Runtime Tuning of STM Validation Techniques

Rui Zhang, Zoran Budimlić, William N. Scherer III, Mackale Joyner

Department of Computer Science, Rice University, Houston, TX 77005

{ruizhang, zoran, scherer, mjoyner}@rice.edu

Abstract

Since the end of the megahertz race in the processor industry and the switch to multicore processors, Software Transactional Memory (STM) has sparked ample interest as a programming model for the now mainstream parallel processing systems. Unfortunately, STM systems still exhibit significant performance overhead over the more traditional, lock-based programming models. A large part of this overhead is due to the STM systems spending significant amount of time validating the state of the shared memory that is visible to a transactions.

Significant improvements to the validation systems for STM have been published recently. Transactional Locking II and Lazy Snapshot validation are two time-based validation techniques that have so far shown the most promising performance for a wide variety of applications. Unfortunately, the performance of these techniques depends heavily on the application: number of concurrent jobs, length of the transaction and the read/write ratio of the shared objects can significantly favor one technique over the other. Moreover, for long-running applications that change their behavior over time, neither of these two techniques is optimal.

In this paper, we present a runtime tuning strategy that uses profiling to determine the most profitable validation technique. Our runtime tuning strategy can behave as an arbitrary mix of Transactional Locking II and Lazy Snapshot techniques depending on the state of the STM system. We evaluate our technique on a set of STM benchmarks and show that our strategy performs within a couple of percent of the best validation strategy for a given static workload scenario, and that it outperforms both of the above techniques by up to 18% in long-running, dynamically-changing scenarios.

1. Introduction

The performance of existing software transactional memory (STM) [2, 3, 4, 5, 6, 8, 11] systems is far from satisfactory. The main factors causing the poor performance of transactional memory systems include bookkeeping and validation [12]. In order to reduce the overhead of bookkeeping and validation, several time-based strategies have been proposed recently [1, 9, 12]. The validation overhead is greatly reduced by introducing the time information into the system. Unfortunately, the performance of these heuristics depends heavily on the state of the system: contention levels, number of concurrent jobs, type of application, lengths of individual transactions etc. Any given static heuristic might perform very well in some situations, but quite poorly in others [13].

Moreover, current time-based software transactional memory implementations face the scalability problem due to the contention over the shared counter that is read and written by contending transactions. The software counter implementation incurs substantial overhead when contention is high. Riegel, Fetzer and Felber propose [10] to use one physical counter or multiple synchronized physical counters to address this problem, which can remove the overhead of updating the counter and reduces the contention compared with a software counter.

In this paper, we propose a profile-driven runtime tuning strategy that changes the behavior as the system performance changes. We compare our strategy with two state-of-the-art time-based strategies: Transaction Locking II and Lazy Snapshot [1, 9]. We show that our strategy achieves performance that is on par with the better of these two given any constant system state. We also demonstrate that in a system that changes behavior over time, our strategy outperforms both of these validation techniques. We also present some experimental results on performance comparison between STM systems using hardware counter and software counter on two dual core systems.

There are three main contributions of this paper. First, we evaluate a threshold hybrid strategy of time-based validation technique and show its performance space characteristics. Second, we design a novel runtime validation tuning strategy based on our observation of the performance characteristics of the threshold hybrid strategy. Our strategy can effectively improve the system throughput in a dynamically changing environment. Third, we present performance comparison of time-based validation using a hardware counter and a software counter on a set of different benchmarks.

The next section discusses existing validation strategies. Section 3 describes the design and implementation of our threshold hybrid validation strategy and discusses its performance characteristics. It also describes the design and implementation of our profile-driven runtime tuning technique. In Section 4 we present the experimental results comparing the performance of our technique against Transactional Locking II and Lazy Snapshot algorithms over a set of benchmarks in varying scenarios (low/high contention, low/high number of concurrent jobs, static/dynamic workload). We also present the performance comparison of hardware counter and software counter in this section. The last section presents the conclusions and some directions for future research.

2. Related Work

Validation is a technique designed to prevent a transaction from observing an inconsistent state in shared data. Validation is closely related to conflict detection. While an exhaustive conflict detection system that detects all possible write/write, read/write, and write/read conflicts does not need validation, it is unfortunately prohibitively expensive.

To increase concurrency, different conflict detection strategies have been developed. *Invisible reads* [6] hide the read of an object from concurrent transactions which are thereafter able to modify the object. This postpones the detection of write after read conflicts to a later time such as in the commit phase. *Lazy writes* [2] hide the write from other transactions and allows modifications to them by other transactions, delaying the detection of read after write conflicts and write after write conflicts to a later time point. The differences among places chosen to detect conflicts also lead to different outcomes of whether a transaction is aborted. But delaying

the detection of conflicts creates the possibility of allowing a transaction to observe inconsistent states. To avoid entering an inconsistent state, a transaction needs to validate the objects it has read at appropriate times in program execution.

Incremental validation [6] is a validation strategy that validates all past invisible reads and lazy writes every time the transaction opens a new object. If any change in the past is detected, the validation fails. This strategy guarantees a consistent state but imposes a substantial overhead [12], since it is essentially a $O(n^2)$ operation where n is the number of objects opened in a transaction.

Lev, Moir and Nussbaum [7] propose a Phased Transactional Memory (PhTM). Their system supports switching between different "phases" that represent different forms of transactional memory support. Our runtime tuning strategy uses a similar approach by adapting the transactional memory system to different workloads. PhTM targets transactional memory systems with hardware support, while our runtime tuning strategy targets validation in time-based STM systems.

Time-based validation strategies guarantee the consistency of the past reads by simply checking whether the timestamp of the object being opened is in the transaction's validity range. This reduces the validation to a couple of comparisons, greatly reducing the overhead introduced by incremental validation.

2.1 Existing Time-based Validation Strategies

In Spear, et al.'s [12] *Global Commit Counter* (GCC) heuristic, a transaction does not perform a validation if no writes were committed in the entire system since the last object was opened. This scheme can avoid many unnecessary validations, leading to performance improvement for situations where writes happen rarely in the system. However, this strategy is still very conservative since a write to a shared object does not affect a particular transaction's consistency if that object is never used in that transaction.

Dice, et al.'s [1] *transactional locking II* (TL II) algorithm is another time-based STM implementation. Their algorithm associates a timestamp with every shared object at the time of its modification. A transaction acquires its own timestamp once at the beginning of execution, and commits successfully if none of the objects opened during the execution has a timestamp newer than the timestamp of the transaction. A transaction aborts if it encounters an object with a newer timestamp. An advantage of this algorithm is that it can completely eliminate the bookkeeping overhead of read-only transactions since it never validates opened objects, making it attractive to situations where contention is low and most transactions commit successfully. Figure 1 shows the major steps of TL II's validation strategy.

For convenience, we explain the meaning of the variables used in figure 1, 2 and 3 here. T is a transaction. O_i is an object. $READ\_ONLY$ indicates whether the transaction is a read-only transaction or not. $T.ts$ is the transaction's timestamp. It is used to reason about the ordering of the transaction and the objects. TS is the variable to save the timestamp to be written into the objects being updated. TSC is the shared timestamp counter. $T.O$ is a list of objects that need to be verified when doing a validation. $VALIDATE(T)$ checks if the invisible reads and lazy writes have been modified or acquired by other transactions. If they are, transaction T is aborted, otherwise it proceeds forward.

Riegel, et al.'s [9] *Lazy Snapshot* algorithm maintains multiple versions for each shared object and uses the global timestamp information to guarantee that the view observed by a transaction is consistent. In their algorithm a transaction can choose the object version to satisfy consistency. When a transaction encounters an object with a newer time stamp, its validity range is extended by doing a full validation. Figure 2 shows the major steps of Lazy Snapshot's validation strategy.

```
1 if READ_ONLY = TRUE then
2     if O_i.ts > T.ts then
3         ABORT(T);
4 else
5     if O_i.ts > T.ts then
6         ABORT(T);
7     else
8         T.O ← T.O ∪ O_i;
```

Figure 1. TL II Validation

```
1 if O_i.ts > T.ts then
2     T.ts ← TSC;
3     VALIDATE(T);
4 T.O ← T.O ∪ O_i;
```

Figure 2. Lazy Snapshot Validation

2.2 Comparison of Existing Time-based Validation Strategies

Global Commit Counter has the least memory overhead since it only adds a shared commit counter. The sizes of objects are unchanged. However, it will perform a full validation every time there is a write in the entire system, even if that write does not affect the current transaction. These unnecessary validations can sometimes lead to a relatively poor performance when compared to the other two algorithms.

Transactional Locking II eliminates both validation and bookkeeping overhead, but it can sometimes be too conservative and abort transactions that could commit successfully. It has the additional memory requirement of adding a timestamp to every object.

In this paper we refer to a variant of the Lazy Snapshot algorithm that only keeps a single version of an object. Compared to GCC, Lazy Snapshot eliminates a superset of full validations. Compared with TL II, Lazy Snapshot extends a transaction's validity range when it encounters an object with a newer timestamp. Lazy Snapshot needs bookkeeping of all its past reads for the purpose of validation. It can execute a transaction to a point closer to the commit than either GCC or TL II. However, this is not always beneficial: it potentially increases the amount of wasted work being performed by a transaction that is doomed to abort. The bookkeeping overhead of Lazy Snapshot is the same as for the GCC, and larger than TL II. The memory overhead of Lazy Snapshot is the same as with TL II and larger than GCC.

The major difference between TL II and Lazy Snapshot is in deciding when a transaction should be aborted. When encountering a new object, Transaction Locking II assumes aborting the current transaction benefits the system the best and aborts it right away, while Lazy Snapshot validates the read set and gives the transaction a further chance to commit.

3. Runtime Tuning of Validation Techniques

Global Commit Counter, Transactional Locking II and Lazy Snapshot all reduce the validation overhead of the incremental validation by leveraging the global time information. They effectively reduce the number of times the transactional system has to perform full validations and thus improve the overall performance. But none of these techniques consistently outperforms the other two in all scenarios. In this section we first present a hybrid time-based validation technique that is essentially a combination of Transactional

Locking II and Lazy Snapshot, and show its performance characteristics. Then we present a runtime tuning technique based on our observations of the performance space characteristics of our threshold hybrid technique.

3.1 Threshold Hybrid Validation Strategy

In Lazy Snapshot, a transaction can conduct several validations before its final commit. In Transactional Locking II, a transaction never performs a full validation for read-only transactions and only performs a full validation for update transactions that reach their commit phase. Our hybrid validation consists of two phases. In the first phase it behaves as Lazy Snapshot: it performs a full validation every time it opens an object with a timestamp newer than the current transaction's *candiadate linearization point* (CLP). It keeps a count of the number of open objects since the start of the transaction, and when this count reaches a certain threshold, it switches to the second phase. In the second phase, our strategy behaves as Transaction Locking II. It does not perform any more bookeeping, and aborts if it encounters a new object.

The behavior of our threshold strategy is controlled by a simple threshold. If this threshold is 0, our hybrid strategy behaves as Transactional Locking II. When the threshold is a large number, it behaves similar to Lazy Snapshot. For a threshold in between, it behaves as a arbitrary mix of Transactional Locking II and Lazy Snapshot. Figure 3 shows the major steps of our threshold hybrid validation strategy.

```
1 if O_i.ts > T.ts then
2     if OPEN_COUNT < THRESHOLD then
3         T.ts ← TSC;
4         VALIDATE(T);
5         T.O ← T.O ∪ O_i;
6     else
7         ABORT(T);
```

Figure 3. Threshold Hybrid Validation

We have experimented with different heuristics for deciding when to switch to the second phase in our validation strategy. One method is to count all open objects and compare the count with the threshold on every full validation or on every object open. Another method is to count all full validations and compare that count to the threshold before doing a full validation. Since we found little difference in the performance of these heuristics, in this paper we assume a method that counts all open objects and does a comparison at every full validation.

One potential danger associated with time-based validation techniques is starvation. There is a possibility that a transaction keeps being aborted because it keeps observing inconsistent state of the shared memory. This will happen more often in TL II than in Lazy Snapshot, since TL II aborts the transaction as soon as it opens a newer object. Our hybrid validation fits in between TL II and Lazy Snapshot as far as possibility of starvation is concerned.

3.2 Implementation

Our implementation is based on Rochester Software Transactional Memory (RSTM) [8] Release 2, which in general offers significant performance gains relative to its predecessor [12]. It supports both visible and invisible reads, and both eager and lazy acquires, and uses deferred updates.

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

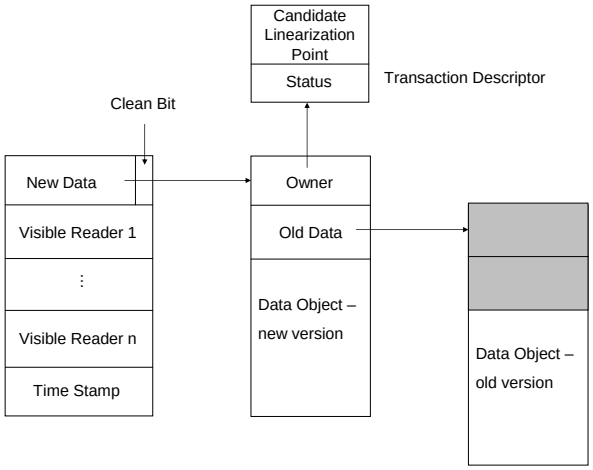

Figure 4. Metadata

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. We extend RSTM by adding a timestamp field to the object header and adding a candidate linearization point field to the transaction descriptor. Figure 4 depicts our modification to the RSTM metadata.

The timestamp TS of each shared object indicates the time when the object is last modified. Closely related to timestamps are candidate linearization points. Each transaction keeps its own candidate linearization point 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 lies relative to all the writes that have happened in the system. Having this knowledge enables the transaction 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. When a transaction starts, its candidate linearization point is initialized to the beginning of the transaction. In Lazy Snapshot, CLP is updated each time a transaction opens a younger object and successfully validates past reads.

We use a global counter TSC to reflect the current time relative to the start of the program. TSC can be read concurrently by all the transactions in the system. The experiments are executed on a SunFire 6800 cache coherent multi-processor machine, with gcc 4.1.2 compiler.

3.3 Benchmarks

We use six benchmarks in our experiments. They include a sorted linked list (LinkedList), a sorted linked list with hand-coded early release mechanism (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; however, it should be noted that we have added a read-only lookup operation to RandomGraph.

LinkedList, LinkedListRelease and RBTree contain values from 0 to 255. HashTable has 256 buckets with overflow chains. LFUCache tracks page access frequency in a simulated web cache using an array-based index and a priority queue. RandomGraph connects

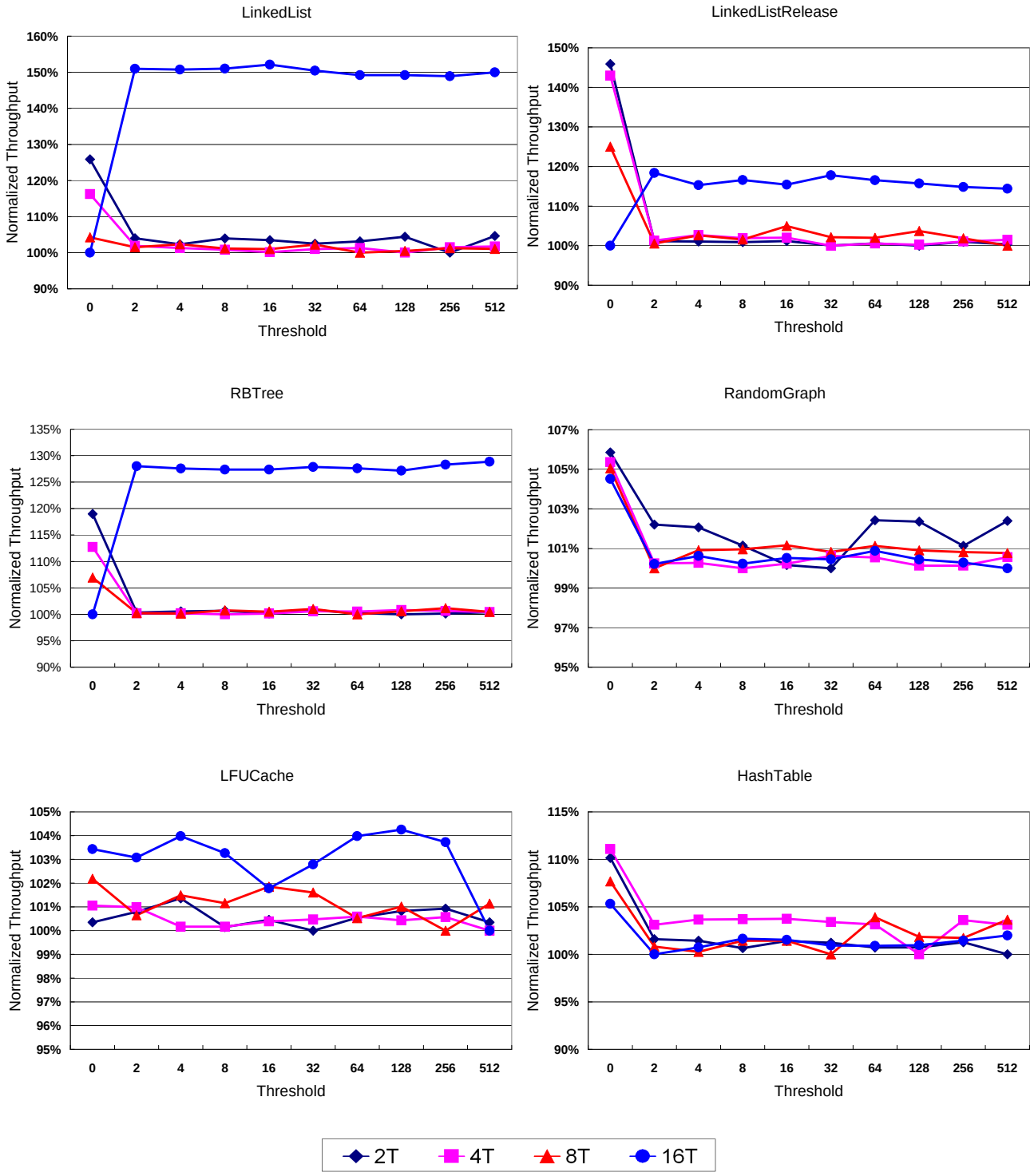

Figure 5. Threshold Hybrid Strategy Performance Space

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.

In order to show the impact of different thresholds on the performance, our experiments measure the throughput of the whole system when threshold changes. To change the contention levels, we set up benchmarks with different numbers of competing threads (from 2 to 16). The lookup/insert/remove ratio per thread is 98%/1%/1%.

Figure 5 shows how the performance of our benchmarks changes with different thresholds. The x-axis of each figure is the threshold. The y-axis of each figure is the throughput normalized to the smallest throughput on each line. We use normalized throughput to show the shape of the performance space of four different thread numbers of each benchmark. For each benchmark, we measured the cases of 2, 4, 8 and 16 threads, denoted with 2T, 4T, 8T and 16T in the legend.

The results show a nearly monotonic performance space most of the time across a variety of benchmarks and contention ratios. In other words, if, for a given scenario, one of the validation techniques (TL II or Lazy Snapshot) performs better than the other, then it will also very often outperform our threshold hybrid technique. This observation leads us to a coarse-grain runtime tuning strategy which switches the runtime validation strategy to the better one of Transactional Locking II and Lazy Snapshot. This runtime tuning strategy is described in the next section.

An interesting observation can be made about the threshold: it appears that in most of our benchmarks the "switching point" in performance happens around a threshold value of two. This is not very surprising after we observed that the average number of validations per transaction in all of our benchmarks is around 4. This also suggests that a different set of applications with a larger number of validations per transaction may create contention scenarios where the threshold hybrid strategy would outperform both of the extreme cases. Investigation into the practical usability of the threshold hybrid technique is a subject of our future research.

3.4 Runtime Tuning of Validation

Our runtime tuning strategy is based on the observation of the nearly monotonic property of the performance space. Since the performance increases or decreases as the threshold increases, the best performance is either achieved with TL II or Lazy Snapshot. The question is how and when to choose between the two validation techniques. Our solution is to monitor the contention change in the system and select the right strategy when the change happens.

Figure 6 shows the major steps of our runtime tuning strategy. The STM system starts by running both Transactional Locking II and Lazy Snapshot for a short time and picking up the better one to continue. When the contention monitor detects a contention change, our runtime tuning system switches to the other strategy for a short time, then compares the performance with the current strategy. If the newly tested strategy is better, the system stays in that state, otherwise it switches back to the original strategy.

Our strategy has two major components - contention monitor and heuristic selector. Contention monitor monitors the runtime contention level and informs the runtime about any changes. The heuristic selector selects the best validation heuristic to continue the execution.

Contention level monitor needs to be sensitive to contention level changes. We use the wall clock time to commit a certain number of transactions as the indicator of the contention level. One advantage of using the wall time as the contention indicator is that it incurs very little overhead. Moreover, wall time is a relatively stable and accurate indicator to filter micro-variations in the TM system performance, which can vary up to 20% for short periods of time in our experiments, even for a constant contention level. In our implementation, the contention level monitor measures the time it takes to commit 10,000 transactions (an arbitrary number). Depending on how fast the application changes its behavior, a larger or smaller number may be more suitable.

If the time spent on two consecutive sequence of transactions varies more than some threshold, the monitor reports it as a contention change and informs the heuristic selector. In our experiments we set the threshold to 5%.

The goal of this paper is to demonstrate that given a way of accurately monitoring the contention change, it is beneficial to use this information to guide the STM runtime to use the more suitable strategy. Designing an accurate contention monitoring scheme for general applications is beyond the scope of this paper. We experimented with different strategies to estimate the contention level, including the ratio of total aborts over commits and recent aborts/commits ratio, but did not find a meaningful correlation between those indicators and the contention level.

```
1 if ContentionChanged = TRUE then
2     T_new ← TestRun(Alternative Strategy);
3     if T_new < T_old * ratio then
4         Use New Strategy;
5     else
6         Use Old Strategy;
```

Figure 6. Runtime Tuning Validation

4. Experimental Results

Our platform for threshold hybrid strategy and runtime tuning strategy evaluation was 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. Our experiments on hardware counter and software counter comparison were based on a Core 2 Duo-based machine and a dual-core Opteron based machine.

We evaluated our runtime tuning strategy in two scenarios. First, we compare our runtime tuning strategy against TL II and Lazy Snapshot in a dynamic scenario where the contention level changes throughout the execution of the program. Second, we compare our runtime tuning strategy against TL II and Lazy Snapshot in a scenario where the lookup/insert/remove ratio of all threads is constant throughout the whole run.

For the dynamically changing scenario, we set up the benchmarks to alternate between 2 thread execution and 16 thread execution (thus changing the contention level) every 5 seconds and measure the average throughput of 3 runs. Each run lasts 25 seconds. For all benchmarks except LFUCache, the lookup/insert/remove ratio of each thread is set to be 98%/1%/1%. The lookup/ insert/remove is randomly generated following an uniform distribution. Figure 7 shows the performance results for the dynamically changing scenario.

Figure 8 shows the throughput (transactions per second) for the six different benchmarks, at different (but constant) contention levels, and different number of threads. The results are normalized to the performance of our threshold hybrid profile-driven algorithm. For the constant contention case, we fix the thread number to be 2, 4, 8 and 16. We measure the average throughput of three runs. Each run takes 5 seconds. Since in the LFUCache benchmark, the

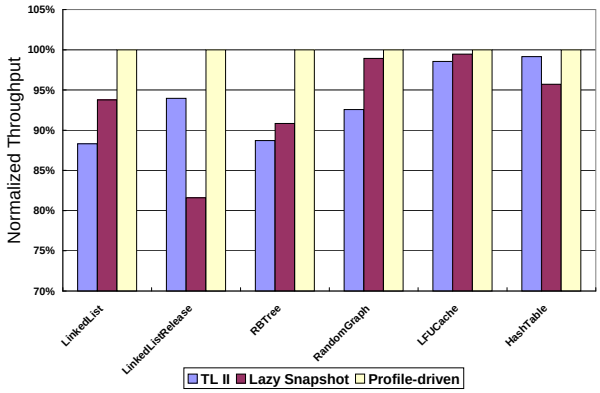

Figure 7. Runtime Tuning Result

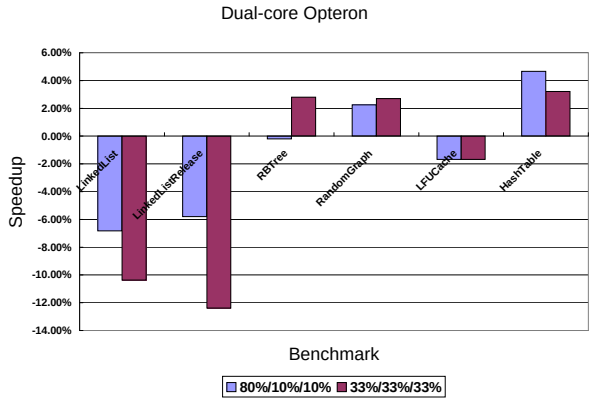

Figure 10. Hardware Counter vs Software Counter

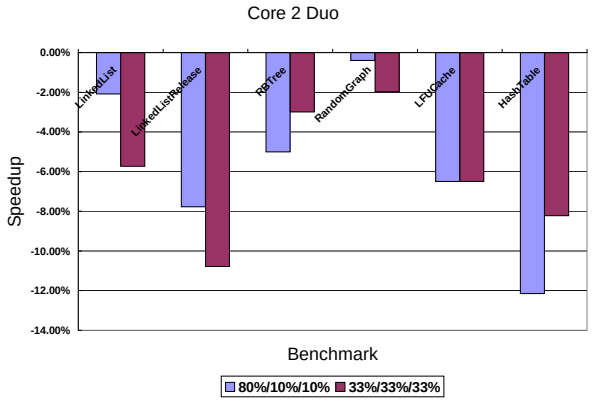

Figure 9. Hardware Counter vs Software Counter

read/write ratio cannot be changed, we only show how the result varies with a change in the number of threads.

Figure 9 and figure 10 show the throughput speedup by using hardware counter over using software counter under 2 threads at different contention ratios. The experiments are done using the Lazy Snapshot validation strategy. Since LFUCache does not have a read/write ratio, the results are identical for the two runs of this benchmark.

4.1 Discussion

In a dynamically changing environment, on figure 7, we observe that our validation tuning strategy outperforms both TL II and Lazy Snapshot in all cases. The performance improvement ranges from a couple of percent up to 18%.

We observe that in the constant contention case on figure 8, our strategy is very competitive with the best fixed strategy for any given scenario. This shows that a) the overhead of our strategy due to additional counters, run-time checks and occasional profiling of a sub-optimal strategy is relatively small, and b) that our strategy quickly converges to the behavior of the better fixed strategy. For constant contention, our strategy performs significantly better than the worse of the two in all cases, and is always within several percent of the performance of the better strategy. Our strategy sometimes even outperforms both TL II and Lazy snapshot because our tuning can find finer-grain contention changes that happen even though the overall lookup/insert/remove ratio is statistically constant, because during the short periods of time the contention level can still change.

The largest benefit of using a runtime tuning strategy presented in this paper is that it prevents the system from running a clearly inferior heuristic for long periods of time. For example, for the LinkedList benchmark in the 16 thread case with 80%/10%/10% lookup/insert/remove ratio in Figure 8, if the system is running Transaction Locking II as the validation strategy, it will suffer a 43% performance degradation compared to the optimal strategy (Lazy Snapshot). For the same benchmark and the same lookup/inser/remove ratio, if the system is running Lazy Snapshot in the 2 thread case, it will suffer a 21% performance penalty compared to the optimal strategy (Transaction Locking II).

On the other hand, if the system is using our runtime profile-driven tuning strategy, it will perform at 6% below optimal in the worst case, and within 2% of optimal in 9 out of 12 cases for the LinkedList example. For an application that dynamically changes its behavior, running our tuning strategy can achieve overall performance that neither TL II or Lazy Snapshot can achieve.

Figure 9 and figure 10 show the throughput speedup using hardware counter over using software counter. Since the thread number is small, in most of the cases, hardware counter loses against software counter. This is not surprising considering the low contention over the counter under 2 threads and the time to read a hardware counter is not very cheap as well. This suggests that for a small scale multi-core system, a software counter can be a better choice than hardware counter. We also observe some performance improvement in the dual-core Opteron case in RBTree, RandomGraph and HashTable. We attribute this to the relatively cheaper hardware counter read on Opteron than Core 2 Duo. We measured 64 cpu cycles and 8 cpu cycles for a timestamp counter read for Core 2 Duo and Opteron respectively. For a larger scale multi-core processor and faster read speed to the timestamp counter, we expect better performance and scalability from a hardware counter based approach.

5. Conclusions and Future Work

A profile-driven strategy for runtime tuning of software transactional memory validation heuristics can be competitive with the the best heuristic for any given constant contention level. In scenarios with dynamically changing contention levels, such a runtime strategy can significantly outperform even the state-of-the-art validation techniques.

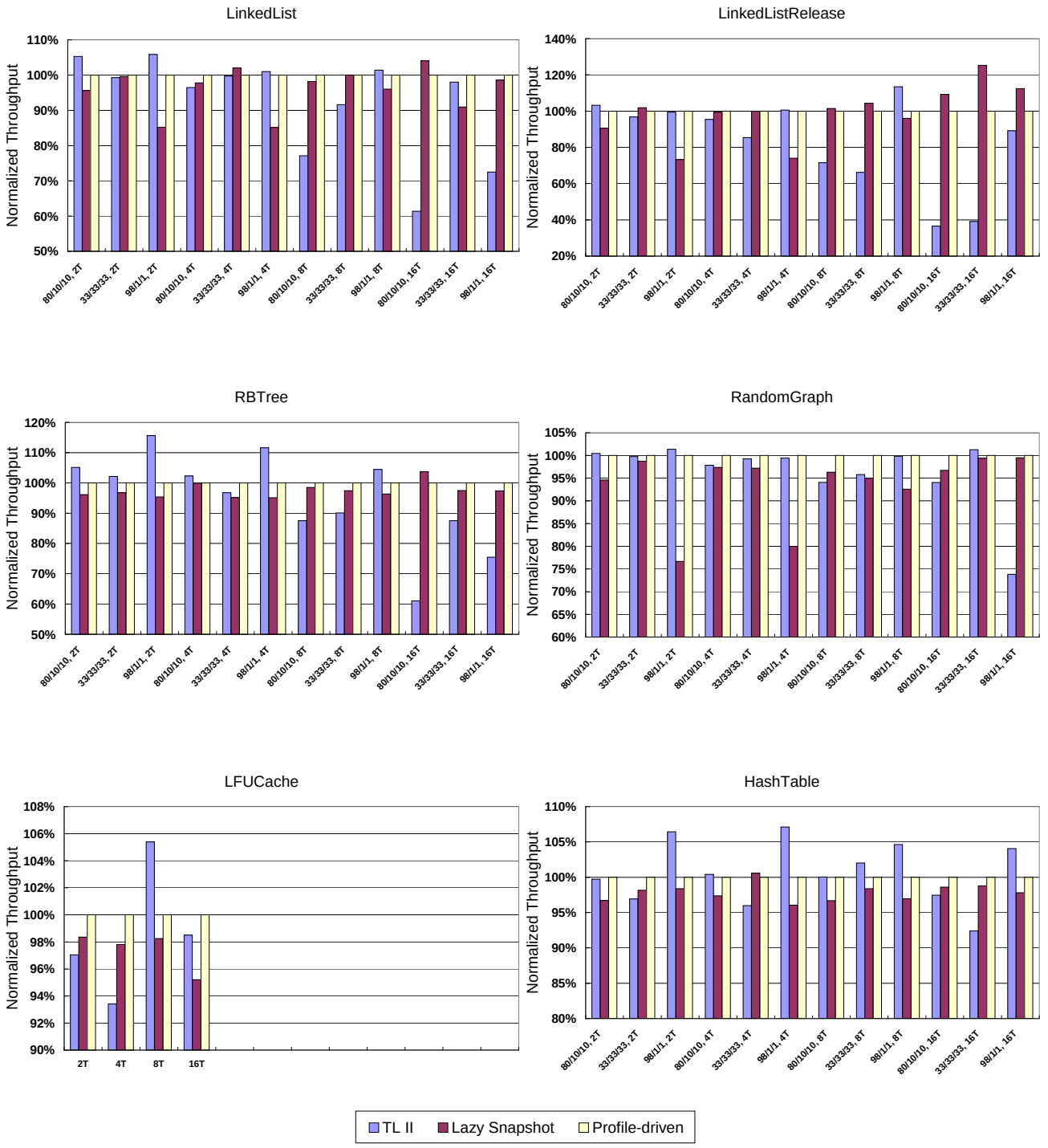

Figure 8. Showing throughput for six different benchmarks, at constant contention levels, with different number of threads

Using a shared software counter to implement the concept of time in time-based validation schemes outperforms using a hardware counter in scenarios where parallelism and contention is low. In scenarios with high parallelism and contetion, schemes that use a hardware counter perform better than schemes using a sofware counter.

In future research, we will investigate the possibility of using different runtime measurements that could suggest changing the validation strategy without performing the runtime profiling of the alternatives. We will also further investigate the practical usability of a threshold hybrid strategy in scenarios where applications perform a mix of short and long-running transactions with small and large number of full validations per each transaction.

References

[1] 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.

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

[3] R. Guerraoui, M. Herlihy, and B. Pochon. Polymorphic contention management. In *DISC '05: Proceedings of the nineteenth International Symposium on Distributed Computing*, pages 303–323. LNCS, Springer, Sep 2005.

[4] 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.

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

[6] 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.

[7] Y. Lev, M. Moir, and D. Nussbaum. PhTM: Phased transactional memory. In *Workshop on Transactional Computing (TRANSACT)*, 2007.

[8] 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.

[9] 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.

[10] T. Riegel, C. Fetzer, and P. Felber. Time-based transactional memory with scalable time bases. In *SPAA '07: Proceedings of the nineteenth annual ACM symposium on Parallel algorithms and architectures*, pages 221–228, New York, NY, USA, 2007. ACM.

[11] 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.

[12] 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.

[13] R. Zhang, Z. Budimlić, and W. N. S. III. Inside time-based software transactional memory. Technical Report TR07-5, Rice University, July 2007.