Selectors: Actors with Multiple Guarded Mailboxes

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

Abstract

The actor programming model is based on asynchronous message passing and offers a promising approach for developing reliable concurrent systems. However, lack of guarantees to control the order in which messages are processed next by an actor makes implementing synchronization and coordination patterns difficult. In this work, we address this issue by introducing our extension to the actor model called *selectors*. Selectors have multiple mailboxes and each mailbox is guarded i.e. it can be enabled or disabled to affect the order in which messages are processed. The view of having guarded mailboxes is inspired by condition variables where a thread checks whether a condition is true before continuing its execution.

Selectors allow us to simplify writing of synchronization and coordination patterns using actors such as *a*) synchronous request-reply, *b*) join patterns in streaming applications, *c*) supporting priorities in message processing, *d*) variants of reader-writer concurrency, and *e*) producer-consumer with bounded buffer. We present solutions to each of these patterns using selectors. Selectors can also be implemented efficiently – we evaluate the performance of our library implementation of selectors on benchmarks that exhibit such patterns and we compare our implementation against actor-based solutions using Scala, Akka, Jetlang, Scalaz, Functional-Java and Habanero actor libraries. Our experimental results for the benchmarks show that using selector-based solutions simplify programmability and deliver significant performance improvements compared to other actor-based solutions.

Categories and Subject Descriptors D.1.3 [*Programming Techniques*]: Concurrent Programming—Parallel programming

AGERE! 2014, October 20 2014, Portland, OR, USA.

http://dx.doi.org/10.1145/2687357.2687360

General Terms Actor Model, Asynchronous Messaging, Guarded Mailboxes, Synchronization, Coordination

Keywords Actor Model, Selector, Guarded Mailboxes, Synchronous Receive, Split-Join Pattern, Message Priorities, Reader-Writer Concurrency, Producer-Consumer with Bounded Buffer

1. Motivation

With the advent of the multicore era, it is clear that improvements in application performance will primarily come from increased parallelism. Programming models that utilize multiple cores offer a scalable solution for the future where core counts are expected to increase. The Actor Model (AM) [1, 11, 12] of concurrency has recently gained popularity, mainly due to the success achieved by its flagship language - Erlang [3]. The AM offers a promising approach for developing reliable concurrent systems. Actors provide a programming model that gives stronger guarantees about concurrent code such as data race freedom and location transparency when compared to traditional shared-memory-based abstractions such as fork-join tasks. The key idea is to encapsulate mutable state and use asynchronous messaging to coordinate activities among actors. This implies that the sender does not wait for a message to be received upon sending it; it immediately continues execution after issuing the send.

Despite being a model of concurrent computation, the AM is not a silver bullet - not all concurrent programming problems are best solved by the AM. Synchronization mechanisms are needed in concurrent programming to control the order in which messages are processed so as to preserve the integrity of objects [34]. However, for developers who are new to the AM, understanding and managing such synchronization and coordination in an asynchronous model might be harder than in the shared-memory model. Coordination patterns involving multiple actors are particularly difficult. The property of no shared state and asynchronous communication can make implementing coordination protocols harder than using established constructs like locks [32]. Solutions for such protocols to support synchronization constraints may require the actor to buffer messages and resend the messages to itself until the message is processed [21]. The resulting code is a petri dish of code that intertwines

both algorithmic logic and synchronization constraints. In addition, recycling of messages in the mailbox is expensive due to additional overhead from message sending and maintenance of internal buffers. There is, hence, a need for an abstraction to support synchronization and coordination mechanisms that integrate well with the AM.

In this work, we describe our extension to the AM to address this issue. We believe that this is a powerful extension, analogous to that of adding condition variables [13] to semaphores. Our extension, called *selectors*, allows actors to have multiple mailboxes (Section 2). While messages can be sent to any specific mailbox of the selector, each of these mailboxes are guarded. The guard refers to the ability to manage which mailboxes are searched for processing the next message of the selector in response to the selector processing its current message. With selectors, the major change to using regular actors is that along with the message, the `send` operation receives an additional argument, the recipient mailbox name. We briefly describe our library based implementation of selectors in Scala in Section 2.1 and evaluate performance against other popular actor frameworks that run on the Java Virtual Machine (JVM).

The selectors model allows us to simplify writing of common synchronization / coordination patterns using actors such as *a*) synchronous request-reply [14], *b*) join patterns in streaming applications [2], *c*) supporting priorities in message processing [23], *d*) variants of reader-writer concurrency [41], and *e*) producer-consumer with bounded buffer [40]. Each of these patterns are further described in Section 3, Section 4, Section 5, Section 6, and Section 7, respectively in the following format. We first summarize the difficulties with an actor-based solution. Next, we present selector-based solutions to each of these patterns. Finally, we provide performance results for the selector-based and actor-based solutions. Our experimental results for the benchmarks show using selector-based solutions deliver significant improvements in performance compared to actor-based solutions. The remaining sections in the paper include Section 8 which discusses related work and Section 9 where we summarize our conclusions and identify opportunities for possible future work.

2. Selectors: Actors with Guarded Mailboxes

We have extended the basic AM and call our extension *Selectors*. Like an actor, a selector is defined as an object that has the capability to process incoming messages. Selectors continue to encapsulate their local state and process one message at a time (as shown in Figure 1). Thus the benefits of modularity from the AM are still preserved and the data locality properties of the AM continue to hold. Selectors differ from actors in in two main ways:

- Selectors have multiple mailboxes to receive messages, senders specify which mailbox a message is targeted to. The sender can be any entity: an actor or a selector. As with actors, sending messages to any of the mailboxes is a non-blocking operation. Messages can be concurrently added to different mailboxes without synchronization and thereby reducing contention.
- Each mailbox maintains a boolean guard which is used to internally *enable* or *disable* the mailbox while the selector is processing messages. The guard does not affect which mailbox can receive messages; it only affects which mailboxes are inspected to select the next message to be processed. Attempting to enable an active mailbox and to disable an inactive mailbox are considered to be no-ops, not errors. It is an error if all the mailboxes become disabled while no message is actively being processed[1].

We observe that a standard actor can just be viewed as a selector with a single mailbox.

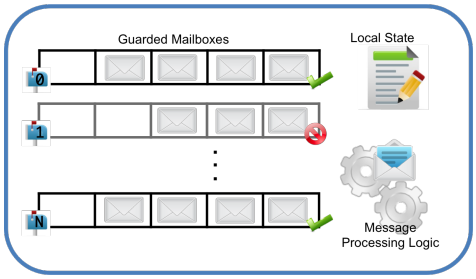

Figure 1: Decomposition of a selector: guarded mailboxes, local state, and message processing logic.

Our extension to actors is inspired by condition variables [13] where a thread checks whether a condition is true before continuing its execution. The thread simply waits for an event that changes the state of the condition variable and enables the waiting thread to continue execution. Similarly, selectors check conditions (guards) on their mailboxes and only process messages from *active* mailboxes. The selector waits for particular mailboxes to become active (state change) before considering messages from the activated mailboxes for processing. Logically, every condition variable is associated with one (or more) boolean condition expressions. Similarly, every mailbox is guarded with one (or more) boolean condition expressions that control when it is enabled.

Life Cycle of a Selector The life cycle of a selector is similar to that of an actor. As shown in Figure 2, a selector is in one of the following three states:

- *new*: An instance of the selector (including its mailboxes) has been created; however, the selector is not yet ready to process messages from its mailboxes. Other entities can send messages to the selector at any of its mailboxes. Initially all the mailboxes of the selector are enabled.

[1] Alternatively, we can treat this case like an `exit` operation (after which new messages can accumulate but will never be processed).

- *started*: A selector moves to this state from the *new* state when it has been started using the `start` operation. It can now receive asynchronous messages and process messages from any active mailbox one at a time. During processing of a message, the selector can enable or disable any of its mailboxes. After completing processing of a message, a message can be selected from any active mailbox to be processed next. While processing a message, the selector should continually receive any messages sent to it without blocking the sender. There is no restriction on the order in which the selector decides to process incoming messages from active mailboxes, thereby leading to non-determinism.
- *terminated*: The selector moves to this state from the *started* state when it has been terminated and will not process any messages in its mailboxes or new messages sent to it. A selector signals termination by using the `exit` operation on itself while processing a message. Any messages sent to a terminated selector will be ignored. A terminated selector may not be restarted.

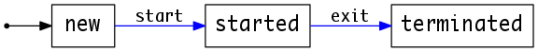

Figure 2: Life cycle of a selector.

Sending messages to a Selector All messages sent to selectors are asynchronous, meaning that the sender continues processing before any reply is received. At its simplest, the `send` operation receives two arguments: the target mailbox name and the actual message to send. Selectors offer flexibility in determining the target mailbox of a message; it may be decided in two ways:

- *By the sender*: The sender can directly specify the target mailbox as an argument in the `send` operation. The sender may have received the mailbox name in a message or the name could be globally known.
- *By the recipient selector*: The recipient can abstract away the decision making of which target mailbox to use from the sender. In such scenarios, the sender just invokes the `send` operation with the message (just like sending messages to actors). The selector can then introduce logic to introspect the message and its internal state to decide the recipient target mailbox.

A selector can choose to use a combination of both schemes where recipient mailboxes are decided by the sender for some messages, while the recipient selects the target mailboxes for the remaining messages.

Declarative Guards Selectors can be further enhanced by introducing explicitly declared predicated guard expressions on mailboxes (making selectors more functional in nature). These guard expressions are *registered* for each mailbox after the selector instance has been created and may be updated later. The selector runtime ensures the mailboxes are enabled or disabled at the end of processing each message based on the result of evaluating the predicate. This avoids the use of imperative `enable` and `disable` operations on mailboxes. This style of using declarative guards also separates the message processing logic from the logic to enable or disable mailboxes. The trade-off is that the style introduces additional overhead of possibly redundant computations of the guard expressions for each mailbox. Section 7 presents an example use case where declarative guards can be used to simplify selector-based solutions.

2.1 Implementation and Micro-benchmarks

Actors, being a model of computation, can be used to emulate selectors. However, doing so would require hand-coded implementations of mailboxes as part of the actor's local state, and would require messages to be recycled or resent. On the other hand, actors can be considered a special case of selectors with just one mailbox that is always enabled. Hence, any actor-based solution to a pattern is also a valid selector-based solution. We chose to implement selectors using the lower-level constructs described in our previous work on actors [16] and we can use the `finish` construct for detecting termination of selectors. Our implementation of selectors is an extension to the actors implementation in the Habanero-Java library [17] and we rely on Java's `ForkJoinPool` to manage the worker threads.

The selector maintains a list of mailboxes; each individual mailbox is implemented as a concurrent linked-list. Each node in the list is a pair of values consisting of *a*) the actual message, and *b*) a state field to track whether this message is pending, is being actively processed, or has been processed. As messages are sent to the selector's mailboxes, a chain of these pairs are built. As each worker picks up a message to process, the state field is updated accordingly to avoid duplicate processing of the message by other concurrent worker threads. Each node in the list is ready for garbage collection once the message inside the node is processed. The size of the mailbox is only limited by the amount of available memory.

Once the selector has been started (via the call to `start()`), it registers a callback to schedule a task as soon as a message becomes available in an active mailbox. A worker thread eventually executes this task, the task then tries to process as many available messages from a particular selector by traversing messages in active mailboxes until there are no more messages to process. Once such a state is reached the worker thread then goes ahead and tries to execute other ready tasks (selectors with pending messages) from its work queue. Before the worker thread is let go, if the selector has not terminated, a callback is registered that triggers an asynchronous task whenever a new message is sent to an active mailbox of the selector. In this scheme if a selector has no work to do, it doesn't consume any worker thread cycles as there are no thread-blocking operations. Hence, this solution

scales well as worker threads are never blocked and are busy doing effective work in the processing of messages. Worker threads become idle only when there is no pending work (tasks in the work queue).

Number of Mailboxes In our model of selectors, the number of mailboxes used can be determined dynamically (based on the distinct mailbox names used in the `send` operations). However in our implementation, the number of mailboxes to use by a given selector is configured during initialization. Each mailbox in a selector is identified by a unique id, a non-negative integer starting at zero. A boolean array is also maintained to track the value of the guard condition whether a given mailbox is active or inactive. This boolean array simplifies the implementation of enabling or disabling mailboxes, making it a constant time operation. Using invalid mailbox ids in `send` operations results in a runtime error.

Message Selection Policy In our current implementation, messages are selected using a Multilevel Queue (MQ) scheduling algorithm [29, 39]. We exploit the fact that messages are permanently assigned to one of the mailboxes. The mailboxes are given priorities using the ordering id of the enumeration values. The MQ scheduling technique ensures that at any given time, the worker thread picks a message from the lowest numbered active mailbox of all those messages that are currently ready to be processed. A message from a lower-priority mailbox is scheduled for execution only if all higher-priority active mailboxes are empty or disabled. One drawback of the MQ policy is that messages in higher numbered mailboxes are vulnerable to starvation and could wait an indefinite amount of time before being processed. An alternative solution, that can help address starvation issues, would be to randomly select messages from an active mailbox to process next. We observe that there is a possibility of a message remaining indefinitely in a mailbox's queue in pure actor models that do not require FIFO processing of messages.

Since our benchmarks (Section 2.1.1) are unaffected by starvation issues, we leave options for other message selection policies as possible future work. In particular, we wish to explore a Least-recently used Round-robin (LR) scheduling algorithm [29, 42]. In LR, the runtime will remember the least-recently used mailbox, *LRUM*, which provided a message to be processed. While searching for the next message to process, the runtime will start the search from the LRUM and return a message if that mailbox is enabled and contains an unprocessed message. Otherwise, the runtime will search for a message by cycling through the remaining mailboxes starting at the mailbox after the current LRUM. When a message is found from an active mailbox, the LRUM will be updated and the message will be scheduled for processing. Thus, LR will use round-robin scheduling with a dynamic time slice which is infinite until the mailbox becomes empty or disabled at which point the time slice becomes zero.

2.1.1 Performance Evaluation

The actor libraries used for comparison with our implementation, Habanero Selector (HS), all run on the JVM. The libraries are: Scala actors (SC) [8], Akka (AK) [35], Functional Java (FJ) [7], Jetlang (JL) [27], Scalaz (SZ) [15], and Habanero Actors (HA) [16, 17, 20]. SC provides event-based actors which allow multiple actors to run on a thread. AK is a framework for building event-driven applications on the JVM and has support for highly performant lightweight actors. SC has been deprecated since version 2.10 and replaced by AK in the standard distribution of Scala since version 2.11.0. FJ is an open source Java library for applying functional programming concepts (including concurrency abstractions) and is intended for use in production applications. JL provides a low-level messaging API in Java that can be used to build actors with the responsibility of ensuring the single message processing rule delegated to the user. SZ has an actor implementation that has a simple API and minimizes latency and maximizes throughput of message passing. HA API is inspired by SC event-based actors, however it does not use exceptions to maintain control flow and uses a push-based linked-list implementation using Java's `AtomicReference` for its mailbox. HS is our implementation of selectors using the mailbox implementation from HA. 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 (instead of `if-then-else` and `instanceof` checks) 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 libraries.

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, Habanero-Java library 0.1.3 (contains both actors and selectors), Scala 2.11.0, Akka 2.3.2, Function-Java 4.1, Jetlang 0.2.12, and Scalaz 7.1.0-M6. Each benchmark was configured to run using thirteen worker threads (the main thread gets blocked after initialization waiting for the computation to complete). We used the same JVM configuration flags (`-Xmx16384m -XX:-UseGCOverheadLimit -XX:+UseParallelGC -XX:+UseParallelOldGC`) and each benchmark 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.

2.1.2 Message Throughput (ForkJoin)

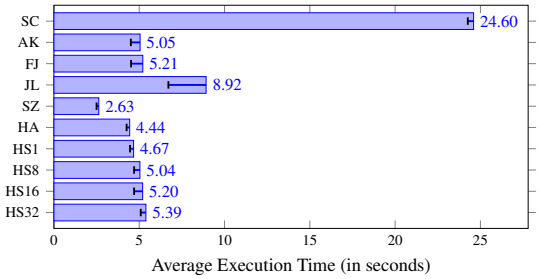

Figure 3: The `Fork-Join` benchmark using 60 actors, each actor was sent 400,000 messages in its mailbox. For the selector versions, the messages were sent round robin to each of the mailboxes.

The results of the actor variant of the Java Grande Forum `Fork-Join` benchmark [6] are shown in Figure 3. This microbenchmark measures messaging throughput, the rate at which messages are processed by the implementation. Each actor does a minimal amount of work processing one message, the actor processes a total of *N* messages before it terminates. Among the actor implementations, SZ, AK, FJ, and HA versions have competitive performance, while JL and SC are slightly slower. HS shares the same implementation scheme as HA, but differs in the way messages are searched from the mailboxes and performs slower than HA. In effect this slowdown reveals the overhead of message lookup due to the introduction of multiple mailboxes and the checking of guards compared to the Habanero actors implementation. Comparing the single mailbox version (HS1) against HA reveals this overhead to be at about 5%. As the number of mailboxes increases, this overhead also increases which is why the HS versions with 8, 16 and 32 (HS8, HS16 and HS32 respectively) mailboxes perform worse than the 1 mailbox version.

2.1.3 Mailbox Contention (Chameneos)

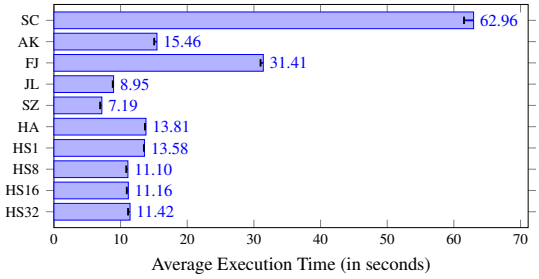

Figure 4: The Chameneos benchmark was run with 500 chameneos (actors) constantly *arriving* at a mall (another actor). There are 8,000,000 meetings between chameneos orchestrated at the mall.

The `Chameneos` microbenchmark, shown in Figure 4, measures the effects of contention on shared resources while processing messages. The original SC implementation was obtained from the public Scala SVN repository [10]. 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 SC version pays the penalty of generating exceptions to maintain control flow in its `react` construct. On the other hand, the AK version is competitive with HA, JL, and SZ. There are no guards applied to the mailboxes in the HS* versions. The messages in the mall selector are processed from the first mailbox before searching the second mailbox for messages and so on, this is not necessarily the order in which the messages were received by the selector. The HS version with one mailbox (HS1) has similar performance as HA as they are effectively similar implementations. The HS* versions with more than one mailbox benefit from having multiple mailboxes as it reduces the synchronization contention while Chameneos actors concurrently send messages to the mall selector. In general, increasing the number of mailboxes improves performance of this microbenchmark. Eventually the overheads of searching messages should start dominating and having higher number of mailboxes should start giving poorer performance (HS32 performs slightly poorer than HS16).

3. Synchronous Request-Response Pattern

The synchronous request-response pattern [14, 25] occurs when a requestor sends a request message to a replier system which receives and processes the request, ultimately returning a message in response. In the AM, the notion of synchronous request-response occurs when an actor sends another actor a message and stalls further processing of messages until it receives a reply to its message. As the AM is asynchronous, this pattern requires two asynchronous messages, a request and a response. While being conceptually elegant, this pattern is hard to implement efficiently, because the requestor actor's single mailbox must handle both the reply message and new messages sent to it from other actors. An option is to use pattern matching on the set of pending messages to implement the receive operation, but this can be expensive to implement due to the increase in time while searching for the next message to process from the mailbox.

```
1  class ReqRespActor extends ScalaActor {
2    def act() {
3      loop {
4        react {
5          case m: SomeMessage =>
6            // a case where we want a response
7            val req = new SomeRequest(m)
8            anotherActor.send(req)
9            receive {
10             case someReply: SomeReply =>
11               ...
12           }
13         case ... => // logic for other messages
14     } } } }
```

Figure 5: Using Scala actors to solve the Request-Response Pattern using thread-blocking `receive`.

The synchronous message passing style available in Scala actors (using `receive` at line 9 in Figure 5) provides programmers with a convenient way of doing messaging round-trips [8]. When the actor receives a message that is not matched, it will stay in the mailbox of the actor and is retried when a new `receive` block is entered. Using `receive` makes the actor heavyweight, since `receive` blocks the underlying thread while the actor is suspended waiting for a reply message. Another practical option to support this feature, to avoid complications in the processing of existing messages, is implemented using some notion of blocking explicitly and usually limits scalability. For example, the *ask* pattern (line 7 of Figure 6) uses a thread blocking `await` on the future's value (line 8). The responding actor completes the future when it sends a response after processing the message thereby unblocking the previously mentioned thread.

```
1  class ReqRespActor extends AkkaActor {
2    def receive = {
3      case m: SomeMessage =>
4        // a case where we want a response
5        val req = new SomeRequest(this, m)
6        implicit val timeout = Timeout(600 seconds)
7        val aFuture = ask(anotherActor, req)
8        val someReply = Await.result(aFuture, Inf)
9        ...
10     case ... => // logic for other messages
11   } }
```

Figure 6: Using Akka actors to solve the Request-Response Pattern. The *ask* pattern uses a thread blocking await on the future's value until the responding actor completes the future by sending a reply.

It is cumbersome to support synchronous reply in a non-blocking manner since it requires preventing the actor from processing other messages in its mailbox. A non-blocking solution involves *stashing* the messages an actor receives until it finds the reply message. Once the reply message has been found and processed, the stashed messages need to be *unstashed* into the actor's mailbox to resume processing of these messages. While a user can code this pattern manually, Akka provides the `become` and `become` operations and `Stash` trait to enable this pattern [35]. The issue with this solution is in the overhead introduced to maintain the stash of messages when the actor is in a *reply-blocked* state and the adding of messages back into the mailbox when the actor exits the reply-blocked state. There is additional overhead if the unstashed messages need to be prepended to the head of the mailbox.

Request-Response with Selectors With selectors, we can define two mailboxes one to receive *regular* messages and another one to receives *synchronous response* messages. Figure 7 has an example code snippet showing how selectors can be used with regular actors to handle the request-response pattern. The selector has two mailboxes: `REGULAR` used to receive normal messages and `REPLY` used to receive synchronous responses. Whenever a selector is expecting a synchronous response, it disables the regular mailbox (line 9) thereby ensuring that the next message it processes will be from its reply mailbox. When the selector disables the regular mailbox, we say the selector is in a *reply-blocked* state and is awaiting a response from the responding entity. When the message is processed by the responder, it replies by sending a message to the reply mailbox of the selector. The selector can then process this response message and enable the regular mailbox to move out of the reply-blocked state (line 14) and continue processing other messages sent to it.

```
1  class ReqRespSelector extends Selector {
2    def process(theMsg: AnyRef) {
3      theMsg match {
4        case m: SomeMessage =>
5          // a case where we want a response
6          val req = new SomeRequest(this, m)
7          anotherActor.send(req)
8          // move to reply-blocked state
9          disable(REGULAR)
10       case someReply: SomeReply =>
11         // process the reply (from REPLY mailbox)
12         ...
13         // resume processing regular messages
14         enable(REGULAR)
15     } } }
16 class ResponseActor extends Actor {
17   def process(theMsg: AnyRef) {
18     theMsg match {
19       case m: SomeRequest =>
20         val reply = compute(m.data)
21         // send to response mailbox
22         sender().send(REPLY, reply)
23       ...
24     } } }
```

Figure 7: Using Selectors to solve the Request-Response Pattern without blocking. In this example the responding entity is an actor and is sending the reply message to the `REPLY` mailbox (line 22).

3.1 Synchronous Reply Benchmarks

We created two synthetic benchmarks to measure the performance of selectors against other actor implementations for the synchronous request-response pattern. The first benchmark computes the Logistic Map [22, 38] using a recurrence relation $x_{n+1} = rx_n(1 - x_n)$. In the benchmark there are three 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. In the selector-based solution, the term actors are instead selectors and the ratio actors always send messages to the workers in their reply mailboxes. Figure 8 displays the results of this benchmark while comparing the selector-based solution against non-blocking actor-based solutions. The HS solutions is at least 1.6× faster than any of the actor solutions. Note that 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. Further,

the selector version was at least 12% smaller (177 lines compared to 202 lines) than all other actor versions.

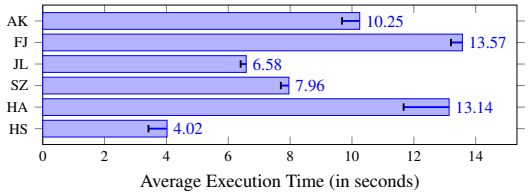

Figure 8: Results of the LogisticMap benchmark using 150 term actors/selectors and 150 ratio actors. Each term actor is responsible for computing 150000 terms. The AK version, which uses blocking ask pattern, runs in over 46 seconds. SC version is not shown on the graph. The SC version, which uses blocking `receive`, runs in over 300 seconds. The SC version, which uses (non-blocking) manual stashing, runs in over 100 seconds.

The second benchmark performs bank transactions between accounts by updating the balances atomically (i.e. the account doesn't process other messages while a transaction is in flight). The source account first decrements its balance and then needs to synchronously wait for the recipient actor to complete participating in the transaction by incrementing its balance with the same amount. During a single transaction, the source account needs to wait for the recipient actor to complete any active transactions it is participating in. The selector version was 10% smaller (153 lines) compared to other actor versions (170 lines in SC version, solutions of other actor variants required a few more lines). The performance results for the benchmark are displayed in Figure 9. SC, AK, JL, SZ and HA are all based on the same principle of stashing and unstashing individual messages in the mailbox. The HS solution uses the selector-based technique described in Figure 7 that relies on enabling and disabling mailboxes. In addition, the `REPLY` mailbox is looked up first to allow messages participating in an *active* transaction to be processed with a preference (see Section 5). This allows the HS version to be over 2× faster than the actor versions.

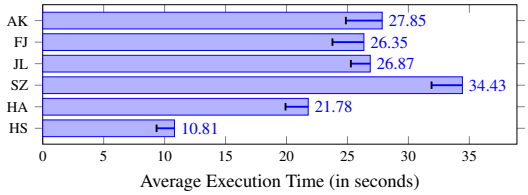

Figure 9: Results of the Bank Transaction benchmark using 1,000 bank accounts and 10 million transactions. The SC version, which uses thread-blocking `receive`, runs in over 200 seconds and is not displayed in the graph. The SC version, which uses manual stashing of messages runs in over 100 seconds.

4. Join Patterns in Streaming Applications

Since actors can be used to pipeline messages, they are a good fit for certain streaming applications. The pipeline parallelism can be exploited by connecting actors in a data flow chain to form producer-consumer pairs and ensuring FIFO order in processing of messages. In such a pipeline network, producers asynchronously propagate data to consumers as data becomes available. However, the join pattern [2, 31] where messages from two or more data streams are combined together into a single message is cumbersome to mimic using actors. The join is also commonly used in the *split-join pattern* where a parallel set of streams, which diverge from a common splitter and converge to a common joiner [33]. Traditional joins are blocking as they need to match inputs from each of the sources and wait until all corresponding inputs become available. An example of an aggregator is shown in Figure 10 where the adder actor is adding streams of corresponding values from three sources.

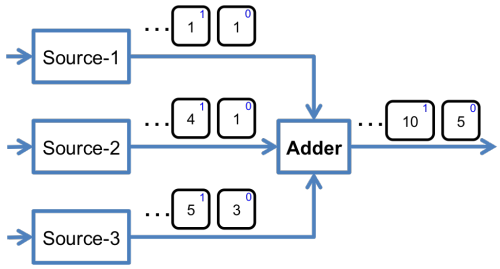

Figure 10: Actor network simulating a join pattern. `Source-1`, `Source-2`, and `Source-3` are producers for data streams. The `Adder` actor aggregates corresponding data items from each of the three sources and sums them up.

The lack of guarantee on which message is processed next, for example by taking the sender of the message into account, makes implementing this pattern troublesome. Complexity is also introduced due to the need for an additional dictionary to keep track of all items *in-flight* from the various sequence numbers. Supporting the join pattern also requires tagging messages with the source and sequence number information. This information is then used to aggregate the messages from the various sources in a dictionary indexed by the sequence number from each of the different sources. When items from all its sources for the oldest (lowest) sequence number is received, the aggregator actor can then reduce the items into a single value and forward it to the consumer in the network. The actor also needs to remove entries from the dictionary for sequences which have been aggregated to avoid memory leaks.

Akka provides support for the aggregator pattern [36]. The implementation allows `match` patterns to be dynamically added to and removed from an actor from inside the message handling logic. However, the implementation does not match the sender of the message which is required to support the join pattern. As such, enforcing the requirement that one message is received from each data stream source requires additional logic.

Join Pattern with Selectors Supporting the join pattern requires tagging the messages with source and sequence num-

bers in an actor-based solution. The requirement for sequence numbers goes away if messages from a sender are processed in the order in which they were sent as the aggregator can implicitly build the sequence numbers for each sender. However, requiring that corresponding items from each source are paired up still requires additional logic such as maintaining a dictionary of received items for each sequence. Figure 11 presents two selector-based solutions for the example problem introduced in Figure 10. The solutions require that the senders send their messages in the correct mailboxes, this can be enforced in one of two ways: *a*) wrapping the send logic in the selector to forward messages from sources to specific mailboxes, or *b*) configuring the sources with specific mailbox names during initialization so that source entities send only to specific mailboxes.

```
1  // process items in any order
2  class AdderAnyOrder(...) extends Selector {
3    var items = Array[Int](numSrcs)
4    var srcMatched = 0
5    def process(theMsg: AnyRef) {
6      theMsg match {
7        case im: ItemMessage =>
8          items(im.sourceId) = im.intValue()
9          // disable the current mailbox
10         disable(im.sourceId)
11         srcMatched += 1
12         if (srcMatched == numSrcs) {
13           val joinResult = computeJoin(items)
14           nextInChain.send(joinResult)
15           // reset locals
16           items = Array[Int](numSrcs); srcMatched = 0
17           // enable all mailboxes for next seq
18           enableAll()
19       } } } }

21 // process items in round-robin order
22 class AdderRoundRobinOrder(...) extends Selector{
23   var items = Array[Int](numSrcs)
24   var srcMatched = 0
25   // expect item from first source
26   disableAllExcept(0)
27   def process(theMsg: AnyRef) {
28     theMsg match {
29       case im: ItemMessage =>
30         items(im.sourceId) = im.intValue()
31         // disable the current mailbox
32         disable(im.sourceId)
33         srcMatched += 1
34         if (srcMatched == numSrcs) {
35           val joinResult = computeJoin(items)
36           nextInChain.send(joinResult)
37           // reset locals
38           items = Array[Int](numSrcs); srcMatched = 0
39         }
40         //enable round-robin mailbox for next seq
41         enable(srcMatched)
42     } } }
```

Figure 11: Using Selectors to solve the Join Pattern problem of Figure 10. The aggregator selector versions (`AdderSelectorAny` and `AdderSelectorRoundRobin`) maintain one mailbox for each source. For simplicity we assume sources are identified by consecutive integers starting at 0.

In the first solution, the `AdderAnyOrder` selector accepts an item (message) from any of the pending sources in the current sequence and does not enforce any order in the receipt of items from sources. As an item is received from a source, the corresponding mailbox is disabled (line 10) to disallow processing items from that source which do not belong to the current sequence. Hence, the set of active mailboxes shrinks and represents the set of sources which do not have a representative item in the current sequence. When items from all sources have been received for the current sequence (line 12), the result is computed (line 13) and forwarded to the next entity in the network (line 14) and all mailboxes are enabled (line 18) to start processing the next sequence.

In the second solution, the `AdderRoundRobinOrder` selector initially disables all the mailboxes except the first mailbox (lines 26) in anticipation of processing an item from the first source. As each item is received, the current mailbox is disabled (line 32) and the mailbox of the next source in round-robin order is enabled (line 41). When one message has been received from each of the sources in the current sequence (line 34), the result is computed (line 35) and forwarded to the next entity in the network (line 36). The currently active mailbox then is the mailbox corresponding to the first source so that processing of items from the next sequence begins.

4.1 Split-Join Benchmark (Filter Bank)

We use the Filter Bank benchmark ported from StreamIt [33] 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 or selectors. The Branches stage involves a split-join to combine the results of individual Bank stages. This join can be efficiently implemented using selectors as there is no need to maintain a dictionary to track each sequence arriving from the different banks. Figure 12 compares the performance of two selector implementations (HS-AnyOrder and HS-RoundRobin described in Figure 11) of the Filter Bank benchmark against actor implementations. The selector versions are at least 28% faster than the actor implementations in AK, SC, FJ, JL, SZ, and HA as overheads from maintaining the dictionary do not exist. The HSAO solution is 338 lines, HSRR is 343 lines while the actor-based solutions are at least 362 lines. The HS-AnyOrder (HSAO) version with selectors is slightly faster than the HS-RoundRobin (HSRR) version as expected with a lack of order guarantee. On benchmarks where there are more joining edges (Filter Bank has only 8), the unordered versions are expected to perform even better.

We implemented the Adder-Join microbenchmark (similar to Figure 10) where a single entity randomly populates messages into the different mailboxes of the `Adder` selector. This enables us to stress test the two implementations (HSRR and HSAO). The results are displayed in Figure 13. We see that the HSRR variant performs better than

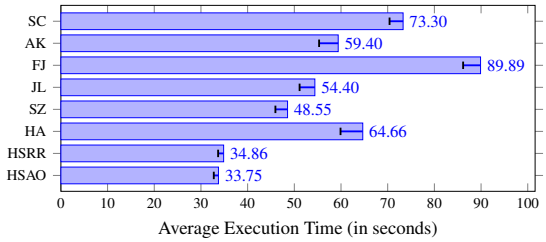

Figure 12: Results of the Filter Bank benchmark results configured to use 8-way join branches. The input used 300, 000 data items and 131, 072 columns.

the HSAO variant for 4-way and 8-way split-joins when using 24 million numbers to add. This is because enforcing the round-robin order simplifies finding the next message to process and helps avoid redundant search on other mailboxes. The performance is close between the two variants for for 12-way splits. The HSAO versions starts performing clearly better from 36-way splits as the round-robin evaluation order enforces an order which constrains the throughput. HSAO version benefits from processing messages for a given sequence out of order and hence earlier compared to the round-robin version.

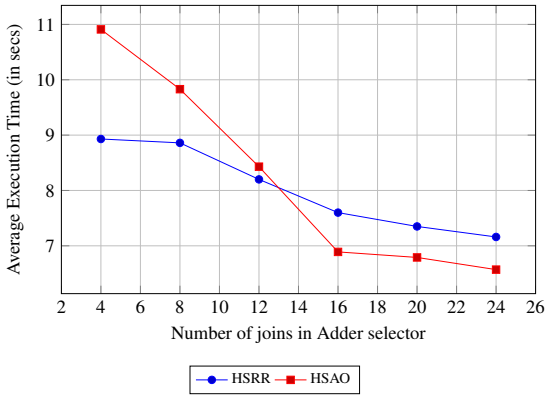

Figure 13: Results of the AdderJoin microbenchmark, configured to add 24 million numbers.

5. Supporting Priorities in Message Processing

In the priority pattern [23], messages with a higher priority are processed before those with a lower priority even if they were sent earlier. This pattern is useful in applications that offer different quality of service guarantees to individual clients where messages from *important* clients are given higher priorities. The priority pattern is also useful while implementing recursive data structure traversal algorithms where nodes deeper in the recursion tree are expected to produce results with higher probability. When solving divide-and-conquer style algorithms using *master-worker* pattern with actors, the master actor divides the work among the worker actors for load balancing and the worker actors report newly generated subproblems back to the master. The master should schedule larger subproblems for processing with higher priority as these subproblems should give rise to more work for the workers and improve the available parallelism. Normally actors do not support priorities while processing messages. Mimicking priorities in an actor based solution is quite cumbersome as messages sent to actors are stored in a single mailbox. One actor-based solution is to use a priority queue to store messages in the mailbox and require the user to provide some means of comparing the messages sent to the actor. However, this approach adds to overhead in the concurrent mailbox implementation.

Message Priorities in Selectors With selectors, we can support priorities for message processing non-intrusively without changing the message processing body (i.e. the `process()` method). The solution relies on the fact that selectors have multiple mailboxes, each mailbox is used to store messages of a given priority. This scheme requires that messages be categorized by priority. The categorization enables determining the target mailbox for the message in the `send` operation and can be enforced either by the sender or by the recipient selector. The mailboxes are then sorted by priority and the selector configured to always process an available message from the highest priority message. For example, consider a selector with `MAILBOX-0`, `MAILBOX-1`, `MAILBOX-2` and so on. At the end of each message processing cycle, the selector checks the mailboxes in their priority order for available messages to process. This ensures the highest priority message is processed before any lower priority ones. The solution allows supporting as many priority levels as deemed necessary by the problem in hand. Note that we do not rely on enabling or disabling particular mailboxes to support priorities.

5.1 Message Priority Benchmarks

We use variants of the NQueens and A* benchmarks to measure the performance of the message priority pattern. In the NQueens benchmark (Figure 14), the goal is to find the first 1.5 million solutions on a board of size 15×15. The divide-and-conquer style is used with master-worker style actors/selectors, the workers report solutions to the master. The master requests the workers to terminate as soon as the 1.5 million solutions are found. Subproblems deeper in the recursion tree are processed with higher priority by the workers. These priorities help in guiding the execution to processing the request which is likely to yield a solution earlier compared to random processing of messages. The AK solution that uses a priority queue comes closest to the HS solution, but the overhead from maintaining the concurrent priority queue limits performance. All the other actor solutions use regular mailboxes and do not support message priorities, they rely on the message order provided by their respective implementations. The HS solution is at least 1.7× faster than any of the actor-based solutions. Some of the performance gain also comes from being able to process the

worker termination signal earlier and avoiding performing redundant work once the required number of solutions have been found.

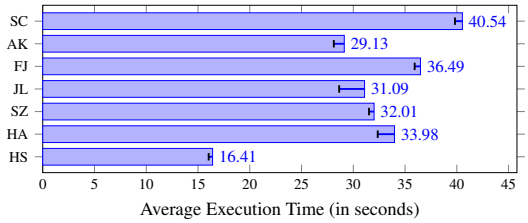

Figure 14: Messages with Priorities: **NQueens - Find 1.5 million solutions**. Board size 15. Sequential threshold 5. 24 workers.

In the A* benchmark (Figure 15), a randomly generated 3-D grid is searched for a path to a target node from an initial source node. The heuristic of euclidean distance to the goal node is used as the priority to promote processing of nodes closer to the goal node to find the solution. The JL solution performs best among the actor-based solutions (faster than the AK priority mailbox version). The HS solution is still about 13% faster than JL and over 17% faster than the other actor-based solutions. The main benefit in this case comes from attempting to process nodes closer to the target node and hence increasing the probability of finding the solution quicker.

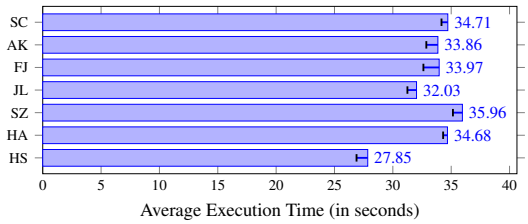

Figure 15: Messages with Priorities: **A* Search**. Grid size 150 × 150 × 150 with 24 workers.

6. Reader-Writer Concurrency

In the ReadersWriters problem [4, 41], there are multiple entities accessing a resource, some reading and some writing. In addition, there is the natural constraint that no entity may access the resource for reading or writing while another process is in the act of writing to it. There are two additional variants:

- The first readers-writers problem: the constraint is added that no reader request shall be kept waiting if the resource is currently opened for reading.
- The second readers-writers problem: the constraint is added that no writer request, once added to the message queue, shall be kept waiting longer than absolutely necessary.

Messages do not have dynamic priorities in actors; this makes it hard to support the variants where messages are processed out of *natural* order. In addition, since message processing is serialized in actors, implementation of truly concurrent data structures with actors is non-trivial. As an example, consider a group of actors of which one is a trivial dictionary actor (example from [28]) that responds to read and write requests. The other actors are clients of the dictionary: one actor does updates (writer), while the others only consult the dictionary (readers). The implementation of the dictionary with actors is easy because the programmer does not need to be concerned with data races: reads and writes to the dictionary are ensured to be executed in mutual exclusion. However, when the number of readers increases the resulting application performs badly precisely because of the benefits of serial execution (lack of data races) of requests to the dictionary actor: there are no means to process read-only messages in parallel and thus the dictionary actor becomes the bottleneck.

Reader-Writer Concurrency with Selectors We extend previous work on intra-actor parallelism [16] on actors to support asynchronous tasks with selectors. The `async` keyword is used to create a parallel task inside a selector's message processing body to process read requests in parallel. Figure 16 shows the template for a reader-writer selector. The `DataDrivenControl (DDC)` construct [19] is used to notify when all current in-flight read tasks have completed (inside the `decrementReaderAndSetDdc()` method). We maintain a **single** mailbox for both the write messages and the read messages (i.e. `WRITE` and `READ` refer to the same mailbox). Once a read message is processed, it bumps a counter that tracks the *in-flight* readers (in line 12) and spawns an asynchronous task to process the read message (in line 16). Once this task completes the counter is decremented to notify the selector that there is one less active reader task (in line 19). If there are no in-flight readers we can process a write request immediately (line 25). Otherwise, when a write message comes in, we disable all mailboxes thus preventing the selector from processing any subsequent messages (line 27). Note that disabling all the mailboxes is not an error as the write message is still being processed. We wait for the *in-flight* readers to complete and register a callback to trigger the asynchronous processing of the write message (line 29). Once the write message processing is completed, we can enable all the mailboxes (in line 32) and the selector continues processing subsequent messages.

The scheme presented solves the readers-writers problem where the messages are processed in the order they appear in the mailboxes. This scheme can further be fine-tuned to solve both the first readers-writers problem and the second readers-writers problem using the support available for priorities in message processing Section 5. No change is required to the `process()` method from Figure 16. For the two variants we need to maintain **two** mailboxes, one for write messages and one for read messages (i.e. `WRITE` and

```
class ReadWriteSelector extends Selector {
  def sendWrite(theMsg: AnyRef) {
    send(WRITE, new WriteMessage(theMsg));
  }
  def sendRead(theMsg: AnyRef) {
    send(READ, new ReadMessage(theMsg));
  }
  def process(theMsg: AnyRef) {
    theMsg match {
      case ReadMessage(messageBody) =>
        // another reader tasks is becoming active
        int activeReaders = incrementReader()
        if (activeReaders == 1) {
          readersDdcRef.set(new DDC())
        }
        async { // process read asynchronously
          processRead(messageBody)
          // notify that reader is no longer active
          decrementReaderAndSetDdc()
        } }
      case WriteMessage(messageBody) =>
        val readerDdc = readersDdcRef.get()
        if (readerDdc.valueAvailable()) {
          // no readers active, safe to process
          processWrite(messageBody)
        } else {
          disable(READ, WRITE) // disable mailboxes
          // wait for in-flight readers to complete
          readerDdc.addResumable(() => {
            processWrite(messageBody)
            // resume processing read/write messages
            enable(READ, WRITE)
          })
        } } }
```

Figure 16: Using Selectors to solve the Reader-Writer Concurrency problem. For simplicity concurrency constraints elided in the code snippet.

READ now refer to different mailboxes). Each write message then ends up disabling all the mailboxes before starting and enabling all the mailboxes after completing the processing of the write message. We only need to add support for priorities to these mailboxes. For the first readers-writers problem, the read mailbox gets higher priority than the write mailbox ensuring all available read messages are processed whenever they appear in the mailbox. For the second readers-writers problem, the write mailbox gets higher priority than the read mailbox ensuring writes are processed as soon as they appear in the mailbox (after *in-flight* readers have completed). As a result we can now support three variants of the reader-writers problem: *a*) arrival order (AO); *b*) first readers-writers (RF); and *c*) second readers-writers (WF).

6.1 Reader-Writer Benchmarks

We use a concurrent dictionary (CD) (Figure 17) and a concurrent sorted linked-list (CSLL) (Figure 18) data structure benchmarks to measure the performance of selectors versus actor-based solutions for reader-writer concurrency. We mix read (lookup by item for CD and lookup by element in CSLL) and write (key-value pair put for CD and element addition for CSLL) requests into the actor/selector representing the concurrent data structure. Pure actor solutions exhibit no concurrency, whereas the HS and HA solutions exhibit intra-actor concurrency. HS solutions allow concurrency from read requests based on AO and also support the RF and WF policies. The intra-selector parallelism in HS* solutions in general allow them to execute much faster than the actor-based solutions. The read and write operations in the CD benchmarks take $O(1)$ time while in the CSLL benchmark they take $O(N)$ time, this affects which of the RF and AO variants perform better. Note that creating and scheduling additional tasks for intra-actor and intra-selector parallelism involves some overhead. When read operations involve comparatively fewer operations and offer a fine granularity of work, we cannot entirely overcome the tasking overhead via computation. As a result, in the CD benchmark which does $O(1)$ work in read operations, the SZ version which has low mailbox contention for the dictionary actor (as seen in Section 2.1.3) can perform competitively with the HS* versions.

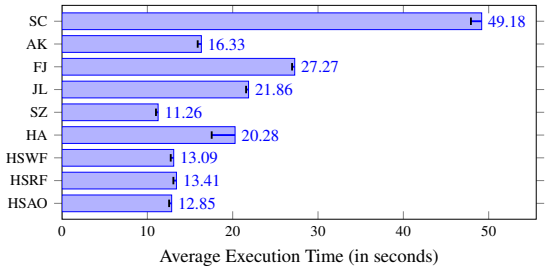

Figure 17: Reader/Writer Concurrency: **Dictionary**. 24 worker actors with a total 400K requests per worker. 10 percent of the requests are write requests.

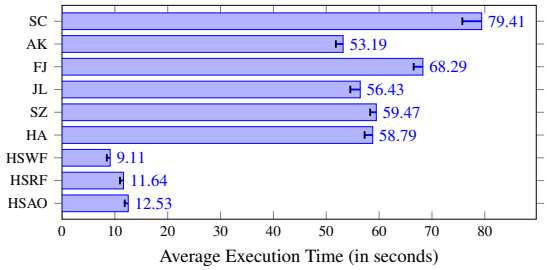

Figure 18: Reader/Writer Concurrency: **Sorted List**. 24 worker actors with a total 15K requests per worker. 10 percent of the requests are write requests.

7. Producer-Consumer Pattern

The producer-consumer with a bounded-buffer is a classic example of a multi-process synchronization problem [13, 40]. In this problem, producers push work into the buffer as it is produced while consumers pull work from the buffer when they are ready. In an actor-based solution to this problem, producers, consumers and the buffer are modeled as actors. The buffer actor acts like a manager and needs to keep track of at least the following scenarios: *a*) whether the data buffer is full or empty; *b*) when consumers request work from an empty buffer, the consumers are put in a queue until work is available; *c*) when producers are ready to produce data and the buffer is full, the producers are put in a queue until

space is available in the data buffer; *d*) notify producers to produce more work when space becomes available in the data buffer. The buffer actor needs to maintain additional queues for the available consumers and producers since there is no way to disable the processing of particular messages and this convolutes the logic of the buffer actor. A simpler scheme where messages from producers or consumers were avoided from being processed when the data buffer is full or empty, respectively, would greatly simplify the buffer actor implementation. Using the pattern matching option remains expensive to implement due to increase in time while searching for the next message to process from the mailbox.

Producer-Consumer with Selectors In a selector-based solution to this problem, we can maintain a selector for the buffer while the producers and consumers continue to be modeled as actors. The buffer selector maintains two mailboxes, one to receive messages from producers and another to receive messages from consumers. Then we can disable processing messages from consumers when the buffer becomes empty, and disable processing messages from producers when the buffer becomes full. Additionally, the buffer selector can provide declarative guards to `enable` or `disable` the mailboxes thus easing programming effort. The `guard` operation is used to register a boolean guard condition on a given mailbox. After each message is processed, the guard for each mailbox is evaluated and the mailbox is `enabled` or `disabled` depending on whether the guard evaluated to `true` or `false`, respectively.

```
1 class BufferSelector extends DeclarativeSelector {
2  def registerGuards() {
3   // disable producer msgs if buffer might overflow
4   guard(MBX_PRODUCER,
5    (theMsg) => dataBuffer.size() < thresholdSize)
6   // disable consumer msgs when buffer empty
7   guard(MBX_CONSUMER,
8    (theMsg) => !dataBuffer.isEmpty())
9  }
10 def doProcess(theMsg: AnyRef) {
11   theMsg match {
12     case dm: ProducerMsg =>
13       // store the data in the buffer
14       dataBuffer.add(dm)
15       // request producer to produce next data
16       dm.producer.send(ProduceDataMsg.ONLY)
17     case cm: ConsumerMsg =>
18       // send data item to consumer
19       cm.consumer.send(dataBuffer.poll())
20       tryExit()
21     case em: ProdExitMsg =>
22       numTerminatedProducers += 1
23       tryExit()
24   } } }
```

Figure 19: Using Selectors to solve solve the Producer-Consumer with Bounded-Buffer Pattern. The Buffer selector maintains two mailboxes, one to receive messages from producers and another to receive messages from consumers. The use of declarative guards separates the `enable` and `disable` logic of mailboxes into the guard registration method, `registerGuards`.

Figure 19 presents such a solution that uses declarative guards to isolate the message processing logic from the logic to enable or disable mailboxes. Whenever data is available in the buffer, the mailbox for messages from consumers is enabled (line 7) to process consume request messages. Whenever a data item is consumed by a consumer, the buffer is guaranteed space and the mailbox for messages from producers is enabled to process messages from producers (line 4) to fill the populate the buffer with data. This scheme avoids having to maintain separate collections of available producers and consumers in a purely actor-based solution and simplifies the logic of the buffer selector.

7.1 Producer Consumer with Bounded Buffer Benchmark

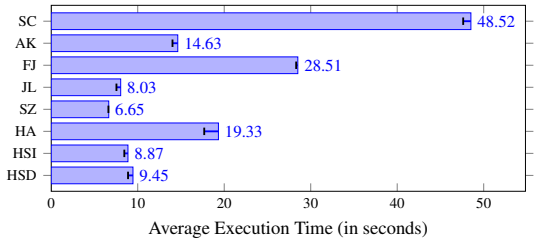

Figure 20: Results of the Bounded Buffer benchmark on bounded buffer size of 6000. There were 5000 producer actors each producing up to 1000 messages. There were 2000 consumer actors.

In the actor-based solution to the producer-consumer with bounded-buffer, the buffer actor is like the mall actor in the Chameneos benchmark (Section 2.1.3), so this benchmark also includes measurement of the mailbox contention overhead. The actor-based solutions have the additional overhead from maintaining additional data structures (adding and removing elements in collections) for the available producers and consumers in the buffer actor. In the selector-based solution, we avoid the need for additional collections by relying on enabling or disabling mailboxes. There are two selector versions, one which uses the declarative guard style (HSD) and another which uses an imperative style (HSI) with manual `enable` and `disable` calls. The `process` method of the buffer selector in the HSI version is about 22% smaller (25 lines) compared to the `process` method of the buffer actor in the actor versions (32 lines). The `process` and `registerGuard` HSD version (22 lines) is further smaller than the HSI version. As seen in Figure 20, the HS* versions now performs similar to the JL and SZ versions (despite having overheads in mailbox contention). Importantly, the HS* performs much better than HA (in the Chameneos benchmark they had similar performance). The HSD version performs about 6.5% slower than the HSI version, this measures the cost of the declarative abstraction where all mailboxes need to be enabled or disabled after each message is processed. As in the Chameneos benchmark (Section 2.1.3), FJ and SC perform much slower compared to the other actor implementations.

8. Related Work

Tomlinson and Singh propose an extension to the AM to control the order in which messages are processed so as to preserve the integrity of objects [34]. They introduce the concept of enabled-sets that define the messages that may be executed in the new state. This idea is similar to the implementation of Scala actors by Haller [8] where the pattern matching clause on messages can be modified dynamically. As mentioned in Section 3, this is expensive to implement due to increased time in searching for the next message to process from the mailbox. In contrast, our approach allows enabling or disabling one of possibly many mailboxes, to control which messages are processed next. It is also efficient to implement as active mailboxes with messages can be found quickly.

Akka provides support for the aggregator pattern [36], however, their implementation does not match the sender of the message which is required to support the join pattern. Ensuring matching of senders requires extra tagging of messages as in a pure actor solution. On the other hand, with selectors, we can elegantly enforce the requirement that one message is received from each data stream source participating in the join (Section 4).

Sulzmann et al. [30], Plociniczak et al. [26] and Haller et al [9] have proposed an extension to actors with `receive` clauses containing multiple message patterns based on Join calculus (not necessarily the same as the split-join pattern). This allows expression of join patterns by specifying an exhaustive list of messages that participate in the join, requiring that messages contain their source actor information, and including guards to restrict pattern matching. This offers a high-level way of synchronizing messages for processing at the cost of increased processing time (compared to our selector extension) in resolving a match in the `receive` clause for a message. To support other general Join calculus patterns, our extension may require the user to explicitly manage the state from the partial matches as each message from the active mailboxes are consumed.

The SALSA actor-oriented language [37] has supported two-level priority message sends since version 0.6.2. In SALSA, regular messages are placed at the end of the mailbox queue. Priority messages are placed at the front of the actor's mailbox, instead of the end. Nystrom presents a solution for supporting priority processing of messages in Erlang using nested non-blocking `receive` operations [24]. The nesting code structure convolves the pattern-matching code for supporting messages with multiple levels of priorities. In their technique, since priorities are based on pattern-matching of individual messages, their solution is inefficient if mailboxes contain many pending messages. As stated in [24], the performance penalty of their approach keeps increasing as they support additional priority levels. Sulzmann et al. [30] also support priorities in their model with similar syntax. They concede that enforcing priorities manually via `otherwise` and nested `receive` statements leads to clumsy code. In contrast to these approaches, our extension prioritizes messages by the mailbox in which they reside and high-priority messages are selected using constant-time operations. The message processing logic in our solution is unaffected by the priority scheme.

Scholliers et al. have proposed the notion of Parallel Actor Monitors (PAM) [28] to extend the AM with support for intra-actor parallelization. PAM can easily solve the symmetric reader-writer problem where messages are processed in the arrival order they appear in the mailbox, but it does not support priorities of messages. De Koster et al. propose the TANK model [5] where actors expose part of their state as a shared read-only resource for other tanks (containers for related actors). The model relies on an approach similar to transactional memory that ensures an actor will always observe a consistent version of a read-only state, even in the face of concurrent updates of the actor that owns that state. In particular, it enables reader-writer concurrency where any number of reader methods can be executing in parallel with at most one writer method thus solving the first reader-writer variant. Our support for priorities enables us to solve the three variants to the reader-writer problem as discussed in Section 6.

9. Summary

We have introduced our extension to the AM called selectors. Selectors have multiple mailboxes and each mailbox is guarded i.e. it can be enabled or disabled to affect the order in which messages are processed. As opposed to using actors, selectors allow us to simplify writing of multiple synchronization and coordination patterns including: *a*) synchronous request-reply; *b*) join patterns in streaming applications; *c*) supporting priorities in message processing; *d*) variants of reader-writer concurrency; and *e*) producer-consumer with bounded buffer. We provided descriptions for solutions to some of these patterns using selectors and also included our results for benchmarks exhibiting these patterns. Our results confirm that selector-based solutions for benchmarks exhibiting these patterns are simpler and execute much faster than actor-based solutions. In the future, we plan explore additional computational patterns that can be expressed easily using selectors. For example, nondeterministic finite automata can be expressed easily using selectors where each input symbol is sent to a different mailbox. This should facilitate implementing multiple message patterns based on Join calculus using selectors.

Availability

Public distributions of the selectors implementation in Habanero-Java library, including documentation and code examples, are available for download at `http://wiki.rice.edu/confluence/display/PARPROG/HJ+Library`. We also plan to make source code Scala versions of the selectors

benchmarks available as part of the Savina Benchmark Suite [18].

Acknowledgments

This material is based upon work supported by the National Science Foundation under Grant No. CCF–0964520. Any opinions, findings and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect those of the National Science Foundation. We thank Karthik Murthy and Alina Sbîrlea for their valuable feedback on early drafts of this paper. We are very grateful to the anonymous reviewers for their insightful comments and suggestions to clarify the contents of the paper. The paper has been substantially improved based on these suggestions.

References

[1] G. Agha. *Actors: a model of concurrent computation in distributed systems*. MIT Press, Cambridge, USA, 1986. ISBN 0-262-01092-5.

[2] Apache Software Foundation. Apache Camel: Aggregator, 2004. URL https://camel.apache.org/aggregator2.html.

[3] J. Armstrong. Concurrency Oriented Programming in Erlang. *Challenge*, November 2002.

[4] P. J. Courtois, F. Heymans, and D. L. Parnas. Concurrent Control with "Readers" and "Writers". *Commun. ACM*, 14(10):667–668, October 1971. ISSN 0001-0782.

[5] J. De Koster, S. Marr, T. D'Hondt, and T. Van Cutsem. Tanks: Multiple Reader, Single Writer Actors. In *Proceedings of the 2013 Workshop on Programming Based on Actors, Agents, and Decentralized Control*, AGERE! '13, pages 61–68, NY, USA, 2013. ACM. ISBN 978-1-4503-2602-5.

[6] EPCC. The Java Grande Forum Multi-threaded Benchmarks, 2001. URL http://www2.epcc.ed.ac.uk/computing/research_activities/java_grande/threads/s1contents.html.

[7] functionaljava.org. functionaljava: A Library for Functional Programming in Java, 2010. URL https://code.google.com/p/functionaljava/.

[8] P. Haller and M. Odersky. Scala Actors: Unifying thread-based and event-based programming. *Theoretical Computer Science*, 410(2–3):202–220, 2009. ISSN 0304–3975. Distributed Computing Techniques.

[9] P. Haller and T. Van Cutsem. Implementing Joins Using Extensible Pattern Matching. In *Proceedings of the 10th International Conference on Coordination Models and Languages*, COORDINATION'08, pages 135–152, Berlin, Heidelberg, 2008. Springer-Verlag. ISBN 3-540-68264-3, 978-3-540-68264-6.

[10] Haller, Philipp. chameneos-redux.scala — FishEye: browsing scala-svn, 2011. URL https://codereview.scala-lang.org/fisheye/browse/scala-svn/scala/branches/translucent/docs/examples/actors/chameneos-redux.scala?hb=true.

[11] C. Hewitt, P. Bishop, and R. Steiger. Artificial Intelligence A Universal Modular ACTOR Formalism for Artificial Intelligence. Proceedings of the 3rd International Joint Conference on Artificial Intelligence, Stanford, CA, August 1973.

[12] Hewitt, Carl and Baker, Henry G. Actors and Continuous Functionals. Technical report, Massachusetts Institute of Technology, Cambridge, MA, USA, February 1978.

[13] C. A. R. Hoare. Monitors: An Operating System Structuring Concept. *Commun. ACM*, 17(10):549–557, Oct. 1974. ISSN 0001-0782.

[14] G. Hohpe and B. Woolf. Enterprise Integration Patterns - Request-Reply, 2003. URL http://www.eaipatterns.com/RequestReply.html. [Online; accessed 3-April-2014].

[15] L. Hupel and typelevel.org. scalaz: Functional programming for Scala, 2010. URL http://typelevel.org/projects/scalaz/.

[16] S. Imam and V. Sarkar. Integrating Task Parallelism with Actors. In *Proceedings of the ACM international conference on Object Oriented Programming Systems Languages and Applications*, OOPSLA '12, pages 753–772, NY, USA, 2012. ACM. ISBN 978-1-4503-1561-6.

[17] 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*, PPPJ'14. ACM, NY, USA, 2014.

[18] S. Imam and V. Sarkar. Savina - An Actor Benchmark Suite. In *4th International Workshop on Programming based on Actors, Agents, and Decentralized Control*, AGERE! 2014, October 2014. Position paper.

[19] Imam, Shams. DataDrivenControl, 2014. URL http://www.cs.rice.edu/~vs3/hjlib/doc/edu/rice/hj/runtime/forkjoin/DataDrivenControl.html.

[20] Imam, Shams and Sarkar, Vivek. Habanero-Scala: Async-Finish Programming in Scala. In *Scala Days 2012*, April 2012.

[21] I. A. Mason and C. L. Talcott. Actor Languages. Their Syntax, Semantics, Translation, and Equivalence. *Theor. Comput. Sci.*, 220(2):409–467, June 1999. ISSN 0304-3975.

[22] R. M. May. Simple mathematical models with very complicated dynamics. *Nature*, 261(5560):459–467, June 1976.

[23] Microsoft Developer Network. Priority Queue Pattern, 2014. URL http://msdn.microsoft.com/en-us/library/dn589794.aspx. [Online; accessed 3-April-2014].

[24] J. H. Nystrom. Priority Messaging Made Easy. In *Proceedings of the 2007 SIGPLAN Workshop on ERLANG Workshop*, ERLANG '07, pages 65–72, NY, USA, 2007. ACM. ISBN 978-1-59593-675-2.

[25] Oracle. Understanding Interaction Patterns, 2011. URL http://docs.oracle.com/cd/E17904_01/doc.1111/e17363/chapter05.htm.

[26] H. Plociniczak and S. Eisenbach. JErlang: Erlang with Joins. In D. Clarke and G. Agha, editors, *Proceedings of the 12th International Conference on Coordination Models and Languages*, volume 6116 of *COORDINATION'10*, pages 61–75. Springer Berlin Heidelberg, 2010. ISBN 978-3-642-13413-5. .

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

[28] C. Scholliers, ı. Tanter, and W. De Meuter. Parallel Actor Monitors: Disentangling Task-level Parallelism from Data Partitioning in the Actor Model. *Sci. Comput. Program.*, 80:52–64, Feb. 2014.

[29] A. Silberschatz, P. B. Galvin, and G. Gagne. *Operating System Concepts*. Wiley Publishing, 8th edition, 2008. ISBN 0470128720.

[30] 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 *COORDINATION'08*. Springer, 2008. ISBN 978-3-540-68264-6.

[31] Sybase Inc. Complex Event Processing: Ten Design Patterns. White paper, Sybase - An SAP Company, Apr 2001. URL http://m.sybase.com/files/White_Papers/CEP-10-Design-Patterns-WP.pdf.

[32] S. Tasharofi, P. Dinges, and R. E. Johnson. Why Do Scala Developers Mix the Actor Model with Other Concurrency Models? In *Proceedings of the 27th European Conference on Object-Oriented Programming*, ECOOP'13, pages 302–326, Berlin, Heidelberg, 2013. Springer-Verlag. ISBN 978-3-642-39037-1.

[33] 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 Parallel Architectures and Compilation Techniques*, PACT '10, pages 365–376, NY, USA, 2010. ACM. ISBN 978-1-4503-0178-7.

[34] C. Tomlinson and V. Singh. Inheritance and Synchronization with Enabled-Sets. In *Conference Proceedings on Object-oriented Programming Systems, Languages and Applications*, OOPSLA '89, pages 103–112, NY, USA, 1989. ACM. ISBN 0-89791-333-7.

[35] Typesafe Inc. Actors - Akka Documentation, 2014. URL http://doc.akka.io/docs/akka/2.3.2/scala/actors.html.

[36] Typesafe Inc. Aggregator Pattern - Akka Documentation, 2014. URL http://doc.akka.io/docs/akka/snapshot/contrib/aggregator.html.

[37] C. Varela and G. Agha. Programming Dynamically Reconfigurable Open Systems with SALSA. *ACM SIGPLAN Notices*, 36(12):20–34, Dec. 2001. ISSN 0362-1340.

[38] Wikipedia, The Free Encyclopedia. Logistic map, 2014. URL http://en.wikipedia.org/wiki/Logistic_map.

[39] Wikipedia, The Free Encyclopedia. Multilevel queue, 2014. URL http://en.wikipedia.org/wiki/Multilevel_queue. [Online; accessed 28-September-2014].

[40] Wikipedia, The Free Encyclopedia. ProducerConsumer Problem, 2014. URL http://en.wikipedia.org/wiki/Producer%E2%80%93consumer_problem. [Online; accessed 3-April-2014].

[41] Wikipedia, The Free Encyclopedia. ReadersWriters Problem, 2014. URL http://en.wikipedia.org/wiki/Readers%E2%80%93writers_problem. [Online; accessed 3-April-2014].

[42] Wikipedia, The Free Encyclopedia. Round-robin scheduling, 2014. URL http://en.wikipedia.org/wiki/Round-robin_scheduling. [Online; accessed 28-September-2014].