Remote Invalidation: Optimizing the Critical Path of Memory Transactions

Ahmed Hassan, Roberto Palmieri, Binoy Ravindran
Electrical and Computer Engineering Department
Virginia Tech
Blacksburg, Virginia, USA
hassan84@vt.edu, robertop@vt.edu, binoy@vt.edu

***Abstract*—Software Transactional Memory (STM) systems are increasingly emerging as a promising alternative to traditional locking algorithms for implementing generic concurrent applications. To achieve generality, STM systems incur overheads to the normal sequential execution path, including those due to spin locking, validation (or invalidation), and commit/abort routines. We propose a new STM algorithm called *Remote Invalidation* (or RInval) that reduces these overheads and improves STM performance. RInval's main idea is to execute commit and invalidation routines on remote server threads that run on dedicated cores, and use cache-aligned communication between application's transactional threads and the server routines. By remote execution of commit and invalidation routines and cache-aligned communication, RInval reduces the overhead of spin locking and cache misses on shared locks. By running commit and invalidation on separate cores, they become independent of each other, increasing commit concurrency. We implemented RInval in the Rochester STM framework. Our experimental studies on micro-benchmarks and the STAMP benchmark reveal that RInval outperforms InvalSTM, the corresponding non-remote invalidation algorithm, by as much as an order of magnitude. Additionally, RInval obtains competitive performance to validation-based STM algorithms such as NOrec, yielding up to 2x performance improvement.**

***Keywords*-Software Transactional Memory; Remote Invalidation; Synchronization;**

I. Introduction

Software Transactional Memory (STM) [1] is a concurrency control paradigm that replaces traditional lock-based synchronization with a complete framework for synchronizing concurrent threads. STM hides the synchronization complexity from programmers, which vastly increases programmability, enabling more complex and generic concurrent applications, while ensuring correctness properties such as *opacity* [2]. To achieve this, STM algorithms perform a number of additional operations during transactional execution, besides the necessary memory reads and writes, such as in-memory logging of transactional metadata, lock acquisition, validation, commit, and abort. Each of these operations incur an overhead, often at different levels of intensity, and taken together, they degrade STM performance [1].

An STM transaction can be viewed as being composed of a set of operations that must execute sequentially – i.e., the transaction's *critical path*. Reducing any STM operation's overhead on the critical path, without violating correctness properties, will reduce the critical path's execution time, and thereby significantly enhance overall STM performance.

One of the major overheads on the transaction critical path is that of the memory-level locking mechanism used. Unlike lock-free and fine-grained locking algorithms, which optimize locking mechanisms according to the application logic (e.g., lazy and lock-free data structures [3]), STM algorithms may acquire more redundant memory-level locks, because they are not aware of the application logic. A clear example is linked-list traversal. In a lazy linked-list [4], traversal is done without any locking, because the logic guarantees that consistency is not violated. However, most STM algorithms are unaware of this logic and hence have to monitor all traversed nodes.

The overhead of locking has been previously mitigated in the literature, largely by using different granularity levels for locking. Fine-grained locking algorithms [5], [6] reduce false conflicts, potentially enabling greater scalability, but at the expense of significantly decreased programmability and sometimes higher cost. In contrast, coarse-grained locking algorithms [7], [8] are easy to implement and debug, have minimal meta-data to manage, but may suffer from false conflicts (so they may serialize code blocks that can safely run in parallel), resulting in poor performance. Moreover, if locks are lazily acquired at commit time, scalability will not be significantly affected by coarse-grained locking. NOrec [7] is an example of an efficient coarse-grained locking algorithm, which gathers reduced meta-data, uses the minimum number of locks (i.e., one global lock), and yields competitive scalability with respect to fine-grained locking algorithms such as TL2 [5].

Another issue in locking is the mechanism of spin locking, which overloads the hardware interconnect with cache-coherency traffic and CAS operations, causing high cache misses, and degrading lock performance. As argued by [9], hardware overheads such as CAS operations and cache misses are critical bottlenecks to scalability in multi-core infrastructures. Although STM algorithms proposed in the literature cover a wide range of locking granularity alternatives, they do not focus on the locking mechanism (which is one of our focuses).

Another overhead is that of the validation/commit routines. In most STM algorithms, validation should not only

guarantee correct encounter-time reads, but also must ensure opacity [2]. To support opacity, transactions must ensure that their read-sets remain consistent during execution so that all committed and aborted transactions always observe a consistent state of the accessed objects. Incremental validation is a typical way to guarantee opacity and is used in several well-known STM algorithms [5], [7]. However, incremental validation suffers from quadratic time complexity. Before each new read, transactions must validate that all previous reads are still consistent. This drawback has led to the development of commit-time invalidation [10], which obligates commit executors to detect conflicting transactions and invalidate them. Accordingly, transactions will only check if they have been invalidated by an earlier transaction before each new read. This way, commit-time invalidation reduces the overhead of validation to linear-time (in terms of read-set sizes), because each transaction checks only one variable per read (which is the invalidation flag). In memory-intensive workloads (i.e., workloads with massive reads and writes), this reduction in validation time improves performance. However, the overhead of invalidation is now added to commit time, which significantly affects a number of workloads. Our work focuses on balancing this overhead, instead of overloading commit routines with all of the work.

In this paper, we propose *Remote Invalidation* (or RInval), an STM algorithm that significantly optimizes the transaction critical path. As previously discussed, one of the main problems of commit-time invalidation is that it shifts the overhead from the validation routine to the commit routine. As both routines are on the critical path of transactions, this can significantly reduce the performance improvement due to invalidation. RInval separates commit and invalidation into two independent routines, and remotely executes them in parallel, on dedicated hardware threads. Thus, we gain the same improvement as commit-time invalidation, without adding more overhead to commit. With coarse-grained locking, this improvement becomes more significant, because commit executors are a bottleneck in the system and must execute as fast as possible to reduce the blocking time of the other transactions.

RInval also parallelizes the invalidation process. The process of traversing all running transactions and invalidating them can be parallelized more, because invalidating each single transaction is independent from invalidating another transaction. Using remote servers[1] to invalidate transactions allows us to exploit this parallelism by dedicating more than one server to invalidate conflicting transactions.

Specifically, in RInval, when client transactions reach the commit phase, they send a commit request to the commit-server, and then they keep looping on a local *status* variable until they receive a response from the server. A transaction's write-set is passed as a parameter of the commit request to enable the server to execute the commit operation on behalf of the clients. The commit-server then passes this write-set to invalidating servers and then starts publishing the write-set in the main memory. In parallel, invalidating servers check all running transactions to detect conflicting ones and signal them as *invalidated*.

Instead of competing on spin locks, in RInval, communication between the commit-server and the clients, as well as communication between the commit-server and the invalidating servers, are done through a cache-aligned requests array. This approach therefore reduces cache misses (which are often due to spinning on locks), and reduces the number of CAS operations during commit[2]. Additionally, dedicating CPU cores for servers reduces the probability of interleaving the execution of different tasks on those cores due to OS scheduling.

Some similar work exists in the literature [11]–[13], but none of them has been specifically tailored for STM commit and invalidation. In particular, RCL [13] uses a similar idea of executing lock-based critical sections in remote threads. RCL performs better than well-known locking mechanisms [14], [15] due to the same reasons that we mentioned (i.e., reducing cache misses, CAS operations, and blocking). Applying the same idea to STM is appealing, because it makes use of the increasing number of cores in current multi-core architectures, and at the same time, allows more complex applications than lock-based approaches.

Gottschlich *et. al*'s commit-time invalidation algorithm [10] is effective only in memory-intensive cases, where the reduction of validation time from quadratic- to linear-time is significant. In other cases, the overhead added to commit can significantly affect performance. RInval almost keeps the commit routine as is, which improves performance in both memory-intensive and non-memory-intensive workloads. Our implementation and experimental studies confirm this claim: RInval is better than commit-time invalidation in almost all cases, sometimes by an order of magnitude, and better than NOrec [7] in a significant number of non-memory-intensive cases by as much as 2x faster.

The paper makes the following contributions:

- We analyze the parameters that affect the critical path of STM transaction execution (Section III), and summarize the earlier attempts in the literature to reduce their effects (Section II). We propose *remote invalidation*, a new STM algorithm, which alleviates the overhead on the critical path (Section IV).
- RInval optimizes both validation and commit. The execution-time complexity of validation is reduced to linear-time from quadratic-time. The commit routine only publishes write-sets, while invalidation is dele-

[1]We will call hardware threads that execute either commit or invalidation remotely as *servers*, and application threads that execute transactions as *clients*.

[2]Spinning on locks and increased usage of CAS operations can seriously hamper performance [3], especially in multicore architectures.

gated to other dedicated servers, running in parallel, thereby improving performance.
- RInval optimizes locking overhead. All spin locks and CAS operations are replaced with optimized cache-aligned communication.
- Through experimentation, we show that RInval outperforms past validation-based (NOrec) and invalidation-based (InvalSTM) algorithms, on both micro-benchmarks and the STAMP benchmark [16], yielding performance improvement up to 2x faster than NOrec and an order of magnitude faster than InvalSTM.

RInval is publicly available as an open-source project at http://www.ssrg.ece.vt.edu/rinval/.

II. Background: Commit-Time Invalidation

The invalidation approach has been presented and investigated in earlier works [10], [17], [18]. Among these approaches, Gottschlich *et. al* proposed commit-time invalidation, (or InvalSTM) [10], an invalidation algorithm that completely replaces version-based validation without violating opacity.

The basic idea of InvalSTM is to let the committing transaction invalidate all active transactions that conflict with it before it executes the commit routine. More complex implementation involves the contention manager deciding if the conflicting transactions should abort, or the committing transaction itself should wait and/or abort, according to how many transactions will be doomed if the committing transaction proceeds.

RInval is based on the basic idea of commit-time invalidation – i.e., conflicts will always be solved by aborting conflicting transactions rather than the committing transaction. Although this may result in reducing the efficiency of the contention manager, it opens the door for more significant improvements, such as parallelizing commit with invalidation, as we will show later in Section IV.

Algorithm 1 shows how InvalSTM works. When a transaction T_i attempts to commit, it tries to atomically increment a global timestamp, and keeps spinning until the timestamp is successfully incremented (line 13). If *timestamp* is odd, this means that some transaction is executing its commit. When T_i attempts to read a new memory location, it takes a snapshot of the timestamp, reads the location, and then validates that timestamp does not change while it reads (lines 3 – 6). Then, T_i checks a *status* flag to test if it has been invalidated by another transaction in an earlier step (lines 7 – 10). This flag is only changed by the commit executor if it finds that T_i's read-set conflicts with the commit executor's write-set. Conflict detection is done by comparing the write bloom filters [19] of the committing transaction with the read bloom filters of all in-flight transactions (line 18). Bloom filters are used because they are accessed in constant time, independent of the read-set size. However, they increase the probability of false conflicts because bloom filters are only compact bit-wise representations of the memory.

Algorithm 1 Commit-time Invalidation

```
 1: procedure READ(address)
 2:     while true do
 3:         x1 = timestamp
 4:         val = read(address)
 5:         if x1 is odd and x1 = timestamp then
 6:             OK
 7:         if OK and tx_status = ALIVE then
 8:             return val
 9:         else if tx_status = ABORTED then
10:             TM-ABORT
11: end procedure

12: procedure COMMIT
13:     while timestamp is odd or ¬CAS(timestamp, timestamp + 1) do
14:         LOOP
15:     if tx_status = ABORTED then
16:         TM-ABORT
17:     for All in-flight transactions t do
18:         if me.write_bf intersects t.read_bf then
19:             t.tx_status = ABORTED
20:     WriteInMemory(req.Tx.writes)
21:     timestamp++
22: end procedure
```

This procedure replaces incremental validation, which is used in a number of STM algorithms [5], [7]. In incremental validation, the entire read-set has to be validated before reading any new memory location. Thus, the overhead of read-validation is a quadratic function of the read-set size in the case of incremental validation, and a linear function in the case of commit-time invalidation. This reduction in validation time enhances the performance, especially for memory-intensive workloads. It is worth noting that both incremental validation and commit-time invalidation have been shown to guarantee the same correctness property, which is opacity [1].

One of the main disadvantages of commit-time invalidation is that it burdens the commit routine with the mechanism of invalidation. In a number of cases, this overhead may offset the performance gain due to reduced validation time. Moreover, InvalSTM uses a conservative coarse-grained locking mechanism, which of course makes its implementation easier, but at the expense of reduced commit concurrency (i.e., only one commit routine is executed at a time). The coarse-grained locking mechanism increases the potential of commit "over validation", because the commit executor will block all other transactions that attempt to read or commit. Other committing transactions will therefore be blocked, spinning on the global lock and waiting until they acquire it. Transactions that attempt to read will also be blocked because they cannot perform validation while another transaction is executing its commit routine (to guarantee opacity).

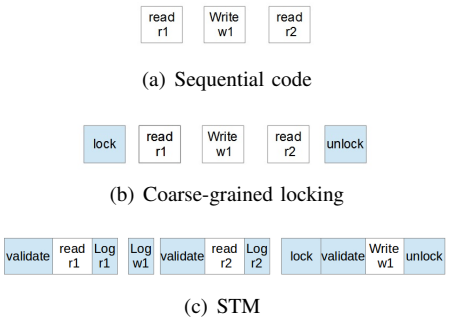
InvalSTM is not the only work that targets reducing the cost of incremental validation. DSTM [18] is an example of partial invalidation which eagerly detects and resolves write-

write conflicts. STM^2 [11] proposes executing validation in parallel with the main flow of transactions. However, it does not decrease the time complexity of incremental validation (like InvalSTM). Moreover, it does not guarantee opacity and needs a sand-boxing mechanism to be consistent [20].

III. Transaction Critical Path

As described in Section II, InvalSTM serializes commit and invalidation in the same commit routine, which significantly affects invalidation in a number of cases and degrades performance (we show this later in this section). Motivated by this observation, we study the overheads that affect the critical path of transaction execution to understand how to balance the overheads and reduce their effect on the critical path.

First, we define the *critical path* of a transaction as the sequence of steps that the transaction takes (including both shared memory and meta-data accesses) to complete its execution. Figure 1 shows this path in STM, and compares it with that in sequential execution and coarse-grained locking.



(a) Sequential code

(b) Coarse-grained locking

(c) STM

Figure 1. Critical path of execution for: (a) sequential, (b) lock-based, and (c) STM-based code

In Figure 1(a), the sequential code contains only shared-memory reads and writes without any overhead. Coarse-grained locking, in Figure 1(b), adds only the overhead of acquiring and releasing a global lock at the beginning and at the end of execution, respectively. However, coarse-grained locking does not scale and ends up with a performance similar to sequential code. It is important to note that fine-grained locking and lock-free synchronization have been proposed in the literature to overcome coarse-grained locking's scalability limitation [3]. However, these synchronization techniques must be custom-designed for a given application situation. In contrast, STM is a general purpose framework that is completely transparent to application logic. In application-specific approaches, the critical path cannot be easily identified because it depends on the logic of the application at hand.

Figure 1(c) shows how STM algorithms[3] add significant overheads on the critical path in order to combine the two benefits of i) being as generic as possible and ii) exploiting as much concurrency as possible. We can classify these overheads as follows:

Logging. Each read and write operation must be logged in local (memory) structures, usually called read-sets and write-sets. This overhead cannot be avoided, but can be minimized by efficient implementation of the structures [3].

Locking. In most STM algorithms, locks are lazily acquired at commit time to increase concurrency. However, for an STM algorithm to be generic, the memory-level locks that transactions acquire cannot be as optimal as those for application-specific implementations. Thus, a trade-off exists between coarse-grained and fine-grained STM algorithms, which has been thoroughly studied in the literature. Although fine-grained algorithms such as TL2 [5] reduce false conflicts and increase concurrency, they are not as simple as coarse-grained algorithms such as NOrec [7] and InvalSTM [10]. Since locks are only acquired at commit time in both [7] and [10], the overhead of lock granularity is minimized, and other considerations such as minimizing the meta-data (saved locally in transactions) become as important as performance. NOrec, for example, uses only one global lock at commit time, which minimizes the meta-data that must be managed. This meta-data minimization opens the door for more enhancements, such as easy integration with hardware transactions [21].

Another important overhead is the locking mechanism. Most STM frameworks use spin locks and CAS operations to synchronize memory accesses. One of the main sources of STM's performance degradation is that all transactions compete and spin on the same shared lock when they are trying to read, write, or commit. This is because spinning on a shared object increases cache misses, and also because CASes are costly operations [9]. In the case of global locking, like in NOrec and InvalSTM, this overhead becomes more significant because spinning in these algorithms is only on one global lock.

Validation. As mentioned before, the validation overhead becomes significant when higher levels of correctness guarantees are required (e.g., opacity). Most STM algorithms use either incremental validation or commit-time invalidation to guarantee opacity. In the case of invalidation, the time complexity is reduced, but with an additional overhead on commit, as we discuss in the next point.

Commit. Commit routines handle a number of issues in addition to publishing write-sets on shared memory. One of these issues is lock acquisition, which, in most STM algorithms, is delayed until commit. Also, most STM algorithms require commit-time validation after lock acquisition to ensure that nothing happened when the locks were acquired. In case of commit-time invalidation, the entire invalidation overhead is added to the commit routines. This means that a committing transaction has to traverse all active transactions to detect which of them is conflicting. As a

[3]Here, we sketch the critical path of NOrec [7]. However, the same idea can be applied to any other STM algorithm.

consequence, the lock holding time is increased. Moreover, if the committing transaction is blocked for any reason (e.g., due to OS scheduling), all other transactions must wait. The probability of such blocking increases if the time of holding the lock increases. Therefore, optimizing the commit routines has a significant impact on overall performance.

Abort. If there is a conflict between two transactions, one of them has to abort. Transaction abort is a significant overhead on the transaction's critical path. The contention manager is usually responsible for decreasing the abort overhead by selecting the best candidate transaction to abort. The greater the information that is given to the contention manager from the transactions, the greater the effectiveness on reducing the abort overhead. However, involving the contention manager to make complex decisions adds more overhead to the transaction critical path.

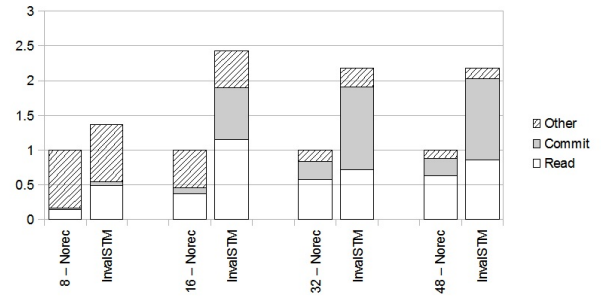


Figure 2. Percentage of validation, commit, and other (non-transactional) overheads on a red-black tree. The y-axis is the normalized (to NOrec) execution time

Figure 2 shows how the trade-off between invalidation and commit affects the performance in a red-black tree benchmark for different numbers of threads (8, 16, 32, and 48). Here, transactions are represented by three main blocks: read (including validation), commit (including lock acquisition, and also invalidation in the case of InvalSTM), and other overhead. The last overhead is mainly the non-transactional processing. Although some transactional work is also included in the later block, such as beginning a new transaction and logging writes, all of these overheads are negligible compared to validation, commit, and non-transactional overheads. Figure 2 shows the percentage of these blocks in both NOrec and InvalSTM (normalized to NOrec).

The figure provides several key insights. When the number of threads increases, the percentage of non-transactional work decreases, which means that the overhead of contention starts to dominate and becomes the most important to mitigate. It is clear also from the figure that InvalSTM adds more overhead on commit so that the percentage of execution time consumed by the commit routine is higher than NOrec. Moreover, this degradation in commit performance affects read operations as well, because readers have to wait for any running commit to finish execution.

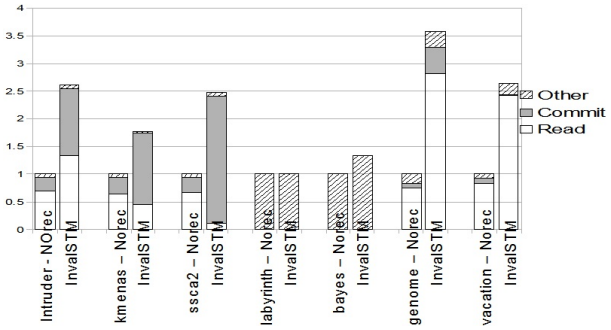


Figure 3. Percentage of validation, commit, and other (non-transactional) overheads on STAMP benchmark. The y-axis is the normalized (to NOrec) execution time

The same conclusion is given in the STAMP benchmark[4]. In Figure 3, the percentage of commit in *intruder*, *kmeans*, and *ssca2*, is higher in InvalSTM than NOrec, leading to the same performance degradation as red-black tree. In *genome* and *vacation*, degradation in InvalSTM read performance is much higher than before. This is because these workloads are biased to generate more read operations than writes. When a committing transaction invalidates many read transactions, all of these aborted transactions will retry executing all of their reads again. Thus, in these read-intensive benchmarks, abort is a dominating overhead. In *labyrinth* and *bayes*, almost all of the work is non-transactional, which implies that using any STM algorithm will result in almost the same performance.

Based on this analysis, it is clear that each overhead cannot be completely avoided. Different STM algorithms differ on how they control these overheads. It is also clear that some overheads contradict each other, such as validation and commit overheads. The goal in such cases should be finding the best trade-off between them. This is why each STM algorithm is more effective in some specific workloads than others.

We design and implement RInval to minimize the effect of most of the previously mentioned overheads. Basically, we alleviate the effect of *i)* locking overhead, *ii)* the tradeoff between validation and commit overheads, and *iii)* abort overhead. The overhead of meta-data logging usually cannot be avoided in lazy algorithms. For locking, we select coarse-grained locking to obtain the advantage of minimal meta-data, and to minimize the synchronization overhead. We also use the remote core locking mechanism [13], instead of spin locking, to reduce the overhead of locks. Validation and commit are improved by using invalidation outside, and

[4]We excluded yada applications of STAMP as it evidenced errors (segmentation faults) when we tested it on RSTM.

in parallel with, the main commit routine. Finally, we use a simple contention manager to reduce the abort overhead.

IV. Remote Invalidation

As described in Section III, *Remote Invalidation* reduces the overhead of the transaction critical path. To simplify the presentation, we describe the idea incrementally, by presenting three versions of RInval[5]. In the first version, called RInval-V1, we show how spin locks are replaced by the more efficient remote core locks. Then, in RInval-V2, we show how commit and invalidation are parallelized. Finally, in RInval-V3, we further optimize the algorithm by allowing the commit-server to start a new commit routine before invalidation-servers finish their work.

A. Version 1: Managing the locking overhead

Spin locks are not usually the best locking mechanism to synchronize critical sections. More efficient locking mechanisms have been proposed in the literature such as MCS [15], Flat Combining [14], and Remote Core Locking (or RCL) [13]. However, not all of these lock algorithms can easily replace spin locking in STM algorithms, because most STM algorithms use sequence locks (not just spin locks) by adding versions to each lock. The versions are used in many STM algorithms to ensure that transactions always see a consistent snapshot of the memory. Sequence locks cannot be directly converted to more complex locks such as MCS or Flat Combining.

RCL, one of the most recent locking mechanisms, targets the same goal that we discussed in Section III, which is to reduce CAS operations and cache misses. RCL dedicates servers to execute critical sections on behalf of application threads. Applications send commit requests to servers using cache-aligned arrays to minimize cache misses. RCL's performance has been shown to be better than MCS and Flat Combining. However, in generic lock-based workloads, RCL's usage is complicated, as it needs some mechanism (e.g., compile-time code re-engineering) to define critical sections and how they are overlapped or nested. Such complex mechanisms may nullify some of RCL's benefits by adding new CAS operations or by causing new cache misses.

RCL can be adopted to replace internal spin locks in STM frameworks, which is more appealing than replacing coarse-grained and generic locks (with RCL) in lock-based applications as described in [13]. Unlike generic critical sections in lock-based applications, in STM, commit routines are well defined, and can easily be executed remotely at dedicated server cores (without the need to re-engineer legacy applications).

[5]We only present the basic idea in the pseudo code given in this section. The source code provides the full implementation details.

RInval-V1 uses this idea: commit routines are executed remotely to replace spin locks. Figure 4 shows how RInval-V1 works. When a client reaches a commit phase, it sends a commit request to the commit-server by modifying a local *request_state* variable to be PENDING. The client then keeps spinning on *request_state* until it is changed by the server to be either ABORTED or COMMITTED. This way, each transaction spins on its own variable instead of competing with other transactions on a shared lock.

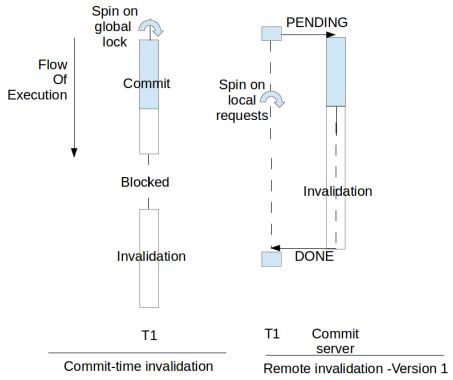


Figure 4. Flow of commit execution in both InvalSTM and RInval-V1

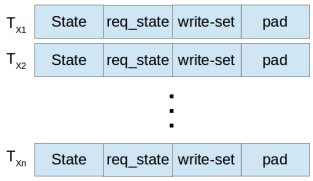


Figure 5. Cache-aligned requests array

Figure 5 shows the structure of the cache-aligned requests array. In addition to *request_state*, the commit-server only needs to know two values: *tx_status*, which is used to check if the transaction has been invalidated in an earlier step, and *write_set*, which is used for publishing writes on shared memory and for invalidation. In addition, padding bits are added to cache-align the request.

Remote execution of commit has three main advantages. First, it removes all CAS operations and replaces them with cache-aligned requests. Second, cache misses due to spin locking are minimized, because each client spins on its own variable. Third, commit-server blocking is minimized, because it is executed on a dedicated core (this is evident in Figure 4). In InvalSTM, if the commit executor is blocked (which is more likely to be blocked than the commit-server), all other transactions must wait until it resumes its execution and releases the lock.

Since we use a coarse-grained approach, only one commit-server is needed. Adding more than one commit-server will cause several overheads: *i)* the design will

become more complex, *ii)* more cores have to be dedicated for servers, *iii)* more CAS operations must be added to synchronize the servers, and *iv)* cache misses may occur among servers. Since we minimize the work done by the commit-server, the overhead of serializing commit on one server is expected to be less than these overheads.

Algorithm 2 Remote Invalidation - Version 1

```
1:  procedure CLIENT COMMIT
2:      if read_only then
3:          ...
4:      else
5:          if tx_status = INVALIDATED then
6:              TxAbort()
7:          request_state = PENDING
8:          loop while request_state ∉ (COMMITTED, ABORTED)
9:  end procedure

10: procedure COMMIT-SERVER LOOP
11:     while true do
12:         for i ← 1, num_transactions do
13:             req ← requests_array[i]
14:             if req.request_state = PENDING then
15:                 if req.tx_status = INVALIDATED then
16:                     req.request_state = ABORTED
17:                 else
18:                     timestamp++
19:                     for All in-flight transactions t do
20:                         if me.write_bf intersects t.read_bf then
21:                             t.tx_status = INVALIDATED
22:                     WriteInMemory(req.writes)
23:                     timestamp++
24:                     req.request_state = COMMITTED
25: end procedure
```

Algorithm 2 shows the pseudo code of RInval-V1[6]. This version modifies the InvalSTM algorithm shown in Algorithm 1.

The read procedure is the same in both InvalSTM and RInval, because we only shift execution of commit from the application thread to the commit-server. In the commit procedure, if the transaction is read-only, the commit routine consists of only clearing the local variables. In write transactions, the client transaction checks whether it was invalidated by an earlier commit routine (line 5). If validation succeeds, the client changes its state to PENDING (line 7). The client then loops until the commit-server handles its commit request and changes the state to either COMMITTED or ABORTED (line 8). The client will either commit or rollback according to the reply.

On the server side, the commit-server keeps looping on client requests until it reaches a PENDING request (line 14). The server then checks the client's *request_state* to see if the client has been invalidated (line 15). This check has to be repeated at the server, because some commit routines may take place after sending the commit request and before the commit-server handles that request. If validation fails, the server changes the state to ABORTED and continues searching for another request. If validation succeeds, it starts the commit operation (like InvalSTM). At this point, there are two main differences between InvalSTM and RInval-V1. First, incrementing the timestamp does not use the CAS operation (line 18), because only the main server changes the timestamp. Second, the server checks *request_state* before increasing the timestamp (line 15), and not after it, like in InvalSTM, which saves the overhead of increasing the shared timestamp for a doomed transaction. Since only the commit-server can invalidate transactions, there is no need to check *request_state* again after increasing the timestamp.

[6]We assume that instructions are executed in the same order as shown, i.e., sequential consistency is assumed. We ensure this in our C/C++ implementation by using memory fence instructions when necessary (to prevent out-of-order execution), and by using volatile variables when necessary (to prevent compiler re-ordering).

B. Version 2: Managing the tradeoff between validation and commit

In RInval-V1, we minimized the overhead of locking on the critical path of transactions. However, invalidation is still executed in the same routine of commit (in serial order with commit itself). RInval-V2 solves this problem by dedicating more servers to execute invalidation in parallel with commit. Unlike the commit-server, there can be more than one invalidation-server, because their procedures are independent. Each invalidation-server is responsible for invalidating a subset of the running transactions. The only data that needs to be transferred from the commit-server to an invalidation-server is the client's write-set. Figure 6 shows RInval-V2 with one commit-server and two invalidation-servers. When the commit-server selects a new commit request, it sends the write bloom filter of that request to the invalidation-servers, and then starts execution. When the commit-server finishes, it waits for the response from all invalidation-servers, and then proceeds to search for the new commit request.

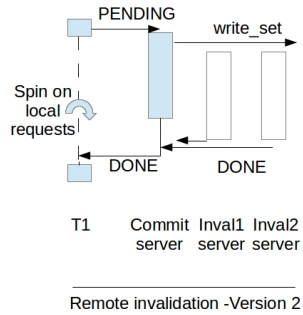


Figure 6. Flow of commit execution in RInval-V2

Selecting the number of invalidation-servers involves a trade-off. According to Amdahl's law, concurrency decreases as the number of parallel executions increases. At some point, adding more invalidation-servers may not have a noticeable impact on performance. At the same time, increasing the number of invalidation-servers requires dedicating more cores for servers, and adds the overhead of servers communication. In our experiments, we found that, on a 64-core

machine, it is sufficient to use 4 to 8 invalidation-servers to achieve the maximum performance.

Adding invalidation-servers does not change the fact that no CAS operations are needed. It also ensures that all communication messages (either between the commit-server and the clients, or between the commit-server and the invalidation-servers) are sent/received using cache-aligned requests. Thus, RInval-V2 inherits the benefits of optimized locking and parallelizing commit-invalidation routines.

Algorithm 3 Remote Invalidation - Version 2

```
1:  procedure COMMIT-SERVER LOOP
2:      while true do
3:          for i ← 1, num_transactions do
4:              req ← requests_array[i]
5:              if req.request_state = PENDING then
6:                  for i ← 1, num_invalidators do
7:                      while timestamp > inval_timestamp do
8:                          LOOP
9:                  if req.tx_status = INVALIDATED then
10:                     req.request_state = ABORTED
11:                 else
12:                     commit_bf ← req.write_bf
13:                     timestamp++
14:                     WriteInMemory(req.writes)
15:                     timestamp++
16:                     req.request_state = COMMITTED
17: end procedure

18: procedure INVALIDATION-SERVER LOOP
19:     while true do
20:         if timestamp > inval_timestamp then
21:             for All in-flight transactions t in my set do
22:                 if commit_bf intersects t.read_bf then
23:                     t.tx_status = INVALIDATED
24:             inval_timestamp += 2
25: end procedure

26: procedure CLIENT READ
27:     ...
28:     if x1 = timestamp and timestamp = my_inval_timestamp then
29:         ...
30: end procedure
```

Algorithm 3 shows RInval-V2's pseudo code. The client's commit procedure is exactly the same as in RInval-V1, so we skip it for brevity. Each invalidation-server has its local timestamp, which must be synchronized with the commit-server. The commit-server checks that the timestamp of all invalidation-servers is greater than or equal to the global timestamp (line 7). It then copies the write bloom filter of the request into a shared $commit\_bf$ variable to be accessed by the invalidation-servers (line 12).

The remaining part of RInval-V2 is the same as in RInval-V1, except that the commit-server does not make any invalidation. If an invalidation-server finds that its local timestamp has become less than the global timestamp (line 20), it means that the commit-server has started handling a new commit request. Thus, it checks a subset of the running transactions (which are evenly assigned to servers) to invalidate them if necessary (lines 21-23). Finally, it increments its local timestamp by 2 to catch up with the commit-server's timestamp (line 24). It is worth noting that the invalidation-server's timestamp may be greater than the commit-server's global timestamp, depending upon who will finish first.

The client validation is different from RInval-V1. The clients have to check if their invalidation-servers' timestamps are up-to-date (line 28). The invalidation-servers' timestamps are always increased by 2. This means that when they are equal to the global timestamp, it is guaranteed that the commit-server is idle (because its timestamp is even).

C. Version 3: Accelerating Commit

In RInval-V2, commit and invalidation are efficiently executed in parallel. However, in order to be able to select a new commit request, the commit-server must wait for all invalidation-servers to finish their execution. This part is optimized in RInval-V3. Basically, if there is a new commit request whose invalidation-server has finished its work, then the commit-server can safely execute its commit routine without waiting for the completion of the other invalidation-servers. RInval-V3 exploits this idea, and thereby allows the commit-server to be *n* steps ahead of the invalidation-servers (excluding the invalidation-server of the new commit request).

Algorithm 4 Remote Invalidation - Version 3

```
1:  procedure COMMIT-SERVER LOOP
2:      if req.request_state = PENDING and req.inval_timestamp ≥
        timestamp then
3:          ...
4:      ...
5:      while timestamp > inval_timestamp + num_steps_ahead do
6:          LOOP
7:      ...
8:      commit_bf[my_index + +] ← req.write_bf
9:      ...
10: end procedure

11: procedure INVALIDATION-SERVER LOOP
12:     ...
13:     if commit_bf[my_index + +] intersects t.read_bf then
14:         ...
15: end procedure
```

Algorithm 4 shows how RInval-V3 makes few modifications to RInval-V2 to achieve its goal. In line 2, the commit-server has to select an up-to-date request, by checking that the timestamp of the request's invalidation-server equals the global timestamp. The commit-server can start accessing this request as early as when it is *n* steps ahead of the other invalidation-servers (line 5). All bloom filters of the requests that do not finish invalidation are saved in an array (instead of one variable as in RInval-V2). This array is accessed by each server using a local index (lines 8 and 13). This index is changed after each operation to keep pointing to the correct bloom filter.

It is worth noting that, in the normal case, all invalidation-servers will finish almost in the same time, as the clients are evenly assigned to the invalidation-servers, and the invalidation process takes almost constant time (because

it uses bloom filters). However, RInval-V3 is more robust against the special cases in which one invalidation-server may be delayed for some reason (e.g., OS scheduling, paging delay). In these cases, RInval-V3 allows the commit-server to proceed with the other transactions whose invalidation-servers are not blocked.

D. Other Overheads

In the three versions of RInval, we discussed how we alleviate the overhead of spin locking, validation, and commit. As discussed in Section III, there are two more overheads that affect the critical path of transactions. The first is logging, which cannot be avoided as we use a lazy approach. This issue is not just limited to our algorithm. Storing reads and writes in local read-sets and write-sets, respectively, is necessary for validating transaction consistency. The second overhead is due to abort. Unlike InvalSTM, we prevent the contention manager from aborting or delaying the committing transaction even if it conflicts with many running transactions. This is because of two reasons. First, it enables finishing the invalidation as early as possible (in parallel with the commit routine), which makes the abort/retry procedure faster. Second, we shorten the time needed to complete the contention manager's work, which by itself is an overhead added to the servers' overhead, especially for the common case (in which writers invalidate readers).

E. Correctness and Features

RInval guarantees opacity in the same way other coarse-grained locking algorithms do, such as NOrec [7] and InvalSTM [10]. Both reads and writes are guaranteed to be consistent because of lazy commit and global commit-time locking. Before each new read, the transaction check that *i)* it has not been invalidated in an earlier step, and *ii)* no other transaction is currently executing its commit phase. Writes are delayed to commit time, which are then serialized on commit-servers. The only special case is that of RInval-V3, which allows the commit-server to be several steps ahead of invalidation. However, opacity is not violated here, because this step-ahead is only allowed for transactions whose servers have finished invalidation.

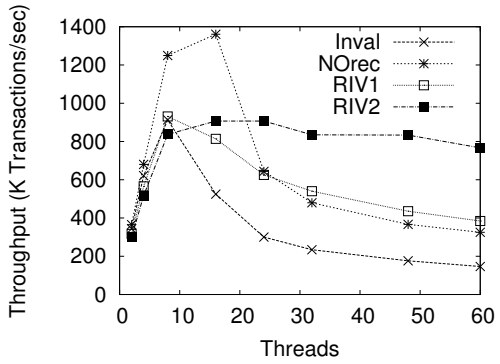
RInval also inherits all of the advantages of coarse-grained locking STM algorithms, including simple global locking, minimal meta-data usage, privatization safety [22], and easy integration with hardware transactions [21]. Hardware transactions need only synchronize with the commit-server, because it is the only thread that writes to shared memory.

V. Experimental Evaluation

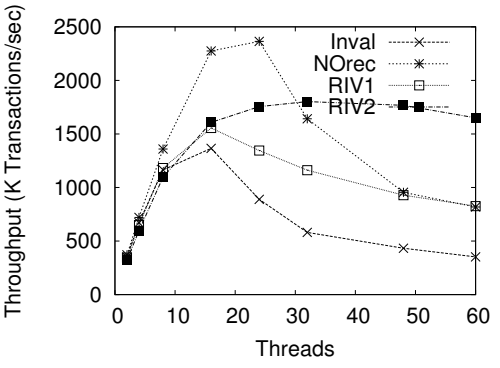
We implemented RInval in C/C++ (compiled with gcc 4.6) and ported to the RSTM framework [23] (compiled using default configurations) to be tested using its interface. Our experiments were performed on a 64-core AMD Opteron machine (128GB RAM, 2.2 GHz).

To assess RInval, we compared its performance against other coarse-grained STM algorithms, which have the same strong properties as RInval, like minimal meta-data, easy integration with HTM, and privatization safety. We compared RInval with InvalSTM [10], the corresponding non-remote invalidation-based algorithm, and NOrec [7], the corresponding validation-based algorithm. For both algorithms, we used their implementation in RSTM with the default configuration. We present the results of both RInval-V1 and RInval-V2 with 4 invalidation-servers. For clarity, we withheld the results of RInval-V3 as it resulted very close to RInval-V2. This is expected because we dedicate separate cores for invalidation-servers, which means that the probability of blocking servers is minimal (recall that blocking servers is the only case that differentiate RInval-V2 from RInval-V3)

We show results in red-black tree micro-benchmark and the STAMP benchmark [16]. In these experiments, we show how RInval solves the problem of InvalSTM and becomes better than NOrec in most of the cases. All of the data points shown are averaged over 5 runs.



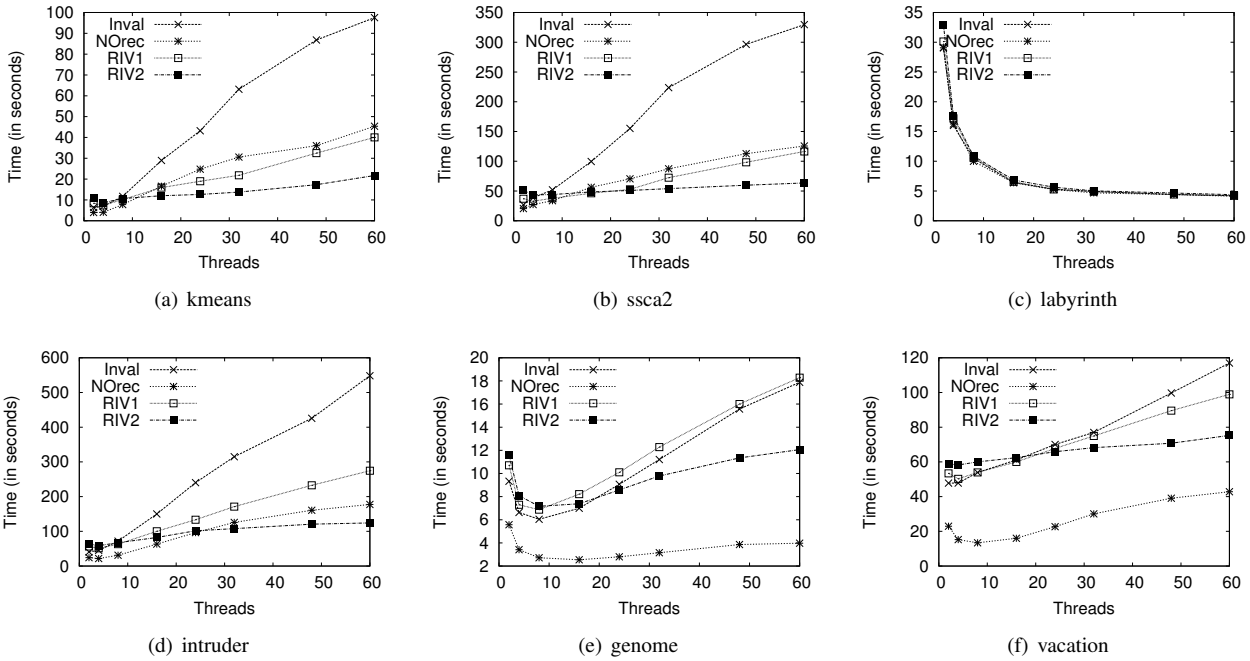
(a) 50% reads



(b) 80% reads

Figure 7. Throughput (K Transactions per second) on red-black tree with 64K elements

Red-Black Tree. Figure 7 shows the throughput of RInval and its competitors for a red-black tree with 64K nodes and a delay of 100 no-ops between transactions, for



(a) kmeans (b) ssca2 (c) labyrinth

(d) intruder (e) genome (f) vacation

Figure 8. Execution time on STAMP benchmark

two different workloads (percentage of reads is 50% and 80%, respectively). Both workloads execute a series of red-black tree operations, one per transaction, in one second, and compute the overall throughput. In both cases, when contention is low (less than 16 threads), NOrec performs better than all other algorithms, which is expected because invalidation benefits take place only in higher contention levels. However, RInval-V1 and RInval-V2 are closer to NOrec than InvalSTM, even in these low contention cases. As contention increases (more than 16 threads), performance of both NOrec and InvalSTM degrades notably, while both RInval-V1 and RInval-V2 sustain their performance. This is mainly because NOrec and InvalSTM use spin locks and suffer from massive cache misses and CAS operations, while RInval isolates commit and invalidation in server cores and uses cache-aligned communication. RInval-V2 performs even better than RInval-V1 because it separates and parallelizes commit and invalidation routines. RInval-V2 enhances performance as much as 2x better than NOrec and 4x better than InvalSTM.

STAMP. Figure 8 shows the results of the STAMP benchmark, which represents more realistic workloads. In three benchmarks ($kmeans$, $ssca2$, and $intruder$), RInval-V2 has the best performance starting from 24 threads, up to an order of magnitude better than InvalSTM and 2x better than NOrec. These results confirm how RInval solves the problem of serializing commit and invalidation, which we showed in Figure 3. In $genome$ and $vacation$, NOrec is better than all invalidation algorithms. This is mainly because they are read-intensive benchmarks, as we also showed in Figure 3. However, RInval is still better and closer to NOrec than InvalSTM. For future work, we can make further enhancements to make these specific cases even better. One of these enhancements is to bias the contention manager to readers, and allow it to abort the committing transaction if it is conflicting with many readers (instead of the classical *winning commit* mechanism, currently used). In *labyrinth*, all algorithms perform the same, which confirms the claim made in Section III, because their main overhead is non-transactional. For brevity, we did not show *bayes* as it behaves the same as *labyrinth*.

VI. Conclusions

There are many parameters – e.g., spin locking, validation, commit, abort – that affect the critical execution path of memory transactions and thereby transaction performance. Importantly, these parameters interfere with each other. Therefore, reducing the negative effect of one parameter (e.g., validation) may increase the negative effect of another (i.e., commit), resulting in an overall degradation in performance for some workloads.

Our work shows that it is possible to mitigate the effect of all of the critical path overheads. RInval dedicates server cores to execute both commit and invalidation in parallel, and replaces all spin locks and CAS operations with server-client communication using cache-aligned messages. This optimizes lock acquisition, incremental validation, and commit/abort execution, which are the most important overheads in the critical path of memory transactions.

REFERENCES

[1] T. Harris, J. Larus, and R. Rajwar, "Transactional memory," *Synthesis Lectures on Computer Architecture*, vol. 5, no. 1, pp. 1–263, 2010.

[2] R. Guerraoui and M. Kapalka, "On the correctness of transactional memory," in *Proceedings of the 13th ACM SIGPLAN Symposium on Principles and practice of parallel programming*. ACM, 2008, pp. 175–184.

[3] M. Herlihy and N. Shavit, *The Art of Multiprocessor Programming, Revised Reprint*. Elsevier, 2012.

[4] S. Heller, M. Herlihy, V. Luchangco, M. Moir, W. Scherer, and N. Shavit, "A lazy concurrent list-based set algorithm," *Proceedings of the 9th International Conference on Principles of Distributed Systems*, pp. 3–16, 2006.

[5] D. Dice, O. Shalev, and N. Shavit, "Transactional locking ii," in *Proceedings of the 20th international symposium on Distributed Computing*. Springer, 2006, pp. 194–208.

[6] T. Riegel, C. Fetzer, and P. Felber, "Time-based transactional memory with scalable time bases," in *Proceedings of the nineteenth annual ACM symposium on Parallel algorithms and architectures*. ACM, 2007, pp. 221–228.

[7] L. Dalessandro, M. Spear, and M. Scott, "Norec: streamlining stm by abolishing ownership records," in *Proceedings of the 15th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*. ACM, 2010, pp. 67–78.

[8] M. Spear, A. Shriraman, L. Dalessandro, and M. Scott, "Transactional mutex locks," in *SIGPLAN Workshop on Transactional Computing*, 2009.

[9] T. David, R. Guerraoui, and V. Trigonakis, "Everything you always wanted to know about synchronization but were afraid to ask," in *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*. ACM, 2013, pp. 33–48.

[10] J. E. Gottschlich, M. Vachharajani, and J. G. Siek, "An efficient software transactional memory using commit-time invalidation," in *Proceedings of the 8th annual IEEE/ACM international symposium on Code generation and optimization*. ACM, 2010, pp. 101–110.

[11] G. Kestor, R. Gioiosa, T. Harris, O. Unsal, A. Cristal, I. Hur, and M. Valero, "Stm2: A parallel stm for high performance simultaneous multithreading systems," in *2011 International Conference on Parallel Architectures and Compilation Techniques (PACT)*. IEEE, 2011, pp. 221–231.

[12] V. Gramoli, R. Guerraoui, and V. Trigonakis, "Tm 2 c: a software transactional memory for many-cores," in *Proceedings of the 7th ACM european conference on Computer Systems*. ACM, 2012, pp. 351–364.

[13] J.-P. Lozi, F. David, G. Thomas, J. Lawall, G. Muller *et al.*, "Remote core locking: migrating critical-section execution to improve the performance of multithreaded applications," in *Proc. Usenix Annual Technical Conf*, 2012, pp. 65–76.

[14] D. Hendler, I. Incze, N. Shavit, and M. Tzafrir, "Flat combining and the synchronization-parallelism tradeoff," in *Proceedings of the 22nd ACM symposium on Parallelism in algorithms and architectures*. ACM, 2010, pp. 355–364.

[15] J. Mellor-Crummey and M. Scott, "Algorithms for scalable synchronization on shared-memory multiprocessors," *ACM Transactions on Computer Systems (TOCS)*, vol. 9, no. 1, pp. 21–65, 1991.

[16] C. Minh, J. Chung, C. Kozyrakis, and K. Olukotun, "Stamp: Stanford transactional applications for multi-processing," in *IEEE International Symposium on Workload Characterization, IISWC*. IEEE, 2008, pp. 35–46.

[17] T. Harris and K. Fraser, "Language support for lightweight transactions," in *ACM SIGPLAN Notices*, vol. 38, no. 11. ACM, 2003, pp. 388–402.

[18] M. Herlihy, V. Luchangco, M. Moir, and W. N. Scherer III, "Software transactional memory for dynamic-sized data structures," in *Proceedings of the twenty-second annual symposium on Principles of distributed computing*. ACM, 2003, pp. 92–101.

[19] B. Bloom, "Space/time trade-offs in hash coding with allowable errors," *Communications of the ACM*, vol. 13, no. 7, pp. 422–426, 1970.

[20] L. Dalessandro and M. L. Scott, "Sandboxing transactional memory," in *Proceedings of the 21st international conference on Parallel architectures and compilation techniques*. ACM, 2012, pp. 171–180.

[21] T. Riegel, P. Marlier, M. Nowack, P. Felber, and C. Fetzer, "Optimizing hybrid transactional memory: The importance of nonspeculative operations," in *Proceedings of the 23rd ACM Symposium on Parallelism in Algorithms and Architectures*. ACM, 2011, pp. 53–64.

[22] M. Spear, V. Marathe, L. Dalessandro, and M. Scott, "Privatization techniques for software transactional memory," in *Proceedings of the twenty-sixth annual ACM symposium on Principles of distributed computing*. ACM, 2007, pp. 338–339.

[23] V. Marathe, M. Spear, C. Heriot, A. Acharya, D. Eisenstat, W. Scherer III, and M. Scott, "Lowering the overhead of nonblocking software transactional memory," in *Workshop on Languages, Compilers, and Hardware Support for Transactional Computing (TRANSACT)*, 2006.