A Comprehensive Study of Conflict Resolution Policies in Hardware Transactional Memory

Ege Akpinar   Saša Tomić   Adrián Cristal   Osman Unsal   Mateo Valero

Barcelona Supercomputing Center

{ege.akpinar,sasa.tomic,adrian.cristal,osman.unsal,mateo.valero}@bsc.es

Abstract

The underlying conflict resolution policy plays a vital role in the performance of most Transactional Memory (TM) systems and there is not a commonly accepted policy. Work carried out in this field contains a wide choice of conflict resolution ideas and authors tend to make their choices according to their workload characteristics. The conflict resolution policy is especially critical in eager-versioning, eager-conflict management Hardware Transactional Memory (HTM) implementations since these implementations are typically optimized for commit and therefore assume that conflicts are rare.

This paper will make a comprehensive study of conflict resolution policies for eager HTM systems. We will re-visit previously studied policies, as well as proposing new policies to mitigate performance bottleneck scenarios. The policies are then evaluated using the STAMP TM benchmark suite and compared against an HTM baseline that employs an elementary conflict resolution policy. The results indicate that it is possible to achieve performance gains of 15% using carefully designed conflict resolution policies.

1. Introduction

In the last decade, power-density problems have rendered complex uniprocessor architectures obsolete. Instead, multi-core architectures, with simpler processing cores, have been proposed as a solution both by the industry and academia [14, 16]. However, this solution will not be viable if the programmers are not able to take advantage of the potential performance benefits of multi-core architectures. Parallel programming for shared memory systems is seen as a particularly difficult undertaking for the average programmer.

Transactional Memory [11, 15], on the other hand, is a promising approach. Ease of programming and high performance claims make it a strong alternative to conventional methods such as lock mechanisms. However, recent research suggests that ease of programming benefits of TM is not yet very clear [7, 19]. Our focus in this paper is not ease of programming but performance. Although transactional memory can be implemented in software (STM) as well as in hardware, we will focus our attention only on hardware transactional memory (HTM) systems. This is not because we think that STM implementations are slower than HTM and are

thus low performance; there are divergent opinions on this matter [4, 6]. It is rather that, it is more important to choose the right, high-performance implementations for HTMs - in our case conflict resolution policies - since they will be less easy to modify or overhaul once they are implemented in silicon.

Hardware transactional memory systems in the literature [18, 30, 33] display a variety of configurations in terms of their conflict detection [27], conflict resolution and version management. Conflict detection can be carried out eagerly (i.e. conflicts are detected when they occur) or lazily (i.e. conflict detection is deferred until a later time, typically until transaction terminates). Similarly, conflict resolution can be carried out eagerly (i.e. conflicts are resolved when they occur) or lazily (i.e. conflict resolution is deferred until a later time, typically until transaction terminates). In addition to these, version management can also be carried out eagerly (i.e. publishing memory updates as they occur) or lazily (i.e. memory updates are not published until transaction terminates). HTMs are usually categorized based on these design decisions. The conflict resolution policy is especially critical in eager-versioning, eager-conflict management HTM implementations since these implementations are typically optimized for fast commit, while conflict management could have higher performance penalties especially for the case of abort.

Conflict resolution stands for the idea of handling conflicts and it is an important characteristic of any HTM. Simply put, when two transactions access the same memory location and at least one of them tries to update it, only one transaction should be allowed to carry on execution. Lazy conflict resolution systems usually defer such decisions until commit time. Eager conflict resolution systems, on the other hand, try to resolve conflicts as soon as they occur and thus, may have very little information about the conflicting transactions (e.g. when transactions conflict very early in their transactions). Trying to resolve conflicts with limited information is an important problem and will be addressed in this paper.

Several metrics have seen high usage for prioritization among conflicting transactions. Timestamp and 'first requester wins' policies are among the most common ones. Timestamp refers to the age of transactions and typically, older transactions are given higher priority than younger transactions. 'First requester wins' policy, on the other hand, operates like a lock mechanism where transactions *lock* objects as they use it and do not release their *locks* until they terminate (abort or commit). However, care should be taken to ensure that deadlock, a common problem with this approach, is avoided. In addition to metric used, it is also possible to evaluate conflict resolution policies based on whether they allow for stalling or not. If stalling is disabled, at each conflict resolution, only one transaction will be allowed to carry on execution and rest of the conflicting transactions will abort. On the other hand, with stalling enabled, some transactions may choose to stall (i.e. wait) in the
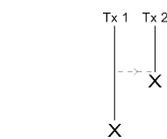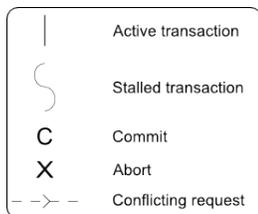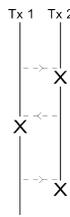
Figure 1. FriendlyFire

Figure 2. Livelock

Figure 3. Deadlock

hope that one or more conflicting transactions will terminate (commit or abort) soon and thus conflicts will be resolved.

With the key goal of improving concurrency, this paper will investigate four different categories of conflict resolution policies: timestamp, size, prioritization according to activity and alternating priorities. We will first talk about conflict resolution in detail, identify common bottlenecks with conflicts and discuss ways to tackle those bottlenecks in Sections 2 and 3. In Sections 4 and 5, a large number of conflict resolution policies will be introduced and results discussed. In Section 6, these policies will be further evaluated against each other as well as against a baseline policy. Conclusions, related work and future work are discussed in Sections 7, 8 and 9 respectively. Following numerous proposals in the field of conflict resolution and contention management, this paper focuses on identifying performance bottlenecks and designing novel conflict resolution policies aimed at tackling the bottlenecks.

2. Challenges in Choosing the Right Conflict Resolution Policy

Conflict resolution plays an essential role in performance. A poorly designed conflict resolution mechanism may degrade performance significantly and even lead to deadlock or livelock. For this reason, conflict resolution requires careful consideration. There are several important design goals. Firstly, it is desirable to guarantee forward progress (i.e. at least one of active transactions eventually commits, at all times). Secondly, deadlock and livelock should be avoided. (Deadlock is the case where transactions start waiting on each other without allowing any transaction to make progress and occurs due to priority inversion or mutual blocking. Livelock is the case where transactions constantly abort without any overall progress.) In addition, it is also desirable to avoid starvation. (Starvation is the case where one or more transactions are constantly blocked by some other transactions. Starvation is rather a fairness issue than performance.) On top of all of these, it is also desirable to have minimal complexity. Conflict resolution mechanisms may require bookkeeping which involve additional hardware support or levy burden on the performance.

For instance, many resolution policies track conflicts with cache line granularity and therefore, they may falsely consider accesses to the same cache line as conflicting accesses even though the accesses may be distinct. This problem is named as *false sharing*. It is possible to increase granularity of conflict detection; however, this entails significant additional complexity.

Transactional memory literature has seen numerous conflict resolution proposals. This paper aims to improve on the work in conflict resolution by introducing more conflict resolution policies and studying their performance. We believe eager resolution systems have more room for improvement related to conflict resolution. In lazy resolution systems, transactions that reach commit stage are usually allowed to commit (do not stall or abort) since it is risky to prevent a transaction from committing with the hope of increasing overall performance. Furthermore, lazy systems have a lot of information about the transactions at the time of conflict resolu-

tion. For these reasons, lazy systems either implement the former convention or resolve conflicts with high amount of information. In contrast, eager resolution systems resolve conflicts based on limited information. Using limited information (as conflicts may occur very early during transactions), the aim is to improve overall concurrency by making local decisions. Since resolution decisions are taken very early, outcomes of different resolution policies are likely to yield different performance outcomes. Therefore, we will focus our attention only on eager systems. Lazy resolution systems will not be covered.

This paper contributes by introducing two new bottleneck scenarios (i.e. InactiveStall and CascadingStall) and devising many unique conflict resolution ideas (such as ideas under Priority or Alternating Priorities categories).

3. Performance Bottlenecks

A conflict pattern refers to an execution scenario among conflicting transactions. Some conflict patterns seem to occur many times throughout execution and thus, they allow room for critical performance gains or loss. For this reason, a conflict resolution policy should take possible conflict patterns into consideration. We will name conflict patterns which may degrade performance significantly as performance bottlenecks.

In [3], authors shed light on increasing concurrency by identifying common case performance bottlenecks. Out of nine bottlenecks (pathologies, as authors name it) presented in the paper, we will focus only on the cases that are applicable to eager systems. In addition, we will identify two more bottlenecks. For convention, naming of **[3]** will be used for the bottlenecks that appear in their paper. Note that suggested remedies may be exact solutions as well as solutions aimed merely at reducing effects of the bottleneck.

FriendlyFire How: A transaction conflicts with (Tx1) and aborts another transaction (Tx2) and later, (Tx1) terminates without success (i.e. abort). Overall, out of two transactions, none made it to commit; thus, all work was wasted.

When: This bottleneck is possible when transactions are allowed to abort other transactions either directly or indirectly, as frequently is the case.

Remedy: A policy in which aborting other transactions is not favored or a policy in which transactions that abort others are given higher priority for commit.

Livelock How: No forward progress occurs overall despite transactions changing state. A typical example is as follows: a transaction (Tx1) aborts another transaction (Tx2) which later aborts the former transaction (Tx1) and this behavior loops.

When: If a transaction that gets aborted by another transaction, is allowed to abort that aborting transaction before it terminates, livelock becomes a critical possibility.

Remedy: A policy in which a transaction that gets aborted by another transaction waits until that aborting transaction is terminated before starting (i.e. Perfect waiting) or a policy in which aborts are
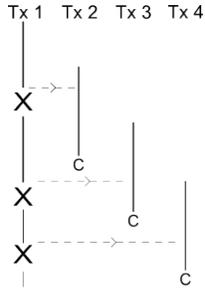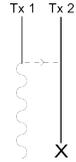
Figure 4. StarvingElder
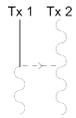

Figure 5. FutileStall
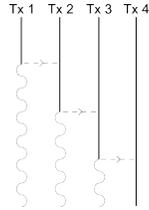

Figure 6. InactiveStall

Figure 7. CascadingStall

recorded so that a transaction that is aborted is not allowed to abort that aborting transaction.

Deadlock How: When two or more transactions wait on each other thus no transaction makes any progress ever. A typical example is as follows: A transaction (Tx2) starts waiting on a transaction (Tx1) which has already been stalled and is waiting on it (Tx2).

When: If no strict ordering among conflicting transactions is maintained (so that conflicting transactions are allowed to acquire conflicting objects and do not release them). Note that stalling behavior often bypasses strict ordering.

Remedy: A definite ordering among conflicting transactions can ensure that only one of the conflicting transactions is allowed to acquire all of the objects it needs while the rest aborts. When stalling is allowed, definite ordering does not suffice and an additional mechanism should be incorporated to detect presence of deadlock.

StarvingElder How: A long transaction (Tx1) constantly gets aborted due to shorter transactions (Tx2, Tx3, Tx4). If it is a finite execution, long transaction will eventually be exempt from this problem since all short transactions will have completed.

When: Stalling is enabled.

Remedy: A policy in which commits are evaluated on a per-core basis and a starving core is forced to take over if rest of the cores have committed recently.

FutileStall How: A transaction (Tx1) stalls on another transaction (Tx2) which eventually aborts. Thus, stalling represents wasted time.

When: Stalling is enabled.

Remedy: A policy in which transactions that stall others are given higher priority for commit; thus, it becomes less likely that they abort after stalling others.

InactiveStall How: A transaction stalls (Tx1) on another transaction (Tx2) which is already stalled. Overall, at least two transactions are stalled and merely waste resources.

When: Stalling on already stalled transactions is enabled.

Remedy: A policy in which stalling is not allowed or stalling on already stalled transactions is not allowed. When a transaction conflicts with a stalled transaction, it may abort it and carry on execution or abort itself instead of stalling.

CascadingStall How: When a transaction (Tx1) stalls on another transaction (Tx2) which then stalls on another transaction (Tx3).

In the worst case, we may have all but one transaction stalled and thus a high amount of wasted resources. Although CascadingStall seems similar to InactiveStall, it actually exploits a much different problem (cascading) as opposed to "stalling on an already stalled transaction".

When: If a transaction that has stalled another transaction is allowed to stall on yet another transaction.

Remedy: A policy in which stalling is not allowed or transactions become less likely to get stalled as they stall other transactions.

(DuelingUpgrades is not included in this paper since the problem is logically inherent in any TM system and its problematic behavior arises due to hardware implementation details which is not the topic of this paper.)

3.1 Perfect Waiting and Perfect Stalling

In some policy derivations, we will make use of ideal backoff and stalling algorithms, which we name as *perfect waiting* and *perfect stalling* respectively.

Perfect waiting refers to an ideal backoff algorithm. When a transaction aborts due to a conflict with another transaction, it should wait until the conflicting transaction terminates (commits or aborts). The simplest remedy is to incorporate a backoff algorithm where transaction will sleep for an arbitrary number of cycles and restart. More advanced algorithms calculate the number of cycles to sleep according to the number of retries the transaction has had, either linearly, exponentially or mixed (e.g. exponential until a threshold and then linear). We will make use of perfect waiting, which ensures that an aborted transaction sleeps precisely until its conflicting transaction terminates. The aborted transaction then wakes up and begins transaction. In [34], a feasible implementation of perfect waiting through thread scheduling modifications is described.

Similar to perfect waiting, perfect stalling refers to an ideal stalling algorithm. Perfect waiting is a very conservative approach and hurts concurrency. For instance with perfect waiting, when a transaction (Tx1) aborts due to a conflict with another transaction (Tx2), it waits until that transaction (Tx2) terminates to begin execution. However, instead of waiting (i.e. Perfect Waiting), it would be more efficient if that transaction (Tx1) started execution right after abort. Perfect stalling ensures that the transaction (Tx1) starts waiting when it first conflicts with the same transaction (Tx2) again. This way, we ensure that transaction (Tx1) does not block the conflicting transaction (Tx2) that has caused its abort earlier and we hope to increase concurrency. This effect is further stronger in cases where conflicting accesses occur near ends of transactions.

4. Methodology

We will derive various conflict resolution ideas and test them against a baseline using STAMP benchmark suite [17]. For baseline, the most basic idea that requires no bookkeeping will be considered. Experiments are carried out using a full system simulator (M5 [1]) modified to support hardware transactional memory. Its default message based coherence protocol is also replaced with a directory based coherence protocol which is more stable.

For comparing conflict resolution policies, we will measure number of ticks spent in parallel sections during applications. More specifically, number of ticks spent after the first transaction to enter a parallel section until the last transaction to exit that parallel section will be measured (named MaxTicksParallelSection).This measurement reflects the total number of ticks that was taken by the whole system to process that parallel section. For better focus on scalability, we will measure sum of MaxTicksParallelSections for all sections in an application and not consider ticks spent during serial sections. Speedup will simply be calculated by comparing

MaxTicksParallelSections values with respect to that of single core executions.

STAMP benchmark suite has been used widely in TM proposals. It contains eight different applications and is designed to represent realistic computational workloads. Applications vary in terms of transaction length, size of read write sets, transaction time and contention levels; thus, the whole suite is a good indicator for any TM system's performance.

5. Conflict Resolution Ideas

In this section, our conflict resolution ideas (some of which resemble already implemented ideas) will be introduced.

5.1 Baseline

For comparison baseline, a very simple policy in terms of book-keeping will be used. In Baseline policy, a transaction that fails to retrieve a cache line (because it is already acquired by another transaction) aborts itself. Thus, it is as if transactions lock cache lines they access. This is a simple policy because it doesn't require any bookkeeping and it can be implemented with very slight modifications to a directory based coherence protocol. The main drawback of this policy is that it risks livelock. However, in our experiments, all STAMP applications ran until completion without problem (e.g. livelock). ('Attacker wins', a policy where requestor of a cache line always takes control of that cache line (aborting other accessors) is also very simple and very common indeed; however, our implementation of 'Attacker wins' failed to complete highly conflicting tests, such as labyrinth, due to livelock.)

5.2 Timestamp

Using timestamps as a basis for conflict resolution is a commonly accepted method [18, 33]. The basic idea is to assign every transaction a timestamp indicating when that transaction started execution. When there is a conflict, resolution is based on the comparison of timestamps of the conflicting transactions (and typically favoring the older one). When a transaction aborts, timestamp of that transaction is either reset or maintained until commit.

We have tested the following policies based on timestamps.

Timestamp-1: At conflicts, older transactions abort the younger ones and carry on execution (no stall).

Avoids: Deadlock, Livelock, StarvingElder, InactiveStall, CascadingStall

Timestamp-2: At conflicts, older transactions start waiting on younger transactions (stall). However, when a younger transaction requests a cache line that is owned by an already stalled older transaction, younger transaction aborts itself (in order to avoid deadlock) and begins perfect waiting.

Avoids: Deadlock, Livelock, FriendlyFire

Timestamp-3: (StarvingElder remedy) For every transaction, a "committed after me" list is maintained. When a transaction commits, that transaction's thread is added to the "committed after me" list of all transactions that are active, aborted or stalled. A transaction's "committed after me" list is reset after its every commit.

At conflicts, transactions whose threads are present in "committed after me" list of conflicting transactions are aborted. If conflict persists after these aborts, second configuration's resolution policy is adopted.

(InactiveStall remedy) In addition to the deadlock mechanism, when an older transaction requests a cache line that is owned by an already stalled younger transaction, younger transaction is aborted. Thus, InactiveStall is avoided.

Avoids: Deadlock, Livelock, StarvingElder, InactiveStall

Timestamp-4: Same as Timestamp-3 configuration except for its InactiveStall policy. Instead of aborting the younger transaction, older transaction is aborted at InactiveStall case.
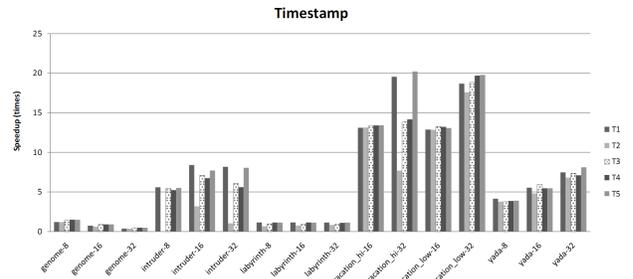

Figure 8. Speedup results for the ideas based on timestamp of transactions. Applications where no significant difference is found are omitted. T1-T5 refer to Timestamp-1 - Timestamp-5 respectively.

Avoids: Deadlock, Livelock, StarvingElder, InactiveStall, FriendlyFire

Timestamp-5: Naïve timestamp and stall policy (Timestamp-2) with perfect stalling enhancement.

Avoids: Deadlock, Livelock, FriendlyFire

Results As Figure 8 suggests, although Timestamp-3 and Timestamp-4 incorporate mechanisms to tackle several types of performance bottlenecks, their added complexity do not pay off and in several tests (e.g. intruder-16, intruder-32 and vacation_hi-32), they get outperformed by Timestamp-1 and Timestamp-5. Overall, as can be seen in the figure, Timestamp-2 yields the worst performance whereas Timestamp-5, which only adds perfect stalling to Timestamp-2, is the best performer. This can be considered a good indicator of how beneficial perfect stalling can be.

5.3 Size

As it is ultimately desirable to reduce amount of wasted work, it is a good idea to abort transactions that have carried out less work as opposed to those that have produced more work. For this reason, transaction size will be used as basis, which is a strong, although not precise, indicator for amount of work carried out. Transaction size denotes the summation of sizes of read-set and write-set of a transaction and its use in conflict resolution has examples in the literature [13]. It is also possible to track the total number of read or write requests issued by a transaction (instead of transaction size); however, the added complexity of tracking every access makes it unfeasible.

A major complexity in using size as a resolution basis arises from the fact that size changes throughout execution. For example, assume that transactions A and B conflict, and transaction A wins the conflict because it is larger. Then assume that transactions A and C conflict where transaction C wins. It is possible that transaction B might grow large in size, enough to beat transaction C.

In other words

Conflict1 -> A is larger than B

Conflict2 -> C is larger than A

Conflict3 -> B is larger than C

It is important to note that, although priority ordering between transactions may change during the same transaction block, deadlock is not possible.

Largeness factor: When comparing two transactions in terms of size, we use a largeness factor so that for a largeness factor l, a transaction is allowed to take over execution only if it is l times larger than the conflicting transaction. It is possible that none of the conflicting transactions are larger than each other by a factor. For instance, in the case where all transactions have the same size (and $l > 1$), none of the transactions will have priority over another. In such a case, in order to avoid deadlock and livelock, current owner of the conflicting cache line is favored.
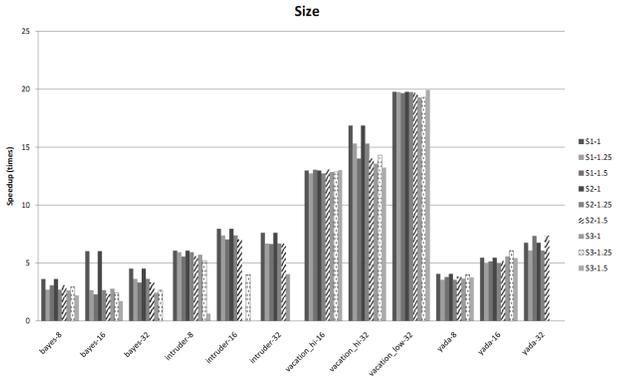
Figure 9. Speedup results for the ideas based on transaction size. Applications where no significant difference is found are omitted. S1-S3 refer to Size-1 - Size-3 and the accompanying number represents largeness factor.

Below is a summary of three different categories of configurations we have tried. For every configuration category, we have tried three different largeness factors 1, 1.25 and 1.5.

Size-1 At conflicts, larger transactions take over and smaller conflicting transactions are aborted. There is no stalling.

Avoids: Deadlock, Livelock, StarvingElder, FutileStall, InactiveStall, CascadingStall

Size-2 At conflicts, larger transactions are favored. However, at conflicts, owner of the conflicting cache line is allowed to resume execution regardless of its size. When a transaction requests a cache line from a larger stalled transaction, it aborts itself and restarts execution. When the aborted transaction again conflicts with the same larger transaction, it is stalled (perfect stalling).

Avoids: Deadlock, Livelock, FriendlyFire

Size-3 At conflicts, larger transactions are favored. However, at conflicts, owner of the conflicting cache line is allowed to resume execution regardless of its size. When a transaction requests a cache line from a larger stalled transaction, small transaction aborts itself and starts perfect waiting.

Avoids: Deadlock, Livelock, FriendlyFire

Results Firstly, when we consider effects of different largeness factors, we can deduce that a factor of 1 performs the best for Size-1 and Size-2 whereas a factor of 1.25 performs the best for Size-2 (Figure 9). When we consider the categories, it is safe to claim that Size-1 and Size-2 perform very similarly and Size-3 is the worst performer.

5.4 Prioritization According to Stalled and Aborted Transactions

Direct prioritization according to stalled and aborted transactions is a contribution of this paper. We expect the idea of increasing priorities of transactions as they stall or abort other transactions to remedy some performance bottlenecks, such as FutileStall and CascadingStall.

Below is a summary of the five different prioritization ideas we have tried. All use the following criteria for prioritization among transactions: number of transactions stalled or aborted due to a transaction.

Priority-1 Transactions gain priority when they stall other transactions. When a transaction requests to acquire a cache line that is already acquired by another transaction and if their priorities are equal, then the current owner is allowed to continue execution. Transactions that are already stalled abort themselves if they
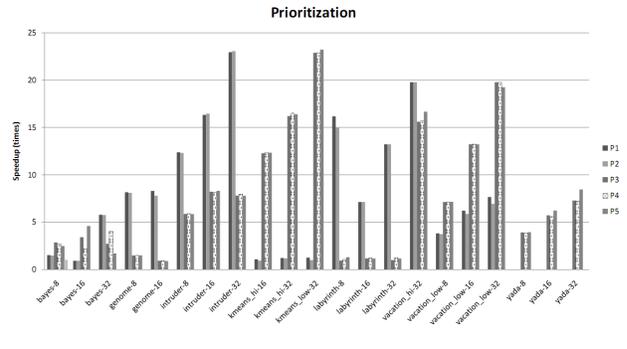

Figure 10. Speedup results for the ideas based on prioritization. Applications where no significant difference is found are omitted. P1-P6 refer to Priority-1 - Priority-6 respectively.

block another transaction; thus, stalled transactions are not allowed to block other transactions.

Avoids: Deadlock, Livelock (FriendlyFire might loop though), InactiveStall

Priority-2 Transactions gain priority as they stall other transactions. In addition, transactions lose priority as they abort other transactions. At conflicts, if the requesting transaction has the highest priority, difference between top two priorities of conflicting transactions is calculated. If this difference is larger than the number of conflicting transactions, then the transaction with the highest priority is allowed to take over. Requesting transaction aborts itself if it doesn't have the highest priority or if its priority difference is not larger than number of conflicting transactions. (i.e. a transaction is allowed to abort n transactions only if its priority is at least n higher than each of conflicting transactions' priorities).

Avoids: Deadlock, Livelock (FriendlyFire might loop though), InactiveStall

Priority-3 Similar to Priority-2. However, transactions gain (not lose) priority as they abort other transactions.

Avoids: Deadlock, Livelock (FriendlyFire might loop though), InactiveStall

Priority-4 Similar to Priority-1. At conflicts, if a transaction loses, it aborts instead of stalling.

Avoids: Deadlock, Livelock (FriendlyFire might loop though), InactiveStall, FutileStall, CascadingStall

Priority-5 Similar to Priority-1, transactions gain priority as they stall others. However, priority is a measure calculated using the number of transactions in conflict. For instance, when a transaction gets stalled due to a conflict with n transactions, all n transactions gain $1/n$ priority.

Avoids: Deadlock, Livelock (FriendlyFire might loop though), InactiveStall

Results We have obtained very varying results and unlike other categories, it is very difficult to identify the best performer. Very interestingly, both P1 and P2 perform very well whereas P3,P4 and P5 all perform very slowly or vice versa as can be seen from Figure 10. Since all ideas actually use the same criteria for resolution, it is challenging to account for these drastic differences; however, it can be deduced that resolution based on prioritization is very sensitive to the way priorities are calculated as well as workload characteristics.

5.5 Alternating Priorities

In addition to performance, fairness is also a desirable attribute for TM systems. In order to maximize fairness, we have tested the following policy: At conflicts, the transaction that has already acquired a cache line is allowed to carry on execution while the re-
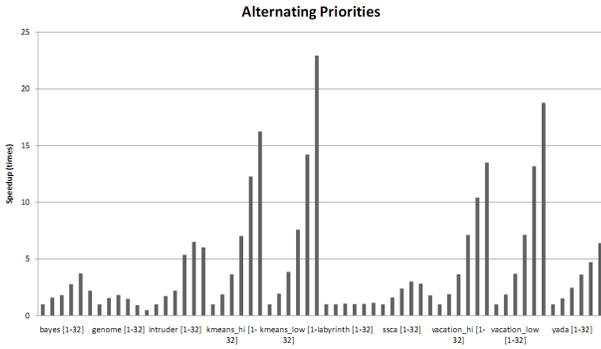
Figure 11. Speedup results for the alternating priorities idea. All STAMP execution results are plotted in the graph (all applications with 1, 2, 4, 8, 16 and 32 core executions).
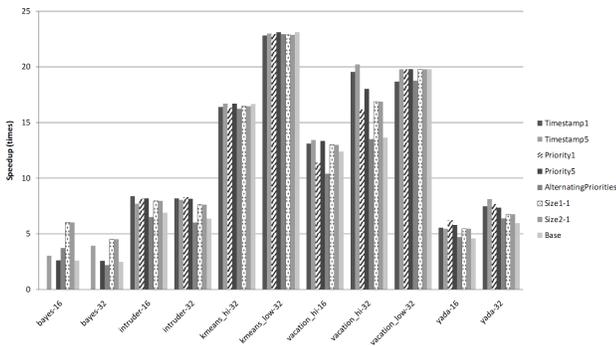

Figure 12. Comparison of top performers from each category. Only tests with significant performance variance is plotted.

questing transaction is aborted. However, when these two transactions conflict again, this time the transaction that has lost the previous conflict will win the conflict. In a sense, all transaction pairs continuously alternate priorities. Perfect waiting is also enabled to avoid deadlock and livelock. (Without perfect waiting, this would be Scherer's Kindergarten policy [13]).

Avoids: Deadlock, Livelock, StarvingElder

Results Except for few tests (e.g. genome and labyrinth), the obtained results perform and scale well (Figure 11).

6. Overall Evaluation

For detailed evaluation, we consider only the best performing policies of each category. When the policies are compared in terms of their speedups, we see little variation across policies for executions with less than 16 cores. For this reason, we will focus our attention on 16 and 32 core execution results only. Figure 12 plots speedup results for candidate best policies and the base policy. Note that for some executions, no significant variation of performance is observed and thus they are omitted in the graph.

Table 1 shows how each chosen candidate policy's speedup compares against that of serial execution (i.e. without transactional memory), based on ticks spent during parallelizable sections (as explained in Section 4). In addition, speedup results of policies are compared against that of the Base policy as a measure of improvement/degradation from the Base policy.

As displayed in Table 1, out of eight candidates chosen, all but one outperforms the base policy. We see that the performance increase amounts to 15% for two of the policies (Timestamp1 and

Prioritization1). The only policy that performs worse than the base policy is the 'Alternating Priorities' policy. This is justifiable since its design goal was fairness, not performance.

Although in many cases, incorporating mechanisms to tackle performance bottlenecks allowed for higher performance, there have been several cases where the simplest policy also performed very well. For instance Timestamp-1, which is one of the best performers (in terms of speedup), is one of the simplest policies studied in this paper. Another best performer, Prioritization-1, can be considered as a contribution of this paper since it was designed to tackle bottlenecks and indeed performed very well.

Another interesting note was to observe that stalling often degraded performance which can be accounted for by the additional bottlenecks introduced due to stalling (such as InactiveStall and FutileStall).

Our goal was to tackle resolving conflicts with limited information; however, it is important to note that transaction sizes within STAMP suite are often very small (transactions hardly ever overflow private caches). Thus, we can expect to evaluate our policies better with more complex applications (with higher number of distinct memory accesses) since longer transactions will allow for more varying outcomes.

7. Conclusion

When hardware complexities of current HTM proposals are taken into consideration, significantly high performance and scalability must be achieved before HTMs are taken seriously in the industry. For this reason, conflict resolution is an important research space. Its design is vital for concurrency and performance as we tried to show in this paper.

This paper has introduced some novel ideas for conflict resolution policies in eager HTMs such as alternating priorities of transactions or prioritizing transactions in many various ways based on the number of transactions they have stalled or aborted. Taking into consideration the most common performance bottlenecks, we were able to improve the performance outcome significantly (i.e. up to 15%). However, as many papers also discuss [3, 9, 13], choosing a best policy is not a trivial task. We have seen that different policies tend to yield highly varying results according to workload characteristics and therefore, deciding on a single policy is risky.

In conclusion, performance bottleneck scenarios should be taken into consideration during conflict resolution policy design. In addition, awareness of different workload characteristics is also vital.

8. Future Work

In addition to performance bottlenecks, there are several other issues that must be handled for TM systems. For future work, we are planning to tackle some issues including support for inevitability [28], transaction nesting, context switch and retry operation [10]. In addition, we believe measuring significance of mentioned performance bottlenecks (by tracing execution logs) for each application would supplement our study thus it will be incorporated in future. Finally, we consider extending our experiments to a larger set of applications (e.g. Eigenbench [12]) and devising more conflict resolution ideas. For instance, we haven't mentioned policies where decisions are based on the execution pattern until that point. We might consider devising new policies where global information is maintained from the beginning of execution to aid in conflict resolution.

9. Related Work

There has been intensive work related to contention management in transactional memory systems. Contention management is

	Timestamp1	Timestamp5	Prioritization1	Prioritization5	Alternating Priorities	Size1-1	Size2-1	Base
genome-16	0,734	0,908	0,921	0,904	0,926	0,907	0,907	0,907
genome-32	0,378	0,483	0,488	0,486	0,489	0,490	0,490	0,480
intruder-16	8,394	7,712	8,157	8,319	6,507	7,956	7,956	6,904
intruder-32	8,185	8,058	8,299	7,792	6,019	7,623	7,623	6,377
kmeans_hi-16	12,274	12,402	12,372	12,334	12,278	12,361	12,361	12,292
kmeans_hi-32	16,405	16,699	16,331	16,389	16,236	16,468	16,468	16,663
kmeans_low-16	14,216	14,223	14,233	14,234	14,204	14,220	14,220	14,246
kmeans_low-32	22,818	22,990	22,943	23,214	22,944	22,875	22,875	23,127
labyrinth-16	1,150	1,136	1,219	1,174	1,030	1,177	1,177	1,147
labyrinth-32	1,149	1,143	1,254	1,155	1,126	1,128	1,128	1,118
ssca2-16	2,765	2,694	2,842	2,801	2,816	2,720	2,720	2,720
ssca2-32	1,792	1,839	1,787	1,809	1,789	1,837	1,837	1,837
vacation_hi-16	13,112	13,436	11,367	13,195	10,407	12,993	12,993	12,394
vacation_hi-32	19,553	20,222	16,172	16,656	13,506	16,876	16,876	13,649
vacation_low-16	12,881	13,066	13,236	13,235	13,171	13,088	13,088	13,001
vacation_low-32	18,686	19,788	19,775	19,252	18,764	19,781	19,781	19,790
yada-16	5,550	5,466	6,207	6,232	4,716	5,464	5,464	4,586
yada-32	7,480	8,121	7,660	8,457	6,390	6,762	6,762	5,962
Linear mean	9,307	9,466	9,181	9,313	8,518	9,151	9,151	8,733
Geometric mean	5,495	5,673	5,661	5,686	5,173	5,548	5,548	5,271
Base - linear	1,066	1,084	1,051	1,066	0,975	1,048	1,048	1,000
Base - geometric	1,043	1,076	1,074	1,079	0,981	1,053	1,053	1,000

Table 1. Speedup results of selected policies and numerical comparison with the base results. 'Linear mean' and 'Geometric mean' rows indicate linear and geometric average values for speedups. 'Base - linear' and 'Base - geometric' rows indicate by what factor a policy differs from the base policy by comparing that policy's linear and geometric averages against those of the base policy. Bayes test is ignored due to lack of data for Timestamp cases.

a broad topic including conflict resolution policies, hardware implementation details of them as well as various others ideas to improve concurrency. Although this paper focuses solely on conflict resolution policies, relevant work to improve contention management, by methods different than modifying conflict resolution policy, will also be mentioned in this section.

In [9], authors work on deriving theory for greedy contention management and evaluate their theoretical findings using contention managers introduced in another related work [13, 23]. A similar work [25] examines three greedy contention managers theoretically. In [8], same authors introduce and analyse *polymorphic contention management*, by which they mean a contention management scheme where contention management policy varies across workloads, transactions and even across different phases of a single transaction.

In [24], authors introduce several contention management policies which are often referenced in the literature. In [23], they randomize their *Karma* contention management policy and argue about its effect on livelock. In [13], they extend their work by including fairness and detailed performance analysis. Scott, along with a number of other authors, detail a strategy and implementation for low complexity contention management in [26]. They argue that their lazy acquire mechanism provides good performance and fairness and avoids common problems such as livelock and starvation.

[3] plays a vital role for our paper. It identifies common performance bottleneck scenarios (most of which are mentioned in this paper) and examines their indicators. For their experiments, they analyze the significance of the pathologies and tackle them with four different HTM system proposals. They conclude that tackling pathologies yield significant performance improvement with low complexity overhead.

In [31], authors tackle starvation problem which is likely for lazy resolution systems. They introduce two schemes (naive and elaborate) and conclude that their schemes provide starvation-free environment with implementation overhead.

In [5], authors develop a mechanism that can be incorporated into software transactional memory systems to reduce collision probabilities. They work on two methods: to prevent conflicting transactions from conflicting again and to calculate probability of collision among transactions. They conclude that they reach significant performance and stability improvement.

Several other methods, which are not mainly based on conflict resolution policies, to improve contention management have also been proposed. In [22], data forwarding between transactions to improve concurrency is implemented and is shown to outperform conventional HTM systems.

In [32], aim is to reduce the time wasted due to rollbacks, via checkpointing. Authors propose a system where transactions predict conflicts and save checkpoints. If prediction turns out to correct, aborted transaction restarts from its most recent nonconflicting checkpoint; thus, saving computation until checkpoint that would otherwise be wasted. Authors conclude that they achieve close to ideal results.

In [29], authors aim to achieve performance gain by tackling false sharing [29] problem. They allow transactions to speculate on stale cache lines and postpone validation. Since transactions don't wait for validation from the coherence protocol, they execute faster; however, if validation later fails, transaction has to abort. Authors argue that their methodology improves performance in cases of false sharing whereas it incurs little overhead otherwise.

A novel approach is introduced in [20]. Authors work on predicting transactional values based on the history of the execution pattern. In another work by the same authors [21], this idea of value prediction is analyzed along with the idea of data forwarding [22]. Authors put forward a new scheme (i.e. LEVC) in an attempt to improve these ideas in terms of performance and overhead.

Another novel approach is introduced in [2], where authors propose repairing transactional data. Aimed at exploiting auxiliary data [2], authors claim that their method achieves significant speedup by repairing auxiliary data and eliminating false sharing.

References

[1] Nathan L. Binkert, Ronald G. Dreslinski, Lisa R. Hsu, Kevin T. Lim, Ali G. Saidi, and Steven K. Reinhardt. The m5 simulator: Modeling networked systems. *IEEE Micro*, 26:52–60, 2006.

[2] Colin Blundell, Arun Raghavan, and Milo M.K. Martin. Retcon: transactional repair without replay. In *Proceedings of the 37th annual international symposium on Computer architecture*, ISCA '10, pages 258–269, New York, NY, USA, 2010. ACM.

[3] Jayaram Bobba, Kevin E. Moore, Haris Volos, Luke Yen, Mark D. Hill, Michael M. Swift, and David A. Wood. Performance pathologies in hardware transactional memory. In *Proceedings of the 34th annual international symposium on Computer architecture*, ISCA '07, pages 81–91, New York, NY, USA, 2007. ACM.

[4] Calin Cascaval, Colin Blundell, Maged Michael, Harold W. Cain, Peng Wu, Stefanie Chiras, and Siddhartha Chatterjee. Software transactional memory: Why is it only a research toy? *Queue*, 6:46–58, September 2008.

[5] Shlomi Dolev, Danny Hendler, and Adi Suissa. Car-stm: scheduling-based collision avoidance and resolution for software transactional memory. In *Proceedings of the twenty-seventh ACM symposium on Principles of distributed computing*, PODC '08, pages 125–134, New York, NY, USA, 2008. ACM.

[6] A. Dragojevic, P. Felber, V. Gramoli, and R. Guerraoui. Why STM can be more than a Research Toy. *Communications of the ACM*, 2010.

[7] V. Gajinov, F. Zyulkyarov, O.S. Unsal, A. Cristal, E. Ayguade, T. Harris, and M. Valero. QuakeTM: parallelizing a complex sequential application using transactional memory. In *Proceedings of the 23rd international conference on Supercomputing*, pages 126–135. ACM, 2009.

[8] R. Guerraoui, M. Herlihy, and B. Pochon. Polymorphic Contention Management. In *Proceedings of the 19th International Symposium on Distributed Computing (DISC'05)*, volume 3724 of *Lecture Notes in Computer Science*, pages 303–323, 2005.

[9] R. Guerraoui, M. Herlihy, and B. Pochon. Towards a Theory of Transactional Contention Managers. In *Proceedings of the 24th ACM Symposium on Principles of Distributed Computing (PODC'05)*, pages 258–264, 2005.

[10] T. Harris, J. Larus, and R. Rajwar. Transactional Memory. *Synthesis Lectures on Computer Architecture*, 5(1):1–263, 2010.

[11] M. Herlihy and J. E. B. Moss. Transactional memory: Architectural support for lock-free data structures. In *Proceedings of the Twentieth Annual International Symposium on Computer Architecture*, 1993.

[12] S. Hong, T. Oguntebi, J. Casper, N. Bronson, C. Kozyrakis, and K. Olukotun. Eigenbench: A simple exploration tool for orthogonal TM characteristics. In *Workload Characterization (IISWC), 2010 IEEE International Symposium on*, pages 1–11. IEEE.

[13] William N. Scherer III and Michael L. Scott. Advanced contention management for dynamic software transactional memory, 2005.

[14] Poonacha Kongetira, Kathirgamar Aingaran, and Kunle Olukotun. Niagara: A 32-way multithreaded sparc processor. *IEEE Micro*, 25:21–29, 2005.

[15] James Larus and Christos Kozyrakis. Transactional memory. *Commun. ACM*, 51:80–88, July 2008.

[16] H. Q. Le, W. J. Starke, J. S. Fields, F. P. O'Connell, D. Q. Nguyen, B. J. Ronchetti, W. M. Sauer, E. M. Schwarz, and M. T. Vaden. Ibm power6 microarchitecture. *IBM J. Res. Dev.*, 51:639–662, November 2007.

[17] C.C. Minh, J.W. Chung, C. Kozyrakis, and K. Olukotun. STAMP: Stanford transactional applications for multi-processing. In *Workload Characterization, 2008. IISWC 2008. IEEE International Symposium on*, pages 35–46. IEEE, 2008.

[18] Kevin E. Moore, Jayaram Bobba, Michelle J. Moravan, Mark D. Hill, and David A. Wood. Logtm: Log-based transactional memory. In *in HPCA*, pages 254–265, 2006.

[19] V. Pankratius, A.R. Adl-Tabatabai, and F. Otto. *Does Transactional Memory Keep Its Promises?: Results from an Empirical Study*. Universitat Karlsruhe, Fakultat fur Informatik, 2009.

[20] Salil Pant and Greg Byrd. A case for using value prediction to improve performance of transactional memory. In *TRANSACT '09: 4th Workshop on Transactional Computing*, feb 2009.

[21] Salil M. Pant and Gregory T. Byrd. Limited early value communication to improve performance of transactional memory. In *Proceedings of the 23rd international conference on Supercomputing*, ICS '09, pages 421–429, New York, NY, USA, 2009. ACM.

[22] Hany E. Ramadan, Christopher J. Rossbach, and Emmett Witchel. Dependence-aware transactional memory for increased concurrency. In *Proceedings of the 41st annual IEEE/ACM International Symposium on Microarchitecture*, MICRO 41, pages 246–257, Washington, DC, USA, 2008. IEEE Computer Society.

[23] William N. Scherer III and Michael L. Scott. Randomization in stm contention management (poster). In *Proceedings of the 24th ACM Symposium on Principles of Distributed Computing*, Las Vegas, NV, Jul 2005. Winner, most popular poster presentation award.

[24] W.N. Scherer III and M.L. Scott. Contention management in dynamic software transactional memory. In *PODC Workshop on Concurrency and Synchronization in Java programs*, pages 70–79. Citeseer, 2004.

[25] Gokarna Sharma, Brett Estrade, and Costas Busch. Window-based greedy contention management for transactional memory. *CoRR*, abs/1002.4182, 2010.

[26] Michael F. Spear, Luke Dalessandro, Virendra J. Marathe, and Michael L. Scott. A comprehensive strategy for contention management in software transactional memory. In *Proceedings of the 14th ACM SIGPLAN symposium on Principles and practice of parallel programming*, PPoPP '09, pages 141–150, New York, NY, USA, 2009. ACM.

[27] Michael F. Spear, Virendra J. Marathe, William N. Scherer Iii, and Michael L. Scott. Conflict detection and validation strategies for software transactional memory. In *In Proc. of the 20th Intl. Symp. on Distributed Computing*, 2006.

[28] Michael F. Spear, Michael Silverman, Luke Daless, Maged M. Michael, and Michael L. Scott. Implementing and exploiting inevitability in software transactional memory. In *In ICPP '08: Proc. 37th IEEE international conference on parallel processing*, pages 59–66, 2008.

[29] F. Tabba, AW Hay, and JR Goodman. Transactional value prediction. *4th ACM SIGPLAN Wkshp. on Transactional Computing*, 2009.

[30] Saša Tomić, Cristian Perfumo, Chinmay Kulkarni, Adrià Armejach, Adrián Cristal, Osman Unsal, Tim Harris, and Mateo Valero. Eazyhtm: eager-lazy hardware transactional memory. In *Proceedings of the 42nd Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO 42, pages 145–155, New York, NY, USA, 2009. ACM.

[31] M. M. Waliullah and Per Stenstrom. Starvation-free commit arbitration policies for transactional memory systems. *SIGARCH Comput. Archit. News*, 35:39–46, March 2007.

[32] MM Waliullah and P. Stenstrom. Intermediate checkpointing with conflicting access prediction in transactional memory systems. In *Parallel and Distributed Processing, 2008. IPDPS 2008. IEEE International Symposium on*, pages 1–11. IEEE, 2008.

[33] Luke Yen, Jayaram Bobba, Michael R. Marty, Kevin E. Moore, Haris Volos, Mark D. Hill, Michael M. Swift, and David A. Wood. Logtmse: Decoupling hardware transactional memory from caches. In *In HPCA 13*, pages 261–272, 2007.

[34] C. Zilles and L. Baugh. Extending hardware transactional memory to support non-busy waiting and non-transactional actions. In *TRANSACT: First ACM SIGPLAN Workshop on Languages, Compilers, and Hardware Support for Transactional Computing*. Citeseer, 2006.