Transactional Memory Retry Mechanisms*

Michael F. Spear, Andrew Sveikauskas, and Michael L. Scott

Technical Report #935

Department of Computer Science, University of Rochester
{spear, asveikau, scott}@cs.rochester.edu

June 2008

Abstract

Software Transactional Memory (STM) systems, if they support condition synchronization, typically do so through a retry mechanism. Using retry, a transaction explicitly self aborts and deschedules itself when it discovers that a precondition for its operation does not hold. The underlying implementation may then track the set of locations read by the retrying transaction, and refrain from scheduling the transaction for re-execution until at least one location in the set has been modified by another transaction.

While retry is elegant and simple, the conventional implementation has several potential drawbacks that may limit both its efficiency and its generality. In this note, we present a retry mechanism based on Bloom filters that is entirely orthogonal to TM implementation. Our retry is compatible with hardware, software, and hybrid TM implementations, and has no impact on memory management or on the cache behavior of shared locations. It does, however, serialize writer transactions after their commit point when there are retrying transactions. We describe our mechanism and compare it to an optimized version of the conventional implementation.

Keywords: Transactional Memory, Condition Synchronization, Bloom Filters

1 Introduction

Software Transactional Memory (STM) systems, if they support condition synchronization, typically do so through a `retry` mechanism [6]. Using `retry`, a transaction explicitly self aborts and deschedules itself when it discovers that a precondition for its operation does not hold. The underlying implementation may then track the set of locations read by the `retry`ing transaction, and refrain from scheduling the transaction for re-execution until at least one location in the set has been modified by another transaction.

When a transaction T_R calls `retry` after reading locations $\{l_1 \dots l_r\}$, the standard implementation modifies the metadata of each location l_i to indicate that any transaction T_W that subsequently writes l_i must wake T_R. After marking all such locations, T_R double-checks its read-set validity (to avoid a timing window) then yields the processor pending wakeup by some T_W. While `retry` is elegant and simple, this implementation has several potential drawbacks that may limit both its efficiency and its generality.

Cache Interference Explicitly marking each location l_i requires exclusive access to l_i's metadata in the cache coherence protocol, in a manner analogous to "visible reader" conflict detection. Previous work suggests that the invalidation of lines in concurrent readers may have a substantial performance cost [13].

*This work was supported in part by NSF grants CNS-0411127, CNS-0615139, CCF-0702505, and CSR-0720796; and by financial support from Intel and Microsoft.

Memory Blowup A retrying transaction T_R must remove itself from metadata retry lists after waking up. When the lists are colocated with shared data (as in "object-based" STM), a transaction that deletes location l_i cannot safely reclaim the associated memory if T_R is waiting on its retry list. In a language with garbage collection, T_R prevents reclamation of only the locations it has marked. With epoch-based lazy reclamation, however [8], the most straightforward implementation would delay the end of the epoch until every retrying transaction has awoken at least once, thereby delaying the reclamation of *any* shared data. Refinements to this scheme would seem to require extensive changes to the memory management system.

Virtualization Overhead While `retry` was initially proposed for software TM, something like it is clearly needed for hardware TM (HTM) as well. In best-effort TM [2], support for retry using "visible readers" appears to necessitate a fallback to software transactions; since hardware transactions cannot sleep and resume, and cannot preserve their state (in particular, their read sets) after self-aborting, even transactions that retry after accessing only a single location must first abort and restart in software mode, then abort, update metadata to reflect the (software-mode) reads, and yield the processor.

Synchronization Overhead When a transaction wakes from sleeping, it must remove itself from retry lists. In nonblocking STM [5, 7, 9], this necessitates a nonblocking implementation of the per-object retry list. Alternatively, the STM may bound the number of retrying transactions, in which case an optimized bit-vector retry implementation, similar to the RSTM visible reader mechanism [10], is appropriate.

In this technical report, we present a retry mechanism based on Bloom filters [1] that is entirely orthogonal to TM implementation. Our retry is compatible with hardware, software, and hybrid TM implementations, and has no impact on memory management or on the cache behavior of shared locations. It does, however, serialize writer transactions after their commit point when there are retrying transactions. In Section 2, we describe our Bloom retry mechanism. We then compare Bloom retry to the default implementation and a visible-reader style alternative in Section 3. Lastly, in Section 4 we draw conclusions and outline future research directions.

2 Retry with Bloom Filters

Our retry mechanism maintains a global set of retrying transactions, each represented by a Bloom filter and a handle object (such as a semaphore) for wakeup. The set implementation is orthogonal to the correctness of the algorithm; when the underlying STM is nonblocking, a nonblocking set can be used. Transactions interact with the set of retrying transactions when retrying and immediately after committing writes.

2.1 The Retry Operation

A transaction T_R wishing to retry constructs a Bloom filter representing the locations it has read and adds this filter, together with its handle, to the global set. In STMs that use ownership records [3, 4, 7, 9, 10], the address of the ownership record can be used to approximate a range of locations. When ownership records are not used by the underlying STM [11, 14], the individual addresses of the read set must be used. In workloads where `retry` is common, the $O(r)$ overhead of building the filter can be spread across the transaction's execution.

After adding to the set, T_R re-validates its read set. If validation fails, T_R removes its filter from the global set and restarts; otherwise, it yields the CPU and awaits notification that it can resume. The validation step, which is also required with visible-reader style `retry`, avoids a window in which the only transaction that can signal wakeup commits between when T_R calls `retry` and when it ultimately posts its filter. When T_R wakes, it removes its filter from the global set and restarts.

Immediately prior to yielding, T_R can safely update its memory management epoch, thus avoiding memory blow-up. Concurrent threads can reclaim shared memory even when it lies in T_R's read set.

Transactions in a hardware or hybrid TM can interact with this mechanism without substantial modification: when `retry` is called, the hardware transaction can pass its read set to a software handler that creates the filter and adds it to the global list.

2.2 The Commit Operation

At commit time, a writing transaction T_W must wake any transaction whose read set intersects T_W's write set. It does so by inspecting the Bloom filters in the global set. When the set is empty, the overhead is a small constant. In contrast, visible-reader retry requires logging during execution, with constant overhead per location written even in the absence of retrying transactions. Bloom filter operations can be performed after all writes are physically committed and all locks are released, so that overhead during the wakeup phase does not delay concurrent transactions.

When the global set is nonempty, Bloom filter retry may be less efficient than visible readers due to (a) the need to inspect all filters and (b) false positives within a given filter. With visible readers, T_W interacts only with retrying transactions that have explicitly marked the metadata of locations written by T_W. (Of course, this may still entail false conflicts in "word-based" systems, when ownership records are located via hashing.)

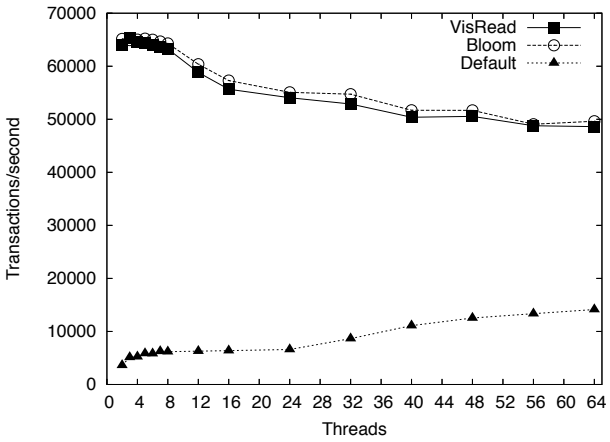
To support Bloom-based wakeup, hardware and hybrid TM systems require a single modification: after the transaction commits, it must pass its write set to a software handler. To keep the overhead of this operation low, it is possible to encode the write set as a Bloom filter, and then use a handler that intersects filters, rather than probing individual locations. In this manner, hardware transactions will cause more spurious wakeups, but with overhead linear only in the number of retrying transactions (instead of overhead linear in the number of writes and the number of retrying transactions). STMs may also opt to use this intersect-only mechanism at commit time.

2.3 An Alternative Implementation with Polling

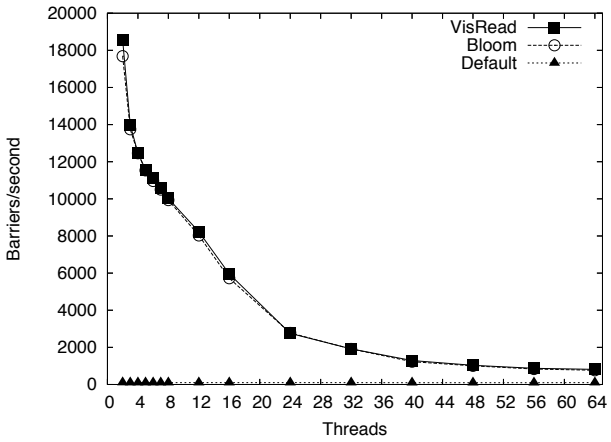
In STMs that broadcast the write sets of committed transactions, such as RingSTM [14], Bloom retry can be implemented via a polling operation that requires no additional global metadata. A retrying transaction need only create a filter of its reads and then compare that filter against published write sets as transactions commit. To reduce the overhead of polling, retrying transactions may choose to sleep for brief periods before checking for new transactions, or to use alert-on-update [15] if available.

Such a polling mechanism is also possible without Bloom filters: a retrying transaction can simply release all locations it has written, and then continuously check its read set validity, restarting the transaction once any read becomes invalid. In STMs such as TL2 [3], where timestamps and ownership records are used, the storage requirements of such a mechanism are minimal.

With polling, writers incur no overhead, as they are not required to identify descheduled transactions whose reads they invalidate. However, the polling operation consumes CPU resources, does not virtualize, and may allow notifications to be missed (for example, during ring overflow in RingSTM). When a missed notification is possible, the retrying transaction can conservatively resume. While there may exist workloads in which polling is advantageous (such as frequent retry transactions with very infrequent wakeup), we do not in general expect it to match the performance of Bloom retry with a global set of retrying transactions.



(a) Transactions updating a data structure in phases



(b) Throughput for a barrier implemented as two transactions

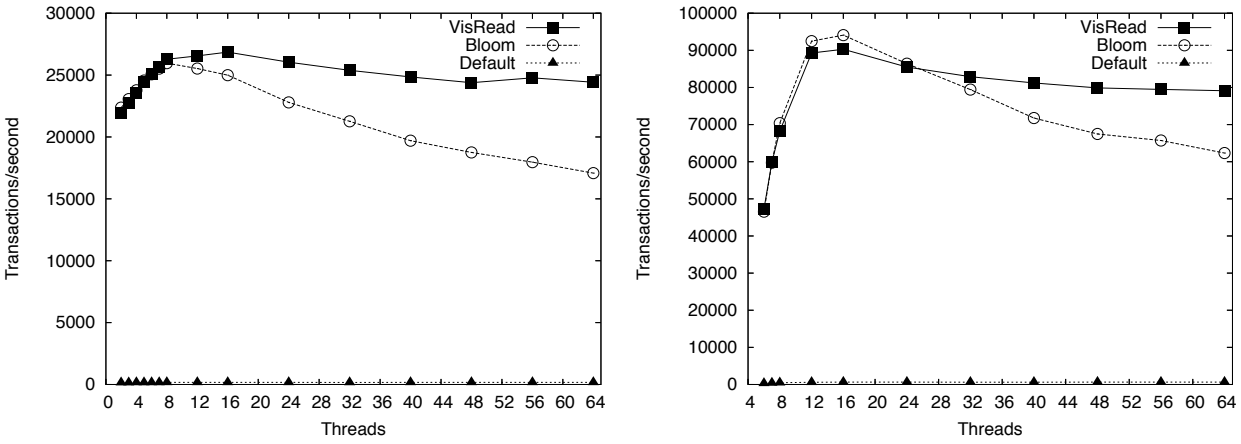
Figure 1: Comparison of Bloom-based retry to optimized visible-reader retry in RSTM. Up to 64 retry transactions are supported via per-object visible reader bitmaps. Bloom filters are configured with 1024 bits and 3 hash functions.

3 Evaluation

We implemented Bloom-based retry as an extension to the RSTM system [16]. In this section we present results on an 8-core (32-thread), 1.0 GHz Sun T1000 (Niagara) chip multiprocessor running Solaris 10. All benchmarks are written in C++ and compiled with g++ version 4.1.1 using –O3 optimizations. Data points are the average of five 5-second trials. Bloom filters are configured with 1024 bits and 3 hash functions.

Systems Compared We contrast Bloom-based retry with two other implementations. The default RSTM `retry` uses oblivious polling. When retry is called, the transaction aborts, calls `usleep(50)` ($50\mu s$ is the shortest observable yield interval supported by the OS), and then restarts. We also compare against a "visible-reader" retry implementation that marks the metadata of objects in a retrying transaction's read set and then yields the CPU pending semaphore-based wakeup. Like RSTM visible readers, the retry marks are implemented as a bitmap in the object header. This implementation requires no dynamic memory allocation, supports simple removal upon wakeup, and facilitates low-cost filtering of duplicates when a retryer has marked multiple objects in a committing transaction's write set. When the maximum number of transactions is small (64 in our experiments) and statically known, we believe this technique provides a lower bound on the overhead of visible reader-style retry.

Transactions updating a data structure in phases: In the "phase" experiment of Figure 1(a), two groups of threads take turns interacting with a list. The first group adds elements to the list to make it contain the set $\{0 \ldots 63\}$, at which point the second group wakes and removes all entries from the list in order. The exact contents of the list determine which thread group is active, and thus retrying transactions often have large read sets. The benchmark admits many spurious wakeups (whenever the list changes without reaching a full or empty state), and the number of threads woken is linear in the number of total threads in the experiment. Neither mechanism should have a significant advantage, but there is considerable memory management since every transaction allocates or frees a list node. In the visible-reader code, RSTM's epoch-based memory management results in a slight additional overhead since all memory reclamation is deferred whenever a transaction sleeps. Since Bloom retry does not impede epoch-based reclamation, there is less memory blowup, which translates to slightly less run-time overhead for the RSTM allocator.



(a) Single Token Passed randomly

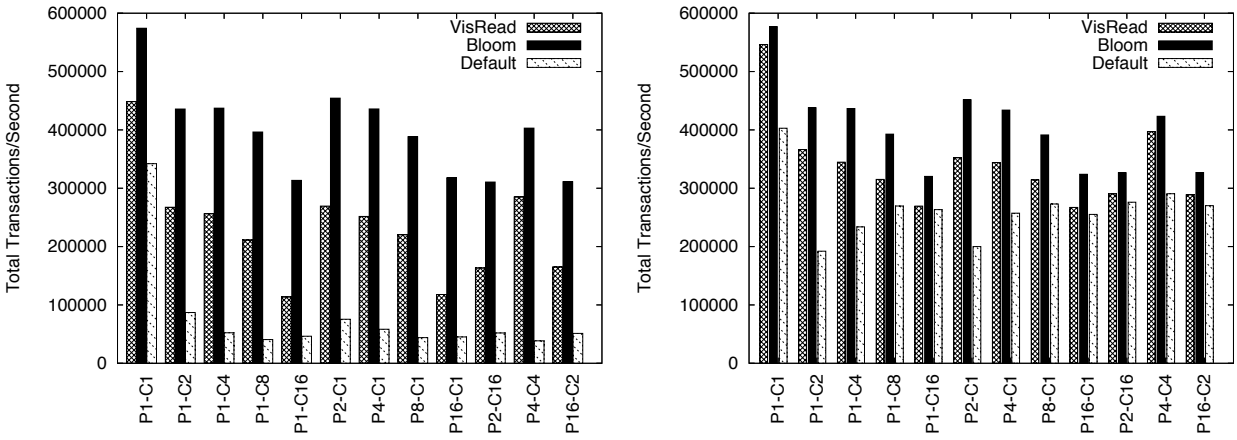
(b) Multiple tokens passed in-order

Figure 2: Bloom-based and visible-reader retry used to pass tokens between threads. When the thread count is close to the token count, the final validation after calling `retry()` results in an abort, rather than a semaphore post.

Non-composable Barrier: The "barrier" experiment of Figure 1(b) employs a non-composable barrier patterned after the experiments of Smaragdakis et al. [12]. The barrier consists of two transactions: in the first, each thread increments a counter; in the second, each thread calls retry until the counter reaches a threshold. The benchmark results in a pathological $O(n^2)$ overhead from spurious wakeups, since each of n counter increment transactions causes $O(n)$ threads to wake in the body of the second transaction. (This is a stress test, not a reasonable way to implement a barrier.) Performance of default (`usleep`) retry is substantially worse than that of the other two methods, but otherwise the overhead of waking transactions dominates, and is equal for visible readers and Bloom retry.

Token Passing: In Figure 2(a), transactions wait to receive a token that is passed randomly between threads. Similarly, in Figure 2(b), transactions use `retry` to wait for one of four tokens that are passed between threads in a fixed order. In both cases, read sets are small, and for low thread levels, visible-reader and Bloom transactions abort and restart in their final validation after calling `retry`. Consequently, they do not yield the CPU, resulting in substantial improvement over the default implementation. At higher thread counts, the likelihood of a transaction yielding rather than aborting during its final validation increases. Up until this point, Bloom performs best because its overhead is lowest in the absence of retryers. However, at high thread levels, transactions do yield the CPU, and visible readers perform best since they do not scan a list of all sleeping transactions at commit time.

Bounded Buffer: We also consider a bounded buffer with variable producer and consumer counts. Our buffer maintains an initialization field in each bucket to prevent conflicts over centralized counters. In Figure 3 we see that across a variety of producer/consumer combinations, Bloom filters outperform visible readers. Since all retrying transactions (either producers or consumers) retry on the same location, visible readers have higher cache contention since they require two atomic read-modify-write operations on that location by each retryer. Even on the Niagara, with its shared L2 cache and write-through policy, the cost is noticable. As the buffer size increases, the likelihood of imbalance decreases, and the incidence of retry decreases. Thus with 1024 entries, retry is rare enough that the default implementation rarely calls `usleep`, and thus ceases to perform an order of magnitude worse. Even so, retry is frequent enough that the benefits of Bloom retry over both visible readers and the default mechanism remain.



(a) Bounded Buffer with 64 entries

(b) Bounded buffer with 1024 entries

Figure 3: Bounded buffer experiments with varying producer (P) and consumer (C) counts.

4 Conclusions and Future Work

Our results are clearly mixed: both Bloom-filter and (optimized) visible-reader retry outperform the default `usleep`-based polling, but neither consistently outperforms the other. Since our visible reader retry imposes a limit on the number of retrying transactions, we are hopeful that Bloom retry may ultimitely offer higher performance, especially for CPUs that provide vector instructions to accelerate Bloom operations.

As we continue to experiment with new transactional workloads, we hope to clarify the tradeoff between retry mechanisms, or to identify application characteristics that favor a particular implementation. Certainly Bloom retry is more orthogonal to the rest of the STM system than visible readers are; it can easily be applied to hardware and hybrid TMs, as well as TMs that do not use ownership records.

Our implementations and microbenchmarks are available as a patch to RSTM, available from the project website [16].

References

[1] B. H. Bloom. Space/Time Trade-offs in Hash Coding with Allowable Errors. *Communications of the ACM*, 13(7):422–426, 1970.

[2] P. Damron, A. Fedorova, Y. Lev, V. Luchangco, M. Moir, and D. Nussbaum. Hybrid Transactional Memory. In *Proceedings of the 12th International Conference on Architectural Support for Programming Languages and Operating Systems*, San Jose, CA, Oct. 2006.

[3] D. Dice, O. Shalev, and N. Shavit. Transactional Locking II. In *Proceedings of the 20th International Symposium on Distributed Computing*, Stockholm, Sweden, Sept. 2006.

[4] P. Felber, C. Fetzer, U. Müller, T. Riegel, M. Süßkraut, and H. Sturzrehm. Transactifying Applications using an Open Compiler Framework. In *Proceedings of the 2nd ACM SIGPLAN Workshop on Transactional Computing*, Portland, OR, Aug. 2007.

[5] K. Fraser. Practical Lock-Freedom. Technical Report UCAM-CL-TR-579, Cambridge University Computer Laboratory, Feb. 2004.

[6] T. Harris, S. Marlow, S. Peyton Jones, and M. Herlihy. Composable Memory Transactions. In *Proceedings of the 10th ACM SIGPLAN 2006 Symposium on Principles and Practice of Parallel Programming*, pages 48–60, Chicago, IL, June 2005.

[7] M. P. Herlihy, V. Luchangco, M. Moir, and W. N. Scherer III. Software Transactional Memory for Dynamic-sized Data Structures. In *Proceedings of the 22nd Annual ACM Symposium on Principles of Distributed Computing*, Boston, MA, July 2003.

[8] R. L. Hudson, B. Saha, A.-R. Adl-Tabatabai, and B. Hertzberg. A Scalable Transactional Memory Allocator. In *Proceedings of the 2006 International Symposium on Memory Management*, Ottawa, ON, Canada, June 2006.

[9] V. J. Marathe, W. N. Scherer III, and M. L. Scott. Adaptive Software Transactional Memory. In *Proceedings of the 19th International Symposium on Distributed Computing*, Cracow, Poland, Sept. 2005.

[10] V. J. Marathe, M. F. Spear, C. Heriot, A. Acharya, D. Eisenstat, W. N. Scherer III, and M. L. Scott. Lowering the Overhead of Nonblocking Software Transactional Memory. In *Proceedings of the 1st ACM SIGPLAN Workshop on Languages, Compilers, and Hardware Support for Transactional Computing*, June 2006.

[11] M. Olszewski, J. Cutler, and J. G. Steffan. JudoSTM: A Dynamic Binary-Rewriting Approach to Software Transactional Memory. In *PACT '07: Proceedings of the 16th International Conference on Parallel Architecture and Compilation Techniques (PACT 2007)*, Brasov, Romania, Sept. 2007.

[12] Y. Smaragdakis, A. Kay, R. Behrends, and M. Young. Transactions with Isolation and Cooperation. In *OOPSLA '07: Proceedings of the 22nd Annual ACM SIGPLAN Conference on Object Oriented Programming Systems and Applications*, Montreal, Quebec, Canada, Oct. 2007.

[13] M. F. Spear, V. J. Marathe, W. N. Scherer III, and M. L. Scott. Conflict Detection and Validation Strategies for Software Transactional Memory. In *Proceedings of the 20th International Symposium on Distributed Computing*, Stockholm, Sweden, Sept. 2006.

[14] M. F. Spear, M. M. Michael, and C. von Praun. RingSTM: Scalable Transactions with a Single Atomic Instruction. In *Proceedings of the 20th ACM Symposium on Parallelism in Algorithms and Architectures*, Munich, Germany, June 2008.

[15] M. F. Spear, A. Shriraman, L. Dalessandro, S. Dwarkadas, and M. L. Scott. Nonblocking Transactions Without Indirection Using Alert-on-Update. In *Proceedings of the 19th ACM Symposium on Parallelism in Algorithms and Architectures*, San Diego, CA, June 2007.

[16] Univ. of Rochester. Rochester Software Transactional Memory. *http://www.cs.rochester.edu/research/synchronization/rstm/*.