Inside Time-based Software Transactional Memory *

Rui Zhang, Zoran Budimlić, William N. Scherer III
Department of Computer Science, Rice University
{ruizhang, zoran, scherer}@cs.rice.edu

ABSTRACT

We present a comprehensive analysis and experimental evaluation of time-based validation techniques for Software Transactional Memory (STM). Time-based validation techniques emerge recently as an effective way to reduce the validation overhead for STM systems. In a time-based strategy, information based on global time enables the system to avoid a full validation pass in many cases where it can quickly prove that no consistency violation is possible given the time information for the current transaction and the object it is attempting to open. We show that none of the current time-based strategies offers the best performance across various applications and thread counts. We also show an adaptive technique which has the potential to achieve an overall best performance based on time information and show some preliminary results we have.

1. INTRODUCTION

The "gigahertz race" in microprocessor manufacturing is over due to the heat dissipation problem [18]. Because of the limited instruction level parallelism, technologies that attempt to utilize it are approaching the point of diminishing returns. In order to continue the improvements in processor performance, and to effectively use the on-chip transistors whose count is still increasing according to Moore's law [17], major processor vendors have resorted to implementing multiple computing units, or "cores", on a single processor chip. The move from single core processors to multi-core processors on various computing platforms such as PC, PDA, cell phone, etc. projects a future where parallelism prevails everywhere. Unfortunately, none of the mainstream programming languages, compilers or programming practices is ready for this change yet.

Parallel programs are notoriously difficult to write and debug. Developing an efficient and correct concurrent program requires well trained programmers and large amounts of effort. Besides the complexity inherent to parallel applications, difficulties also arise from the programming model used for parallel programming. For shared memory systems, lock-based synchronization is currently the most popular synchronization mechanism. Unfortunately, locks have some well known issues: deadlock, priority inversion and convoying [11], to name the few. To design lock-based parallel programs, developers have to exercise extreme care in ensuring program correctness and a satisfactory level of performance. This process is error-prone; moreover, the concurrent programming bugs are notoriously difficult to find and remove.

The need is clear for a new programming model that fully utilizes these parallel architectures but also allows easier programming, maintenance and debugging. Transactional Memory (TM) [11] is a promising technique for this problem because: a) it provides a higher level synchronization abstraction than locks; b) the underlying synchronization is managed by a runtime system transparent to the programmer; and c) deadlocks, priority inversions and convoying cannot happen in a TM. Unfortunately, the performance of existing transactional memory systems is far from satisfactory. Compared with lock based counterparts, software based on transactional memory is commonly slower by an order of magnitude or more. The factors causing the poor performance of transactional memory systems include bookkeeping and validation [26]. In order to reduce the associated overhead, researchers have proposed different strategies. Among them, strategies based on global time stamps emerge recently as an effective way to reduce such overhead [19, 5, 26]. In this report we present a comprehensive analysis and evaluation of the strengths and weaknesses of time-based software transactional memory systems. We show that though time-based strategies improve over the incremental strategy to some extent, none of existing strategies is a clear winner across various benchmarks and contention degrees. We also exhibit some ongoing research results on improving these algorithms at Rice University .

The next section discusses related work that has been done in software transactional memory and in validation techniques for it. Section 3 describes the validation strategies and their design variations; Section 4 presents throughput comparison of different implementation variations. Finally, we conclude the report and suggest some directions for future research.

2. RELATED WORK

2.1 Transactional Memory

The lock-based programming model requires programmers not only to identify the parallelism in an application, but also to provide a complete lock/unlock solution for correctness and performance. In contrast, transactional memory provides a straightforward synchronization model. Instead of specifying explicitly where and how to acquire and release locks, the programmer just needs to specify which blocks of code are to be executed atomically. Synchronization is handled automatically by the underlying transactional memory system.

*This work was supported by grant R62710-792 from Microsoft Corporation.

This higher level of abstraction separates the concerns of functionality and correctness. It alleviates the responsibility of a programmer for concurrent program synchronization. It also allows for the design of an intelligent transactional memory system that executes transactions efficiently by utilizing and balancing a variety of runtime information.

While transaction processing has been successfully used in the database community for years, there are several distinct characteristics which make memory transactions different from database transactions. First, transactions in transactional memory systems reside in a much faster storage media than database systems. Second, transactions in transactional memory systems face a more complex environment than transactions in database systems. Third, transactions in transactional memory systems do not need to face durability requirement which is critical to database systems.

Herlihy and Moss [11] proposed the first transactional memory implementation, based on extending a hardware cache coherence protocol. Different flavors of transactional memory systems have been implemented since it. They include hardware transactional memory, software transactional memory and hybrid transactional memory.

Hardware Transactional Memory. Hardware transactional memory (HTM) [11, 27] is a hardware system supporting the semantics of memory transactions. HTM introduces new hardware features to achieve the best performance. HTM is usually faster than software implementation, but the transaction size is limited by the available hardware resources. Moreover it requires some hardware features that do not exist in today's off-the-shelf systems.

Software Transactional Memory. Software transactional memory (STM) is implemented using atomicity primitives such as compare-and-swap (CAS) or locks. STM has the advantages of not requiring additional hardware support from existing hardware and being able to support unbounded transaction size [24, 9, 12, 10, 8, 1, 7, 2, 6]. Shavit and Touitou [24] proposed the first implementation of software transactional memory. Herlihy, et al. [10] proposed the first dynamic software transactional memory.

Software transactional memory systems can be categorized into groups by several criteria. Based on the granularity of the data that the transactions operate on they can be divided into *object based* implementation such as RSTM [16], *cache line* based implementation [21], and *word based* implementation [8]. Based on how the ownership of an object is acquired: *eager acquire* tries to acquire ownership of the needed object/memory location at the opening phase [8], while *lazy acquire* delays the acquisition until the commit phase [10]. Based on how read-only objects/words are treated: *visible reads* expose read-only accesses to other transactions while *invisible reads* keep read-only accesses hidden from other competing transactions [16]. Finally, based on how shared data is updated, STM systems can be divided into *direct update* systems that make changes to the shared memory directly and roll back to the original state when the transaction is aborted, and *deferred update* systems that make changes in a cached copy first and then make the changes visible to the outside world in one atomic operation. Direct update leads to fast commits and slow aborts, while deferred updates have slow commits and fast aborts. Therefore direct update is more suitable for the situation where abort only happens rarely compared with commit. Another distinction worth pointing out is that in current STM implementations the semantics of direct updates is different from the serializability semantics [14].

Hybrid Transactional Memory. Hybrid transactional memory combines hardware support and software implementation. It combines the faster synchronization of hardware with the unlimited transaction size of software systems [25, 4, 13].

2.2 Existing Validation Strategies

Validation is a technique to keep a transaction from observing an inconsistent state in shared data and from performing some irreversible operation that results in corrupt program state. To avoid entering an inconsistent state, a transaction needs to validate the objects it has read (including the writes that are lazily acquired) at appropriate times in program execution. When the inconsistency does not lead to some harmful operation, *toleration* is another feasible solution [14]; however, it is outside of the scope of this report.

Validation is closely related to conflict detection. If the system is designed to detect all possible write/write, read/write, and write/read conflicts, there is no need for validation, since every attempt by a transaction to modify an object opened by another transaction will result in abortion of one or the other transaction. Unfortunately, such system would need to track all reads and writes, using expensive atomic operation such as compare-and-swap. This would incur performance penalties that completely eliminate the benefit of early conflict detection, and potential reduced concurrency due to the inability to execute two transactions concurrently beyond some conflicting point.

Different conflict detection strategies have been developed to reduce the bookkeeping overhead and to increase concurrency. For instance, *invisible reads* hide the read of an object from concurrent transactions which are thereafter able to modify the object. This makes write after read no longer a conflict. *Lazy writes* hide the write from other transactions and allows modifications to them by other transactions, making read after write and write after write no longer a conflict. Scott [22] proposes a mixed invalidation strategy. His observation is that two transactions having write/write conflict have no chance to both commit, but two transactions with read/write or write/read have the chance to both commit if the transaction with the read commits first.

Relaxed conflict detection reduces the work of bookkeeping and increases potential concurrency, but it creates the possibility of observing an inconsistent state. This can be handled either by rolling back partial updates or by inconsistency toleration.

Incremental validation [10] is a validation strategy used in some software transactional memory systems. It validates all past reads and lazy writes every time the transaction opens a new object. When any change in the past is detected, the validation fails. This strategy guarantees a consistent state but imposes a substantial overhead, since it is essentially a $O(n^2)$ operation where n is the number of objects opened in a transaction. In a sorted linked list implementation for example, the overhead of incremental validation can occupy around 70% of the total execution time for a list with 256 distinct elements.

Some research has been done to reduce the overhead of validation. Lev and Moir [15] propose a global conflict counter combined with per-object read counter to reduce the validation overhead. In their method, readers increment the read counters at open and decrement them when they commit. Writers increment the conflict counter when they acquire an object whose read counter is nonzero. Therefore a reader can skip incremental validation if the global conflict counter has not been changed since it was last checked. Unfortunately, this strategy forces every read to modify the shared data (the per-object counters) and consequently exhibits generally weak performance [26].

3. TIME-BASED STM

Two major issues in software transactional memory implementation are conflict detection and validation. Conflict detection discovers the cases where two transactions perform two operations on an object at the same time and at least one of the operations is a write. Validation is a technique that detects harmful inconsistencies in the global memory state.

Two transactions cannot execute in parallel if they have a conflict. If there is a conflict, transactions must linearize one after the other. If transaction A writes an object O, and transaction B reads the old value of O, then A and B can execute in parallel only if B linearizes before A [14]. This imposes a partial ordering on the transactions executing in parallel. Concurrent execution cannot continue if there is a cycle in this ordering; one transaction must be aborted to break the cycle. One could imagine using cycle detection analysis [23] to guarantee sequential consistency, but it is prohibitively expensive.

Moreover, it is very difficult (and often impossible) to reason at compile time about the precise dependencies and ordering of operations for programs operating on the dynamic data structures for which transactional memory is often used, so a lot of research is focuses on runtime scheme that can efficiently order transactions and reduce overhead.

3.1 Time-based Validation Strategies

Time-based validation strategies have at least two advantages compared with previous validation strategies. First, they are able to use a non-closed memory system. Memory used in transactions needs to be recycled to be used outside of transactions. Usually this resorts to either garbage collected languages like Java or specially designed memory systems for languages like C/C++. Time-based validation strategies reason about the consistency using time which is not relevant to the memory system and is therefore not confined to any specific memory systems.

Second, time-based algorithms can guarantee the consistency of the past reads by simply checking whether the time stamp of the object being opened is in the transaction's validity range. This greatly reduces the overhead introduced by incremental validation.

Spear, et al.'s [26] *Global Commit Counter* heuristic does not perform a validation if no writes were committed in the whole system since the last object was opened. Since if the transaction knows that nobody modified anything in the whole system there is no need to check this fact for each individual object. This scheme can avoid many unnecessary validations, leading to performance improvements for situations where only a few threads are competing for a shared data structure and writes happen rarely. Unfortunately, this is still very conservative since there are many cases where a write *has* appeared in the system, but that write does not affect the consistency of the current transaction.

Riegel, et al.'s [19] *Lazy Snapshot* Algorithm maintains different versions for each shared object and uses the global time stamp information to guarantee the view observed by a transaction is consistent. In their algorithm each shared object keeps multiple versions that a transaction can choose from to satisfy consistency. In cases necessary, the validity range can be extended through validating the read object set.

Dice, et al.'s [5] *transactional locking II* (TL II) algorithm is another time-based STM implementation. Their algorithm also uses a global time stamp counter. TL II does not maintain multiple versions of an shared object. A transaction acquires the global time stamp only once at the beginning of the transaction. One advantage of this algorithm is that it can eliminate the bookkeeping overhead of read only transactions since it never validates opened objects. This makes their algorithm especially attractive to cases where contention is low and most transactions commit successfully.

In order to compare these three algorithms fairly, we implemented them based on the same STM. Our implementation associates a timestamp TS with each shared object. TS indicates the time when the object is last modified. Closely related to timestamps are *candidate linearization points* (CLP). CLP is the approximate upper bound of the validity range in lazy snapshot algorithm. Each transaction keeps its own candidate linearization point, CLP, which indicates the last time this transaction has observed a consistent state. This data allows the transaction to explicitly know where a potential linearization point lays relative to all the writes that have happened in the system. Having this knowledge makes the transaction able to quickly decide to skip validation if it knows that the object it is trying to open has not been modified since its current CLP.

We will assume that there exists a global timestamp counter TSC that reflects the current time relative to the start of the program and that can be read concurrently by all the transactions in the system. The implementation details of such a counter using hardware or software techniques are discussed in Section 3.3.

When a transaction starts, its candidate linearization point (CLP) is initialized to the beginning of the transaction. When opening an object, the transaction compares its CLP with the timestamp of the object. If the object is older than CLP, it is opened without validation. If the object is younger than CLP, validation is needed.

CLP is updated each time a transaction opens a younger object and successfully validates past reads. In our implementation, the candidate linearization point is updated conservatively to reflect the time immediately before the validation.

Figure 1 and figure 2 illustrate respectively the major steps of the validation strategy in lazy snapshot algorithm and transactional locking II algorithm. In figure 1, if the transaction linearizes after the creation time of the object it is opening, there is no need for

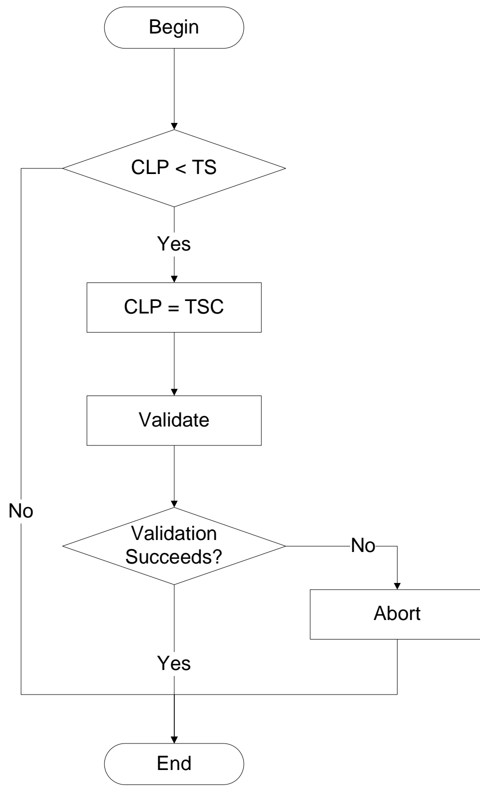

Figure 1: Lazy Snapshot

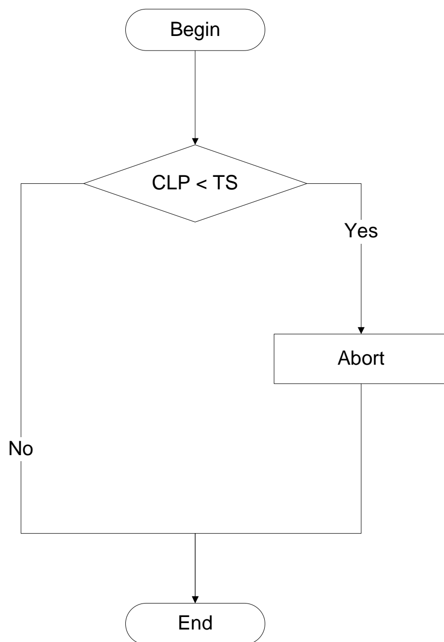

Figure 2: Transactional Locking II

validation since the object is older than the transaction and cannot possibly violate the data consistency. If the object is younger than

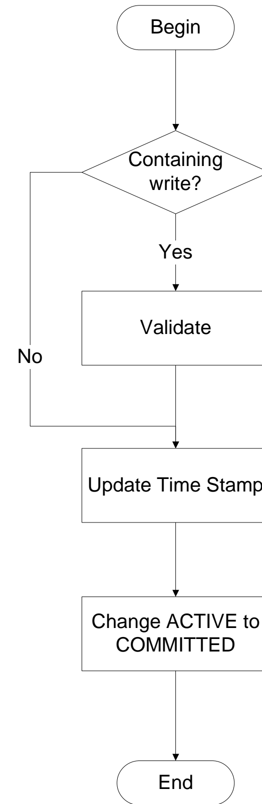

Figure 3: Commit

the current transaction candidate linearization point, then a full validation is needed. If the validation fails, the transaction aborts. If the validation succeeds, the transaction can continue. In figure 2, whenever a transaction encounters an object newer than itself, the transaction aborts.

We would like to point out a key observation here: a successful validation means that the candidate linearization point of the transaction can be updated to the current time, since the validation confirms that the transaction has a consistent view of the world at that point in time. This avoids additional unnecessary validations when opening subsequent objects that have been modified after the previous candidate linearization point, but before the new one.

Figure 3 illustrates the necessary steps to commit. A full validation is needed before the commit phase if the transaction is writing any objects. This is to ensure that there is no object O that was open for read by the transaction (and whose value could potentially have been used to compute the values of an object W being written) that has been modified and committed in the meantime by some other transaction, creating an inconsistent state (W would be created using the old value of O yet it would be recorded as being younger than the new value of O). Just before the actual commit (done by using compare and swap to change the status of the transaction from $ACTIVE$ to $COMMITED$), the timestamps of all the objects the transaction is writing are updated to the current time.

3.2 Comparison

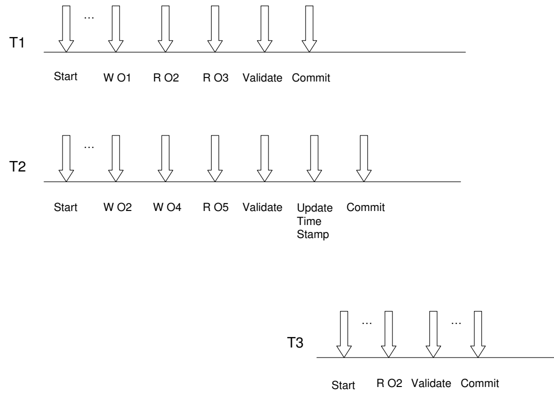

Figure 4: Example

In this section, we illustrate the cases where these algorithms differentiate from each other.

In figure 4, there are three threads T_1, T_2 and T_3. Let's assume that T_1, T_2 and T_3's candidate linearization points have all been updated to the point after O_1 through O_5's latest modifications. For lazy snapshot algorithm, when T_1 opens O_1 in write mode and O_2, O_3 in read mode, it does not need to validate any of the past reads since O_1, O_2 and O_3 are older than T_1's candidate linearization point. For thread T_2, when it opens O_2, O_4 and O_5, it does not need to validate since none of their objects is younger than T_2's candidate linearization point. The validation due to the open of O_2 and O_4 in write mode is postponed to just before the commit phase. For thread T_3, when it opens O_2, it needs to validate because O_2 is updated in T_2.

In global commit counter algorithm, T_1 and T_2 might need to validate more times when there is any write commit happened in the system, even if it was not updating any of the objects O_1 through O_5. While in transactional locking II, T_1 and T_2 can be aborted if any of O_1 to O_5 is modified.

Lazy snapshot algorithm eliminates a superset of full validations compared to that of global commit counter and TL II. As discussed before, eliminating validation and permitting the transaction to move forward closer to its commit point is not always beneficial: it potentially increases the amount of wasted work being performed by a transaction that is doomed to abort anyway. Detailed performance evaluation will be shown in next section.

Each of these three algorithms has their shining points. Global commit counter has the least memory overhead by just adding a shared commit counter. TL II eliminates both validation and bookkeeping overhead. Lazy snapshot algorithm is the most aggressive one among the three to move transactions closer to commit by extending validity ranges and choosing among multiple versions.

In spite of their difference, all of them can reason to some accuracy about when the accessed shared object set is consistent by leveraging the time information. One point to notice is that all of these three algorithms are still conservative at proving consistency. By the range of cases that each algorithm can prove consistent, they can be ordered as global commit counter $<$ TL II $<$ lazy snapshot.

Another aspect of the comparison if the space overhead. When each shared object is large, the additional field increases the memory requirement by a small amount. In our implementation, for lazy snapshot algorithm and TL II, the space overhead to the shared object is 10%-33% in the worst case (invisible reads). The extra space needed by the global commit counter algorithm is minimal. It only needs to add a global commit counter to the whole system.

3.3 Implementation Details

We implemented TL II and Lazy Snapshot based on RSTM. We add a timestamp field to each shared object. The timestamp is 32-bit in our software counter implementation and 64-bit in our hardware counter implementation. Global commit counter algorithm is already included in RSTM Release 2.

In the software counter version, the global timestamp counter can be a bottleneck because it serializes all updates to it. The good news is that only transactions having at least one object to write need to update the counter, and the common characteristic of transactions is they are usually short and mostly reads [3].

Riegel, et al. [20] propose a scalable counter to solve the possible bottleneck of the global time stamp counter. In a chip multiprocessor, it is possible to build a hardware clock counter register shared among the cores on the chip (such counters exist on Intel's Core 2 Duo and on AMD Athlon X2 multicore chips). This counter gets updated by hardware on every cycle, so no external updates by the STM system are needed. Unfortunately, the reads of this shared register are expensive on current hardware (64 cycles on Core 2 Duo), so its performance should be further evaluated on future CMP processors. We believe that a more efficient hardware implementation could be done in future multicore processors, and that even with the current inefficient implementation this solution will scale much better with the increase of the number of cores on a chip.

Another issue is the overflow of the global counter. The counter is implemented using compare-and-swap instruction. When only a 32-bit CAS is available, having a counter wider than 32 bits needs to take care of the atomicity requirement. While a 32-bit timestamp counter can still support up to 4G transactions with writes. The number it can support in our implementation is somewhat less than 4G because we count the commit attempts instead of the successful commits. The reason for counting the attempts is that the transaction descriptor status field and the timestamp counter are two separate fields. Without an atomic support of MCAS, there is no efficient way to update both atomically. Therefore we conservatively count commit attempts instead of only the successful commits.

One solution to the overflow problem is to set up an interrupt that halts the whole system, aborts all the current transactions and resets the counter to 0 at certain intervals. The interval is related to the maximum amount of transactions per second. In our current implementation this update would only need to be done approximately every 30 minutes.

Hardware counters on today's multicore chips are 64-bit, and with a clock frequency of 2GHz, the counter will wrap around approx-

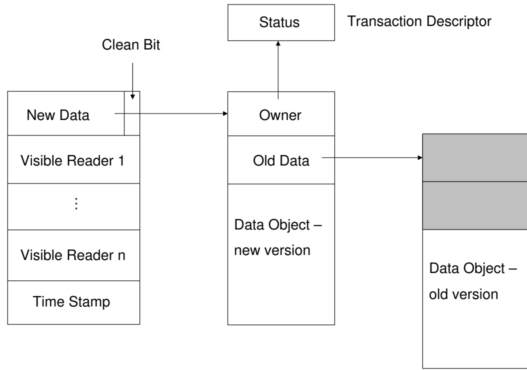

Figure 5: Metadata

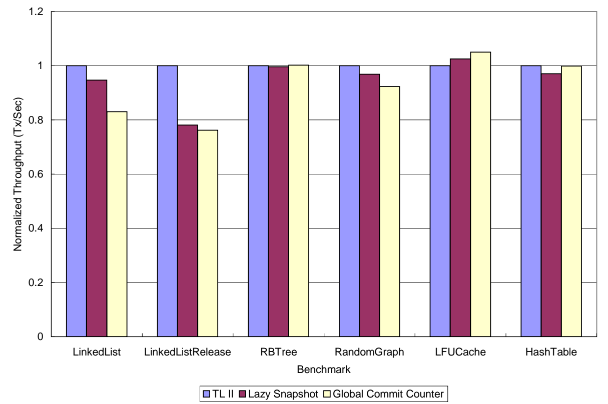

Figure 6: 2 Threads

imately every 293 years, so for all practical purposes the overflow problem does not exist in this case.

4. EXPERIMENTAL RESULTS

We conducted the experiments based on Release 2 of the Rochester Software Transactional Memory (RSTM) [16], which in general offers significant performance gains relative to its predecessor [26]. We use the global commit counter algorithm implemented in RSTM and implement lazy snapshot and TL II on RSTM. In order to compare these three algorithms fairly, we changed the validity range extension algorithm by performing a full validation, which is to verify if any of the object references has changed. The experiments are executed on a SunFire 6800 cache coherent multiprocessor machine. The compiler we used is gcc 4.1.1.

RSTM is a fast, nonblocking C++ library built to support transactional memory in C++. It accesses an object through its header. The header has a clean bit to indicate whether the object is owned by a transaction. The object header points directly to the current version of the object. Each thread maintains a transaction descriptor that indicates the status of the thread's most recent transaction. The transaction descriptor also maintains lists of objects opened for read-only and read-write access.

RSTM supports both visible and invisible reads, and both eager and lazy acquires, and uses deferred updates. We extend RSTM by adding a time stamp field to the object header and adding a candidate linearization point field to the transaction descriptor. Figure 5 depicts our modifications to the RSTM metadata.

The hardware version of heuristic leverages a system counter register for the global time stamp. The software version uses a shared software counter for this purpose. The advantage of the software version is that it requires no hardware support and can thus be deployed onto any hardware platform that supports RSTM. However, time stamp updates to a shared counter are serialized and may represent a performance bottleneck.

The hardware version eliminates the potential bottleneck from accessing the shared counter, but in exchange, it incurs the (hardware) overhead of synchronizing a timestamp register across all processors and cores. Further, explicit hardware counter registers are not supported on many platforms, and may be contended for by other concurrently executing applications. Because of these limitations, we restrict evaluation of our hardware heuristic to a Core 2 Duo-based machine.

4.1 Benchmarks

We use six benchmarks in our experiments. They include a sorted linked list (LinkedList), a sorted linked list with hand-coded early release (LinkedListRelease), a red black tree (RBTree), a hash table (HashTable), a web cache simulation using the least-frequently-used page replacement policy (LFUCache), and an adjacency list-based undirected graph (RandomGraph). All of these benchmarks are taken from the RSTM Release 2 distribution.

For the LinkedList, LinkedListRelease, RBTree and HashTable benchmarks, we evaluate the mix of operations which contains 80% lookups and 10% inserts and removes. It should be noted that this mix still represent fairly high contention scenarios: 20% of all operations are updates. The operations are generated following the uniform distribution.

HashTable has 256 buckets with overflow chains. The red black tree contains values in the range of 0 - 255. The linked list benchmarks contain values from 0 to 255. LinkedListRelease uses early release to avoid false conflicts.

LFUCache tracks page access frequency in a simulated web cache using an array-based index and a priority queue.

The RandomGraph benchmark has 50% insert and 50% remove operations. The benchmark connects four randomly chosen neighbors with the newly inserted node. When any node is inserted or removed from the graph, the vertex set and the degree of every node are updated accordingly. The graph is implemented using a sorted list of nodes. Each node has its own sorted list of neighbors. Transactions in RandomGraph exhibit a high probability of conflicting with each other.

4.2 Test Methodology

We evaluated each benchmark on a SunFire 6800 with 16 UltraSPARC III processors running at 1.2 GHz.

We ran each benchmark with a variety of threads, using the standard RSTM test driver. We report peak results from three five-second runs.

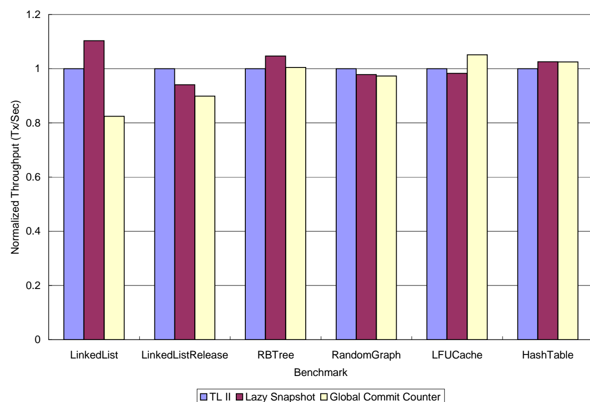

Figure 7: 4 Threads

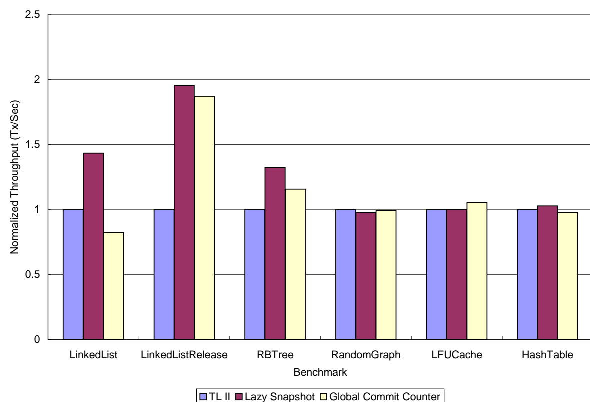

Figure 10: 12 Threads

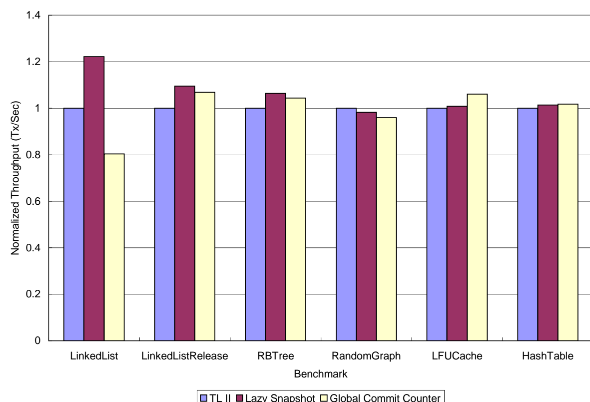

Figure 8: 6 Threads

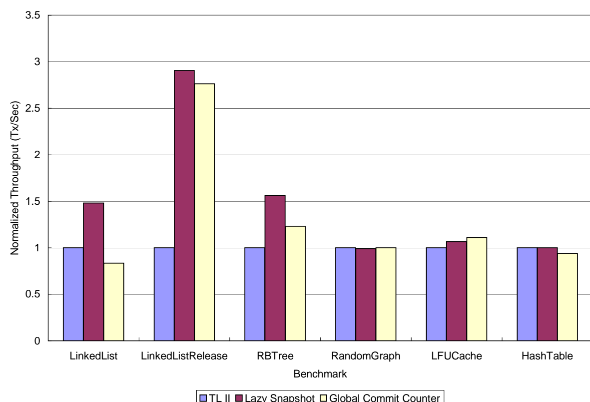

Figure 11: 16 Threads

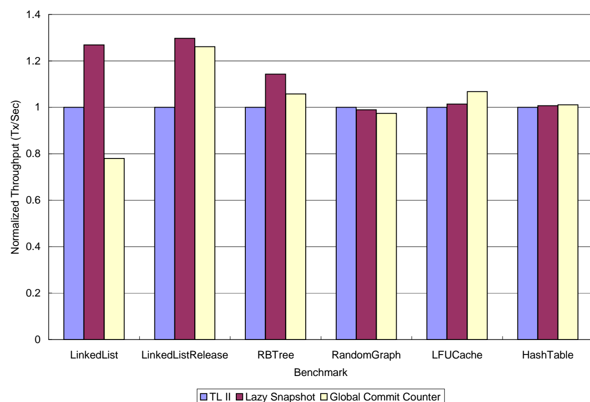

Figure 9: 8 Threads

The experiments using a software counter were conducted on the SunFire 6800. Figure 6 to figure 11 show the transactions finished per second of these algorithms. The results are normalized to transactional locking II.

4.3 Discussion

Figure 6 to figure 11 we report in this section are the overall throughput normalized to transactional locking II algorithm. The performance improvements of all these three algorithms over the naïve incremental validation approach are not shown in this report.

We observe some interesting results from these figures. First, there does not exist a simple best algorithm among the three. For different thread number and different benchmarks, each of the three algorithms scores at least one win.

Transactional Locking II shows its biggest win when thread number is small. It seldom wins when thread number is large. We think this is because transactional locking II lacks the flexibility of extending the validity range like lazy snapshot algorithm. Without extending the validity range, a transaction aborts whenever it encounters an object having a newer timestamp. This makes it lose the possible successful commit. But for cases where thread number is small or contention is low, transactional locking II wins because the bookkeeping overhead it saves exceeds the possible gains by extending validity ranges.

Lazy snapshot algorithm shows excellent performance in most of the cases in our experiments. The ability to extend validity ranges allows it to move closer to the possible commit than the other two algorithms. Its disadvantage is the extra bookkeeping overhead required compared with transactional locking II.

From the perspective of throughput performance, global commit counter wins the least cases in all three competitors. We think this is because of its pessimistic nature about when a transaction has to validate its past reads. This algorithm reports a lot more false positives than the other two, especially lazy snapshot algorithm. But

global commit counter requires the least extra memory which gives it an edge when memory space is tight. Another thing to notice is in the LFUCache benchmark, global commit counter achieves the best throughput. We attributes this to the all writes and zipf distribution of data access pattern of LFUCache.

LinkedList and LinkedListRelease exhibit the biggest performance difference among the six benchmarks, because they have a large read set to validate and their access pattern is comparatively simple.

RBTree's data structure allows more potential parallelism than LinkedList since tree traversals can be confined to different parts of the tree. However, the rebalancing step after a tree update can affect a large part of the tree, increasing the potential for conflict. Moreover, the transactions in RBTree are very short since the tree consists of only 256 elements, causing the small overhead a strategy imposes to create a larger relative impact on performance.

For LFUCahce, the re-heapify step when swapping out an item needs to move the object to its destination by swapping with all other objects on the path, which are all open for write. This step potentially conflicts with any other transactions using any object in the heap, thus greatly reducing the overall parallelism. Different validation techniques will have a relatively small overall impact.

One pattern we observe from the results is that the parallel scalability of many benchmarks reaches saturation at around 10 threads; performance drops after this point. We attribute this behavior to the increased amount of wasted work in high contention cases and to the concurrency capacity of data structures. Data structures such as HashTable inherently support more parallelism than data structures such as LinkedList. Therefore increasing the number of threads beyond the saturation of the available concurrency does not result in better performance.

The lazy snapshot algorithm shows slight performance degradation in a few high-contention, low-parallelism cases, when compared to Spear et al's algorithm, we note that these algorithms are fully compatible with regular validation techniques. Hence, an intelligent transactional memory runtime that adapts between strategies is fully feasible and could eliminate the drawbacks of our approach.

5. CONCLUSIONS AND FUTURE WORK

We have presented a comprehensive analysis and comparison of recently emerged time-based STM algorithms. By using the information of global time, these algorithms are able to avoid aborting transactions that have opened objects for read-only access in some cases where that object is subsequently written to by a competing transaction. In particular, if it can infer that every object accessed in the transaction belongs to a consistent snapshot of the transactional memory at a point in time prior to the competing write, validation of earlier reads can be skipped.

We are designing an adaptive runtime strategy that will fine-tune the validation approach depending on the amount of contention: more pessimistic for high contention scenarios (earlier conflict detection and earlier aborts), more optimistic for low contention (allow the transactions to continue even if they may fail to commit). Figure 12 shows some preliminary result for 16 threads when using a short commit history to fine tune the size of objects the next transaction should bookkeep. From the result we have, the adaptive strategy is promising.

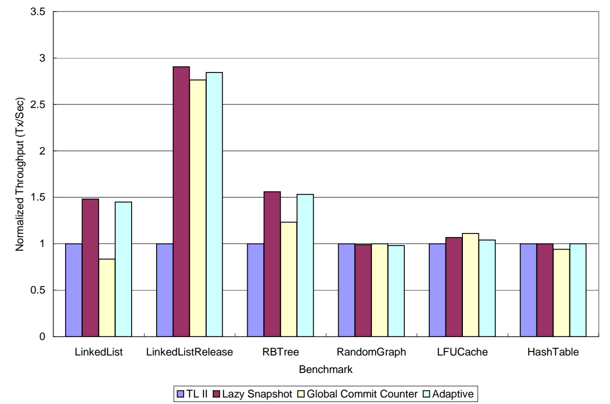

Figure 12: Adaptive Strategy

6. ACKNOWLEDGMENTS

We would like to acknowledge the contributions of the late Ken Kennedy, who for a long time led a multi-institutional effort on multicore computing and who was particularly enthusiastic about this project and participated in its early stages.

We also thank Michael Spear for helping us obtain and install the RSTM implementation on our systems. We thank John Mellor-Crummey and James Larus for sharing their insights in personal discussions.

7. REFERENCES

[1] A.-R. Adl-Tabatabai, B. T. Lewis, V. Menon, B. R. Murphy, B. Saha, and T. Shpeisman. Compiler and runtime support for efficient software transactional memory. In *Proceedings of the 2006 Conference on Programming language design and implementation*, pages 26–37. Jun 2006.

[2] C. S. Ananian and M. Rinard. Efficient object-based software transactions. In *Proceedings, Workshop on Synchronization and Concurrency in Object-Oriented Languages*, San Diego, CA, Oct 2005. In conjunction with OOPSLA'05.

[3] J. Chung, H. Chafi, C. Cao Minh, A. McDonald, B. D. Carlstrom, C. Kozyrakis, and K. Olukotun. The common case transactional behavior of multithreaded programs. In *12th International Symposium on High Performance Computer Architecture (HPCA)*. Feb 2006.

[4] P. Damron, A. Fedorova, Y. Lev, V. Luchangco, M. Moir, and D. Nussbaum. Hybrid transactional memory. In *ASPLOS-XII: Proceedings of the 12th international conference on Architectural support for programming languages and operating systems*, pages 336–346, New York, NY, USA, 2006. ACM Press.

[5] D. Dice, O. Shalev, and N. Shavit. Transactional locking ii. In *Proceedings of the 20th International Symposium on Distributed Computing, DISC 2006*. Springer, Sep 2006.

[6] D. Dice and N. Shavit. What really makes transactions faster? In *Proceedings of the First ACM SIGPLAN Workshop on Languages, Compilers, and Hardware Support for Transactional Computing*. Jun 2006.

[7] K. Fraser. *Practical lock freedom*. PhD thesis, Cambridge University Computer Laboratory, 2003. Also available as Technical Report UCAM-CL-TR-579.

[8] T. Harris and K. Fraser. Language support for lightweight transactions. In *OOPSLA '03: Proceedings of the 18th annual ACM*

SIGPLAN conference on Object-oriented programing, systems, languages, and applications, pages 388–402, New York, NY, USA, 2003. ACM Press.

[9] T. Harris, M. Plesko, A. Shinnar, and D. Tarditi. Optimizing memory transactions. *SIGPLAN Not.*, 41(6):14–25, 2006.

[10] M. Herlihy, V. Luchangco, M. Moir, and W. N. Scherer III. Software transactional memory for dynamic-sized data structures. In *PODC '03: Proceedings of the twenty-second annual symposium on Principles of distributed computing*, pages 92–101, New York, NY, USA, 2003. ACM Press.

[11] M. Herlihy and J. E. B. Moss. Transactional memory: architectural support for lock-free data structures. In *ISCA '93: Proceedings of the 20th annual international symposium on Computer architecture*, pages 289–300, New York, NY, USA, 1993. ACM Press.

[12] A. Israeli and L. Rappoport. Disjoint-access-parallel implementations of strong shared memory primitives. In *PODC '94: Proceedings of the thirteenth annual ACM symposium on Principles of distributed computing*, pages 151–160, New York, NY, USA, 1994. ACM Press.

[13] S. Kumar, M. Chu, C. J. Hughes, P. Kundu, and A. Nguyen. Hybrid transactional memory. In *Proceedings of Symposium on Principles and Practice of Parallel Programming*, Mar 2006.

[14] J. R. Larus and R. Rajwar. *Transactional Memory*. Morgan & Claypool, 2006.

[15] Y. Lev and M. Moir. Fast read sharing mechanism for software transactional memory (poster). In *Proc. of the 24th Annual ACM Symp. on Principles of Distributed Computing*. July 2004.

[16] 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 *TRANSACT 06': Proceedings of the Workshop on Languages, Compilers, and Hardware Support for Transactional Computing*, 2006.

[17] G. E. Moore. Cramming more components onto integrated circuits. *Readings in computer architecture*, pages 56–59, 2000.

[18] K. Olukotun and L. Hammond. The future of microprocessors. *ACM QUEUE Magazine*, September 2005.

[19] T. Riegel, P. Felber, and C. Fetzer. A lazy snapshot algorithm with eager validation. In *Proceedings of the 20th International Symposium on Distributed Computing, DISC 2006*, volume 4167 of *Lecture Notes in Computer Science*, pages 284–298. Springer, Sep 2006.

[20] T. Riegel, C. Fetzer, and P. Felber. Time-based transactional memory with scalable time bases. In *19th ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*, Jun 2007.

[21] B. Saha, A.-R. Adl-Tabatabai, R. L. Hudson, C. C. Minh, and B. Hertzberg. Mcrt-stm: a high performance software transactional memory system for a multi-core runtime. In *PPoPP '06: Proceedings of the eleventh ACM SIGPLAN symposium on Principles and practice of parallel programming*, pages 187–197, New York, NY, USA, 2006. ACM Press.

[22] M. L. Scott. Sequential specification of transactional memory semantics. In *ACM SIGPLAN Workshop on Transactional Computing*. Jun 2006. Held in conjunction with PLDI 2006.

[23] D. Shasha and M. Snir. Efficient and correct execution of parallel programs that share memory. *ACM Trans. Program. Lang. Syst.*, 10(2):282–312, 1988.

[24] N. Shavit and D. Touitou. Software transactional memory. In *PODC '95: Proceedings of the fourteenth annual ACM symposium on Principles of distributed computing*, pages 204–213, New York, NY, USA, 1995. ACM Press.

[25] A. Shriraman, V. J. Marathe, S. Dwarkadas, M. L. Scott, D. Eisenstat, C. Heriot, W. N. Scherer III, and M. F. Spear. Hardware acceleration of software transactional memory. In *TRANSACT 06': Proceedings of the Workshop on Languages, Compilers, and Hardware Support for Transactional Computing*, 2006.

[26] M. F. Spear, V. J. Marathe, W. N. Scherer III, and M. L. Scott. Conflict detection and validation strategies for software transactional memory. In *DISC06: 20th Intl. Symp. on Distributed Computing*, 2006.

[27] J. M. Stone, H. S. Stone, P. Heidelberger, and J. Turek. Multiple reservations and the oklahoma update. *IEEE Parallel Distrib. Technol.*, 1(4):58–71, 1993.