Software Partitioning of Hardware Transactions*

Lingxiang Xiang and Michael L. Scott

Technical Report #993

Department of Computer Science, University of Rochester
{lxiang,scott}@cs.rochester.edu

August 2014

Abstract

Best-effort hardware transactional memory (HTM) allows complex operations to execute atomically and in parallel, so long as hardware buffers do not overflow, and conflicts are not encountered with concurrent operations. We describe a programming technique and compiler support to reduce both overflow and conflict rates by *partitioning* common operations into read-mostly (planning) and write-mostly (completion) operations, which then execute separately. The completion operation remains transactional; planning can often occur in ordinary code. High-level (semantic) atomicity for the overall operation is ensured by passing an application-specific *validator* object between planning and completion.

Transparent composition of partitioned operations is made possible through fully-automated compiler support, which migrates all planning operations out of the parent transaction while respecting all program data flow and dependences. For both micro- and macro-benchmarks, experiments on IBM z-Series and Intel Haswell machines demonstrate that partitioning can lead to dramatically lower abort rates and higher scalability.

1 Introduction

Twenty years after the initial proposal [13], hardware transactional memory is becoming commonplace: early efforts by Azul [6] and Sun [7] have been joined by Intel [25] and three separate projects at IBM [5, 14, 21]. All of these—and all that are likely to emerge in the near future—are *best effort* implementations: in the general case, a transaction may abort and retry not only because of an actual data conflict with some concurrent transaction, but also because of various hardware limitations—notably, the size or associativity of the space used to buffer speculative reads and writes.

One of the most appealing aspects of transactional memory—and a major advantage over locks—is the ability to *compose* larger transactions out of smaller ones without sacrificing performance or risking deadlock. As TM becomes more widely used, we can expect that many transactions will incorporate smaller, pre-existing operations (typically library calls). Unfortunately, because of their increased size, composed transactions tend to be precisely the ones that place the highest pressure on hardware TM limits [25].

*This work was supported in part by NSF grants CCF-0963759, CCF-1116055, CNS-1116109, CNS-1319417, CCF-1337224, and CCF-1422649, and by support from the IBM Canada Centres for Advanced Studies.

```
1  atomic { // users: a shared hashtable; accounts: a shared rb-tree
2    User* u = 0;
3    if ((u = htableFind(users, streetAddress) != 0) {
4      for (int i=0; i<u→numAccounts; i++) {
5        Account *acct = rbtreeFind(accounts, u→accts[i]);
6        sum += acct→val;
7        acct→lastAccessedTime = timeStamp;
8      }
9    } else {
10     u = (User*)malloc(sizeof(User));
11     htableInsert(users, streetAddress, u);
12     initUser(u);
13   }
14 }
```

Figure 1: A big transaction for account maintenance.

Consider the example transaction in Figure 1, which includes query and update operations on two separate data structures, with data dependences and nonlinear control flow among these operations. This code may have difficulty completing as a pure hardware transaction: its spatial footprint may be too large for hardware bufferings, and its temporal duration may increase the likelihood of conflict aborts, particularly given that conflict in any constituent operation will abort the entire transaction.

One way to improve the odds of completion for large transactions on best-effort HTM is to pull read-only preliminary ("planning") work out of the transaction, thereby reducing both the memory footprint and the temporal duration of the work that remains. Afek et al. [1] explored this approach in the context of software TM; they call it "consistency oblivious programming," because the work that is removed from the transaction must be able to tolerate an inconsistent view of memory. We have pursued a similar partitioning of lock-based critical sections, to shorten critical path length [22, 23]. In subsequent work, we suggested that partitioning might reduce the abort rates of hardware transactions [24]; this suggestion was echoed by Avni and Kuszmaul [2]. In all this work, the key observation is that a transaction (or critical section) can often check the continued validity of a plan computed in advance more quickly—and with fewer memory references—than it could regenerate the plan from scratch.

Unfortunately, while the partitioning of small transactional operations is often straightforward (at least for the experts who write libraries), its naive application is incompatible with composability. In Figure 1, for example, suppose we have access to a partitioned version of the rbtreeFind operation. We cannot simply take the planning part of the lookup at line 5 and hoist it out of the parent transaction without knowing what was (will be?) returned by the hash table lookup at line 3. Avni and Suissa have suggested [3] that one suspend an active transaction and perform planning in non-speculative mode, but while this may reduce the transaction's spatial footprint, it does nothing for temporal duration. The suspension mechanism, moreover, is supported on only one current hardware architecture [5], where it is quite expensive.

We assume that the partitioning of operations like rbtreeFind will continue to be performed by hand. Our contribution is to automate composition, with no need for special hardware. We call our approach *partitioned transactions* (ParT). For purposes of exposition, we distinguish between *operations*, which are partitioned by hand, and *transactions*, which are partitioned automatically.

Given a library of partitioned operations, each of which follows a set of well-defined structural and semantic rules, we allow the programmer to write—and compose—transactions that call these

operations in their original unpartitioned form. Automatic compiler support then extracts the planning portions of the embedded operations and hoists them out of the outermost parent transaction, along with sufficient "glue code" to preserve all inter-operation control and data dependences.

We call the halves of a partitioned operation its *planning operation* and its *completion operation*. The completion operation is always performed by (or subsumed in) a hardware transaction; the planning operation may be a hardware transaction, or it may run in ordinary code. The halves of a partitioned transaction are its *planning phase* and its *completion phase*. The completion phase is always a hardware transaction; the planning phase runs in ordinary code, possibly peppered with smaller hardware transactions used by planning operations. A summary of the plan for each individual operation (including expected return values and side effects) is carried through the planning phase and into the completion phase, automatically, by means of *validator* objects held in thread-local storage.

Returning to Figure 1, suppose we have a partitioned implementation of htableInsert. Planning operation htableInsertP figures out where to insert a key in the table and saves this position in an appropriate validator object, hidden in thread-local storage. Completion operation htableInsertC accesses V and validates its plan. It then either performs any necessary updates (if the plan is still valid) or performs the whole operation from the beginning (if the plan is no longer valid). Given similar partitioned implementations of rbtreeFind, htableFind, and malloc, our compiler will generate code along the lines of Figure 2. The planning phase (lines 1–13) tries to call as many planning operations as possible (note that initUser has been removed). Unlike regular code, which must run to completion, the planning phase may safely stop at any time—e.g., because it encounters inconsistent data, or because the savings from further planning is unlikely to be worth the cost.

The completion phase (lines 15–28) is almost identical to the original transaction, except that completion operations are called instead of the original unpartitioned versions. So where is the magic? Inside the partitioned operations and in the validator objects that carry information from each planning operation to the corresponding completion operation (and to any subsequent planning operations that may need to see its side effects).

Assuming our code is for an outermost transaction, the completion phase may abort for any of the usual reasons, at which point there are several options. If we choose to repeat the completion phase, we can shortcut any suboperations whose previous plans still validate successfully. As a last resort, a transaction that aborts repeatedly can—as in current run-time systems for HTM—retry with a global lock.

ParT provides several benefits to hardware transactions simultaneously. By shrinking the footprint of transactions, partitioning reduces the likelihood that hardware buffering limits will be exceeded. By shrinking both footprint and duration, it reduces the likelihood of conflicts with concurrent transactions. By re-executing only those planning operations whose plans are no longer valid, it achieves the effect of partial rollback when the completion phase aborts. Finally, to the extent that the planning and completion phases access common locations, the former serves to warm up the cache for the latter, further reducing the duration of the completion transaction.

On the IBM zEnterprise EC12 [14], on which we ran many of our experiments, cache warm-up has an extra benefit. On this particular machine, requests within a transaction to promote a cache line from shared to exclusive state may cause the transaction to abort. The abort can be avoided by using an exclusive prefetch to transition directly from invalid to exclusive state, but only if the line has not yet been read within the transaction. This situation poses a dilemma for operations like list updates: we don't want to exclusively prefetch every list node (since that would make all transactions conflict with one another), but we don't know *which* node to update until after we have read it. With partitioned transactions, we can issue an appropriate, limited number of exclusive prefetches immediately before beginning the completion transaction.

```
1  plan { // non-atomic region, invisible to other threads
2    User* u = 0;
3    if ((u = htableFindP(users, streetAddress) != 0) {
4      for (int i=0; i<u→numAccounts; i++) {
5        Account *acct = rbtreeFindP(accounts, u→accts[i]);
6        sum += acct→val;
7        acct→lastAccessedTime = timeStamp;
8      }
9    } else {
10     u = (User*)mallocP(sizeof(User));
11     htableInsertP(users, streetAddress, u);
12   }
13 } // keep only validator-related modifications hereafter
14
15 complete { // a hardware transaction
16   User* u = 0;
17   if ((u = htableFindC(users, streetAddress) != 0) {
18     for (int i=0; i<u→numAccounts; i++) {
19       Account *acct = rbtreeFindC(accounts, u→accts[i]);
20       sum += acct→val;
21       acct→lastAccessedTime = timeStamp;
22     }
23   } else {
24     u = (User*)mallocC(sizeof(User));
25     htableInsertC(users, streetAddress, u);
26     initUser(u);
27   }
28 }
```

Figure 2: A conceptual partitioning for the transaction of Figure 1.

2 ParT Execution Model

Given the non-atomicity of planning, a partitioned transaction needs to satisfy two basic requirements to be equivalent to the original transaction: (1) the planning phase should be logically side-effect free: its execution should be invisible to other threads, it should impact the completion phase only through the precomputation of useful hints, and it should never exhibit erroneous or undefined behavior; (2) the completion phase, given a correct planning phase, should produce the same results as the original transaction. This section formalizes these requirements, and begins the enumeration of rules that must be followed when writing partitioned operations. We continue the enumeration in Section 3.

2.1 Histories

Following standard practice, we model a computation as a *history* of events, including reads, writes, and transaction begin and end. We assume that histories are well formed—in particular, that tbegin and tend events are paired and properly nested in each thread subhistory, and all the events of any outermost transaction are contiguous in the overall history (that is, reflecting atomicity, the events of a transaction do not interleave with events in other threads). For simplicity, we assume a static partition between thread-local and shared data (the model could easily be extended to accommodate privatization and publication if desired). We ignore events within aborted transactions, since they

have no visible effect. We also ignore reads and writes of transaction-local state, since these have no impact outside the transaction.

For the sake of the formalism, we assume the availability of checkpoint and restore primitives in each thread, to modify the behavior of both shared and thread-local memory. (In our compiler, these are implemented using a cache of recent changes, which is consulted before each access in the planning phase, and discarded at the end.) In a well-formed history, checkpoint and restore occur in un-nested pairs, outside of any transaction. Their purpose is to change the mapping from reads to writes. Specifically, if a read of location l in thread t occurs within a checkpoint-restore pair in t's subhistory, the read will return the most recent write to l by t within the checkpoint-restore pair; if there is no such write, the read will return the most recent write to l in the overall history. Finally, we assume the availability of per-thread *scratchpad* memory, which is not affected by checkpoint and restore.

2.2 Partitioned Operations

A contiguous subsequence of the events within a transaction may correspond to an *operation* (the invocation of a method) on some abstract object O. As suggested informally in Section 1, we allow the library programmer to rewrite a method m of O as a *planning* method m^P and a *completion* method m^C. We impose a variety of restrictions on this rewriting, which we enumerate throughout this section and the next.

1. Planning and completion methods must take the same arguments, and produce the same results, as the original methods they replace. All three must must be written in a strict object-oriented style: they must touch no memory other than their own parameters and the state of their common object. Planning and completion methods may, however, read and write scratchpad memory, which is not accessed outside of operations.

We assume, in any well-formed history, that m is called only inside of transactions. Our compiler will preserve this property for m^C; thus, both m and m^C can comprise straightforward sequential code. By contrast, we expect calls to m^P to occur outside of any program-level transaction; therefore

2. Each planning method must be written in such a way that its execution will linearize with arbitrary concurrent executions of original, completion, and planning methods of the same object.

The simplest way to ensure this linearizability is to place m^P within its own small hardware transaction; alternatively, the code may be written in a nonblocking or hybrid style. We also require that

3. A planning method must make no change to the abstract state of its object.

4. When invoked on any given abstract state of their common object, original and completion methods must produce the same return value(s) and make the same changes to their object's abstract state.

5. Planning methods must "see" the object state corresponding to recent planning operations in the same thread. More precisely, an execution of m^P, performed by thread t, must produce the same return value(s) that m would have produced, if every planning operation performed by t since the last ordinary or completion operation had been replaced with the corresponding ordinary operation.

In practice, this last requirement implies that m^P must memoize (in scratchpad memory) the changes to O's abstract state that would have been performed by m.

2.3 Partitioned Transactions

Consider a history H containing a transaction T. Suppose that for every operation o_i in some prefix of T, we have available partitioned implementations o_i^P and o_i^C. Suppose further that we are able to replace the sequence

tbegin s_0 o_1 s_1 o_2 s_2 ... o_k s_k ...

where the s_i are sequences of non-method-call events, with

checkpoint s_0 o_1^P s_1 o_2^P s_2 ... o_k^P s_k restore
tbegin s_0 o_1^C s_1 o_2^C s_2 ... o_k^C s_k ...

The first line of this new sequence is the planning phase; the second is (a prefix of) the completion phase. If the two sequences remain jointly contiguous in the history, we claim they will have the same behavior as the sequence they replaced. The proof is straightforward: the o_i^P operations have no impact on the abstract states of their objects; the o_i^C operations have the same impact as the o_i operations they replace; and any impacts of the o_i^P operations or s_i sequences on thread-local or (other) shared state will be reversed by the restore operation.

In practice, of course, the new planning phase has no guarantee of contiguity: it no longer lies within a hardware transaction. Our compiler addresses this issue by generating code (assisted by planning methods) to double-check consistency after every shared-memory read or planning operation. If a conflicting operation in another thread has intervened, the planning phase stops early—in effect truncating itself as of the end of the consistent prefix.

3 Toward A Library of Partitioned Operations

Building on the framework of Section 2, we now turn to the practical details of writing partitioned operations. These will in turn support the compiler framework of Section 4. As noted in Section 1, previous projects have explored how to create partitioned operations [1, 22, 23, 24, 2]. We review the idea here, casting it in a form amenable to our composition efforts.

As a general rule, it makes sense to partition an operation if much of its work can be off-loaded to the planning operation, and if validation of the plan is comparatively cheap. Perhaps the most obvious examples fit a *search-then-modify* pattern, where the search is time consuming (and may have a large memory footprint), but the continued validity of a found location can quickly be checked via local inspection. Other examples fit a *compute-then-modify* pattern: in transactions resembling some variant of y = expensive_pure_function(x), the expensive function can be precomputed in a planning operation; later, if the value of x has not changed, the completion operation can use the precomputed y. Additional examples appear in Section 3.5.

3.1 The Basic Partitioning Template

To partition method foo of a concurrent object, the library designer creates two new functions foo_P and foo_C. Both should have the same argument types and return type as foo. Function foo_C will always be called within an atomic block, and, like foo, may be written without regard to synchronization. For foo_P, which will be called outside the main transaction, the library designer must devise a linearizable nonblocking implementation. Transactions can of course be used to simplify this task. In the limit, the entire planning operation can be placed in a transaction.

Informally, foo_P "figures out what to do" and embeds this plan in a *validator* object. We have already noted (in rule 5 of Section 2.2) that the validator must capture the (abstract) changes

```
1  setA = {5}
2  atomic {
3    setA.remove(5)
4    if setA.contains(5)
5      ... // an infinite loop
6  }
```

(a) original transaction

```
7  setA = {5}
8  plan {
9    setA.remove_P(5)
10   if setA.contains_P(5)
11     ... // an infinite loop
12 }
```

(b) planning phase

Figure 3: A transaction (a) and the planning phase of its partitioned implementation (b).

that would have been made to foo's object if foo had been called directly. This information serves to carry dependences from one planning operation to the next. Continuing our list of partitioned operation requirements,

6. A validator must precisely specify its *plan*—the concrete changes that need to be made to foo's object by the completion operation.

7. The validator must carry whatever information is required to confirm the continued validity of the plan. This will typically be a set of predicates on certain fields of foo's object.

In previous work, we have explored several ways to build a validator [23]; we review these briefly in Section 3.3. The details are up to the library designer: our composition mechanism (Section 4) does not access validators directly. To assist designers, we provide what amounts to a simple key-value store in thread-local memory. A planning operation and its corresponding completion operation need only agree on a key (often the id of their mutual object) in order to pass information between them. Multiple operations on the same object can, if desired, share a validator.

If validation succeeds, foo_C should execute "fast path" code that effects the changes to foo's object captured by the validator's plan. Otherwise, it should abandon the plan and switch to a fallback code path, which is commonly the original operation foo. It is worth emphasizing that planning operations and validators are simply optimizations: in the degenerate case, we can always fall back on the original code.

3.2 Dependences and Consistency

Rule 5 specifies that if a transaction includes multiple operations on the same shared object, each successive operation must see the changes envisioned by its predecessors. The need for this rule can be seen in the transaction of Figure 3a. Line 5 should never be executed, because line 4 always returns false. In the planning phase of the ParT transaction (Figure 3b), if the dependence were not respected (i.e., if setA.contains_P were unable to see the results of setA.remove_P), line 10 would return true and behave incorrectly. Fortunately, dependences need not generally be captured at the level of reads and writes. In Figure 3b, it suffices for the validator to contain a concise indication that "5 has been removed from A."

As we shall see in Section 4, our compiler algorithm safeguards the consistency of planning in composed transactions by incrementally checking the continued validity of any shared data read outside partitioned operations. To supplement this checking, we add a final rule to our list of partitioned operation requirements:

8. A planning operation must add to the read set of the surrounding planning phase enough locations to guarantee that if the operation's return value is no longer valid, at least one of the added locations will have changed.

We provide library designers with a ParT_readset_put operation that serves this purpose.

3.3 Semantics-based Validation

Several validation strategies are possible in partitioned operations. Perhaps the most obvious is to double-check the values of all data read in the planning operation. This is essentially the strategy of *split hardware transactions* [17]. Conceptually, it transfers the whole read set of the planning operation over to the completion operation. Consistency is trivially ensured at the implementation level. For a compute-heavy planning operation, this strategy may be entirely appropriate: for these, the goal is to reduce the temporal window in which the top-level transaction is vulnerable to spurious (e.g., false sharing) conflicts. Unfortunately, for a search-heavy planning operation, passing the full read set may make the completion as likely to cause an abort as the original monolithic operation.

Complete consistency of all read locations is not always required for high-level correctness, however: the data to be modified may depend on only a portion of the read set, or on some property that does not always change when the values of the data change. In search structures, for example, it is often possible to verify, locally, that a location is the right place to make an update: how one found the location becomes immaterial once it has been found.

Local verification may even succeed when local values have changed. In a sorted list, if the planning operation suggests inserting key k after node X, and passes a pointer to X in the validator object, the completion operation may succeed even if another node has already been inserted after X, so long as k still belongs between the two.

In other cases, application-level version numbers can be used to concisely capture the status of natural portions of a data structure. We used such numbers to good effect in our MSpec work [23], at various granularities. Seen in a certain light, application-level version numbers resemble the ownership records (ORecs) of a software TM system. They protect the entire read set, but allow rapid validation by employing a data-to-Orec mapping specifically optimized for the current data structure.

3.4 Extended Example: Red-black Tree

A red-black balanced tree supports log-time insert, remove, and find operations with an attractively small constant factor. Each operation begins with a tree search, which can be moved into an explicitly speculative planning operation [24, 2]. To make this planning amenable to composition, we must structure it according to rules 1–8.

The code skeleton for partitioned insert is shown in Figure 4 (lines 2–29), as it would be written by a library designer. To ensure correct execution, we use a type-preserving allocator [18] for nodes, and increment a node's version number, ver, when the node is deallocated, reallocated, or modified. Like the structure described by Avni and Kuszmaul [2], the tree is *threaded* with predecessor (pred) and successor (succ) pointers.

The internal lookup operation serves to initialize a validator. To support arbitrary key types and to avoid the possibility that a speculative search will find itself on the wrong branch of the tree due to a rotation, we call lookup within a small transaction, delimited by ParT_plan_htm_begin and ParT_plan_htm_end. These primitives differ from the usual htm_begin and htm_end in that repeated aborts cause fallback not to a software lock, but to a ParT_plan_stop routine (corresponding to

```
1  #pragma plan_for RBTree::insert
2  bool RBTree::insert_P(KeyT &k) {
3    ParT_plan_htm_begin();
4    ParT_readset_validate();
5    RBValidator *v = getRBValidator(this, k);
6    if (v == 0) { // first access to (this, k)
7      v = newRBValidator(this, k);
8      lookup(v, k);
9      ParT_readset_put(&v→curr→ver);
10   }
11   ParT_plan_htm_end();
12   bool ret = !v→localExist;
13   v→localExist = true;
14   if (!v→exist) allocNode_P();
15   return ret;
16 }
17
18 #pragma complete_for RBTree::insert
19 bool RBTree::insert_C(KeyT &k) {
20   RBValidator *v = getRBValidator(this, k);
21   if (v && v→isValid()) {
22     if (!v→exist) {
23       Node *n = allocNode_C();
24       ... // insert n as v→curr's neighbor
                and rebalance the tree
25     }
26     return !v→exist;
27   } else
28     return insert(k); // fallback path
29 }

30 struct RBValidator {
31   RBTree::Node *curr; // last accessed node
32   int ver; // snapshot of curr's version
33   bool exist, localExist;
34   bool isValid() {return curr&&curr→ver==ver;}
35 };
36
37 void RBTree::lookup(RBValidator *v, KeyT &k) {
38   Node *p = root→right; // root is a dummy node
39   v→curr = root; v→ver = root→ver;
40   while (p) {
41     v→curr = p; v→ver = p→ver;
42     if (k == p→key) {v→exist=true; break;}
43     p = k < p→key ? p→left : p→right;
44   }
45 }
```

Figure 4: Partitioned insert method (lines 1–29) and validator (lines 30–35) for partitioned red-black tree.

the restore primitive of Section 2) that truncates the planning operation and any planning phase in which it appears, transferring control immediately to the completion operation or phase. All rb-tree validators are kept in thread-local storage, indexed by the address of the tree and a key (line 5). To minimize allocation cost, a fixed number of these validators are pre-allocated; if the supply is exhausted, newRBValidator will call ParT_plan_stop internally. If a desired key doesn't exist, memory space is reserved for a new node (line 14), which will be claimed in the completion phase (line 23), again using implicit thread-local storage.

A version-based implementation of the validator is shown in lines 30–35. During planning, we initialize it with the last accessed node (curr), a snapshot of its *version number* (ver), and an indication as to whether the desired key was found (exist). In a subsequent completion operation (e.g., RBTree::insert_C, which is called inside the completion phase of the parent transaction), the isValid method will succeed only if (1) the key has been searched for by at least one planning operation (curr!=0), and (2) curr has not subsequently been modified (curr→ver==version). Note that isValid will never return a false positive. It may return a false negative, but this will simply trigger execution of the fallback path (line 28).

The dependence chain through insert_P, remove_P, and find_P is tracked by the localExist flag of the (key-specific) validator. Consistency of the parent transaction's planning phase (Section 4.2.2) is ensured by by adding appropriate version numbers to the read set used by that transaction (line 9) and by performing additional calls to ParT_readset_validate (e.g., line 4) when consistency is needed within a planning operation.

3.5 Additional Examples

As transactional memory becomes more widely adopted, we envision a library of partitioned operations that can be used, transparently, to reduce transaction duration, memory footprint, and consequent abort rates. In all cases, manually partitioned operations, constructed once by an expert, can be called at arbitrary points within a larger transaction, using the same syntax that would have been used for the unpartitioned operation (as explained in detail in Section 4, the compiler hides all details of separating out the planning phase).

Collections: Ordered and unordered sets, mappings, and buffers are among the most common shared abstractions. Operations often start with a search component that is easily moved to a planning operation, and verified quickly in the completion operation. In some cases (e.g., the red-black tree of Section 3.4), the planning phase will use its own hardware transaction to ensure a consistent view of memory. In other cases (e.g., a sorted linked list), the planning operation can (with a bit of care) be written to tolerate inconsistency, and run without HTM protection.

Memory allocation: Depending on the choice of allocator, calls to malloc and its relatives may be a significant component of transaction run time and a major source of conflicts. Partitioning can pull the actual allocation out of the transaction. The completion phase confirms that the space was indeed pre-allocated, and simply uses it. An additional "clean up" hook (not described in detail here) may be used at the end of the transaction to reverse any allocations that were mis-speculated, and not actually needed in the completion phase. In contrast to naive, manual pre-allocation, which allocates the maximum amount of space that might be needed in any execution of the transaction, our compiler-supported partitioning mechanism will reflect conditional branches within the transaction, avoiding unnecessary memory churn.

Object initialization: Object constructors (initializers) are often called immediately after allocation. A constructor is easily partitioned if—as required by rule 1—it modifies only the state of the object. The planning phase performs all the writes; the completion phase confirms that the constructor arguments have not changed since the planning phase. Writes in a mis-speculated plan are unnecessary, but semantically harmless.

Commutative operations: Operations that commute with one another need not necessarily execute in the serialization order of the transactions in which they appear. Operations such as random number generation, unique id generation, or lookup in a memoization table can be performed entirely in a planning phase; the validator simply encapsulates the result.

4 Automatic Composition

Transactions in real applications are often composed of multiple simpler operations. Our partitioned hardware transactions (ParT) work with existing HTM, and support the composition of multiple partitioned operations. Given a library of such operations, we allow the programmer to write code like that of Figure 1, which the compiler will then partition into code like that of Figure 2.

4.1 Language Interface

To enable automatic composition, we provide the following compiler directives:

#pragma part is placed before the code of an atomic block to instruct the compiler to transform the transaction to its ParT form. In Figure 1, the directive would be inserted before line 1, prompting the compiler to automatically create the desired composed partition. Given a ParT library that covers many operations, a TM programmer can easily request partitioning with minimal changes to the source code.

#pragma plan_for func and **#pragma complete_for func** allow a ParT library designer to link planning and completion methods to an original method with name func, so the compiler knows that func is partitionable and what its planning and completion methods are. In Figure 4, these have been used at lines 1 and 18. As required by rule 1, planning and completion methods take the same parameters as the original method.

#pragma stop tells the compiler that planning should stop at this point. This directive allows the programmer to fine-tune performance by precomputing only the initial portion of a transaction.

4.2 Automatic Partitioning

Starting with annotated source code, our compiler synthesizes a planning method for each identified top-level atomic block. The algorithm is recursive, so atomic blocks can be nested: every function and atomic block that calls a partitionable operation (and that may itself be called, directly or indirectly, from a to-be-partitioned transaction) is partitioned into planning and completion phases; these can then be called by higher-level functions. The compiler rewrites the original atomic block as the code of the completion phase.

4.2.1 Synthesizing Planning Code

The goal of synthesis is to generate a minimal and safe planning phase that covers as many planning operations as possible. The synthesis algorithm begins by cloning and extracting the code of a ParT transaction as a separate function. It then passes this function to SynthesizePlan (Algorithm 1) to generate the composed planning function.

All functions called directly or indirectly from a ParT transaction are categorized as one of the following:

Partitionable: These have pre-defined planning functions (identified by #pragma plan_for), so there is no need to synthesize them.

Unsafe: These include functions with no source code available, library functions (unless specified as partitionable operations), OS APIs, I/O operations, and various other special cases. They preclude the use of planning for the remainder of the transaction.

Others: These may require (recursive) synthesis to generate their planning functions. They include the function created for the outermost ParT transaction. An "Other" function will typically comprise two kinds of code: calls to other functions and the skeleton or *glue code* that connects these functions calls.

Algorithm 1: SynthesizePlan(func, fTable)

```
   Input: a function func, a function information table fTable
 1 if func is marked as a partitionable operation then
 2 |   fTable[func].stopBefore ← false;
 3 |   fTable[func].plan ← GetPragmaPlanFor (func);
 4 else if func is unsafe then
 5 |   fTable[func].stopBefore ← true;
 6 |   fTable[func].plan ← null;
 7 else
 8 |   planFunc ← CloneFunction (func);
 9 |   foreach #pragma stop in planFunc do
10 |   |   replace #pragma stop with function call to ParT_plan_stop
11 |   foreach function call C in planFunc do
12 |   |   ... // code for checking recursion of func is omitted;
13 |   |   f ← C.calledFunction;
14 |   |   if f not in fTable then
15 |   |   |   SynthesizePlan(f, fTable);
16 |   |   if not fTable[f].stopBefore then
17 |   |   |   replace C with function call to fTable[f].plan;
18 |   |   else
19 |   |   |   insert function call to ParT_plan_stop before C;
20 |   PrunePlan (planFunc);
21 |   InstrumentPlan (planFunc);
22 |   fTable[func].stopBefore ← false;
23 |   fTable[func].plan ← planFunc;
```

SynthesizePlan inspects the source code of each "Other" function. It inserts ParT_plan_stop before any call to an unsafe operation (line 19) and replaces calls to "Partitionable" and "Other" functions with calls to their planning functions (line 17).

PrunePlan (line 20) reduces the size of the generated planning function. First, any code strictly dominated by a call to ParT_plan_stop is removed. Second, if planFunc is a top-level transaction, we perform backward slicing on each call to a partitioned operation. Instructions that do not belong to any of these slices are removed, leaving only the "glue" instructions necessary to invoke the subsidiary planning functions.

InstrumentPlan (line 21) deals with data consistency in the glue code. Load and Store instructions are instrumented if they may read or write shared memory. The instrumentation redirects them to the read and write logs of the planning phase, and validates all previous reads when a new location is read. More details about data consistency are given in Section 4.2.2.

Function atomic1_P in Figure 5 is the compiler-synthesized planning function for the transaction of Figure 1, assuming no partitioned operation inside initUser. Lines 6 and 12 of Figure 1 have been removed by PrunePlan, since they do not belong to any backward slicing of partitionable operation calls. More advanced alias analysis (e.g., data structure analysis[16]) could further remove line 7 of Figure 1 (line 8 of Figure 5).

4.2.2 Ensuring Consistency

As noted in Section 3.2, the planning phase of a partitioned operation, which is not automatically atomic, must be designed to ensure its own internal consistency, much like the implementation of an atomic block in a software TM system. Outside planning operations, which are responsible for

```
1  void atomic1_P(User* user, const char* strAddress, HashTable* users, RBTree* accounts) {
2    User *u = 0;
3    if ((u = htableFind_P(users, strAddress) != 0) {
4      int tmp0 = ParT_read_32(&u→accountNum);
5      for (int i=0; i< tmp0; i++) {
6        int tmp1 = ParT_read_32(&u→accts[i]);
7        Account *acct = rbtreeFind_P(accounts, tmp1);
8        ParT_write_32(&acct→lastAccessedTime, timeStamp);
9      }
10   } else {
11     u = (User*)malloc_P(sizeof(User));
12     htableInsert_P(users, strAddress, u);
13   }
14 }

16 // ParT transaction
17 if (setjmp(ParT_jmpbuf)==0) // planning phase
18   atomic1_P(user, strAddress, users, accounts);
19 ParT_plan_stop();
20 htm_begin(); // commit phase
21 User *u = 0;
22 if ((u = htableFind_C(users, strAddress) != 0) {
23   for (int i=0; i<u→accountNum; i++) {
24     Account *acct = rbtreeFind_C(accounts, u→accts[i]);
25     sum += acct→val;
26     acct→lastAccessedTime = timeStamp;
27   }
28 } else {
29   u = (User*)malloc_C(sizeof(User));
30   htableInsert_C(users, strAddress, u);
31   initUser(u, strAddress);
32 }
33 htm_end();
```

Figure 5: Compiler-generated ParT code (source-level equivalent) for the transaction of Figure 1.

their own consistency, function InstrumentPlan in Algorithm 1 employs STM-like instrumentation to buffer reads and writes in thread-private logs. The read log allows us to validate, in the wake of each new read, that all previously read locations still hold their original values. Planning operations, when necessary, can also insert some subset of their reads into the read log, to ensure that their return values remain consistent (an example appears in Figure 4, line 9). The write log allows glue code to "see its own writes." In contrast to STM, the write log is discarded by ParT_plan_stop when the planning phase naturally ends or an inconsistency occurs; all we need going forward are the validators created by the constituent planning operations.

4.2.3 Synthesizing Completion Code

The composed completion phase is comparatively easy to construct: it is almost the same as the code of the original transaction/function, except that each function call that was replaced by Algorithm 1 in the planning phase will be replaced by its completion function in the corresponding position in the completion phase. The generated ParT equivalent of the transaction in Figure 1

appears in Figure 5 (lines 17–33), where the the use of setjmp allows ParT_plan_stop to escape any nested context.

When validation of the plan for operation O fails within the completion phase of composed transaction T, fallback code will re-execute only operation O, automatically salvaging everything else in the transaction and continuing the completion phase. If the outer, hardware transaction of the completion phase aborts and retries, the fact that plans are constructed outside the transaction means that we will similarly salvage every plan whose validator still returns true. These properties essentially constitute a *partial rollback* mechanism, likely resulting in shorter turn-around time and higher throughput than would be available without partitioning.

A simple example can be seen in transactions that end with a reduction (e.g., the update of a global sum). Unlike a monolithic composed transaction, a ParT completion phase that aborts due to conflict on a reduction variable can generally salvage the planning phases of all constituent partitioned operations.

4.3 Run-time Support and Optimizations

The ParT run-time system manages the execution of composed planning. As described in previous sections, we maintain data consistency by keeping read and write logs for each planning phase, and performing incremental validation each time a new address is read and each time a planning operation requires consistency. The validation is value-based and employs a small hardware transaction, so no ownership records or locks are needed.

As in an STM system, read/write logging and validation impose nontrivial overheads. Read and write logs tend to be considerably smaller than in STM, however. Moreover, as discussed in Section 4, the planning phase, unlike a software transaction, does not have to execute to its end. These observations enable several optimizations:

Limiting the size of read/write logs. The cost of validation and instrumentation goes up with the size of the read/write logs. Because most shared reads/writes happen inside (uninstrumented) partitioned operations, the read/write set of the planning phase is usually small. To guard against pathological cases, we stop planning if a predefined limit is exceeded. Small size allows us to employ a fast and simple structure for the logs.

Merging partitioned operations. The more partitioned operations a planning phase contains, the more instrumentation and validation is required in the intervening glue code. We provide an interface for advanced programmers to reduce these overheads by merging the execution of several planning operations into one hardware transaction.

Switching between ParT and non-ParT transactions. Some transactions are not worth partitioning. A transaction with low contention and a small footprint, for example, is likely to succeed in HTM with few retries. At the opposite extreme, a transaction whose planning never works should always execute the monolithic alternative, even if HTM will fail and must fall back to STM or a global lock. Deciding which version is better—ParT or non-ParT—is difficult at coding time, as the answer may depend on the input, and may vary across execution phases. A dynamic switching strategy may make a better choice at run time. One possible strategy is to use a short period of time to profile non-ParT transaction abort rates, and then chose the atomic blocks for which to employ the partitioned code over some longer span of time. As ParT and non-ParT code can safely run together, the switching policy has little overhead. We implemented this adaptive policy in our compiler, but did not employ it

in the experiments of Section 5. Further experiments with adaptation are a subject for future work.

4.4 Limitations

While we are able to automate the composition of partitioned operations (and transactions that contain them), we require that the partitioned operations themselves already exist. The most significant limitation of our work is therefore the effort involved in constructing such operations. In future work we hope to develop tools to assist the library designer.

Because we end a planning phase when we encounter an unsafe function (Section 4.2), ParT will have little or no benefit when a call to such a function appears early in a transaction. Given the overhead of instrumentation on planning reads and writes, it is also possible for the cost of partitioning to outweigh the concurrency benefits, especially when contention is low. Likewise, if the work that remains in the completion phase still takes too long, or consumes too much memory, to fit in a hardware transaction, it is still possible that threads will serialize. (Even then, planning may help, if it is able to move work off of the serial path.) Despite these limitations, we have found ParT to be an effective means of enhancing HTM, as described in the following section.

5 Experimental Evaluation

Our experiments were conducted on two different HTM-capable machines—an IBM zEnterprise EC12 mainframe server (in a virtual machine with 16 dedicated cores) and a 4-core Intel Haswell Core i7-4470 machine. Both machines implement best-effect HTM and provide a similar ISA interface to the software. (The zEC12 also supports special "constrained" transactions that are guaranteed to complete successfully in hardware; these are not used in our experiments.) The transactional region is marked by a pair of begin/end instructions. The hardware guarantees *strong isolation* of the transactional execution. A transaction may abort for various reasons, including interrupts, restricted instructions, excessive nesting depth, capacity overflow, and conflicting accesses. Upon an abort, execution branches to a software handler. Intermittent aborts (e.g., fetch/store conflicts) may be worth retrying; persistent aborts (e.g., overflows) may not.

The **IBM zEnterprise EC12** [14] is a multi-socket machine. Each processor chip contains six single-threaded, out-of-order superscalar cores with a clock speed of 5.5 GHz. Each core has a private 96 KB 6-way associative L1 data cache and a private 1 MB 8-way associative L2 data cache, with 256 B cache lines. Cores on the same chip share a 48 MB L3 cache. The tag of each L1 line includes a bit to indicate membership in the read set of a transaction. When a line that was read in a transaction is evicted from the L1 cache, it is tracked by an LRU-extension vector instead. Since the L1 and L2 are both write through, proper handling of transactional writes requires changes to the *store cache*, a queue of 64 half-lines, which buffers and merges stores before sending them on to the L3 cache. To maintain isolation, hardware stalls departures from the store cache during transactional execution. Write set size is thus limited by the store cache size and the L2 cache size and associativity.

Our **Intel Core i7-4470** machine has a single processor with 4 SMT cores (8 hardware threads). Each core has a private 32 KB, 8-way associative L1 data cache and a private 256 KB, 8-way associative L2 cache, with 64 B cache lines. The 4 cores share an 8 MB L3 cache. The HTM system is implemented on top of an existing cache design [25]. Transactional data are tracked in the L1 data cache, at cache-line granularity. If a written line is evicted from the cache, the transaction will abort. Evicted reads are tracked in a secondary structure that supports a larger read set, at the risk of a higher rate of false conflicts.

Benchmark	Source	Description	#	Comp.
sorted DL list		insert/delete elements	2	N
rb-tree	[8]	insert/delete elements	3	N
equiv. sets	[23]	move elements between sets	1	N
account	Fig 1	account management	2	Y
genome	STAMP	gene sequencing	2	Y
intruder	STAMP	network intrusion detector	1	Y
vacation	STAMP	online travel reservation system	3	Y
UtilityMine	RMS-TM	utilization-based item sets mining	1	N
memcached	ver1.4.9	in-memory key value store	4	Y

Table 1: Summary of benchmarks. The "#" column is the static number of ParT transactions, "Comp." indicates if transactions comprise multiple partitioned operations.

5.1 TM Compiler and Runtime

On Haswell, we implemented ParT as an LLVM 3.3 optimization pass, taking the bitcode of the entire program as its input. On the zEC12, where we did not have access to production-quality LLVM support, benchmarks were hand-modified to mimic the LLVM output. (With a bit more effort, we could have engineered support into one of the z compilers: there is nothing inherently LLVM-specific about the partitioning algorithm.)

For software fallback, the TM run-time library uses a global test-and-test_and_set lock with exponential backoff. If a completion phase aborts for a non-persistent reason, we retry the transaction up to MAX_RETRIES times before switching to the lock. Otherwise, we retry up to MAX_PERS_RETRIES times, rather than immediately reverting to the lock, because the "persistent" diagnostic flag is only a hint, and a retry may succeed despite it. To avoid repeated aborts on the same conflict, we delay briefly before restarting a transaction. Hardware transactions in planning operations are handled in the same way, except that the whole parent planning phase will stop after a fixed number of retries.

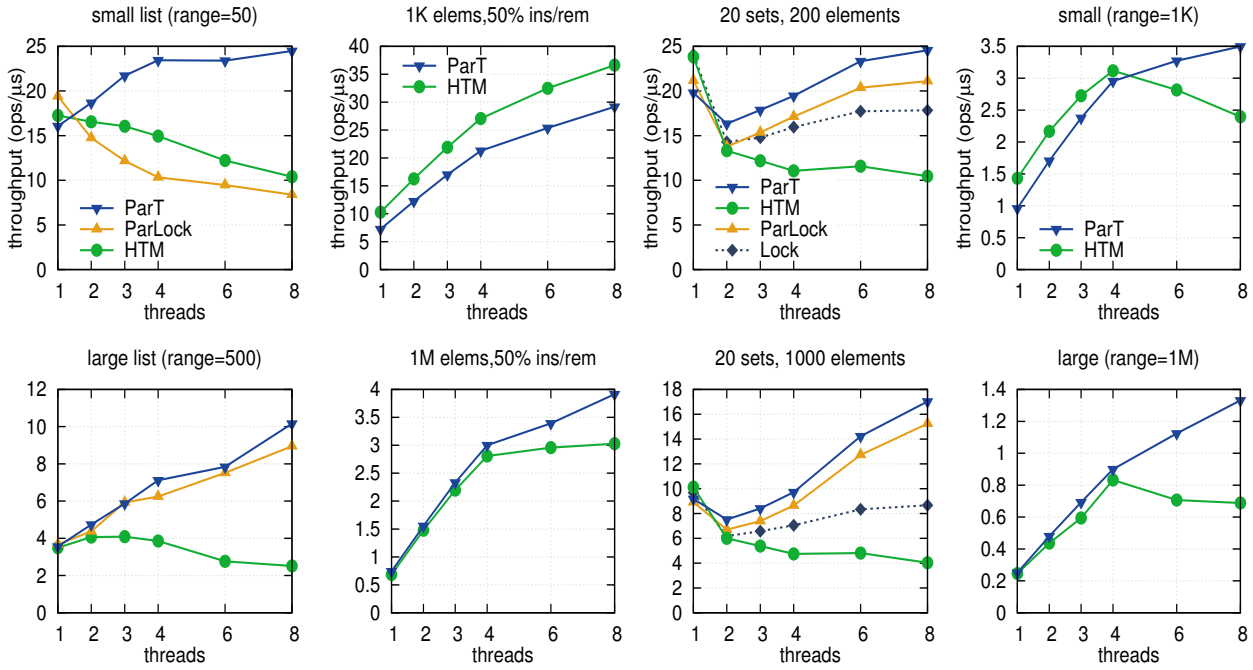
We set MAX_RETRIES and MAX_PERS_RETRIES to 8 and 5, respectively, on the zEC12, and to 10 and 3 on Haswell: these values delivered the best overall performance for the HTM baseline. Read and write logs were sized at 32 and 16 entries, respectively.

5.2 Benchmark Suite

We use four microbenchmarks and five larger applications to evaluate ParT, and to compare it to other techniques. Table 1 lists these benchmarks, including their provenance, a description of their major behaviors, and the number of ParT transactions in the source.

The microbenchmarks in the top half of the table all contain operations with a nontrivial search phase. In our tests, operations on the data structure are invoked as often as possible for a period of 1 second; our performance graphs plot throughput in operations per microsecond. In all microbenchmarks, we pre-populate thread-local free lists with enough data nodes to eliminate the impact of memory allocation.

The macrobenchmarks in the bottom half of Table 1, from the STAMP [19] and RMS-TM [15] benchmark suites, were chosen because their transactions are complex and good candidates for ParT optimization. All four were run with the recommended non-simulation inputs. The memcached macrobenchmark is a slightly modified version of memcached 1.4.9. Critical sections protected by three major locks (cache_lock, slabs_lock, and stat_lock) were replaced with hardware transactions.



(a) DL list (b) red-black tree (c) equivalence sets (d) account

Figure 6: Microbenchmark performance on Haswell. The Y axis indicates throughput.

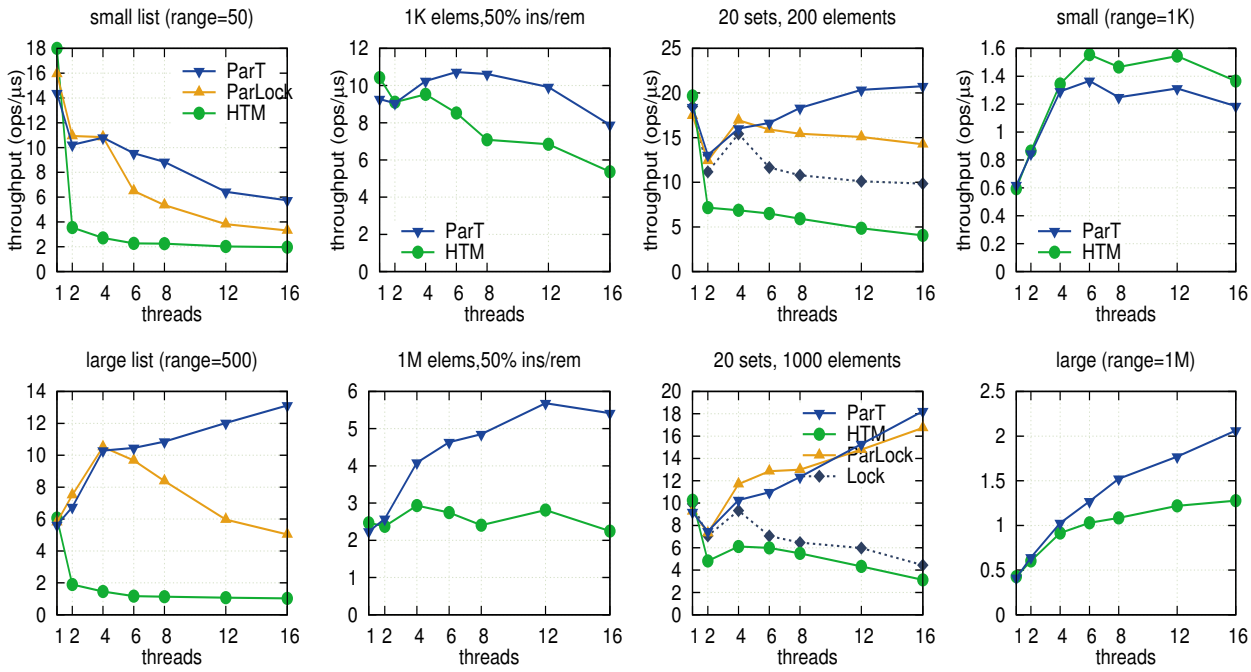
To stress the server, we bypass networking and inject input (dumped by memslap) directly into relevant functions. For each application, we report speedup over the sequential code. The STAMP benchmarks, as distributed, use malloc/free for the sequential version but a customized thread-local allocator for the TM version. We found the thread-local version to be significantly faster; for the sake of fair comparisons, we re-configured the sequential code to use this version instead.

Source code on the zEC12 was compiled with IBM's XL C/C++ compiler with –O3 –qstrict flags (memcached fails to compile due to missing system headers); on Haswell, we used LLVM 3.3 with –O3. Reported results are the average of 5 runs each, though no significant performance variation was observed.

5.3 Microbenchmark Results

Sorted Doubly-linked List Both insert and remove operations travel the list from its head node until an appropriate position is found for modification. The planning operations search, speculatively, for the position at which to insert/delete the key and save that position (a pointer to the last node whose key is less than the given one) in the validator. Unlike the HTM-based planning in a red-black tree, the planning phase here does not employ a hardware transaction. Where a pure HTM implementation might read every node in a list, the completion operations for insert and remove read only two. This significantly reduces HTM resource pressure and possible data conflicts.

Performance results for different implementations appear in Figures 6a and 7a. The three curves represent pure HTM, ParT, and a partitioned implementation in which the completion phase comprises a lock-based critical section rather than a transaction (ParLock, which is not composable). ParT is clearly the best among the three. With a shorter list, ParLock spends



(a) DL list (b) red-black tree (c) equivalence sets (d) account

Figure 7: Microbenchmark performance on the zEC12. The Y axis indicates throughput.

significant time on lock contention, which ParT avoids. Pure HTM must fall back to a global lock for many of its transactions, and doesn't do well at all.

Red-black tree The partition is described in Section 3.4. Two different data set sizes are used in the tests: in the smaller, keys are randomly selected from [0, 1K); in the larger, keys are randomly selected from [0, 1M). Throughput results appear in Figures 6b and 7b. In the smaller tree, HTM can finish most transactions without aborting, so the overhead of planning and validation is essentially wasted in ParT. In the larger tree, relative performance is reversed: pure HTM experiences many aborts, while partitioning shrinks the footprint of completion transactions enough for them to succeed much more often.

Equivalence Sets The equivalence set data structure comprises a collection of sorted linked lists, which partition a universe of elements; each operation moves a specified element from its current set to a given, specified set. Insertion in the new set requires a search of the list, which is done in the list's insert_P method in the ParT implementation.

Throughput for multiple implementations is shown in Figures 6c and 7c. The "Lock" curve uses per-set fine-grained locking. "ParLock" is a variant of ParT that uses similar locking (instead of a transaction) to protect the completion phase. All five curves dip at 2 cores, due to the cost of coherence misses. ParT scales better than fine-grained locking, which in turn outperforms the baseline HTM in this experiment: in the absence of partitioning, we have a "stress-test" workload, where speculation is rarely successful.

Account Management This synthetic benchmark updates account information stored in two shared data structures. The transaction for account insertion and update appears in Figure 1,

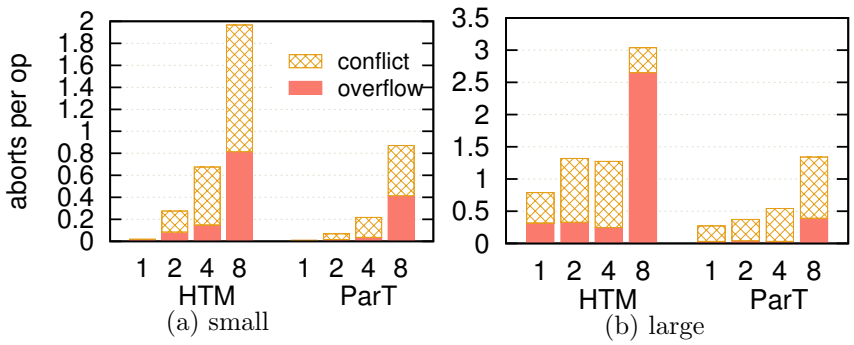


Figure 8: Profiling of abort reasons for account on Haswell.

where user information is indexed by addresses. The transaction for account deletion (not shown) removes a user and all linked accounts.

In our tests, address strings are 35 characters long and each user has 4 accounts on average. Figures 6d and 7d plot throughput for two data sets: the smaller one (more conflicts) contains 1K unique addresses; the larger one (more overflows) contains 1M unique addresses. Figure 8 shows the number of conflict and overflow aborts per completed operation on Haswell (aborts in the planning phases are included in "ParT" bars). In both cases, ParT significantly reduces both conflicts and overflows (by more than 50% in every workload) and thus results in better scalability, though pure HTM provides higher throughput when transactions are small and contention is low.

5.4 Macrobenchmark Results

Three of the standard STAMP benchmarks make heavy use of shared container objects. We built a ParT library containing partitioned versions of sorted linked list, hashtable, red-black tree, and memory allocation methods to enable automatic partitioning. Other than the addition of pragmas, no modifications were made to the source code of transactions.

Genome The first phase of the application removes duplicate gene segments. Segments are divided into chunks, and every chunk is transactionally inserted into a hash set. The hash set is non-resizable, so some buckets may contain hundreds of segments. In the original transaction, insertion of a segment could involve a long search phase to find the proper position in a sorted linked list. A data conflict in the insertion would invalidate all active insertions in the same transaction. ParT optimizes the transaction by moving all planning operations to a compound planning phase. By eliminating almost all load/store overflows and reducing load/store conflicts (Figures 10a and 12a), ParT leads to significantly better scalability on both machines (Figures 9a and 11a).

Intruder This application processes network packets in parallel—in capture, reassembly, and detection phases. Reassembly is the most complex phase. It uses a red-black tree to map each session id to a list of unassembled packets belonging to that session. If all packets from a session are present in the list, the list is removed from the tree and its packets are assembled as a complete stream and inserted back to a global queue, which is the principal locus of conflicts. The entire reassembly phase is enclosed in a transaction, which we use a #pragma part to optimize.

The planning operations for both tree search and list insertion have a low failure rate (2.2% and 2.6%, respectively, at 16 threads on the zEC12), meaning that if a conflict occurs on the global

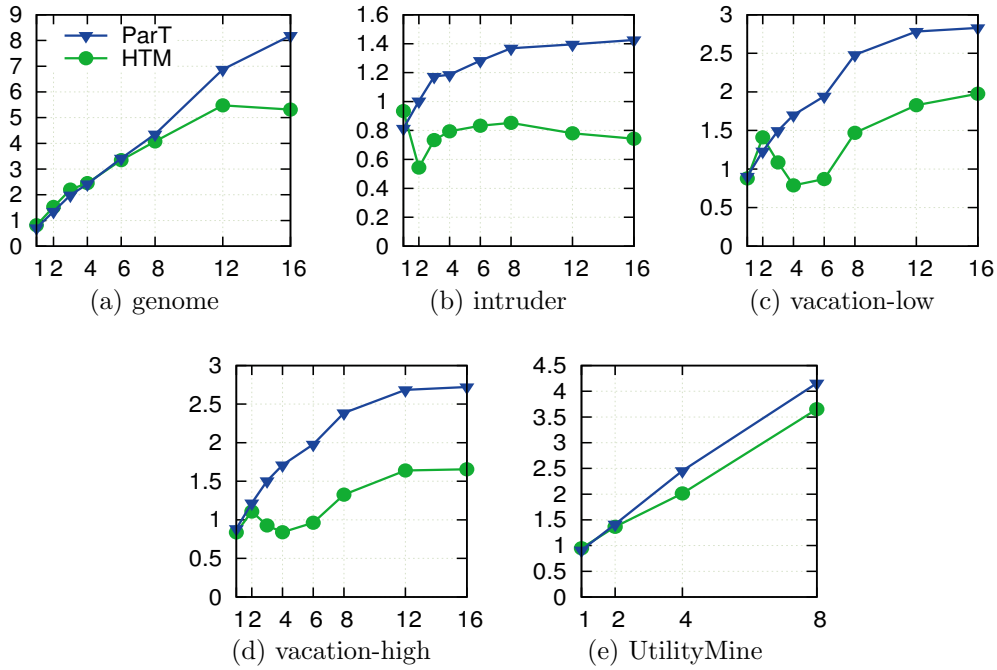


Figure 9: Macrobenchmark performance on the zEC12. The Y axis indicates speedup over the sequential version.

queue, in the next re-execution, ParT can skip tree and list search. This dramatically reduces the duration and footprint of re-executed transactions. ParT almost doubles the throughput of the program (Figure 9b) on the zEC12. On Haswell, ParT starts to outperform the baseline when hyperthreading is used.

Vacation This application manages reservations for cars, flights, and rooms in a database implemented as a set of red-black trees. Each transaction creates, cancels, or updates a reservation in the database. Creation consumes the largest share of execution time. ParT optimization is straightforward, but the compiler must process multiple levels of plan functions to reach the tree operations.

On the zEC12 (Figures 9c and 9d), ParT lags slightly behind the original implementation on 1 or 2 threads. It outperforms the original on 3 or more threads, however. One improvement comes from the elimination of overflows, which often happen near the end of a big transaction and are quite expensive. Interestingly, as shown in Figures 10c and 10d, ParT increases load conflicts. At the same time, because the completion transactions are so much shorter than in the original code, and most of the planning can be preserved on retry, less work is wasted by each abort, and overall throughput still improves. On Haswell, ParT eliminates most aborts of the composed commit phase (Figures 12c and 12d) and therefore brings significant performance improvement.

UtilityMine This application spends most of its time in a transaction that updates the "utility" of items according to data read from a database file. The main transaction scans a utility array, whose size is input dependent, to locate a specific item. If the item is found, its utility is increased; otherwise, a new item is inserted. We optimize this transaction by replacing the most common path, in which the item is found, with a function call, and using #pragma plan_for amd #pragma

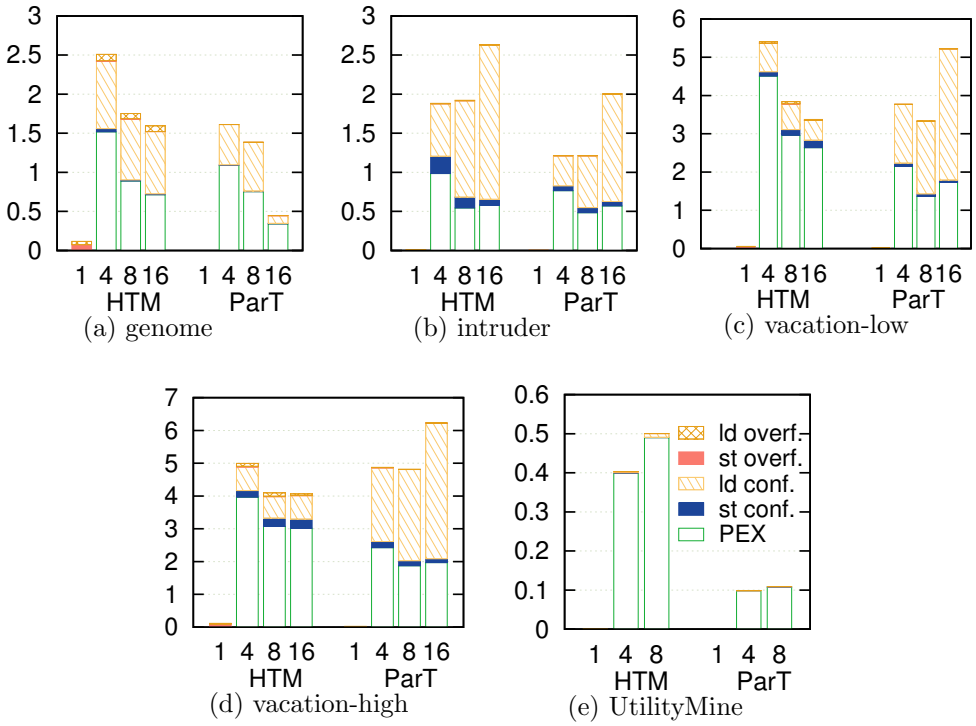


Figure 10: Aborts per transaction for TM macrobenchmarks (completion transactions only) on the zEC12.

complete_for to build an ad-hoc partitioned operation. In the planning function, the position of the found item is saved in a validator, whose is_valid method confirms that the array has not been deallocated, and the item is still in the same position. In addition to reducing transaction duration and footprint, partitioning allows us to issue prefetch instructions on the zEC12; these avoid a "promote-to-exclusive" (PEX) case in the coherence protocol, which can sometimes lead to spurious aborts.

Only 1, 2, 4, and 8-thread configurations are available for this benchmark. Executing with the standard input sets, transactions rarely overflow, so the benefit of ParT, as shown in Figure 10e, comes mainly from a reduction in the low but still significant number of PEX aborts (note the modest scale of the y axis). Running with larger input sets, data overflow could emerge as a major issue in the baseline case, while ParT would still be ok. In general, ParT can reduce the chance of sudden performance anomalies with changes in program input.

Memcached This widely used server application stores key/value pairs in a global hash table and in an auxiliary table used for resizing operations. In the presence of concurrency, the table structure itself does not incur many conflicts, but transactions also access the table's size field and global statistics information, increasing the chance of contention. We partitioned three major table access functions (assoc_find, assoc_insert, and assoc_delete) using version numbers for validation [23]. Transactions that call these functions, directly or indirectly, are then partitioned by the compiler.

Another second major source of aborts arises in the function do_item_alloc, which begins with a relatively long search for a possibly expired item in a list of slabs, in the hope of performing fast space reuse instead of slow allocation. As this particular search is commutative, we do it in the function's planning operation, transactionally. As shown in Figure 12f, by pulling table and

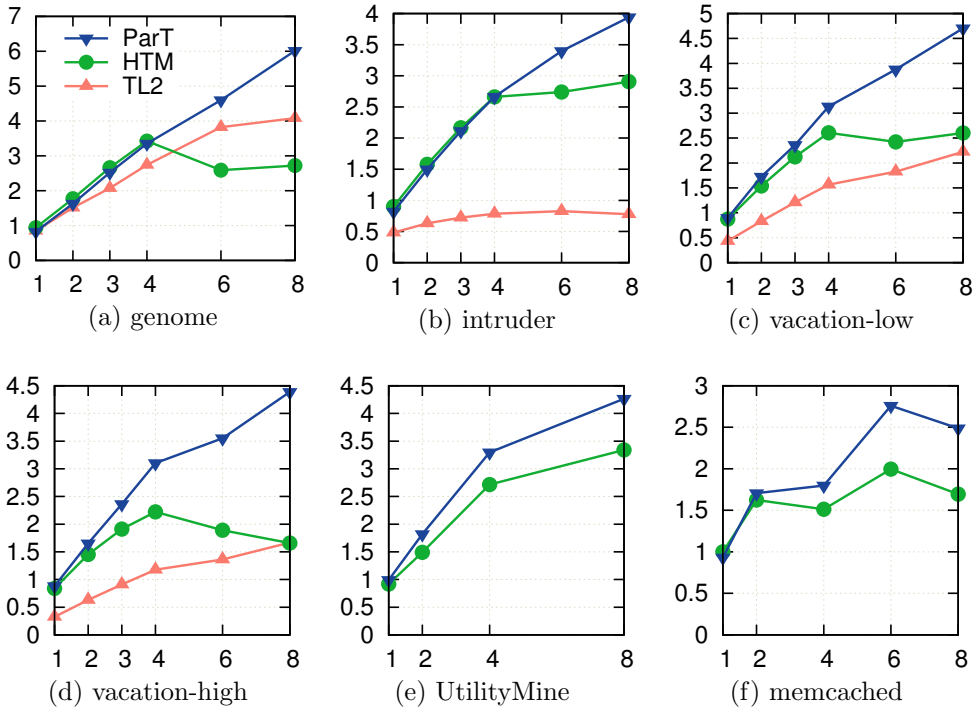


Figure 11: Macrobenchmark performance on Haswell. The Y axis indicates speedup over the sequential version.

list searches out of transactions, ParT significantly reduces the incidence of overflow aborts. Also, as transactions become shorter, the conflict window is narrowed, a phenomenon that is further enhanced by ParT's partial rollback. As a result, ParT improves performance by roughly a third at 8 threads (Figure 11f).

6 Related Work

ParT draws inspiration from our lock-based MSpec/CSpec work [23]. There the goal was to reduce the size of critical sections by removing work that could be executed speculatively in advance—ideally with the aid of a compiler that deals with mechanical issues of code cloning and data race prevention. In MSpec/CSpec, pre-speculation serves to shorten the application's critical path; with partitioned transactions it serves to reduce transaction conflicts. Both systems share the benefit of cache warmup. ParT adds the benefits of reduced cache footprint, composability, and partial rollback.

The manual partitioning of operations in ParT also resembles the *consistency oblivious programming* (COP) of Avni et al. [1, 2], in which a search-then-modify operation is divided into a non-atomic search followed by atomic validation and modification. Avni and Suissa have extended COP to accommodate composition [3], but mainly for software TM. In an HTM system, their technique would require special instructions to suspend and resume a transaction. This would seem to preclude the use of additional transactions during planning—something that ParT uses freely. Planning during suspension would also do nothing to shorten the temporal duration of the main transaction.

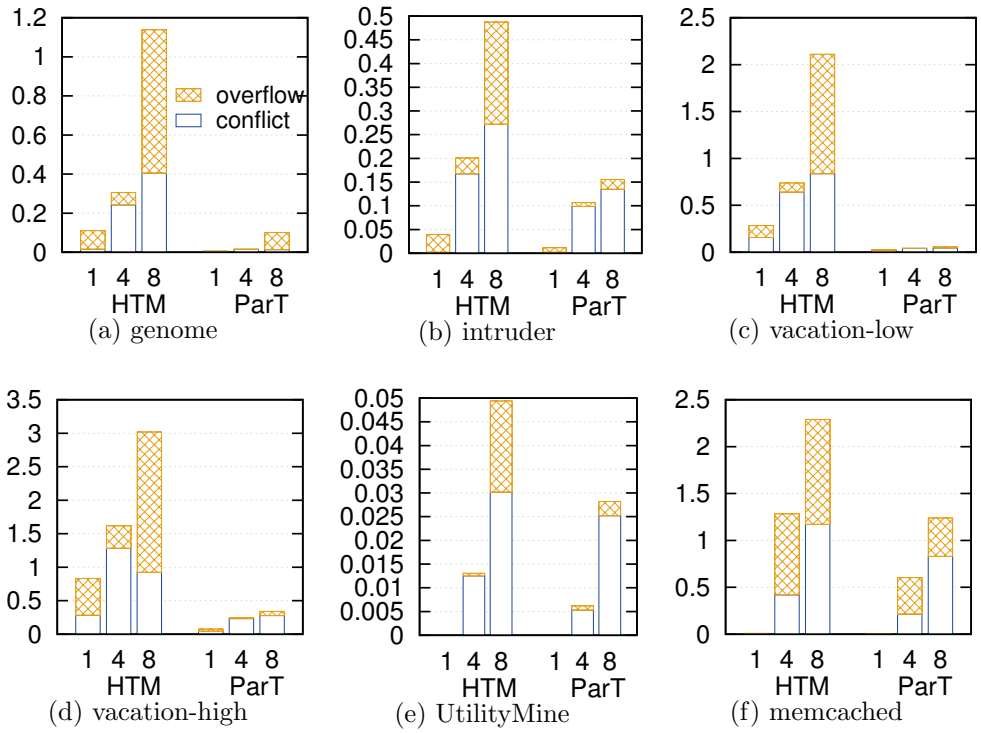


Figure 12: Aborts per transaction for TM macrobenchmarks (completion transactions only) on Haswell.

For search-based operations, partitioned transactions bear a certain resemblance to *early release* [12] and *elastic transactions* [9], both of which allow a (software) transaction to remove no-longer-needed locations from its read set. From the perspective of usability, however, we believe that programmers will find it easier to specify what they *do* need than to specify what they don't.

In comparison to all these alternatives, ParT has the advantage of working with existing HTM; early release, elastic transactions, and composed COP would all require new hardware instructions, and would introduce thorny issues of false sharing within cache lines. Finally, only ParT allows partial rollback of an aborted transaction: in the present of contention, valid planning work can be leveraged during retry. Table 2 summarizes these comparisons.

In mechanism—if not in purpose—partitioned transactions also resemble the *split hardware transactions* of Lev et al. [17]. In that work, a transaction is divided into two or more *segments*,

		ParT	CSpec[23]	ElasT[9]	COP[2]	COP-c[3]
application		wide	limited	limited	limited	limited
support composition?		Yes	No	No	No	Yes
require special HTM features?		No	No	early release	No	suspend/ resume
benefits	smaller footprint	Yes	Yes	Yes	Yes	Yes
	shorter duration	Yes	Yes	No	Yes	No
	partial rollback	Yes	No	No	No	No

Table 2: Comparison of HTM programming techniques

each of which logs its reads and writes (in software) and passes the logs to the following segment. A segment that needs to be consistent with its predecessors begins by re-reading the locations in their read logs. The final segment re-reads everything, and then performs the updates in the write log. While log maintenance incurs a nontrivial penalty, performance is still better than with STM, because conflict detection remains in hardware.

The principal goal of split hardware transactions is to support true closed and open nesting, in which an inner transaction can abort without aborting (the prefix of) its parent transaction, or commit even if the parent aborts. Lev et al. suggest that the mechanism could also be used for debugging and ordered speculation (loop parallelization). In contrast, the principal goal of partitioned transactions is to increase scalability by exploiting programmer knowledge of high-level program semantics. Rather than mechanically transfer read and write logs from one segment to the next, we transfer only as much information as we need to validate the planning phase and perform the commit phase. Low-level partitioned operations must of course be implemented manually by the library designer, but once they exist, the compiler can automate the rest, and we can expect decreases in abort rate that are unlikely with split transactions, where read and write logs are likely to induce a net increase in cache footprint.

Other connections are more tenuous. Transactional boosting [11] and transactional predication [4] exploit high-level semantic information to reduce the cost of nested operations in a software TM system, but we see no way to employ them with HTM. The Foresight mechanism [10] facilitates composition, but for conservative, lock-based systems. True closed nesting [20] offers the possibility of partial rollback, but again it is not supported by current HTM.

7 Conclusions

As hardware transactional memory becomes more widely used, programmers will need techniques to enhance its performance. We have presented one such technique: partitioned hardware transactions (ParT). The key idea is to extract the read-mostly *planning* portions of common operations and to execute them—either in ordinary software or in smaller transactions—before executing the remaining *completion* transaction. To ensure atomicity, a *validator* object carries information across planning operations and into the corresponding completion operation, allowing the latter to confirm, quickly, that the planning work is still valid. Automatic compiler support allows partitioned operations—and transactions that include them—to compose as easily and safely as traditional monolithic transactions, with no special hardware support.

We tested ParT on both the IBM zEnterprise EC12 (currently the most scalable HTM-capable architecture) and a smaller Intel Haswell machine. Using a variety of examples, including three micro- and five macrobenchmarks, we demonstrated that ParT can yield dramatic performance improvements—often making the difference between scalable and nonscalable behavior.

We conclude that ParT is a valuable addition to the "TM programmer's toolkit." Topics for future work include integration with software and hybrid TM; compiler support for nontransactional planning phases, in the style of CSpec [23]; and dynamic choice of fallback strategies based on runtime statistics.

References

[1] Y. Afek, H. Avni, and N. Shavit. Towards consistency oblivious programming. In *Proc. of the 15th Intl. Conf. on Principles of Distributed Systems (OPODIS)*, Toulouse, France, Dec. 2011.

[2] H. Avni and B. Kuszmaul. Improve HTM scaling with consistency-oblivious programming. In *9th SIGPLAN Wkshp. on Transactional Computing (TRANSACT)*, Salt Lake City, UT, Mar. 2014.

[3] H. Avni and A. Suissa. TM-pure in GCC compiler allows consistency oblivious composition. In *Joint Euro-TM/MEDIAN Wkshp. on Dependable Multicore and Transactional Memory Systems (DMTM)*, Vienna Austria, Jan. 2014.

[4] N. G. Bronson, J. Casper, H. Chafi, and K. Olukotun. Transactional predication: High-performance concurrent sets and maps for STM. In *29th ACM Symp. on Principles of Distributed Computing (PODC)*, Zurich, Switzerland, July 2010.

[5] H. W. Cain, M. M. Michael, B. Frey, C. May, D. Williams, and H. Le. Robust architectural support for transactional memory in the Power architecture. In *40th Intl. Symp. on Computer Architecture (ISCA)*, Tel Aviv, Israel, June 2013.

[6] C. Click Jr. And now some hardware transactional memory comments. Author's Blog, Azul Systems, Feb. 2009. `www.azulsystems.com/blog/cliff/2009-02-25-and-now-some-hardware-transactional-memory-comments`.

[7] D. Dice, Y. Lev, M. Moir, and D. Nussbaum. Early experience with a commercial hardware transactional memory implementation. In *14th Intl. Conf. on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, Washington, DC, Mar. 2009.

[8] D. Dice, O. Shalev, and N. Shavit. Transactional Locking II. In *20th Intl. Conf. on Distributed Computing (DISC)*, Stockholm, Sweden, Sept. 2006.

[9] P. Felber, V. Gramoli, and R. Guerraoui. Elastic transactions. In *23rd Intl. Conf. on Distributed Computing (DISC)*, Elche/Elx, Spain, Sept. 2009.

[10] G. Golan-Gueta, G. Ramalingam, M. Sagiv, and E. Yahav. Concurrent libraries with Foresight. In *34th SIGPLAN Conf. on Programming Language Design and Implementation (PLDI)*, Seattle, WA, June 2013.

[11] M. Herlihy and E. Koskinen. Transactional boosting: a methodology for highly-concurrent transactional objects. In *13th SIGPLAN Symp. on Principles and Practice of Parallel Programming (PPoPP)*, Salt Lake City, UT, Feb. 2008.

[12] M. Herlihy, V. Luchangco, M. Moir, and W. N. Scherer, III. Software transactional memory for dynamic-sized data structures. In *22nd ACM Symp. on Principles of Distributed Computing (PODC)*, Boston, MA, July 2003.

[13] M. Herlihy and J. E. B. Moss. Transactional memory: Architectural support for lock-free data structures. In *20th Intl. Symp. on Computer Architecture (ISCA)*, San Diego, CA, May 1993.

[14] C. Jacobi, T. Slegel, and D. Greiner. Transactional memory architecture and implementation for IBM System z. In *45th Intl. Symp. on Microarchitecture (MICRO)*, Vancouver, BC, Canada, Dec. 2012.

[15] G. Kestor, V. Karakostas, O. S. Unsal, A. Cristal, I. Hur, and M. Valero. RMS-TM: A comprehensive benchmark suite for transactional memory systems. In *2nd ACM/SPEC Intl. Conf. on Performance Engineering (ICPE)*, Karlsruhe, Germany, Mar. 2011.

[16] C. Lattner and V. Adve. Automatic pool allocation: Improving performance by controlling data structure layout in the heap. In *26th SIGPLAN Conf. on Programming Language Design and Implementation (PLDI)*, Chicago, IL, June 2005.

[17] Y. Lev and J.-W. Maessen. Split hardware transactions: True nesting of transactions using best-effort hardware transactional memory. In *13th SIGPLAN Symp. on Principles and Practice of Parallel Programming (PPoPP)*, Salt Lake City, UT, Feb. 2008.

[18] M. M. Michael and M. L. Scott. Simple, fast, and practical non-blocking and blocking concurrent queue algorithms. In *15th ACM Symp. on Principles of Distributed Computing (PODC)*, Philadelphia, PA, May 1996.

[19] C. C. Minh, J. Chung, C. Kozyrakis, and K. Olukotun. STAMP: Stanford transactional applications for multi-processing. In *IEEE Intl. Symp. on Workload Characterization (IISWC)*, Seattle, WA, Sept. 2008.

[20] J. E. B. Moss and A. L. Hosking. Nested transactional memory: Model and architecture sketches. *Science of Computer Programming*, 63(2):186–201, Dec. 2006.

[21] A. Wang, M. Gaudet, P. Wu, J. N. Amaral, M. Ohmacht, C. Barton, R. Silvera, and M. Michael. Evaluation of Blue Gene/Q hardware support for transactional memories. In *21st Intl. Conf. on Parallel Architectures and Compilation Techniques (PACT)*, Minneapolis, MN, Sept. 2012.

[22] L. Xiang and M. L. Scott. MSpec: A design pattern for concurrent data structures. In *7th SIGPLAN Wkshp. on Transactional Computing (TRANSACT)*, New Orleans, LA, Feb. 2012.

[23] L. Xiang and M. L. Scott. Compiler aided manual speculation for high performance concurrent data structures. In *18th SIGPLAN Symp. on Principles and Practice of Parallel Programming (PPoPP)*, Shenzhen, China, Feb. 2013.

[24] L. Xiang and M. L. Scott. Composable partitioned transactions. In *5th Wkshp. on the Theory of Transactional Memory (WTTM)*, Jerusalem, Israel, Oct. 2013.

[25] R. M. Yoo, C. J. Hughes, K. Lai, and R. Rajwar. Performance evaluation of Intel transactional synchronization extensions for high-performance computing. In *Intl. Conf. on High Performance Computing, Networking, Storage and Analysis (SC13)*, Denver, CO, Nov. 2013.