

False Conflict Reduction in the Swiss Transactional Memory (SwissTM) System

Aravind Natarajan

Department of Computer Engineering
The University of Texas at Dallas
Richardson, TX - 75080, USA
Email: aravindn@utdallas.edu

Neeraj Mittal

Department of Computer Science
The University of Texas at Dallas
Richardson, TX - 75080, USA
Email: neerajm@utdallas.edu

Abstract—Software Transactional Memory (STM) is a programming paradigm that allows a programmer to write parallel programs, without having to deal with the intricacies of synchronization. That burden is instead borne by the underlying STM system. SwissTM is a lock-based STM, developed at EPFL, Switzerland. Memory locations map to entries in a lock table to detect conflicts. Increasing the number of locations that map to a lock reduces the number of locks to be acquired and improves throughput, while also increasing the possibility of false conflicts. False conflicts occur when a transaction that updates a location mapping to a lock, causes validation failure of another transaction, that reads a different location mapping to the same lock. In this paper, we present a solution for the false conflict problem and suggest an adaptive version of the same algorithm, to improve performance. Our algorithms produce significant throughput improvement in benchmarks with false conflicts.

Keywords-Software Transactional Memory; SwissTM; false conflict; history logging;

I. INTRODUCTION

Advances in Very Large Scale Integration (VLSI) design and Computer Architecture have helped sustain an exponential performance improvement in sequential programs, for nearly 50 years. However, with processor designers now shifting their focus towards multi-core systems, parallel programming has acquired new importance. Parallel programming suffers from algorithm design and debugging complexities. Coarse grained locking schemes offer implementation advantages over fine grained approaches, but that is at the cost of performance.

Software Transactional Memory (STM) [1], [2] is an approach that aims to achieve middle ground between performance and ease of programming on parallel architectures. By treating concurrent memory accesses as database transactions that either appear to commit atomically or abort without any trace, consistency can be achieved. Programmers using Transaction Memory (TM) systems are required to explicitly demarcate the shared data accesses within a transactional construct, as they would normally do with critical sections. However, the burden of synchronization is now shifted to the TM system. This is particularly appealing as it does not require a drastic change in programming style. As a result STMs are widely being pursued by a number

of research groups and there are currently several STM implementations available [3]–[9].

The STM design space is very vast, and one of the most fundamental choices is that of a lock based or non-blocking STM. Lock based STM implementations have been shown to perform better than non-blocking ones owing to the lesser validation overhead involved. *SwissTM* [9] is one such lock-based TM implementation, developed by the Distributed Programming Laboratory at EPFL in Lausanne, Switzerland. SwissTM uses a mixed conflict detection scheme [9] and has a two-phase contention manager. It has been shown in [9] that SwissTM outperforms RSTM and TLII.

SwissTM uses a global lock table to detect conflicting transactional accesses to memory locations. However, we have observed that *false conflicts* can significantly impact performance in SwissTM. A false conflict in SwissTM occurs when a committing transaction updating a memory location, x , that maps to a lock causes validation failure of another memory location, y , that maps to the same lock even when y has not been modified in the mean time. Such aborts can impact performance significantly, especially when a frequently read memory location maps to the same lock as a contention hotspot. We propose and implement an algorithm, that we call *History Logging (HL)*, for SwissTM that produces significant performance improvement in benchmarks that suffer from false conflicts. The additional overhead of our approach causes performance to drop in cases where we do not have a significant number of false conflict aborts. In those cases the original SwissTM algorithm, which we term *No Logging (NL)*, performs better than *HL*. Hence, we also present *Adaptive History Logging (AHL)*, that improves upon the throughput of *HL* in such cases.

The remainder of the paper is organized as follows. In Sections 2 and 3 we give a short overview of the types of STMs and introduce SwissTM. Sections 4 and 5 describe the False Conflict Problem and our proposed solutions. We present the benchmarks used in our simulations in section 6, the results in Section 7 and our conclusions in section 8.

II. BACKGROUND AND RELATED WORK

A Transactional Memory system allows an application programmer to write concurrent code, without the

complications of critical sections. There are several choices that need to be made while designing an STM, each impacting performance. One of the most fundamental choices is that of the granularity at which an STM detects conflicts. Word-based STMs [5], [7], [9] detect conflicting accesses to memory words, while Object-based STMs [4], [6], [8], [10] extend object-oriented languages and detect conflicting accesses to objects. STMs can also be classified based on the method of update. In a *direct update* STM system [11], a transaction modifies the shared object in-place during the transaction. The STM system ensures that the object is not updated by another transaction till the transaction commits by using concurrency control mechanisms. The updating transaction maintains an undo-log to restore the system to the original state in case of an abort. In a *deferred update* STM [4], [6], [8]–[10], a transaction updates a private copy of the shared object. Upon commit, the transaction replaces the shared object with its private copy. If the transaction aborts, the copy is discarded.

Non-blocking STMs [4]–[6], [8] use primitives like Compare-and-Swap (CAS) and Load-Linked/Store-Conditional (LL/SC) to achieve atomicity. Blocking Synchronization based STMs [3], [7], [9], [11], [12] use conventional concurrency mechanisms like locks, semaphores and monitors. Though such STMs are prone to deadlock and livelock, those problems can be alleviated by the use of timeouts to detect and abort stalled transactions. STM systems that are blocking have been shown to perform better [9], [12] than those that use non-blocking synchronization owing to lower validation overhead.

Detecting conflicts early in a transaction’s execution can help prevent doomed transactions from doing wasted work. This is particularly useful in situations where two transactions attempt to write to the same object concurrently. However, such a policy does not work well in cases where conflicts can be resolved without aborts. In case of a read-write conflict, both transactions can commit as long as the reader commits first. Detecting conflicts late, i.e. at commit time, is more effective in such cases. Scott et al. [13] proposed a *mixed* conflict detection scheme, where write-write conflicts are detected early, and read-write conflicts are detected late. This scheme exploits the advantages of both the early and late schemes.

III. SWISSTM

SwissTM [9] is a lock-based and word-granularity STM. It provides a *mixed* conflict detection scheme that uses (a) *encounter time locking* for handling write-write conflicts, and (b) *commit time locking* for read-write conflicts. By detecting write-write conflicts early, SwissTM avoids the wasted computation of a doomed transaction. Detecting read-write conflicts late allows greater concurrency. It also provides a *two phase* contention manager that dynamically

switches from the *polite* [14] policy, which favors short transactions, to the *greedy* [15] policy for larger transactions. Every Memory word m , in SwissTM, is mapped to a pair of locks in a global lock table - the read lock, $m.r$ -lock and the write lock, $m.w$ -lock. A transaction T that writes to m acquires $m.w$ -lock at encounter time to prevent other transactions from concurrently writing to m . T also acquires $m.r$ -lock at commit time to prevent other concurrent transactions from reading m and hence observing an inconsistent state. SwissTM uses a global commit counter heuristic [16] to reduce validation overhead. Transactions maintain a read log to help in validation, and a write log to store updates. During validation, a transaction compares the version numbers of locations accessed (as stored in the read log) with the current version numbers as stored in the corresponding read locks, and aborting on a mismatch. During commit, a transaction stores the changes to the memory locations and updates the version numbers.

IV. FALSE CONFLICT PROBLEM

In the previous section we presented a brief description of SwissTM. The number of consecutive memory words that map to a specific lock pair can be specified by the user. By increasing the extent of a lock entry, we can reduce the number of locks that need to be obtained and released by a transaction. Increasing the number of entries that map to a given lock pair reduces the number of locks that need to be acquired, due to data access locality, and hence reduces the validation time [9]. However, this results in increased abort rates due to the introduction of *false conflicts*, as described below.

Let k memory locations $\{x_1, x_2, x_3, \dots, x_k\}$ map to the same lock pair, (w -lock, r -lock). Consider a transaction T_w that owns w -lock and updates location x_j , for some j , $1 \leq j \leq k$. Consider another transaction T_r concurrent with T_w , that only reads location x_i , for some i , $1 \leq i \leq k$, and $i \neq j$. Note that T_r cannot write to a location x_l , $1 \leq l \leq k$, as T_w owns w -lock and hence T_r would have either aborted when it tried to acquire w -lock or waited for T_w to release w -lock. If T_r were to commit before T_w , validation of x_i would succeed. However, if T_w committed first and no transaction updating x_i committed before T_r , it would update r -lock with the new version number. As a result at commit time, T_r would detect a mismatch in version numbers and would abort, even though the location that T_r read is different from the location that T_w updated. This is known as a false conflict.

False conflicts pose a serious problem because they cannot be foreseen by an application programmer and can cause drastic reductions in throughput.

V. SOLUTIONS TO THE FALSE CONFLICT PROBLEM

In this section we present our solutions to reduce aborts due to false conflicts. We call our first scheme *History*

Logging (HL) and our second scheme Adaptive History Logging (AHL).

A. History Logging

Consider a transaction T , that reads memory location m , that maps to read lock entry $m.r\text{-}lock$, with lock table index $m.index$. T validates its read log upon encountering a location with a version number greater than its timestamp. During validation T compares the version numbers stored in the read log, with the current version numbers for those locations. If a mismatch is detected, T aborts.

By storing a small number of the most recent updates made to a lock table entry, in a data structure called the *changeLog*, a transaction whose validation would normally fail can make an informed decision if the conflict is a false conflict, or a legitimate one. In order to be able to successfully detect false conflicts on a lock entry, a validating transaction must have information about all changes that have been made to entries that map to the lock. The *changeLog* has the same number of rows as the lock table, and hence only one transaction (the one holding the corresponding write lock) can write to a row in the *changeLog* at a time, while multiple transactions can concurrently read from a row. Furthermore, each row of the *changeLog* contains a specified number of buckets, which store the version numbers overwritten and the address that was modified during a commit.

If during validation, transaction T detects that the version number $m.version$ of memory location m in its read log, does not match the current version $currentVersion$ stored in $m.r\text{-}lock$, T checks the buckets in the *changeLog* with index $m.index$. In order to ensure consistency, all operations in the *changeLog* start at the index of the least recently updated entry. This enables a validating transaction to check if the *changeLog* is being updated concurrently. If T detects that the version mismatch is due to a false conflict, it replaces the version number in its read log with $currentVersion$. Otherwise, T aborts.

1) *Validation Algorithm*: Consider a transaction T_v that has $(version, address)$ pair, (v_1, a_1) in its read log. During validation, T_v determines that the version number associated with a_1 , in the read lock, is now v_2 . In the *NL* (no logging) scheme T_v would restart. However, in the *HL* scheme, we use the information stored in *changeLog* to determine if address a_1 was modified. All entries in the *changeLog* are initially invalid.

The algorithm for validation proceeds as follows: T_v first determines the index of the least recently updated bucket, $oldestIndex$. T_v then obtains the version number, $oldestVersion$, stored at that bucket in *changeLog*. If $oldestVersion$ is larger than v_1 or is invalid, T_v rolls back. Otherwise, T_v proceeds to check the *addressModified* fields for all the buckets in the *changeLog*. If a_1 has been modified in a bucket and the corresponding version number is greater than or equal to v_1 , T_v rolls back. Finally, After checking all

addresses in the *changeLog*, T_v ensures that the *changeLog* was not concurrently modified during validation. This is done by reading the version number stored at $oldestIndex$ and comparing it with $oldestVersion$. If both values are same, meaning *changeLog* was not concurrently modified, T_v updates (v_1, a_1) to (v_2, a_1) . Otherwise, T_v rolls back.

Algorithm 1: Validation Algorithm for HL

```

validate( $T_v$ )
begin
  foreach entry in read-log do
    if entry.version  $\neq$  currentVersion then
      if isFalseConflict( $T_v$ , entry) then
        entry.version  $\leftarrow$  currentVersion ;
      else return false ;
    return true;
  end
end

isFalseConflict( $T_v$ , entry)
begin
  index  $\leftarrow$  mapAddressToIndex(entry) ;
  oldestBucket  $\leftarrow$  least recently updated bucket in
  changeLog[index];
  oldestVersion  $\leftarrow$  version number in oldestBucket ;
  if oldestVersion is invalid or
  oldestVersion > entry.version then
    return false;
  foreach bucket in changeLog do
    if bucket.addressModified = entry.address and
    bucket.version  $\geq$  entry.version then
      return false;
  if oldestBucket is unmodified since its last read
  then return true;
  else return false;
end

```

2) *Commit Algorithm*: Consider a committing transaction T_c , that has validated its read log, and that owns the write-lock $w\text{-}lock$ with index $index$ in the lock table. In order to commit, T_c must store its updates to the *changeLog*. The commit algorithm is as follows: T_c first determines the number of updates made to unique addresses that map to $index$. If the number of updates is greater than the number of buckets in the *changeLog*, T_c invalidates all *changeLogBuckets* that map to $index$, and then proceeds to update the values and release locks. This is because T_c would overwrite its own entries. Otherwise, T_c determines the index of the least recently updated bucket, $oldestIndex$ and uses $oldestIndex$ to T_c obtain the version number $oldestVersion$ stored at that bucket in *changeLog*. T_c then compares the version numbers of subsequent buckets to $oldestVersion$ and invalidates them in case of a match. This ensures that a validating transaction does not see an incomplete set of updates. Then, T_c proceeds to iterate through its write log,

storing the recorded version numbers and modified addresses in *changeLog*. After all changes have been recorded, T_c stores the new value of *oldestIndex* updates the memory locations and releases the write locks.

Algorithm 2: Commit Algorithm in HL

```

commit ( $T_c$ )
begin
  if isReadOnly ( $T_c$ ) then
    return
  Obtain read lock on all entries in write log;
  obtain new version number;
  if not(validate $T_c$ ) then
    release locks and roll back ;
  else
    foreach entry in write-log do
      if updateCount > changesRemembered then
        invalidate all buckets for that index;
      else
        store updates to changeLog;
        update index of oldest bucket;
        update memory locations;
        release locks;
    end
end

```

B. Adaptive History Logging

In the *AHL* scheme, we keep track of the number of transactional aborts that occur on a particular lock table entry. Initially we start in the original *NL* scheme, where we abort on any mismatch in version numbers. If the abort count on a certain lock table entry exceeds a user-defined threshold, we switch to the *HL* mode for that lock entry. The system stays in *HL* mode for a small *window* of aborts on that lock entry, during which we track the number of false conflicts that have occurred, in addition to the number of aborts. If we determine that logging is proving beneficial, we continue in *HL* mode. Otherwise, we reset the abort count and switch back to *NL* scheme. In this way we keep switching between the modes if aborts continue to occur, while minimizing the time spent logging if it does not provide any benefit. For lock entries that do not incur a significant number of aborts transactions continue in accordance with the *NL* scheme.

Thus, the algorithm proceeds in two phases. In phase 1, the transactions proceed in accordance with the *NL* scheme, where we do not record any entries in the log and abort on any version mismatch. If the number of aborts on a lock entry exceeds a threshold, we switch to phase 2, where committing and validating transactions advance according to *HL*. If aborts continue to occur, it means that they are likely due to legitimate conflicts and hence we switch back to phase 1 to reduce overhead.

1) *Validation Algorithm:* Let *abortCount* denote the number of validation aborts that have occurred and *falseConflictCount* denote the number of false conflicts that have been detected on a lock.

Phase 1: In this phase, the boolean *logging* field of the *changeLog*, for that index, has a value *false*. The validating transaction T_v , on detecting a version number mismatch, increments the abort count (atomically, using the *atomic_ops* library [17]) for that index and rolls back.

Phase 2: In phase 2, a validating transaction detecting a version mismatch, proceeds to read *changeLog* as in *HL*, to determine if a false conflict has occurred. If it has, then T_v increments *falseConflictCount*, updates the version number in its read log, and proceeds. Otherwise, it increments *abortCount* and aborts.

2) *Commit Algorithm:* A committing transaction T_c first checks if logging is already enabled for every lock entry that it updates. If logging is enabled and proving beneficial, the transaction stores the changes in the *changeLog*, updates memory locations and releases locks. Otherwise, the transaction disables logging for that lock entry, invalidates the *changeLog* and resets the value of *abortCount*.

If logging is not enabled for an entry, at commit time T_c checks the value of *abortCount* for that entry and determines if logging is necessary. If the value of *abortCount* is greater than *abortThreshold*, T_c stores the changes in the *changeLog*, enables logging for that entry, updates the memory locations and releases locks. If, however, it determines that logging is not necessary, it simply updates memory locations and releases locks.

VI. BENCHMARKS

In this section we present an overview of the different benchmarks we use in our experiments.

A. Hash Table

We compare the performance of the different schemes on a three-tier hash table. Transactions perform insert, delete and look-up operations on values chosen from three non-overlapping keyspaces, K_1 , K_2 and K_3 , where $|K_1| < |K_2| < |K_3|$. The probability of choosing a value from K_1 is higher than that of K_2 , which is in turn higher than that of choosing a value from K_3 . The three tier approach simulates scenarios where certain keys are generated more frequently than others. Tests were performed on two variants of the hash table benchmark — one with resizing disabled and the other with resizing enabled. The resizing variant provides us with long transactions that resize the table in addition to the read-only look-up transactions and small insert and delete transactions of the non-resizing variant.

B. Red-Black Tree

A variant of the red-black tree benchmark provided by SwissTM is also used to make performance comparisons.

Transactions insert, delete and look-up values in the red-black tree. Here too, we use a three-tier approach where values are chosen with different probabilities from three non-overlapping keyspaces K_1 , K_2 and K_3 , where $|K_1| < |K_2| < |K_3|$. This benchmark provides us with a mixture of small and medium size transactions with which we run our simulations.

VII. SIMULATION RESULTS

We compare the performance of *HL* and *AHL* with that of the default SwissTM scheme. All tests were carried out on a 2 processor dual-core AMD Opteron 285 2.6 GHz 1024 KB cache machine with 16 GB of RAM running the Linux Operating System. We use the default lock table size of 2^{22} entries provided, for the *NL* scheme experiments, and reduce the size of the lock table to 2^{18} entries for the *HL* and *AHL* tests. This ensures that the memory utilization of the logging schemes is approximately the same as the original SwissTM algorithm. We set the number of buckets in the *changeLog* to be 4. For *AHL* runs, *abortThreshold* is set to 100,000 and *window* to 2500. All results were averaged over 50 runs. Each run was carried out for 50 seconds.

A. Hash Table

Read-write, *write-dominated* and *read-dominated* workloads were used to simulate both variants of the hash table. In *read-write* workload we used 40% read, 40% insert and 20% delete transactions. For the *write-dominated* workload we used 70% insert, 20% delete and 10% read transactions. In the *read-dominated* workload case we used a mixture of 70% read, 20% insert and 10% delete operations. We set $|K_1|$, $|K_2|$ and $|K_3|$ to be 100000, 1000000 and 10000000 respectively. The probability of choosing a key from K_1 was set to 0.8, that of choosing a key from K_2 to be 0.15 and with the remaining probability from K_3 . For the resizing variant the initial size of the table was set to 10 buckets, with resizing being performed every time the number of elements exceeded twice the number of buckets in the table. In the case without resizing, the size of the table was fixed at 1310720 buckets.

Experiments revealed that the hash table benchmark suffered from significant performance degradation owing to false conflicts. This happens when the memory locations that store the number of elements and the size of the hash table map to the same lock. The former is updated very frequently, while the latter is read by every transaction to determine the bucket to which a key hashes. Therefore, validation of the table size does not succeed in many cases, accounting for a drastic reduction in performance.

Without Resizing: Figures 1 and 2 show the comparison of the performance of the different schemes in the read-write and write-dominated workload configurations respectively. In both cases we observe that the performance of all schemes is approximately the same when we have 2 threads. As

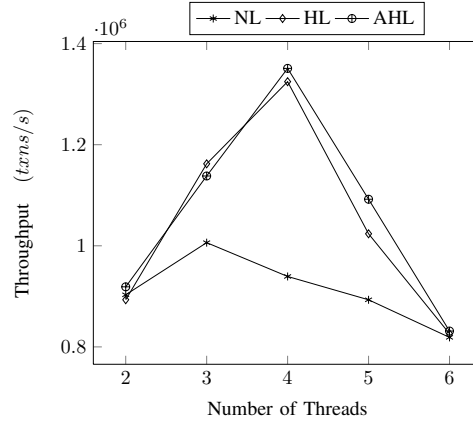


Figure 1. Three Tier Hash Table without resize, Read-Write Workload.

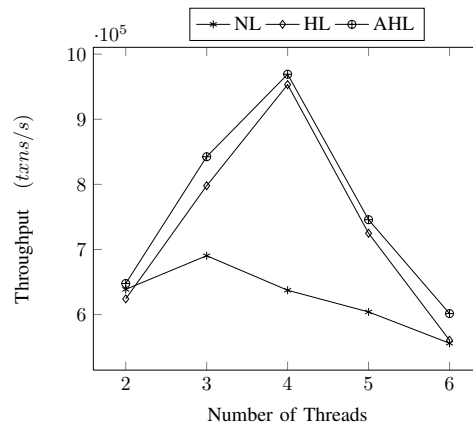


Figure 2. Three Tier Hash Table without resize, Write Dominated Workload.

the number of threads increases we observe that *HL* and *AHL* outperform *NL*. Performance improvements of 49% and 52% were observed for *HL* and *AHL* respectively, for the write-dominated workload. For the read-write workload *HL* outperformed *NL* by 41%, while *AHL* showed a 43% improvement over *NL*, in the 4 threaded runs. As the number of threads exceeds the number of available cores, the performance deteriorates, as can be seen from the figures. The simulation results for the read dominated workload are shown in Figure 3. For the runs with 4 threads *HL* and *AHL* prove beneficial and a throughput improvement of 14% and 17% respectively are observed over *NL*. Due to the increased percentage of read only transactions in this workload, the additional overhead incurred in logging causes the performance to drop relative to *NL* for other runs.

With resizing: The results of the read-write and write-dominated simulations for the resizing variant of the three-tier hash table are shown in Figures 4 and 5 respectively. In both cases we observe that *HL* and *AHL* outperform *NL* in the case of 3, 4 and 5 threads. Since our test system had 4 cores, the increased overhead of the logging schemes

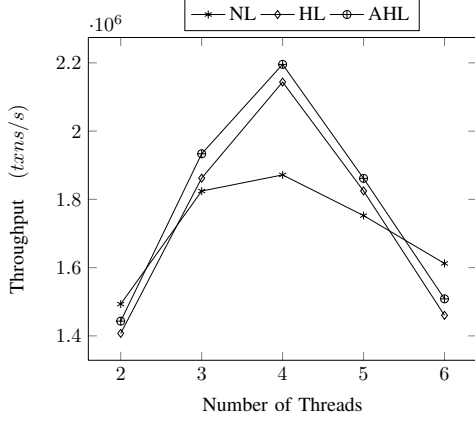


Figure 3. Three Tier Hash Table without resize, Read Dominated Workload.

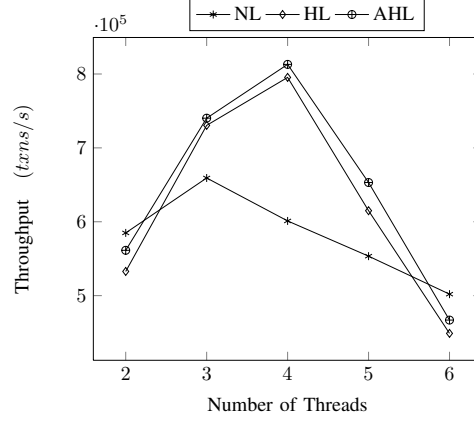


Figure 5. Three Tier Hash Table with resize, Write Dominated Workload.

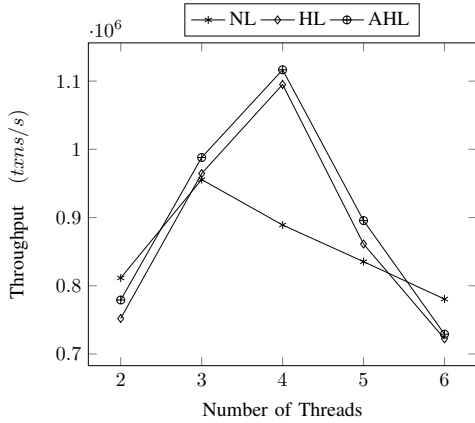


Figure 4. Three Tier Hash Table with resize, Read-Write Workload.

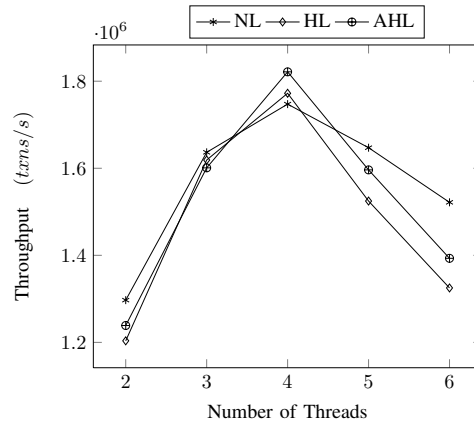


Figure 6. Three Tier Hash Table with resize, Read Dominated Workload.

coupled with the thread switching overhead is responsible for the drop in performance relative to *NL* for the case with 6 threads.

For the read dominated case, as shown in Figure 6, it can be seen that the overhead incurred by the logging schemes negates any performance advantage obtained by reducing the number of false conflict aborts.

Analysis of Aborts: Figure 7 shows the number of validation failure aborts for the write dominated workload for the non-resizing variant of the hash table. The results for the read-write and read-dominated workloads were similar, and hence we present the results only for the write dominated case. It can be seen that this abort count is significantly less for the *HL* and *AHL* schemes, relative to *NL*, across all runs. The number of write-locked aborts for the same workload is shown in Figure 8. Write-locked aborts occur when a transaction's contention manager determines that it must restart because the location it is trying to write to has already been locked by another transaction. It is clear from Figure 8 that the *HL* and *AHL* schemes have a significantly higher number of such aborts. We believe that this is due to

the increased transactional length in the logging schemes. It must also be noted here that as the number of threads exceeds the number of cores, the number of write-locked aborts significantly increases, for all schemes. We believe that this is once again due to the increased transaction length, now including the thread switching overhead.

Finally, Figure 9 shows the validation aborts as a percentage of the total aborts. It can be seen that for the *NL* scheme, validation aborts form more than 99% of the total aborts for the cases with 2, 3 and 4 threads, while forming greater than 90% aborts in the cases with 5 and 6 threads. For the *HL* and *AHL* schemes the number of validation aborts as a fraction of the total aborts is less.

B. Red-Black Tree

We used a read-write workload consisting of 40% read, 40% insert and 20% delete transactions, to compare the performance of the different schemes on this benchmark. Here, we set $|K_1|$, $|K_2|$ and $|K_3|$ to be 4095, 65535 and 1048575 respectively. The probability of choosing a key

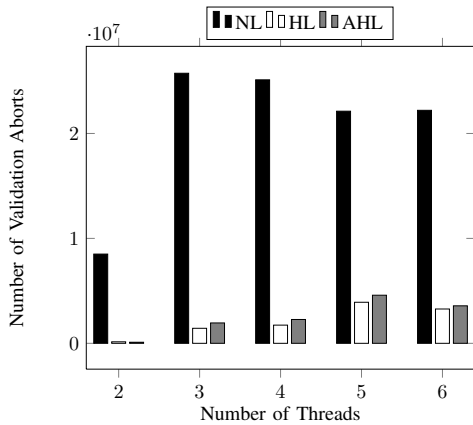


Figure 7. Validation Abort Count Comparison, Three Tier Hash Table without resize, Write Dominated Workload.

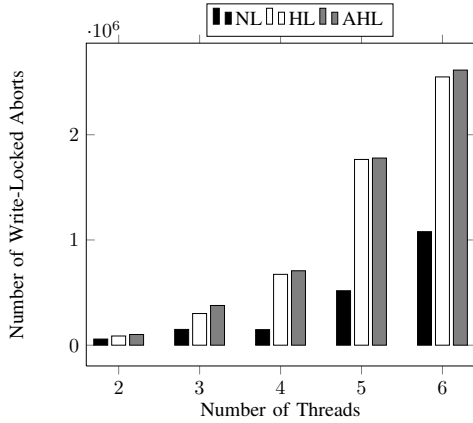


Figure 8. Write-Locked Abort Count Comparison, Three Tier Hash Table without resize, Write Dominated Workload.

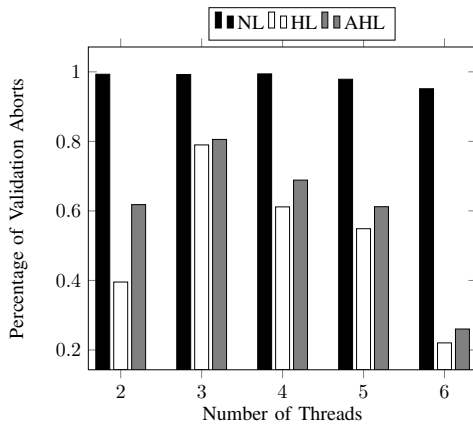


Figure 9. Validation Abort Fraction Comparison, Three Tier Hash Table without resize, Write Dominated Workload.

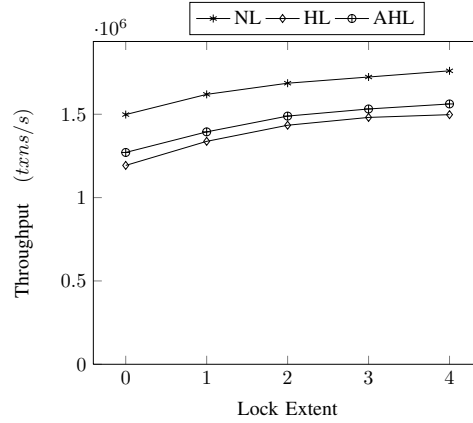


Figure 10. Three Tier Red-Black Tree, Read-Write Workload.

from K_1 was set to 0.8, that of choosing a key from K_2 to be 0.15 and with the remaining probability from K_3 . For this benchmark we compare the transactional throughput versus the lock extent. The lock extent is the logarithmic value of the number of consecutive words that map to a lock. The lock extent was varied from 0 (separate lock for each consecutive word) to 4 (16 consecutive words mapping to the same lock), and tests were carried out using 4 threads. Evaluation of the results show that logging does not prove beneficial in this benchmark. Hence, *HL* and *AHL* do not perform as well as *NL* as can be observed in Figure 10. We see a performance degradation of about 20% for the *HL* scheme and between 12% - 15% for *AHL*. The increased logging and validation overhead accounts for a significant fall in throughput in *HL*. In the *AHL* case, the commit time overhead incurred in determining if the switch to logging is necessary, causes the performance to drop relative to *NL*. The results for the write dominated and read-dominated workloads showed a similar degradation in performance relative to the *NL* scheme.

The graph also illustrates the advantage of increased lock extent. The throughput in the case lock extent = 4 is, on an average, 22% higher than when lock extent = 0.

VIII. CONCLUSION

In this paper, we have described a logging based scheme to reduce aborts due to false conflicts in SwissTM. To preserve some of the advantages of the original algorithm, we have developed a more adaptive logging scheme. Experimental results have shown that in the cases where false conflicts contributed to a significant drop in performance, our algorithms performed much better than the original SwissTM implementation. Hash tables are very important data structures, forming parts of more complicated applications, and our schemes perform significantly better than the original algorithm. However, in other benchmarks, the increased validation and commit time overhead of the

schemes impacted performance negatively, and the red-black tree demonstrates that shortcoming in our algorithm. In the near future we intend on observing the performance of our algorithms with other benchmarks like STMBench7 [18] and the STAMP benchmark suite [19], that are perhaps more indicative of the applications handled by STMs. We would also like to observe the impact of false conflicts in other STM implementations and extend the applicability of our work to those systems as well.

REFERENCES

- [1] M. Herlihy and J. E. B. Moss, "Transactional memory: Architectural support for lock-free data structures," in *Proceedings of the 20th Annual International Symposium on Computer Architecture*, May 1993.
- [2] N. Shavit and D. Touitou, "Software transactional memory," in *Proceedings of the 14th ACM Symposium on Principles of Distributed Computing*, Aug 1995.
- [3] J. Gottschlich and D. A. Connors, "Dracostm: A practical c++ approach to software transactional memroy," in *Proceedings of the 2007 ACM SIGPLAN Symposium on Library-Centric Software Design*, Oct 2007.
- [4] R. Guerraoui, M. Herlihy, and B. Pochon, "Polymorphic contention management," in *Proceedings of the 19th International Symposium on Distributed Computing*, Sept 2005.
- [5] T. Harris and K. Fraser, "Language support for lightweight transactions," in *Object-Oriented Programming, Systems, Languages, and Applications*, Oct 2003.
- [6] V. J. Marathe, M. F. Spear, C. Heriot, A. Acharya, D. Eisenstat, W. N. Scherer III, and M. L. Scott, "Lowering the overhead of software transactional memory," in *ACM SIGPLAN Workshop on Transactional Computing*, Jun 2006.
- [7] P. Felber, C. Fetzer, and T. Riegel, "Dynamic performance tuning of word-based software transactional memory," in *Proceedings of the 13th ACM SIGPLAN Symposium on Principles and practice of parallel programming*, Feb 2008.
- [8] M. Herlihy, V. Luchangco, M. Moir, and I. William N. Scherer, "Software transactional memory for dynamic-sized data structures," in *Proceedings of the 22nd ACM Symposium on Principles of Distributed Computing*, July 2003.
- [9] A. Dragojević, R. Guerraoui, and M. Kapalka, "Stretching transactional memory," in *Proceedings of the 2009 ACM SIGPLAN conference on Programming language design and implementation*, June 2009.
- [10] K. Fraser, "Practical lock freedom," Ph.D. dissertation, Cambridge University Computer Laboratory, 2003.
- [11] B. Saha, A.-R. Adl-Tabatabai, R. L. Hudson, C. C. Minh, and B. Hertzberg, "Mcrst-stm: a high performance software transactional memory system for a multi-core runtime," in *Proceedings of the 11th ACM SIGPLAN symposium on Principles and practice of parallel programming*, March 2006.
- [12] D. Dice, O. Shalev, and N. Shavit, "Transactional locking II," in *Proceedings of the 20th International Symposium on Distributed Computing*, Sept 2006.
- [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*, July 2006.
- [14] W. N. Scherer, III and M. L. Scott, "Advanced contention management for dynamic software transactional memory," in *Proceedings of the 24th annual ACM symposium on Principles of distributed computing*, May 2005.
- [15] R. Guerraoui, M. Herlihy, and B. Pochon, "Toward a theory of transactional contention managers," in *Proceedings of the 24th annual ACM symposium on Principles of distributed computing*, May 2005.
- [16] T. Riegel, P. Felber, and C. Fetzer, "A lazy snapshot algorithm with eager validation," in *Proceedings of the 20th International Symposium on Distributed Computing*, Sept 2006.
- [17] "The atomic_ops project." [Online]. Available: http://www.hpl.hp.com/research/linux/atomic_ops
- [18] A. Dragojevic, R. Guerraoui, and M. Kapalka, "Dividing Transactional Memories by Zero," in *Proceedings of 3rd ACM SIGPLAN Workshop on Transactional Computing*, Feb 2008.
- [19] C. Cao Minh, J. Chung, C. Kozyrakis, and K. Olukotun, "STAMP: Stanford transactional applications for multi-processing," in *Proceedings of The IEEE International Symposium on Workload Characterization*, Sept 2008.