An Efficient Software Transactional Memory Using Commit-Time Invalidation *

Justin E. Gottschlich, Manish Vachharajani, and Jeremy G. Siek

Department of Electrical, Computer, and Energy Engineering, University of Colorado at Boulder
{gottschl, manishv, jeremy.siek}@colorado.edu

Abstract

To improve the performance of transactional memory (TM), researchers have found many eager and lazy optimizations for *conflict detection*, the process of determining if transactions can commit. Despite these optimizations, nearly all TMs perform one aspect of lazy conflict detection in the same manner to preserve serializability. That is, they perform *commit-time validation*, where a transaction is checked for conflicts with previously committed transactions during its commit phase. While commit-time validation is efficient for workloads that exhibit limited contention, it can limit transaction throughput for contending workloads.

This paper presents an efficient implementation of *commit-time invalidation*, a strategy where transactions resolve their conflicts with in-flight (uncommitted) transactions *before* they commit. Commit-time invalidation supplies the contention manager (CM) with data that is unavailable through commit-time validation, allowing the CM to make decisions that increase transaction throughput. Commit-time invalidation also requires notably fewer operations than commit-time validation for memory-intensive transactions, uses zero commit-time operations for dynamically detected read-only transactions, and guarantees full opacity for any transaction in $O(N)$ time, an improvement over incremental validation's $O(N^2)$ time. Our experimental results show that for contending workloads, our efficient commit-time invalidating software TM (STM) is up to $3\times$ faster than TL2, a state-of-the-art validating STM.

Categories and Subject Descriptors: D.1.3 [Concurrent Programming]: Parallel Programming.

General Terms: Algorithms, Design, Performance.

Keywords: Commit-Time Invalidation, Software Transactional Memory.

1. Introduction

Transactional memory (TM) is a modern concurrency control paradigm that provides a simple parallel programming model to reduce the difficulty of writing parallel programs [15, 28]. Many TMs use an optimistic concurrency model in which all operations are executed concurrently and the operations which violate serializability [15, 22] are undone. For TMs to provide an optimistic concurrency model such that the transaction commit order is serializable, transactions are generally atomic and isolated [17]. Unfortunately, maintaining atomicity and isolation incurs computational overhead that some researchers argue is too great for TM's practical adoption [2]. To address these concerns, researchers have found innovative ways to reduce the overhead of atomicity and isolation by optimizing conflict detection [4, 5, 18, 24, 30, 31].

Conflict detection, the process of determining if transactions can commit [30], is usually implemented as a conservative overestimation of transaction serializability. A transaction can commit when it is *consistent*; that is, it is free of transactional conflicts. A *transactional conflict* is defined as the non-null intersection between one transaction's write elements and another transaction's read and write elements [15, 19, 24]. A *true conflict* requires that at least one transaction be aborted for the TM's commit order to remain serializable. A *false conflict* exists when a specific commit order is chosen, rather than a transaction to abort, that preserves serializability. While significant work has been done in the area of conflict detection and resolution, nearly all TMs perform *commit-time validation*, a strategy where a single transaction's read elements, and sometimes its write elements, are checked for consistency at commit-time.

Commit-time validation typically uses version numbers associated with memory to track transactional conflicts [25]. In general, the version numbers of a transaction's read and write elements (also known as *read* and *write sets*) are compared against the version numbers of the same memory stored globally. If a version mismatch is found, the validating transaction is aborted since a previously committed transaction has updated the same memory. If no mismatch is found the transaction is consistent and can be committed. While commit-time validation is efficient for workloads that exhibit little contention, it limits *transaction throughput*, the number of transactions that commit per second, for contending workloads. This is because it does not determine how many in-flight transactions will be aborted due to a transaction's commit.

In this paper we consider *commit-time invalidation*, a conflict detection strategy in which transactional conflicts are found by comparing the memory of a committing transaction against the memory of in-flight transactions. Commit-time invalidation differs from commit-time validation in that all a committing transaction's conflicts with in-flight transactions are found and resolved *before* the transaction commits. Conflicts are sent to the *contention manager* (CM), the process that decides which transactions make forward progress [11, 14, 26], for resolution. The CM resolves conflicts by either (1) aborting all conflicting in-flight transactions, (2) aborting the committing transaction, or (3) stalling the committing transaction until the conflicting in-flight transactions have commit-

* This work was supported in part by Raytheon Company.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

CGO'10, April 24–28, 2010, Toronto, Ontario, Canada.
Copyright © 2010 ACM 978-1-60558-635-9/10/04...$10.00

ted or aborted [30]. Through this mechanism, commit-time invalidation can notably increase transaction throughput when compared to commit-time validation for contending workloads.

Although invalidation is not a new idea [7, 9, 13, 14, 27, 29, 30], to the best of our knowledge, no prior work has implemented an efficient TM – i.e., a TM that is competitive with the state-of-the-art – that only uses invalidation. Inefficiencies found in prior attempts have steered TM research toward validation. In this paper we demonstrate that a TM which only uses commit-time invalidation can be implemented efficiently. In doing so, this paper presents the following contributions:

1. Full commit-time invalidation can increase transaction throughput by supplying a CM with more information than is possible using commit-time validation, allowing the CM to make informed and efficient decisions.
2. Optimized commit-time invalidation is asymptotically faster than validation for memory-intensive transactions.
3. Commit-time invalidation requires zero operations to identify conflicts in dynamically detected read-only transactions and ensures opacity[1] [12] for any transaction in $O(N)$ time, where N is the number of elements in the transaction's read set, an improvement over incremental validation's $O(N^2)$ time.
4. Our efficient commit-time invalidating software TM (STM), InvalSTM, is over $3\times$ faster than TL2 [4], a state-of-the-art validating STM, for certain contending workloads.

2. Background

In this section we present a history of invalidation and explain why prior efforts have only partially explored it, leaving many of its powerful optimizations unexplored.

Overview of Invalidation vs. Validation One of the most computationally expensive aspects of TM is the process of detecting conflicts (i.e., discovering when the commit of two or more transactions will result in an execution order that is not serializable).

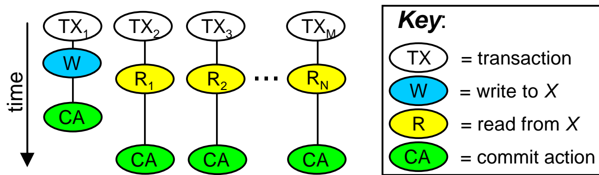

Figure 1. 1-Writer and N-Readers: A Highly Contending, Highly Concurrent Workload.

If, for a moment, we restrict ourselves to commit-time conflict detection, we can see why an invalidating TM can exploit more transaction throughput than a validating TM. Consider the scenario depicted in Figure 1 where one transaction writes to variable X and N transactions subsequently read the value of X. A TM using commit-time validation (see Figure 2 for an overview) and *lazy write acquisition*[2], will successfully validate the writer transaction at its CA (the commit action). The writer will then update X's global value and version number and commit. However, this behavior will cause all N readers to abort. When the readers reach their CA, they will be required to abort because their view of X will be inconsistent with main memory due to TX_1's commit. Thus,

[1] Opacity is the property where doomed transactions are identified before they can execute harmful operations [12].

[2] Lazy write acquisition is where written memory is exclusively acquired at the transaction's commit phase [17].

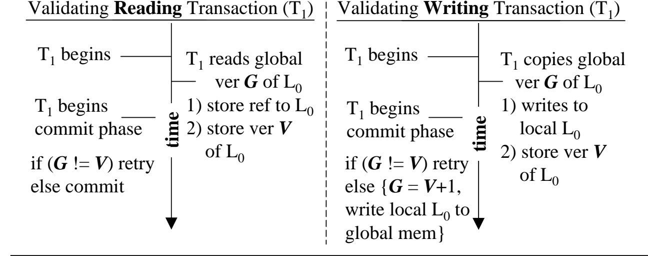

Figure 2. Transaction Using Commit-Time Validation.

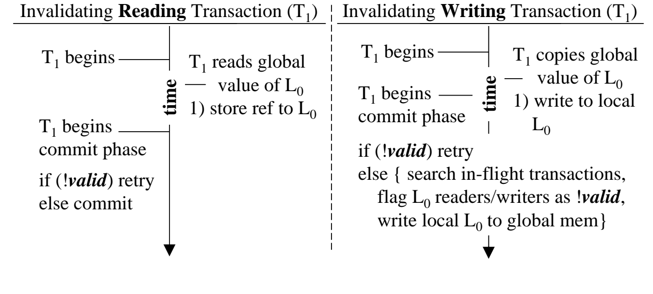

Figure 3. Transaction Using Commit-Time Invalidation.

commit-time validation effectively eliminates all concurrency between the readers and writer, a serious issue if N is large like it is in a number of workloads and systems [8, 21, 32].

Now, consider commit-time invalidation (see Figure 3 for an overview) for the scenario in Figure 1. When TX_1 reaches CA it scans all N in-flight transactions for conflicts. Each conflict is sent to the CM which, based on the number of contending *read-only transactions* (transactions that only read memory), can make an informed decision to abort TX_1 and allow the concurrent commit of the N readers. When N is large, this behavior dramatically increases transaction throughput.

Furthermore, in theory, an invalidating TM can always make the same decisions as a validating TM. Therefore, in cases when validation is efficient, such as non-contending workloads, invalidation can behave as validation does, making invalidation more powerful. Unfortunately, prior to this work, validating TMs have proven to be more efficient in practice. To understand why this is so, we must first discuss the different types of conflicts, conflict detection strategies, and the different strategies for maintaining read and write sets.

Types of Conflicts. Conflicts arise when two or more in-flight transactions access the same memory before committing and come in three varieties: *W-W*, *W-R*, and *R-W* [24]. W-W conflicts occur when two transactions write to the same memory. W-R conflicts occur when one transaction writes to a memory location and another transaction subsequently reads it (or vice versa for R-W conflicts). Without considering dependence-aware TM [23], W-W conflicts are generally classified as true conflicts [30], while W-R and R-W conflicts may be false; there may exist a serializable commit order so W-R and R-W conflicts can be resolved without transactional aborts.

Eager and Lazy Conflict Detection. Conflict detection can be performed *eagerly* or *lazily*. Eager conflict detection happens sometime before a transaction commits; lazy conflict detection happens at commit-time. TMs generally perform some conflict detection at commit-time even if the majority is handled eagerly.

Visible and Invisible Read and Write Sets. To detect conflicts, TMs maintain read and write sets (i.e., the set of locations a trans-

action has read and written, respectively). A transaction's read or write set is said to be *visible* if it can be seen by other transactions, otherwise it is called *invisible*. For invalidating TMs to detect W-W conflicts, write sets must be visible so that write sets from different transactions may be compared. Likewise, for invalidating TMs to detect W-R or R-W conflicts, read sets must be visible.

2.1 The Rise and Fall of Invalidating TMs

Partial invalidation is the process in which a TM performs some invalidation, either eagerly or lazily, but does not guarantee all conflicts with in-flight transactions are resolved before a transaction commits. Because of this some conflicts are missed by invalidation and subsequently require resolution through validation. Partial invalidation was first implemented in Herlihy et. al's DSTM for eager W-W conflicts, and in Harris and Fraser's WSTM for lazy W-W conflicts [13, 14]. Scott followed by proposing several invalidation techniques, which were used in Spear et al.'s *mixed invalidation* (using eager W-W and lazy W-R / R-W invalidation) in RSTM [27, 30]. Fraser and Harris's OSTM followed by implementing lazy invalidation for W-W conflicts [7].

To maximize concurrency, these TMs use *non-blocking synchronization*: they avoid the use of locks for shared data structures and instead rely on wait-free, lock-free (OSTM), or obstruction-free (DSTM, RSTM) synchronization. To maintain the visible read and write sets needed for invalidation, they use ownership records or *orecs*, which are data structures that associate memory elements with the transactions that access them [28]. On the first read or write of each memory location, the transaction is added to the orec for that location. Upon commit, the transaction is removed from all the orecs in which it was added.

In their most efficient implementation, orecs are computationally expensive. Spear et al. explain that maintaining complete visible readers per transactional memory location, necessary for W-R / R-W invalidation, incurs too much overhead for a TM to be practical [30]. They found the overhead associated with managing such readers costs more than the $O(N^2)$ overhead of *incremental validation*, the process of revalidating all of a transaction's read elements each time a new memory location is opened for reading (i.e., when a memory location is first read) [17, 30].

To gain some of the benefits of invalidation without incurring the full penalty of orecs, Spear et al.'s mixed invalidation uses one word per memory location to track readers. This reduces maintenance overhead, but limits W-R invalidation to 32 conflicts (or 64 on a 64-bit architectures) per memory location. Thus, the system must still perform version-based validation for > 32 threads.

Some *lock-based* STMs (TL2, RingSTM, and JudoSTM), those STMs which use mutual exclusion operations at their core, avoid the overhead of invalidation by not using it at all. TL2, for example, does not use invalidation, yet through its space and time optimizations, is able to perform efficient orec-based validation [4]. RingSTM and JudoSTM, on the other hand, do not use invalidation nor do they use orecs. RingSTM uses a `ring` structure to efficiently perform eager validation only against those transactions on the `ring` in which it is necessary [31], while JudoSTM uses a *value-based conflict detection* data structure to perform validation with reduced atomic instruction overhead [20]. In all three cases, invalidation is avoided entirely.

Although invalidation was proposed as early as 2003 [13, 14], implementation overhead has kept its use limited. In this paper, Section 3 explores the concurrency potential of an efficient fully invalidating TM by presenting an asymptotic analysis of version-based validation in contrast with commit-time invalidation. The section shows that invalidation provides opportunities over validation that can increase transaction throughput, making it a superior conflict detection strategy. Section 4 then shows how we efficiently implement the data structures needed for full invalidation within InvalSTM, and Section 5 evaluates InvalSTM against TL2, a state-of-the-art validating STM.

3. The Promise of Full Invalidation

In this section we demonstrate that full invalidation offers numerous benefits over validation. We show that *any* fully invalidating TM can perform opacity and conflict detection more efficiently than a validating one. In the case of contending workloads, we illustrate how full invalidation can increase transaction throughput over validation. In addition, we demonstrate that a fully invalidating TM that uses search time optimized read and write sets can perform conflict detection for memory-intensive transactions in notably fewer operations than what is needed for the most efficient validation techniques.

Full Invalidation. For a TM to be *fully invalidating*, each transaction must resolve its conflicts before it commits. A *resolved conflict* is one that is eliminated by aborting or stalling one or more transactions to preserve serializability (as described in [22]). Therefore, when a transaction begins its commit phase in a fully invalidating TM, the TM need only check in-flight transactions for conflicts against the committing transaction. This is because fully invalidating TMs require conflicts to be resolved before transactions commit, ensuring the only unresolved conflicts are those with uncommitted, in-flight transactions. While there are numerous ways to build a fully invalidating TM, commit-time invalidation may be the least computationally expensive way. As such, for the remainder of this paper when we speak of full invalidation we mean a system that at least, and most often only, uses commit-time invalidation.

3.1 Conflict Detection and Opacity

TMs perform conflict detection to determine which transactions can commit. However, as noted by Guerraoui and Kapalka, conflict detection alone is insufficient. TMs must also ensure *opacity*, a correctness criterion that requires each transactional read return a value that is consistent with its execution, and that doomed transactions be aborted before a subsequent transactional read or write returns from its call [12]. Guerraoui and Kapalka show that even an isolated, uncommitted transaction can cause adverse program side-effects if a single transactional read is inconsistent.

To demonstrate this, consider a program invariant where variables X and Y are always one discrete value apart (+/-1). If transaction T_1 reads X as value 2, the operation X/Y will be defined, because Y will be either 3 or 1. However, if transaction T_2 performs $X = 1$, $Y = 0$ and commits after T_1 reads X but before it reads Y, the program invariant will be violated. If T_1 is permitted to read Y, the X/Y operation will result in a divide by zero exception. An opaque TM avoids this by verifying all of a transaction's reads are consistent before opening a new item for reading or writing. In this case, when T_1 opens Y for reading, the TM would identify T_1's view of X is inconsistent and force it to abort before returning the value of Y. Thus to be correct, in addition to detecting and resolving conflicts, TMs must also be opaque.

3.1.1 Validation

Many TMs ensure opacity and perform conflict detection using the same technique; they perform *incremental validation* (DSTM, RSTM, SXM, and TL2), a process in which each element in a transaction's read set is checked for consistency each time a new element is opened for reading [30]. Incremental validation performs $O(N)$ operations N times, where N is the number of elements in the transaction's read set, resulting in $O(N^2)$ operations [17]. Every validation a transaction performs prior to commit-time is an opacity check (and still results in $O(N^2)$ worst-case time because

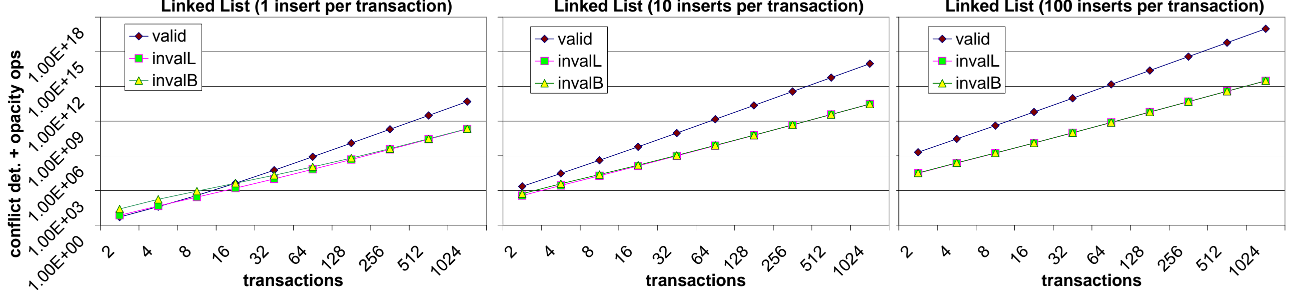

Figure 4. Conflict Detection and Opacity Overhead of Linked List for Validation and Invalidation.

$N * (N - 1) \in O(N^2)$). These eager operations are not intended to commit the transaction. Instead they ensure opacity by avoiding abnormal program behavior that would ensue from accessing stale values. The final validation operation, performed precisely once during a transaction's commit phase, is a conflict detection operation performed specifically to ensure a transaction can commit.

Given a series of M non-conflicting, committing transactions, the below equation represents the opacity and conflict detection operations sufficient for incremental validation. The variable r_i is the ith committing transaction's read set size. [3]

$$o_v(M) = \sum_{i=1}^{M} \sum_{j=1}^{r_i} j$$

The inner sum represents the number of opacity operations performed for each transaction up to and including its commit-time conflict detection operation. The outer sum includes the opacity and conflict detection operations for all M committing transactions.

3.1.2 Invalidation

Invalidation uses two different techniques for opacity and conflict detection. Invalidating TMs perform opacity by checking a boolean $valid$ flag, and perform conflict detection by identifying conflicts between a committing transaction and all in-flight transactions. The invalidation processes for opacity and conflict detection are explained below.

Opacity Checks. Each transaction has a $valid$ flag that is initially true. It is set to false when the TM decides to abort the transaction to resolve a conflict (Figure 3). When a transaction opens a new element for reading, the TM checks the transaction's $valid$ flag – an $O(1)$ time operation. If it is false, the transaction is aborted before the new element's memory is returned. If the $valid$ flag is true, the transaction continues to execute normally. Because opacity is checked incrementally, it takes $O(N)$ time to complete per transaction, an improvement over incremental validation's $O(N^2)$ time. However, to properly set each transaction's $valid$ flag, the TM performs commit-time invalidation for each transaction. The commit-time invalidation algorithm behaves differently based on the type of transaction.

Conflict Detection for Writers. A *writer*, a transaction that writes to at least one memory location and reads any number of locations, must resolve its conflicts before it commits. These conflicts are limited to in-flight transactions that access memory (via read or write) that the writer has modified. When the conflicts are found the CM can perform any one of the following actions: (1) set the committing transaction's $valid = false$ and abort it, (2) set the conflicting in-flight transactions' $valid = false$ so they will abort or (3) stall the committing transaction until the conflicting transactions commit or abort. A key characteristic when using commit-time invalidation is that a committing transaction's conflicts are identified (and resolved) prior to committing; this characteristic is paramount to unlocking concurrency.

Given a series of M non-conflicting, committing transactions, the below equation represents the opacity and conflict detection operations sufficient for full invalidation. The variables r_i and w_i are the ith committing transaction's read and write set size. F_i is the number of in-flight transactions at the time of the ith committing transaction. r_j and w_j are the jth in-flight transaction's read and write set sizes. s_{rj} and s_{wj} are the search time complexity associated with the jth transaction's read and write algorithms.

$$o_i(M) = \sum_{i=1}^{M} \left(r_i + \sum_{j=1}^{F_i} w_i(s_{rj}(r_j) + s_{wj}(w_j)) \right)$$

The inner sum performs conflict detection for the ith committing transaction against all in-flight transactions (F_i). The number of operations sufficient to identify conflicts with each transaction is based on the jth transaction's read and write set algorithm's worst-case search time when holding r_j and w_j number of elements, represented by $s_{rj}(r_j)$ and $s_{wj}(w_j)$. The ith transaction compares its write set for overlaps with reads and writes of in-flight transactions, resulting in $s(r_j) + s(w_j)$. Finally, w_i is multiplied by the sum of $s(r_j)$ and $s(w_j)$, because each search operation is performed w_i times, the number of elements in the committing transaction's write set. [4]

The outer sum performs the incremental opacity checks for all M committing transactions. The checks are performed with a single boolean comparison and are performed each time the ith transaction opens a new element for reading, represented by r_i.

Conflict Detection for Read-Only Transactions. Notice that read-only transactions have $w_i = 0$, which reduces the above equation to $o_i(M) = \sum_{i=1}^{M} r_i$. In other words, read-only transactions perform *zero* conflict detection operations. Furthermore, unlike prior optimizations, invalidating read-only transactions do not need to be identified as read-only before they execute to achieve this benefit. [5] Therefore, dynamically detected read-only transactions in any fully invalidating TM are guaranteed to perform zero conflict detection operations. This is a notable benefit because many trans-

[3] A TM using a global clock does not need to perform incremental validation if the global clock has the same value as when the transaction first began. This indicates no transaction has committed since it began [4]. However, the clock must still be read at each opacity check.

[4] This is not true for algorithms that perform set intersection in constant time, such as Bloom filters. In such cases, w_i becomes some constant C.

[5] TL2 has a space complexity optimization for read-only transactions, yet, it requires the transaction be known as read-only before it is executed [4].

actions that are statically read-write may be dynamically read-only much of the time.

3.2 An Analysis of Opacity and Conflict Detection Efficiency

To demonstrate the efficiency of opacity and conflict detection using full invalidation, consider a scenario in which N transactions are appending to a linked list in which only one transaction can commit. For this scenario, each time a transaction commits all other in-flight transactions must abort and restart, regardless of the conflict detection strategy. However, as illustrated in Figure 4 full invalidation has a lower opacity cost and uses fewer conflict detection operations as transactions grow in size, resulting in a more efficient TM. Figure 4 shows the number of transactions vs. the number of operations required for conflict detection, as per the earlier equations.

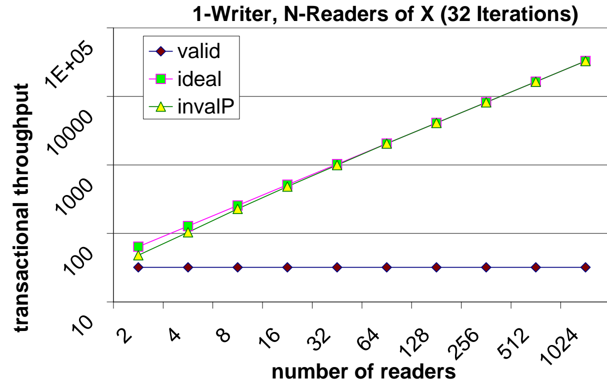

Figure 5. Transaction Throughput for 1-Writer / N-Readers of Single Variable.

Our model assumes all transactions reach their commit phase before conflicts are identified, requiring that each transaction perform incremental opacity (via version-based validation or invalidation's *valid* flag). Figure 4 demonstrates the performance of (1) incremental validation (valid), (2) commit-time invalidation using a logarithmic-time search, i.e., $s_{rj}(r_j)$ and $s_{wj}(w_j)$ at $O(logN)$ (invalL), and (3) commit-time invalidation using a constant-time search, i.e. $s_{rj}(r_j)$ and $s_{wj}(w_j)$ at $O(1)$ (invalB). While the commit-time invalidation logarithmic- and constant-time search displays little theoretical operational difference in Figure 4, their practical executions are critically different (see Section 4). In Figure 4, as the number of elements inserted per transaction grows (from 1, to 10, to 100), the performance delta between validation and invalidation widens each time by an order of magnitude (from 10^2, to 10^3, to 10^4 operational difference), illustrating our prior point that incremental validation's overhead worsens as transactions access more memory highlighting commit-time invalidation's efficiency for memory-intensive transactions.

3.3 An Analysis of Transaction Throughput

Finally, we turn our attention to highly contending, highly concurrent workloads, where concurrency can be exploited but only if the CM makes informed decisions about which transactions to commit and which to abort.

We analyze the scenario shown in Figure 1, where one transaction writes to X followed by N transactions reading X. We assume lazy write acquisition and that the writer reaches its commit phase first, followed by the N readers. Using this model, Figure 5 displays the amount of transaction throughput (y-axis) achieved as N increases (x-axis) using (1) version-based validation (valid), (2) ideal throughput or unfair commit-time invalidation (ideal), and (3) priority-based commit-time invalidation (invalP) [10, 29]. The priority-based CM policy ensures transactions will not starve, while simultaneously promoting a high degree of transactional concurrency. It behaves in the following way. Each new transaction begins with a priority of 1. Each time a transaction is aborted, its priority is incremented by 1. Once it commits, the transaction's priority is reset to 1. For a transaction to abort other transactions, its priority must be greater than or equal to the sum of all the transactions it is attempting to abort.

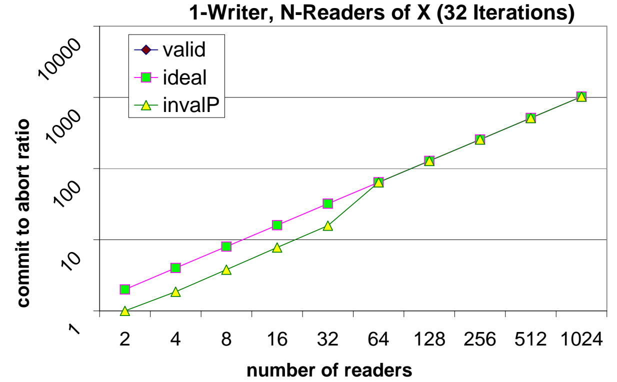

Figure 6. Commit to Abort Ratio for 1-Writer / N-Readers of Single Variable.

Figure 5 shows 32 iterations of 1-writer / N-readers, where priority-based invalidation eventually achieves $\approx 10^3\times$ greater transaction throughput than version-based validation (for 1024 readers). [6] Figure 6 shows the commit to abort ratio of the Figure 5. Validation's commit to abort ratio ranges from 0.5 (2 transactions) to 0.000977 (1024 transactions), while priority-based invalidation ranges from 1 (a $2\times$ difference) to 1024 (a $10^6\times$ difference).

4. InvalSTM: A Fully Invalidating STM

In this section, we explain the design of InvalSTM, our fully invalidating STM. InvalSTM uses commit-time invalidation for all conflicts (i.e., W-W, W-R, and R-W). InvalSTM also uses lazy write acquisition because it provides the CM with one-to-many conflicts at commit-time. These one-to-many conflicts are necessary for the CM to make informed decisions that can increase transaction throughput. *Eager write acquisition*, which exclusively acquires write locations as the transaction executes them, sends eager one-to-one conflicts to the CM. These one-to-one conflicts prevent the CM from seeing the entire view of conflicts and, because these conflicts are eager, force the CM to make speculative decisions that limit transaction throughput.

4.1 A Design Overview

As explained in Section 2, maintaining visible read sets through orecs can be expensive. InvalSTM addresses this in the same way JudoSTM [20], NOrec [3], and RingSTM [31] do, by avoiding orecs altogether. Instead, InvalSTM stores read and write sets inside a transaction object. In addition, because lock-based STMs have emerged with strong performance – DracoSTM [9], Ennal's STM [6], RingSTM [31], and TL2 [4] – InvalSTM uses mutual exclusion locks as its core synchronization type.

For InvalSTM to perform commit-time invalidation its transactions must be prevented from adding new memory elements to

[6] Our tests of up to 32,768 iterations show that priority-based invalidation remains $\approx 10^3\times$ faster than version-based validation, although with each increased N^2 iteration invalidation grows ($\approx 1.5\times$) faster.

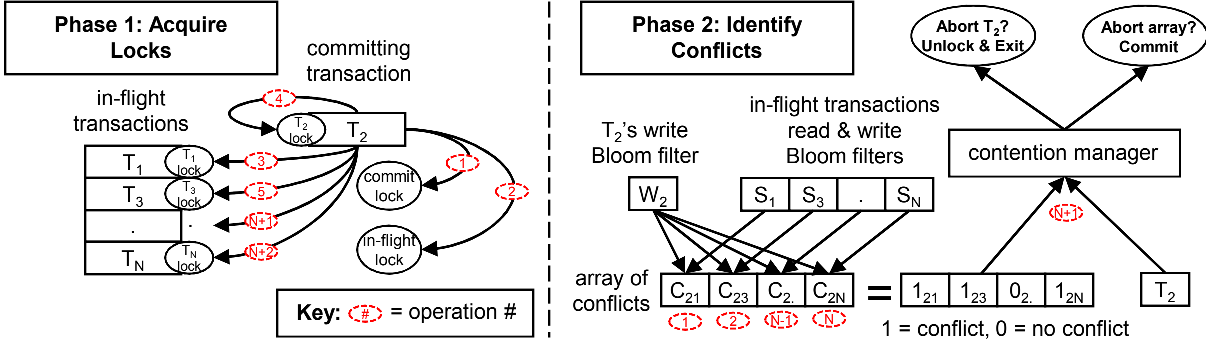

Figure 7. An Example of InvalSTM's Commit-Time Invalidation Process.

their read and write sets while a transaction commits. Without this restriction, a conflicting memory element may be added to an in-flight transaction's read or write sets after it has been found to be free of conflicts. To prevent this unwanted behavior, InvalSTM associates a lock with each transaction. Before performing commit-time invalidation, InvalSTM acquires the transactional locks of all in-flight transactions to ensure the invalidation phase will be performed without extraneous modification to the in-flight transaction's read and write sets.

While this addresses the above concern, it creates a new problem: a serialization point is created from the beginning of a transaction's commit phase until its end. To minimize the negative impact of this serialization point, InvalSTM compresses read and write sets within Bloom filters, which align with caches for fast access and constant time ($O(1)$) set intersection.

The remainder of this section discusses, in detail, the main design points summarized above. Listed below is a summary of the unique optimizations that emerge from this design.

- Full invalidation is supported, gaining all the benefits highlighted in Section 3, including boosted concurrency from informed CM decisions, zero conflict detection operations for read-only transactions and efficient conflict detection for memory-intensive transactions.
- Read sets can be stored in imprecise, compressed, and contiguous storage that reduce the time and space complexity to perform invalidation while also reducing cache line eviction rates.
- Per-memory locking (orecs) is no longer necessary. Instead, locks are associated with each transaction which can drastically reduce atomic (fenced) operations when transactions are memory-intensive [20, 31].
- Visible read sets using per-transaction storage require zero operations to cleanup, a significant savings when compared to visible read sets using per-memory (orec) storage.

4.2 A Lock-Based STM

In addition to a lock per transaction, InvalSTM uses two global locks: a commit and in-flight lock. The commit lock limits the commit phase to a single transaction. The in-flight lock is used to limit modification of the in-flight transaction list to a single thread.

Before performing commit-time invalidation, the commit lock is acquired to prevent two or more transactions from concurrently committing. InvalSTM disallows this behavior because concurrently committing transactions are not guaranteed they will have the execution time to invalidate all other concurrently committing transactions unless committing transactions are prevented from exiting the commit phase until all other committing transactions have completed their invalidation. While concurrently committing transactions would increase concurrent work, it introduces livelock scenarios as new transactions enter the commit phase, creating a perpetual cycle of invalidation. This livelock cycle can decrease or even halt throughput, so InvalSTM prohibits it by limiting the commit phase to a single transaction.

The in-flight lock is acquired before commit-time invalidation is performed so new transactions cannot be put in-flight. This is needed for two reasons. First, it creates a sequential locking order based on the transactions that are currently in-flight (as seen in Figure7). Second, it prevents a livelock that could occur as invalidated transactions are removed and placed back in-flight, requiring cyclic invalidation.

While these additional locks complicate the design, their absence would reduce concurrency in the following ways. First, if only the commit lock was used, all transactions would be required to obtain it when adding elements to their read and write sets. By using the commit and transactional locks, transactions can concurrently add elements to their read and write sets, so long as no transaction is committing. Second, if the in-flight lock was removed and instead the commit lock was used to add or remove transactions from the in-flight set, transactions could not simultaneously begin the commit phase and modify the in-flight transaction list. While it is true that committing transactions generally do need to obtain the in-flight lock, they do not always need it immediately. Transactions whose $valid = false$ can perform nearly all of their cleanup code prior to requiring the in-flight lock. In addition, read-only transactions only require the commit lock momentarily to identify that they are in fact read-only and to check that they are $valid$. Once checked, these transactions release the commit lock, but retain the in-flight lock to remove themselves from the list.

4.3 Serialized Commit

A downside of InvalSTM's locking design is that it creates a serialization point during a transaction's commit phase. This serialization point limits commits to one transaction at a time and prevents in-flight transactions from adding new elements to their read and write sets while a transaction commits. To minimize the negative impact of this serialization point, read and write sets are stored in Bloom filters which speed up the invalidation process by performing set intersection in constant worst-case time [1]. Below is the modified operational overhead equation ($o_{ib}(M)$ for M transactions) from Section3.1.2 when read and write sets use Bloom filters to perform full invalidation.

$$o_{ib}(M) = \sum_{i=1}^{M} r_i + (2kw * (F_i))$$

Commit-time invalidation is handled by $\sum_{i=1}^{M} 2kw * (F_i)$ where w is the number of words per bit vector, k is the number of bit vectors per Bloom filter, and F_i are the in-flight transactions

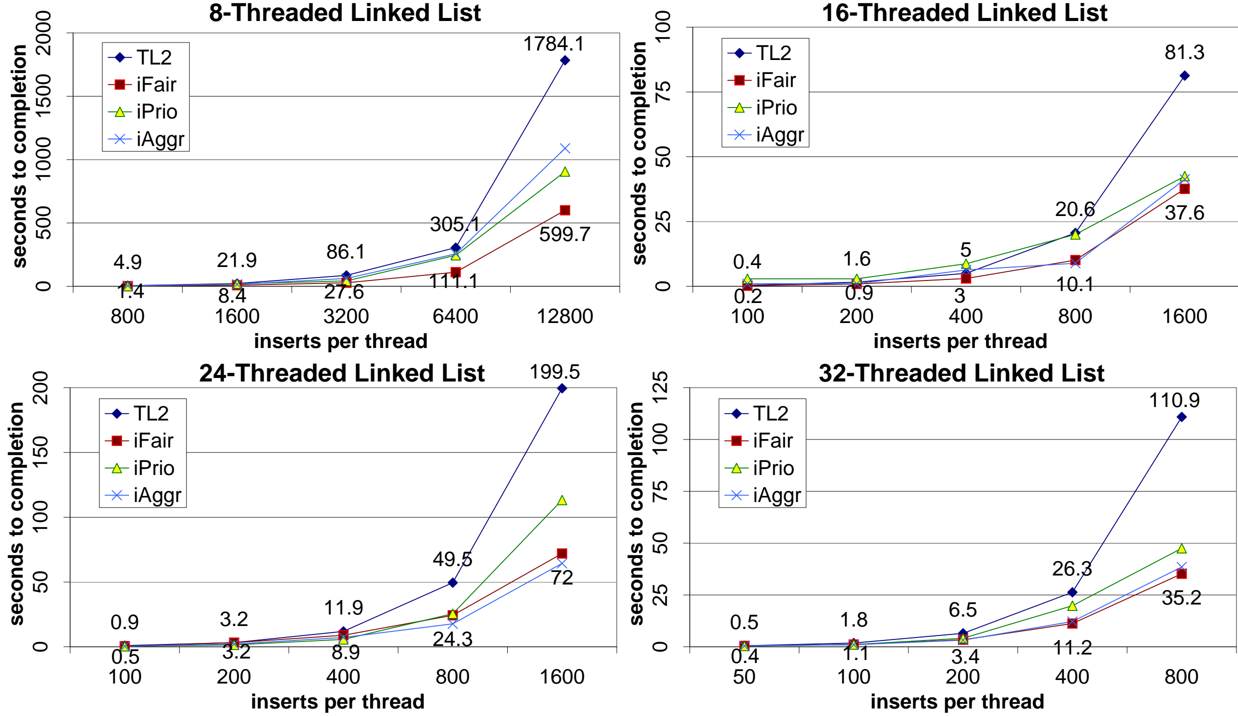

Figure 8. Linked List Benchmarks.

at the time the ith transaction is committing. The opacity checks, which are performed throughout the transaction's lifetime, add r_i (number of elements in the transaction's read set) to each summation. By using Bloom filters, the original search operations required for a single transaction is reduced from w_i to 1 (Section 3), because the conflicts between one transaction and another are found in a single set intersection. kw represents the original equation's search time of $s(w_j)$ and $s(r_j)$. Since kw must be performed twice (once per read and write set) the result is $2kw$. Because these operations must be done for each in-flight transaction, we multiply $2kw$ by F_i.

For each transaction, InvalSTM currently uses a fixed 2^{16} bits per bit vector and two bit vectors per Bloom filter. Although we experimented with a wide variety of Bloom filter configurations, our early experiments indicate the current size performs the best overall for our tested benchmarks. We expect to extend our research in this area as we analyze more benchmarks.

4.4 Transaction Implementation

In InvalSTM each transaction object contains its own read and write sets. Read sets store memory locations, while write sets store memory locations plus a copy that is used to buffer transactional writes for lazy write acquisition. The memory locations for read and write sets are stored in separate Bloom filters. This is done so different types of conflicts can be handled by the CM in different ways. For example, if a committing, writer transaction has only W-R conflicts, the CM can choose to stall the writer until the reader transactions commit. If read and write sets were not separated, the CM would only be able to resolve conflicts via abort.

Because write sets store written data along with memory locations, each transaction contains an additional map that associates written data with its memory location. This data structure is necessary in addition to the Bloom filter used for write sets, because lazy write acquisition must commit memory in such a way that false positives are not possible [1]. Otherwise the TM could update incorrect memory locations.

Opacity Checks. In a fully invalidating TM, ensuring a transaction has no conflicts is inexpensive (an $O(1)$ operation). Therefore, InvalSTM performs opacity checks on all transaction calls, not just the ones in which it is necessary. This adds some overhead, yet, we have found it improves system performance because it can identify doomed transactions early.

Reading and Writing Transactional Memory. When a transaction accesses a memory element for reading or writing, the STM performs a read-only lookup to see if the transaction has already accessed the element. This lookup requires no locking, since the operation is not changing the read or write sets. If the lookup is successful, the appropriate value is returned. If not, the transaction's lock is acquired, the memory address (and value if necessary) is inserted into the correct set, and the transaction's lock is released.

Committing. Figure 7 provides a high-level view of the commit-time invalidation process. For brevity, some details are omitted from the diagram. Those include the priority elevation for aborted transactions, the removal of aborted transactions from the in-flight set to reduce in-flight lock contention, the CM's usage of transaction size (read + write sets) as a discriminator for the abort protocol, and the short-circuited logic used for read-only transactions.

The commit-time invalidation process, shown in Figure 7, begins with Phase I where the commit and in-flight locks are acquired. This ensures no other transaction can commit or be started while a transaction is committing. Next, the committing and in-flight transactions' associated locks are acquired in a sequential order to avoid deadlock. Phase II then identifies the conflicts the committing transaction has with the in-flight transactions. If conflicts exist, the CM is sent the batch of conflicts and allows it to make the decision on which transactions are aborted or stalled.

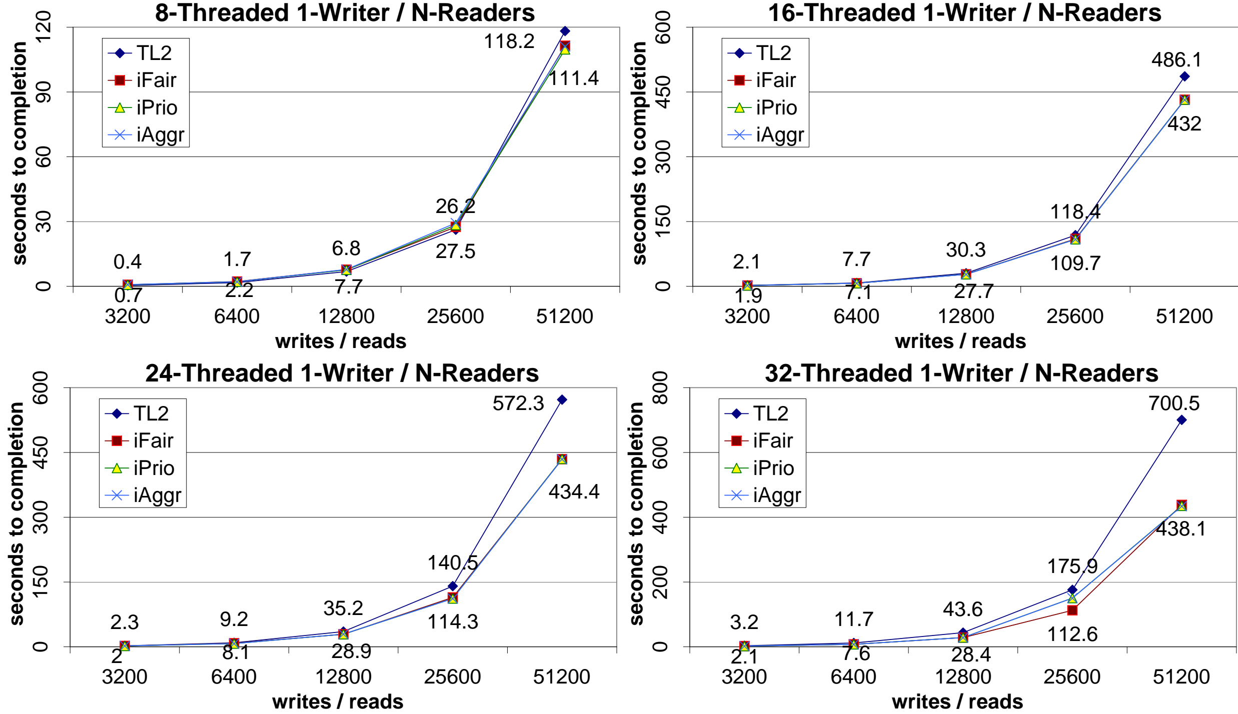

Figure 9. 1-Writer, N-Reader Benchmarks.

An important detail of InvalSTM's design is that in-flight transactions can make forward progress during the entire commit-time invalidation process. The two exceptions are (1) transactions cannot concurrently commit while another transaction is already committing and (2) they cannot add new memory elements to their read and write sets. Those limitations aside, our anecdotal experiments have shown that the forward progress of transactions while another transaction is committing significantly increases overall throughput for workloads that access a shared memory element multiple times (e.g., a head or sentinel node, a global counter, etc.).

False Positives Preventing Forward Progress. As Bloom filters can emit false positives, there is a chance these false positives will prevent forward progress. To avoid this scenario, we use a runtime threshold R that, once exceeded by our abort to commit ratio, switches our TM from using Bloom filters to using red-black trees for read and write sets, ensuring false positives are avoided. After some programmable period of time T, our system reverts back to using Bloom filters for read and write sets. In our experiments, however, this threshold is never reached.

5. Experimental Results

In this section we present the experimental results of InvalSTM, using commit-time invalidation, and TL2, the state-of-the-art validating STM. The benchmarks were run on a 1.0 GHz Sun Fire T2000 supporting 32 concurrent hardware threads with 32 GB RAM. The TL2 implementation is from RSTM.v4, University of Rochester's STM library collection. For all the graphs in this section, the y-axis shows the total execution time in seconds (lower is always better). The x-axis represents the workload executed rather than the usual threads, since, as shown in Section3, invalidation performs more efficiently than validation as transactions access more memory. Since the number of threads is constant per graph, four graphs are used per benchmark each with a different thread count and/or workload configuration.

5.1 Contention Manager Variants

We tested three CM variants with our benchmarks: iFair (invalidation fair), iPrio (invalidation prioritized) and iAggr (invalidation aggressive). iAggr ensures the first transaction to enter the commit phase commits. It demonstrates how commit-time invalidation performs when it does not use conflict information to make informed decisions. In other words, iAggr captures the conflict detection operational difference between invalidation and validation.

iPrio associates a priority with each transaction. A transaction's priority is raised each time it aborts and is reset each time it commits. A transaction can commit if it has the highest priority of all conflicting in-flight transactions. A transaction can also commit if it has the largest read set size of all in-flight transactions or its read and write set size is larger than the average read and write set size of all conflicting transactions plus their cumulative priority.

iFair associates a priority with each transaction and raises and resets the transaction's priority in the same manner as iPrio. Unlike iPrio, a transaction can commit if its read and write set size is greater than a weighted average of the in-flight transaction's read size and their priority. A transaction can also commit if its read and write set size is greater than any of the in-flight transaction's read set size. In addition, if an in-flight transaction's read set size is $10^2\times$ greater than the committing transaction's read set size and its priority is $2^3\times$ greater, iFair will abort the committing transaction in favor of the higher priority, larger in-flight transaction.

Of the three CM variants, iFair performs the best overall. While iAggr and iPrio each perform well under certain conditions, iFair consistently performs as well or better and, in some cases, outperforms TL2 by more than $3\times$ (Figure 8, 32-Threaded Linked List). Based on the performance improvement as the concurrency widens from 8 to 32 threads, it seems that one can speculate that commit-time invalidation's performance will only improve over TL2 as the number of concurrent transactions grow.

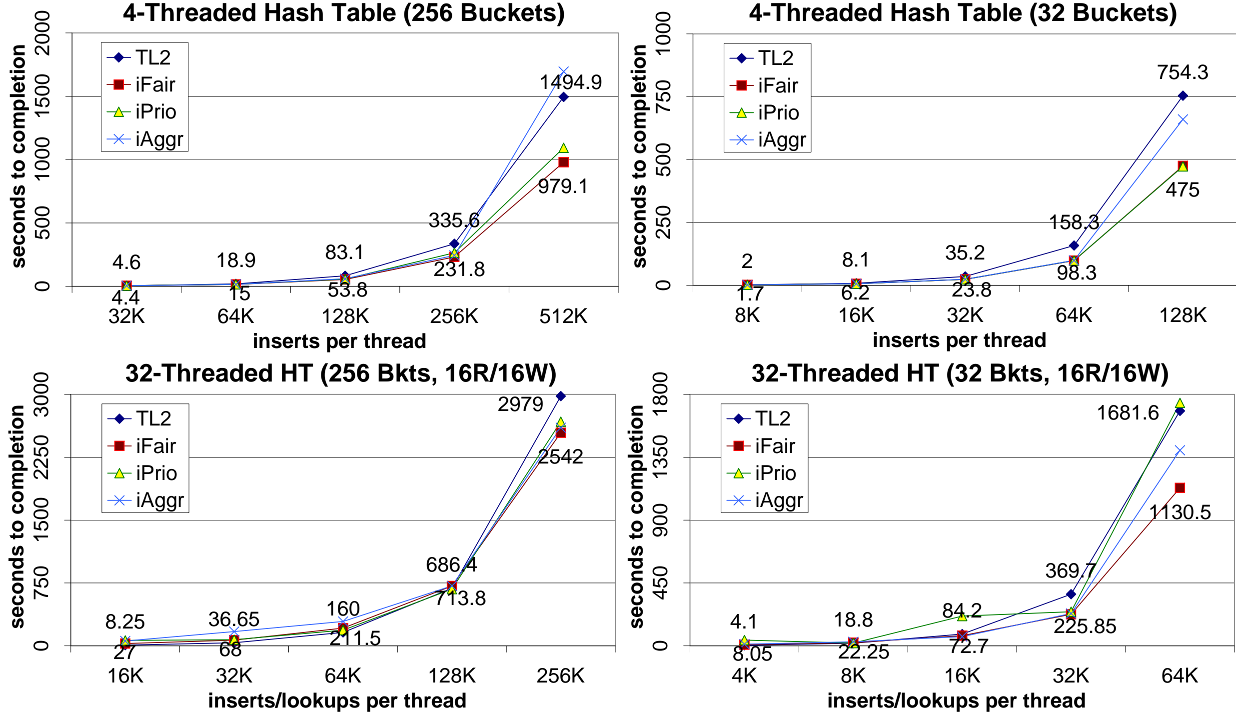

Figure 10. Hash Table Benchmarks.

5.2 Linked List

Our linked list benchmarks are shown in Figure 8. Each linked list benchmark populated a single linked list with N concurrently executing threads. Each thread inserted the same number of elements (i.e., T_1 inserts 0-99, T_2 inserts 100-199, etc.) and the insert operation was a transaction. iFair performed most consistently, especially in the 8-threaded benchmark where its CM policy drives it to outperform the other CM policies by $\approx 2\times$.

In the linked list benchmarks, InvalSTM outperforms TL2 from $\approx 2\times$ to $\approx 3\times$. At nearly all data points, as the workload increases InvalSTM improves its efficiency over TL2. For the final data point in the 32-threaded benchmark, InvalSTM's iFair is $3.15\times$ faster than TL2. It is important to note that the larger threaded benchmarks perform less work than the smaller threaded benchmarks (e.g., the 32-threaded workload inserts $<=$ 800 nodes per transaction, while the 8-threaded one inserts $<=$ 12,800 nodes). However, the performance difference for InvalSTM and TL2 is roughly the same for all threaded executions. This suggests that if equivalent work was executed for the 32-threaded benchmarks, the performance difference would significantly favor InvalSTM.

5.3 1-Writer / N-Readers

For highly contending but also highly concurrent workloads commit-time invalidation performs well. This is demonstrated in the 1-writer / N-reader benchmark shown in Figure 9. The 1-writer / N-reader benchmark was implemented using a linked list where the writer performs a fixed number of appends and each reader performs an iterative lookup. Both the append and lookup are transactions. While the performance difference between InvalSTM's CM strategies and TL2 for 8 and 16 threaded workloads is small, the difference between the 24 and 32 threaded workloads is notable. The 32 threaded benchmark shows iFair outperform TL2 by $\approx 1.6\times$. The reason for this is straightforward: lower threaded workloads emit fewer aborts because contention on the data is minimal. As readers are added the contention increases as do the number of aborts. This creates a scenario where early notification of doomed transactions, a low overhead benefit of invalidation, is critical in improving performance.

Read-only Transactions. While the 1-writer / N-reader performance difference favors InvalSTM by only $\approx 1.6\times$, this margin is notable because TL2 has a space optimization for read-only transactions (though transactions must be flagged as read-only prior to executing). Commit-time invalidation has a time optimization that can defer the discovery of read-only transactions until commit-time. However, to be fair to TL2, we only leverage read-only optimizations for statically tagged read-only transactions. Since N of the transactions are read-only (where N = 7, 15, 23, and 31), both systems heavily exploit their read-only transaction optimizations for this benchmark. Although TL2's read-only optimizations are impressive, commit-time invalidation's read-only optimizations seem to have more impact on performance for this scenario.

5.4 Hash Tables

The hash table experiments are shown in Figure 10 and are implemented using N-bucketed lists (N = 32 and 256). The hash function is a modulo operation on the number of buckets. Each benchmark used a single hash table which was concurrently populated by N number of threads and used the same insert conditions as the linked list example. The performance improvement of InvalSTM over TL2 are $1.48\times$ and $1.58\times$ for the 256 and 32 bucketed hash tables, respectively. For the 32-threaded 16 reader / 16 writer benchmarks, InvalSTM is faster than TL2 by $1.17\times$ for the 256 bucketed hash table and $1.48\times$ for the 32 bucketed one.

Although these performance improvements for InvalSTM are lower than the linked list experiments, they are meaningful because a bucketed hash table is generally considered a concurrent data structure as operations are distributed across numerous, simultaneously accessible buckets [16]. However, after the buckets reach

a certain threshold of size, transactions begin to contend since appending elements to densely populated buckets requires more transactional execution time. Notice that even for the 32-threaded hash table benchmarks, InvalSTM outperforms TL2 by up to $\approx 50\%$.

6. Conclusion

This paper presented an efficient implementation of commit-time invalidation, a strategy where transactions resolve all of their conflicts with in-flight transactions before they commit. We provided an early example where substantial concurrency could be exploited by invalidation, but would be missed by validation. We then explained the opportunities invalidation provides over validation such as, more complete information sent to the CM enabling it to make better decisions to increase concurrency, faster conflict detection time for memory-intensive transactions, and transactional opacity checking in $O(N)$ time instead of incremental validation's $O(N^2)$ time. We then explained InvalSTM, a new TM design that performs commit-time invalidation using transaction-based read and write sets, time and space efficient Bloom filters for read and write sets, and an orec-free design that uses transaction locks rather than memory locks to reduce atomic primitive instruction overhead. The experimental results section compared InvalSTM's commit-time invalidation to TL2's state-of-the-art validation. We presented three CM strategies for InvalSTM which outperformed TL2, in some cases by upwards of $3\times$.

Acknowledgments

We thank the University of Rochester for their TL2 implementation in RSTM. We thank the anonymous reviewers for their constructive feedback; we have tried our best to integrate their insightful suggestions. We also thank Michael F. Spear for his early support of our initial design. We are also grateful to Michael for the discussions we had with him that helped us arrive at our final design. Lastly, we thank J Smart and Jim Pastoor of Raytheon Company for their ongoing support.

References

[1] B. H. Bloom. Space/time trade-offs in hash coding with allowable errors. In *Communications of the ACM*, 1970.

[2] C. Cascaval, C. Blundell, M. Michael, H. W. Cain, P. Wu, S. Chiras, and S. Chatterjee. Software transactional memory: Why is it only a research toy? In *ACM Queue*, 2008.

[3] L. Dalessandro, M. F. Spear, and M. L. Scott. NOrec: Streamlining STM by abolishing ownership records. In *Proceedings of the Symposium on Principles and Practice of Parallel Programming*, 2010.

[4] D. Dice, O. Shalev, and N. Shavit. Transactional locking II. In *Proceedings of the Symposium on Distributed Computing*, 2006.

[5] D. Dice and N. Shavit. Understanding tradeoffs in software transactional memory. In *Proceedings of the Symposium on Code Generation and Optimization*, 2007.

[6] R. Ennals. Software transactional memory should not be obstruction free. In *Intel Research Cambridge Tech Report*, 2006.

[7] K. Fraser and T. Harris. Concurrent programming without locks. In *ACM Transactions on Computer Systems*, 2007.

[8] H. Garcia-Molina and G. Wiederhold. Read-only transactions in a distributed database. In *ACM Transactions on Database Systems*, 1982.

[9] J. E. Gottschlich and D. A. Connors. DracoSTM: A practical C++ approach to software transactional memory. In *Proceedings of the Symposium on Library-Centric Software Design*, 2007.

[10] J. E. Gottschlich and D. A. Connors. Extending contention managers for user-defined priority-based transactions. In *Proceedings of the Workshop on Exploiting Parallelism with Transactional Memory and other Hardware Assisted Methods*, 2008.

[11] R. Guerraoui, M. Herlihy, and B. Pochon. Toward a theory of transactional contention managers. In *Proceedings of the Symposium on Principles of Distributed Computing*, 2005.

[12] R. Guerraoui and M. Kapalka. On the correctness of transactional memory. In *Proceedings of the Symposium on Principles and Practice of Parallel Programming*, 2008.

[13] T. Harris and K. Fraser. Language support for lightweight transactions. In *Proceedings of the Conference on Object Oriented Programming, Systems, Languages and Applications*, 2003.

[14] M. Herlihy, V. Luchangco, M. Moir, and I. William N. Scherer. Software transactional memory for dynamic-sized data structures. In *Proceedings of the Symposium on Principles of Distributed Computing*, 2003.

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

[16] M. Herlihy and N. Shavit. *The Art of Multiprocessor Programming*. Elsevier, Inc., 2008.

[17] J. R. Larus and R. Rajwar. *Transactional Memory*. Morgan & Claypool, 2006.

[18] V. J. Marathe and M. Moir. Toward high performance nonblocking software transactional memory. In *Proceedings of the Symposium on Principles and Practice of Parallel Programming*, 2008.

[19] K. E. Moore, J. Bobba, M. J. Moravan, M. D. Hill, and D. A. Wood. LogTM: log-based transactional memory. In *Proceedings of the Conference on High-Performance Computer Architecture*, 2006.

[20] M. Olszewski, J. Cutler, and J. G. Steffan. JudoSTM: A dynamic binary-rewriting approach to software transactional memory. In *Proceedings of the Conference on Parallel Architecture and Compilation Techniques*, 2007.

[21] J. K. Ousterhout, H. Da Costa, D. Harrison, J. A. Kunze, M. Kupfer, and J. G. Thompson. A trace-driven analysis of the UNIX 4.2BSD file system. Technical report, Berkeley, CA, 1985.

[22] C. H. Papadimitriou. Serializability of concurrent data base updates. Technical report, Cambridge, MA., 1979.

[23] H. E. Ramadan, C. J. Rossbach, and E. Witchel. Dependence-aware transactional memory for increased concurrency. In *Proceedings of the Symposium on Microarchitecture*, 2008.

[24] H. E. Ramadan, I. Roy, M. Herlihy, and E. Witchel. Committing conflicting transactions in an STM. In *Proceedings of the Symposium on Principles and Practice of Parallel Programming*, 2009.

[25] B. Saha, A.-R. Adl-Tabatabai, R. L. Hudson, C. C. Minh, and B. Hertzberg. McRT-STM: a high performance software transactional memory system for a multi-core runtime. In *Proceedings of the Symposium on Principles and Practice of Parallel Programming*, 2006.

[26] W. N. Scherer and M. L. Scott. Advanced contention management for dynamic software transactional memory. In *Proceedings of the Symposium on Principles of Distributed Computing*, 2005.

[27] M. L. Scott. Sequential specification of transactional memory semantics. In *Proceedings of the Workshop on Transactional Computing*, 2006.

[28] N. Shavit and D. Touitou. Software transactional memory. In *Proceedings of the Principles of Distributed Computing*, 1995.

[29] M. F. Spear, L. Dalessandro, V. Marathe, and M. L. Scott. A comprehensive strategy for contention management in software transactional memory. In *Proceedings of the Symposium on Principles and Practice of Parallel Programming*, 2009.

[30] M. F. Spear, V. J. Marathe,W. N. Scherer III, and M. L. Scott. Conflict detection and validation strategies for software transactional memory. In *Proceedings of the Symposium on Distributed Computing*, 2006.

[31] M. F. Spear, M. M. Michael, and C. von Praun. RingSTM: scalable transactions with a single atomic instruction. In *Proceedings of the Symposium on Parallelism in Algorithms and Architectures*, 2008.

[32] W. Vogels. File system usage in Windows NT 4.0. In *Proceedings of the Symposium on Operating Systems Principles*, 1999.