Toxic Transactions *

Yujie Liu and Michael Spear
Lehigh University
{yul510, spear}@cse.lehigh.edu

Abstract

In Transactional Memory workloads, conflicts among transactions necessitate that some transactions either block, or else abort. The more common approach, aborting, is under-studied. Much effort has explored the use of schedulers and gatekeepers when the global abort rate is high. Handling short spikes in the abort rate, especially due to a single transaction with a high likelihood of conflict, is less well understood.

We refer to such transactions as "toxic", and suggest that they be *prioritized*, to prevent repeated aborts. To this end, we propose an Hourglass Contention Manager, which blocks some transactions at their begin point whenever a toxic transaction is detected. We show this mechanism to offer low overhead, and argue that it is an attractive alternative to existing contention management policies in transactional memory.

1. Introduction

In the database community, the incidence of deadlocks indicates one of a few possible scenarios for a workload, with each of these scenarios best served by a different resolution strategy [18]. In the worst case, deadlocks are frequent, due to high contention, and the best approach is to introduce a "gatekeeper" that limits the number of active transactions to a level that reduces contention to a manageable level. On the spectrum from "no aborts" to "gatekeeper required", there are several interesting points, each of which has a specific recommendation on how to improve throughput.

An obvious question then is how to map these conditions to Transactional Memory [22] (TM). Whereas a database will typically only abort a transaction after detecting a true deadlock, a TM runtime typically uses optimistic concurrency control, and thus aborts transactions whenever failing to do so *might* lead to inconsistent accesses to memory. We assume that the correspondence between aborts in an optimistic TM implementation and deadlocks in a database are equivalent. Given this mapping, we observe that recent efforts to resolve conflicts in TM by employing a scheduler [2–5, 14, 15, 20, 24, 29] or by limiting the number of in-flight transactions [36] are fulfilling the role of a "gatekeeper". In such a light, these efforts can be understood as different approaches to reducing the latency of gatekeepers to a magnitude acceptable for transactions running at memory speed.

Another long-standing approach to addressing aborts is to use randomized exponential backoff. While the use of backoff is usually traced back to the Ethernet protocol, again, there is a clear explanation in databases: when deadlock can be characterized by a small and highly-connected graph (e.g., a handful of transactions all conflict with each other), then gently perturbing the rate at which those transactions retry will suffice to prevent the deadlock from repeating. In nonblocking TM implementations, this perturbation has the additional benefit of preserving the progress guarantees of the TM, making it an appealing approach to obstruction-freedom [21, 28].

When perfect knowledge of transaction conflicts is available at run-time (e.g., in hardware TM (HTM)), stalling some transactions at commit time [30] can prevent some aborts. Like backoff, stalling is a local decision that perturbs the order of transactions. Unlike backoff, stalling can very precisely prevent asymmetric conflicts (reader vs. writer) from causing aborts.

There are two additional problematic abort behaviors, neither of which is especially well studied in the context of TM. The first is when aborts are *exceedingly rare*. While the conventional wisdom is that aborts are "bad", such an opinion can be taken too far: protecting all transactions with a single mutex can prevent all aborts, but at the cost of preventing concurrent transactional execution. Similarly, if aborts can be avoided completely for some workload, perhaps the use of transactions is unnecessary.

Since latency is a crucial consideration for memory transactions, we argue that low abort rates can be symptomatic of two throughput problems: the current choice of TM implementation may have inherent bottlenecks that are effectively serializing all transactions, or the current choice of TM implementation may not be optimistic enough. In Section 2 we present some simple examples that demonstrate these scenarios.

The final problematic abort behavior is the one upon which this paper is focused: some transactions are destined to conflict with almost every other conceivable transaction. For convenience, we call these transactions "toxic", though again in truth this is a problem well studied in the database community. As a trivial example, suppose transaction T inserts into a RBTree, and the resulting tree rebalance propagates to the root: since every access to the tree must read the root, every concurrent transaction will conflict with T. Depending on how concurrency control is implemented, T may starve, or T and all conflicting transactions may livelock.

The most obvious solution is to fall back to a "Serial" mode, in which only the toxic transaction T may run. The key questions are how quickly this mode should be selected, and how to make it fast. TM implementations tend to consider this mode as a last resort, to be selected only when all else fails [26, 34, 35]. We argue that it is often best to prioritize the toxic transaction.

We refer to a policy that either lets all transactions run, or only one, as an "Hourglass" contention manager (CM). In this paper, we propose a low-overhead Hourglass CM, which maintains high performance by only loosely enforcing the "all or one" property. We measure the incidence of toxic transactions on the STAMP benchmark suite [8], and then evaluate Hourglass CM performance. Lastly, we extrapolate to best-effort HTM [1, 9], where we find that the Hourglass CM can prevent wasted work when conflict detection is too aggressive.

* This research was sponsored in part by the National Science Foundation Grant CNS-1016828

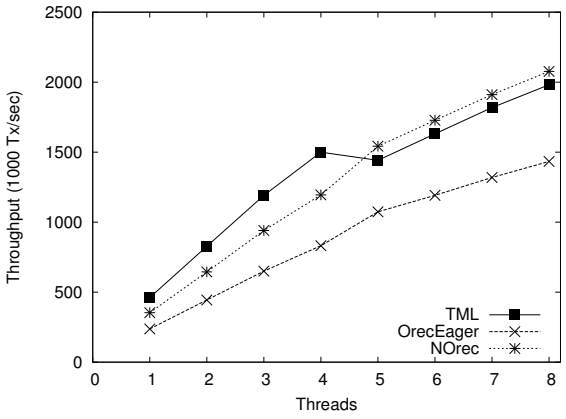


Figure 1. Red-Black Tree with 90% lookup ratio and 20-bit keys. Aggressive speculation (e.g., using TML) can offer superior performance, even though aborts increase substantially.

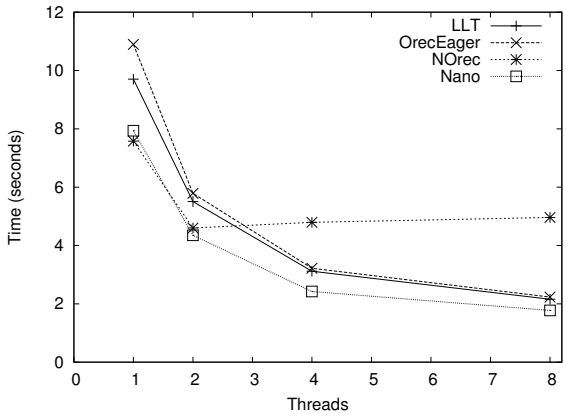


Figure 2. Performance of selected STM algorithms on the STAMP SSCA2 benchmark.

2. The Problem of Too Few Aborts

As mentioned briefly in Section 1, a low abort rate is not necessarily good. Of course aborts represent a scenario in which progress is not being made. However, from an implementation perspective, low aborts may suggest that an implementation is too conservative in its speculation, or simply ill-suited for a workload.

We first present a simple Red-Black tree experiment, using the TML algorithm [10]. In this test, the tree holds 20-bit keys, and 90% of operations are lookups. The test was performed on a machine with a 4-core (8-thread) Intel Xeon W3540 CPU. TML uses a single bit for conflict detection, such that any writer causes all readers to abort. This is an especially aggressive form of speculation: the assumption is that conflicts will be so rare that the overhead of recovery is not worth its cost.

As Figure 1 shows, TML offers excellent throughput out to 8 threads. The difference in aborts is staggering: at 8 threads, NOrec has 191 aborts for 6.5M commits; OrecEager has 778 aborts for 4.5M commits; and TML has 15M aborts with 6.2M commits. TML is not a general-purpose STM algorithm—it cannot scale at all on smaller trees with higher writer ratios—but still, by speculating more aggressively it can sometimes achieve higher performance, despite orders of magnitude more aborts.

Low aborts can also indicate a bottleneck in the STM implementation. To demonstrate this, we use the SSCA2 benchmark from STAMP [8]. All transactions are writers, and no transaction accesses more than 5 locations. We compare the performance of a variant of TL2 [12] called "LLT", a variant of LSA [16, 27] called "OrecEager", and NOrec [11].

We also consider the performance of an algorithm ("Nano") that has no internal bottlenecks. In Nano, opacity [19] is achieved via quadratic validation overhead (e.g., validation after every read [21]), a table of 1024 ownership records are used for detecting conflicts, and all writes are buffered. Clearly, Nano is only suitable for small transactions. However, as Figure 2 shows, Nano outperforms more mature and general-purpose algorithms.

In the SSCA2 experiment, there are 22M transactions. At 8 threads, LLT aborts under 1000 times, NOrec aborts under 200 times, and OrecEager aborts under 600 times. In contrast, Nano has 244K aborts. In experiments on a multi-chip system (not shown), aborts decreased for LLT, OrecEager, and NOrec, while execution time went up; meanwhile Nano continued to scale and incur higher abort rates. The cause is simple: for tiny transactions, there is metadata contention within the "mature" STM implementations. Thus low abort rates can be indicative of bottlenecks in the algorithm, and asymptotically worse STM implementations that abort more but avoid bottlenecks may be preferable.

To be clear, we are not advocating that aborts are an absolute good, but we do caution that low abort rates cannot be considered in isolation. Characteristics such as transaction size, number of chips, and read-only frequency can favor abort-prone algorithms. We leave more in-depth study of this phenomenon for future study.

3. The Hourglass Contention Manager

The remainder of this paper focuses on toxic transactions. We define a toxic transaction to be one that repeatedly aborts. Past work in both the database and TM communities has shown that when there are many toxic transactions, then either there is no concurrency in the application, or else there are simply too many transactions in-flight. Shutting off concurrency, or using schedulers and gatekeepers, address these two situations.

When toxic transactions are infrequent, the challenge is to resolve them effectively without incurring too much overhead. Recent contention management research for scalable workloads focuses on providing a low-overhead strategy to handle the common case. For example, the self-abort mechanisms of TL2 [12] and TinySTM [16], the livelock-free mechanism in RingSTM [33] and NOrec [11], and the Patient policy [32] all avoid the per-access bookkeeping of earlier contention management [28] while still providing good common-case performance.

3.1 Coarse-grained Contention Managers

In each of the STM algorithms mentioned above, a single, simple strategy for resolving conflicts is hard-coded into the implementation. This approach is also common to most HTM proposals. Accepting that the internals of conflict management cannot be changed without significant tinkering, we define a class of contention management (CM) policies that are invoked only at the boundaries of transactions. Such policies are compatible with hardware and software TM, since they are not invoked during transaction execution. We call such policies Coarse-grained Contention Managers (CCMs). The interface for a CCM is given in Figure 3.

The aim of CCM is to serve as a high level "congestion controller" built upon the abstract interface of transactions. A TM algorithm notifies the CCM when it begins, commits or aborts a transaction. In contrast with traditional contention managers, CCM embeds no event handlers within transactional read/write instrumentation. Thus, they are completely decoupled from TM algorithms.

```
interface CCM
    void onBegin(Tx * tx);
    void onCommit(Tx * tx);
    void onAbort(Tx * tx);
```

Figure 3. The Coarse-grained Contention Manager Interface

```
void BackoffCM::onBegin(Tx * tx) nop();

void BackoffCM::onCommit(Tx * tx) nop();

void BackoffCM::onAbort(Tx * tx)
    doBackoff();
```

Figure 4. The Backoff Contention Manager

CCM are unaware of the *causes* of conflicts between transactions, and instead, they are notified with the *result* of a conflict. For instance, if transactions A and B conflict, and the conflict causes B to abort, then B's CCM will reach the `onAbort` method. This method will only know that B aborted; it will not know the identity of A, nor the nature of the access that caused the conflict.

We briefly mention that the first contention management policy for the obstruction-free DSTM, exponential backoff, is a CCM. Figure 4 depicts the implementation: BackoffCM is notified when a transaction aborts and the `doBackoff()` operation is triggered. When a transaction begins or commits, the BackoffCM performs no operation. DSTM's "Aggressive" policy is identical to BackoffCM, except that the `onAbort` method is also a `nop`.

While CCMs lack the detailed information needed to make optimal conflict resolution decisions, they have the potential to be very simple and broadly applicable, since they treat TM implementations as a black-box: they can be combined to any TM as "plug-ins" without modifying the internal implementations. An important consequence is compatibility with transactional hardware, such as best-effort HTM. Both OneTM [6] and Hofmann's "Minimal HTM" [23] use similar mechanisms to handle overflow, with OneTM suggesting its use to support forward progress.

3.2 The Hourglass CM

Exponential backoff is an exceptionally simple CM policy. However, despite its widespread use, we feel that it is a little bit "dirty"—it works well in practice, but without leaving the implementor comfortable that it will work for a not-yet-encountered workload. This characteristic is particularly true since backoff may need to be re-tuned on a per-workload basis, and since backoff offers no guarantees of livelock or starvation prevention.

Backoff's role is simply to perturb a schedule, so conflicting transactions do not overlap on subsequent attempts. We now present a more overt mechanism for ensuring the same outcome. We call this CM "Hourglass"; its pseudocode appears in Figure 5.

In the Hourglass CM, a transaction counts its consecutive aborts. If this number exceeds some threshold, the transaction assumes it is toxic, and it attempts to gain exclusive access to a special token. When the token is held, only the holder can begin a transaction. Any new transaction (or retry of an aborted transaction) blocks at begin time until the toxic transaction commits and releases the token. The toxic transaction never blocks when it retries.

Unlike irrevocability [34, 35], the hourglass does not perform any global synchronization among threads before allowing the token-holder to begin, and it does not immediately shut off concurrency. Thus if transaction A aborts, gains the token, and restarts, it is possible that other transactions B_k are in-flight. No new transactions, apart from A, can begin, and any transaction among B_k that aborts will not be able to restart until A commits and releases its token. Thus only if A takes a long time will the total number of running transactions decrease to 1.

```
global    int hourglass = 0;
tx_local  int in_hourglass = 0;
tx_local  int consec_aborts = 0;

void HourglassCM::onBegin(Tx * tx)
    if (!tx->in_hourglass)
        while (hourglass);

void HourglassCM::onCommit(Tx * tx)
    if (tx->in_hourglass)
        tx->in_hourglass = 0;
        tx->consec_aborts = 0;
        hourglass = 0;

void HourglassCM::onAbort(Tx * tx)
    tx->consec_aborts++;
    if (tx->in_hourglass) return;
    if (tx->consec_aborts > THRESHOLD)
        if (CAS(hourglass, 0, 1)
            tx->in_hourglass = 1;
```

Figure 5. The Hourglass Contention Manager

Note, too, that the token holder may still abort. Aborts are a property of the underlying TM implementation, which CCM does not change, and the toxic transaction does not initially run in isolation. Under the assumption that every transaction commits *or* aborts in a finite time, Hourglass CM can guarantee progress and fairness: once a transaction A acquires the token, only in-flight transactions can complete before A, after which point A is guaranteed to run in isolation.

In comparison to Backoff, the Hourglass is slightly more expensive since there is a load and a branch during transaction begin. Compared to schedulers and gatekeepers, Hourglass should be much cheaper, since there are no writes in the common case. However, Hourglass is less precise than schedulers and gatekeepers.

One significant benefit of the Hourglass, relative to Backoff, is progress. Thus far, we have defined a toxic transaction A to be one that aborts repeatedly; we have said nothing about the impact of that transaction on others. One possibility is that A's repeated attempts cause other transactions to fail, due to conflicts. However, it is also possible that A's failure means that concurrent transactions will run slower (for example, suppose that A is resizing a hash table; until the table resizes, all hash operations will collide on a small number of buckets, each of which will have a long chain). This guarantee of progress, with low common-case overhead, makes the Hourglass an appealing alternative to Backoff.

3.3 Hourglass Variants

When toxic transactions are rare, it is unlikely that any transaction will fail to acquire the hourglass token. However, if there are many transactions simultaneously identified as toxic, only one can acquire the hourglass while the others wait at begin time. Once the token holder commits and releases the hourglass, the others will resume, probably abort, and then compete to acquire the hourglass. For high levels of toxic transactions, it may be desirable to block toxic transactions until they acquire the hourglass token. This "Strong" Hourglass variant is depicted in Figure 6. Such an approach is clearly less opportunistic than the default Hourglass. We leave its evaluation as future work.

```
void StrongHourglassCM::onAbort(Tx * tx)
    tx->consec_aborts++;
    if (tx->in_hourglass) return;
    if (tx->consec_aborts > THRSHOLD)
        while (!CAS(hourglass, 0, 1)
            tx->in_hourglass = 1;
```

Figure 6. The Strong Hourglass Contention Manager

```
global    int hourglass = 0;
tx_local  int in_hourglass = 0;
tx_local  int consec_aborts = 0;
tx_local  int backoff = 4;

void HourglassCM::onBegin(Tx * tx)
    if (!tx->in_hourglass)
        for (tmp in 0 .. 2 << backoff)
            if (!hourglass) return;

void HourglassCM::onCommit(Tx * tx)
    if (tx->in_hourglass)
        tx->in_hourglass = 0;
        tx->consec_aborts = 0;
        backoff = 4;
        hourglass = 0;

void HourglassCM::onAbort(Tx * tx)
    tx->consec_aborts++;
    if (tx->in_hourglass) backoff++; return;
    if (tx->consec_aborts > THRESHOLD)
        if (CAS(hourglass, 0, 1)
            tx->in_hourglass = 1;
```

Figure 7. The Nonblocking Hourglass Contention Manager

The other drawback of the Hourglass is that it is blocking: once a transaction acquires the token, no other transactions may start. Our Nonblocking Hourglass (Figure 7) replaces begin-time blocking with begin-time exponential backoff. By allowing the toxic transaction to specify the backoff duration, we can adapt to long-running toxic transactions, while also bounding the time that any transaction will block while waiting for a toxic transaction to commit. Note that the Nonblocking Hourglass may require some per-workload tuning of the initial backoff value. We also expect it to have higher overhead than the blocking Hourglass.

4. Evaluation

To assess the impact of Hourglass CM, we applied it to several popular STM algorithms running STAMP benchmarks on the x86 architecture. We present results for 13 STM algorithms: ByteEager and ByteLazy (TLRW [13] variants), CToken (an unpublished Orec-based algorithm that keeps overhead low via a commit token), InvalSTM [17], LLT (a TL2 [12] variant), NOrec, OrecEager, OrecLazy, OrecFair [32], OrecELA (OrecLazy with ELA semantics), RingSW, TLI (an unpublished, optimized InvalSTM variant), and TML. All STMs were implemented in the common adaptive RSTM framework [31].

4.1 How Toxic is STAMP?

Our first finding is that transactions in STAMP are rarely toxic. Figure 8 shows the percentage of transactions in selected STAMP workloads that commit on their first attempt, within 4 attempts, and in more than 4 attempts. Note that the y axis is not zero-based for all charts. Also, note that these experiments did not use any contention management; conflicts were detected and resolved based on the implicit policy of the corresponding STM algorithm. Except for OrecFair, aborted transactions restarted immediately.

KMeans (low contention), which matches the high-contention case, is not shown. Similarly, due to space constraints we leave out charts of Genome, SSCA2, and Vacation. For these tests, the only interesting result was that unless a TM algorithm uses too coarse of a granularity of conflict detection, repeated aborts are extremely rare. Specifically, Inval [17], TLI (an optimized InvalSTM), and RingSW [33] had higher aborts due to their use of 1024-bit signatures; TML aborts were even higher. All other algorithms had less than 1% of transaction attempts abort.

We see these charts as a complement to the abort analysis performed by Blake et al. [5]. In particular, we add an understanding of how often a manifested conflict repeats. While the overall number of toxic transactions is low, their behavior is concerning. Without any contention management, high rates of repeated aborts occur. The following table presents brief information about each workload. Perhaps the last line is most telling: on SSCA2, the LLT algorithm (a variant of TL2) had only one transaction abort more than 16 times, and it aborted 221 times.

	Algorithm	>16 aborts	Most aborts
Bayes	ByteEager	15	57K
Genome	OrecEager	17	225
Intruder	LLT	4334	2409
KMeans	ByteLazy	169K	665
SSCA2	LLT	1	221

For the OrecEager algorithm, we computed the average number of aborts for transactions that aborted more than 4 times. We ultimately concluded that a threshold of 2 was sufficient for identifying toxic transactions.

Workload	Average Aborts
Bayes	8
Genome	6.58
Intruder	15.33
KMeans (high)	13.66
KMeans (low)	12.65
Labyrinth	8.5
SSCA2	5.2
Vacation (high)	11.01
Vacation (low)	10.35

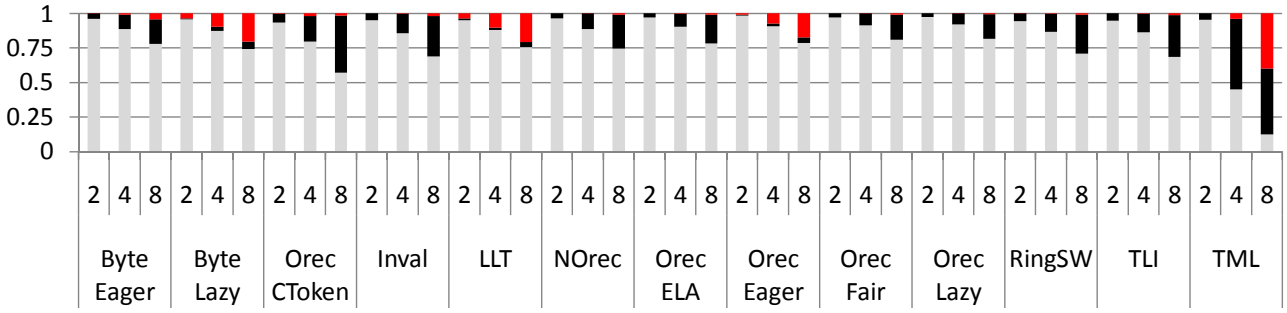
4.2 Comparing Hourglass and Backoff CM

The NOrec, OrecEager, and OrecLazy algorithms are among the best all-around performers on STAMP. Given the infrequency of toxic transactions, we explore the cost that adding any CM to these algorithms would introduce. Figure 9 presents the speedup of these algorithms when using backoff or Hourglass CM, normalized to performance without any contention management.

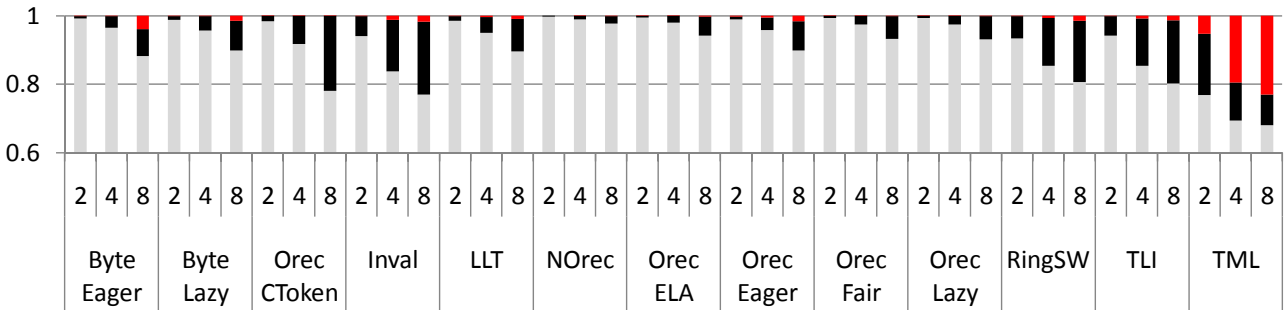
With the exception of Bayes, the speedups are negligible. Since the contention managers are rarely invoked, this result is expected. Thus the addition of CM support to a good STM algorithm does not appear to have a cost, but does offer the hope that throughput will be improved for high-conflict workloads. When we omit the Bayes workload (which is known to have unpredictable behavior even in the absence of CM variation), the geometric mean speedup of either mechanism is stable and close to 1.

4.3 Comparison to Polka

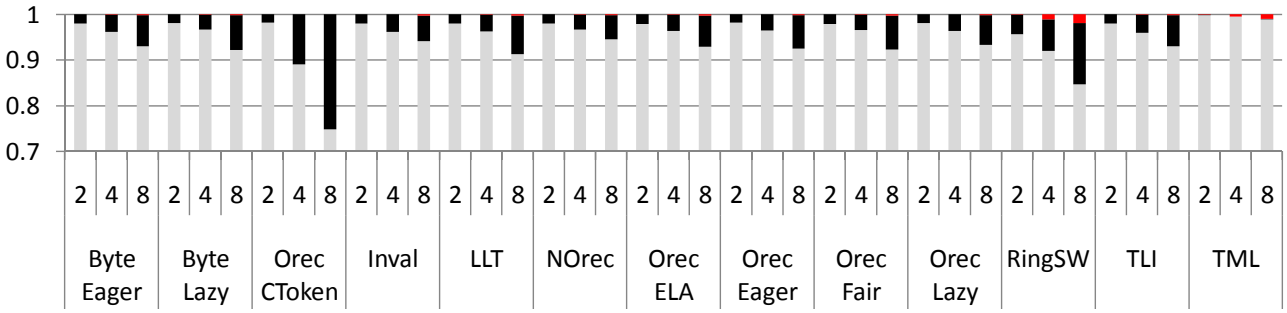
In experiments that we do not present, we found that CCM significantly outperforms the Polka CM [28]. This is not a surprise:



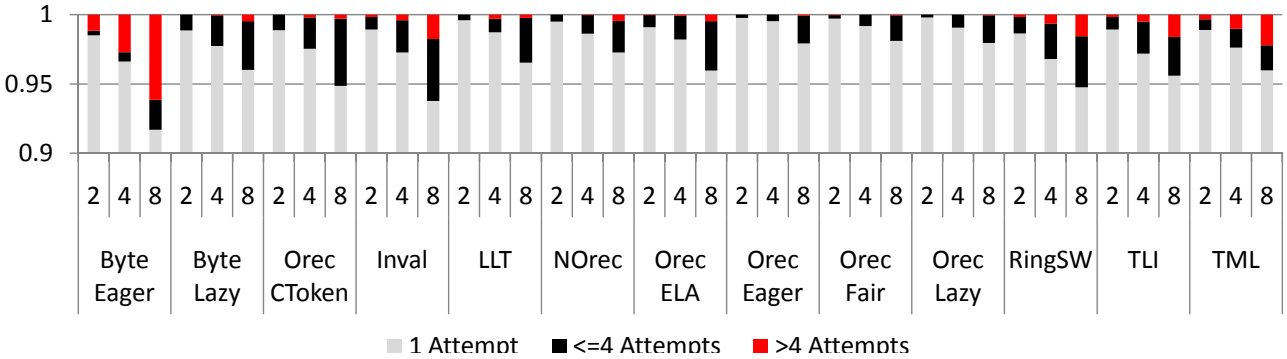
(a) KMeans (high contention)



(b) Intruder



(c) Labyrinth



(d) Bayes

Figure 8. Percentage of transactions that abort frequently in selected STAMP workloads.

	NOrec Backoff	NOrec Hourglass	OrecEager Backoff	OrecEager Hourglass	OrecLazy Backoff	OrecLazy Hourglass
Geometric Mean	0.995	1.002	0.989	0.957	1.001	0.989
Min Speedup	0.851	0.812	0.730	0.631	0.910	0.789
Max Speedup	1.176	1.377	1.177	1.034	1.492	1.243
Geometric Mean (Excluding Bayes)	0.989	0.998	0.998	0.968	0.988	0.986

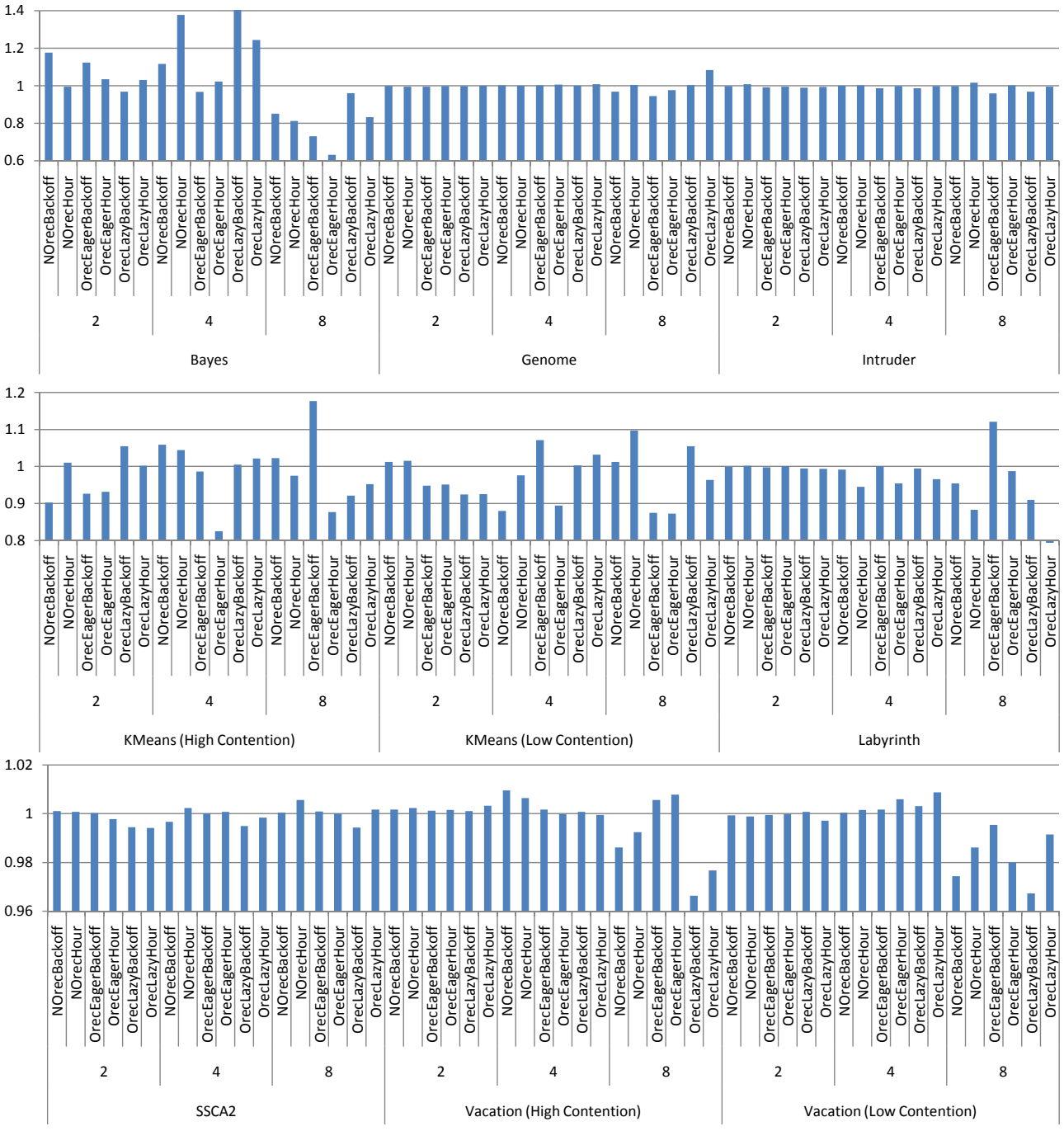


Figure 9. Speedup vs. configurations with no contention management for NOrec, OrecEager, and OrecLazy algorithms on STAMP.

Polka is designed for nonblocking TM; thus it requires the implementation to support remote aborts, share per-thread metadata, and perform per-access CM instrumentation. Blocking TM systems can avoid these costs [12, 16, 32]. Furthermore, in blocking TM the benefit of making fine-grained CM decisions is offset by the requirement that after transaction A aborts in-flight transaction B, it cannot make progress; it must first wait for B to release its locks.

4.4 Summary of Results

We found that in STAMP, toxic transactions are rare, but still challenging, as they can abort 1000 times or more. This suggests that STAMP may not be a good benchmark for evaluating CM: Apart from these toxic transactions, there is little need for any CM for most mature STM implementations. On the x86, where latency and throughput appear to depend heavily on the complexity of the CM, simple contention managers are likely to offer the best performance in the common case.

We also observed that the frequency of toxic transactions, and their cost in consecutive aborts, are dependent on the STM but not the workload. For example, NOrec had zero toxic transactions in many workloads, and its average number of consecutive aborts was much lower than more eager STM algorithms. These observations suggest that an Hourglass CM should be tuned to the STM algorithm, rather than a workload, which is a reasonably simple task.

Lastly, we note that the inherently weak nature of the guarantees made by Hourglass CM does not result in significantly lower abort rates, but the nature of repeated aborts changes. The behavior is STM-specific. Once a transaction flags itself as "toxic", no other transactions can start. In NOrec, the abort was by definition due to another transaction committing. Thus with T in-flight threads, at the point where transaction A becomes toxic it can abort at most T times (e.g., when it conflicts with all of those in-flight transactions, and they all commit before A). In OrecEager, A could abort repeatedly when trying to access a location locked by in-flight transaction B. When A becomes toxic, no new transactions start, but until B commits, A will continue to encounter B's lock and abort. In our measurements, we found that consecutive aborts for NOrec+Hourglass never exceeded 14, whereas consecutive aborts for OrecEager+Hourglass could still grow very large, though without having a noticeable impact on performance.

5. Extrapolating to Best-Effort HTM

Best-effort HTM is characterized by its desire to avoid deep changes to the CPU in order to support TM. Paramount is an aim to leave the cache coherence protocol intact, which results in a conflict detection strategy known as "requester wins" or "attacker wins". This policy is prone to livelock even for small transactions at low thread levels.

While a full system simulation is outside the scope of this paper, we sought to evaluate the Hourglass CM for Best-Effort HTM through a functional simulation. We extended our TLRW implementation to support remote abort. Since TLRW uses visible reads, the resulting system detects every conflict as soon as it manifests, and thus closely resembles attacker wins conflict detection. Of course, latencies are incomparable between our attacker-wins STM and true best-effort HTM.

To determine the impact of Hourglass CM on an attacker-wins STM, we compare 5 systems: three variants of our attacker wins STM with (a) no contention management, (b) exponential backoff on abort, or (c) Hourglass CM; our attacker-wins STM extended with timestamps, as proposed by Bobba et al. [7]; and a responder-wins implementation, like LogTM [25], achieved by using TLRW as-is. We tested these systems on a 4-core/8-thread Intel Xeon W3540 CPU, using a simple Red-Black tree. The result appears in Figure 10.

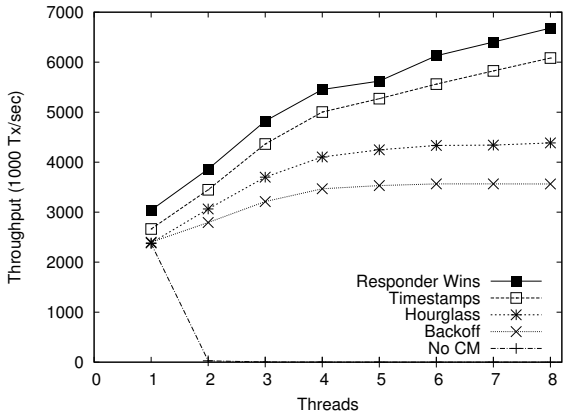


Figure 10. Evaluation of Hourglass CM on a Red-Black tree with 33% lookups and 8-bit keys.

While a crude experiment (the cost of supporting remote aborts causes a gap between systems at 1 thread), the results are nonetheless positive. Changing the coherence protocol, as Bobba proposes, is the most effective way to improve throughput. However, simply using Hourglass CM with a threshold of 2 consecutive aborts suffices to prevent the livelock experienced by the "No CM" system, and outperforms a tuned exponential backoff CM. Considering that an Hourglass CM requires no workload-specific tuning, whereas backoff parameters can be sensitive to application characteristics, we encourage further exploration of the Hourglass CM by HTM designers. We expect that the desire to keep implementation costs low will favor moving policy to software, and the Hourglass CM is among the simplest general-purpose CM policies for HTM.

6. Conclusions

In this paper we measured the frequency of aborts in the STAMP benchmark suite for a number of STM algorithms. We found that in general, high numbers of consecutive aborts (a condition we call "toxic transactions") are rare, even though an individual toxic transaction may abort thousands of times. This finding suggests that STAMP may not be a good benchmark for evaluating contention managers, and that consecutive aborts may not be a good indicator of pathology.

We also proposed a simple contention management strategy called "Hourglass." The key idea behind the Hourglass is that a toxic transaction should be prioritized, by forbidding transactions from starting (or restarting) until the toxic transaction commits. While similar in effect to exponential backoff, Hourglass CM is arguably simpler and more attractive: it does not require per-workload tuning, it provides fairness, and it guarantees progress.

In our evaluation on the x86, we showed that Hourglass CM delivers its guarantees while offering comparable performance to existing CM policies. We also showed that for the extremely aggressive conflict detection policies that are common in HTM, Hourglass CM can improve throughput without requiring changes to silicon.

Both backoff and Hourglass CM are instances of a simple, boundary-only variant of contention management which we refer to as coarse-grained CM. Such policies have the advantage of leaving per-access TM protocols intact and pushing conflict management off the critical path (e.g., to the abort handler). We hope that our results and suggestions will encourage the exploration of what hardware and software support are needed to produce new, novel coarse-grained CM policies that can further enhance performance and provide even stronger guarantees.

References

[1] Advanced Micro Devices, Inc. Advanced Synchronization Facility: Proposed Architectural Specificaiton. Technical Report Publication #45432, rev. 2.1, Advanced Micro Devices, Inc, Mar. 2009. Available as developer.amd.com/assets/45432-ASF_Spec_2.1.pdf.

[2] M. Ansari, M. Luján, C. Kotselidis, K. Jarvis, C. C. Kirkham, and I. Watson. Steal-on-Abort: Improving Transactional Memory Performance through Dynamic Transaction Reordering. In *Proceedings of the Fourth International Conference on High Performance Embedded Architectures and Compilers*, Paphos, Cyprus, Jan. 2009.

[3] H. Attiya, L. Epstein, H. Shachnai, and T. Tamir. Transactional contention management as a non-clairvoyant scheduling problem. In *Proceedings of the 25th ACM Symposium on Principles of Distributed Computing*, Denver, CO, Aug. 2006.

[4] H. Attiya and A. Milani. Transactional Scheduling for Read-Dominated Workloads. In *Proceedings of the 13th International Conference on Principles of Distributed Systems*, Dec. 2009.

[5] G. Blake, R. Dreslinski, and T. Mudge. Proactive Transaction Scheduling for Contention Management. In *Proceedings of the 42nd IEEE/ACM International Symposium on Microarchitecture*, Dec. 2009. ISBN 978-1-60558-798-1.

[6] C. Blundell, J. Devietti, E. C. Lewis, and M. Martin. Making the Fast Case Common and the Uncommon Case Simple in Unbounded Transactional Memory. In *Proceedings of the 34th International Symposium on Computer Architecture*, San Diego, CA, June 2007.

[7] J. Bobba, K. E. Moore, H. Volos, L. Yen, M. D. Hill, M. M. Swift, and D. A. Wood. Performance Pathologies in Hardware Transactional Memory. In *Proceedings of the 34th International Symposium on Computer Architecture*, San Diego, CA, June 2007.

[8] 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*, Seattle, WA, Sept. 2008.

[9] S. Chaudhry, R. Cypher, M. Ekman, M. Karlsson, A. Landin, and S. Yip. Rock: A High-Performance Sparc CMT Processor. *IEEE Micro*, 29(2):6–16, March–April 2009.

[10] L. Dalessandro, D. Dice, M. Scott, N. Shavit, and M. Spear. Transactional Mutex Locks. In *Proceedings of the Euro-Par 2010 Conference*, Ischia-Naples, Italy, Aug–Sep 2010.

[11] L. Dalessandro, M. F. Spear, and M. L. Scott. NOrec: Streamlining STM by Abolishing Ownership Records. In *Proceedings of the 15th ACM Symposium on Principles and Practice of Parallel Programming*, Bangalore, India, Jan. 2010.

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

[13] D. Dice and N. Shavit. TLRW: Return of the Read-Write Lock. In *Proceedings of the 22nd ACM Symposium on Parallelism in Algorithms and Architectures*, Santorini, Greece, June 2010.

[14] S. Dolev, D. Hendler, and A. Suissa. CAR-STM: Scheduling-Based Collision Avoidance and Resolution for Software Transactional Memory. In *Proceedings of the 27th ACM Symposium on Principles of Distributed Computing*, Toronto, ON, Canada, Aug. 2008.

[15] A. Dragojević, R. Guerraoui, A. Singh, and V. Singh. Preventing Versus Curing: Avoiding Conflicts in Transactional Memories. In *Proceedings of the 28th ACM Symposium on Principles of Distributed Computing*, Calgary, AB, Canada, Aug. 2008.

[16] P. Felber, C. Fetzer, and T. Riegel. Dynamic Performance Tuning of Word-Based Software Transactional Memory. In *Proceedings of the 13th ACM Symposium on Principles and Practice of Parallel Programming*, Salt Lake City, UT, Feb. 2008.

[17] J. Gottschlich, M. Vachharajani, and J. Siek. An Efficient Software Transactional Memory Using Commit-Time Invalidation. In *Proceedings of the 2010 International Symposium on Code Generation and Optimization*, Toronto, ON, Canada, Apr. 2010.

[18] J. Gray and A. Reuter. *Transaction Processing: Concepts and Techniques*. Morgan Kaufmann, 1993.

[19] R. Guerraoui and M. Kapalka. On the Correctness of Transactional Memory. In *Proceedings of the 13th ACM Symposium on Principles and Practice of Parallel Programming*, Salt Lake City, UT, Feb. 2008.

[20] T. Heber, D. Hendler, and A. Suissa. On the Impact of Serializing Contention Management on STM Performance. In *Proceedings of the 13th International Conference on Principles of Distributed Systems*, Dec. 2009.

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

[22] M. P. Herlihy and J. E. B. Moss. Transactional Memory: Architectural Support for Lock-Free Data Structures. In *Proceedings of the 20th International Symposium on Computer Architecture*, San Diego, CA, May 1993.

[23] O. Hofmann, C. Rossbach, and E. Witchel. Maximum Benefit from a Minimal HTM. In *Proceedings of the 14th International Conference on Architectural Support for Programming Languages and Operating Systems*, Long Beach, CA, Mar. 2009.

[24] W. Maldonado, P. Marlier, P. Felber, A. Suissa, D. Hendler, A. Fedorova, J. Lawall, and G. Muller. Scheduling Support for Transactional Memory Contention Management. In *Proceedings of the 15th ACM Symposium on Principles and Practice of Parallel Programming*, Bangalore, India, Jan. 2010.

[25] K. E. Moore, J. Bobba, M. J. Moravan, M. D. Hill, and D. A. Wood. LogTM: Log-based Transactional Memory. In *Proceedings of the 12th International Symposium on High-Performance Computer Architecture*, Feb. 2006.

[26] Y. Ni, A. Welc, A.-R. Adl-Tabatabai, M. Bach, S. Berkowits, J. Cownie, R. Geva, S. Kozhukow, R. Narayanaswamy, J. Olivier, S. Preis, B. Saha, A. Tal, and X. Tian. Design and Implementation of Transactional Constructs for C/C++. In *Proceedings of the 23rd ACM Conference on Object Oriented Programming, Systems, Languages, and Applications*, Nashville, TN, USA, Oct. 2008.

[27] T. Riegel, P. Felber, and C. Fetzer. A Lazy Snapshot Algorithm with Eager Validation. In *Proceedings of the 20th International Symposium on Distributed Computing*, Stockholm, Sweden, Sept. 2006.

[28] W. N. Scherer III and M. L. Scott. Advanced Contention Management for Dynamic Software Transactional Memory. In *Proceedings of the 24th ACM Symposium on Principles of Distributed Computing*, Las Vegas, NV, July 2005.

[29] G. Sharma, B. Estrade, and C. Busch. Window-Based Greedy Contention Management for Transactional Memory. In *Proceedings of the 24th International Symposium on Distributed Computing*, Cambridge, MA, Sept. 2010.

[30] A. Shriraman and S. Dwarkadas. Refereeing Conflicts in Hardware Transactional Memory. In *Proceedings of the 23rd ACM International Conference on Supercomputing*, Yorktown Heights, NY, June 2009.

[31] M. Spear. Lightweight, Robust Adaptivity for Software Transactional Memory. In *Proceedings of the 22nd ACM Symposium on Parallelism in Algorithms and Architectures*, Santorini, Greece, June 2010.

[32] M. F. Spear, L. Dalessandro, V. J. Marathe, and M. L. Scott. A Comprehensive Strategy for Contention Management in Software Transactional Memory. In *Proceedings of the 14th ACM Symposium on Principles and Practice of Parallel Programming*, Raleigh, NC, Feb. 2009.

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

[34] M. F. Spear, M. Silverman, L. Dalessandro, M. M. Michael, and M. L. Scott. Implementing and Exploiting Inevitability in Software Transactional Memory. In *Proceedings of the 37th International Conference on Parallel Processing*, Portland, OR, Sept. 2008.

[35] A. Welc, B. Saha, and A.-R. Adl-Tabatabai. Irrevocable Transactions and their Applications. In *Proceedings of the 20th ACM Symposium on*

Parallelism in Algorithms and Architectures, Munich, Germany, June 2008.

[36] R. Yoo and H.-H. Lee. Adaptive Transaction Scheduling for Transactional Memory Systems. In *Proceedings of the 20th ACM Symposium on Parallelism in Algorithms and Architectures*, Munich, Germany, June 2008.