A Relativistic Enhancement to Software Transactional Memory

Philip W. Howard
Portland State University

Jonathan Walpole
Portland State University

Abstract

Relativistic Programming is a technique that allows low overhead, linearly-scalable concurrent reads. It also allows joint access parallelism between readers and a writer. Unfortunately, it has so far been limited to a single writer so it does not scale on the write side.

Software Transactional Memory (STM) is a technique that allows programs to take advantage of disjoint access parallelism on both the read-side and write-side. Unfortunately, STM systems have a higher overhead than many other synchronization mechanisms so although STM scales, STM starts from a lower baseline.

We propose combining relativistic programming and software transactional memory in a way that gets the best of both worlds: low-overhead linearly-scalable reads that never conflict with writes and scalable disjoint access parallel writes. We argue for the correctness of our approach and present performance data that shows an actual implementation that delivers the promised performance characteristics.

1 Introduction

Transactional Memory (TM) is a popular methodology for writing parallel programs. It automatically allows parallelism and detects and resolves conflicts without requiring the complexities of fine grained locking nor non-blocking synchronization. Unfortunately, current Software Transactional Memory (STM) implementations have tended to be slow [3].

Many researchers admit that for performance and other reasons, it is often desirable to access some data non-transactionally. This is called privatization and it is difficult to do correctly [4, 10, 14, 15]. The non-transactional accesses can break the isolation guarantees of the transactional memory system. Other researchers propose using non-transactional concurrent algorithms to perform operations on concurrent data structures, but doing so in a way that the operations can be composed with transactions.[2, 8] These approaches improve performance over a pure transactional approach, but the uncontended performance is still worse than a sequential algorithm. We propose a novel method whereby read-only operations can be performed non-transactionally without violating the isolation guarantees of transactions. With our method, read-only operations proceed almost as-if they were sequential (single threaded) with only local operations required for synchronization. This allows them to perform similarly to a sequential algorithm and to scale linearly. Our method imposes only minimal overhead or complexity on transactional memory systems.

The inspiration for our approach comes from a methodology referred to as relativistic programming [9, 17, 16]. Relativistic programming allows reads to proceed independent of writes. This is made safe by requiring writers to keep the data always-consistent. We propose combining relativistic write algorithms with a software transactional memory system. This allows writes to proceed concurrently, and allows read-only accesses to be performed non-transactionally (reads proceed relativistically). Relativistic programming is described briefly in section 2. In section 3 we describe how to make an STM that is compatible with relativistic reads. In section 4 we discuss a red-black tree implementation that uses relativistic reads and transactional updates. We argue for the correctness of our implementation and present performance data showing its superiority over either a standard relativistic implementation or a standard transactional memory implementation.

2 Introduction to Relativistic Programming

Relativistic programming [9, 17, 16] is a methodology that allows readers to proceed concurrently with writers without requiring synchronization between the readers

and writers. For this to work, writers must keep the data in an always-consistent state. Read Copy-Update (RCU) is a well known relativistic technique [12, 5, 11]. RCU keeps a linked list always-consistent from a reader's point of view by requiring a writer to make copies of nodes that need to be changed. The changes are made in the copy local to the writer and are not made visible to readers until the changes are complete.

Relativistic programming requires that read-section be explicitly defined and that readers not hold references to the data structure outside these read-secitons. These read sections are used to define a grace period as follows: A grace period extends until all read sections that existed at the beginning of the grace period have terminated. Grace periods are used to insert write-side delays to guarantee the safety of certain operations. The most common use for grace periods is in memory reclamation. When a node is removed from a data structure, readers may still hold references to that node. But once a grace period has expired, all references have been released so it is safe to reclaim the memory for that node.

Figure 1 illustrates a second case where a grace period is required. Consider a linked list that stores key-value pairs, but not in sorted order. If a node is going to be moved within the list, a copy of the node is placed in the new location before the original is removed. If the new location is earlier in the list than the original, and the original is removed immediately after the copy is inserted, it is possible for readers to scan the list and never see the node. Consider a reader at node B prior to the insertion of node X'. If node X is removed prior to being scanned by that reader, the reader will not see the key X. However, if a grace period is inserted between the the insertion of X' and the removal of X, all readers are guaranteed to have seen the key X.

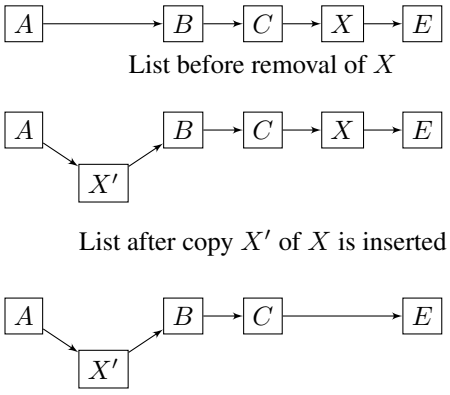

Figure 1: Linked list move example showing three steps in moving a node within a list. A grace period is required between the second and third steps.

When memory is freed, there must be a grace period before the memory is reclaimed. Most relativistic programming implementations include an rp-free primitive that arranges for an asynchronous free of the memory. This allows a writer that needs to free the memory to continue without having to wait for the grace period.

Because of the weak ordering of modern processors, and because reads are happening concurrent with writes, some relativistic writes require a memory barrier to be executed prior to the write. For example, before a new node is made visible to readers, all the updates involved in initializing the node must be visible to the readers. Relativistic programming implementations include an rp-produce primitive to enforce this ordering.

There are a number of implementations of the relativistic programming primitives mentioned above [11, 5]. These have been used to build a variety of relativistic data structures including linked lists, hash tables, and red-black trees. [12, 17, 9]. These implementations do not exhibit the complexities that accompany many non-blocking synchronization implementations. In fact, the relativistic implementations are not significantly more complicated than implementations that use coarse grained locking.

3 Relativistic Transactional Memory

The existence of relativistic readers imposes the following requirements on transactional writers:

1. The transactional memory system must be weakly atomic.

2. Updates that eventually get rolled back must not be visible to readers.

3. The transactional memory system must honor the program order of writes to memory. Normal compiler optimizations and out-of-order execution units on modern processors are allowed, but a wholesale reordering as is done by many STMs is not allowed. In particular, all writes in program order prior to an rp-produce must be visible to readers prior to the rp-produce, and all rp-produce operations must be executed in program order.

4. The transactional memory system must honor grace period delays between memory writes.

The first requirement is necessary because relativistic reads are supposed to be completely outside the transactional system. A strongly atomic transactional memory system would include the relativistic reads as part of its atomicity guarantee. This would impact read performance.

The second requirement is necessary because in our system, read-only operations happen outside the transactional memory system. As a result, read-only operations must not see any writes that will not be committed. This requirement can be met by software transactional memory systems that use a re-do log rather than update-in-place. This way, only committed updates are visible to readers.

The third requirement is necessary because relativistic writers must exercise care in the order of updates to preserve the always-consistent state of the data structure. Most software transactional memory systems give the appearance of atomicity of transactions: other threads see all of the updates or none of them. As a result, the order in which updates are made visible in memory is irrelevant. However, in our system, readers can see partially completed commits. Our implementation preserves ordering by saving every transactional write in the re-do log in program order. If the same address is written multiple times, each write will have a separate entry in the re-do log. At commit time, the entries in the re-do log are executed in order. In addition, the rp-produce primitives are marked in the re-do log so the memory barriers involved in these operations will be preserved.

The fourth requirement is necessary because the grace period is required between the time that one write is visible to readers and the time a subsequent write is visible to readers. The writes are made visible to readers at commit time, so the grace period must be honored at commit time. Our implementation records the request for a grace period in the re-do log. Commits will be delayed when this entry in the re-do log is encountered.

Our STM is derived from swissTM [6, 7]. We chose this as our starting point because it is a recent implementation that claims high performance and because it uses invisible reads and a re-do log rather than update-in-place. Our goal in making changes to swissTM was ease of implementation. We did not optimize for performance. Despite this, there was very little difference in performance between our implementation and the original (see section 4.2 for details). We made the following changes to the swissTM version dated 2009-09-10 which was downloaded on 2010-10-29:

1. Added a new log to store all updates in program order. The original log is used for loading values from within a transaction and for validating a transaction prior to commit. The new log is used at commit time to perform the memory writes.

2. Added a primitive to add a grace period to the transaction.

3. Added a primitive to force a memory barrier before particular writes as indicated by rp-produce.

4. Added a primitive to add an rp-free to the transaction.

5. Changed the commit code to make changes to memory based on the new log rather than the original log.

4 Relativistic Transactional Red Black Trees

We transactionalized our previously developed relativistic red black tree [9]. The lookup operation was left unmodified so that reads proceed outside transactions. All accesses in the insert and delete operations were transactionalized.

Since lookups proceed outside transactions, our implementation has the same low-overhead linear-scalability lookup characteristics as the original. The original implementation used mutual exclusion on the write side so there was only a single writer at a time. Since our writers are transactional, they are slower than the original, however, they scale.

4.1 Correctness

In this section, we make a brief argument for the correctness of our approach. We focus on three aspects: the validity of relativistic approaches in general, the impact of transactional updates on the relativistic reads, and the impact of our modifications on the integrity of the transactional memory system.

4.1.1 Correctness of Relativistic Programming

Relativistic programming allows different readers to see concurrent updates in different orders. Imagine two readers that are both concurrent with updates A and B. One reader might see A prior to B. The other might see B prior to A. Our claim is that this is acceptable in any system where updates are commutable [13, 9, 1]. Updates are commutable if the order of applying the updates does not affect the resulting state of the abstract type. For example, it does not matter what order elements are added to a set.

The scope of re-ordering is limited: Only updates that are concurrent with the same lookup are subject to reordering. The transactional memory system will ensure that updates are isolated from each other, but since lookups happen outside the transactional system, this isolation does not extend to lookups.

4.1.2 Transactional Impacts on Relativistic Reads

Our implementation makes a record of every store within a transaction including which require preceding memory

barriers. The record also includes when grace periods are required. The record is in program execution order. At commit time, this log is used to make the actual changes to memory. Memory barriers and grace periods are enforced. As a result, a reader can not distinguish between the standard relativistic implementation and the transactionalized relativistic implementation—the updates to memory happen in the same order and include memory barriers and grace periods in the same location in both implementations. The only difference is that in the relativistic implementation the changes to memory are made when the primitives are called; in the transactional implementation, the changes to memory are delayed until commit time. Since a reader can not see this difference, we claim that if the original implementation was valid from the reader's point of view, the transactionalized implementation is correct as well.

4.1.3 Integrity of the Transactional Memory System

The integrity of the transactional memory system is preserved for the following reasons:

1. The original swissTM system used invisible reads. As a result, read-only transactions can not invalidate a transaction that performs updates. Removing these read-only transactions from the transactional memory system has no impact on the validity of the transactions that perform updates.

2. We added additional meta-data (the log of all operations), but did not alter any of the original meta-data. Isolation and the atomicity of updates are determined and guaranteed based on the original meta-data. We did not alter any of the code that performs the validity checks nor that enforces the atomicity guarantees. As a result, our implementation will have the same conflict detection properties and atomicity guarantees as the original swissTM.

3. We changed the order and timing of updates to memory. However, from a transactional point of view, a transaction will see either all or none of the updates, so their order does not matter. The insertion of grace periods within a commit will extend the duration of the commits. However, the transactional memory system must tolerate arbitrary delays during a commit because a committing transaction may get interrupted or rescheduled by the operating system at any time.

4. The wait for grace period does not block for any other transaction. It only blocks for relativistic readers which are outside the transactional system. As a result, these delays can not lead to deadlock within the transactional system.

For these reasons, we claim that if the original transactional memory system is correct, the relativistic implementation is correct as well.

4.2 Performance

Performance data is presented for the following implementations:

RP the original relativistic programming implementation

swissTM an implementation using the original swissTM

swissRP a non-relativistic implementation using the modified swissTM

RP-STM a relativistic implementation using swissRP for updates

Performance data was collected on a four processor quad-core Intel Xeon system (16 total cores) running Linux 2.6.32.

The following process was used to collect performance data: A red-black tree of size N was initialized by inserting keys chosen at random from the range $1..2N$. A specified number of threads was started and each thread performed operations (lookup, insert, delete) on the tree using random values from the range $1..2N$. The ratio between lookups and updates is controlled to view the performance at different update rates. The ratio between inserts and deletes is 1:1 so the tree remains the same size. The test reports the number of operations performed over a one second interval.

The tests presented here are for trees with $N = 65536$ nodes. The following set-ups were used:

1. Fixed update rate and varying thread count: this setup measures scalability.

2. Fixed thread count and varying update rate: this setup measures sensitivity to update rate.

Figures 2 and 3 show the read and write scalability (100% reads and 100% writes) of RP, swissTM, swissRP, and RP-STM. The nolock lines represent the performance of an optimized single threaded implementation. There is no discernible difference between swissTM and swissRP. This indicates that the changes made to swissTM did not affect read or write performance. Read performance of RP and RP-STM are very similar. Single threaded write performance of RP is about double that of transactional memory approaches. However, as the thread count increases, the transactional memory approaches far exceed that of RP.

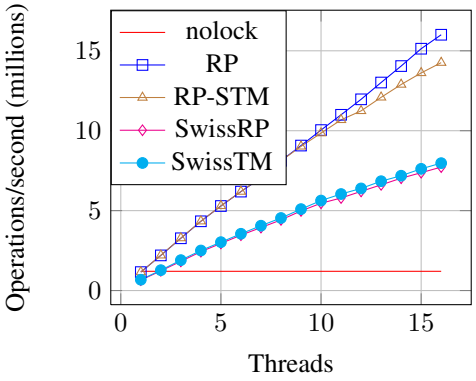

Figure 2: Read scalability of various approaches. Note that the RP and RP-STM lines are on top of each other as are the swissRP and swissTM lines.

Figure 4 shows the performance sensitivity to update rate. The data was collected with 16 threads. For low update rates, the performance of RP and RP-STM are very similar. This is because the performance is dominated by read performance. As the update rate increases, RP-STM shows marked improvement over RP. This is because the write side of RP uses a lock so its write performance is limited to no better than that of a single thread. For very high update rates, the performance of RP-STM and swissRP are similar. This data shows that RP-STM is a good choice over the full range of update rates.

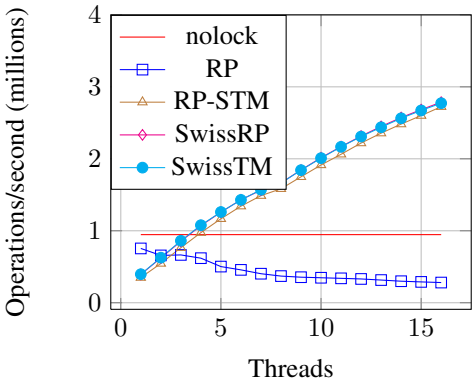

Figure 3: Write scalability of various approaches. Note that the swissRP and swissTM lines are on top of each other.

5 Future Work

Updates to a red-black tree consist of a search phase followed by an update phase. The search phase consists of finding the location within the tree where an update needs to take place. The update phase consists of making the actual changes to the tree. Since the search phase is essentially a lookup, it should be possible to perform a relativistic (non-transactional) lookup as the search phase. This would have two effects: the relativistic lookup is faster than a transactional lookup which would increase throughput, and the relativistic lookup would reduce the read-set of the transaction thereby reducing the conflict footprint. Unfortunately, a relativistic lookup included as the search phase of a transactional update introduces many of the same problems encountered with privatization [15, 10, 14, 4]. We are investigating ways to detect and avoid these problems so the full benefit of relativistic reads can be realized.

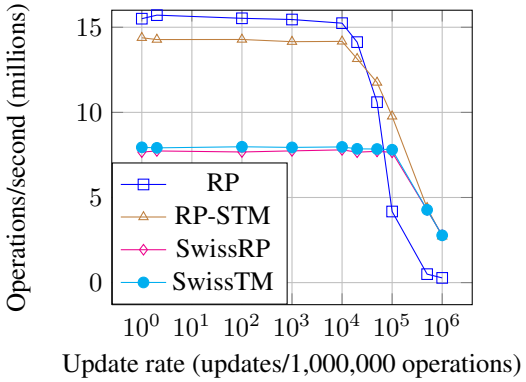

Figure 4: Performance sensitivity to update rate for 16 threads. Note: the SwissRP and SwissTM lines are almost on top of each other.

6 Conclusions

We have shown that relativistic reads can be combined with transactional updates in a way that preserves the safety and consistency of both the reads and writes. The benefits of our approach are that read-only operations do not have the high overhead of software transactional memory. We demonstrated that our system has the low overhead and linear scalable read performance of a relativistic programming approach and that our system has the write performance and scalability of a pure transactional memory system. This represents the best of both the relativistic programming and transactional memory systems.

Acknowledgments

This material is based upon work supported in part by the National Science Foundation under Grant No. CNS-0719851.

References

[1] BIRMAN, K. P. The process group approach to reliable distributed computing. *Commun. ACM 36* (December 1993), 37–53.

[2] BRONSON, N. G., CASPER, J., CHAFI, H., AND OLUKOTUN, K. Transactional predication: high-performance concurrent sets and maps for stm. In *Proceeding of the 29th ACM SIGACT-SIGOPS symposium on Principles of distributed computing* (New York, NY, USA, 2010), PODC '10, ACM, pp. 6–15.

[3] CASCAVAL, C., BLUNDELL, C., MICHAEL, M., CAIN, H. W., WU, P., CHIRAS, S., AND CHATTERJEE, S. Software transactional memory: Why is it only a research toy? *Queue 6* (September 2008), 46–58.

[4] DALESSANDRO, L., SPEAR, M. F., AND SCOTT, M. L. Norec: streamlining STM by abolishing ownership records. In *Proceedings of the 15th ACM SIGPLAN symposium on Principles and practice of parallel programming* (New York, NY, USA, 2010), PPoPP '10, ACM, pp. 67–78.

[5] DESNOYERS, M. *Low-Impact Operating System Tracing*. PhD thesis, École Polytechnique de Montréal, December 2009. [Online]. Available: `http://www.lttng.org/pub/thesis/desnoyers-dissertation-2009-12.pdf`.

[6] DRAGOJEVIC, A., FELBER, P., GRAMOLI, V., AND GUERRAOUI, R. Why STM can be more than a Research Toy. *Communications of the ACM* (2010).

[7] DRAGOJEVIĆ, A., GUERRAOUI, R., AND KAPALKA, M. Stretching transactional memory. In *Proceedings of the 2009 ACM SIGPLAN conference on Programming language design and implementation* (New York, NY, USA, 2009), PLDI '09, ACM, pp. 155–165.

[8] HERLIHY, M., AND KOSKINEN, E. Transactional boosting: a methodology for highly-concurrent transactional objects. In *Proceedings of the 13th ACM SIGPLAN Symposium on Principles and practice of parallel programming* (New York, NY, USA, 2008), PPoPP '08, ACM, pp. 207–216.

[9] HOWARD, P. W., AND WALPOLE, J. Relativistic red-black trees. Tech. Rep. 1006, Portland State University, 2011. `http://www.cs.pdx.edu/pdfs/tr1006.pdf`.

[10] MARATHE, V. J., SPEAR, M. F., AND SCOTT, M. L. Scalable techniques for transparent privatization in software transactional memory. In *Proceedings of the 2008 37th International Conference on Parallel Processing* (Washington, DC, USA, 2008), IEEE Computer Society, pp. 67–74.

[11] MCKENNEY, P. E. Kernel korner: using RCU in the Linux 2.5 kernel. *Linux J. 2003*, 114 (2003), 11.

[12] MCKENNEY, P. E. *Exploiting Deferred Destruction: An Analysis of Read-Copy-Update Techniques in Operating System Kernels*. PhD thesis, OGI School of Science and Engineering at Oregon Health and Sciences University, 2004. Available: `http://www.rdrop.com/users/paulmck/RCU/RCUdissertation.2004.07.14e1.pdf` [Viewed October 15, 2004].

[13] SHAPIRO, M., PREGUIÇA, N., BAQUERO, C., AND ZAWIRSKI, M. A comprehensive study of Convergent and Commutative Replicated Data Types. rr 7506, inria, rocq, Jan. 2011. `http://hal.archives-ouvertes.fr/inria-00555588/`.

[14] SHPEISMAN, T., MENON, V., ADL-TABATABAI, A.-R., BALENSIEFER, S., GROSSMAN, D., HUDSON, R. L., MOORE, K. F., AND SAHA, B. Enforcing isolation and ordering in STM. In *Proceedings of the 2007 ACM SIGPLAN conference on Programming language design and implementation* (New York, NY, USA, 2007), PLDI '07, ACM, pp. 78–88.

[15] SPEAR, M. F., MARATHE, V. J., DALESSANDRO, L., AND SCOTT, M. L. Privatization techniques for software transactional memory. In *Proceedings of the twenty-sixth annual ACM symposium on Principles of distributed computing* (New York, NY, USA, 2007), PODC '07, ACM, pp. 338–339.

[16] TRIPLETT, J., HOWARD, P. W., MCKENNEY, P. E., AND WALPOLE, J. Generalized construction of scalable concurrent data structures via relativistic programming. Tech. Rep. 14, Portland State University, Mar. 2011. `http://www.cs.pdx.edu/pdfs/tr1104.pdf`.

[17] TRIPLETT, J., MCKENNEY, P. E., AND WALPOLE, J. Scalable concurrent hash tables via relativistic programming. *SIGOPS Oper. Syst. Rev. 44* (August 2010), 102–109.