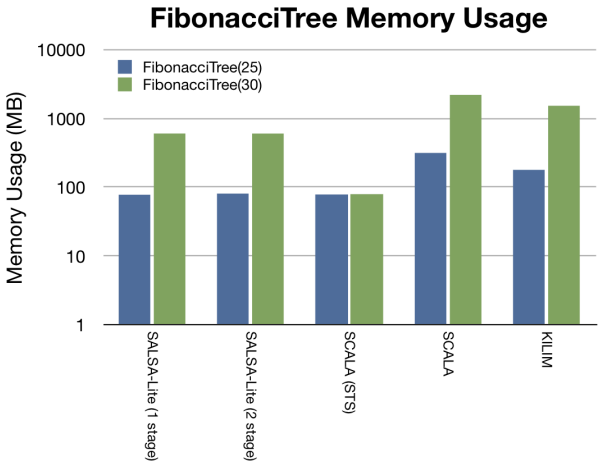

Fig. 9. Memory usage of FibonacciTree(25) and FibonacciTree(30) for SALSA with a single stage and the smart implementation with four stages, Kilim, and Scala with a single thread scheduler and thread pool. It should be noted that JVM memory usage was capped at 2000MB, and Scala with a thread pool could not allocate more than this amount of memory for FibonacciTree(30).

Scala with the single thread scheduler had the best memory usage, as it used recursion and method invocation on objects instead of actor creation and actual message passing, and thus had the interesting property of not requiring extra memory for larger Fibonacci numbers. Apart from this, SALSA had similar memory use for both a single and multi stage runtime, at 80MB for FibonacciTree(25) and 600MB for FibonacciTree(30). Kilim required the next least memory for FibonacciTree(25) and (30), around 180MB and 1530MB respectively. Scala required 300MB for FibonacciTree(25), and significantly more

[5] For FibonacciTree(30), stage 0 would be assigned FibonacciTree(28), stage 1 would be assigned FibonacciTree(27), stage 2 would be assigned FibonacciTree(27) and stage 3 would be assigned FibonacciTree(26).

memory for FibonacciTree(30), reaching the imposed limit of 2000MB with a thread pool runtime.

5 Discussion

This work describes an extremely efficient hash based runtime, in which actors are highly lightweight and independent from the threading mechanism used. The SALSA Lite runtime uses stages, similar to heavyweight actors, each with their own thread of control and mailbox, to simulate the concurrent execution of multiple lightweight actors. Each actor is assigned to a *stage*, either by an application developer using first class support to intelligently co-locate frequently communicating actors and give heavyweight actors their own thread, or by the SALSA Lite runtime. An added benefit of using this stage based runtime is that it automatically maps actor garbage collection to object garbage collection, and SALSA Lite can directly use Java's garbage collection for local (or non-distributed) concurrent programs.

Because of the stage based runtime and semantically guaranteed fairness and encapsulation, SALSA Lite significantly improved message passing performance and memory usage over earlier versions of SALSA. Results show that SALSA Lite's runtime is significantly faster than other existing actor implementations, two times faster than Kilim, between two and ten times faster than Scala, and over an order of magnitude faster than Erlang for the ThreadRing, Chameneos-Redux and FibonacciTree benchmarks. Additionally, with a similar result to single thread Kilim, SALSA Lite has perfect fairness using a single stage for the Chameneos-Redux benchmark. SALSA's memory usage was also less than Kilim and Scala using a thread pool scheduler, however not less than Scala's single thread scheduler which did not increase for larger FibonacciTree numbers due to its recursive method invocation, as opposed to actor creation based, strategy.

6 Future Work

While this paper describes SALSA Lite's non-distributed runtime and semantics in detail, SALSA also provides location transparency and mobility for distributed computing [5]. This lays the groundwork for extending the runtime presented with support for distributed applications with minimal overhead. Additionally, the strategy used for mapping actor garbage to local garbage will not work for distributed applications, so efficient distributed garbage collection is also required, e.g. [30, 31, 28].

Performance is limited by SALSA being implemented in Java. For example, in Section 4, Erlang consistently has the fastest startup time. While the Java implementation does have many benefits (like use of Java's libraries), it should be possible to have a significantly faster actor implementation if it was built up from a lower level, as done in ABCL [32] or `libcppa` [14]. This would also allow for a purely actor model implementation, and if developed in C or C++ would allow easy use of MPI, GPUs and many-integrated core accelerator cards, resulting in

an actor language for high performance computing with the additional benefits of transparent parallelism and mobility.

Further, in work done by Plevyak et al. [33] and in systems like ABCL [32], compile and runtime optimizations are used to process messages using local non-parallel function calls when applicable. This can result in significant performance increases, as it utilizes the stack instead of the heap, and message passing typically requires the creation of an additional message object and passing information about the messages sender and potential receiver for return values. SALSA Lite currently utilizes the heap for all message passing, with each message requiring creation of a message object, which is slower than a pure object method invocation. Another area of future work is to investigate strategies for using the stack and method invocation when possible, e.g.when multiple actors are on the same stage and are passing messages to each other which do not require continuations.

Further, to guarantee fairness in this runtime, currently a programmer needs to identify actors which could potentially process unbounded messages and assign them to their own stages. While this significantly reduces overhead, it may be desirable to have the runtime automatically enforce fairness by quarantining actors with long running messages to their own stage. A future area of research is to evaluate ways of enforcing fairness without significant overhead; as many applications do not require this enforcement.

The SALSA Lite runtime uses a hash based strategy to determine what stages actors are assigned to. An interesting avenue of research would be examining other scheduling strategies for assigning actors to stages in an intelligent manner; for example, in the ThreadRing benchmark, there is no reason to divide the actors over multiple stages and suffer from context switching. The runtime presented also gives first class support for assigning actors to threads. Previous work with the Internet Operating System (IOS) has shown that dynamic reconfiguration of distributed SALSA programs can be used to improve performance [34, 35]. The stage based runtime can be extended to enable local mobility of actors, allowing actors to dynamically change what stage they are assigned to. It also may be possible to improve application performance through intelligent middleware that profiles the runtime and rearranges actors based on their communication patterns.

Acknowledgements

This work has been partially supported by the National Science Foundation under NSF CAREER Award No. CNS-0448407, and by the Air Force Office of Scientific Research under Grant No. FA9550-11-1-0332.

References

1. Hewitt, C.: Viewing control structures as patterns of passing messages. Artificial Intelligence **8** (1977) 323–364

2. Agha, G.: Actors: a model of concurrent computation in distributed systems. MIT Press, Cambridge, MA, USA (1986)
3. Kim, W., Agha, G.: Efficient Support of Location Transparency in Concurrent Object-Oriented Programming Languages. In: Proceedings of Supercomputing'95. (1995) 39–48
4. Agha, G., Jamali, N.: Concurrent programming for distributed artificial intelligence. In Weiss, G., ed.: Multiagent Systems: A Modern Approach to DAI. MIT Press (1999)
5. Varela, C., Agha, G.: Programming dynamically reconfigurable open systems with SALSA. SIGPLAN Not. **36** (2001) 20–34
6. Varela, C.: Worldwide Computing with Universal Actors: Linguistic Abstractions for Naming, Migration, and Coordination. PhD thesis, U. of Illinois at Urbana-Champaign (2001) http://osl.cs.uiuc.edu/Theses/varela-phd.pdf.
7. Miller, M.S., Shapiro, J.S.: Robust composition: Towards a unified approach to access control and concurrency control. PhD thesis, Johns Hopkins University (2006)
8. Miller, M., Tribble, E., Shapiro, J.: Concurrency among strangers. Trustworthy Global Computing (2005) 195–229
9. Armstrong, J.: Programming Erlang: Software for a Concurrent World. Pragmatic Bookshelf (2007)
10. Srinivasan, S., Mycroft, A.: Kilim: Isolation-typed actors for Java. In Vitek, J., ed.: ECOOP. Volume 5142 of Lecture Notes in Computer Science., Springer (2008) 104–128
11. Haller, P., Odersky, M.: Actors that unify threads and events. In: Proceedings of the 9th International Conference on Coordination Models and Languages (COORDINATION). (2007) 171–190
12. Vermeersch, R.: Concurrency in Erlang and Scala: The actor model (2009) http://ruben.savanne.be/articles/concurrency-in-erlang-scala.
13. Varela, C., Agha, G.: What after Java? From Objects to Actors. Computer Networks and ISDN Systems: The International J. of Computer Telecommunications and Networking **30** (1998) 573–577 Proceedings of the Seventh International Conference on The World Wide Web (WWW7), Brisbane, Australia.
14. Schmidt, D.C.T.C., Hiesgen, R., Wählisch, M.: Native actors–a scalable software platform for distributed, heterogeneous environments. (2013)
15. Karmani, R.K., Shali, A., Agha, G.: Actor frameworks for the JVM platform: a comparative analysis. In: PPPJ '09: Proceedings of the 7th International Conference on Principles and Practice of Programming in Java, New York, NY, USA, ACM (2009) 11–20
16. Agha, G.A., Mason, I.A., Smith, S.F., Talcott, C.L.: A foundation for actor computation. Journal of Functional Programming **7** (1997) 1–72
17. Jang, M.W.: The Actor Architecture Manual. Department of Computer Science, University of Illinois at Urbana-Champaign. (2004)
18. Astley, M.: The Actor Foundry: A Java-based Actor Programming Environment. Open Systems Laboratory, University of Illinois at Urbana-Champaign. (1998–99)
19. Rougemaille, S., Arcangeli, J.P., Migeon, F.: Javact: a Java middleware for mobile adaptive agents. (2008)
20. Rettig, M.: Jetlang (2008–09) http://code.google.com/p/jetlang/.
21. Valois, J.D.: Lock-free data structures. (1996)
22. Alexandrescu, A.: Lock-free data structures. C/C++ User Journal (2004)
23. Herlihy, M., Luchangco, V., Moir, M.: The repeat offender problem: A mechanism for supporting dynamic-sized lock-free data structures. (2002)

24. Herlihy, M., Luchangco, V., Martin, P., Moir, M.: Nonblocking memory management support for dynamic-sized data structures. ACM Transactions on Computer Systems (TOCS) **23** (2005) 146–196
25. Isenhour, P.: Faster deep copies of java objects. (http://javatechniques.com/blog/faster-deep-copies-of-java-objects/) Accessed: 2/26/2013.
26. Armstrong, J., Virding, R.: One pass real-time generational mark-sweep garbage collection. In: In International Workshop on Memory Management, Springer-Verlag (1995) 313–322
27. Kafura, D., Washabaugh, D., Nelson, J.: Garbage collection of actors. SIGPLAN Not. **25** (1990) 126–134
28. Wang, W.: Distributed Garbage Collection for Large-Scale Mobile Actor Systems. PhD thesis, Rensselaer Polytechnic Institute (2006)
29. Wang, W., Varela, C.A.: Distributed garbage collection for mobile actor systems: The pseudo root approach. In: Proceedings of the First International Conference on Grid and Pervasive Computing (GPC 2006). Volume 3947 of Lecture Notes in Computer Science., Taichung, Taiwan, Springer (2006) 360–372
30. Kamada, T., Matsuoka, S., Yonezawa, A.: Efficient parallel global garbage collection on massively parallel computers. In: Proceedings of the 1994 conference on Supercomputing, IEEE Computer Society Press (1994) 79–88
31. Wang, W.J., Varela, C., Hsu, F.H., Tang, C.H.: Actor garbage collection using vertex-preserving actor-to-object graph transformations. In: Advances in Grid and Pervasive Computing. Volume 6104 of Lecture Notes in Computer Science., Bologna, Springer Berlin / Heidelberg (2010) 244–255
32. Taura, K., Matsuoka, S., Yonezawa, A.: An efficient implementation scheme of concurrent object-oriented languages on stock multicomputers. ACM SIGPLAN Notices **28** (1993) 218–228
33. Plevyak, J., Karamcheti, V., Zhang, X., Chien, A.A.: A hybrid execution model for fine-grained languages on distributed memory multicomputers. In: Proceedings of the 1995 ACM/IEEE conference on Supercomputing (CDROM). Supercomputing '95, New York, NY, USA, ACM (1995)
34. Desell, T., Maghraoui, K.E., Varela, C.A.: Malleable applications for scalable high performance computing. Cluster Computing (2007) 323–337
35. Maghraoui, K.E., Desell, T., Szymanski, B.K., Varela, C.A.: The Internet Operating System: Middleware for adaptive distributed computing. International Journal of High Performance Computing Applications (IJHPCA), Special Issue on Scheduling Techniques for Large-Scale Distributed Platforms **20** (2006) 467–480

A Fibonacci.salsa

A simple concurrent Fibonacci program in SALSA. The SALSA syntax is extremely similar to Java's syntax, and it can utilize all of Java's libraries (lines 9, 26). The `new` command creates a (concurrent) actor (lines 20 and 21), and `<-` sends asynchronous messages (lines 11, 20, 21). If a message or result of a message requires the result of another message (lines 11, 20, 21) it will not be sent until the required result has been sent with the `pass` statement (lines 16, 18, 20, 21), similar to a `return` statement. The constructor taking a array of arguments serves as an actor's `main` method.

```
1: behavior Fibonacci {
2:    int n;
3:
4:    Fibonacci(int n) {
5:        self.n = n;
6:    }
7:
8:    Fibonacci(String[] arguments) {
9:        n = Integer.parseInt(arguments[0]);
10:
11:        self<-finish( self<-compute() );
12:    }
13:
14:    int compute() {
15:        if (n == 0) {
16:            pass 0;
17:        } else if (n <= 2) {
18:            pass 1;
19:        } else {
20:            pass new Fibonacci(n-1)<-compute() +
21:                 new Fibonacci(n-2)<-compute();
22:        }
23:    }
24:
25:    ack finish(int value) {
26:        System.out.println(value);
27:    }
28: }
```

B ThreadRing.salsa

A simple concurrent ThreadRing program in SALSA. A JoinDirector (line 20) is an actor that provides a method for waiting for a group of messages to complete before sending another message. After an actor completes a message, it sends a join message to the JoinDirector (lines 27 and 30), which will resolve after it has received a number of messages specified by sending a resolveAfter message (line 32). In this case, only after the JoinDirector receives threadCount messages, the forwardMessage will be send (line 33).

```
1: import salsa_lite.language.JoinDirector;
2:
3: behavior ThreadRing {
4:    ThreadRing next;
5:    int id;
6:
```

```
 7:    ThreadRing(int id) {
 8:       self.id = id;
 9:    }
10:
11:    ThreadRing(String[] args) {
12:       if (args.length != 2) {
13:          System.out.println("Usage: java ThreadRing <threadCount> <hopCount>");
14:          pass;
15;       }
16:
17:       int threadCount = Integer.parseInt(args[0]);
18:       int hopCount = Integer.parseInt(args[1]);
19:
20:       ThreadRing first = new ThreadRing(1);
21:       JoinDirector jd = new JoinDirector();
22:
23:       ThreadRing next = null;
24:       ThreadRing previous = first;
25:       for (int i = 1; i < threadCount; i++) {
26:          next = new ThreadRing(i + 1);
27:          previous<-setNextThread(next) @ jd<-join();
28:          previous = next;
29:       }
30:       next<-setNextThread(first) @ jd<-join();
31:
32:       jd<-resolveAfter(threadCount) @
33:       first<-forwardMessage(hopCount);
34:    }
35:
36:    ack setNextThread(ThreadRing next) {
37:       self.next = next;
38:    }
39:
40:    ack forwardMessage(int value) {
41:       if (value == 0) {
42:          System.out.println(id);
43:          System.exit(0);
44:       } else {
45:          value--;
46:          next<-forwardMessage(value);
47:       }
48:    }
49: }
```