Composability for Application-Specific Transactional Optimizations

Rui Zhang Zoran Budimlić William N. Scherer III
Department of Computer Science, Rice University, Houston, Texas 77005
Email: {ruizhang, zoran, scherer}@rice.edu

Software Transactional Memory (STM) has made great advances towards acceptance into mainstream programming by promising a programming model that greatly 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 the specific application. It also tends to break 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. Our proposed extensions give the programmer a similar level of flexibility and performance when optimizing the STM application as the existing practices, while preserving composability. We evaluate our 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.

I. INTRODUCTION

Achieving higher performance is a major driving force for concurrent programming. With the catalyst “power wall” crisis that the hardware industry has been facing for several years now, the transition from sequential programming to concurrent programming has arrived quickly and decisively. This leads to a very urgent requirement for programming tools to lower the complexity of concurrent programming. These tools need to provide elegant, easy-to-use programming constructs as well as competitive performance. As a construct, Transactional Memory [10] provides many appealing features that conceal most of the complexity in concurrent programming, which greatly reduces the difficulty of writing concurrent programs.

Though transactions have been widely and successfully used for a long time in databases, they still face many problems as a general programming construct. One key obstacle to widespread adoption is their performance overhead. This 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 [10] and consistency [10], two key properties of transactions, so the programmer has to encode by hand consistency checking into the application. However, this type of optimization can also break composability [10], another desirable property of transactions; nesting transactions optimized in such a way can lead to incorrect results. In this paper we identify the composability problem associated with optimizing Software Transactional Memory (STM) [4], [5], [6], [7], [9], [12], [17] 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.

Why would anyone want to bypass TM system by accessing the shared memory directly in the first
place and lose isolation, consistency, composability and reuse guarantees? Would it not be a better idea to forego the transactional memory completely and just design a non-blocking algorithm from the ground up, if the performance is of such importance? The answer is that the TM system still guarantees transaction atomicity, a very important property, even in the presence of optimizations that bypass the TM system calls and read the shared memory directly. Ensuring application correctness without the atomicity guarantees provided by the TM system would require very significant effort on the part of the programmer — a nonblocking algorithm for even simple data structures such as queues is a publishable result, for example.

A. Performance Overhead in STM

Performance overhead comes from several different aspects of an STM system. For example, an STM system must perform additional work to guarantee consistency, even though consistency can be trivially guaranteed in a sequential program. In order to provide this guarantee, an STM system has to perform sufficient recording of transactional reads to validate them when needed. This additional work can sometimes entirely negate the initial purpose of using a parallel implementation: better performance [18]. The argument for parallelization does not hold if the performance of a parallel version trails that of a sequential counterpart that takes less effort to develop. Though a lot of research has been conducted on designing efficient STM systems to reduce the overhead, the performance of software transactional memory is still far from being solved.

Although part of the overhead of STM, such as consistency checking in general, is unavoidable due to the nature of concurrency, other parts arise because of the conservative assumptions made by the STM system. Designed as a general programming model, the STM system lacks application-specific knowledge. This lack of information forces the STM system to perform additional work that a more application-specific approach could avoid. For example, many STM systems keep a set of past transactional reads [9], [12] and validate that a transaction is still consistent by checking whether any of these variables has changed. If any of the past reads has changed, the transaction is deemed to be inconsistent and aborted. This type of validation is very conservative and gives false positives in many execution scenarios, since a change of a value that has been previously read by a transaction does not necessarily imply an order cycle among transactions. Consequently, the STM system will abort many transactions that could have been allowed to complete.

In timestamp-based STM systems such as TL II [3], the systems often assume that if an object being opened is newer than the transaction’s timestamp, a conflict has occurred, another conservative design. Conservativeness can also be found in other aspects of STM design. For example, some systems [16] mix the use of timestamp- and list-based validation to reduce the number of false positives.

Another part of the overhead comes from the optimistic nature of STM systems. The STM system allows multiple transactions to run concurrently, hoping that they will not conflict. But transactions do conflict and thus abort. Worse, the amount of conflict often increases with the number of threads concurrently running in the system, which leads in turn to yet more wasted work.

For some applications, the overhead from the STM system can be high enough to make programming based on STM less desirable. Consequently, for certain applications, the programmer may have enough performance gain incentive to design custom consistency guarantee mechanisms. This would effectively reduce part of the system overhead associated with the STM system. Importantly, though the performance gains can be substantial, the work to apply such optimizations is not trivial and requires good understanding of both the application and of concurrent programming.

B. Optimization vs. Composability

In the course of optimizing STM applications through reading the shared data directly, programmer 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 cleanup 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 III). 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 hide the implementation details, so the programmer can reuse the optimized transactions with only understanding of the programming interface but not the actual implementation details. Importantly, these extensions provide composability support and thus allow optimized transactions to be composed into larger ones.

We illustrate the composability problem itself in section III. In section IV we show our proposed interface extensions. Then we show the experimental results in section V. Afterwards, we discuss the results and conclude in section VI and section VII respectively.

II. RELATED WORK

Improving STM performance has recently been a focus of numerous research efforts. Researchers have explored different STM designs, careful performance tuning of the system, and different validation techniques such as Lazy Snapshot [16], Transactional Locking II [3], etc. These efforts focus primarily on pure system performance, allowing the programmer to write transactions following strict transaction specifications. However, 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 transactional applications. Using these techniques sacrifices some programmability for better performance. Since they are not strictly conforming to the 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 lowering the overhead of shared memory access [2].

Early release is a technique first proposed by Herlihy, et al. [9] to reduce contention in their DSTM system. Early release allows a transaction to release some transactional reads before the transaction commits. A transactional read, once released, does not conflict with other concurrent transactions or incur the associated validation overhead. This reduces the probability of aborting the transaction and improves the 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] is a technique to exploit 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 necessary 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 constructs associated with raw memory access. For example, consider a nested transaction composed of two transactions, each of which inserts a node to a list. If the application only requires that these two transactions either both complete or neither of them complete but does not care if another transaction intervenes in between, open nesting can be used to improve performance. Similar
to early release, using open nesting requires deep understanding of the application’s semantics and is more difficult than a pure transactional approach. Moreover, open nesting is 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 read, unlike a regular transactional read, does 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 do not suffer the transactional overhead of recording the reads and validating them. This is possible through additional programming effort of making correctness guarantees 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.

III. COMPOSABILITY PROBLEM

In this section, we identify the composability problem in the user’s practice to improve STM application’s performance by partially bypassing the TM system’s consistency guarantee mechanisms. This type of user practice integrates application-specific knowledge into the code in order to recover performance lost due to the conservativeness of the STM system. For example, a programmer may choose not to use the general validation technique of the TM system, but instead to read directly from the shared memory and maintain necessary data structures by himself to ensure the correctness of the transaction. The programmer can still freely rely on the TM system for other guarantees such as atomicity and consistency of the transactional reads.

One motivating example to show this composability problem is the Labyrinth application in the STAMP [1] benchmark.

The Labyrinth benchmark 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 pair of source and destination, the routing process expands its neighbor frontier till the destination is found. Then the routing process rewinds back to the source for a feasible route. The benchmark handles multiple pairs of sources and destinations in parallel. A straightforward pure TM implementation conducts the computation in the shared matrix, and generates many transactional reads along the search from the source to the destination. Given multiple concurrent threads, these reads have high probability to conflict with transactional writes and greatly decrease performance. In order to improve performance, the benchmark is implemented so 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 user-customized consistency checking algorithm. The consistency algorithm is used to write back routes found back to the shared memory. The customized consistency checking only needs to verify the nodes on the route and therefore greatly reduces the contention level and achieves much better performance.

Figure 1 is a pseudo code sketch of 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 when all cells on the route have not been changed in the meantime. The check of whether the cells are 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 transaction. For example, if the user wants to route a pair of sources and destinations in one transaction, he cannot use a nested transaction that has two of the routing transaction in Labyrinth. One of the problems is that in a delayed update TM model, the updates of the first transaction are delayed until the top level transaction commits. This makes the read of the array in the second transaction get the stale value and leads to incorrect results.

Below, we will take a 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 benchmark suite [1] use a linked list and it is reasonable to expect that the programmer would want to optimize the list transactions in ways similar to the one 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.

For example, imagine a set data structure based on a sorted linked list that supports three transactions: insert, remove, and lookup, where the insert transaction incorporates two steps. The first step is to iterate through the list to find the location to insert the new node. The second step performs node insertion. A very straightforward approach, illustrated by the pseudo code in Figure 2, would be to use pure transactions and open every shared object transactionally.

While this is a very straightforward and simple way of creating such transactions, this approach does not scale very well with multiple threads. The main reason is that the nodes opened up to the insert point will incur many conflicts with other transactions, even though many of these conflicts can be shown to be benign. Knowing this, one of the ways to improve performance is to use the technique similar to the lazy concurrent list-based set algorithm developed by 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. The correctness is ensured by additional consistency checking after the neighboring nodes have been opened transactionally.

As we will illustrate later in Section V, the differences in overall application performance between accessing the data directly and accessing through the transactional interface can be very significant, suggesting that the optimized version is the way to go. Unfortunately, even though the insert transaction works well by itself within the context of the list based set, it is not composable. For example, the programmer may want to have two insert transactions in the nested transaction as in figure 4.

Another example is that the programmer may want to insert a node to the set when such a node does not exist in the set and do this in a single transaction. The transaction will look similar to figure 5.

In a pure TM implementation, the above two examples will work as expected. However, neither of the above nested transactions work correctly when the optimized versions of insert and lookup are used. There are two reasons that make the

```c
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* new_node = TM_MALLOC(sizeof(Node));

        new_node->next = curr;
        new_node->key = key;

        TM_WRITE(prev->next, new_entry);
    }
}
```

Fig. 2. Insert - Pure TM version
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 * new_node = (Node*)TM_MALLOC(sizeof(Node));
            new_node->next = curr;
            new_node->marked = FALSE;
            new_node->key = key;
            TM_WRITE(prev->next, new_node);
        } else {
            TM_RESTART();
        }
    }
}

Fig. 3. Insert - Uncomposable version

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

Fig. 4. A Composed List Insertion Example

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

Fig. 5. A Composed Conditional List Insertion Example

The first problem arises in the delayed-update STM systems [9], [12]. In a delayed-update STM system, transactional writes are first made to a cached copy rather than into the shared variable. The cached copy is only committed to be visible to other threads when the transaction is committed. In the nested transaction, transactional writes in the first `insert` are not committed until the entire nested transaction commits. So the 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. In comparison, in the pure TM version, all reads from the shared memory are transactional. Transactional reads respect the read-after-write dependencies 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 the STM systems that use eager updates that immediately update the transactional writes.

The hidden update problem can be seen more clearly in the following, even simpler example in figure 6.

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 not through the TM interface and therefore does not suffer the associated performance overhead. Let us also assume that the application’s semantics allows the transaction A to be implemented this way so that it does not interfere 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 exam-
ple, consider a nested transaction in figure 5 that is composed of one lookup transaction and one insert transaction. This transaction inserts node x only when \(\text{lookup}(x)\) indicates x is not already in the list. Note that all direct reads in \(\text{lookup}(x)\) are not recorded in the transaction and therefore cannot be validated later in \(\text{insert}(x)\). \(\text{insert}(x)\) is dependent on the validity of the condition of \(\text{lookup}(x)\), so if another concurrent transaction inserts x into the list after \(\text{lookup}(x)\) but before \(\text{insert}(x)\), 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, allowing optimizations that bypass TM system calls and access the shared memory directly breaks transactional composability because the read-after-write and write-after-read dependence might not be respected properly when such transactions are nested.

Even though the read-after-write problem only occurs in the STM systems with a delayed write, we will address the composability for such systems as well, since delayed writes are frequently used in existing STM systems because they have the advantage of having a smaller conflict window and enable greater potential concurrency.

IV. 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.

- \(\text{TxFastRead}\) encapsulates a fast read operation from shared memory. It also provides a hook for necessary operations to guarantee composability. \(\text{TxFastRead}\) provides comparable performance compared to a raw read of a shared memory location. \(\text{TxFastRead}\) alone does not completely guarantee the consistency of fast reads; rather, it must work together with our \(\text{TxFlush}\) extension operation. \(\text{TxFastRead}\) should be used at every place where a direct shared read would have been used in an optimized transaction. \(\text{TxFastRead}\) 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.

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

A. Composability Mechanisms

We experimented with two mechanisms for ensuring composability, we call them lookup scheme and partial commit scheme. With both, fast path \(\text{TxFastReads}\) return the shared value without any additional bookkeeping or validation. On the slow path, \(\text{TxFastRead}\) 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 less conflicts in the optimized transactions even when used as nested transactions.

Lookup Scheme: Our lookup scheme solves read-after-write dependencies by searching previous transactional writes. It ensures read consistency by recording the transaction reads in a fast read set.

This scheme does not commit transactional writes before the entire nested transaction commits. Therefore transactional writes are not committed to the
shared memory when \textit{TxFlush} is called. In this scheme, \textit{TxFastRead} either looks up the address from within the write list or reads directly from shared memory. In particular, in the context of a nested transaction, \textit{TxFastRead} first searches the write list; however, in a non-nested context, this is skipped and the read is performed directly from shared memory. If the transaction is nested and the address is not found in the write set, then the transactional read set need to be validated (note that the fast read set is not validated here).

Figure 7 shows the pseudo code of the \textit{TxFastRead} function in lookup scheme.

In lookup mode, \textit{TxFlush} merges the fast read set to the read set. Following nested transactions can validate the consistency of the enclosing transaction by validating only the read set. This addresses the inconsistency problem in nested transactions. Figure 8 shows the pseudo code for the \textit{TxFlush} operation in the lookup scheme.

**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.

In PCM, if the transaction is not nested, the shared value is returned directly. Otherwise, fast-read operations first check if the address to be read is locked. If it is locked, the transaction either aborts or waits for a while; otherwise, the fast read performs necessary validation and returns the shared data or aborts. In the partial commit scheme, the read does not search the write set. This is particularly useful when the write set is large and there are many variables in the write set. Note that the fast read is not a transactional read because it does not provide a 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 9 shows the pseudo code for the \textit{TxFastRead} operation in the partial commit scheme.

In PCM, \textit{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 10 shows the pseudo code for \textit{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 similarly. They
both perform a direct read of the shared memory and return that value. The difference is in their slow path. The lookup scheme’s slow path needs to search the write set for every fast read so it suffers the associated performance penalty. The PCM scheme’s slow path does not search the write set and therefore can be faster here especially when a large write set needs to be searched. In the lookup scheme, transactional writes are only locked when the enclosing transaction commits and therefore the locks are held for a shorter period of time. This can reduce the contention over the locks. In the partial commit scheme, transactional writes are committed in steps when each nested transaction commits. Therefore the locks of early transactions are held for a longer time compared to lookup mode. This can lead to higher contention on the locks.

$TxFush$ can be exposed to the programmer or inserted by a compiler after every nested transaction. While exposing it may allow the programmer to identify the cases where $TxFush$ is not necessary and further optimize the performance, it could also increase complexity of the code and the possibility of writing erroneous code.

B. 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 it is 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 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 to preserve the order). Even if this new transaction is implemented using the pure TM interface without any optimizations, it will still break the existing hand-optimized code. The reason is that adding a transaction that changes a node’s value breaks the assumptions made in the original hand-optimized code, namely 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.

C. Composability Guarantees

As discussed in Section III, the composability problem can arise due to two issues — ignored read-after-write dependence and consistency violations.

Here, we provide an informal argument as to why our extensions guarantee transaction composability. The argument for general composability of transactions in our scheme can be made inductively from the four claims below for two transactions that are nested within a larger one. When composing new transactions, transactional reads in the new code containing calls to the nested transactions can be considered as simple and very small transactions. Transactional writes are not allowed in the new code, as discussed in the previous subsection. Following the strict nested transaction semantics, larger transactions containing more than two nested transactions can be rewritten with deeper nesting so that every level of nesting contains only two transactions. There is (conceptually) a $TxFush$ at the beginning and at the end of every transaction that is optimized by using fast reads.
1) **Pure transaction followed by a pure transaction** does not violate consistency or read after write dependencies. This is trivially true in all STM systems.

2) **Pure transaction followed by an optimized transaction.** The reads in the pure transaction are recorded in the read set, which is validated later in the optimized transaction on every fast read. The writes within the pure transaction will be kept in the write set in the lookup scheme, which will be validated (in both schemes) against fast reads in the following optimized transaction. The writes from the pure transaction will be committed to shared memory in the partial commit scheme during the \(TxFlush\) between the two transactions, so they will be locked to the access from other threads, but can be accessed (either through fast reads or through transactional reads) from the following optimized transaction.

3) **Optimized transaction followed by a pure transaction.** In both schemes, the \(TxFlush\) between the transactions will merge the fast reads from the optimized transaction into the pure transactional read set, so the pure transaction will validate all memory accesses against those reads. The writes in both the optimized and the pure transaction are transactional so they will be either kept in the write set or partially committed in the \(TxFlush\) operation and validated accordingly in the pure transaction.

4) **Optimized transaction followed by an optimized transaction.** The fast reads from the first optimized transaction will be merged into the pure read set during the \(TxFlush\) operation, so they will be validated against during the fast reads within the second optimized transaction. Transaction writes can be shown to be correct similar to the 2) and 3) above.

V. Experimental Results

We conducted our experiments on six benchmarks: Labyrinth, Genome, Intruder, Vote, sorted linked list, and nested sorted linked list. 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 [19].

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 unlike the uncomposable version, the 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.

- The **Labyrinth** benchmark is a maze routing application. It uses Lee’s algorithm [11] to find the shortest-distance routes for a set of sources and destinations in a three dimensional matrix. The algorithm expands from the source point using a breadth first search. The search is guaranteed to reach the destination and a reverse back search formulates the route. The maze size used in the benchmark is 512*512*7. The routing process involves many memory writes that can have an enormous impact on performance if all are performed in the shared memory. In order to improve performance, the program is optimized by first copying the matrix to a local array through non-transactional reads and doing all subsequent computation on the local array. When a route is found, it is written back to the shared memory. The version provided within STAMP benchmark
suite is the uncomposable version. We created two composable versions by changing the local array copy to fast reads. The pure TM version is created by changing the local array copy to transactional reads.

- The **Genome** benchmark is a gene sequencing program. It takes a number of DNA segments and matches them to reconstruct the original genome. In the phase one of the benchmark, all segments are put into a hash set to remove segment duplicates. To allow concurrent access, the hash set is implemented as a set of unique buckets, each of which is implemented as a linked list. In our experiments, we created different versions of the Genome by modifying the list data structure.

- The **Intruder** benchmark 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 the different versions of Intruder.

- The **Vote** benchmark is an application to simulate a voting process. It supports three operations — vote, count and modify. The underlying data structure maintaining the voting information is a binary search tree. Each node contains a two fields — voter and candidate voted by the voter. The `vote(ssn, candidate)` transaction casts a vote for a candidate on behalf of the voter with his ssn. The `count(candidate)` transaction returns the total number of votes a candidate has got. This transaction is a nested transaction composed of two transactions - verify and cast_vote. The `verify(ssn)` transaction verifies if the voter with a given ssn has voted. The `cast_vote(ssn, candidate)` transaction casts the actual vote if this voter has not voted yet. The `modify(ssn, candidate)` transaction changes a voter's vote to the new candidate if the voter with a given ssn exists. Since `vote(ssn, candidate)` is a nested transaction, the uncomposable version does not work correctly here. So we created three different versions of the benchmark: pure TM (where shared data is accessed through transactional memory interface), and two versions that access the data through our fast reads composable interface (with both the lookup scheme and the partial commit scheme as the underlying implementation). There are 65,536 possible unique voters. The mix of operations of count, vote and modify is 10%, 80% and 10%.

- The **set** benchmark is an application that 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. In our experiments, there are all together 512 possible unique keys in the set. The operation mix of insert, remove and lookup is 10%, 10% and 80%.

- The **nested set** benchmark is a nested version of application above. 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. For example, a nested insert transaction is shown in figure 4. The number of unique keys and operation mix is the same as the set above.
by a significant margin relative to a pure TM implementations. We also observe some performance overhead compared to the direct read version in cases where an uncomposable version exists.

Labyrinth is the application that shows the largest performance gain, clearly illustrating that certain types of applications can achieve enormous performance benefits by applying application-specific optimizations. In fact, 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). This is because many of the transactional reads in the pure TM version cause an immense number of conflicts and effectively create a live lock

VI. DISCUSSION

In Labyrinth, Set, Vote, and Genome we observe that the optimized version’s performance improves
in the application. For this reason, we have omitted the pure TM results for Labyrinth in Figure 11, in order to illustrate the performance differences between the uncomposable optimized version of the benchmark and the composable version using our TM interface.

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 12, 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 lookup scheme fares better in this benchmark than the partial commit scheme. The partial commit scheme encounters performance issues for more than 4 threads and the results are clipped in Figure 12.

Vote has a nested transaction vote that is composed of a lookup transaction to verify whether the voter has voted and an insert transaction that casts the actual vote. Figure 13 shows the results. We observe a performance improvement up to 150% when comparing our composable version with the pure TM version. There is no uncomposable version of this benchmark. The results of lookup scheme and PCM scheme are very close and are almost overlapped on the graph.

For the set benchmark based on a sorted linked list, Figures 14 and 15 show the results for single and nested transactions. We observe up to a 1.8x performance improvement over the pure TM version with our lookup scheme. We also observe that the performance overhead compared with the uncomposable version is up to 47% for the single thread case. The overhead is majorly from the extra 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 which indicates better scalability. For the case of nested transactions, the performance improvement is up to 1.3x. Though the two transactions nested 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 that there is no dependence between transactions and remove the $TxFlush$ call.

The last benchmark, Intruder, does not show performance improvement (Figure 16), but rather a minimal (couple of percent) performance degradation when compared to the pure TM version. The reason is that the list operations in Intruder are only used in nested transactions and infrequently at that. The trade-off of spending additional time to set up a separate read set and merge it to the transactional read set does not pay off in this case. Note that an uncomposable version of this benchmark does not even 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 partial commit scheme in most cases. The larger time window in which we hold locks in the partial commit scheme increases the possibility of a conflict. The advantage of the partial commit scheme is that it does not require later nested transactions to search the past write set. For applications that feature an expensive search process, we expect that the partial commit 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, 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:** The pure TM version of Labyrinth is implemented by first copying the entire matrix to a local version transactionally. Then, routing is performed in the local matrix. The route found is written back to the global matrix. This is very similar to the original algorithm used in the Labyrinth benchmark.

A second approach would be to perform the routing computation in the global matrix directly. The first step of the routing algorithm is a breadth first search from the source to the destination. In this phase, the each cell visited is marked with a distance to the source. The first step stops when the destination is found. The second step of the algorithm searches from the destination backward to the source for the shortest path from source to destination. The third step is to clean up the marks left by the first step. We expect that this approach would perform just as bad as the first one, since the updates of marks in the first and third step will cause an enormous number of conflicts and the application would be very vulnerable to live locks. We implemented the first approach in our experiments.

- **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. 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.

VII. **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.

**References**


