Speculative Concurrent Processing with Transactional Memory in the Actor Model

Yaroslav Hayduk, Anita Sobe, Derin Harmanci,
Patrick Marlier, and Pascal Felber*

University of Neuchatel, Switzerland

Abstract. The *actor* model has been successfully used for scalable computing in distributed systems. Actors are objects with a local state, which can only be modified by the exchange of messages. One of the fundamental principles of actor models is to guarantee sequential message processing, which avoids typical concurrency hazards, but limits the achievable message throughput. Preserving the sequential semantics of the actor model is, however, necessary for program correctness.
In this paper, we propose to add support for speculative concurrent execution in actors using *transactional memory* (TM). Our approach is designed to operate with message passing and shared memory, and can thus take advantage of parallelism available on distributed and multi-core systems. The processing of each message is wrapped in a transaction executed atomically and in isolation, but concurrently with other messages. This allows us (1) to scale while keeping the dependability guarantees ensured by sequential message processing, and (2) to further increase robustness of the actor model against threats due to the rollback ability that comes for free with transactional processing of messages. We validate our design within the Scala programming language and the Akka framework. We show that the overhead of using transactions is hidden by the improved message processing throughput, thus leading to an overall performance gain.

Keywords: Concurrency, actors, transactional memory, speculative processing.

1 Introduction

The actor model, initially proposed by Hewitt [1], is a successful message-passing approach that has been integrated into popular frameworks [2]. The actor model introduces desirable properties such as encapsulation, fair scheduling, location transparency, and data consistency to the programmer. It also perfectly unifies concurrent and object-oriented programming. While the data consistency property of the actor model is important for preserving application safety, it is arguably too conservative in concurrent settings as it enforces sequential processing of messages, which limits throughput and hence scalability.

* Contact author: Anita Sobe, University of Neuchatel, Emile-Argand 11, CH-2000 Neuchatel, Switzerland. E-mail: `anita.sobe@unine.ch`, fax: +41 32 718 2701.

In this paper, we address this limitation by proposing a mechanism to boost the performance of the actor model while being faithful to its semantics [2]. The key idea is to apply speculation, as provided by transactional memory (TM), to handle messages concurrently as if they were processed sequentially. In cases where these semantics might be violated, we rely on the rollback capabilities of TM to undo the operations potentially leading to inconsistencies.

We see a high potential for improvement in scenarios where actors maintain state that is read or manipulated by other actors via message passing. With sequential processing, access to the state will be suboptimal when operations do not conflict (e.g., modifications to disjoint parts of the state, multiple read operations). TM can guarantee safe concurrent access in most of these cases and can handle conflicting situations by aborting and restarting transactions.

Speculation can also significantly improve performance when the processing of a message causes further communication. Any coordination between actors requires a distributed transaction, which we call *coordinated transaction*. We combine coordinated transactions and TM to concurrently process messages instead of blocking the actors while waiting for other transactions to commit.

We have implemented our approach in the Scala programming language and the integrated Akka framework [3]. We evaluate our approach using a distributed linked list benchmark already used with other concurrent message processing solutions [4]. We show that concurrent message processing and non-blocking coordinated processing can considerably reduce the execution time for both read-dominated and write-dominated workloads.

The rest of the paper is organized as follows. We first give an overview of the actor models, transactional memory, and related work in Section 2. We then discuss the limits of the sequential message processing in Section 3 and propose improvements to sequential message processing in Section 4. We describe our implementation in Section 5 and present evaluation results in Section 6. We finally conclude in Section 7.

2 Background and Related Work

Actor models are inherently concurrent. They are widely used for implementing parallel, distributed, and mobile systems. An actor is an independent, asynchronous object with an encapsulated state that can only be modified locally based on the exchange of messages. It comprises a mailbox in which messages can be queued, as well as a set of dedicated methods for message processing [5].

The actor model provides *macro-step semantics* [6] by processing messages sequentially. As a consequence, it also guarantees the following properties:

(1) Atomicity. The state of an actor can only be observed before or after operations took place, therefore changes on the state are perceived either all at once or not at all.

(2) Isolation. The actor model forbids any concurrent access to the local state of an actor. This means that any operation on the state of the actor is done as if it were running alone in the system.

These characteristics make actor models particularly attractive and contribute to their popularity. Numerous implementations of actor models exist in many languages like Java, C, C++, and Python. We decided to use Scala, which is a general-purpose language that runs on top of the JVM and combines functional and object-oriented programming patterns. The recent versions of Scala integrate the Akka Framework [3] for realizing actors. They also supports transactional memory (TM) [7], a programming model that provides atomicity, isolation, and rollback capabilities within transactional code regions [8]. The programmer simply has to demarcate the blocks of instructions that must execute atomically and the TM performs all the necessary bookkeeping to ensure that the target code is executed in a transaction, i.e., the consistency of data accessed within the block are not affected by concurrent accesses. TM provides built-in support for check-pointing and rollback, which we exploit for controlling concurrent message processing.

Existing actor frameworks such as surveyed by Karmani et al. in [2] do not include TM and differ regarding the way they handle parallelism. As an example, implementations of Habanero-Scala and Habanero-Java [4] introduce parallelism by mixing the actor model with a fork-join design (*async-finish* model). Actors can start concurrent sub-tasks (*async* blocks) for the handling of a single message. Since the processing of a message terminates only when all sub-tasks are finished, this approach enforces sequential handling of messages. To alleviate this restriction and improve scalability, Habanero also allows messages to be processed in parallel. To ensure that the actor's state is not accessed concurrently, a *pause and resume* model that works similar to *wait and notify* is used. While processing a message, the actor can spawn external sub-tasks it must then pause to avoid intermediate modification to its state. When the sub-tasks finish with changing the state, the actor resumes its operation and can process further messages. While this approach avoids concurrent access to an actor's state, it must be used carefully as it provides no protection against synchronization hazards such as data races and deadlocks.

Parallel actor monitors (PAM) [9] support concurrent processing by scheduling multiple messages in actor queues. Using PAM, the programmer must understand the concurrency patterns within the application and define application-specific schedulers. This may prove particularly challenging for applications where concurrency patterns vary during execution. In contrast, our approach (see Section 4) removes any programmer intervention and automatically allows concurrent executions when possible. Further, we do not break the original actor semantics at any time, while using an inappropriate scheduler with PAM can cause inconsistencies.

3 Problem Statement

Despite its inherent concurrency, the actor model requires sequential message processing. While this requirement is a deliberate choice to avoid synchronization

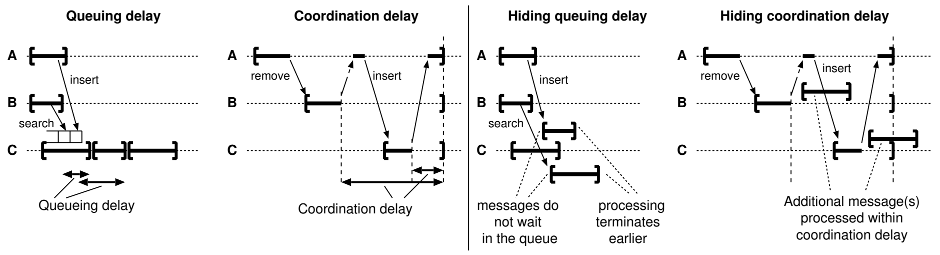
Queuing delay
Coordination delay
Hiding queuing delay
Hiding coordination delay
A
B
C
insert
search
remove
Queueing delay
Coordination delay
messages do not wait in the queue
processing terminates earlier
Additional message(s) processed within coordination delay

Fig. 1. Sequential (left), concurrent and non-blocking coordinated processing (right) and their effect on execution time.

hazards, it unnecessarily limits the performance (i.e., throughput) of individual actors, in particular when they execute on multi- or many-core computers.

We elaborate the problem of sequential processing with the help of the examples depicted in the left part of Figure 1. They involve three actors (A, B and C) performing operations as illustrated on their respective time lines (horizontal dashed line). The transfer of a message between actors is indicated by an arrow and its processing is indicated by thick solid lines, where we explicitly mark the beginning and end of processing with brackets. In our examples, the actors are responsible for maintaining a distributed linked list of ordered integers. Actor B stores the first part of the list and actor C the second part, while actor A acts as a client and performs operations on the list (e.g., `search`, `insert`, `remove`).

Sequential processing in the actor model limits performance in two ways. On a single actor, a message that arrives while another one is processed is enqueued for later handling, which leads to a *queuing delay*. In a distributed transaction, actors block at a commit barrier while waiting for the finalization of all other actors, which leads to a *coordination delay*.

Queuing delay: In the first block of Figure 1 we depict the delay that is introduced upon arrival of multiple messages. If actor A and B send messages to actor C, which is busy, both messages are stored in a queue. Note that if there is no other delay, actors A and B do not block. The queuing delay is the time a message has to wait at a given actor from arrival to the start of its processing.

Coordination delay: In the second block of Figure 1 we depict a common communication pattern. Consider that actor A wants to move the value x from the list of actor B to the list of actor C. For doing so, it sends messages `remove(x)` to actor B and `insert(x)` to actor C. To fulfil the macro-step semantics in actor A, the list operations have to be part of a coordinated transaction, which commits when all three actors successfully finish their task. The coordinated commit protocol defines a barrier on which actors B and C block until they can resume processing other messages. Hence, the coordination delay describes the time actors have to wait after finishing their own tasks until the distributed transaction commits.

4 Message Processing Model

Queuing delays are inherent to the structure of the actor model and its sequential processing operation; their reduction may become particularly important for actors that receive many messages. Further, upon coordination delay, actors block and thus cannot perform any useful work. We claim that these delays are unnecessarily long and can be significantly reduced by thoughtful changes to the message processing of actors.

Our main idea is based on the observation that we can guarantee atomicity and isolation if we encapsulate the handling of messages inside transactions. Thanks to the rollback and restart capability of transactions, several messages can be processed concurrently, even if they access the same state. We call this approach *concurrent message processing*. Additionally, we exploit the characteristics of transactions to avoid blocking actors while waiting for a coordinated commit (*non-blocking coordinated processing*).

To explain the principle of our two optimizations, consider the same example as in Figure 1 with actor A performing operations on a list stored on actors B and C. The right part of Figure 1 illustrates how the delays caused by sequential processing can be reduced.

Queuing delay: By processing several messages concurrently on a single actor, we can reduce the queuing delay as shown in the first block of Figure 1. Therefore, if A and B send a message to C, which is currently busy, the messages do not have to wait. If the transactions do not conflict and can immediately commit, the queuing delay is avoided.

Coordination delay: Actor A wants to move a value from the list of actor B to the list of actor C, which requires both an insertion and a removal action. This is typically achieved using a coordinated transaction. To ensure consistency, however, participating actors cannot process new messages until the coordinated transaction commits. By exploiting the speculative properties of transactions, one can avoid blocking the actors and allow concurrent execution of independent transactions (e.g., as in actors B and C in the second block of Figure 1), therefore hiding the coordination delay.

5 Implementation

To hide the delays as explained before, we extend Scala version 2.10.2. Specifically, we concentrated on two parts of Scala: the Akka framework version 2.10.0 and the Scala-STM [10] library version 0.7. Akka provides a clean and efficient implementation of the actor model for the JVM. Scala-STM supports transactional memory in Scala and, while it adds some overhead for checkpointing and concurrency control, it is particularly non-invasive and well integrated in the language. In the following we describe the specific changes we made to Akka and Scala-STM to realize the proposed optimizations.

Concurrent processing of messages. The concurrent message processing only involves changes of the message handling provided by the Akka framework.

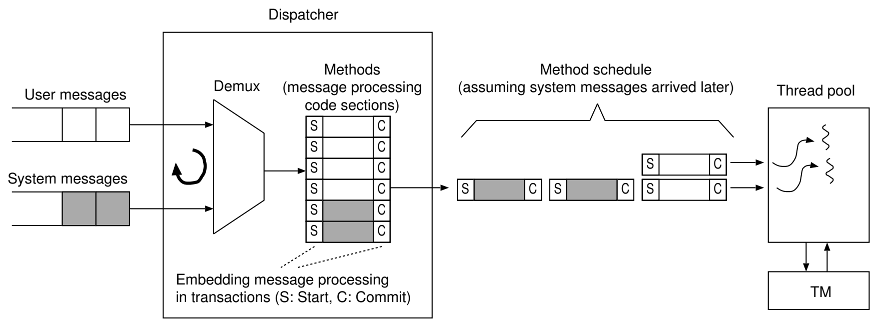
Dispatcher
User messages
System messages
Demux
Methods (message processing code sections)
S
C
Embedding message processing in transactions (S: Start, C: Commit)
Method schedule (assuming system messages arrived later)
Thread pool
TM

Fig. 2. Implementation of concurrent message processing.

Specifically, we changed the behavior of the actor's mailbox. In the original Akka implementation a dispatcher is responsible for ensuring that the same mailbox is not scheduled for processing messages more than once at a given time. Another particularity of Akka is that every actor has one mailbox with two queues: the first one stores user messages, i.e., messages received from other actors, while the second one is used for maintaining system messages specific to Akka, which control lifecycle operations (i.e., start, stop, resume). Once a user message is scheduled, the dispatcher checks first if there are any system messages. Then, all existing system messages are treated before the user message. The same is done after the processing of the user message. As system messages are rare, actors spend most of their time processing user messages.

To facilitate concurrent message processing, we reimplemented the mailbox and message treatment as shown in Figure 2. System messages are handled as in the sequential case, before and after user messages, but instead of processing user messages one at a time, we process them concurrently by batches. Each user message from a batch is submitted to a thread-pool for execution. The actual message processing is performed concurrently inside a transaction, as indicated by the start (S) and commit (C) events in the figure. For the transactional handling of messages we use the default Scala-STM. If the concurrent operations are independent, we can hide the queuing delay as illustrated in Figure 1.

Non-blocking coordinated transactions. The non-blocking coordinated transaction alters the commit behavior of the Scala-STM. By default, a coordinated transaction is blocking (see Figure 1). All actors participating to a coordinated transaction must reach a commit barrier before any other message can be processed.

Consider the case of a transaction that executes a block of code corresponding to the processing of a message. After the transactional code is executed, the STM makes an attempt to commit the changes, possibly rolling back and trying again upon failure. In the process of a commit, several steps are performed: (1) locks for the variables accessed in the transaction are obtained; and (2) if the transaction belongs to a coordinated transaction, an external *decider* is consulted. The coordinated transaction's commit barrier blocks as long as some of

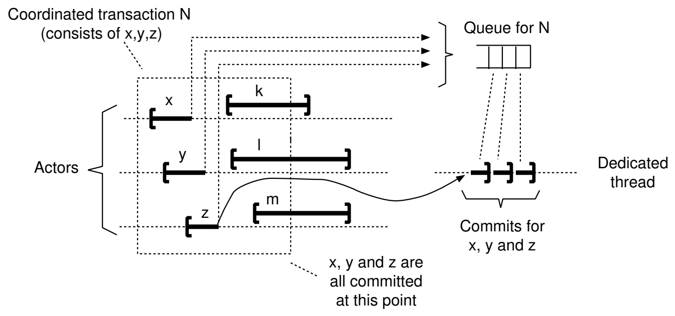

Fig. 3. Sketch of the implementation of non-blocking coordinated processing.

its transactions are still executing. Once they have all successfully completed, the commit barrier is unlocked and control is returned to the caller. After the external decider returns, the transaction does final sanity checks and flushes outstanding writes to main memory. In our implementation, illustrated in Figure 3, we perform the following operations instead of blocking the thread when waiting for other parties to arrive at the barrier. We first save the current transaction descriptor in a queue (*queue for N*). Then, we return from the atomic block immediately, bypassing any additional logic associated with the commit operation. By doing so, we do not fully commit the transaction; we instead suspend its commit at the point where it would normally block.

While an actor is busy with a coordinated transaction, it can handle other messages concurrently, hiding the coordination delay as illustrated on transactions *k*, *l*, and *m* in Figure 1. If concurrent messages are independent, they can commit immediately. If there is a write-write conflict, the processing of one of the messages rolls back. Read-write conflicts represent a special case: if the coordinated transaction reads a value and a concurrent message wants to write the same value, we delay the commit of the write until the coordinated transaction completes. In a system comprising multiple actors, it is likely that several coordinated transactions execute concurrently. Each coordinated transaction uses its own queue to store its suspended *pre-committed* transactions (`N` corresponds to the identifier of the coordinated transaction in the figure). Hence, we do not mix pre-committed transactions belonging to different coordinated transactions. To resume the commit, a dedicated thread is notified when all the parties belonging to the same coordinated transaction have completed their work.

6 Evaluation

Our optimizations are expected to be most useful in applications where state is shared among distributed actors. Hence, to evaluate our approach, we use a benchmark application provided by Imam and Sarkar [4] that implements a stateful distributed sorted integer linked-list. The architecture considers two types of actors: *request* and *list* actors. Request actors only send requests such as `lookup`, `insert`, `remove`, and `sum`. List actors are responsible for handling a range of values (buckets) of a distributed linked list. In a list with l actors, where each actor can store at most n elements representing consecutive integer values, the i^{th} list actor is responsible for elements in the $[(i-1) \cdot n, (i \cdot n) - 1]$ range, e.g., in a list with 4 actors and 8 entries, each actor is responsible for two values. A request forwarder matches the responsible list actors with the incoming requests. We extend this benchmark to evaluate different facets of our proposed optimizations. We evaluate different workloads, different numbers of actors holding elements of the list, etc. For the `sum` operation, each actor holds a variable that represents the current sum of all its list elements, called `partial_sum`, which is updated upon insertion and removal. When computing the sum of the whole list, we only accumulate the partial sum of each list actor without the need of traversing all list elements. While the lookup, insert, and remove operations execute on a single list actor, the sum operation needs to traverse all the list actors in order to return the partial sums. Hence, the sum operation involves multiple list actors. The original benchmark did not initially ensure atomicity of the `sum` operation; we therefore changed the implementation so that the computation of the sum is performed within a coordinated transaction.

We execute the benchmark on a 48-core machine equipped with four 12-core AMD Opteron 6172 CPUs running at 2.1GHz. Each core has private L1 and L2 caches and a shared L3 cache. The sizes of both instruction and data caches are 64KB at L1, 512KB at L2, and 5MB at L3.

We apply each of the extensions—concurrent message processing and non-blocking coordinated processing—to a read-dominated workload and then to a write-dominated workload. Each sample corresponds to the geometric mean of 7 runs. We first evaluate both extensions separately to better assess their benefits and drawbacks, i.e., for the first results, non-blocking coordinated processing does not include concurrent message processing. Then, we conduct experiments with both approaches combined. Their performance is compared against sequential message processing, i.e., using default Akka/Scala constructs without transactions for read and write operations. The sum operation is put into a coordinated transaction as provided by Akka/Scala. We finally complete our evaluation with a comparison against the Habanero-Scala implementation.

Our experiments are either read-dominated (lookup) or write-dominated (insert, remove). These workloads additionally contain a number of sum operations, which are treated as coordinated messages. More precisely, each actor performs $R = x\%$ reads, $W = y\%$ writes, and $S = z\%$ sum operations, where $R + W + S = 100\%$. Since sum requests are likely to be rare in comparison with other operations, we keep this parameter constant at $S = 1\%$. For read-dominated workloads, we choose $R = 97\%$ and $W = 2\%$. The write-dominated workload is configured with $R = 1\%$ and $W = 98\%$.

Insert and remove operations are handled independently and are chosen at random, but each evaluation run gets the same input. We vary the number of list and request actors, each of the latter sending 1,000 requests. The request actors wait for a response before sending the next message. The list can contain

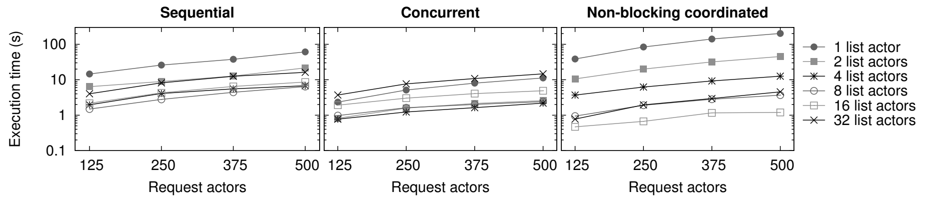

Fig. 4. Execution time for sequential, concurrent, and non-blocking coordinated message processing on a read-dominated workload.

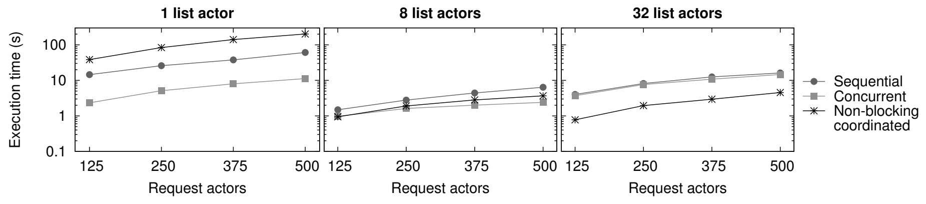

Fig. 5. Execution time for sequential, concurrent, and non-blocking coordinated message processing using 1, 8, and 32 list actors on a read-dominated workload.

a maximum of 41,216 integers split evenly between actors. For instance, if there are 32 list actors, each will be responsible for 1,288 buckets. The list is pre-filled with 20% of its capacity.

6.1 Read-dominated workload

Figure 4 presents the results for the three separate scenarios: sequential message processing, concurrent message processing, and non-blocking coordinated processing. On the x-axis we show the effects of increasing the number of request actors (125–500), while the y-axis displays the execution time in seconds (log scale), i.e., the time needed to finish processing all requests. The lower the execution time, the better. One can see in all cases that adding more request actors leads to higher execution times, which is not surprising because the workload becomes higher.

The concurrent execution time (Figure 4, center) is lower than the sequential scenario up to 16 list actors, which indicates that allowing multiple messages to be processed concurrently introduces an immediate performance gain. In our linked-list example, messages sent from request actors R1 and R2 to a list actor L1 will be treated within a transaction. If there are no conflicts—which is likely in a read-dominated workload—concurrent transactions commit successfully. Therefore, the time required to process a batch of messages will be equal to the execution time of the longest associated transaction $(\max(T_1, T_2, \ldots))$ instead of the sum of all execution times $(\sum T_i)$. The performance with 16 and 32 actors starts to degrade because the workload provides less exploitable concurrency. With 32 list actors the performance is even worse than with a single list actor.

When considering non-blocking coordinated message processing (Figure 4, right), the explanation for the increase of the execution time for concurrent processing with 16 and 32 list actors is clear: when the number of list and request actors is high, coordinated transactions are likely to fail because of increased contention. Non-blocking coordinated message processing allows us to reduce this execution time considerably. Actors can process other messages while the sum operation is in progress. The reduction of execution time is especially high for read-dominated workloads, because a lookup operation and the read of the partial sum are independent operations. With large numbers of list and request actors, however, the likelihood of insert and remove operations increases significantly.

Figure 5 shows a more detailed comparison for an increasing number of list actors. The left graph in presents the execution time for a single list actor. One can see that concurrent message processing improves the execution time considerably, while non-blocking coordinated processing exhibits worse performance than sequential message processing. Indeed, since there is only one list actor, no coordinated transactions are executed, i.e., the sum operation only returns the partial sum of the current list actor. Hence, the execution time of the non-blocking coordinated processing shows the overhead of executing all operations inside transactions. When increasing the number of list actors, this overhead is compensated by the benefits of non-blocking coordinated processing. When increasing the number of list actors to at least 8 (Figure 5, center), the contention of coordinated transactions increases and non-blocking processing performs even better than concurrent message processing. When the number of coordinated transactions and write-write conflicts becomes too high, concurrent message processing yields performance similar to sequential processing, as can be observed in the right graph of Figure 5 for 32 list actors. In contrast, non-blocking coordinated transactions lead to significantly lower execution times than both sequential and concurrent message processing.

To summarize our findings so far, concurrent message processing has the highest impact if the number of list actors is low because each will have more messages to process, i.e., the penalty from serialization is more important and the workload provides more exploitable concurrency. The opposite trend can be observed with non-blocking coordinated transactions: they benefit most when the number of list actors is high because coordinated transactions become longer, i.e., the penalty of the blocking operation is higher and contention is relatively low. Therefore, the combination of both techniques is expected to provide good overall performance for all considered scenarios.

6.2 Write-dominated workload

We expect to observe more conflicts with a write-dominated workload because each insert and remove operation also modifies the value of the partial sum. As a consequence the execution time generally increases in comparison with the read-dominated load, as shown by the graphs in Figure 6.

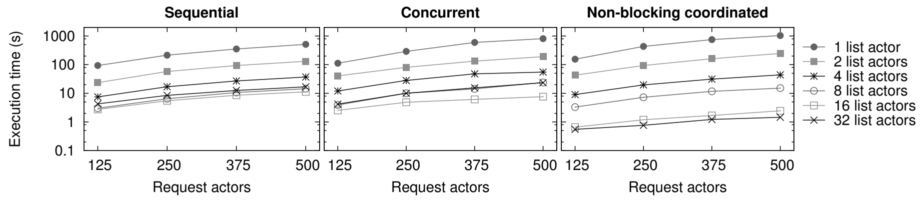

Fig. 6. Execution time for sequential, concurrent, and non-blocking coordinated message processing on a write-dominated workload.

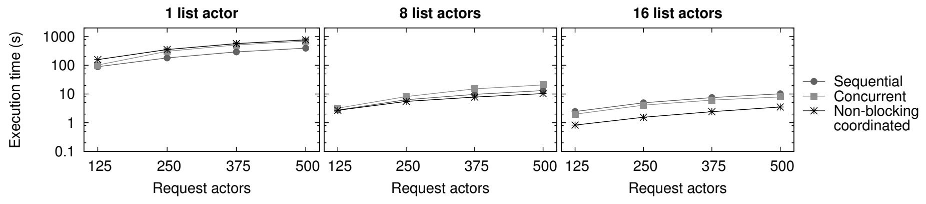

Fig. 7. Execution time for sequential, concurrent, and non-blocking coordinated message processing using 1, 8, and 16 list actors on a write-dominated workload.

Sequential processing (Figure 6, left) performs similarly to the read-dominated workload case. The execution time first improves when adding more list actors. Then, we observe similar execution times for 8 and 16 list actors, and the degradation starts for 32 actors as the impact of coordinated transactions becomes more significant. Concurrent processing (Figure 6, center) provides better overall performance, but the best improvement is obtained with 16 list actors when there is sufficient exploitable concurrency. Finally, with non-blocking coordinated processing, performance improves with the number of list actors. The execution time is better than concurrent processing starting from 4 list actors. The reason is that sum operations are read operations. Thus, for coordinated transactions a write-write conflict is not possible. We expect that write operations in coordinated transactions conflicting with other writes lead to execution times close to concurrent processing.

Figure 7 shows the execution times of sequential, concurrent, and non-blocking coordinated processing for various sizes of list actors. With a single list actor (Figure 7, left), we observe that concurrent and non-blocking coordinated processing perform poorly due to the many write-write conflicts and resulting aborts.

With 8 list actors (Figure 7, center) the performance of non-blocking coordinated processing becomes close to sequential processing, whereas concurrent processing still has a higher execution time. Finally, with 16 list actors the advantage of non-blocking coordinated processing is obvious, while concurrent processing now performs similarly to sequential processing because the coordination delays dominate.

Summing up the write-dominated workload results, we conclude that concurrent processing becomes less beneficial when the likelihood of write conflicts

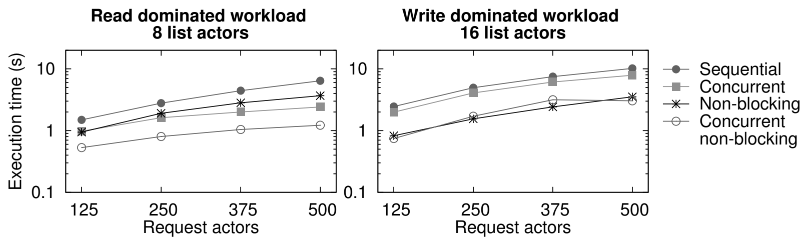

Fig. 8. Execution time for sequential, concurrent, non-blocking coordinated, and combined message processing.

is high. In some cases, the high number of roll backs becomes high enough that sequential processing should be preferred. Coordinated transactions have a high influence on the execution time and significantly improve performance with many list actors. Therefore, a write-dominated workload can benefit more from non-blocking coordinated transactions than concurrent ones, and it is debatable whether the latter extension should be used at all when the number of write-write conflicts becomes very high.

6.3 Non-blocking concurrent processing

We conducted the same experiment as before, but combined concurrent and non-blocking coordinated message processing, which we call *non-blocking concurrent processing*, for the read and write-dominated workload. In the read workload, concurrent processing leads to lower execution times when the number of list actors is below 16. Indeed, one can observe that the performance of non-blocking concurrent processing is even better than non-blocking coordinated processing (Figure 8, left). When we increase the number of list actors, we see again the same behavior as for the write-dominated workload. The combination is thus useful when both concurrent and non-blocking coordinated processing lead to a lower execution time than sequential processing.

The results for the write-dominated workload show that concurrent processing does not have much influence on the execution time (Figure 8, right). In the 16 buckets scenario, pure concurrent processing has execution times similar to sequential processing, while non-blocking concurrent processing results in performance close to non-blocking coordinated processing.

To fully exploit the capabilities of the proposed mechanisms, it is therefore necessary to properly understand the nature of the workload. If it is read-dominated and the number of list actors is high, one should favor non-blocking coordinated processing. If the number of list actors is low, one should rather use non-blocking concurrent processing. Finally, with a write-dominated load, one should prefer non-blocking coordinated processing or even switch back to sequential processing.

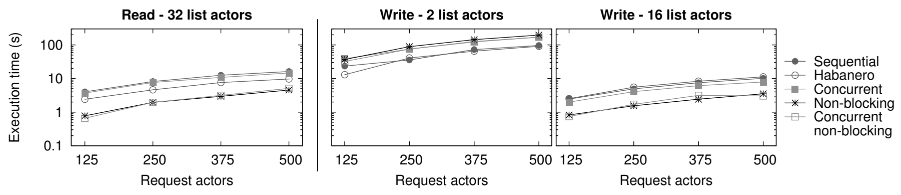

Fig. 9. Execution time comparison with Habanero-Scala.

6.4 Comparison to Habanero-Scala

In the original Habanero-Scala benchmark shown in Figure 20 in [4], the authors ran their experiments with 64 list actors, each responsible for 400 buckets. Additionally, the authors used a workload with a balanced mix of reads and writes (50:50). 50 read actors are used to access the list elements, with 32,000 accesses by actor. For these settings, Habanero-Scala (*LightActor* implementation) performed slightly better than default sequential processing (Akka). Note that the LightActor implementation of the list benchmark does not spawn any sub-tasks. It use a *finish* construct instead of the default countdown latch for coordinating list and request actors of the list. Thus, the difference in performance is due to their lightweight implementation of actors, a custom task scheduler, and a different thread-pool implementation.

To show the full capabilities of our approach, we executed our experiments with Habanero-Scala. For this, we use the default Habanero-Scala constructs for read and write operations (no transactions). The behavior of the sum operation is provided by a barrier implemented in the list actors using the *DataDriveFuture* construct (including sub-tasks) of Habanero-Scala.

For the read-dominated workload, Habanero-Scala performs similarly to sequential message processing, except for 32 buckets where Habanero-Scala performs better by approximately 40%. There, the difference is higher, because of the penalty of the blocking coordinated sums used in the sequential implementation. As seen in the left graph of Figure 9, our approach outperform Habanero-Scala by 50 to 70%.

In the write-dominated workload, Habanero-Scala performs again similarly to sequential message processing. With our speculative extensions, the contention is too high for less than 8 list actors. With 16 list actors the improvement of our approach is significant (Figure 9, right).

6.5 Discussion

Our approach, which combines concurrent and non-blocking coordinated processing, guarantees the same correctness as the original actor model. We perform the processing of messages within transactions, which means that concurrent operations on the actor's state will execute atomically and in isolation. Therefore, conflicting operations will be serialized, but independent messages should be processed concurrently.

It is important to note that the actor model does not impose any order on the processing of messages that are in its incoming queues. Therefore the non-deterministic order in which transactions will commit does not break the semantics of the actor model. If ordering were required, we could extend the STM as proposed in [11] to enable parallel processing but commit transactions in order.

As we rely on transactional memory to process messages equivalently to a serial execution and we preserve the original actor model, concurrent processing also provides the same correctness guarantees. The same applies for non-blocking coordinated processing. All messages are handled within transactions, which means that the conflicts are handled as for concurrent message processing. However, for read-write conflicts (e.g., concurrent sum and insert operations) the order will be preserved by our delayed commit mechanism.

An issue that currently limits our approach is that the code of transactified message processing should not contain any action that is not under the control of the TM, such as I/O or OS library code (irrevocable code). Strategies for supporting irrevocable actions are left to the responsibility of the underlying TM and are not specific to our extensions.

Our approach has the important benefit of being adaptabile. The transactional processing of a message can be aborted at any time without side effects on the current state of the actors. This implies that each of our extensions can be enabled or disabled at any time during execution, and one can switch from one extension to the other within the same execution. Such flexibility can be exploited to play with trade-offs between performance and resource utilization. On the performance side, messages need to be processed anyways, either sequentially or concurrently. Hence, if we have idle resources and we can process messages concurrently, the overall task can be completed in a shorter time. On the resource utilization side, the adaptability of our approach allows us, for example, to apply simple energy-efficiency strategies that enable or disable transactional execution, possibly even temporarily switching off some cores, in order to fit the consumption of the application to the desired energy requirements.

7 Conclusion

The actor model implements synchronization by the means of message passing. This decoupled communication paradigm is particularly scalable since it allows multiple actors to perform independent computations concurrently as they do not share state. However, each actor processes arriving messages sequentially.

To address this limitation, we proposed an approach that combines transactional memory (TM) and actors as implemented by the Akka Framework. Incoming messages are dequeued in batches and processed speculatively inside transactions. The atomicity, consistency, and isolation properties of TM guarantee that messages do not interfere when being processed concurrently. In addition, as the actor model does not impose any order on the handling of user messages in the incoming queues, our approach preserves its semantics.

To further improve concurrency, we also extended the coordinated transaction mechanism of Scala-STM to support non-blocking operations. The traditional design prevents actors involved in a coordinated transaction to process any additional message until the transaction commits. The resulting delays can be especially high when actors are distributed on several nodes and communication has non-negligible latency. We solve this issue by speculative concurrent message processing.

Together, these two mechanisms can significantly lower the queuing and coordination delays, and hence increase concurrency. We implemented both mechanisms in the Scala language using the integrated Akka framework. Experiments on a 48-core server show that our extensions provide important performance benefits over sequential processing on both read-dominated and write-dominated workloads.

Acknowledgements. This research has been funded in part by the European Community's Seventh Framework Programme under the ParaDIME Project (www.paradime-project.eu), grant agreement no. 318693.

References

1. Hewitt, C., Bishop, P., Steiger, R.: A universal modular actor formalism for artificial intelligence. In: IJCAI'73: Proceedings of the 3rd international joint conference on Artificial intelligence. (1973) 235–245
2. Karmani, R.K., Shali, A., Agha, G.: Actor frameworks for the jvm platform: a comparative analysis. In: PPPJ '09: Proceedings of the 7th International Conference on Principles and Practice of Programming in Java. (2009) 11–20
3. Haller, P.: On the integration of the actor model in mainstream technologies: the scala perspective. In: Proceedings of the 2nd edition on Programming systems, languages and applications based on actors, agents, and decentralized control abstractions. AGERE! '12, New York, NY, USA, ACM (2012) 1–6
4. Imam, S.M., Sarkar, V.: Integrating task parallelism with actors. In: Proceedings of the ACM international conference on Object oriented programming systems languages and applications. OOPSLA '12 (2012) 753–772
5. Agha, G.A., Mason, I.A., Smith, S.F., Talcott, C.L.: A foundation for actor computation. J. Funct. Program. **7**(1) (1997) 1–72
6. Karmani, R.K., Agha, G.: Actors. In: Encyclopedia of Parallel Computing. (2011) 1–11
7. Goodman, D., Khan, B., Khan, S., Luján, M., Watson, I.: Software transactional memories for scala. Journal of Parallel and Distributed Computing (2012)
8. Harris, T., Larus, J., Rajwar, R.: Transactional Memory, 2nd Edition. 2nd edn. Morgan and Claypool Publishers (2010)
9. Scholliers, C., Tanter, E., Meuter, W.D.: Parallel actor monitors. In: SBLP'10: 14th Brazilian Symposium on Programming Languages. (2010)
10. ScalaSTM. http://nbronson.github.com/scala-stm/
11. Brito, A., Fetzer, C., Sturzrehm, H., Felber, P.: Speculative out-of-order event processing with software transaction memory. In: DEBS '08: Proceedings of the international conference on Distributed event-based systems. (2008) 265–275