Transparent Support for Partial Rollback in Software Transactional Memories

Alice Porfirio, Alessandro Pellegrini,
Pierangelo Di Sanzo, and Francesco Quaglia

DIAG, Sapienza, University of Rome

Abstract. The Software Transactional Memory (STM) paradigm has gained momentum thanks to its ability to provide synchronization transparency in concurrent applications. With this paradigm, accesses to data structures that are shared among multiple threads are carried out within transactions, which are properly handled by the STM layer with no intervention by the application code. In this article we propose an enhancement of typical STM architectures which allows supporting partial rollback of active transactions, as opposed to the typical case where a rollback of a transaction entails squashing all the already-performed work. Our partial rollback scheme is still transparent to the application programmer and has been implemented for x86-64 architectures and for the ELF format, thus being largely usable on POSIX-compliant systems hosted on top of off-the-shelf architectures. We integrated it within the TinySTM open-source library and we present experimental results for the STAMP STM benchmark run on top of a 32-core HP ProLiant server.

1 Introduction

Software Transactional Memory (STM) [1] stands as a programming paradigm tailored for the development of concurrent applications. By leveraging on atomic transactions, STM relieves the programmers from the burden of explicitly writing complex, error-prone thread synchronization code. In fact, programmers are only requested to wrap critical-section code within transactions. In STM, data conflicts are handled by means of conflict detection and management (CDMAN) algorithms, such as the ones presented in [2–6]. However, most of the proposed schemes rely on implementations where the rollback of a transaction entails squashing all the work carried out during its execution, despite the fact that part of the work can be still valid. To cope with this issue, in this article we present the design and implementation of a partial rollback scheme that is able to avoid rolling back an entire transaction, thus allowing portions of the carried-out transactional work to be saved. This directly provides a reduction on the number of machine instructions required for finalizing a given transaction instance. Such a reduction is achieved via minimal housekeeping overhead, in terms of machine instructions required for supporting the partial rollback scheme, since we exploit optimized approaches for the management of partial-log operations (e.g. of automatic variables) during the execution of transactional code blocks.

Further, with our partial rollback scheme, mutual consistency of shared data and thread-private data is automatically and transparently guaranteed. This removes the need for explicitly identifying and annotating the private data (e.g. the local variables) that need to be rollbackable, as instead is requested when relying on typical facilities offered by STM implementations [2, 3].

We achieve complete transparency towards the application-level code via an instrumentation tool, which adds partial log/undo capabilities without requiring any intervention by the programmer. We integrated and tested our proposal within the TinySTM open-source library [3], exploiting the Commit-Time-Locking (CTL) algorithm natively supported by TinySTM. Further, the instrumentation tool has been tailored to the Executable and Linkable Format (ELF) and to x86-64 processors, thus allowing supporting partial rollback operations for TinySTM-based applications run on top of POSIX compliant systems and widely diffused hardware architectures. Some experimental data highlighting performance advantages from our proposal, when compared to the traditional case of complete squashing for rolled back transactions, are also reported for the case of the STAMP STM benchmark [7] run on top of a 32-core HP ProLiant server equipped with 32GB of RAM memory and running Linux.

The remainder of this paper is structured as follows. In Section 2, related work is discussed. Details on our algorithmic extensions of CTL aimed at partial rollback are provided in Section 3. The actual implementation of the support for partial rollback within TinySTM is presented in Section 4. Performance data are provided in Section 5.

2 Related Work

Solutions aimed at the reduction of the waste of CPU-time associated with rolled back STM transactions can be found in [8–10]. The main approaches underlying these proposals entail (i) (dynamically) regulating the amount of concurrent threads to a well suited value (see, e.g., [9]), which is expected to avoid thrashing due to transaction aborts caused by excessive data conflicts, and/or (ii) a-priori sequentializing transactions when the abort rate exceeds specific thresholds (see, e.g., [10]). All these proposals are orthogonal to our one since none of them is tailored to the reduction of the waste of work via partial save of the effects of the execution of transactions.

Considering partial rollback in STM systems as the specific target, a few solutions have been proposed in [11–13]. Differently from what we present in this article, the proposal in [11] is limited to the management of partial rollback operations on shared data, thus not supporting rollback of thread-private data. As a consequence, mutual consistency between shared and private data within the partial rollback scheme is demanded from the programmer, while our approach enforces full transparency. The proposals in [12, 13] consist of an architectural specification of partial rollback supports, which has however not been implemented in any real environment, and has been evaluated only via simulation. Instead, we provide a real implementation within the TinySTM framework. Additionally, the work in [12, 13] bases partial rollback on a traditional approach where shared and private objects are marked as updated via dirty-bitmaps and are logged into per-object undo stacks. Instead, our proposal does not rely on any explicit management of dirty-bitmaps, and packs log information by clustering thread-private data via optimized log operations of the thread stack, which are based on ranges of memory addresses defining target regions for the memory write instructions executed along the transaction.

3 Partial Rollback

3.1 Target CDMAN: Commit-Time-Locking plus Read Validation

We target the CTL algorithm, as used in implementations such as TL2 [2] and TinySTM [3]. The algorithm relies on a *global version clock* (*gvc*), namely a global shared counter, which is atomically incremented (e.g. via Compare-And-Swap—CAS—operations) by any thread whenever it commits a transaction that updated shared data. Also, each (size-tunable) set of shared memory objects, such as memory words for the case of word-based STM, is associated with its own meta-data consisting of (A) a lock-bit and (B) a timestamp. This association is supported by means of hash functions taking as input the shared-object memory address. When a transaction commits, the updated *gvc* value is reflected as the new timestamp of the written objects.

Upon (re-)starting a transaction, a thread stores the current value of the *gvc* into a local variable called transaction start-timestamp (*tst*). Upon a transactional read operation from a shared object, the corresponding memory address is added to the transaction read-set, while, upon a write operation, the destination-object address and the value to be stored are both added to a transaction write-set (note that the written value is not yet stored into the actual target location). In addition, when executing a transactional read operation, it is checked in advance if the shared object has already been written by the transaction (by checking the content of the transaction write-set). In the positive case, the value stored within the write-set is returned. Otherwise, the lock-bit associated with the object is sampled to check whether it is set to 1, which means that the object is currently locked by a concurrent transaction. If the lock-bit has value 1, the reading transaction gets aborted (possibly after waiting for the lock release for a while). Otherwise, the object value and its timestamp are re-sampled along with the lock-bit in order to check if (A) the timestamp is less than or equal to the *tst* of the reading transaction, and (B) the object is not currently re-locked. If both the checks succeed, it means that no concurrent transaction has modified the object in the interval between the start of the current transaction and the actual read operation, hence the value read is still *valid*. Otherwise, the transaction gets aborted and then restarted.

Upon attempting to commit a writing transaction[1], the thread tries to acquire the locks associated with all the objects belonging to the transaction write-

[1] For read-only transactions the commit operation is unnecessary as no shared objects have to be updated.

set. This is done by attempting to set the lock-bit associated with each of these objects (e.g via CAS operations). If at least one lock acquisition fails, the transaction is aborted and restarted. Otherwise, the transaction read-set gets validated. Namely, for each object belonging to the read-set, the associated current timestamp is compared with the *tst* value in order to check if it was modified after starting the transaction. Object modifications by concurrent transactions imply that the object timestamp is greater than *tst* since it reflects updates in the *gvc*, generated by successfully committing transactions. Hence, if the timestamp of at least one object has been modified, then the transaction is aborted and restarted. Otherwise the transaction can successfully commit, thus storing object-values kept within the transaction write-set in the destination memory areas and releasing all the acquired locks.

A mechanism used in combination with this scheme is called *snapshot extension*. When the thread performs a transactional read from an object that has been updated by a concurrent transaction (which would lead to an abort) this mechanism checks if all the object values returned by previously executed transactional read operations of the transaction (if any) are still valid. If yes, the snapshot seen by the transaction is still consistent, hence the transaction is not aborted. In this case, the *tst* is updated to the value of the *gvc* sampled immediately before performing the check.

3.2 The Partial Rollback Scheme

In our partial rollback scheme, we rely on snapshot extension as a basis for managing the *tst*. However, we devise an approach where snapshot extension is exploited according to a sequential validation scheme, which is used to determine the maximum amount of transactional work that can be considered as still valid on the basis of the current state of shared data. Specifically, upon the read of an invalid object-value, the previously executed read operations are revalidated in order of their occurrence within the transaction, until all of them are found to be still valid, or validation fails for one of them. The first invalid read operation along the sequence is the restoration point for our partial rollback scheme, hence all the subsequent work performed by the transaction (if any) is squashed.

Coherency between read and write sets within the partial rollback scheme has been achieved by determining causality relations (i.e. temporal ordering) among transactional read/write operations within each transaction, which are logged as part of the representation of read/write sets. Hence, all the transactional write operations that are detected as being causally dependent on invalid read operations are also squashed from the write set.

As hinted before, we have also devised a scheme for partially rolling back thread-private data, in a consistent manner with respect to squashing operations of read and write sets. This is based on identifying the memory upper/lower bound for any log/restore operation within the stack, to be used to correctly manage thread-private data within the partial rollback scheme. On the other hand, both upper and lower bounds for these operations change over time depending on how flow-control and memory updates are materialized across different routines while the transactional code block's execution is still in progress.

Overall, beyond the already depicted handling of read/write sets (reflecting the access to shared data), operated while managing incremental snapshot extensions, our partial rollback scheme deals with the management of thread-private data according to the following actions: (A1) upon invoking `TM_begin` along the thread, the stack pointer is used to determine the upper and lower bounds of the stack region (initially empty) to be logged in case updates of thread-private data occur along the transaction. This region may be enlarged (by moving its bounds) when a write operation occurs within the stack along the transaction's execution. If the write touches data above (resp. below) the upper (resp. lower) bound, such a bound is moved to the top (resp. bottom) address of the touched memory area; (A2) upon invocation of any `TM_read` operation along the thread, a recovery image for the whole memory segment in between the current upper and lower bounds for the target stack region is created, together with a recovery image for the processor context; (A3) upon an incremental-snapshot extension operation (as depicted above), the stack/processor recovery image associated with the first no more valid `TM_read` along the sequence (as determined in A2) is restored, using an incremental-restore technique similar to the one proposed in [14]; (A4) upon successful invocation of `TM_end` along the thread, which determines actual commitment, the recovery images associated with the transaction execution path (as determined in A2) are discarded.

The above scheme, in particular in point A3, provides facilities for consistently rolling back thread-private data even in cases of complete squashing of the performed transactional work (e.g. due to invalidation of the object accessed upon the first read operation along the transactional code block). This is a relevant facility along the line of simplifying the development of application code.

Two additional points devise discussion. First, generation of stack frames restoration images is subject to a set of optimizations which we will depict later on in Section 4. Second, with the devised approach, we make update operations occurring within the stack rollbackable even if they occur via pointer-based access. Specifically, whenever any routine is started-up within the thread execution flow, if any pointer is received in input which allows the access to stack memory locations (namely stack frames) associated with other functions living along the thread, then any write access is automatically handled via the recovery images depicted above.

The only case not covered, in terms of ability to rollback, is related to updates occurring within global data that are inherently outside the control of the STM layer (e.g. non-transactional global variables). However, with the STM approach, these are typically avoided since the target is synchronization-transparent management of global (inherently shared) data structures.

4 Implementation

The logic required for handling partial rollback operations within TinySTM entails: (A) Identifying the execution points where recovery images for the thread stack need to be taken; (B) Implementing the actual log/restore logic for the stack, and combining it with the management of read/write sets. Some parts of this logic have been implemented via proper modification of TinySTM internals, while an instrumentation tool [15] has been used in order to provide complete transparency to the application layer in terms of exploitation of partial rollback capabilities. Within the whole code modification/instrumentation process, both the above targets have been achieved via instrumentation rules that allow nesting within the code a block of machine instructions to be executed right before any call from the application software to the `TM_read` function offered by the TinySTM API. This allows us to transparently take control exactly when we need to create a stack recovery image, namely before actually accessing the target transactional object in read mode. If the read operation is invalid, the additional logic included within the TinySTM layer is used in order to exploit stack restoration images for supporting partial rollback.

In order to correctly create a recovery image, which includes the current processor context right before the call to `TM_read`, the instrumentation tool has been used to transparently inline within the application ELF a functional block structured as:

```
boundaries = recompute_boundaries();
getcontext(&cpu_state);
stm_store_context_in_readset(&cpu_state);
```

With this approach, stack/processor information associated with the current state of execution of the function calling `TM_read` is sampled, with no modification of stack pointer/content and CPU image performed by the code block. On the other hand, the creation of the stack recovery image is performed by `stm_store_context_in_readset`, a function we have added to TinySTM, which performs an optimized management of stack log operations as we will explain.

Creation of Stack Recovery Images Our approach to the creation of a recovery image for the stack of the transactional thread at a given point in the execution is based on two optimizations. The first one deals with an incremental approach for the determination of the stack portions that actually need to be logged in order to correctly achieve a recovery image for the whole stack content. The second one deals with how to perform the actual copy of the memory areas required for creation of the restoration image. The two optimizations are explicitly thought to be used in combination.

As for the first optimization, a snapshot of the stack content at a given point of execution is built by combining the latter available stack log plus a log of only those portions that have been modified up to the point of interest. This is an incremental approach, that has already been exploited in literature, but typically according to a page-based logging approach (see, e.g., [16]). Instead, we incrementally build the recovery image of the stack by logging data according to arbitrary granularity, and by organizing them in a chain (realized in the read set) as described in [14]. Also, the idea underlying our incremental scheme is to determine the stack portions to be logged in such a way that they are finally contained within a single area formed by contiguous memory locations. This allows nesting the aforementioned second optimization related to efficient support for memory copies in case of adjacency of the addresses characterizing source and destination areas for the copy operation.

To achieve incremental logging, the application level ELF is again transparently instrumented by allowing the insertion of a code block before any machine-level memory-write operation (e.g. `mov` instructions) which, by analysing the current state of the processor, determines the actual virtual address to be targeted by the update, and the size of the touched memory area. In case the update falls within the stack (namely the lower address of the area to be touched is not less than the current stack pointer value), the write operation deals with the content of the stack, and needs to be made rollbackable. In this case, the interval of virtual addresses $[I_1, I_2]$ involved in the update is determined, which in turn identifies a stack portion to be logged upon the creation of the subsequent recovery image (as the one containing all the memory locations in between the addresses I_1 and I_2). We note that for some machine-level memory instructions, write access to the stack occurs by default, such as for the case of `push` and `call`.

In fact, multiple write operations can occur before the point where the creation of the stack recovery image occurs. As an example, an additional write may involve the stack portion in the interval of addresses $[I_3, I_4]$. In such a case, instead of explicitly maintaining the list of stack portions to be logged and restored, we adopt a clustering approach where we identify the actual area to be logged as the one between a minimum address value computed as $I^- = \min(I_1, I_3)$ and a maximum address value computed as $I^+ = \max(I_2, I_4)$. In other words, we always log a contiguous segment of the stack, which contains all the modified stack locations and possibly some non-modified locations. This is done so that the actual log (namely memory copy) operation can be achieved by using a single machine instruction, such as the `movs` instruction of the x86 instruction set. This is the second optimization.

As an additional note, this approach is combined with a check on the actual top-boundary of the stack such that, when considering an incrementally built recovery image associated with stack-pointer value x, any memory location within the stack with address $y < x$, possibly belonging to the previous recovery image, is logically marked as non-relevant for the incremental construction of the current image. Further, we emphasize that our approach, based on a boundary check on the actual modified region of the stack, up to the transactional read operation, copes well with common optimizations offered by modern compiling toolchains. In particular, standard compilers might decide to use, where available, the stack base registers (e.g. `ebx` on x86 architectures) as general purpose ones. This speeds up the program's execution by enlarging the set of information which the processing unit is able to maintain within its internal state. On the other hand, this makes it impossible to determine which is the current function's

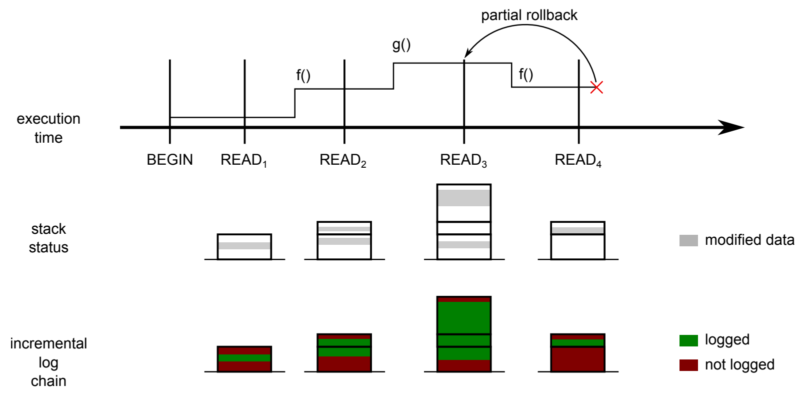

Fig. 1. Stack management within partial rollback.

stack frame. Our solution is able to cope with this scenario, since we do not need to explicitly know the base of the stack zone for any specific function.

Stack Recovery Operations As mentioned, upon the detection of an inconsistent read, instead of relying on the classical rollback scheme, we perform a partial rollback operation, which entails restarting the execution of the conflicting transaction from an intermediate point such that every operation before it is still considered valid. In order to effectively restart from within a transaction, we must restore every aspect of the execution context. If on the one hand the processor state is restored via the standard System V `setcontext` library function—using a previously stored snapshot—in order to successfully cope with automatic variables we must undo any modification concerning the stack frames of the functions living along the thread execution. In Figure 1 we show how we build partial stack logs during the execution of the transaction. In the above example, upon the execution of $READ_4$, an inconsistency is discovered, and given the failure of the snapshot extension protocol we trigger our partial rollback operation. In the example, $READ_2$ is selected as being the most recent read operation entailing a still-valid value, therefore the execution is restarted from $READ_3$. In particular, in $READ_3$'s read set we can find the portion of the stack which was modified between the execution of $READ_2$ and $READ_3$. This is restored together with the aforementioned processor context, and together with other incremental portions of the stack from previous logs, as described in [14].

5 Experimental Results

We present some experimental results achieved with the STAMP STM benchmark suite [7], specifically with `ssca2` and `kmeans` applications. The former is a transactional implementation of the Scalable Synthetic Compact Applications 2 (SSCA2) benchmark [17], where a graph kernel is used to build a directed, weighted multi-graph using adjacency arrays and auxiliary arrays. In particular,

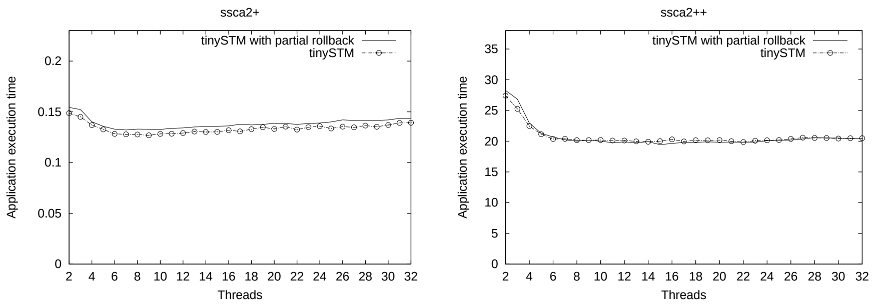

Fig. 2. Results with ssca2.

threads concurrently add nodes to the graph, and transactions are used to synchronize accesses to the adjacency arrays. Data contention in ssca2 is relatively low, making this benchmark effective for assessing the overhead produced by our partial rollback implementation with respect to the traditional rollback scheme, evaluating it mostly for the forward execution. The second one, i.e. kmeans, is a transactional implementation of a partition-based clustering method [18]. A cluster is represented by the mean value of all the objects it contains, and during the execution of this benchmark the mean points are updated by assigning each object to its nearest cluster center, based on Euclid distance. This benchmark relies on threads working on separate subsets of the data and uses transactions in order to assign portions of the workload and to store final results concerning the new centroid updates. Given the reduced amount of shared data structures being updated by transactions, in this benchmark it is more likely to incur in logical contention when a larger number of threads is used for the computation, which would allow us to better assess the benefits deriving from our partial rollback scheme. Also, we note that this is not a best case for our approach, since the amount of work saved from partial rollback is reduced, so that the overhead generated by the CPU/stack state saving/restoring is not completely amortized, which gives rise to a significative test case.

By the specification of STAMP, both the above applications can be characterized by two parameters. One is the size of the dataset, which has been changed in between '+' (indicating medium) and '++' (indicating large). The second one, particularly used for the `kmeans` benchmark, indicates the actual requirements of the transactions, in terms of, e.g., actual span of the accesses onto the dataset and, correspondingly, CPU requirements. This parameter is denoted as 'high' (indicating high demand) and 'low' (indicating reduced demand).

The execution latency that we have observed for `ssca2` while varying the number of threads (namely the number of used CPU-cores) up to 32 is shown in Figure 2, where each reported sample is the average value across 4 runs. As hinted before, this benchmark is characterized by relatively simple transactions, accessing a reduced amount of shared data, which leads the actual transactional work to be a relatively reduced percentage of the whole work carried out. This is reflected in that the actual data contention is very reduced, with the obvious out-

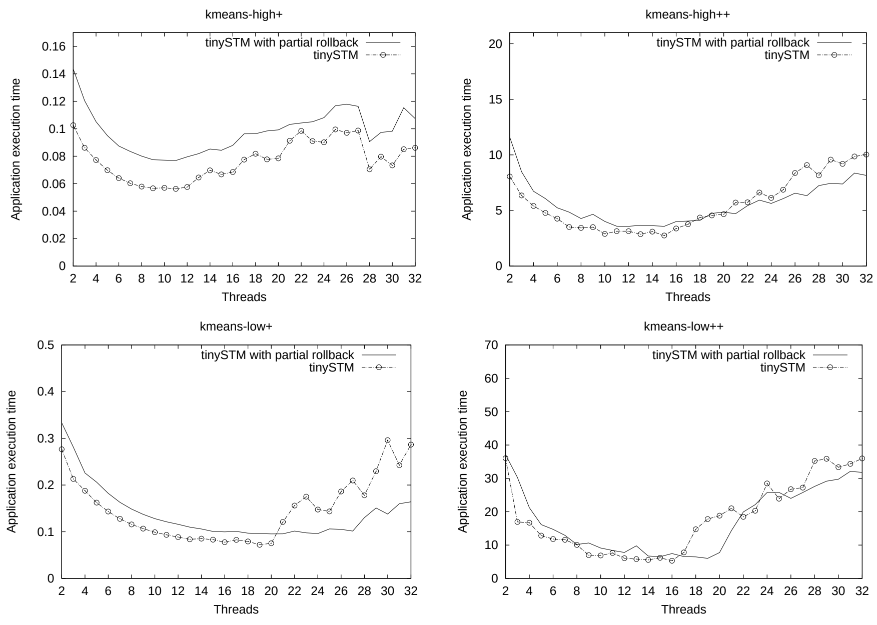

Fig. 3. Results with kmeans.

come that partial rollback schemes cannot be expected to provide performance improvements, given the almost null amount of rolled back transactions. On the other hand, for this scenario we observe a small amount of overhead from the support for partial rollback. Specifically, for the case of the '+' configuration, we observe an overhead which is on the order of 7% for most of the considered concurrency levels (namely numbers of threads). Such an overhead is further reduced when considering the '++' configuration. The increased data set leads to reduced locality, making the execution slower than '+'. This causes the overhead from the CPU-context/stack logging mechanisms for partial rollback to be less evident.

In Figure 3 the results observed for the `kmeans` benchmark are reported, with each sample expressing again the mean value over 4 runs. In this setting, the most unfavorable configuration for our partial rollback scheme is 'high+', where the transactions have higher requirements, and access a reduced dataset. This leads to scenarios of high data conflict, likely occurring in the early phase of a transaction execution (due to reduced size of the accessed dataset). This leads the partial rollback scheme to induce a non-minimal amount of logging overhead, while not allowing a significative save of work for rolling back transactions due to the fact that rollbacks typically require transactions to resume from the beginning of their execution. This phenomenon is alleviated when considering the 'high++' configuration, where the increased size of the dataset leads to scenarios where at least a portion of the performed transactional work can be successfully saved, since the dataset largeness gives rise to dynamics where the likelihood of conflicting in the early phase of transaction execution gets reduced. This leads the partial rollback scheme to exhibit increased effectiveness, especially with a higher concurrency level. In such a case, in fact, the partial rollback scheme achieves up to 20% reduction of the benchmark execution time on 32 CPU-cores.

The 'low+' configuration of kmeans gives rise to execution dynamics that are not so far from those observable for the 'high++' configuration. Specifically, here transactions conflict while accessing a reduced data set, but they exhibit reduced resource requirements. Hence, also in this case there is no bias towards conflicting in the early phase of the execution. As a consequence, the partial rollback scheme operates effectively, especially when the level of concurrency is increased. Specifically, it provides reductions of the benchmark execution latency on the order of 40% as soon as the number of used CPU-cores is greater than 23. On the other hand, for reduced concurrency levels, the impact of transaction rollback gets reduced, which leads to the scenario where the partial rollback scheme induces logging overhead that does not get compensated by revenues while partially rolling back transactions. Such an overhead is better absorbed when running the 'low++' configuration (e.g. due to the aforementioned reduced locality phenomena within the benchmark). Hence for this scenario, we observe that partial rollback provides similar performance as the one achievable by the traditional scheme when the level of concurrency is limited, while it provides some performance advantages when this level gets increased, which leads to scenarios where the transaction rollback phenomenon is more relevant, thus rendering more useful the partial save of transactional work already carried out.

6 Summary

In this article we have presented the design and implementation of a support for application-transparent partial rollback in STM, tailored to contention managers relying on lazy (i.e. at commit-time) lock acquisition and on read validation mechanisms. It has been integrated within the open source TinySTM package, and has been tested on top of a 32-core HP ProLiant machine by running applications selected from the STAMP benchmark suite. By the data, our proposal allows for performance improvements in scenarios characterized by non-minimal data contention.

References

1. Shavit, N., Touitou, D.: Software transactional memory. In: Proceedings of the 14th Annual ACM Symposium on Principles of Distributed Computing, ACM Press (August 1995)
2. Dice, D., Shalev, O., Shavit, N.: Transactional Locking II. In: Proceedings of the 20th International Symposium on Distributed Computing. (2006) 194–208
3. Felber, P., Fetzer, C., Riegel, T.: Dynamic performance tuning of word-based software transactional memory. In: Proceedings of the 13th ACM SIGPLAN Symposium on Principles and practice of parallel programming. PPoPP, ACM (2008) 237–246

4. Herlihy, M.P., Moss, J.E.B.: Transactional memory: architectural support for lock-free data structures. SIGARCH Comput. Archit. News **21** (May 1993) 289–300
5. Spear, M.F., Dalessandro, L., Marathe, V.J., Scott, M.L.: A comprehensive strategy for contention management in software transactional memory. SIGPLAN Notices **44**(4) (February 2009) 141–150
6. Lev, Y., Luchangco, V., Marathe, V.J., Moir, M., Nussbaum, D., Olszewski, M.: Anatomy of a scalable software transactional memory. In: Proceedings of the 4th ACM SIGPLAN Workshop on Transactional Computing. TRANSACT, ACM (2009)
7. Cao Minh, C., Chung, J., Kozyrakis, C., Olukotun, K.: STAMP: Stanford transactional applications for multi-processing. In: Proceedings of the IEEE International Symposium on Workload Characterization. ISWC (September 2008)
8. Rughetti, D., Di Sanzo, P., Ciciani, B., Quaglia, F.: Machine learning-based self-adjusting concurrency in software transactional memory systems. In: Proceedings of the 20th IEEE International Symposium On Modeling, Analysis and Simulation of Computer and Telecommunication Systems. MASCOTS, IEEE Computer Society (August 2012) 278–285
9. Ansari, M., Kotselidis, C., Jarvis, K., Luján, M., Kirkham, C., Watson, I.: Advanced concurrency control for transactional memory using transaction commit rate. In: Proceedings of the 14th international Euro-Par conference on Parallel Processing. Euro-Par, Springer-Verlag (2008) 719–728
10. Yoo, R.M., Lee, H.H.S.: Adaptive transaction scheduling for transactional memory systems. In: Proceedings of the twentieth annual symposium on Parallelism in algorithms and architectures. SPAA, ACM (2008) 169–178
11. Lupei, D.: A study of conflict detection in software transactional memory. Master's thesis, University of Toronto, the Netherlands (2009)
12. Gupta, M., Shyamasundar, R.K., Agarwal, S.: Article: Clustered checkpointing and partial rollbacks for reducing conflict costs in stms. International Journal of Computer Applications **1**(22) (February 2010) 80–85
13. Gupta, M., Shyamasundar, R.K., Agarwal, S.: Automatic checkpointing and partial rollback in software transaction memory (January 2012) US Patent 20110029490.
14. Pellegrini, A., Vitali, R., Quaglia, F.: Di-DyMeLoR: Logging only dirty chunks for efficient management of dynamic memory based optimistic simulation objects. In: Proceedings of the 2009 ACM/IEEE/SCS 23rd Workshop on Principles of Advanced and Distributed Simulation. PADS, IEEE Computer Society (2009) 45–53
15. Pellegrini, A.: Hijacker: Efficient static software instrumentation with applications in high performance computing (poster paper). In: Proceedings of the 2013 International Conference on High Performance Computing & Simulation. HPCS, IEEE Computer Society (July 2013)
16. Santoro, A., Quaglia, F.: Transparent state management for optimistic synchronization in the High Level Architecture. In: Proceedings of the 19th Workshop on Principles of Advanced and Distributed Simulation, IEEE Computer Society (June 2005) 171–180
17. Bader, D.A., Madduri, K.: Design and implementation of the hpcs graph analysis benchmark on symmetric multiprocessors. In: Proceedings of the 12th international conference on High Performance Computing. HiPC'05, Berlin, Heidelberg, Springer-Verlag (2005) 465–476
18. Bezdek, J.C.: Pattern Recognition with Fuzzy Objective Function Algorithms. Kluwer Academic Publishers, Norwell, MA, USA (1981)