Empirical Study of Actor-based Runtime Verification

Nastaran Shafiei
SGT Inc./Ames Research Center, Moffett Field, California

Klaus Havelund
Jet Propulsion Laboratory, Pasadena, California

Peter Mehlitz
SGT Inc./Ames Research Center, Moffett Field, California

June 2020
Since its founding, NASA has been dedicated to the advancement of aeronautics and space science. The NASA scientific and technical information (STI) program plays a key part in helping NASA maintain this important role.

The NASA STI Program operates under the auspices of the Agency Chief Information Officer. It collects, organizes, provides for archiving, and disseminates NASA's STI. The NASA STI Program provides access to the NASA Aeronautics and Space Database and its public interface, the NASA Technical Report Server, thus providing one of the largest collection of aeronautical and space science STI in the world. Results are published in both non-NASA channels and by NASA in the NASA STI Report Series, which includes the following report types:

- **TECHNICAL PUBLICATION.** Reports of completed research or a major significant phase of research that present the results of NASA programs and include extensive data or theoretical analysis. Includes compilations of significant scientific and technical data and information deemed to be of continuing reference value. NASA counterpart of peer-reviewed formal professional papers, but having less stringent limitations on manuscript length and extent of graphic presentations.

- **TECHNICAL MEMORANDUM.** Scientific and technical findings that are preliminary or of specialized interest, e.g., quick release reports, working papers, and bibliographies that contain minimal annotation. Does not contain extensive analysis.

- **CONTRACTOR REPORT.** Scientific and technical findings by NASA-sponsored contractors and grantees.

- **CONFERENCE PUBLICATION.** Collected papers from scientific and technical conferences, symposia, seminars, or other meetings sponsored or co-sponsored by NASA.

- **SPECIAL PUBLICATION.** Scientific, technical, or historical information from NASA programs, projects, and missions, often concerned with subjects having substantial public interest.

- **TECHNICAL TRANSLATION.** English-language translations of foreign scientific and technical material pertinent to NASA's mission.

Specialized services also include organizing and publishing research results, distributing specialized research announcements and feeds, providing information desk and personal search support, and enabling data exchange services.

For more information about the NASA STI Program, see the following:

- Access the NASA STI program home page at [http://www.sti.nasa.gov](http://www.sti.nasa.gov)
- E-mail your question to help@sti.nasa.gov
- Phone the NASA STI Information Desk at 757-864-9658
- Write to:
  NASA STI Information Desk
  Mail Stop 148
  NASA Langley Research Center
  Hampton, VA 23681-2199
Empirical Study of Actor-based Runtime Verification

Nastaran Shafiei
SGT Inc./Ames Research Center, Moffett Field, California

Klaus Havelund
Jet Propulsion Laboratory, Pasadena, California

Peter Mehlitz
SGT Inc./Ames Research Center, Moffett Field, California

National Aeronautics and Space Administration
Ames Research Center
Moffett Field, California

June 2020
Abstract

In this work, we study how the use of concurrent monitors in a runtime verification system can improve performance as well as applying indexing. We specifically target concurrent monitoring systems that adopt the actor programming model which is just one way of developing concurrent systems. We present our runtime verification framework MESA (MEssage-based System Analysis) which allows for building actor-based monitoring systems. We perform empirical study by conducting experiments using monitoring systems with different numbers of monitor actors and different layers of indexing. This report describes our experiments, evaluates our results, and discusses challenges faced during the study.

1 Property of Interest

For the experiment, we used the SWIM Flight Data Publication Service (SFDPS) as the system under observation. SWIM is a highly distributed system that consolidates data from many sources. We have observed incidents where the SFDPS messages are not distributed in the right order by the SWIM server. We choose sfdps-seq-order as the property of interest for this experiment which checks for the chronological order of the SFDPS messages, that is, it verifies that the SFDPS messages with the same call sign (flight identifier) are ordered by their time tags. This property is specified as a data-parameterized finite state machine using the Daut DSL [1, 2]. The class sfdpsOrderMonitor which is presented in Figure 1 implements sfdps-seq-order. The daut.Monitor is the key class in Daut that implements the monitor capabilities.

Daut also provides an indexing capability within monitors to improve their performance. It allows for defining a function from events to keys where keys are used as entries in a hash map (of type scala.collection.immutable.HashMap) to obtain those states which are relevant to the event. Using this approach, a Daut monitor only iterates over an indexed subset of states instead of the entire set of current states. To use Daut indexing in a monitor, one needs to override the method keyOf of daut.Monitor. Some of the actor-based systems used in the experiment employ Daut indexing which is implemented by the class sfdpsOrderMonitor_indx. As shown in Figure 2, sfdpsOrderMonitor_indx overrides keyOf with an implementation that given an event, it returns the call sign associated to the event. Note that FlightState and TrackCompleted are the two possible events in the input trace, and the parameter cs represents the flight call sign.

The sfdps-seq-order property is simple and it leads to a small service time, which is the time used to process the message within the monitor objects. To mitigate issues associated to microbenchmarking on JVM, we use a feature of Daut that allows for defining sub-monitors within a Daut monitor object. We define the class sfdpsOrderMultiMonitor (Figure 1) which is a Daut monitor and maintains a list of sfdpsOrderMonitor instances as its sub-monitors. The size of the list is provided by user in the configuration file using the key sub-monitor-count. Similarly, the class sfdpsOrderMultiMonitor_indx (Figure 2) is defined which maintains a
list of sfdpsOrderMonitor_indx instances as its sub-monitors.

trait MesaMonitor {
  def verifyEvent(event: Any): Unit
}

class DautMonitor(val config: Config) extends daut.Monitor[Any] with MesaMonitor {
  def verifyEvent(event: Any) : Unit = {
    verify (event)
  }
}

class sfdpsOrderMonitor(config: Config) extends DautMonitor(config) {
  always {
    case FlightState(_, cs, _, _, _, _, date1, _) =>
      watch {
        case FlightState(_, 'cs', _, _, _, _, date2, _) =>
          date2.isAfter (date1)
      }
  }
}

class sfdpsOrderMultiMonitor(config: Config) extends DautMonitor(config) {
  val size = config.getInt("sub-monitor-count") – 1
  val m = for (i <- 0 to size) yield new sfdpsOrderMonitor(config)
  monitor(m: _*)
}

Figure 1: Daut monitor capturing the property sfdps-seq-order.

class sfdpsOrderMonitor_index(config: Config) extends sfdpsOrderMonitor(config) {
  override protected def keyOf(event : Any): Option[String] = {
    event match {
      case FlightState(_, cs, _, _, _, _, _) => Some(cs)
      case TrackCompleted(_, cs, _, _, _, _, _) => Some(cs)
    }
  }
}

class sfdpsOrderMultiMonitor_index(config: Config) extends DautMonitor(config) {
  val size = config.getInt("sub-monitor-count") – 1
  val m = for (i <- 0 to size) yield new sfdpsOrderMonitor_index(config)
  monitor(m: _*)
}

Figure 2: Daut monitor capturing the property sfdps-seq-order with indexing.
2 Monitoring Systems

This section presents the actor-based monitoring systems used in our experiment. These systems are illustrated in Figure 3. The outermost white boxes represent actors and the vertical lines between them represent the publish/subscribe communication channels. It can be seen that all the systems have the same data acquisition and data processing phases, and they are only different in their monitoring phase. They use the *instant-reply* actor to acquire the input data and the *event-gen* actor to process the data. The *instant-reply* actor accesses an archive containing recorded AFDPS data messages in the XML format, and as it reads the messages, it publishes them to the *sfdps* channel instantly. The *event-gen* actor obtains the XML messages by subscribing to the channel *sfdps*, turns each message into an event of type *FlightState* or *TrackCompleted*, and publishes the event object to the channel *events*. *FlightState* captures the state of the flight, and *TrackCompleted* indicates that the flight has reached its final destination.

MESA provides components referred to as dispatchers which are configurable and used in the monitoring phase of the runtime verification to determine how the check for the property of interest is distributed among different *monitor actors*. The actors that contain one or more monitor instances are referred to as monitor actors. The MESA dispatchers which are implemented as actors are not tied to a specific application and they can be extended towards different systems under observation. As explained below, these components are key to our experiments.

Let \( n \), in Figure 3, be the total number of different call signs in the input sequence of SFDPS messages. The gray boxes inside the actors represent monitor instances, where \( M \) refers to instances of type *sfdpsOrderMultiMonitor* (Figure 1) and \( MI \) refers to instances of type *sfdpsOrderMultiMonitor_indx* (Figure 2). The white box contained inside each monitor instance contains those call signs monitored by this instance. Next, we explain the monitoring phase for each actor-based monitoring system in more details.

- **monitor-indexing** - the monitoring phase includes one actor with a single monitor object of type *sfdpsOrderMultiMonitor_indx* that checks for all the events in the input sequence (see Figure 3a). The monitor actor obtains event objects from the channel *events* and feeds them to the underlying monitor object by invoking its *verify* method. The monitor instance employs indexing within the monitor. In a way, the monitoring phase of this configuration is equivalent to directly using the Daut tool which processes the trace sequentially.

- **dispatcher-indexing** - the monitoring phase includes a dispatcher actor (of type *gov.nasa.mesa.dispatchers.IndexingDispatcher*) which creates monitor instances of type *sfdpsOrderMultiMonitor* on-the-fly, and feeds them with incoming events. The dispatcher generates one monitor instance per call sign (see Figure 3b). It stores the monitor instances in a hash map (of type *scala.collection.mutable.HashMap*), using call signs as entries to the hash map. Thus, in this configuration indexing is applied inside the dispatcher actor, and not inside the monitor instances. The dispatcher obtains
event objects from the channel \( \texttt{events} \), and starting with an empty hash map, for each new call sign, it adds a new monitor instance to the hash map. Moreover, for an event object with the call sign \( \texttt{cs}_i \), the monitor actor invokes the \texttt{verify} method of the monitor instance \( \texttt{M}_i \).

- **concurrent** - the trace analysis is performed concurrently by employing one monitor actor per call sign (see Figure 3c). Each monitor actor includes a monitor instance of type \texttt{sfdps}\texttt{OrderMultiMonitor}. The runtime verification phase includes a dispatcher actor (of type \texttt{gov.nasa.mesa.dispatchers.ConcurrentDispatcher}) that obtains the event objects from the channel \( \texttt{events} \), and for each new call sign, it generates a monitor actor on-the-fly. The dispatcher actor uses indexing by storing the references of the monitor actors in a hash map (of type \texttt{scala.collection.mutable.HashMap}) using call signs as entries to the hash map. For an event object with the call sign
the dispatcher then forwards the event object to the associated actor, monitor-actor-i, using the point-to-point communication paradigm. Then the monitor actor invokes the verify method on its underlying monitor instance. For an event object of type of TrackCompleted which indicates the end of the flight, the dispatcher also sends a termination request to the monitor actor.

- **bounded-concurrent** - the monitoring phase is similar to the monitoring phase in the concurrent configuration, except there is a limit on the number of monitor actors generated. Therefore, one monitor actor could be associated to more than one call sign. Moreover, monitor actors contains instances of type sfdpsOrderMultiMonitor_indx which employs indexing inside the monitor instance. This configuration also uses the dispatcher actor of type gov.nasa.mesa.dispatchers.ConcurrentDispatcher with the key actor-monitor-count set to the maximum number of monitor actors in the configuration file.

The main features of these monitoring systems which are key to the experiment are summarized in Table 4

<table>
<thead>
<tr>
<th></th>
<th>monitor-indexing</th>
<th>dispatcher-indexing</th>
<th>concurrency</th>
<th>bounded</th>
</tr>
</thead>
<tbody>
<tr>
<td>monitor-indexing</td>
<td>✓</td>
<td>×</td>
<td>×</td>
<td>×</td>
</tr>
<tr>
<td>dispatcher-indexing</td>
<td>×</td>
<td>✓</td>
<td>×</td>
<td>×</td>
</tr>
<tr>
<td>concurrent</td>
<td>×</td>
<td>✓</td>
<td>✓</td>
<td>×</td>
</tr>
<tr>
<td>bounded-concurrent</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
</tbody>
</table>

Figure 4: The main features of the monitoring systems presented in Figure 3.

### 2.1 Configurations

In order to build the monitoring systems presented in Section , one needs to specify the actors and the way they are connected in configuration files in the HOCON format [3] which is a JASON dialect. MESA receives the configuration file as an input and generates the actor-based system accordingly. As an example, see Figure 5 which shows the input configuration file used to build the bounded-concurrent monitoring system. You can find the one-to-one correspondence between the actors specified within the actors block in the configuration file and the actors in Figure 3d. The dispatcher type SfdpsConcurrentDispatcher extends the type ConcurrentDispatcher towards SFDPS messages.
actors = [
    {
        name = "instant−replay",
        class = "gov.nasa.mesa.dataAcquisition.InstantReplayActor",
        write−to = "sfdps",
        reader = {
            class = "gov.nasa.mesa.dataAcquisition.TextArchiveReader",
            pathname = "/sfdps.xml.gz",
            buffer−size = 32768
        },
    },
    {
        name = "event−gen",
        class = "gov.nasa.race.actor.TranslatorActor",
        read−from = "sfdps",
        write−to = "events",
        translator = {
            class = "gov.nasa.race.air.translator.FIXM2FlightObject"
        },
    },
    {
        name = "bounded−concurrent−dispatcher",
        class = "gov.nasa.mesa.dispatchers.nextgen.SfdpsConcurrentDispatcher",
        read−from = "events",
        actor−monitor−count = 250,
        monitor = {
            name = "sfdps−order−monitor",
            monitor−actor−class = "gov.nasa.mesa.core.MonitorActor",
            monitor−object.class = "gov.nasa.mesa.nextgen.monitors.sfdpsOrderMultiMonitor.inx",
            sub−monitor−count = 2000
        }
    }
]

Figure 5: The configuration file specifying the concurrent−bounded actor-based monitoring system.

3 Performance Measurement of Actors

In the experiment, first, we measure the overall performance of MESA using different settings. To investigate the underlying factors behind run time results further, we also measure different performance parameters for individual dispatcher and monitor actors in the monitoring system. This section explains how these parameters are measured.

The performance parameters that we consider are the average service time, the average wait time for messages in the actor’s mailbox, and the average size of actor’s mailbox queue obtained after enqueuing a message. Figure 6 illustrates the relevant points at which we record data to measure the actors performance met-
The interesting points are when a message is enqueued into and dequeued from the mailbox queue, and when the actor starts processing and finishes processing a message. The relevant data recorded at these points are stored in data containers (See Figure 7) which are defined as case classes and published to the Akka publish-subscribe EventStream channel. We also implement an actor called stat-collector which is subscribed to EventStream, and collects MailboxStats and MsgProcessingStats data containers and processes them into statistics. First, we explain how the actors performance parameters are measured in MESA.

Figure 7: Data containers used to store data processed by the stats-analyzer actor.

### 3.0.1 Measuring Service Time

Extended from RACE actors, the default behavior of MESA actors is defined by the method receiveLive which is of type of PartialFunction[Any, Unit] and processes the incoming messages. To measure the service time, we implement the type InstrumentedReceive, shown in Figure 8. InstrumentedReceive extends PartialFunction[Any, Unit], and has constructor parameters of type of PartialFunction[Any, Unit] (called delegatee) and MesaActor. InstrumentedReceive overrides the apply method with an implementation that invokes delegatee on the incoming message and records the time before and after invoking delegatee. It then creates a data container of type MsgProcessingStats, and publishes it to the EventStream channel (by invoking...
forwardStats on the given MesaActor). MsgProcessingStats objects which are collected by stat-collector include the identity of the actor processing the message and the sender actor, and the time before and after processing the message.

class InstrumentedReceive (val delegatee: PartialFunction[Any,Unit], val actor: MesaActor)
extends PartialFunction[Any,Unit] {
  override def apply(msg: Any): Unit = {
    if (msg.isInstanceOf[RaceSystemMessage]) delegatee(msg)
    else {
      val start = System.currentTimeMillis
      delegatee(msg)
      val end = System.currentTimeMillis
      val stats = MsgProcessingStats(actor.self, start, end)
      actor.forwardStats(stats)
    }
  }

  def isDefinedAt(x: Any): Boolean = delegatee.isDefinedAt(x)
}

Figure 8: InstrumentedReceive replaces the receiveLive function in MESA actors to measure the service time.

Every MESA actor has the boolean property stats.service which is false by default. If set to true in the configuration file, when initializing the actor, MESA replaces the receiveLive with a new instance of InstrumentedReceive using the actor's receiveLive object and reference as its constructor parameters.

3.0.2 Measuring Mailbox Metrics

To obtain information from actors mailboxes, we implement a new mailbox that extends the default mailbox implementation in Akka. Akka allows for specifying the actor's mailbox in the configuration file. If no mailbox is specified for an actor, the default mailbox is used. Creating a new Akka mailbox requires a queue implementation of type MessageQueue, and a factory class of type MailboxType to instantiate the queue for corresponding actors. My mailbox implementation includes the class StatsQueue that extends MessageQueue and implements the mailbox queue. It also includes the mailbox factory class StatsMailboxType that extends MailboxType, and it is used to instantiate StatsQueue.

StatsQueue has a final field queue of type java.util.concurrent.ConcurrentLinkedQueue which stores the messages that are waiting in the mailbox. The two key methods which are declared in MessageQueue and implemented in StatsQueue are enqueue and dequeue. These methods are used to enqueue the message into the actor mailbox and dequeue the next message from the mailbox to be processed by the actor, respectively. The method enqueue creates a data container of type MsgEntryStats, and adds it to the end of queue. MsgEntryStats (see Figure 7) stores the original message, encapsulated by an Envelope object in
Akka, along with the size of the mailbox queue, the receiver actor identity, and the message entry time into the queue.

The method `dequeue` obtains the next `MsgEntryStats` from `queue`, and creates a data container of type `MailboxStats` and publishes it to `EventStream`. It also returns the original message of type `Envelope` to the actor for processing. `MailboxStats` (see Figure 7) includes the size of the queue after the message entered the mailbox, the identities of the receiver and sender actors, and the message entry time to and the exit time from the mailbox queue. Using this information for every message in the actor's mailbox, `stat-collector` computes the average wait time for a message in the actor's mailbox. It also computes the average of the mailbox size recorded when enqueueing messages. Note that this does not exactly reflect the average size of the mailbox since it does not take into account the mailbox changes in between enqueues which can stay empty for a while. Therefore, we refer to it as average sampled mailbox size.

The new mailbox type is defined in the configuration file by adding the following lines in the configuration file.

```plaintext
stats-collector-mailbox {
    mailbox-type = "gov.nasa.mesa.reporting.stats.StatsMailboxType"
}
```

Every MESA actor has the boolean property `stats.mailbox` which is `false` by default. If that is set to `true` in the configuration file, when creating the actor, MESA uses `StatsQueue` for the actor's mailbox implementation instead of the default one.

## 4 Akka Threading Model

Before presenting the experiment, we briefly explain the underlying threading model in Akka. Scheduling all actors code in Akka is performed by dispatchers which are low level components built into the Akka toolkit. Note that these dispatchers are completely different from the dispatchers components implemented in MESA. To avoid confusion, in some context, we refer to the ones implemented by Akka as Akka dispatchers. Akka dispatchers are responsible for management of actors mailboxes and the threading strategy. They push messages into actors mailboxes, and associate threads from the thread pools to actors to process messages in their mailboxes. Akka provides a fix number of dispatchers to choose from. User can also assign a certain Akka dispatcher to a group of actors. The built-in dispatcher types in Akka are as follows.

- **Dispatcher** is the default Akka dispatcher which associates all the assigned actors to one thread pool.
- **PinnedDispatcher** provides an actor with an exclusive access to a single thread.
- **BalancingDispatcher** redistributes messages from busy actors to the ones with empty mailboxes.
• CallingThreadDispatcher is only used for testing, and uses the current thread to execute any actor.

Akka provides configuration parameters to tune dispatchers to specific needs. The parameter \texttt{throughput} represents the maximum number of messages processed by the actor before the assigned thread is returned to the pool. The parameter \texttt{throughput-deadline-time} represents the deadline for the actor to process messages each time it executes. One can also specify the underlying thread pool implementation used in the Akka dispatcher by setting it’s the executor component using the parameter \texttt{executor}. By default, Akka uses \texttt{fork-join-executor} which relies on the \textit{work-stealing} pattern where threads always try to find tasks from the submitted tasks to the pool and the ones created by other running tasks. Akka also includes \texttt{thread-pool-executor} which offers a dynamic thread pool that can decrease or increase in size depending on how busy or idle the threads are. Akka also allows users to implement their own customized executor.

The number of threads in the pool is another measure that can be tuned. With too few threads which may cause low CPU utilization, the actors are not able to keep up with the arrival of messages. With too many threads, the context switch time between threads increases which leaves less time for processing the threads. The Akka dispatcher configuration provides three parameters to specify the thread pool size. The parameters \texttt{parallelism-min} and \texttt{parallelism-max} represent the minimum and maximum number of threads, respectively, and \texttt{parallelism-factor} is a factor to calculate the number of threads based on available processors. The size of the thread pool is \texttt{parallelism-factor} multiplied by the number available processors. The number available processors is the value returned by the method \texttt{java.lang.Runtime.availableProcessors()} which gives the maximum number of logical cores available to the virtual machine. If the calculated thread pool size is smaller than \texttt{parallelism-min} or larger than \texttt{parallelism-max} then the thread pool size becomes \texttt{parallelism-min} or \texttt{parallelism-max}, respectively. Moreover, to set the thread pool size to a specific value, one could set both \texttt{parallelism-min} and \texttt{parallelism-max} to that value.

We use the default Akka dispatcher setting in the experiment which is as follows. All actors use the Dispatcher implementation with the default value 5 for \texttt{throughput}, 0 for \texttt{throughput-deadline-time} which indicates no time limit, and \texttt{fork-join-executor} for the executor. Moreover, \texttt{parallelism-min} and \texttt{parallelism-max} are set to 8 and 64, and \texttt{parallelism-factor} is set to 3.

## 5 Experiment

The experiment is divided into two parts. The first part, presented in Section 3.0.2, evaluates the overall performance of the different MESA actor-based systems presented in Section . The second part, presented in Section 3.0.2, evaluates the performance of individual dispatcher and monitor actors.
5.1 Setup

The experiment is performed on an Ubuntu 18.04.3 LTS machine with 31.1 GB of RAM and 10 Intel® Xeon® W-2155 processors with a base frequency of 3.30 GHz and hyper-threading which provides 20 threads (2 x number of cores). The initial Java heap size and the maximum Java heap size, represented by the Xms and Xmx JVM parameters, are set to 12 GB. Moreover, the time is measured using java.lang.System.nanoTime().

For all the runs in this experiment, we use an archive including recorded SFDPS messages as an input. We evaluate the configurations against an input trace, T, including 200,000 messages where each message corresponds to an event of type FlightState or TrackCompleted. T includes data from 3215 different flights. Therefore, the value of n in Figure 3) is 3215. The number of sub-monitors in sfdfsOrderMultiMonitor and sfdfsOrderMultiMonitor_indx is set to 2000 in all the configurations. Moreover, the value returned by java.lang.Runtime.availableProcessors() is 20 where with parallelism-factor set to 3, the size of the thread pool for Dispatcher (the default Akka dispatcher) becomes 60.

5.2 System Performance

To evaluate the overall performance of the actor-based monitoring systems presented in Section , using a bash script, each configuration is executed 10 consecutive times. The average time of the 10 runs used to process the input trace T for the monitoring systems monitor-indexing, dispatcher-indexing, concurrent, and bounded-concurrent are compared in Figure 10. Note that in all the graphs, the time is presented in millisecond, and the legend bcon stands for bounded-concurrent followed by the number of monitors.

Considering the 3215 different call signs in T, monitor-indexing includes one monitor actor which contains one monitor object that tracks all 3215 flights. The dispatcher-indexing system creates one actor that contains a hash map including 3215 monitor objects where each monitor object handles messages from one flight. The concurrent monitoring system creates 3215 monitor actors on-the-fly where each monitor actors handles messages from one flight. The bounded-concurrent monitoring system creates 250 monitor actors where each monitor actor tracks 12 to 13 call signs.

The results show that monitor-indexing exhibits the worst performance followed by the dispatcher-indexing monitoring system. The discrepancy between the run time for these two monitoring systems reveals that applying indexing at the dispatcher level is more efficient that indexing within the monitor. The results also show that bounded-concurrent exhibits the best performance, being slightly faster than the concurrent monitoring system. Table 9 includes ratios that compare the run times for the input trace T.
To investigate the contributing factors behind different run times closer, we run each configuration in Figure 10 once along with VisualVM which is a Java profiler that provides a visual interface to monitor CPU utilization, garbage collector activities, memory usage, and heap data. Figure 11 shows the CPU usage which represents the percentage of total computing resources in use during the run.

It can be seen that the CPU utilization for monitor-indexing is mostly under 20%, and for dispatcher-indexing is mostly between 60% and 40%. These amounts are significantly lower than the CPU utilization for the concurrent and bounded-concurrent monitoring systems which is mostly above 90%. The main reason is that monitor-indexing and dispatcher-indexing use a single actor in the monitoring phase which processes messages sequentially and therefore, they do not take advantage of available cores. Whereas, concurrent and bounded-concurrent use concurrent actors in the monitoring phase. In general, increasing the number of concurrent monitor actors increases the CPU utilization which can lead to a better performance. However, increasing the number of concurrent actors also comes with additional overhead including increase in context-switching between actors which could slow down the system.

It can be seen that in Figure 11, the CPU utilization profile for all the configurations goes down before the system terminates. Once the input trace \( T \) is fully processed, a termination request is sent to all the live actors in the system by the
master actor. The systems with higher number of actors could take longer to shutdown. The run time reported in these experiments is from the start of the run to the point that the system is processed the entire input trace, before the termination mechanism is even triggered. Therefore, the termination process does not influence the results.

Figure 12 shows the heap data for each configuration obtained from VisualVM. The spikes in the profiles are indicative of short living objects collected during a garbage collection cycle which are the messages exchanged between actors in these runs. It can be seen that the heap data consumption for all the runs is below 10G.

Note that in a way the concurrent monitoring system can be seen as a special case of bounded-concurrent where the maximum number of monitors to be created is equal to the number of call signs. Figure 13 compares concurrent with the bounded-concurrent where the maximum number of monitor actors is set to 3215. It can be seen that the bounded-concurrent system is slower. That is due to two reasons. The extra work that goes to maintaining the groups of monitor actors in the case of bounded-concurrent can cause overhead. Moreover, in bounded-concurrent, indexing is applied inside the monitor objects, and since only one call sign is assigned to each monitor instance, the monitor indexing does not

Figure 11: The CPU usage profiles obtained by VisualVM.
Figure 12: The heap data profile obtained by VisualVM.

improve the performance and causes additional overhead.

Figure 13: concurrent versus bounded-concurrent.

To see how the number of monitor actors impact the run time, we compare the bounded-concurrent run times with different numbers of monitor actors, including 125, 250, 500, 1000, 2000, and 3215 actors. Note that increasing the number of monitor actors imposes less load on each monitor actor. The number of flights
tracked by each monitor actors for the systems with 125, 250, 500, 1000, 2000, and 3215 monitor actors is 25-26, 12-13, 6-7, 3-4, 1-2 (with majority 2), and 1 flight, respectively. The average of 10 runs for these configurations are compared in Figure 14. It can be seen that the system performs best with 250 monitor actors, and from there as the number of monitor actors increases the run time increases except when using 1000 monitor actors which exhibits a performance slightly better than the case with 500 monitors. The graph also shows that the system with 125 monitor actors is slightly slower than the system with 250 monitor actors. Increasing the number of monitor actors decreases the load on individual monitor actors, however, it increases overhead from creating and maintaining more actors.

![Graph showing run times with different number of monitor actors.](image)

**Figure 14:** The bounded-concurrent run times with different number of monitor actors.

Using VisualVM, we obtain the CPU utilization and the heap data profiles for the configurations presented in Figure 14. Figure 15 illustrates the CPU utilization profiles for these configurations. It can be seen that in all cases the bounded-concurrent system maintain a high CPU utilization which is mostly over 90%. Figure 16 illustrates the heap data profiles. It can be seen that there is not much discrepancies between the heap profiles as well.
Figure 15: The CPU utilization profiles for bounded-concurrent.
Figure 16: The VisualVM heap data profiles for bounded-concurrent.
5.3 Actors Performance Analysis

This section shows the results for individual actors in the monitoring systems using the features explained in Section 3.0, which are referred to as ASF (stands for actor statistics features) from now on. To activate ASF, the properties stats.mailbox and stats.service are set to true for dispatcher and monitor actors. Moreover, the stat-collector actor is added to the actor system by adding the following block to the configuration file. The experiment presented is this section is also conducted on the Ubuntu machine and uses the inputs and settings presented in Section 3.0.2. Before presenting the results for individual actors, we show the overhead from activating ASF:

```{  
name = "stat−collector"
  class = "gov.nasa.mesa.reporting.stats.StatCollector"
  output−path = "path/to/stat−collector/results"  
}
```

5.3.1 ASF Overhead

Figure 17 compares the different MESA settings with and without ASF. Note that the postfix asf in all the graphs represents runs with ASF activated. It can be seen that all the monitoring systems presented in Figure 17 slow down with ASF activated. Moreover, comparing the overheads reveals that the monitor-indexing system exhibits the largest overhead from activating ASF. Note that in the monitor-indexing setting, ASF is only activated for one monitor, whereas, for concurrent and bounded-concurrent, ASF is activated for the dispatcher and all the monitor actors. Therefore the overall overhead in these systems is larger since there are more messages exchanged between the actors and the stat-collector actor.

Figure 18 and 19 show the CPU usage and the heap data profiles for monitor-indexing, dispatcher-indexing, concurrent, and bounded-concurrent using 250 monitors. To make comparison easier, the figure also includes the results without ASF which are already presented earlier. The results in Figure 18 reveal that activating these features increases the CPU utilization for monitor-indexing and dispatcher-indexing. The reason is that these systems have a very few actors, and one additional runnable actor, stat-collector, which always has messages in its mailbox can lift the CPU utilization. Whereas, for concurrent and bounded-concurrent which already have many actors with high CPU utilization, adding an additional actor does not raise the CPU utilization noticeably.

It can be also seen from Figure 18 that the shutdown process which is the flat part of the profile at the end becomes longer with ASF activated. The shutdown request is sent from the master actor to all the actors in the system in the form of a message that is placed in the actors mailboxes. The longer shutdown process is due to the wait on the stat-collector actor to process the messages waiting in its mailbox. Figure 19 shows that activating ASF leads to larger heap data in all cases. This is due to the overhead from the messages forwarded to stat-collector.
Figure 17: Comparing the MESA settings with and without ASF.

Figure 20 compares the run times for bounded-concurrent with and without ASF using different numbers of monitors. It can be seen that with ASF activated the system with 125 monitor actors gives the best performance. Moreover, with ASF activated, the system with 2000 monitor actors shows a similar performance to the system with 500 monitor actors. Without ASF, using 2000 monitors led to a worse performance than using 500 monitors.

Figure 20: Comparing the bounded-concurrent runs with and without ASF.

As observed for other configuration in Figure 19, activating ASF increases the heap data usage for bounded-concurrent as well. Figure 21 illustrates the heap data profiles when using 250, 500, 2000, and 3215 monitors before and after ASF.
Figure 18: Comparing the CPU utilization profiles with and without ASF.
Figure 19: Comparing the heap usage profiles with and without ASF.
Figure 21: Comparing the heap usage profiles with and without ASF.
5.3.2 Actors Performance Results

Next, we present the results for individual actors in the monitoring systems presented in this report. Figure 22 compares the dispatcher actors parameters for the systems monitor-indexing, dispatcher-indexing, concurrent, and bounded-concurrent using 250 monitors. It shows the average service time, the average wait time in the mailbox queue, and the average sampled mailbox queue size for the dispatcher actor in each setting. These parameters have the highest value for the monitor-indexing system followed by dispatcher-indexing. This reveals that for this particular system under observation, it is more efficient to apply indexing at the dispatcher level than applying indexing at the monitor level when using one actor in the monitoring phase. These results are also aligned with the results presented in Figure 17.

Figure 23 compares the dispatcher actors performance metrics for bounded-concurrent with different numbers of monitor actors. It can be seen that the average service time increases as the number of actors increases. This is aligned with the fact that using more monitor actors increases the load of the dispatcher actor since it needs to generate more monitor actors. Moreover, in general the average message wait time in the queue slightly increases as the number of actors increase, except for the case with 1000 monitors whose wait time is less than the wait time in the case with 500 monitors. In general, this implies the longer the service time in the dispatcher actor, the longer the wait in the dispatcher mailbox. The figure shows that the average sampled mailbox size for the dispatcher actors in all cases only varies in a small range, where the queue is shorter for the dispatcher with smaller service time.

Figure 24 compares the monitor actors performance metrics for bounded-concurrent with different numbers of monitor actors. It can be seen that the average service time for monitor actors varies in a small range, and in general, except for the case with 1000 monitor actors, increases as the number of monitor actors increases. Decreasing the number of monitor actors increases the load on individual actors since each monitor actor deals with higher number of flights. On the other hand, applying indexing within the monitor actors help with improving their performance, and for cases with higher number of monitor actors, indexing within the monitor can lead to overhead as also shown in Figure 13.

The second bar graph in Figure 24 also shows that the systems with less monitor actors have higher average wait time in the queue. This could be due to the higher arrival rate of messages for these monitors. The last bar graph in Figure 24 shows that the average sampled mailbox size for monitor actors in all settings is almost 1. That is, on average, every time a message is placed in the monitor actor mailbox, there is no other messages in the queue in all cases. Note that this measure does not reflect the arrival rate of messages since it does not take into account the periods where the mailboxes are empty.

6 Discussion

Throughout the experiment, we observed various competing sources of overhead
in the MESA actor-based monitoring systems. These factors, which are as follows, need to be taken into consideration when configuring values of the related parameters.

Figure 22: Comparing the dispatcher performance metrics for the different settings in MESA.
Figure 23: Comparing the dispatcher performance metrics for different numbers of monitor actors.

- **Sequential processing of messages** - limiting the number of concurrent monitor actors on a multicore machine can lead to a low CPU utilization. On the other hand, increasing concurrency can increase CPU utilization. Therefore,
Figure 24: Comparing the monitors performance metrics for different numbers of monitor actors.

distributing the input trace among multiple actors rather than using one single monitor actor to process the entire trace sequentially can improve the performance.
• **Actor instantiation** - there is a slight overhead associated with creating a new actor. According to Akka documentation the actor base memory overhead is roughly 300 bytes per instance.

• **Runnable actors** - runnable actors are those actors with non-empty mailbox queues which need to be scheduled to run and process their messages. Assigning actors to threads from the thread pool and context switching between them impose overhead. Therefore, the more runnable actors, the more overhead imposed.

• **Size of mailbox queues** - since MESA uses unbounded mailboxes, increase in the number of messages waiting to be processed in the actor’s mailbox increases the heap data, and depending on the JVM heap size this could lead to more frequent garbage collector cycle. Garbage collection consumes CPU resources which can limit the CPU resources that go to running the monitoring system and therefore, it slows it down.

The combination of these factors can make one setting perform better than the other one. As shown earlier in Figure 10, the bounded-concurrent system with 250 monitors perform the best. This show that for this particular system under observation, the competing factors outlined above lead to an optimal performance when using 250 concurrent monitors.

The main conclusion from the experiments is that the concurrency can improve the performance of a monitoring system considerable. Moreover, there is an optimal number of concurrent monitors which depends on wide variety of competing factors. Given the system under observation, there is no clear way to find the optimal number of concurrent monitor actors upfront. What MESA provides is a platform that can facilitates the processing of finding that optimal number. It provides building blocks for monitoring systems which are highly configurable. One can easily tweak these parameters in the configuration file to observe and study different runs. MESA can be also used to determine the number of concurrent monitor actors dynamically.

### References

