Dynamic Message Processing and Transactional Memory in the Actor Model

Yaroslav Hayduk(✉), Anita Sobe(✉), and Pascal Felber(✉)

University of Neuchatel, Switzerland
{yaroslav.hayduk,anita.sobe,pascal.felber}@unine.ch

Abstract. With the trend of ever growing data centers and scaling core counts, simple programming models for efficient distributed and concurrent programming are required. One of the successful principles for scalable computing is the actor model, which is based on message passing. Actors are objects that hold local state that can only be modified by the exchange of messages. To avoid typical concurrency hazards, each actor processes messages sequentially. However, this limits the scalability of the model. We have shown in former work that concurrent message processing can be implemented with the help of transactional memory, ensuring sequential processing, when required. This approach is advantageous in low contention phases, however, does not scale for high contention phases. In this paper we introduce a combination of dynamic resource allocation and non-transactional message processing to overcome this limitation. This allows for efficient resource utilization as these two mechanisms can be handled in parallel. We show that we can substantially reduce the execution time of high-contention workloads in a micro-benchmark as well as in a real-world application.

1 Introduction

Recent scaling trends lead to ever growing data centers and cloud computing is gaining attention. Further, the scaling trends at the CPU level let us expect increasing core counts in the following years. This causes limitations of performance gains as it is difficult to program distributed and concurrent applications efficiently. The current methods using shared memory do not keep up with the hardware scaling trends and might need to be abandoned. The actor model, initially proposed by Hewitt [1], is a successful message passing approach that has been integrated into popular local and distributed frameworks [2]. An actor is an independent, asynchronous object with an encapsulated state that can only be modified locally based on the exchange of messages. Received messages are processed sequentially avoiding the necessity of locks. With increasing numbers of actors the model is inherently parallel. To sum up, the actor model introduces desirable properties such as encapsulation, fair scheduling, location transparency, and data consistency to the programmer.

While the data consistency property of the actor model is important for preserving application safety, it is arguably too conservative in concurrent settings

A. Bessani and S. Bouchenak (Eds.): DAIS 2015, LNCS 9038, pp. 94–107, 2015.
DOI: 10.1007/978-3-319-19129-4_8

as it enforces sequential processing of messages, which limits throughput and hence scalability. With sequential processing, access to the state will be suboptimal when operations do not conflict, e.g., modifications to disjoint parts of the state and multiple read operations.

In previous work [3], we improved the message processing performance of the actor model while being faithful to its semantics. Our key idea was to use transactional memory (TM) to process messages in an actor concurrently as if they were processed sequentially. TM provides automatic conflict resolution by aborting and restarting transactions when required. However, we noticed that in cases of high contention, the performance of parallel processing dropped close to or even below the performance of sequential processing. To improve the performance of such high contention workloads, we propose to parallelize the processing of messages in a transactional context with messages that can run in a non-transactional context.

First, we determine the optimal number of threads that execute transactional operations as the performance of an application is dependent on the level of concurrency [4], [5]. To avoid high rollback counts in high contention phases fewer threads are desired. In contrast to low contention phases where the number of threads can be increased. Second, we extract messages for which we can relax the atomicity and isolation and process them as non-transactional messages.

Much of the contention in our tests was caused by read-only operations on TM objects [3]. Rollbacks could be avoided by relaxing the semantics of read operations such as proposed by Herlihy et al. [6] who introduced *early release*, which allows to delete entries from the read set. Another way is to suspend the current transaction temporarily, which is called *Escape Action* [7]. These approaches are only partly realized in current STMs such as the Scala STM (based on CCSTM [8]). Scala STM uses *Ref* objects to manage and isolate the transactional state. The Ref object does not permit accessing its internal state in a non-transactional context. Instead, Scala STM provides an *unrecorded read* facility, in which the transactional read does not create an entry in the read set but bundles all meta-data in an object. At commit time the automatic validity check is omitted, but may be done by the caller manually. However, in the presence of concurrent writes on the same value, the unrecorded read might cause conflicts and hence performance would degrade. Our proposal is to break the isolation guarantees of the Ref object in specific cases, i.e., if we limit the reads to specific isolated values, we can omit using the unrecorded read and grant direct access. By using direct access for a number of scenarios in the context of the actor model, we can process a substantial amount of read-only messages while not conflicting with messages processed in regular transactions (TM messages).

Possible candidates for such read-only operations can accept inconsistencies while not interrupting with transactional states. Examples are operations that can be used to make heuristic decisions, or operations that are known to be safe because of algorithm-specific knowledge. Furthermore, debugging and logging the current state in long-running applications are candidates for read-only operations.

The read-only and the TM messages may require different levels of concurrency. Following this observation, during high contention phases, we reduce the number of threads processing regular TM messages, which in turn allows us to increase the number of threads processing read-only messages. By handling these two message types separately (i.e., providing a separate queue for each of the message types) we can optimally use all available resources. We show the applicability of our approach by using a micro-benchmark and a real-world application.

This paper is organized as follows: In Section 2 we give an overview on related work. In Section 3 we introduce the basics of our former work and in Section 4 we discuss the proposed extensions. The benchmarks are described and evaluated in Section 5. Finally, we conclude our paper.

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 [9].

The actor model provides *macro-step semantics* [10] by processing messages sequentially. As a consequence, it also guarantees the following properties:

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.

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 popular 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 [11] for implementing actors. Scala also supports transactional memory (TM) [12], a programming model that provides atomicity, isolation, and rollback capabilities within transactional code regions [13]. TM provides built-in support for checkpointing and rollback, which we exploit for controlling concurrent message processing. Existing actor frameworks such as those 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 [14] introduce parallelism by mixing the actor model with the fork-join model (*async-finish* model). Actors can start concurrent sub-tasks (*async* blocks) for the handling of a single message.

When all sub-tasks complete their execution, the actor resumes its operation and can process further messages. While this approach avoids concurrent access

to the 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) [15] 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.

To optimally use the resources and to improve performance, researchers proposed several mechanisms to match the level of concurrency with the current workload. Heiss and Wagner [5] discuss the problem of thrashing in concurrent transactional programs. Thrashing is a phenomenon that takes place in phases of high contention in which it is likely that the throughput suddenly drops. They propose three ways to avoid thrashing. First, they propose to set an upper bound that sets the maximum number of concurrent transactions; second, they propose to use analytical models for preventing high contention phases; and third, they propose to monitor the current load and decide dynamically on the best level of concurrency. The last approach is seen as a dynamic optimization problem that considers the relationship of concurrency level and throughput. Similarly, Didona et al. [4] propose to dynamically adjust the level of concurrency according to the number of commits and aborts. The optimal number of threads is found with the help of two phases: (1) measurement phase and (2) decision phase. In the first phase the application is profiled with a fixed number of threads, in the second phase a hill-climbing approach is applied to increase and decrease the number of threads according to the given workload, maximizing the transaction's throughput based on successful commits. While the basic idea is interesting, the variation of the thread count based on the throughput might be disadvantageous for transactions with different granularity.

3 Concurrent Message Processing

To motivate our proposed work, we relate to the enhancements of the Actor Model as presented in our former work [3]. There, we reduced the execution time without violating the main characteristics of the Actor Model. Our main idea was 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. The concurrent message processing only changes the message handling provided by the Akka framework as integrated in Scala 2.10.0. Specifically, we altered the behavior of the actor's mailbox processing code. 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. We adapted the dispatcher to assign each message processing to a thread of the thread pool. Further, each message

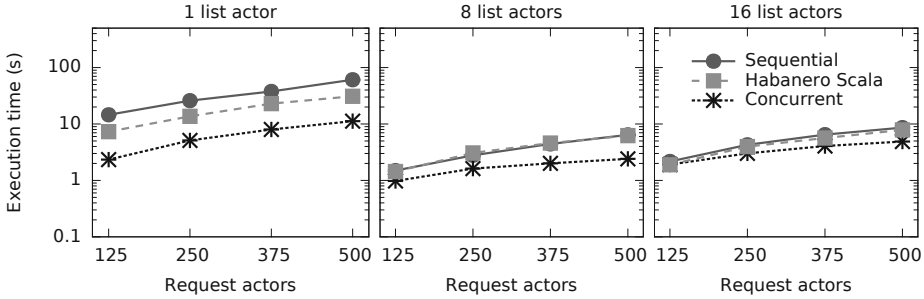

Fig. 1. Concurrent message processing in a read-dominated workload within a shared linked list

processing is handled in a transaction for which we use the default Scala STM. Our work performed well for read-dominated as well as write-dominated workloads and we outperformed the state-of-the-art Habanero Scala [14] in their distributed list benchmark shared amongst list actors with 97% of reads, 2% writes and 1% sum operations created by request actors. Figure 1 shows performance improvements over sequential processing and Habanero Scala for all numbers of list actors in a shared linked list. 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. A higher number of list actors also leads to higher contention, increased rollback counts and hence decreased performance. With 16 list actors, concurrent processing as well as Habanero Scala perform close to sequential processing.

4 Dynamic Concurrent Message Processing

For improving the performance in high contention workloads we propose the following combination of methods. First, we adapt the level of concurrency for processing actor messages according to the current contention level. Second, we extract read-only message processing from the transactional context. And third, we exploit the fact that the two types of messages do not interfere and occupy the existing threads with both types of messages according to the current contention level. As a result, we occupy all threads with work. To differentiate between these two types of messages, we adapted the concurrent mailbox implementation in Akka (as part of Scala 2.10) to handle two different queues as shown in Figure 2. One queue collects the messages to be processed in a transactional context (STM messages), and another one for the processing of read-only messages. We further adapted the actor message dispatcher such that it picks messages from both queues and forwards them for processing to the thread pool. Our new dispatcher automatically adapts the number of STM messages and read-only messages to be processed according to the current contention of the workload. The detailed principles of processing STM messages and read-only messages are described in the following sections.

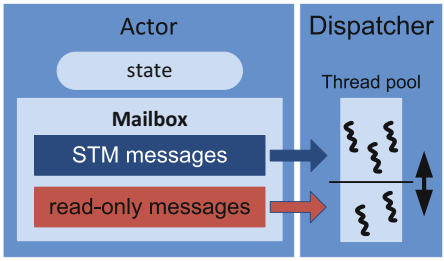

Fig. 2. The different handling of STM messages and read-only messages. Dispatcher assigns a dynamic number of threads for message processing.

4.1 STM Message Processing

At the beginning of an application we start from a random number of threads from the thread pool to process transactional messages (STM messages) and then, driven by a predefined threshold α, the level of concurrency is adapted. The dispatcher assigns the rest of the threads to process read-only messages. α is dependent on the knowledge of the current number of commits and rollbacks of transactionally processed messages and their ratio. Instead of focusing on the throughput such as Didona et al. [4], we are able to support transactions of any granularity by steering α with the commit-to-rollback ratio. If the current commit-to-rollback ratio is lower than α, we divide the number of threads processing transactional messages by two; if it is higher, we multiply them by two. We chose the commit-to-rollback ratio in combination with the fixed α threshold due to its simplicity. To find the right α value for the current workload, we consider a short profiling phase monitoring the relation of commits and rollbacks.

4.2 Read-only Message Processing

The second category of messages are read-only messages. They are handled in a separate queue from the messages that require TM context. Scala STM [16] is based on CCSTM [8] (which extends SwissTM [17]). It is a write-back TM, where writes are cached and written to the memory on commit. Further, it provides eager conflict detection for writes and lazy conflict detection for reads. Validation is done based on a global time stamp. In Scala STM transactional objects are encapsulated in so-called *Ref*s. This implies that any access to a transactional object has to be within an atomic block, which ensures strong atomicity and isolation.

We argue that in some cases we can relax isolation, e.g., for performing approximate read-only operations. A ubiquitous example is a sequential data structure such as a linked list. While traversing the list in read-only mode, a concurrent write of a value, which has been already read, causes a conflict. For such cases, Scala STM provides a mechanism called *unrecorded read*. An unrecorded read is a transactional read, which returns an `UnrecordedRead` object containing the

value, the version number before read, and a validity field of the read. The validity field returns true when there were no changes to the value, which helps to resolve the ABA problem [8].

In Scala STM the unrecorded read can be accessed through calling the *relaxedGet* method. By using it, we can perform a read that will not be validated during commit. The *relaxedGet* method has to be executed inside of an atomic block:

```
atomic{
  val unvalidatedValue = ref.relaxedGet({(_,_) => true})
}
```

Unrecorded reads do not yield new entries in the read set, but still need to ensure reading the latest version of a TM object. Scala STM checks for concurrent writes and forces eager conflict detection, which, in turn, causes a rollback of the writing transaction. As we see later this resolution strategy leads to similar runtime behavior as of the regular transactions.

To remove the overhead associated with unrecorded reads, we propose to provide direct access to the Ref object's data (which is safely possible due to the write-back characteristic of Scala STM). We extended Scala STM's Ref implementation with a *singleRelaxedGet* method and provide an example for generic references below.

```
class GenericRef[A](@volatile var data: A) extends BaseRef[A] {
   ...
   def singleRelaxedGet(): A = data
}
```

Its usage is then straightforward as shown in the next example:

```
val relaxedValue = ref.singleRelaxedGet()
```

As the *data* variable in *GenericRef* is marked as *volatile*, the *singleRelaxedGet* read-only operation can therefore safely interleave with other transactional write operations while guaranteeing to see the latest result. Moreover, *singleRelaxedGet* does not interfere with other transactional operations, i.e., it cannot force another transaction to roll back.

5 Evaluation

Our optimizations are expected to be most useful in applications where state is shared among many actors. Hence, to evaluate our approach, we use a benchmark application provided by Imam and Sarkar [14] that implements a stateful distributed sorted integer linked-list. It is the same benchmark as in our former work and shown in Section 3 in comparison to Habanero Scala. This benchmark is relevant as sequential data structures are typical applications for relaxed atomicity and isolation (see, e.g., *early release*). The read-only operation is a non-consistent sum that traverses each list element.

To show wider applicability we also consider a real-world scientific application that is used to simulate the hydraulic subsurface. The read-only operations in this application are used to control progress and debugging and hence should not interfere with any regular operation.

The thread pool is configured to support two scenarios. First, we specify a static ratio, in which 90% of threads are assigned to process STM messages and the rest processes read-only messages. In the second case we consider a dynamic ratio, but ensure that the number of threads assigned to process any message type never drops below the ratio of 10%. We compare the performance of our proposed *singleRelaxedGet* to the default *atomic* block and the *relaxedGet* method provided by Scala STM.

We execute the benchmarks 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.

5.1 List Benchmark

The list benchmark comprises two actor types: *request* and *list* actors. Request actors 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. We implemented a list element as an object containing a *value* field and a *next* field, which is wrapped in a Scala STM Ref object.

In a list with l list actors, where each actor stores 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 in total, each actor is responsible for two values. A request forwarder matches the responsible list actors with the incoming requests. For the *sum* operation, we traverse each list element in every list actor. This operation is read-only and does not necessarily report a consistent value. It should not conflict with other accesses to the list.

We run the benchmark with 32 threads (to be able to divide and multiply the thread count by two) in 7 runs from which we take the median throughput and number of rollbacks for the results. Also, we create 8 list actors that are in responsible for 41,216 list elements. We create 500 request actors, where each actor sends 1,000 messages to the list. After each of the request actors finished their 1,000 requests the benchmark terminates.

To read a value from the list, we consider three different options: (1) the regular transactional read (*node.next.get()*), (2) the unrecorded read accessed via (*node.next.relaxedGet()*) and (3) our direct access (*node.next.singleRelaxedGet()*) method.

In our experiments we consider a write-dominated and a mixed workload. The write-dominated workload is configured as follows: Each request actor sends 98% of requests to modify the list (insert or remove), 1% of lookup requests and 1% of sum requests.

In the first experiments, we evaluate the impact of different access approaches to the sum operation if the threads are assigned statically. The static approach (90% STM:10% read-only) reserves 3 threads (10%) for the processing of read-only messages and 29 threads (90%) for processing STM messages. Figure 4 demonstrates the message throughput (left) and the rollback count over time (right). The shorter the line, the better the execution time. The *singleRelaxedGet()* outperforms the other operations with respect to both, execution time (50%) and throughput (40%). However, we observe a drastic increase of rollbacks (90%). This observation is counter intuitive, as one would expect to have a lower number of rollbacks to achieve higher throughput. In fact, when we use *atomic* and *relaxedGet()* to implement the sum operation, we cause significantly more read-write conflicts. Scala STM resolves them by waiting for the write operations to finish to follow up with the execution of the read operations. On the contrary, when we use the *singleRelaxedGet()* operation, we remove transactional read operations, which increases the likelihood of concurrent write operations. As a result, we get more write-write conflicts, which are typically resolved eagerly by rolling back one of the transactions.

Since the performance of transactional operations can be further improved, we dynamically determine the optimal number of threads as described in Section 4.1. We schedule a thread, which obtains the total number of commits and rollbacks every second, and decides the optimal number of threads to schedule for the processing of STM and read-only messages. Initially, the thread counts are randomly chosen and within the first two seconds the number of threads converges to the optimal values with the help of α. For determining α we profile the application for a short time. In the case of the list benchmark this results in $\alpha = 0.21$, hence if the commit-to-rollback ratio is below α the number of threads will be reduced else increased (multiplied or divided by two).

In Figure 4 we see that the throughput and rollback values for the *atomic* and *relaxedGet()* operations are not significantly different when compared to the static approach. This behavior is expected as the operations interfere with the concurrently executing transactional write operations. This causes more contention as both the STM messages and the read-only messages are conflicting.

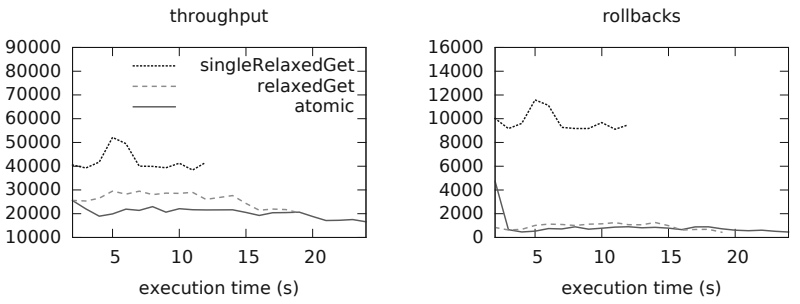

Fig. 3. Throughput (left) and rollback count (right) for the list write-dominated workload: static thread allocation

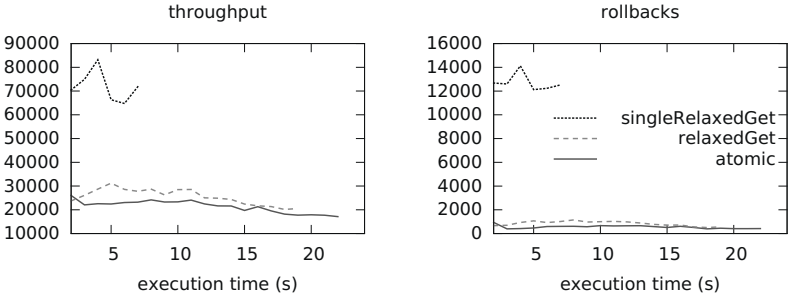

Fig. 4. Throughput (left) and rollback count (right) for the list write-dominated workload: dynamic thread allocation

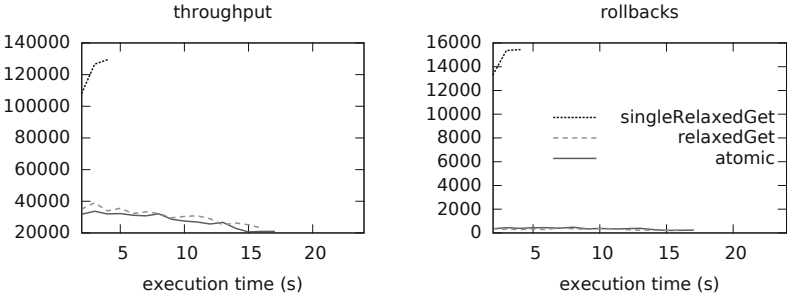

Fig. 5. Throughput (left) and rollback count (right) for the list mixed workload: static thread allocation

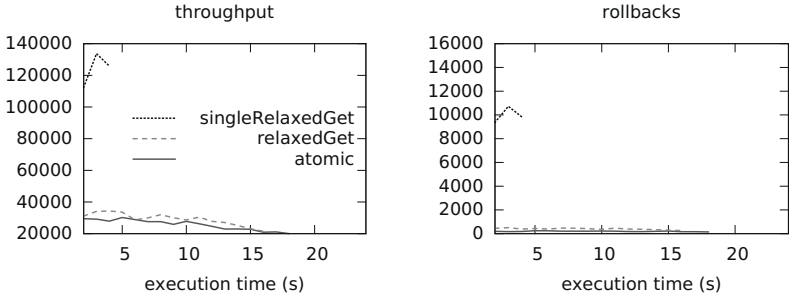

Fig. 6. Throughput (left) and rollback count (right) for the list mixed workload: dynamic thread allocation

On the contrary, the *singleRelaxedGet()* operation never conflicts with other list operations. Hence, we can efficiently use the threads not processing STM messages resulting in increased throughput (65%) and reduced runtime (70%).

In the mixed read-write workload, consisting of 50% of requests to modify the list, 49% lookup requests and 1% sum requests, we show the applicability of

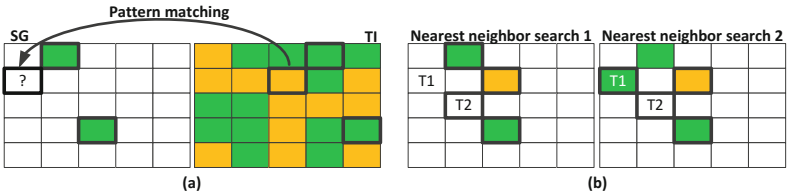

Fig. 7. (a) Matching SG point with the same pattern in the TI. (b) Two possibilities of closest neighbor pattern with probable invalidation of the closest neighbor selection if T1 finishes before T2.

singleRelaxedGet in a more read-dominant scenario. With the help of a profiling phase, we set α to 0.08. Figures 5 and 6 show similar results as the write-dominated list benchmark, amplifying the benefits of *singleRelaxedGet*. The dynamic thread assignment further reduces the rollback counts in comparison to the static assignment. In order to compare to Figure 1 the request ratio has to be increased, leading to results below 1 second for the *singleRelaxedGet*. It would hence clearly outperform the default implementation of Habanero Scala.

5.2 Simulation of the Hydraulic Subsurface

The aim of multiple-point geostatistical simulations is to simulate the hydraulic properties of the subsurface. A number of techniques has been developed for executing the simulation, the most popular of them is multiple-point geostatistics [18], which analyzes the relationship between multiple variables in several locations at a time. For its implementation, we can use the Direct Sampling simulation introduced by Mariethoz et al [19]. A simulation consists of a Training Image (TI) and a Simulation Grid (SG). The task is to fill unknown points of the simulation grid according to the known points from the training image (see Figure 7(a)). The algorithm starts with selecting a random point x to be filled in the SG. Then, it locates n closest neighbors, which are already filled. The neighbors and x are considered as a pattern, for which the algorithm searches in the TI. If found, x can be filled.

Given that in the field of hydraulic subsurface simulation the simulation grid and the training image are of a very large size, sequential processing is not efficient. Two cases of parallelization are possible: (1) the parallelization of searching within the training image, (2) the parallelization of the node filling within the simulation grid. In this paper we concentrate on the parallelization of the node filling within the SG as parallelizing the TI would only comprise independent read operations. Our implementation of the actor-based simulation considers a *main actor* that stores the SG and the TI. Consider Figure 7(b) having two points $T1$ and $T2$ to be simulated concurrently. We can see that these are close and if simulated, are likely to be part of the n closest neighbor pattern. Thus, the point that finishes first invalidates the current simulation of the other point and a synchronization mechanism is required. In our implementation, all SG

points are protected by a Scala STM Ref object, which causes a rollback once a SG point is simulated that has been visited during a nearest neighbor search. Besides the main actor we implemented a number of worker actors, either simulating the SG points (*simulation actors*) or responsible for logging the current state (*log actors*). *Simulation actors* claim a number of points to be simulated and initiate the closest neighbor search and the TI matching for each of the points at the *main actor*. We run the benchmark with 32 *worker actors*, with each *simulation actor* requesting to process 30 simulation points at once, finally sending 167 messages during the benchmark.

Researchers using the multiple-point geostatistics simulation usually validate the results manually by visualizing the simulated result. Since the simulation can take several hours, intermediate results are of interest; we can use them to see whether the simulation is on the right track. This case is especially of interest, because once the points are simulated, they do not change anymore. Therefore, these intermediate results do not require full consistency and should not interfere with the main simulation. We select these messages to be handled as read-only messages sent by *log actors*. The list benchmark considered a constant number of sum messages while here, each *log actor* sends a request to the *main actor*, repeating it upon receiving a response. In the experiments we consider 8 *log actors*. The size of the SG is 750 times 750 elements and we investigate the 15 closest neighbors for each point, finally we profiled $\alpha = 1.2$. We use the sample image provided by the MPDS [19] distribution as the TI.

As shown in Figure 8 the static thread allocation results for throughput suggest that it is possible to increase the number of processed query messages with the help of *singleRelaxedGet*, while reducing the execution time in comparison to the *atomic* case by approximately 40 seconds. The total number of rollbacks, however, remains stable. Figure 9 (left) demonstrates that the dynamic thread allocation improves the execution time of the *singleRelaxedGet* by another 20 seconds. The throughput seems to decrease in comparison to the static scenario, which is not true considering the total throughput. While the amount of work performed is in both scenarios the same, we are able to successfully process more messages in the period of second 20-50 in the dynamic scenario. This can be seen

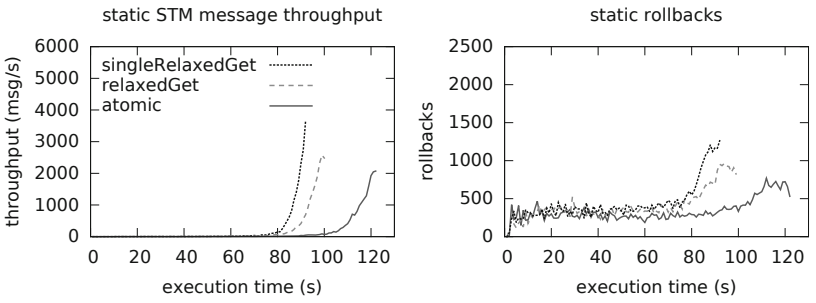

Fig. 8. Throughput (left) and rollbacks (right) of STM messages and read-only messages over time: static thread allocation

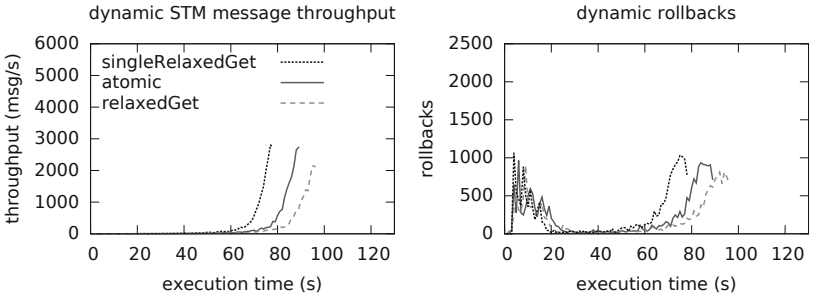

Fig. 9. Throughput (left) and rollbacks (right) of STM messages and read-only messages over time: dynamic thread allocation

by the low rollback counts shown in Figure 9 (right) for the same period. In the beginning of the execution, however, the rollback counts reflect the effects of adjusting the concurrency level. In total the dynamic scenario results in lower rollback counts than the static scenario. In all of the cases *singleRelaxedGet* in combination with the dynamic handling of STM messages is able to outperform concurrent message processing.

6 Conclusions

In this paper we introduce dynamic concurrent message processing in the actor model for high contention workloads. We propose to extract read-only messages from the transactional context if consistency is not required as well as adapt the number of threads to the workload. By handling transactional messages with a different level of concurrency we can efficiently use the remaining resources (threads in a thread pool) for processing read-only messages. We showed the applicability of our approach as well as candidates for messages processed with relaxed consistency. In a list benchmark and a real-world application we demonstrated that our approach helps reducing the execution time, while also decreasing the rollback counts. In future work we target to improve the performance by introducing a learning phase (e.g., with the help of hill-climbing) for guiding the level of concurrency.

Acknowledgement. The research leading to these results has received funding from the European Community's Seventh Framework Programme [FP7/2007-2013] 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: International Joint Conference on Artificial Intelligence (IJCAI), pp. 235–245 (1973)

2. Karmani, R.K., Shali, A., Agha, G.: Actor frameworks for the jvm platform: A comparative analysis. In: International Conference on Principles and Practice of Programming in Java (PPPJ), pp. 11–20 (2009)
3. Hayduk, Y., Sobe, A., Harmanci, D., Marlier, P., Felber, P.: Speculative concurrent processing with transactional memory in the actor model. In: Baldoni, R., Nisse, N., van Steen, M. (eds.) OPODIS 2013. LNCS, vol. 8304, pp. 160–175. Springer, Heidelberg (2013)
4. Didona, D., Felber, P., Harmanci, D., Romano, P., Schenker, J.: Identifying the optimal level of parallelism in transactional memory systems. In: Gramoli, V., Guerraoui, R. (eds.) NETYS 2013. LNCS, vol. 7853, pp. 233–247. Springer, Heidelberg (2013)
5. Heiss, H.U., Wagner, R.: Adaptive load control in transaction processing systems. Universität Karlsruhe (1991)
6. Herlihy, M., Luchangco, V., Moir, M., Scherer III, W.N.: Software transactional memory for dynamic-sized data structures. In: Annual Symposium on Principles of Distributed Computing (PODC), pp. 92–101. ACM (2003)
7. Moravan, M.J., Bobba, J., Moore, K.E., Yen, L., Hill, M.D., Liblit, B., Swift, M.M., Wood, D.A.: Supporting nested transactional memory in logtm. ACM SIGPLAN Notices 41(11), 359–370 (2006)
8. Bronson, N.G., Chafi, H., Olukotun, K.: CCSTM: A library-based STM for Scala. In: Annual Scala Workshop at Scala Days (2010)
9. Agha, G.A., Mason, I.A., Smith, S.F., Talcott, C.L.: A foundation for actor computation. Journal of Functional Programming, Cambridge University Press 7(1), 1–72 (1997)
10. Karmani, R.K., Agha, G.: Actors. In: Padua, D. (ed.) Encyclopedia of Parallel Computing, pp. 1–11. Springer (2011)
11. Haller, P.: On the integration of the actor model in mainstream technologies: A Scala perspective. In: Programming Systems, Languages and Applications based on Actors, Agents, and Decentralized Control Abstractions (AGERE), pp. 1–6 (2012)
12. Goodman, D., Khan, B., Khan, S., Luján, M., Watson, I.: Software transactional memories for scala. Journal of Parallel and Distributed Computing, Elsevier 73(2), 150–163 (2013)
13. Harris, T., Larus, J., Rajwar, R.: Transactional Memory. 2nd edn. Morgan and Claypool Publishers (2010)
14. Imam, S.M., Sarkar, V.: Integrating task parallelism with actors. In: International Conference on Object Oriented Programming Systems Languages and Applications (OOPSLA), pp. 753–772. ACM (2012)
15. Scholliers, C., Tanter, E., Meuter, W.D.: Parallel actor monitors. In: Brazilian Symposium on Programming Languages (SBLP) (2010)
16. ScalaSTM, http://nbronson.github.com/scala-stm/
17. Dragojević, A., Guerraoui, R., Kapalka, M.: Stretching transactional memory. ACM Sigplan Notices 44(6), 155–165 (2009)
18. Guardiano, F.B., Srivastava, R.M.: Multivariate geostatistics: Beyond bivariate moments. Geostatistics Tróia 1, 133–144 (1993)
19. Mariethoz, G., Renard, P., Straubhaar, J.: The Direct Sampling method to perform multiple-point geostatistical simulations. Water Resources Research, American Geophysical Union 46(11), 1–14 (2010)