Int J Parallel Prog (2016) 44:1359–1383
DOI 10.1007/s10766-016-0427-4

Purge-Rehab: Eager Software Transactional Memory with High Performance Under Contention

Abubakar Siddique[1] · **Mohammad Ansari**[2] · **Mikel Luján**[3]

Received: 27 November 2014 / Accepted: 5 April 2016 / Published online: 16 April 2016

Abstract Transactional memory is a programming model that attempts to make parallel programming easier. Transactional memory uses either eager (at encounter time) or lazy (at commit time) validation to check for conflicting accesses between concurrent transactions, and researchers have been divided over which approach is best. Despite this, there is little work in the literature evaluating their comparative performance. One exploration of this topic using microbenchmarks found that lazy outperforms eager and increases its advantage as contention rises. Furthermore, eager was prone to livelock when accesses were irregular, which resulted in starvation and low throughput. We revisit the comparative performance of eager and lazy under contention using a larger set of microbenchmarks, more complex benchmarks from STAMP, and against multiple state-of-the-art STMs: word-based RSTM, TinySTM and SwissTM. We validate earlier findings that eager suffers from livelock, especially when accesses are irregular. This inspired Purge-Rehab: a lightweight mechanism for controlling transaction restarts in eager validation to reduce livelock and thus improve throughput and reduce starvation. Compared to lazy validation, Purge-Rehab achieves higher performance in five benchmarks, similar in four, and is lower in only one, showing that eager valida-

✉ Mohammad Ansari
mmansari@uqu.edu.sa

Abubakar Siddique
absiddique@uqu.edu.sa

Mikel Luján
mikel.lujan@manchester.ac.uk

1 Science and Technology Unit, Umm Al Qura University, Makkah, Saudi Arabia

2 Department of Computer Science, Umm Al Qura University, Makkah, Saudi Arabia

3 School of Computer Science, University of Manchester, Manchester, UK

tion can achieve high performance under contention. Purge-Rehab is implemented in word-based RSTM, but is applicable to any eager STM.

Keywords Software transactional memory · Eager validation · Lazy validation · Conflict detection · Contention management · Livelock · Starvation

1 Introduction

The many-core revolution challenges programmers to exploit concurrency if they want higher performance. Traditional concurrent programming utilizes locks for synchronization and a programmer must explicitly acquire and release locks to safeguard accesses to shared variables. Explicit synchronization is known to be challenging and may lead to race conditions, livelock, starvation, or priority inversion, which are difficult to reproduce and thus difficult to fix. Transactional Memory (TM) attempts to ease this challenge by making it easier to develop robust concurrent programs. Sections of code that access shared variables are marked as *transactions*, and a runtime automatically checks for conflicts between concurrently executing transactions. If a conflict is found, one of the transactions is automatically aborted and restarted to resolve the conflict.

TMs need to validate executing transactions to check if a transaction has a conflict with any concurrently executing or recently committed transactions. Validation can be achieved eagerly (at encounter time) or lazily (at commit time). Researchers have traditionally been divided over which one gives better throughput. For example, some have contended that eager validation increases transaction throughput as contention rises because it aborts a transaction upon conflict, whereas lazy validation wastes work by continuing to execute a doomed transaction until its commit stage and only then aborts [5,7]. Others have argued the opposite; lazy exploits potential parallelism more effectively, and that eager's early aborts lead to livelock between conflicting transactions rather than higher throughput [21–23]. However, the performance of eager and lazy are seldom directly contrasted in the literature.

Spear et al. [23] performed a comparative analysis of eager and lazy validation in Software TM (STM). They found that a carefully designed lazy (i) outperforms eager under contention, (ii) has the propensity to avoid pathologies that decrease throughput, and (iii) can avoid starvation through priority scheduling of transactions. On the other hand, they found that eager (iv) is prone to livelock under contention, which reduces its throughput, (v) increases aborts, and (vi) benchmarks with irregular access patterns (pathology benchmarks) encounter severe livelock that results in starvation and near-zero throughput.

Eager validation monitors for transactional conflicts throughout each transaction's lifetime. This enables it to detect conflicts and abort transactions early, which should reduce wasted work. However, the constant monitoring results in a large period of time over which conflicts can be detected, which has several disadvantages when contention occurs. First, the large period of time increases the opportunity for conflicts to be detected in the first place, and thus aborts to occur. This may harm performance if conflicts are detected late in a transaction's lifetime, because it has already performed

a significant amount of work that will be discarded if it is aborted. Second, it increases the chance that an aborted transaction will restart and conflict with the same opponent again. Many contention managers [8,9,18,19] only abort the opponent, and since the restarted transaction is the one likely to re-detect the conflict, two transactions may enter a cycle of conflicts and aborts between each other. Third, it raises the chance for such conflict cycles amongst several transactions that access heavily contended data. Together, these issues may induce livelock when an application experiences contention.

This article presents a technique called Purge-Rehab to reduce livelock in eager by addressing some of the issues above. In Purge-Rehab, each thread has a queue (called a rehabQueue) for holding rehabilitation transactions. Livelock is reduced by placing aborted transactions in the rehabQueue of the conflict-winning transaction's thread instead of restarting them immediately. These transactions are restarted sequentially once the conflict-winning transaction commits, which eliminates the chance for the two transactions to repeatedly abort one another. Purge-Rehab manages starvation by processing any transactions in the rehabQueue first. The thread executing the aborted transaction fetches a new transaction from the application, if available.

We evaluate the performance of eager and lazy using several modern word-based STMs: RSTM [17], SwissTM [6,24] and TinySTM [25] across five microbenchmarks and five STAMP benchmarks. Purge-Rehab is implemented on top of eager RSTM to enable a direct comparison with the state-of-the-art lazy RSTM of Spear et al. [23], as well as the other STMs. The results validate that eager does struggle when there is heavy contention, but that Purge-Rehab (i) outperforms lazy in five cases by $1.5\text{x} - 3\text{x}$ and is competitive in four cases, (ii) averts livelock in benchmarks with irregular access patterns (pathology benchmarks), (iii) decreases wasted work, and (iv) manages starvation better than lazy. Only in one case does lazy outperform the eager STMs. This article shows that it is possible to design eager to be competitive with lazy, but also that there is no outright winner.

Section 2 introduces TM and illustrates how it is used to parallelize applications. Sect. 3 details Purge-Rehab's implementation and design decisions in RSTM. Section 4 presents the experimental evaluation, including empirical and statistical analysis, throughput, wasted work and starvation. Sect. 5 discusses related work, and Sect. 6 concludes the article.

2 Parallelizing with Transactional Memory

TM is a concurrent programming model that eases the programmer burden of exploiting parallelism. The rise of multi-cores has increased the need for software to be parallelized, which has led to intensified research in TM. Based on database transaction theory, TM removes the difficulties in ensuring safe access to shared data by automatically detecting and resolving concurrent accesses to them.

Herlihy and Moss proposed the idea of Transactional Memory (TM) as a hardware mechanism [11]. However, its exploitation required hardware-level changes, and these are typically expensive and time consuming to implement. Later, Shavit and Touitou proposed Software Transactional Memory (STM) [20], and TM became available as

a library or language-level construct. This section briefly introduces TM, presents a simple example that illustrates how TM may be used, highlights the potential benefits it offers over classic lock-based synchronization, and briefly covers implementation details relevant to the contributions of this article.

2.1 Transactional Memory

Safe access to shared data in parallel programs has traditionally required lock-based synchronization, which is challenging to employ. Lock-based applications may suffer from race conditions, livelock, deadlock, and priority inversion. These are programmer errors that may be difficult to reproduce and resolve. In addition, achieving balance between high scalability and locking overhead through appropriate lock granularity often requires an experienced hand.

Transactional memory (TM) is a programming model that aims to reduce the challenges associated with constructing parallel programs using lock-based synchronization. Lomet [14] presented the concept of atomic actions in 1977, which considered using database transactions for concurrency-safe access to shared data. Database transactions have ACID properties: atomicity, consistency, isolation, and durability. The first three are desirable for thread-safe access to shared data.

TM introduces language constructs that enable a programmer to specify a section of code as a *transaction*. The programmer simply marks as transactions those sections of code that access shared data, and a runtime layer manages the complexity of achieving mutual exclusion between concurrent transactions *automatically*. Thus, TM guarantees atomicity, consistency, and isolation among transactions, but abstracts away the implementation of how these guarantees are achieved, which reduces the effort involved in writing robust parallel programs.

The runtime layer is responsible for *conflict detection*, and monitors the read and write accesses of transactions. If a read/write or write/write access to a shared variable is performed by two concurrent transactions, the runtime layer is responsible for *conflict resolution*, which typically involves *aborting* and *restarting* one of the transactions. Deciding which transaction to abort is determined by a *contention manager*. Changes made to shared variables by aborted transactions are reset to the original values they had before the transaction had started. Transactions that complete their execution *commit* and make their changes to shared variables permanent.

2.2 TM Versus Locks

As a small example of the simplicity offered by TM, consider a shared counter `x` that should be incremented in a concurrency-safe way. Lock-based pseudo code is shown in Listing 1, and TM-based pseudo code is shown in Listing 2.

Listing 1 Safe lock-based update of shared variable 'x'.

```
while (!have_lock(xlock) && numAttempts < maxAttempts )
    {
```

```
        try_getlock(xlock, 10); // Timeout to avoid deadlock

        if (have_lock(xlock)) {
            x = x + 1;
            release_lock(xlock);
            break;
        } else {
            numAttempts++;
        }
    }

    if (numAttempts == maxAttempts) {
        // Unable to acquire the lock; something went wrong.
        // Do any detection, recovery and/or graceful
            failure.
    }
```

Listing 2 Safe TM-based update of shared variable 'x'.

```
atomic {
    x = x + 1;
}
```

The lock-based code burdens the programmer to indicate the specific locks needed (`xlock`), and specify a timeout to prevent deadlock. The programmer must then check if the lock is acquired, and only then perform the update, and remember to release the specific lock. Additionally, failing to acquire the lock may be indicative of a failure elsewhere in the application due to a locking related programming error. For example, another thread may have acquired several locks, including `xlock`, but then deadlocked. Depending on the applications specification, the programmer may also need to write additional code to manage such a failure, which is not shown in Listing 1. The challenge of keeping the code correct rises as the code becomes complex, and accesses more shared data. For example, the programmer must also ensure locks are acquired and released in the same order by all such critical sections of code.

In TM, the update is simply wrapped in a transaction, shown here using the keyword `atomic`. Even this trivial example quickly reveals the simplicity afforded by TM. The programmer does not have to indicate any specific locks, nor indicate timeouts to prevent deadlock, nor remember to release the lock, nor write additional code to handle locking failures. TM provides such safety implicitly.

2.3 Eager and Lazy Validation

Validating the reads and writes of a transaction for the purpose of conflict detection may be done *eagerly* or *lazily*. Eager validation checks accesses as they are encountered, whereas lazy checks all accesses at the end. If no conflicts exist, then in practice eager and lazy validation achieve similar performance, as shown in Fig. 4 by the lazy and eager variants of word-based RSTM. If a conflict exists between two transactions, then eager validation will detect it early, and abort one of the transactions before it wastes processor time in executing the remainder of the transaction. However, the victim transaction may have been aborted prematurely, because it may be the case

that the opponent itself later aborts, which could have opened the way for the victim to continue and commit. Lazy validation could waste processor time by executing a transaction to the end before it detects a conflict, but may also improve performance by avoiding premature aborts.

There is little work in the literature that attempts to determine whether eager or lazy achieves higher performance when there is contention between transactions. The work of Spear et al. [23] presented one of the first comprehensive evaluations, and determined that while classic lazy STMs did perform poorly compared to eager STMs, a tuned lazy STM could achieve significantly higher performance than an eager STM. They motivated the work presented in this article to develop a tuned eager STM that outperforms lazy STM.

3 Purge-Rehab

This section explains the design and implementation of Purge-Rehab in eager RSTM, and how it attempts to reduce livelock and starvation. RSTM is used because it is the only modern word-based, high performance STM that implements both eager and lazy validation. Implementing Purge-Rehab on top of eager RSTM enables a direct comparison with the tuned lazy RSTM of Spear et al. [23]. In Sect. 4.2 we show that the performance of eager and lazy RSTM is very similar when there is no contention.

3.1 Reducing Livelock

Listing 3 Pseudocode of the default RSTM algorithm.

```
//Main transaction execution loop run by each thread
while (numTx < maxTx && execTime < timeLimit) {

    tx = getTransaction();

    while (!tx.isCommitted()) {

        tx.execute();

        if (tx.isAborted()) {
            tx.doAbortProcessing();
        } else {
            numTx++;
            tx.doCommitProcessing();
        }
    }
}

//Code executed upon conflict detection
loserTx = contentionManager.resolveConflict(myTx,
    otherTx);
loserTx.setAborted(true);
```

Figure 1 and Listing 3 illustrate how transactions are executed by threads in eager and lazy RSTM. By default, each thread executes transactions from the benchmark.

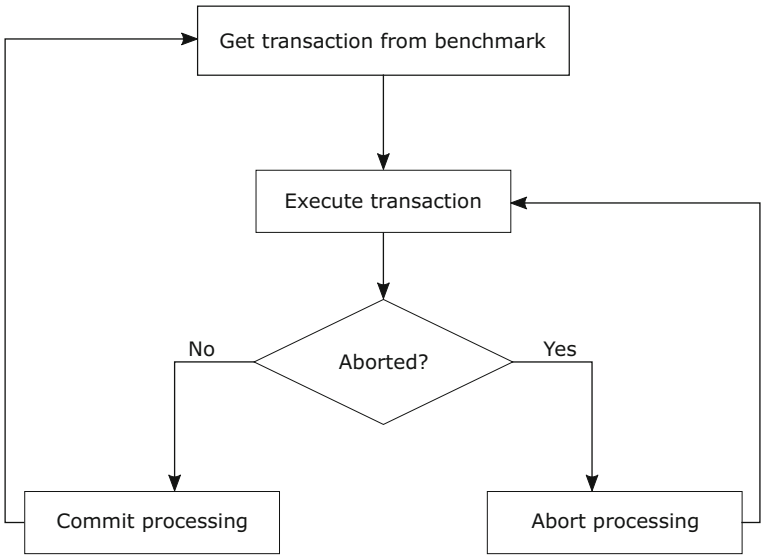

Fig. 1 Simplified flow of the default RSTM algorithm

A transaction that detects a conflict with another may abort itself or its opponent by updating a status flag from *active* to *aborted*. The thread executing the aborted transaction observes the status change (not necessarily immediately), performs any abort processing, and restarts the transaction immediately. If a transaction completes without any conflicts, its thread performs any required commit processing, which changes its status to *committed*. The thread then fetches a new transaction.

Listing 4 Pseudocode of Purge-Rehab in RSTM.

```
//Main transaction execution loop run by each thread
while (numTx < maxTx && execTime < timeLimit) {

    tx = getTransaction();
    tx.rehabID = -1;

    while (!tx.isCommitted()) {

        tx.execute();

        if (tx.isAborted()) {
            tx.doAbortProcessing();
            if (tx.rehabID != -1) {
                rehabQueue[rehabID].push(tx)

                while (tx = rehabQueue[myThreadID].pop()
                    ) {
                    rehabQueue[rehabID].push(tx)
                }

                break;
            }
```

```
            } else {
                numTx++;
                tx.doCommitProcessing();
                if (!rehabQueue.isEmpty()) {
                    tx = rehabQueue.pop()
                    continue;
                }
            }
        }
    }

    //Code executed upon conflict detection
    loserTx = contentionManager.resolveConflict(myTx,
        otherTx);
    if (loserTx != myTx) {
        loserTx.rehabID = myThreadID //ID of winner tx
            thread
    }
    loserTx.setAborted(true);
```

In eager, one of the reasons for livelock is that encounter time conflict detection entails a large conflict detection window. Consequently, an aborted and restarted transaction may re-conflict with the same opponent if the latter is still active. As contention increases, transactions may form clusters that repeatedly restart one another over highly contended shared data. Additionally, several contention management policies use back off upon detecting a conflict to give the opponent transaction a grace period to commit before aborting it. However, this keeps the detecting transaction active for longer, and increases the possibility for livelock further.

In lazy, the conflict detection window is small because it is only done at the end. Even if a transaction restarts due to a conflict, it is unlikely to conflict with the same opponent again. The opponent was either near the end of its own execution and aborted the victim transaction shortly before committing, or had recently committed and made conflicting changes to shared data that require the victim to restart itself. This implies that lazy does not suffer from livelock in the same way that eager does.

Figure 2 and Listing 4 show how Purge-Rehab modifies the execution flow of eager validation to reduce livelock due to repeat conflicts. In Purge-Rehab, each thread has a `rehabQueue` for storing conflict-losing (rehabilitation) transactions, and this is implemented by an Intel Thread Building Blocks [13] concurrent queue. The concurrent queues are stored in a global dynamic array, and each thread's `rehabQueue` is indexed by its unique thread ID. Additionally, each transaction's metadata is expanded with a field called `rehabID` to store the index of a `rehabQueue`, and its default value is -1.

Upon conflict the winning thread sets its own thread ID in the `rehabID` field of the losing transaction, and then aborts it. Once the losing transaction's thread notices the status change and completes any abort processing, it uses the `rehabID` to insert the aborted transaction in the `rehabQueue` of the winning thread, and acquires a new transaction from the benchmark. This prevents two transactions that have already conflicted with each other from being executed concurrently, and reduces livelock. Eager RSTM uses invisible reads so a transaction may abort itself at the end of its execution when validating its reads. In this case, if the opponent transaction has already

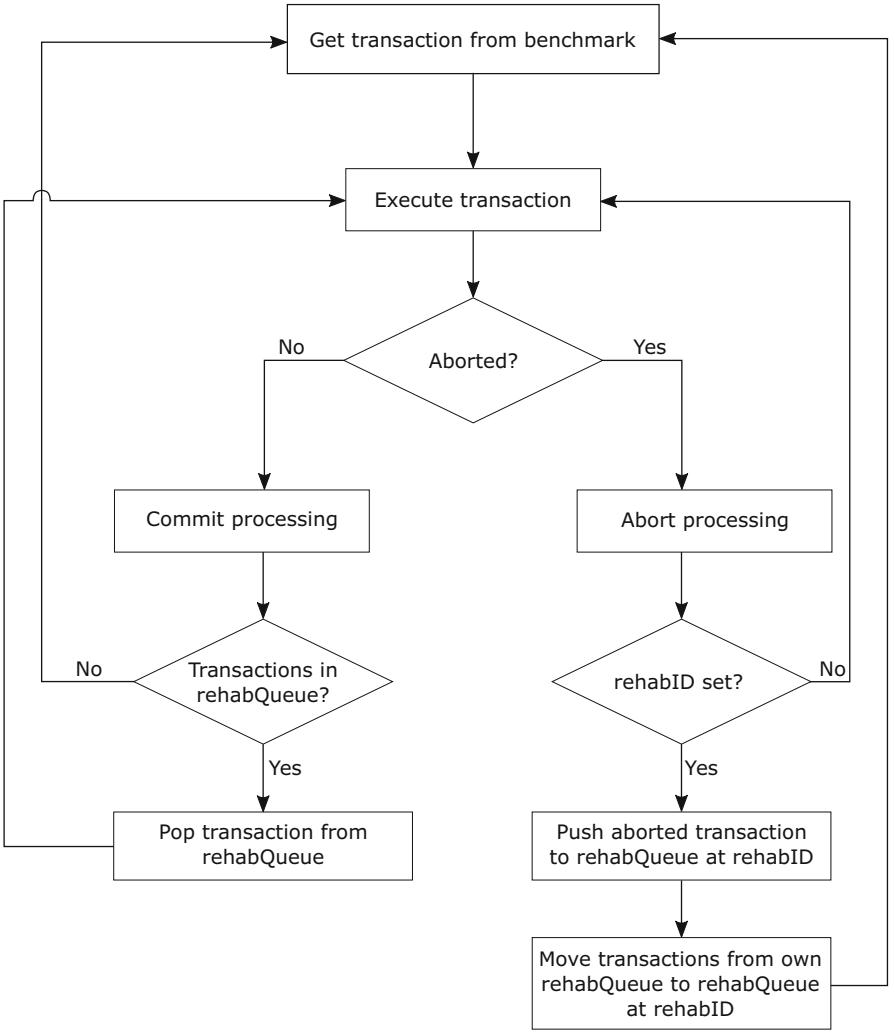

Fig. 2 Simplified flow of Purge-Rehab in RSTM

committed then the self-aborting transaction does not set its `rehabID` and restarts as usual.

3.2 Reducing Starvation

The scheme described above may increase starvation if transactions are left in rehabQueues for extended periods of time. Purge-Rehab uses two strategies to address such starvation. First, once a thread commits a transaction, it executes any transactions in its own rehabQueue before attempting to get a new transaction from the benchmark. Second, suppose thread 0 executing T1 wins one or more conflicts, aborting opponents, which are now queued in rehabQueue 0. It then conflicts with transaction T2,

executed by thread 1, and loses. Transaction T1 will be queued in rehabQueue 1. At this stage, thread 0 still has rehabilitation transactions in its rehabQueue that lost to T1. Their execution is delayed while they are in the queue, which increases starvation. Three approaches are discussed below, which trade-off starvation and livelock, for executing rehabilitation transactions.

The first approach (not shown) is for thread 0 to immediately pop rehabilitation transactions from its rehabQueue and execute them until none are left. Intuitively, this seems to increase starvation the least, because transactions held in a rehabQueue start executing as soon as the thread commits or aborts its current transaction. However, this may increase livelock (and thus, starvation) if thread 1 starts executing T1 while thread 0 concurrently executes rehabilitation transactions (as they had conflicted with T1 in the past). The second approach (shown) is for thread 0 to move all transactions in its rehabQueue to the rehabQueue of thread 1 at the same time that it places T1 in that rehabQueue. This prevents livelock from increasing as T1 will not re-conflict with transactions it has aborted in the past, but at the cost of increasing starvation for those transactions. However, there may be transactions in the rehabQueue that were not aborted by T1, and moving them may unnecessarily delay them. The third approach (not shown) extends the second approach and requires thread 0 to search through its rehabQueue and move only those transactions that were aborted by T1 to the rehabQueue of thread 1. As with the second approach, this prevents livelock from increasing, but only increases starvation for those transactions that were aborted by T1. The remaining transactions in the rehabQueue of thread 0 will likely be executed sooner than those aborted by T1, and thus be less starved. However, searching the rehabQueue makes moving rehabilitation transactions an O(n) operation, whereas it is O(1) in the second approach as the head of one queue is linked to the tail of other.

All three approaches were implemented and evaluated, and exhibited similar throughput and starvation results in low contention scenarios since transactions rarely moved between rehabQueues when using the second or third approaches. Under high contention however, the second approach achieved noticeably higher throughput. Analysis showed that rehabilitation transactions re-executed in the first approach conflicted and aborted several times, while the overhead of the third approach's rehabQueue search and pop/push of individual transactions between the concurrent queues was high. Consequently, the first and third approaches did not reduce starvation noticeably over the second approach, but did reduce throughput. The evaluation section only presents results from the second approach.

3.3 Cache Locality

It may seem that Purge-Rehab gives away performance by retrying an aborted transaction on a different thread and not exploiting cache locality by retrying on the original thread. However, in practice there is little cache locality to be exploited: in the ten benchmarks used in the evaluation , there was a 2 % increase in L1 Data misses in the worst case according to `perf`. Theoretically, aborted short transactions often have little L1 cache locality since the conflicting data invalidates a cache line that may contain most of the data accessed by the transaction. Retrying such transactions on the same

thread leads to an L1 cache miss anyway. Longer transactions that use multiple cache lines often write several times. If contention is high, then it is likely that multiple cache lines will be invalidated and need refetching, again giving limited L1 cache locality. In STMs that use undo logs, such as RSTM, cache locality is reduced further upon abort. Finally, executing a transaction on a different thread means that data will likely be found in the L2 cache, and the cost of fetching data from there is more than offset by the reduction in livelock as evidenced by the increase in throughput in the Sect. 4.

3.4 Example Execution

Figure 3 illustrates Purge-Rehab by way of example. Suppose that we have three threads with IDs 0, 1, and 2. These threads execute three transactions T1, T2 and T3 respectively. Suppose transaction T2 conflicts with T1 and T3 at different stages of its execution. Suppose that these conflicts are resolved in favor of T2. Thread 1 sets the rehabID for T1 and T3 to 1, and then aborts them. Threads 0 and 2 notice the status change, perform abort processing, and insert their transactions (T1 and T3, respectively) into the rehabQueue at index 1. They then get new transactions from the benchmark. Once thread 1 commits T2, it checks its rehabQueue at index 1, and proceeds to pop and execute any transactions found there.

4 Evaluation

This section evaluates the impact of Purge-Rehab on throughput, wasted work, and starvation in ten benchmarks with varying levels of contention. The evaluation compares eager RSTM, eager RSTM with Purge-Rehab, lazy RSTM, SwissTM, and TinySTM. All these STMs are considered high-performance.

4.1 Configuration and Benchmarks

RSTM [17] implements object-based and word-based transactional memory. Word-based RSTM is performance-tuned and consistently outperforms the object-based implementation by several fold. Purge-Rehab has been implemented and evaluated in both, but this article presents only the more relevant word-based results. Note that object-based results favor Purge-Rehab more than the word-based results presented here, because the object-based lazy is not as tuned as the word-based lazy.

We use the best-performing variants of word-based eager and lazy in RSTM, which utilize invisible reads, extendable-timestamps and the Patient and Passive contention managers. Four variants are presented: lazy with both contention managers (Lazy-Patient, and Lazy-Passive), eager with Passive (Eager-PRDisabled), and eager with Passive and Purge-Rehab (Eager-PREnabled). The Patient contention manager is designed for lazy, and is not conducive to high performance when used with eager.

Eager RSTM variants (including Purge-Rehab), TinySTM [25], and SwissTM [24] share similarities in that they are all word-based, lock-based, weakly atomic STMs that use eager validation, invisible reads, and extendable timestamps. RSTM and TinySTM

threadID	rehabID	Transaction Status
0	-1	T1-ACTIVE
1	-1	T2-ACTIVE
2	-1	T3-ACTIVE

Transaction T2 conflicts with T1 and T3 and wins in both cases. T2 sets the rehabID in T1 and T3 to 1, and aborts them.

0	1	T1-ABORTED
1	-1	T2-ACTIVE
2	1	T3-ABORTED

After abort processing, threads 0 & 1 check their respective rehabID. Since it is "1", instead of restarting the aborted transactions they put them in the rehabQueue at array index 1 and themselves go for new transactions.

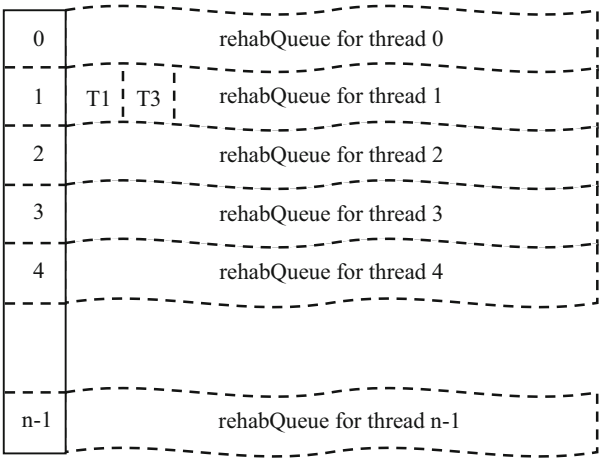

Fig. 3 Example flow of Purge-Rehab execution

both use a passive contention manager that always aborts the transaction that detects the conflict. SwissTM uses a two-phase contention manager [6] that uses the same strategy as RSTM and TinySTM when transactions are short (abort the detecting transaction), but uses the Greedy strategy [8] for long transactions, which aborts younger transactions. SwissTM determines transaction length based on the number of writes performed by a transaction, which is ten by default. Livelock is possible when the passive contention manager is used since multiple transactions may repeatedly attempt

Table 1 Microbenchmark parameters

Benchmark	Duration (S)	Range	Modify	Lookup
LLThread	10	1024	67	33
RBTree	10	1024	67	33
ListOverwriter	10	1024	67	33
RandomGraph	10	1024	100	0
StridePathology	10	1024	12	88
WWPathology	10	1024	100	0

Table 2 STAMP parameters

Benchmark	Arguments
Genome	-g16384 -s64 -n16777216
Intruder	-a10 -l128 -n262144 -s1
Kmeans	-m15 -n15 -t0.00001 -i random-n65536-d32-c16
Labyrinth	-i random-x512-y512-z7-n512
Vacation	-n50 -q40 -u40 -r128 -t83886

to write to hot data and abort themselves. SwissTM utilzes random back off when a transaction restarts, and increases the back off period based on the number of times the transaction has aborted. All RSTM variants including Purge-Rehab utilize undo logs while SwissTM and TinySTM use redo logs.

The configuration of the ListOverwriter, RandomGraph, RBTree, StridePathology, and WWPathology (write-write pathology) microbenchmarks are detailed in Table 1, and are identical to those used by Spear et al. [23]. Genome, Intruder, Kmeans, Labyrinth, and Vacation from the STAMP benchmark suite [16] are used with the default '++' parameters as shown in Table 2. Only Vacation's parameters were modified because (1) less than 0.5 % of transactions abort using the default parameters which is not useful for evaluating performance under contention, and (2) it is simple to induce higher contention in Vacation by modifying the parameters, which is not the case for all STAMP benchmarks. Some STAMP benchmarks were not used due to compilation errors. The experiments are performed on a 2 × 16-core Intel Xeon-E5 system with 64 GB RAM, running CentOS 6.3, and compiled with gcc 4.4.6. The results presented are averaged over ten runs and experiments go up to 32 threads.

4.2 Empirical Study

We first examine the performance of all STMs when there are no conflicts to ascertain if they offer similar throughput and scalability. Figure 4 shows the results from LLThread, which is a synthetic linked list benchmark where each thread operates on its own private linked list. There is a marginal difference (3.4 %) between RSTM variants including Purge-Rehab, which suggests that they are all optimized similarly, and

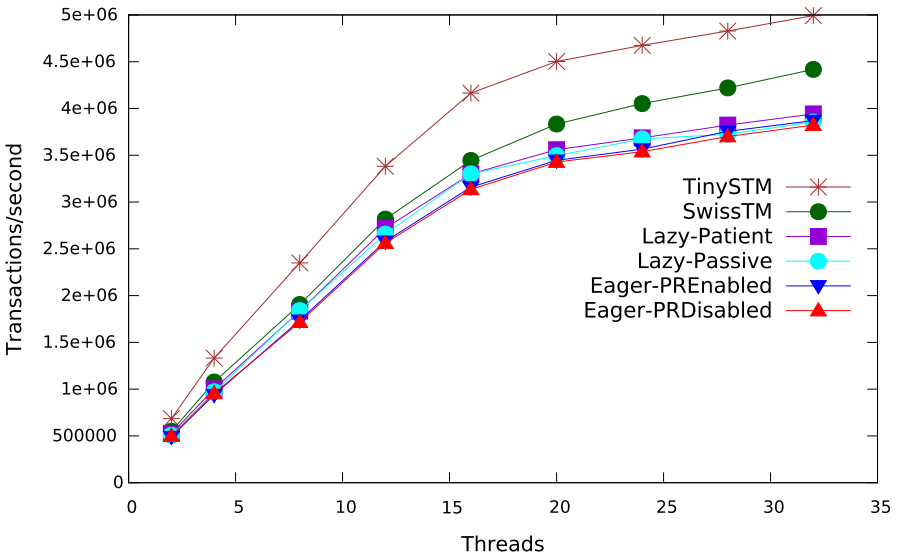

Fig. 4 LLThread: Each thread has a private linked list in this benchmark so that there are no conflicts, to show that eager and lazy empirically offer similar throughput

any significant performance difference between them in benchmarks with contention is likely due to the validation scheme used (eager or lazy). TinySTM outperforms all RSTM variants by about 33 % at all thread counts, and SwissTM does so by about 12 %. Consequently, when performance between the STMs varies by these amounts it could be attributed to the implementation rather than the validation scheme.

Figure 5 presents the speedup of all STMs as contention steadily increases using the EigenBench [12] contention test. EigenBench can isolate various application characteristics to test the performance impact of varying a single application characteristic such as contention. Ordinarily, one might choose to increase the work done by transactions in a benchmark to increase contention. However, it is then unclear how much performance variation is attributable to increased contention, and how much to increased transaction length. It can also be complex to determine how much extra work per transaction is needed to increase contention by regular intervals. The EigenBench contention test enables contention to be varied in isolation, and predictably. It uses an equation to determine the probability of contention based on certain EigenBench parameters.

Table 3 shows the parameters used to obtain these results. The EigenBench approach to the contention test was to use all the cores on the experimental platform, thus our test uses 32 threads. Purge-Rehab has a clear advantage over all the other STM variants that remains consistent across a wide range of contention values. The advantage is impressive because EigenBench transactions write to random locations in an array, and the ccNUMA architecture of the experimental platform implies the application is largely memory-bound. Purge-Rehab's advantage might increase in compute-bound applications, or those with better cache locality characteristics, provided there is a similar probability for conflict as shown in this test. Note that all the STM variants

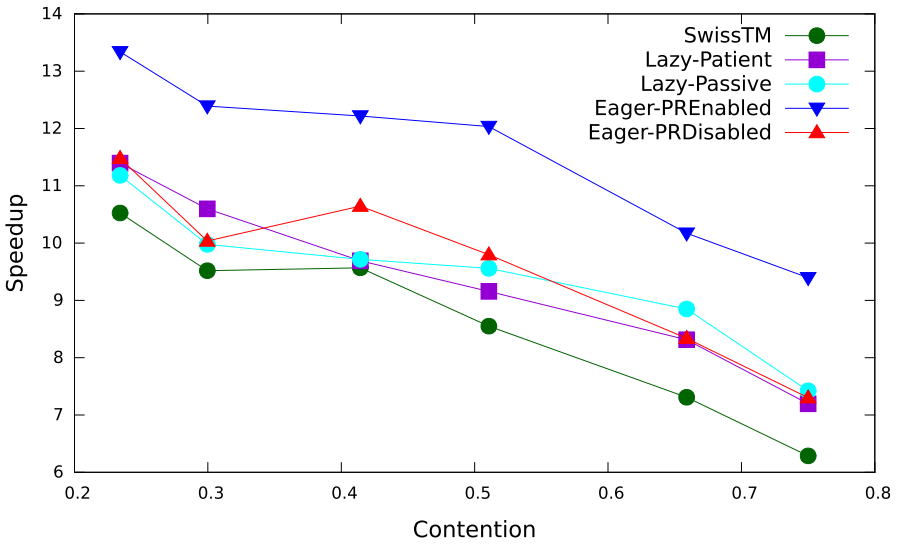

Fig. 5 EigenBench contention test to evaluate the performance of STM variants as contention increases. Contention is the probability of conflict and ranges from 0 to 1.0. Speedup versus a maximum of 32 threads

Table 3 EigenBench contention test parameters

N	lct	R1	W1	A1	Pconf
32	0	5	45	256k	0.23
32	0	5	45	196k	0.29
32	0	5	45	128k	0.41
32	0	5	45	96k	0.51
32	0	5	45	64k	0.65
32	0	5	45	48k	0.76

Pconf is the calculated probability of conflict using these parameters, which is labelled contention in Fig. 5

perform similarly as contention reaches its limits of 0 or 1.0, as they represent no conflicts and all conflicts, respectively.

4.3 Throughput

Figure 6a–e show the throughput results for the microbenchmarks. In combination with Fig. 7a–e, the microbenchmarks can be divided into three relative groups: high contention (WWPathology, StridePathology), medium contention (RandomGraph, ListOverWriter), and low contention (RBTree). The throughput of Eager-PREnabled degrades comparatively similarly to the other variants when threads are increased beyond the available 16 cores.

Figure 6a shows results for WWPathology, which contains a doubly-linked list, and half the transactions start from each end of the list concurrently, writing to each element until they have written to all elements in the list. This benchmark has been used

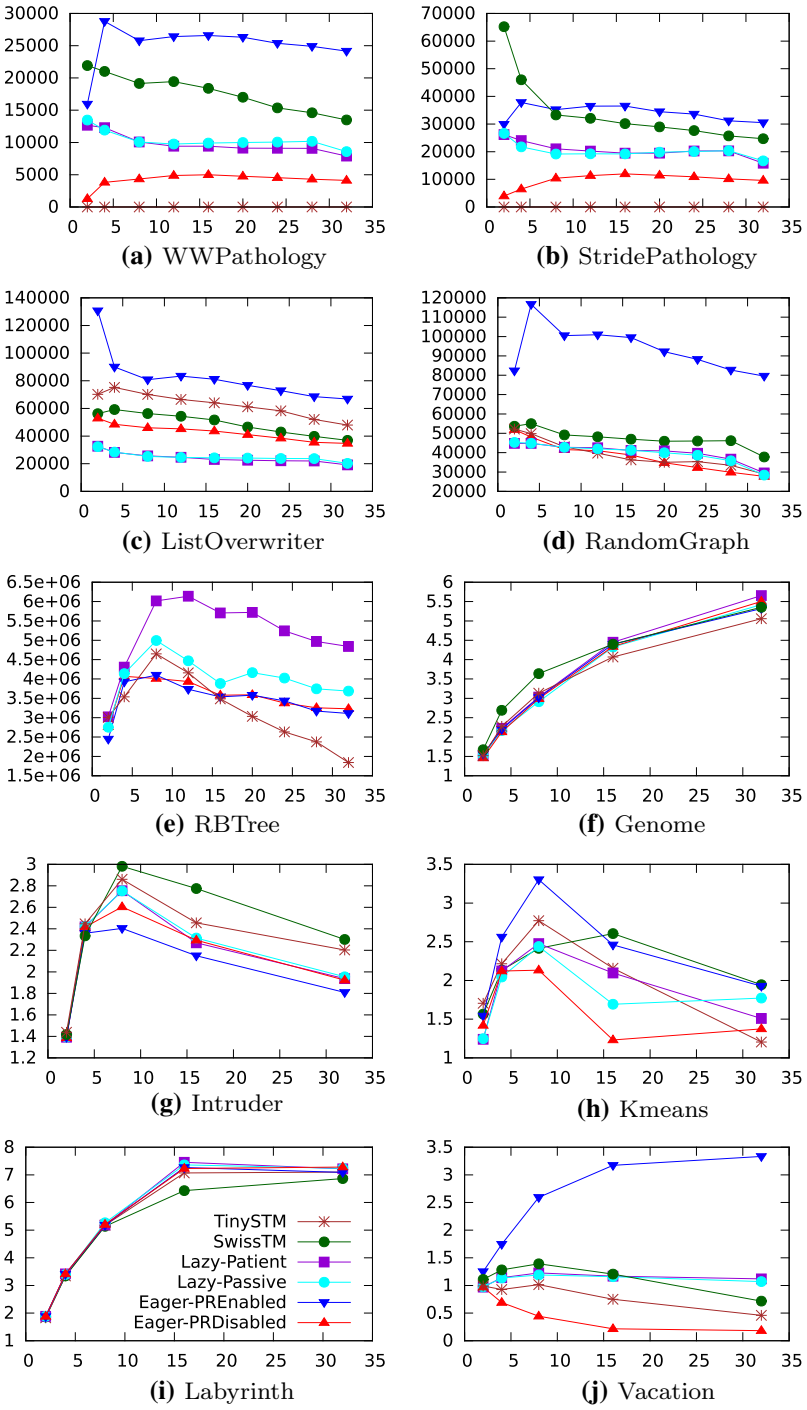

Fig. 6 Throughput for microbenchmarks and speedup for STAMP benchmarks. Numbers of threads on x-axis. Higher is better

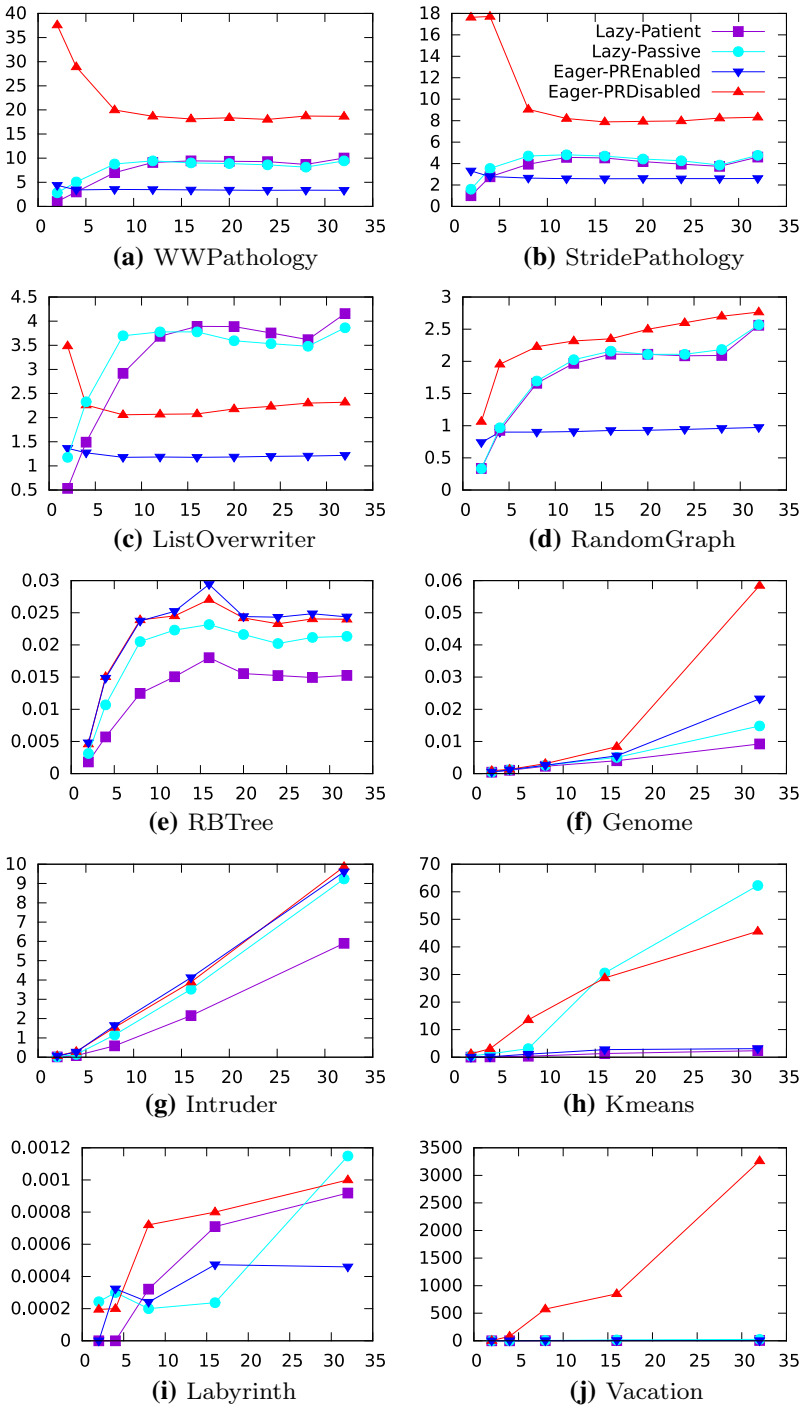

Fig. 7 Abort to commit ratio for all benchmarks under RSTM variants. *Numbers of threads* on x-axis. *Lower* is better

to show that eager livelocks under high contention and irregular access patterns. Eager cannot detect all conflicts early since half the transactions start from each end of the list. Rather, many transactions will abort near the start of their work as they detect conflicts with other transaction starting at the same side of the list, and then further aborts after 50 % of the list has been traversed by a transaction. Lazy RSTM variants outperform Eager-PRDisabled and TinySTM, with the latter livelocking. Eager-PRDisabled does not livelock entirely because its undo logs have the impact of backing off on abort. In SwissTM most transactions switch to the Greedy contention manager because they are write-heavy, and Greedy's timestamp-based contention management reduces livelock caused by the irregular accesses. Disabling the second phase and back-off-on-abort in SwissTM verified that it exhibited livelock similar to TinySTM (not shown). However, SwissTM still encounters some livelock due to transactions that detect a conflict before they can switch to the Greedy contention manager. These transactions abort and restart, possibly repeatedly. Eager-PREnabled reduces livelock most because it rehabilitates aborted transactions immediately, regardless of their length. It offers the best throughput that is around 2.5× better than lazy RSTM variants, and 5× better than Eager-PRDisabled.

Figure 6b shows results for StridePathology, which is identical to WWPathology except that transactions read all elements in the linked list and write to every 8th element starting from the nth element, which is randomly selected. StridePathology has a larger proportion of read-write conflicts compared to WWPathology, which are detected at the end of the transaction due to invisible reads. Write-write conflicts still exist and the irregular access pattern leads to Eager-PRDisabled having lower throughput than the lazy RSTM variants. As in WWPathology, TinySTM livelocks, Eager-PRDisabled undo logs act as backoff on abort to soften livelock, and SwissTM reduces it further with its two-phase contention manager and randomized back off upon abort. Eager-PREnabled has a smaller advantage over SwissTM since the use of invisible reads means that, for many transactions, conflicts are detected later than in WWPathology, which prevents doomed transactions from being aborted sooner.

Figure 6c shows results for ListOverwriter, where transactions write to each element they encounter in a linked list until they reach a random target element. Although contention is high due to write-write conflicts, the access pattern is regular, and eager can detect conflicts very early. This is reflected in the results as lazy RSTM variants have the lowest throughput due to detecting conflicts too late and executing doomed transactions to completion. Throughput differences between the eager STM variants are similar to those seen in LLThread in Fig. 4. except for Eager-PREnabled which almost doubles throughput over Eager-PRDisabled. Figure 7c shows that Eager-PREnabled has far fewer aborts, thus its performance boost is likely a result of reducing the livelock caused by transactions repeatedly conflicting and aborting.

Figure 6d shows results for RandomGraph, which is an undirected graph represented by adjacency lists (each node has a sorted linked list of its neighbors). Nodes are stored in a global sorted linked list. Insert operations add a new node to the global linked list, and connect it to four randomly chosen neighbors by modifying the source and target nodes' adjacency lists (the new neighbors are located by re-traversing the global linked list). Remove operations similarly modify the global linked list and neighbors' adjacency lists. Contention in this benchmark is due to the relatively long

transactions accessing multiple linked lists, and re-traversing the global linked list in the add operation. Eager-PRDisabled, TinySTM, and the lazy RSTM variants perform within 20 % of each other. The lazy STMs underperform by completely executing long transactions that are doomed. Eager-PRDisabled and TinySTM restart aborted transactions, but the likelihood of re-conflicting and re-aborting is high since most transactions have a long duration. SwissTM is 15–30 % higher because its two-phase manager prevents long running transactions from self-aborting due to conflict with younger transactions, particularly add operations that re-traverse the global linked list. Eager-PREnabled improves throughput by about 250 % because it is efficient at reducing wasted work from doomed transactions, and because threads fetch new transactions upon abort, and it is less likely that the new transactions will access the same nodes as concurrent transactions.

Figure 6e shows results for RBTree[1], in which transactions search down a red-black tree for a target node, and insert/remove operations additionally modify the tree back upwards. The downward read-only search phase employs pointer dereferencing, which consumes a large portion of each transaction's time. Updates to nodes back up the tree are faster, since most nodes are in cache. This behavior disadvantages eager STMs since transactions will perform the time-consuming search using invisible reads, but then conflict and self-abort while writing during the final stages of the transaction. Consequently, Lazy-Patient outperforms all eager STMs by around 85 %, and Lazy-Passive by 45 %. Eager STMs perform similarly, with TinySTM degrading more than the others when the number of threads rises beyond the number of cores. Eager-PREnabled provides little benefit over Eager-PRDisabled since a transaction's conflict is detected near the end of the its work, and the opponent will have finished by the time the conflicting access is retried. This is the only benchmark in which eager is not competitive with lazy, and is an opportunity for further exploration.

Figure 6f–j show speedup results for the STAMP benchmarks used[2]. These benchmarks generally have less contention compared to some of the previous microbenchmarks so there is less opportunity for Purge-Rehab to have an impact, and differences between eager and lazy are less prominent. In combination with Figure 7f–j, the benchmarks can be divided into three relative groups: high contention (Vacation), medium contention (Kmeans, Intruder), and low contention (Genome, Labyrinth). Overall an eager STM is always among the top performers, Eager-PREnabled is similar to or better than lazy RSTM variants in four benchmarks, and has a worst-case drop of 10 % in one benchmark (Intruder–8 threads). In Genome, Intruder, and Kmeans, SwissTM and TinySTM improve over eager RSTM variants similar to their advantage in LLThread.

Genome scales to 16 threads and the RSTM variants, including Purge-Rehab, have almost identical results. Labyrinth is similar with results in a tighter range since the benchmark is more memory bound. Results for Intruder also follow a similar pattern, with all STMs within 20 % of each other, but scaling stops at 8 threads due to contention. Eager-PREnabled has no advantage over Eager-PRDisabled because

[1] SwissTM is missing from RBTree because its results were odd, and it seems to be an implementation issue.

[2] Note to reviewers: We were not able to get all STAMP benchmarks to run successfully in our environment.

transactions write to data at the end, like in RBTree. Additionally, this benchmark has the shortest transactions [15], and the overhead of Eager-PREnabled rehabilitating transactions causes a 5 % drop in performance compared to Eager-PRDisabled.

Kmeans has small transactions like Intruder so it also scales only to 8 cores, but the performance difference between the STMs is a little more prominent at 8 threads than in Intruder due to higher contention in this benchmark. Eager-PREnabled achieves the highest speedup because transactions write early and regularly, and the accesses are regular, similar to the situation in ListOverwriter. Eager-PREnabled is 55 % higher than Eager-PRDisabled at 8 threads.

Vacation's input parameters were changed so that transactions modify a larger number of variables, because this was a simple way to significantly increase contention. Transactions imitate customers booking cars, flights, and hotels atomically, and effectively read and write random data, so accesses are irregular like WWPathology and StridePathology, but transactions are shorter and scalability is possible since accesses are not as pathological. Nevertheless, there are similar trends: TinySTM and Eager-PRDisabled perform worse than lazy RSTM variants because self-aborting increases livelock in irregular accesses. SwissTM is better than those eager STMs, again because its two-phase contention manager prioritizes transactions that write a few locations without conflicting. However, it drops below lazy at 32 threads because more conflicts are detected early and fewer transactions get to the second phase of the contention manager. Eager-PREnabled's rehabilitation scheme has a significant impact in this benchmark because, like in RandomGraph, the aborting thread fetches a new transaction, and that has less likelihood of conflicting with concurrent transactions.

4.4 Aborts and Wasted Work

Figure 7 shows the abort to commit ratios in the RSTM variants for the benchmarks. These are presented to evaluate the impact of Purge-Rehab on wasted work. Only RSTM variants are presented here to see the direct impact of using Purge-Rehab on eager RSTM, and how that compares to lazy RSTM. In benchmarks with contention such as Figure 7a–d, the major portion of wasted work performed by Eager-PRDisabled is from transactions that repeatedly conflict and abort, whereas by lazy RSTM variants it is due to executing doomed transactions. Purge-Rehab reduces repeat conflicts in eager RSTM to remove livelock, and these results show that the gains in throughput presented earlier are matched by reductions in wasted work. For example, Fig. 6a showed Eager-PREnabled outperformed lazy RSTM variants by almost 3x in WWPathology, and in Fig. 7a there is a matching reduction in wasted work to one-third.

4.5 Starvation

We consider starvation to be any delay experienced by a transaction between its start and its commit due to repeatedly aborting, backing off on abort, or waiting in a rehabQueue when Purge-Rehab is used. However, sampling the exact start and end times of transactions adds considerable overhead, and may be inaccurate for small transactions. Instead, the percentage of transactions committed by each thread (out of

100) may be used as a measure of starvation, because any thread with a proportionally lower commit percentage likely executed transactions that aborted several times before finally committing. As in the previous section, only RSTM variants are presented here to see the direct impact of using Purge-Rehab on eager RSTM, and how that compares to lazy RSTM.

Figure 8 shows the percentage of transactions committed by each thread in a 16-thread run. In the microbenchmark results eager RSTM variants exhibit less variance in commit percentages than lazy RSTM variants, and consequently starvation is less severe. For example, in StridePathology the lazy RSTM variants' threads commit percentages range between 0.2 and 15.2 %, whereas the eager RSTM variants range between 5 and 8 %. Variance also drops as contention drops, but remains high in lazy RSTM variants in all benchmarks, except in the low contention benchmark RBTree where Lazy-Passive has similar variance to the eager RSTM variants, and Lazy-Patient has slightly higher variance.

Eager RSTM variants continue to do well across most STAMP benchmarks. Low contention benchmarks Genome and Labyrinth exhibit little difference in variance, and Eager RSTM variants have lower variance in Kmeans. Eager-PREnabled seems to increase variance in Vacation, but this is because the benchmark is designed to make each thread commit a specific number of transactions, thus it appears all the other RSTM variants have no variance at all. Since Eager-PREnabled rehabilitates transactions, some threads execute more transactions than others, which misleadingly appears as starvation in this case. Intruder is the only benchmark where eager RSTM variants have visibly worse starvation than lazy RSTM variants. Intruder's transactions are similar to RBTree's, and eager RSTM variants also had relatively larger variance in that benchmark. Variance increases in Intruder because it has higher contention than RBTree (see Figs. 7e and 7g). Still, this variance is not as pronounced as lazy's variance in the higher contention microbenchmarks.

Eager-PREnabled reduces variance in commit percentages over Eager-PRDisabled, which indicates that it reduces starvation. The rehabQueue was never longer than 20 transactions, which occurred in WWPathology. A fair variant of lazy exists that uses dynamically increasing transaction priorities to reduce variance in commit percentages. In practice it reduces variance, achieving results slightly worse than the eager RSTM variants. However, it requires read visibility and experimental results showed the overhead reduces throughput significantly, making any comparison on starvation less meaningful.

5 Related Work

Spear et al. [23] consider lazy as the higher performing strategy when contention occurs because it avoids pathologies that decrease throughput. They introduce optimizations to reduce overhead in lazy, and priority scheduling to reduce starvation in lazy. This article shows that it is possible to tune eager such that it outperforms lazy in a number of contention cases while maintaining low starvation.

Titos-Gil et al. [26] improve the performance of eager hardware transactional memory (HTM) under contention, and show that an optimized eager HTM could outperform

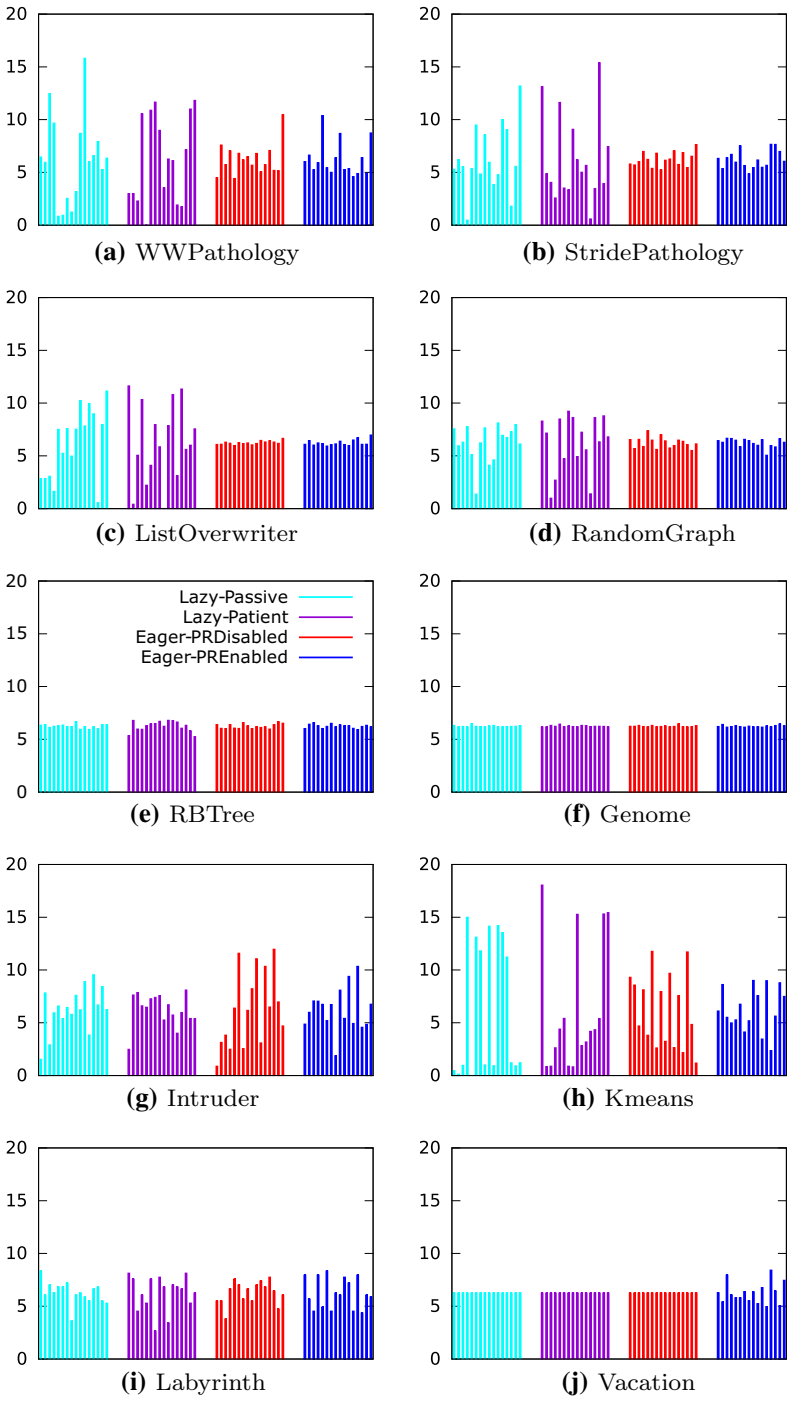

Fig. 8 Percentage of transactions committed by each thread in a 16-thread run to analyze starvation

a lazy HTM. Their approach to optimizing was very different to ours since it was in HTM. We have shown that an eager STM can be improved to outperform lazy STM in many cases under contention as well.

Harris et al. [10] introduced a retry mechanism where a programmer may define a condition for which an aborted transaction must wait before restarting. The thread executing the transaction blocks waiting for the condition. In contrast, Purge-Rehab is an automatic scheduling technique that does not require programmer input, and does not incur the high overheads of blocking and resuming threads.

Bai et al. [3] proposed a key-based technique to put transactions that are likely to conflict in a single queue for sequential execution. The key calculation requires application specific information, which may be complex or impossible for several applications, as well as an overhead on the programmer. Purge-Rehab queues transactions without requiring any application-specific information or programmer input.

Dolev et al. [4] presented CAR-STM, which also maintains per-core queues of transactions, and enqueues transactions when they conflict, similar to Purge-Rehab. However, CAR-STM was not implemented in a state-of-the-art word-based STM, and its overhead may degrade throughput in such a setting. CAR-STM was only evaluated against eager implementations, and its starvation properties were not considered. Purge-Rehab has been shown to have lower starvation and higher throughput compared to lazy, even in cases where lazy previously outperformed eager.

Ansari et al. [1] presented Steal-on-Abort, where transactional jobs are submitted to thread pools by application threads and are executed asynchronously. Transactions steal opponents that they abort in order to avoid repeat conflicts. Like CAR-STM, the thread pool framework may have high overhead when implemented in a high performance STM, and it was only evaluated against eager implementations.

Wang et al. [27] presented a machine learning derived framework for adapting TM variables (including validation strategy) at runtime based on offline and online learning to achieve high performance. Their work is orthogonal to ours, and could be extended to enable or disable Purge-Rehab.

Yoo and Lee [28] presented Adaptive Transaction Scheduling which sleeps threads that have been aborting transactions above a user-defined threshold. The threads are enqueued in a global queue, and threads are resumed sequentially. Atoofian [2] presents another scheme to sleep/resume threads. Like Harris' retry mechanism, the overhead of sleeping/resuming threads is likely to be too high in performance STMs. Additionally, neither explore performance with respect to lazy, nor investigate starvation.

6 Conclusion

The performance of transactional memory implementations is constantly being improved. Towards this end, this article focuses on improving eager validation software transactional memory, and shows that eager can be competitive with lazy. Eager's large window for detecting conflicts can lead to repeated aborts between transactions, and this in turn can lead to livelock that degrades throughput. We design Purge-Rehab to reduce livelock by rehabilitating transactions so that they do not repeatedly abort each other, and discussed how design alternatives affected starvation.

We implemented Purge-Rehab in eager RSTM and shown that empirically both eager and lazy RSTM are implemented to have high throughput and scalability, and that Purge-Rehab scales better under contention. We reconfirm that pathological benchmarks do cause livelock in eager STMs, and that Purge-Rehab is efficient at reducing livelock. Its overhead is low enough for eager to achieve higher throughput than lazy in several high contention benchmarks, including those that have irregular access patterns. Analysis shows that this is a result of reducing wasted work in repeated aborts. However, lazy stays on top in the red-black tree benchmark, which is pathological for any eager STM with invisible reads due to a long read phase followed by a short write phase in each transaction. Additionally, Purge-Rehab shows resilience to starvation, whereas lazy is more prone to it.

To the best of our knowledge, this article presents one of the few comparisons between eager and lazy using modern word-based STMs, and where eager is shown to outperform lazy in several benchmarks. We hope it encourages further study of how both eager and lazy validation may be tuned in the pursuit of performance.

Acknowledgments This work was supported by grant INF-1678 under the National Science and Technology Innovation Plan run by the King Abdulaziz Center for Science and Technology. Abubakar Siddique is supported by the Science and Technology Unit at Umm Al Qura University. Dr. Mikel Luján is supported by a Royal Society University Research Fellowship.

References

1. Ansari, M., Kotselidis, C., Jarvis, K., Luján, M., Kirkham, C., Watson, I.: Steal-on-abort: Improving transactional memory performance through dynamic transaction reordering. In: HIPEAC '09: Fourth International Conference on High Performance and Embedded Architectures and Compilers, January 2009
2. Ehsan, A.: Improving performance of software transactional memory through contention locality. J. Supercomput. **64**, 527–547 (2013)
3. Bai, T., Shen, X., Zhang, C., Scherer, W.N., Ding, C., Scott, M.L.: A key-based adaptive transactional memory executor. In: IPDPS '07: Proceedings of the 21st International Parallel and Distributed Processing Symposium, March 2007
4. Dolev, S., Hendler, D., Suissa, A.: CAR-STM: Scheduling-based collision avoidance and resolution for software transactional memory. In: PODC '07: Proceedings of the 26th Annual ACM Symposium on Principles of Distributed Computing, August 2008
5. Dragojević, A., Guerraoui, R., Kapałka, M.: Dividing transactional memories by zero. In: TRANSACT '08: 3rd ACM SIGPLAN Workshop on Transactional Computing, February 2008
6. Dragojević, A., Guerraoui, R., Kapalka, M.: Stretching transactional memory. In: ACM Sigplan Notices, volume 44, pages 155–165. ACM, 2009
7. Felber, P., Fetzer, C., Riegel, T.: Dynamic performance tuning of word-based software transactional memory. In: PPoPP '08: Proceedings of the 13th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming. ACM Press, February 2008
8. Guerraoui, R., Herlihy, M., Kapałka, M., Pochon, B.: Robust contention management in software transactional memory. In: SCOOL '05: Workshop on Synchronization and Concurrency in Object-Oriented Languages, October 2005

9. Guerraoui, R., Herlihy, M., Pochon, B.: Toward a theory of transactional contention managers. In: PODC '05: Proceedings of the 24th Annual Symposium on Principles of Distributed Computing. ACM Press, July 2005
10. Harris, T., Marlow, S., Peyton-Jones, S., Herlihy, M.: Composable memory transactions. In: PPoPP '05: Proceedings of the tenth ACM SIGPLAN symposium on Principles and practice of parallel programming, June 2005
11. Herlihy, M., Moss, J.E.B.: Transactional memory: Architectural support for lock-free data structures. In: ISCA '93: Proceedings of the 20th Annual International Symposium on Computer Architecture. ACM Press, May 1993
12. Hong, S., Oguntebi, T., Casper, J., Bronson, N., Kozyrakis, C., Olukotun, K.: Eigenbench: a simple exploration tool for orthogonal tm characteristics. In: Proceedings of the IEEE International Symposium on Workload Characterization (IISWC'10), IISWC '10, pp. 1–11, Washington, DC, USA, 2010. IEEE Computer Society
13. Intel. Thread building blocks. https://www.threadingbuildingblocks.orgs. Accessed Sept 2014
14. Lomet, D.B.: Process structuring, synchronization, and recovery using atomic actions. In: Proceedings of an ACM conference on Language design for reliable software, March 1977
15. Walther, M., Patrick, M., Pascal, F., Adi, S., Danny, H., Alexandra, F., Julia, L.L., Gilles, M.: Scheduling support for transactional memory contention management. ACM Sigplan Not. **45**(5), 2010
16. Minh, C.C., Chung, J.W., Kozyrakis, C., Olukotun, K.: Stamp: stanford transactional applications for multi-processing. In: IISWC '08: Proceedings of the IEEE International Symposium on Workload Characterization. IEEE, 2008
17. RSTM. Homepage. http://www.cs.rochester.edu/research/synchronization/rstm. Accessed Sept 2014
18. Scherer III, W., Scott, M.: Contention management in dynamic software transactional memory. In: CSJP '04: Workshop on Concurrency and Synchronization in Java Programs, July 2004
19. Scherer III, William, Scott, Michael: Advanced contention management for dynamic software transactional memory. In: PODC '05: Proceedings of the 24th Annual Symposium on Principles of Distributed Computing. ACM Press, July 2005
20. Shavit, N., Touitou, D.: Software transactional memory. In: PODC '95: Proceedings of the 14th Annual ACM Symposium on Principles of Distributed Computing. ACM Press, Aug 1995
21. Shriraman, A., Dwarkadas, S.: Refereeing conflicts in hardware transactional memory. In: ICS '09: Proceedings of the 23rd international Conference on Supercomputing, 2009
22. Shriraman, A., Spear, M.F., Hossain, H., Marathe, V.J., Dwarkadas, S., Scott, M.L.: An integrated hardware-software approach to flexible transactional memory. In: ISCA '07: Proceedings of the 34th Annual International Symposium on Computer Architecture. ACM Press, 2007
23. Spear, M.F., Dalessandro, L., Marathe, V.J., Scott, M.L.: A comprehensive strategy for contention management in software transactional memory. In: PPoPP '09: Proceedings of the 14th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, 2009
24. SwissTM. Homepage. http://lpdserver.epfl.ch/transactions/wiki/doku.php?id=swisstm. Accessed Sept 2014
25. TinySTM. Homepage. http://www.tmware.org/tinystm. Accessed Sept 2014
26. Gil, R.T., Negi, A., Acacio, M.E., Garcia, J.M., Stenstrm, P.: Eager beats lazy. IEEE Trans. Parallel Distrib. Syst. **24**, 2192–2201 (2013)
27. Wang, Q., Kulkarni, S., Cavazos, J., Spear, M.F.: A transactional memory with automatic performance tuning. ACM Trans. Arch. Code Opt. (TACO) **8**(4), 54 (2012)
28. Yoo, R.M., Lee, H.H.S.: Adaptive transaction scheduling for transactional memory systems. In: SPAA '08: Proceedings of the 20th annual symposium on Parallelism in algorithms and architectures, March 2008