Composability for Application-specific Transactional Optimizations

Rui Zhang, Zoran Budimlić, William N. Scherer III
Department of Computer Science, Rice University

Software Transactional Memory (STM) has made great advances towards acceptance into mainstream programming by promising a programming model that significantly reduces the complexity of writing concurrent programs. Unfortunately, the mechanisms in current STM implementations that enforce the fundamental properties of transactions — atomicity, consistency, and isolation — also introduce considerable performance overhead. This performance impact can be so significant that in practice, programmers are tempted to leverage their knowledge of a specific application to carefully bypass STM calls and instead access shared memory directly. While this technique can be very effective in improving performance, it breaks the consistency and isolation properties of transactions, which have to be handled manually by the programmer for a specific application. It also breaks another desirable property of transactions: composability.

In this paper, we identify the composability problem and propose two STM system extensions to provide transaction composability in the presence of direct shared memory reads by transactions. *TxFastRead* gives the programmer the performance of a direct memory access to the shared memory data when used from a non-nested transaction, while performing necessary bookkeeping to guarantee composability when used from a nested transaction. *TxFlush* ensures consistency between nested transactions that use *TxFastRead* and can be inserted by the programmer or automatically by the compiler. These extensions give the programmer a similar level of flexibility and performance when optimizing the STM application as existing practices, while preserving composability. We propose two implementation schemes for these extensions: *Lookup Scheme* and *Partial Commit Scheme*. We evaluate our implementation of these extensions on several benchmarks on a 16-way SMP. The results show that our extensions provide performance competitive with hand-optimized non-composable techniques, while still maintaining transactional composability.

1. Introduction

One problem Transactional Memory faces is its performance overhead. The overhead can be sufficiently large to compel programmers to carefully bypass certain TM calls based on application-specific knowledge. Such an optimization usually breaks *isolation* and *consistency* [10], two key properties of transactions, forcing the programmer to encode consistency checking into the application by hand. However, this type of optimization can also break *composability*, another desirable property of transactions; nesting transactions optimized in such a way can lead to incorrect results. In this paper we identify a composability problem associated with optimizing Software Transactional Memory (STM) [4, 5, 6, 7, 9, 12, 16] application performance using application-specific knowledge. We also propose an extension to STM systems that addresses this problem. Our proposed extension offers comparable flexibility in optimizing STM performance while maintaining the composability.

1.1 Optimization vs. Composability

In the course of optimizing STM applications through reading the shared data directly, programmers can rely on the STM system for atomicity guarantees, but rely on themselves for consistency and isolation guarantees. We note that atomicity is guaranteed provided that user-level code only reads directly from the shared data and write access to shared data is performed via transactional interfaces. Guaranteeing atomicity for user-level direct write updates would require mechanisms such as user-level clean up procedures, which is beyond the scope of this work.

There is a lurking composability problem associated with the existing practice of optimizing transactions by performing direct reads to shared memory (which we explain in Section 3). When a transaction is optimized based on application-specific information this way, it can no longer be safely composed to write larger transactions. In order to compose these optimized transactions, a developer would need to both understand the implementation details of the optimized transaction and reason about the correctness. This is very unfriendly to a software developer.

To meet the need of further optimizing STM applications performance, we propose STM system extensions to enable programmers to perform such optimizations. These extensions provide composability support and thus allow optimized transactions to be composed into larger ones.

We illustrate the composability problem itself in section 3. In section 4 we show our proposed interface extensions. We show the experimental results in section 5, discuss the results and conclude in sections 6 and 7 respectively.

2. Related Work

In order to improve STM performance, researchers have explored different STM designs, careful performance tuning of the system, and different validation techniques. These efforts focus on pure system performance, allowing the programmer to write transactions following strict specifications. There are also other types of work that consider approaches to performance improvement through relaxing some of the transactional properties and extending the tools programmers can use to encode program-specific knowledge in their applications. These techniques sacrifice some programmability for better performance. Since they are not strictly conforming to TM requirements, programmers need to spend more time and effort on reasoning about correctness of their applications. These techniques include early release [9], open nesting [13, 14, 15], and reduced-overhead shared memory access [2].

Early release was first proposed by Herlihy et al. [9] to reduce contention in the DSTM system. It allows a transaction to release reads before the transaction commits. A transactional read, once released, does not conflict with other concurrent transactions or incur validation overhead. This reduces the probability of aborting the transaction and improves overall performance. Early release requires more programming effort because it lays the burden of ensuring there are no conflicts on the programmer. Besides the programming effort, transactions using early release are not composable, so this technique affects program modularity as well. For example, a

transaction A verifies if a node exists in a sorted linked list. It can go through the list and early release every node it opens. Thus, when transaction A finishes, all reads are already released. If the information gathered in transaction A is used in a later nested transaction B, the later transaction can no longer validate if the condition is still true because all relevant nodes have been released.

Open nesting [13, 14, 15] exploits a higher semantic level of concurrency than is defined at the physical memory access layer where TM systems usually reside. It allows transactions to commit even in the presence of a physical conflict not affecting the application's semantics. The system usually needs to support virtual constructs (not necessarily locks) that can be mapped to the semantic level where the significant conflicts actually happen. A TM system supporting open nesting works with these high level constructs rather than raw memory access. For example, nested transactions are commonly used with resource allocation, where it does not matter which instance of a resource goes to a transaction. In the case of physical memory, all that matters is that different transactions allocate disjoint blocks; two transactions that both allocate memory transactionally need not conflict. Open nesting requires deep understanding of the application's semantics and is more difficult than a pure transactional approach. Also, as pointed out by Ni et al. [15], open nesting can potentially create deadlock when nested and is therefore not composable.

SNAP is a low overhead interface for shared memory access [2]. It provides functions to get, validate, and upgrade the snapshot of an object. SNAP reads, unlike regular transactional reads, do not involve bookkeeping of any information. Instead, the read returns a snapshot of the object. SNAP validation can verify if a snapshot held by the program is still valid. The programmer can also upgrade a read snapshot to a write snapshot when needed. Memory accesses in SNAP mode are neither bookkept nor validated; rather, the programmer manages correctness at the application level. It gives the programmer the flexibility to optimize some memory accesses. The programmer explicitly manages the way a memory location is accessed and the switch between transactional and non-transactional modes. SNAP access mode is not composable as well.

3. Composability Problem

In this section, we identify a composability problem in the practice of partially bypassing TM consistency mechanisms. This practice leverages application-specific knowledge in order to recover performance lost due to STM conservativeness. In particular, a programmer may choose not to use the general TM validation techniques, but to instead read directly from shared memory and to manually maintain data needed to ensure correctness. The programmer can still freely rely on the TM for other guarantees such as atomicity and consistency of the transactional reads and writes.

3.1 Definitions

Following the normal convention, we say that a transaction has a particular semantics that it is supposed to provide, and that it is correct if it provides these semantics under all concurrent execution scenarios. We further say that a transaction is *pure* if it contains no accesses to shared memory – reads or writes – unless those accesses are performed via the transactional interface. In particular, a direct read or write of memory causes a transaction to be impure. While the purity of a transaction is independent of its correctness, substantial application-specific knowledge must be employed by the programmer in order to ensure correctness of impure transactions.

We say that a transaction is *composable* if it provides the same semantics when nested inside another transaction as it does when executed as a standalone transaction, regardless of the concurrent execution environment. We claim without proof that every pure transaction is also composable; however, the next subsection details several examples where impure transactions are not composable. (In fact, we show that they can give incorrect results when nested in certain contexts.)

In Section 4, we take a first step towards remedying these composability problems by introducing an extension to transactional memory that supports direct read access to memory. This extension consists of a fast wrapper for direct reads and a flush method that performs special processing on memory locations that have been read directly. Our core claim is that proper application of our TM extension converts transactions that are impure because of direct reads into reusable, composable transactions. Due to space constraints, we omit proof of this claim, which may be found in Zhang's doctoral thesis [17].

```
atomic {
    copy global_matrix to local_matrix;
    try to find a route in local_matrix;
    if a route is found
        add the route to global_matrix
}
```

Figure 1. Labyrinth Transaction Pseudocode

3.2 Motivating Examples

One example that shows a composability problem comes from the Labyrinth application in the STAMP [1] benchmark suite. It uses Lee's algorithm [11] to find routes for a set of sources and destinations in a three-dimensional matrix modeled after circuit board routing. For each source-destination pair, the routing process expands a neighbor frontier until the destination is found, then rewinds back to the source for a feasible route. The TM implementation conducts computation in the shared matrix, generating many transactional reads along the search from source to destination. These reads have a high probability of conflicting with transactional writes, greatly decreasing performance. In order to improve performance, the benchmark is implemented such that the matrix is first copied to a local array through non-transactional reads, and then all subsequent computation is done on the local array. The implementation employs a custom consistency checking algorithm, which is used to write back found routes to shared memory. The custom consistency check only needs to verify nodes on the route; hence it obtains reduced contention and better performance.

Figure 1 sketches the algorithm used in Labyrinth. In the beginning of the transaction, a local matrix is created by copying from the global matrix. The read from the global matrix is a direct memory read without using TM interface. When a route is written back to the shared memory, the program only writes it back if all cells on the route have not been changed in the meantime. Checking whether cells have been changed is done by the programmer.

This transaction is well formed to run concurrently with other transactions even though it reads from the global matrix non-transactionally. The transaction is atomic, consistent, and isolated within the application's context because the programmer carefully manages the consistency issues. But the composability problem arises when several of these transactions are merged into a larger one. For example, if the user wants to route two source-destination pairs in a single transaction, he cannot use a nested transaction. The problem is that in a delayed update TM model, the updates of the first transaction are delayed until the top level transaction commits, making the read of the array in the second transaction get a stale value, leading to incorrect results.

Let us look at a simple linked list example to illustrate the composability problem more clearly. Lists are frequently used in transactional applications. For example, both Genome and Intruder benchmarks from the STAMP suite [1] use a linked list and it is

```
atomic insert(List* list_ptr, int key) {
    Node* prev = TM_READ(list_ptr->head);
    Node* curr = TM_READ(prev->next);
    while(TM_READ(curr->key) < key) {
        prev = curr;
        curr = TM_READ(curr->next);
    }
    if (TM_READ(curr->key) != key) {
        Node* newN = TM_MALLOC(sizeof(Node));
        newN->next = curr;
        newN->key = key;
        TM_WRITE(prev->next, newN);
    }
}
```

Figure 2. Insert - Pure TM version

reasonable to expect that a programmer might want to optimize list transactions in ways similar to that explained above. Unfortunately, list transactions are very often called within nested transactions. User optimizations would very likely break the composability of the transactions, making the code with nested transactions incorrect. This is exactly where the composability support described in this paper can help the programmer.

Imagine a set data structure based on a sorted linked list that supports three operations: insert, remove, and lookup, where the insert transaction iterates through the list to find an insertion point then adds the node. A very straightforward approach, illustrated by the pseudocode in figure 2, would be to use pure transactions and open every shared object transactionally.

While this is a straightforward way of creating such transactions, this approach does not scale with multiple threads. Nodes read prior to the insertion point will incur many conflicts with other transactions, yet most of these conflicts can be shown to be benign. Knowing this, one way to improve performance is to use a technique similar to the lazy concurrent list-based set algorithm of Heller et al. [8]. In the optimized version shown in figure 3, the code searching for the location to insert the node reads the intermediate nodes directly (without going through the TM system calls). The correctness of this approach is ensured by validating that the two nodes neighboring the insertion point did not change during the insertion by using an additional field, *marked*, in each node that indicates whether a node has been removed or not. The optimized algorithm relies on the TM system to make sure the neighboring two nodes are consistently opened. Correctness is ensured via additional consistency checks *after* the neighboring nodes have been opened transactionally.

As we will illustrate later in Section 5, the differences in overall application performance between accessing data directly and accessing via the TM interface can be very significant, suggesting that the optimized version is the way to go. Unfortunately, the insert transaction with direct memory access is not composable, as illustrated by the following examples of nested transactions:

atomic { *insert(x); insert(y);* }.

atomic { *if (lookup(value) == FALSE) insert(value);* } .

In a pure TM implementation, both examples will work as expected. However, neither nested transaction works correctly when the optimized versions of *insert* and *lookup* are used. There are two reasons that make the optimized version uncomposable.

3.3 Lurking Composability Problems

The first problem arises in delayed-update STM systems [9, 12]. In a delayed-update STM system, transactional writes are first made to a cached copy rather than to the shared variable. The cached copy is only visible to other threads after the transaction commits. In a nested transaction, transactional writes in the first *insert* are not committed until the entire nested transaction commits. So the

```
atomic insert_optimized(List* ptr, int key) {
    Node* prev = ptr->head;
    Node* curr = prev->next;
    while(curr->key < key) {
        prev = curr;
        curr = curr->next;
    }

    bool p_m = (bool_t)TM_READ(prev->marked);
    bool c_m = (bool_t)TM_READ(curr->marked);
    Node* next = TM_READ(prev->next);
    if (!p_m && !c_m && next == curr) {
        if (TM_READ(curr->key) != key) {
            Node* newN = (Node*)TM_MALLOC(sizeof(Node));
            new_N->next = curr;
            newN->marked = FALSE;
            newN->key = key;
            TM_WRITE(prev->next, newN);
        }
    } else
        TM_RESTART();
}
```

Figure 3. Insert - Uncomposable version

Transaction	Memory state		
		global_a: 0	
B / A: local_a = global_a	local_a: 0	global_a: 0	global_a is not 1 because B has not committed
local_a++	local_a: 1	global_a: 0	
TM_WRITE(global_a, local_a)	local_a: 1	global_a: 0	
A: local_a = global_a (local_a is a stale value)	local_a: 0	global_a: 0	
local_a++	local_a: 1	global_a: 0	
TM_WRITE(global_a, local_a)	local_a: 1	global_a: 0	
	wrong!	global_a: 1	

Figure 4. Hidden update illustration

direct reads in the second *insert* read stale values of the transactional writes in the first *insert*. Consequently the computation of the nested transaction is no longer consistent. By comparison, in a pure TM version, all reads from the shared memory are transactional. Transactional reads respect the read-after-write dependences across transactions and therefore do not incur this problem. We call this problem the *hidden update problem*, and it is tied to the implementation choice to use delayed updates within the STM. The hidden update problem does not occur in STM systems that use eager updates that immediately update transactional writes.

The hidden update problem can be seen more clearly in the following, even simpler example in figure 4. Transaction A reads the shared *global_a* variable and saves it to a local copy *local_a* through a direct memory read. It then increments *local_a* and writes the new value to *global_a* using a transactional write. Note that the read of *global_a* is nontransactional and therefore does not suffer the associated performance overhead. Let us also assume that the application's semantics allow transaction A to be implemented this way without interfering with other transactions in the application.

The hidden update problem arises when we nest two calls to transaction A in a nested transaction B. Suppose *global_a* is initialized to 0. The expected value of *global_a* is 2 after executing

transaction B. But the second A's read of *global_a* reads the stale value 0; the final result of transaction B will be 1.

The second problem is a consistency issue. Direct reads do not force bookkeeping of any information in the transactional system. Therefore later transactions are not able to validate the consistency of previous optimized nested transactions. For example, consider the example above of a transaction that is composed of one lookup and one insert transaction. This transaction inserts node x only when *lookup(x)* indicates x is not already in the list. Note that direct reads in *lookup(x)* are not recorded in the transaction, hence cannot be validated later in *insert(x)*. *insert(x)* is dependent on the validity of the condition of *lookup(x)*, so if another concurrent transaction inserts x into the list after *lookup(x)* but before *insert(x)*, the resulting list will have a duplicate element. The nested transaction is not able to discover this inconsistency. A pure TM implementation does not have this problem since all necessary information is recorded for all transactional reads and later nested transactions are able to validate the consistency of previous transactions.

To summarize, optimizations that bypass TM system calls and access shared memory directly break transactional composability because read-after-write and write-after-read dependences might not be respected properly when such transactions are nested.

Even though the read-after-write problem only occurs in the STM systems with delayed writes, we address composability for such systems as well: delayed writes are frequently used in existing STM systems because they have the advantages of smaller conflict windows and greater potential concurrency.

4. Fast Read Interface Extension

To meet both the need of optimizing the performance of STM applications using application-specific knowledge and of providing composability for optimized transactions so they can be more widely reused, we propose to extend the transactional memory programming interface with two additional operations. We introduce these two operations and their semantics next. We designed these extensions to maximize the extent to which programmers can benefit from optimizations embedded in the optimized transaction.

TxFastRead encapsulates a fast read operation from shared memory. $TxFastRead$ provides comparable performance compared to a raw read of a shared memory location. $TxFastRead$ alone does not completely guarantee the consistency of fast reads; rather, it must work together with our $TxFlush$ extension operation. $TxFastRead$ should be used at every place where a direct shared read would have been used in an optimized transaction. $TxFastRead$ does not employ the same heavyweight bookkeeping as does a regular transactional read, but adapts to execution context instead. It incurs no performance penalty in non-nested contexts, yet provides composability when nested. Full details of this operation may be found later in this section.

TxFlush is the counterpart operation for $TxFastRead$. First, it ensures that read-after-write dependencies are respected. Second, it provides the necessary operations to guarantee consistency. $TxFlush$ should be placed at places where these properties might be broken; typically, this is immediately before and after an optimized sub-transaction.

4.1 Composability Mechanisms

We experimented with two mechanisms for ensuring composability, *lookup scheme* and *partial commit scheme*. With both, fast path $TxFastRead$ returns the shared value with no additional bookkeeping or validation. On the slow path, $TxFastRead$ does perform some bookkeeping and validation. We carefully designed these two schemes so that the optimizations applied by the programmer can be preserved as much as possible. Within the optimized transaction, even when it is in a nested transaction, the TM system validation does not validate the fast reads. These fast reads are only validated when they are merged into the transactional read set. Therefore we expect fewer conflicts in the optimized transactions even when used as nested transactions.

```
TxFastRead(addr) {
    if not nested
        return *addr;
    else if addr is in write set
        return value in write set;
    else if validate succeeds
        record in fast read set;
        return *addr;
    else
        abort;
}
```

Figure 5. Fast Read in Lookup Scheme

```
TxFlush() {
    merge fast read set to read set;
    clear fast read set;
}
```

Figure 6. Flush in Lookup Scheme

Lookup Scheme: This approach solves read-after-write dependences by recording the transaction reads in a fast read set. Transactional writes are committed only when the entire nested transaction commits. $TxFlush$ does not commit any writes. $TxFastRead$ either looks up the address from within the write list or reads directly from shared memory. In the context of a nested transaction, $TxFastRead$ first searches the write list. If the address is not found in the write set, then the transactional read set needs to be validated (note that the fast read set is not validated here). In a non-nested context, the read is performed directly from shared memory. Figure 5 shows pseudocode for $TxFastRead$ in this scheme.

In lookup scheme, $TxFlush$ merges the fast read set to the read set. Following nested transactions validate the consistency of the enclosing transaction by validating only the read set. Figure 6 shows pseudocode for $TxFlush$.

Partial Commit Scheme: Our partial commit scheme (PCM) solves the read-after-write dependencies by eagerly updating shared memory when a nested transaction commits. It solves the inconsistent read problem by recording fast reads.

If the transaction is not nested the shared value is returned directly. Otherwise, fast-read operations first check if the location to be read is locked, and if that is the case the transaction either aborts or waits for a while; otherwise, the fast read returns the shared data after performing validation. In the PCM scheme, the read does not search the write set, which can be useful when the write set is large. Note that the fast read is nontransactional; it provides no mechanism to guarantee consistency with other reads. The flush operation performs a partial commit that commits all pending writes to shared memory and locks them. It also merges the fast read set to the transactional read set. The nested transactions also needs to save necessary information to clean up the committed writes if the entire transaction fails. Figure 7 shows pseudocode for $TxFastRead$ in this scheme.

In PCM, $TxFlush$ not only merges fast read set into the read set, but it also commits the write set to shared memory. The transaction holds locks for the committed writes, so accesses from other threads will detect a conflict and abort. Figure 8 shows the pseudo code for $TxFlush$ operation in the partial commit scheme.

Both schemes have their advantages and drawbacks. On the fast path (when the transaction is not nested), both schemes perform

```
TxFastRead(addr) {
    if not nested
        return *addr;
    else if addr is not locked and validation succeeds
        record in fast read set;
        return *addr;
    else
        clean up partial commits;
        abort;
}
```

Figure 7. Fast Read in PCM Scheme

```
TxFlush() {
    merge fast read set with read set;
    clear fast read set;
    commit current transactional writes;
}
```

Figure 8. Flush in PCM Scheme

similarly, since they directly read the shared memory. The difference is in their slow path. The lookup needs to search the write set for every fast read so it suffers an associated performance penalty. The PCM scheme does not search the write set and therefore can be faster here especially when the write set is large. The lookup scheme locks transactional writes only when the enclosing transaction commits, so it holds the locks for a shorter period of time, reducing contention over the locks. The PCM scheme commits the transactional writes when each nested transaction commits, so it holds the locks of early transactions for a longer time, which can lead to a higher contention on the locks.

The main goal of our techniques is to provide the programmer with a choice to optimize transaction performance while still preserving composability. Our extensions do not create an alternative to STM, but rather provide optimization hooks for the programmers that have the will and expertise to bypass the STM interface. Those who do not use our interface suffer neither performance penalties nor any change in STM semantics.

4.2 Composability vs. Reuse

We would like to point out here that the techniques we have described in this paper do not guarantee full reuse of the transactions, only composability. Composability is only a part of the reuse, even though a very important one. While this can be seen as a limitation of our approach, we want to point out that hand-optimized transaction code that bypasses the TM interface also cannot be reused in general, within a nested transaction or otherwise. Therefore our original claim that we provide composability for optimized code that bypasses the pure TM interface still stands.

To illustrate the reuse issue, let us consider the sorted linked list described earlier. It supports three transactions: lookup, insert, and remove. Suppose the programmer wants to add a new *increment* transaction that increments a value of a node by 1 (swapping it with the next node if necessary). Even if this new transaction is implemented using the pure TM interface without any optimizations, it will still break the existing hand-optimized code: adding a transaction that changes a node's value breaks the original assumption that only insert, remove, and lookup can be performed on the list, which made the optimization possible. This is true even if the programmer has used our extended TM interface to implement the optimizations. The programmer would have to revisit the assumptions made about the whole application and re-implement the optimized transactions with the new *increment* transaction in mind.

However, we also note that our TM interface extension *does* allow for the *increment* transaction to be implemented by simply making it a composition of a remove and an insert transaction. No changes to the existing code would be necessary.

Our extension still enables a significant amount of reuse, with a restriction that the new code does not contain transactional writes. If the added transactions only perform transactional reads and/or call the existing transactions, everything will perform as expected. Otherwise, a programmer will have to revisit the assumptions about the whole application and re-implement the optimized transactions.

5. Experimental Results

We conducted our experiments on six benchmarks: Labyrinth, Genome, Intruder, Vote, Set, and Nested Set. The experiments are performed on a 16-core SMP machine with four quad-core Intel Xeon CPU E7330 running at 2.40GHz. Three of the benchmarks in our experiments - Labyrinth, Genome, and Intruder - are from the STAMP [1] benchmark suite. The other three were developed previously by the authors [18].

We have (where applicable) four versions of each benchmark — pure TM, uncomposable, lookup scheme, partial commit scheme. We refer to the version that strictly follow TM requirements as the pure TM version. In the pure TM version, every read from or write to the shared memory passes through the TM interface. The pure TM version enjoys all the benefits from the TM system and requires the least effort to develop. We refer to the version using direct shared memory reads as the uncomposable version. The programmer is responsible for ensuring correctness through implementing isolation and consistency by hand. This version requires much more programmer effort. The versions that use our proposed fast read interface include the lookup scheme and partial commit scheme. In both schemes, the direct accesses in the uncomposable version were replaced with the calls to the fast read interface. Similar to the uncomposable version, the programmer needs to guarantee the correctness. But transactions using our fast read interface can still be composed into larger transactions.

We implemented our extension on top of TL II [3]. All of the experiments are performed using the lazy acquire mode in TL II.

Labyrinth is a maze routing application described in Section 3, operating on a 512*512*7 3D matrix. The runtime parameters used in our experiments for Labyrinth were "-i inputs/random-x512-y512-z7-n512.txt". ***Genome*** is a gene sequencing program. It takes a number of DNA segments and matches them to reconstruct the original genome. In phase one of the benchmark, all segments are put into a hash set (implemented as a set of unique buckets, each of which is implemented as a linked list) to remove segment duplicates. We created different versions of Genome by modifying the list data structure. The runtime parameters used in our experiments for Genome were "-g16384 -s64 -n16777216". ***Intruder*** is a signature-based network intrusion detection application. It scans network packets for matches against a known set of intrusion signatures. The main data structure in the capture phase is a simple non-transactional FIFO queue. Its reassembly phase uses a dictionary (implemented by a self-balancing tree) that contains lists of packets that belong to the same session. We modified the list data structure used in the dictionary to create different versions of Intruder. The runtime parameters used in our experiments for Intruder were "-a10 -l512 -n262144 -s1". ***Vote*** simulates a voting process that supports three transactions — vote, count and modify. The underlying data structure is a binary search tree. The `vote(ssn, candidate)` transaction casts a vote for a candidate on behalf of the voter, and is composed of `verify(ssn)` transaction (verifies if a voter has voted) and the `cast_vote(ssn, candidate)` transaction (casts the actual vote if this voter has not voted yet). Since `vote` is a nested transaction, there is no correct uncomposable version. There are 65,536 possible unique voters. The mix of operations of count, vote and modify is 10%, 80%, and 10%. ***Set*** implements

a set using a sorted linked list. It supports three operations and each is implemented as a transaction - insert, remove and lookup. `Insert(key)` inserts a key to the set. `Remove(key)` removes a key from the set. `Lookup(key)` searches for the key in the set. There are 512 unique keys in the set. The operation mix of insert, remove and lookup is 10%, 10% and 80%. ***Nested Set*** is a nested version of *Set*. It has three nested transactions, nested insert, nested remove, and nested lookup. Each of the nested transaction has two of the corresponding single transactions within it. The number of unique keys and operation mix is the same as in the *Set* above.

6. Discussion

Figures 9 and 10 show the performance results that we have collected for the six benchmarks that we have tested. *Labyrinth* and *Set* on Figure 9 illustrate the overhead that our extensions introduce over the direct shared memory access approach (that is also uncomposable). *Genome*, *Vote*, *Nested Set* and *Intruder* on Figure 10, which all contain nested transactions, illustrate the performance improvement that our extensions provide over a pure TM implementation. Direct memory access for applications on Figure 10 would result in incorrect behaviour, therefore there are no "uncomposable" versions of those benchmarks.

In *Labyrinth*, *Set*, *Vote*, and *Genome* we observe that using our fast read interface, the performance improves by a significant margin over pure TM implementations. We also observe some minor performance overhead over the direct read version.

Labyrinth shows the largest performance gain, clearly illustrating that certain types of applications can achieve enormous performance benefits through application-specific optimizations. Because many of the transactional reads in the pure TM version cause an immense number of conflicts and effectively livelock, the pure TM version of the Labyrinth version runs so slowly that we were unable to finish the experiments for cases with more than 4 threads. The two thread case takes over a day to complete (compared to about 50 seconds for the hand-optimized uncomposable version and the composable version using our TM interface extension). Thus, we have omitted the pure TM results for Labyrinth in Figure 9.A.

In *Genome*, the list operations are either a stand alone transaction or a part of a nested transaction that can be easily shown to be independent of other list operations. As shown in Figure 10.A, we observe a performance improvement of 26% in the 4 thread case using our lookup scheme. Across all thread counts, we observe a performance improvement ranging from 18% to 26%. The partial commit scheme encounters performance issues for more than 4 threads and the results are clipped in Figure 10.A, because the locked writes keep forcing other transactions to abort. The reason is that in the partial commit scheme transactional writes update the shared memory every time a $TxFlush$ is called. These writes will hold their locks from the point the $TxFlush$ is called until the entire nested transaction commits. This increases the contention time compared with the lookup scheme and pure TM version that only hold the lock in the final commit phase.

For *Vote*, we observe a performance improvement of up to 150% over the the pure TM version. There is no uncomposable version of *Vote*. The results of lookup and PCM schemes are nearly identical.

For the list-based *Set* benchmark, Figures 9.B and 10.C show the results for single and nested transactions. We observe up to 80% performance improvement over the pure TM version with our lookup scheme, and a 47% overhead over the uncomposable version for the single thread case. The overhead is mainly from the additional work of maintaining the fast read set and merging it to the transactional read set. The performance overhead decreases as the amount of parallelism increases, indicating better scalability. For nested transactions, the performance improvement is up to 30%. Though the two nested transactions clearly have no dependencies, the benchmark is implemented by assuming they might and inserting $TxFlush$ between them. Better performance could be achieved if the programmer were to take advantage of this knowledge and remove the $TxFlush$ call.

The last benchmark, *Intruder*, shows an on average minimal performance improvement of a couple percent (Figure 10.B) over the pure TM version with our lookup scheme, and on average minimal performance loss of a couple percent over our partial commit scheme. The list operations in Intruder are only used in nested transactions and infrequent. Therefore the optimized list operations performs very similar to their pure TM versions. Note that an uncomposable version of this benchmark also does not exist.

Following are some of the design choices and issues that arose while developing the TM interface extension, in no particular order:

Lookup Scheme vs. Partial Commit Scheme: In our experiments, the lookup scheme outperforms the PCM scheme in most cases. The larger time window in which we hold locks in the PCM scheme increases the possibility of a conflict. The PCM scheme does not require later nested transactions to search the past write set. For applications with an expensive search process, we expect that the PCM scheme would achieve better performance.

Non-in-place update STM: We only evaluate an in-place update TM system in our experiments. For STM systems that do not use in-place update [12, 9], similar problems will arise when the programmer wants to read directly the visible copy. We will explore non-in-place update STM systems in the future.

Fast Read Advantages and Disadvantages: Our fast read extension addresses the problem that application-specific optimized transactions do not integrate with the STM runtime/library to provide composability guarantees. Under the hood, the extension provides the necessary link between the programmer and the consistency guarantee mechanism integrated in the STM system. Therefore the places to use the extension are similar to ones where the original optimizations apply. The major performance incentive of the optimization is to eliminate part of the time-consuming validation work involved in a pure STM implementation. If an application's consistency depends on most of its past reads, the work it can save could become rather limited and might not be worth the effort of developing an optimized version.

Pure TM version implementation of Labyrinth: We implemented the pure TM version of Labyrinth by copying the entire matrix to a local version transactionally, performing routing using the local matrix, then writing the found route back to the global matrix. This is very similar to the original algorithm used in the STAMP Labyrinth benchmark. An alternative approach would perform routing in the global matrix directly. However, the algorithm uses a breadth first search which marks the distance to the source to find the shortest path from the destination backward to the source, and then cleans up marks left by the first step. We expect that this approach would perform badly, due to marking updates in the first and third steps causing an enormous number of conflicts.

TxFlush Programmer Visibility: For correctness, a $TxFlush$ is required between every transactional write to a memory location loc and a subsequent fast read to loc. If $TxFlush$ is exposed to the programmer, he/she can reason about the dependencies between nested transactions and only insert $TxFlush$ where strictly necessary. This could increase code complexity and the possibility of writing erroneous code. Alternatively, one can envision a compiler that inserts $TxFlush$ before and after every nested transaction that uses the fast read interface, then analyzes the dependencies between shared memory accesses and removes the unnecessary $TxFlush$ calls, further improving the performance of our proposed techniques. In this paper, we have used a straightforward and conservative approach that inserts $TxFlush$ before and after every nested transaction that uses the fast read interface.

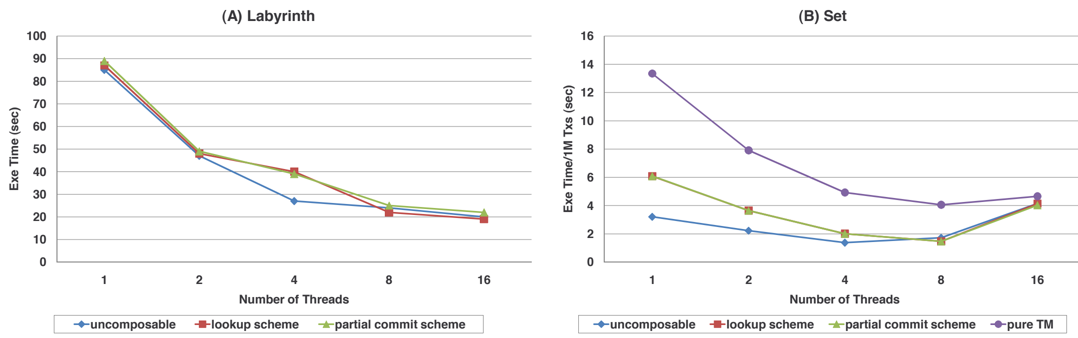

Figure 9. Showing execution time for two different benchmarks with different number of threads

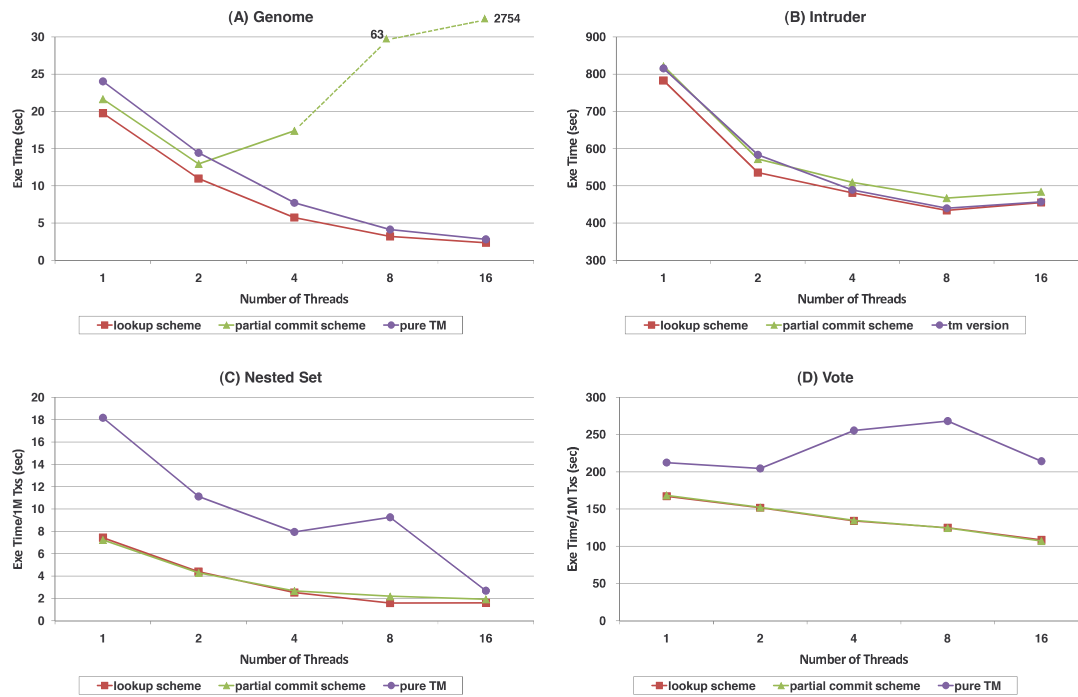

Figure 10. Showing execution time for four different benchmarks with different number of threads

7. Conclusions and Future Work

In this paper, we have identified that common programmer practices in optimizing transactional applications by bypassing TM system calls and accessing the shared memory directly, in addition to breaking consistency and isolation (which have to be handled by the programmer manually), also break another desirable property of transactions: composability.

We have proposed two extensions to the TM system interface: $TxFastRead$ and $TxFlush$. These extensions enable the programmer to access the shared memory in a much more controlled manner, without breaking transaction composability. We have presented two techniques for implementing these system extensions: lookup scheme and partial commit scheme. We implemented these techniques on top of the TL II software transactional memory implementation and demonstrated on a set of benchmarks that we can obtain performance that is competitive to the non-composable hand optimized code, while preserving composability.

In summary, compared to the existing practices, our system extensions require similar programmer effort (the programmer still needs to manually ensure isolation and consistency), provide similar performance, and preserve composability.

For future work, we will investigate the composability issues in non-in-place updates for transactional writes. We will also explore compiler optimizations to further simplify the extended TM interface available to the programmer for optimizations.

Acknowledgments

We are grateful to the anonymous reviewers for commentary that has substantially improved the presentation of this paper. We are particularly indebted to reviewer #3, whose "reverse engineering" feedback was spot-on and key to improving the accessibility of this work.

References

[1] C. Cao Minh, J. Chung, C. Kozyrakis, and K. Olukotun. STAMP: Stanford transactional applications for multi-processing. In *IISWC '08: Proceedings of The IEEE International Symposium on Workload Characterization*, September 2008.

[2] C. Cole and M. Herlihy. Snapshots and software transactional memory. In *PODC Workshop on Concurrency and Synchronization in Java Programs*, 2004.

[3] D. Dice, O. Shalev, and N. Shavit. Transactional locking II. In *Proceedings of the 20th International Symposium on Distributed Computing, DISC 2006*. Springer, Sep 2006.

[4] K. Fraser. *Practical lock freedom*. PhD thesis, Cambridge University Computer Laboratory, 2003. Also available as Technical Report UCAM-CL-TR-579.

[5] R. Guerraoui, M. Herlihy, and B. Pochon. Polymorphic contention management. In *DISC '05: Proceedings of the nineteenth International Symposium on Distributed Computing*, pages 303–323. LNCS, Springer, Sep 2005.

[6] T. Harris and K. Fraser. Language support for lightweight transactions. In *OOPSLA '03: Proceedings of the 18th annual ACM SIGPLAN conference on Object-oriented programing, systems, languages, and applications*, pages 388–402, New York, NY, USA, 2003. ACM Press.

[7] T. Harris, M. Plesko, A. Shinnar, and D. Tarditi. Optimizing memory transactions. *SIGPLAN Not.*, 41(6):14–25, 2006.

[8] S. Heller, M. Herlihy, V. Luchangco, M. Moir, , W. N. Scherer III, and N. Shavit. A lazy concurrent list-based set algorithm, 2005.

[9] M. Herlihy, V. Luchangco, M. Moir, and W. N. Scherer III. Software transactional memory for dynamic-sized data structures. In *PODC '03: Proceedings of the twenty-second annual symposium on Principles of distributed computing*, pages 92–101, New York, NY, USA, 2003. ACM Press.

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

[11] C. Y. Lee. An algorithm for path connection and its application. In *IRE Trans. Electronic Computer, EC-10*, 1961.

[12] V. J. Marathe, M. F. Spear, C. Heriot, A. Acharya, D. Eisenstat, W. N. Scherer III, and M. L. Scott. Lowering the overhead of nonblocking software transactional memory. In *TRANSACT 06': Proceedings of the Workshop on Languages, Compilers, and Hardware Support for Transactional Computing*, 2006.

[13] J. E. B. Moss. Open nested transactions: Semantics and support. In *WMPI Poster*, 2005.

[14] J. E. B. Moss and A. L. Hosking. Nested transactional memory: model and preliminary architecture sketches. In *OOPSLA Workshop on Synchronization and Concurrency in Object-Oriented Languages (SCOOL)*, 2005.

[15] Y. Ni, V. S. Menon, A.-R. Adl-Tabatabai, A. L. Hosking, R. L. Hudson, J. E. B. Moss, B. Saha, and T. Shpeisman. Open nesting in software transactional memory. In *PPoPP '07: Proceedings of the 12th ACM SIGPLAN symposium on Principles and practice of parallel programming*, pages 68–78, New York, NY, USA, 2007. ACM.

[16] 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 *PPoPP '06: Proceedings of the eleventh ACM SIGPLAN symposium on Principles and practice of parallel programming*, pages 187–197, New York, NY, USA, 2006. ACM Press.

[17] R. Zhang. *Performance Optimizations for Software Transactional Memory*. PhD thesis, Rice University, 2010.

[18] R. Zhang, Z. Budimlić, and W. N. Scherer, III. Commit phase in timestamp-based STM. In *SPAA '08: Proceedings of the twentieth annual symposium on Parallelism in algorithms and architectures*, pages 326–335, New York, NY, USA, 2008. ACM.