STM in the Small: Trading Generality for Performance in Software Transactional Memory

Aleksandar Dragojević
I&C, EPFL, Lausanne, Switzerland
aleksandar.dragojevic@epfl.ch

Tim Harris
Microsoft Research, Cambridge
tharris@microsoft.com

Abstract

Data structures implemented using software transactional memory (STM) have a reputation for being much slower than data structures implemented directly from low-level primitives such as atomic compare-and-swap (CAS). In this paper we present a specialized STM system (SpecTM) that allows the program to express additional knowledge about the particular operations being performed by transactions—e.g., using a separate API to write transactions that access small, fixed, numbers of memory locations. We show that data structures implemented using SpecTM offer essentially the same performance and scalability as implementations built directly from CAS. We present results using hash tables and skip lists on machines with up to 8 sockets and up to 128 hardware threads. Specialized transactions can be mixed with normal transactions, allowing fast-path operations to be specialized for greater performance, while allowing less common cases to be expressed using normal transactions for simplicity. We believe that SpecTM provides a "sweet spot" for expert programmers developing scalable data structures.

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

Keywords Lock-Free Data Structures, Parallel Programming, Transactional Memory

1. Introduction

Over recent years a great deal of attention has been paid to the design and implementation of transactional memory systems. Transactional memory (TM) allows a program to make a series of memory accesses appear to occur as a single atomic operation [16, 17, 29].

EuroSys'12, April 10–13, 2012, Bern, Switzerland.

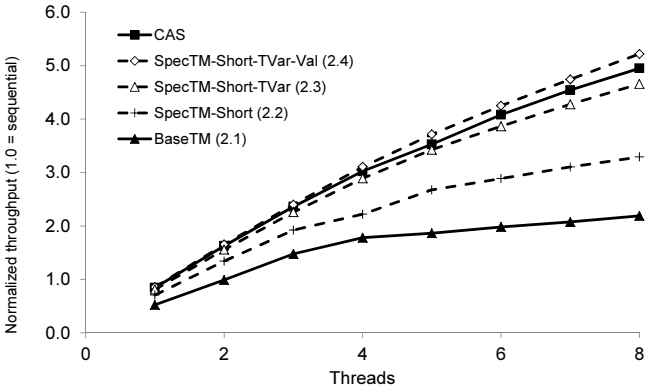


Figure 1. Throughput of operations on a hash table (90% lookups), normalized to optimized sequential code.

Atomicity can simplify the design of shared memory data structures. Unfortunately, data structures rarely perform well when built using software implementations of transactional memory (STM) [2, 8]. Figure 1 illustrates this for a hash table benchmark (in Section 2 we return to the details of the particular experimental setting). The figure shows the throughput of operations for a hash table built using a state-of-the-art STM system (labeled BaseTM), and an implementation built directly from atomic compare-and-swap (labeled CAS). With a single thread the performance of the baseline STM system is less than half that of sequential code. The CAS-based implementation is both faster than the STM-based implementation, and it scales better.

STM and CAS can be seen as two points in a spectrum of programming abstractions. On the one hand, STM can be relatively easy to use, but perform relatively poorly. On the other hand, CAS can be extremely difficult to use, but can perform extremely well. In this paper we examine intermediate points in this spectrum. Compared with STM, how much simplicity do we lose in order to achieve good performance? Compared with CAS, how much performance do we lose in order to make programming appreciably simpler?

Our main contribution is to show that data structures built using specialized STM systems can match the performance and scalability of existing CAS-based data structures. We

call our system SpecTM. We explore three kinds of specialization (Section 2):

First, SpecTM provides specialized APIs for short transactions. These APIs shift responsibilities from the STM's implementer to the STM's user: for instance, the STM's user must provide sequence numbers to their data accesses (one function for the first read in a transaction, a different function for the second, etc.). This shift in responsibility removes book-keeping work from the STM implementation.

Second, SpecTM provides specialization over the placement of STM meta-data, allowing pieces of meta-data to be located on the same cache lines as the application's data structures. In contrast, traditional STM systems often use an automatic hash-based mapping from data to meta-data, requiring multiple cache lines to be accessed.

Third, SpecTM provides a value-based validation mechanism in which the per-word meta-data is reduced to a single bit reserved in each data item. Requiring a "spare" bit for the use of the STM system restricts the kind of data that the STM can work over. However, on a 64-bit machine, the remaining 63 bits can nevertheless accommodate typical integer values, and pointers to aligned data structures. We show how, for many workloads, we can use value-based validation without needing to maintain shared global version numbers.

Importantly, specialized transactions can be mixed with normal transactions that use a traditional STM interface. This enables SpecTM to be used in two ways. First, specialized transactions can be used to build a new data structure, with the knowledge that long transactions are available as a fall-back for any cases that are difficult to write. Second, a data structure can be built using ordinary transactions, and then SpecTM can be used to optimize the common cases. Section 3 illustrates this using a detailed example. We show how SpecTM is used in building a skip list, using specialized transactions for the most common forms of insertion and deletion, and normal transactions for more complex cases.

The kind of specialized interface we provide in SpecTM is not appropriate for all settings. We are explicitly giving up some of the generality of TM as a synchronization mechanism in order to achieve better performance than existing implementations. In effect, we are returning to the original application of TM in building shared memory data structures, as espoused by Herlihy and Moss [17], and explored in the first STM design [29]. Another way to view SpecTM is that it provides a little bit more than early implementations of multi-word compare-and-swap (CASN) [13, 22]. Unlike CASN, SpecTM transactions are dynamic (allowing a transaction to atomically read a series of locations, rather than specifying all of the locations in a single function call). In addition, unlike early CASN implementations, SpecTM transactions can be mixed with traditional transactions.

Section 4 describes the implementation of SpecTM, and evaluates its performance and scalability. In this paper we focus on building data structures such as the hash tables and skip lists. The motivation for this focus is the central role of these data structures in key-value stores and in-memory database indices. The evaluation uses two systems: a 16-core machine and a larger 128-core machine. Our experiments show that specialized STM performs almost as well as existing lock-free algorithms, with slowdowns of less than 5%. Also, specialized STM scales as well as the lock-free algorithms on the 128-core machine even when the contention is high.

We describe related work (Section 5) and conclude (Section 6).

2. Specializing the TM Interface

In this section we introduce the interfaces exposed by SpecTM. We illustrate the use of SpecTM using the running example of a simple double-ended queue (dequeue). This queue supports `PushLeft`, `PopLeft`, `PushRight`, `PopRight` operations to add and remove items to either end. We focus on `PopLeft` and sketch how it would be written using a traditional STM (Section 2.1), and then how it would be written using SpecTM (Sections 2.2–2.4). (We use a basic dequeue as a concise example: a scalable implementation would be more complex [18].)

To give a feeling for SpecTM's performance, Figure 1 shows how some variants scale on 1–16 cores. The baseline TM system (BaseTM) follows the TL2 algorithm of Dice *et al.* [4], extended with timebase extension from Riegel *et al.* [27], and the hash-based write-set design of Spear *et al.* [30]. The CAS-based hash table is implemented from Fraser's design [11]. All the implementations use epoch based memory management, also following Fraser's design.

Building on BaseTM, we discuss the contributions of each form of specialization in the corresponding sections below (Section 2.2–2.4). In Section 4 our main results consider more combinations of design choices, and they examine scalability on larger machines.

2.1 Traditional STM (BaseTM)

Here is the `PopLeft` operation implemented using the traditional STM interface exposed by BaseTM:

```
void *Items[QUEUE_SIZE] = { NULL };
int LeftIdx = 0;
int RightIdx = 0;

void *PopLeft(void) {
  void *result = NULL;
  TX_RECORD t;
  do {
    Tx_Start(&t);
    int li = Tx_Read(&t, &LeftIdx);
    void *result = Tx_Read(&t, &Items[li]);
    if (result != NULL) {
      Tx_Write(&t, &(Items[li]), NULL);
      Tx_Write(&t, &LeftIdx, (li+1)%QUEUE_SIZE);
    }
  } while (!Tx_Commit(&t));
  return result;
}
```

The queue is built over an array which holds the items at indexes `LeftIdx`–`RightIdx-1` (wrapping around the

array, modulo the queue size). Queue elements must be non-NULL, allowing NULL values to be used to indicate the presence of empty slots (and to distinguish a completely empty queue from a completely full queue). Consequently, PopLeft starts by reading the left index, and reading the array item at that index. If the item is non-NULL, then the left index is advanced, and the array slot is cleared.

Compared with a sequential implementation, the STM adds three main costs: (*i*) book-keeping required when starting a transaction (e.g., recording a snapshot of the processor state so that the transaction can be restarted upon conflict), (*ii*) managing the transaction record on each read and write (e.g., incrementing read/write pointers into the transaction's logs, and, in the case of reads, ensuring that they see any earlier writes in the same transaction), (*iii*) visiting meta-data locations to perform concurrency control.

Our interface for short transactions (Section 2.2) addresses the first two of these costs. The third cost is addressed by our interface for explicit transactional data (Section 2.3), and for integrating meta-data within application data structures (Section 2.4).

2.2 Short Transactions

The key ideas for SpecTM's interface are to require the programmer to indicate the sequencing of operations within a transaction, and to require the programmer to avoid write-to-read dependencies within a transaction. Here is PopLeft re-implemented using SpecTM:

```
void *PopLeft(void) {
  void *result = NULL;
  TX_RECORD t;
restart:
  int li = Tx_RW_R1(&t, &LeftIdx);
  void *result = Tx_RW_R2(&t, &Items[li]);
  if (!Tx_RW_2_Is_Valid(&t)) goto restart;
  if (result != NULL) {
    Tx_RW_2_Commit(&t, (li+1) % QUEUE_SIZE, NULL);
  } else {
    Tx_RW_2_Abort(&t);
  }
  return result;
}
```

Compared with the previous section, there are five main changes: (*i*) The read operations include sequence numbers in the function signature (_R1 for the first read, _R2 for the second, etc.). The first read implicitly starts the transaction. (*ii*) Transactions can access only a small number of locations (four in our implementation, which can be increased in a straightforward manner). (*iii*) Each access must be to a distinct memory location. (*iv*) The processor state is not saved implicitly at the start of a transaction and restored upon conflict; the programmer is responsible for calling ..._Is_Valid to detect conflicts, and for restarting the transaction if needed. (*v*) The commit function signature includes the total number of locations accessed, along with the new values to be stored at each of them.

This example illustrates the trade-off we are studying. The specialized interface requires the programmer to be able to provide sequence numbers on their accesses (and so it would be ill-suited to a series of reads performed in a loop). In addition, SpecTM requires all of the writes to be deferred until commit-time. Imposing these restrictions reduces the book-keeping required by the STM. To see why, consider the operation of a typical STM system using deferred updates [16] (similar optimizations are possible when compared with STM systems using eager updates, but we omit the details for brevity). With deferred updates, the STM must log the new values written by a transaction, and copy them to target memory locations only upon successful commit. This logging means that transactional reads need to search the update log so that they see early writes by the same transaction. SpecTM's restrictions mean that: (*i*) There is no need for an update log, because the values being written are provided at commit-time. (*ii*) For the same reason, read-after-write checks are no longer necessary. (*iii*) For read-write transactions, the implementation can eagerly acquire a write lock at the time of the read, eliminating the need for commit-time read-set validation. (*iv*) Focusing on short transactions means that the set of all locations accessed can be held in a fixed-size array inline in the TX_RECORD. Similarly, there is no need to track operation indices, as they are provided by statically the program. This is in contrast to traditional STM systems which must track indices dynamically.

```
typedef void *Ptr;
// Single read/write/CAS transactions:
Ptr Tx_Single_Read(Ptr *addr);
void Tx_Single_Write(Ptr *addr, Ptr newVal);
Ptr Tx_Single_CAS(Ptr *addr, Ptr oldVal, Ptr newVal);

// Read-write short transactions:
Ptr Tx_RW_R1(TX_RECORD *t, Ptr *addr_1);
Ptr Tx_RW_R2(TX_RECORD *t, Ptr *addr_2);
...
bool Tx_RW_1_Is_Valid(TX_RECORD *t);
bool Tx_RW_2_Is_Valid(TX_RECORD *t);
...
void Tx_RW_1_Commit(TX_RECORD *t, Ptr val1);
void Tx_RW_2_Commit(TX_RECORD *t,
                    Ptr val_1, Ptr val_2);
...
void Tx_RW_1_Abort(TX_RECORD *t);
void Tx_RW_2_Abort(TX_RECORD *t);
...
// Read-only short transactions:
Ptr Tx_RO_R1(TX_RECORD *t, Ptr *addr_1);
Ptr Tx_RO_R2(TX_RECORD *t, Ptr *addr_2);
...
bool Tx_RO_1_Is_Valid(TX_RECORD *t);
bool Tx_RO_2_Is_Valid(TX_RECORD *t);
...
// Commit combined read-only & read-write transactions:
bool Tx_RO_1_RW_1_Commit(TX_RECORD *t, Ptr val1);
bool Tx_RO_1_RW_2_Commit(TX_RECORD *t,
                         Ptr val_1, Ptr val_2);
...
// Upgrade a location from RO to RW:
bool Tx_Upgrade_RO_1_To_RW_2(TX_RECORD *t);
...
```

Figure 2. SpecTM API for short transactions.

Figure 2 shows the full API for writing short transactions. In addition to the short read/write transactions that occur in this example, SpecTM provides:

Single-operation transactions. The `Tx_Single_*` functions perform transactions that access a single location: either read, write, or compare-and-swap. These operations synchronize with concurrent transactions—e.g., a `Tx_Single_Read` will not read uncommitted writes. More generally, the `Tx_Single_*` operations are linearizable [19] and so if read r_1 sees a value written by a transaction Tx_A then a subsequent read r_2 must see all Tx_A's writes. The single-operation API enables further optimizations, completely eliminating any logging to an explicit transaction record.

Short read-only transactions. As with the RW transaction in the `PopLeft` example, SpecTM provides short read-only transactions. We distinguish read-only transactions from read-write transactions so that an implementation can handle the two kinds of transaction differently (e.g., using encounter-time locking for locations in a short read-write transaction, but invisible reads for read-only transactions [16]).

A read-only transaction is started with a call to `Tx_RO_R1`, which also performs the first read. Subsequent reads are performed using `Tx_RO_R2`, ... The `Tx_RO_*_Is_Valid` functions test whether or not the transaction is currently valid, with variants specialized to the size of the read set. There are no explicit commit or abort functions. Successful validation serves in the place of commit. If a program wishes to abort a read-only transaction then it can simply discard the transaction record.

As with read-write transactions, short read-only transactions avoid the need to dynamically allocate logs and maintain log indices.

Combining read-only and read-write transactions. A single transaction may mix the `Tx_RO_*` operations for the locations that it only reads, and the `Tx_RW_*` operations for the locations that it both reads and writes. The two sets of locations must be disjoint. A set of commit functions with names such as `Tx_RO_`x`_RW_`y`_Commit` is provided to commit these transactions: x refers to the number of locations read, and y to the number of locations written. As with the `Tx_RW_*_Commit` functions, the values to write are supplied to the commit operation.

Finally, if a transaction wishes to "upgrade" a location from read-only access to read-write access, then the function `Tx_Upgrade_RO_`x`_To_RW_`y function indicates that index x amongst the transaction's existing reads has been upgraded to form index y in its writes—x may be any of the locations read previously, and y must be the next write index.

This form of upgrade is used when the value read from one location determines whether or not another location will be updated. For instance, consider a double-compare-single-swap operation which checks that two locations `a1` and `a2` hold expected values `o1` and `o2` respectively and, if they do, writes `n1` to `a1`. This function can be written:

```
bool DCSS(void **a1, void **a2,
          void *o1, void *o2,
          void *n1) {
 TX_RECORD t;
restart:
  if (Tx_RO_R1(&t, a1) == o1 &&
      Tx_RO_R2(&t, a2) == o2 &&
      Tx_Upgrade_RO_1_To_RW_1(&t)) {
    if (Tx_RO_2_RW_1_Commit(&t, n1)) return true;
  } else if (Tx_RO_2_Is_Valid(&t)) return false;
  goto restart;
}
```

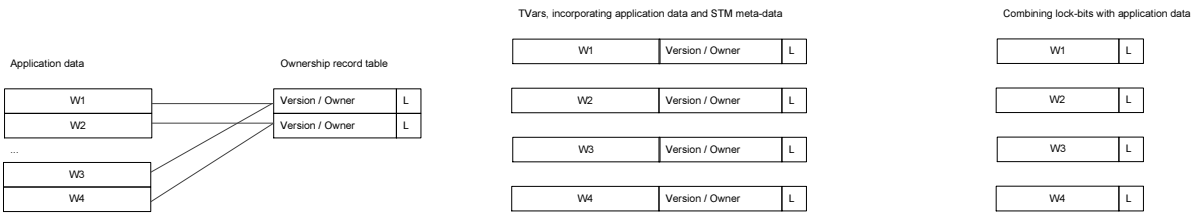
This `DCSS` function reads from `a1` and `a2`. If the values seen match `o1` and `o2` then it upgrades `a1` to be the first location in the write set. If this upgrade succeeds, then `DCSS` attempts to commit the short transaction, with 2 entries in the read set and the single location in the write set.

Building data structures using short transactions. Our implementations of data structures using short transactions can be seen as a mid-point between building directly from CAS and building with a traditional TM interface. On the one hand, when compared with traditional TM, SpecTM burdens the programmer with adding sequence numbers to their reads and writes, and ensuring that the read-set and write-set remain disjoint. On the other hand, when compared with CAS, SpecTM provides the programmer with an abstraction for making multi-word atomic updates.

In practice, when using SpecTM, we start by splitting operations into a series of short atomic steps, each of a statically-known size. As we show in the case study in Section 3, in our skip list implementation we use short RW transactions to atomically add and remove an item when it appears only at level-1 or level-2 in the skip list. We use general purpose transactions for the (rarer) cases of nodes that appear at higher levels. Short and general purpose transactions can be mixed as they use the same STM meta-data. Our hash table and skip list implementations are more complex than ones built using traditional TM implementations. However, they are simpler than the equivalent lock-free algorithms as the use of multi-word atomic updates simplifies the most complex parts of these data structures.

Code complexity. Using short SpecTM transactions interface from Figure 2 can easily result in mistakes by programmers (e.g. using a wrong function name or a wrong index). Incorrect uses of the SpecTM interface can typically be detected at runtime. For performance, we do not implement such checks in non-debug modes. Furthermore, short transactions do not compose as well as traditional transactions, as they require static operation indices. Short transactions are intended for performance critical code and programmers are often willing to sacrifice composability of such code, so this is not a major limitation. For example, code using CAS or locks is routinely used when performance is important, despite its poor composability.

Performance. The "SpecTM-Short" line in Figure 1 shows the impact on performance of using short transactions. In particular, note how short transactions remove much of the



(a) Meta-data held in a table of ownership records, indexed by a hash function.

(b) Meta-data co-located with application data in `TVars` (Section 2.3).

(c) One lock-bit of meta-data held in each data item (Section 2.4).

Figure 3. Different ways of organizing STM meta-data in variants of SpecTM.

overhead at 1-thread without harming scalability. This follows from our focus on designing the SpecTM API to remove much of the book-keeping of typical STM systems: we are performing the same synchronization work.

2.3 Explicit Transactional Variables

The second way in which SpecTM specializes the STM interface is to allow transactional data to be located alongside meta-data. Typical STM systems for C/C++ are built without control over data structures, and instead place their meta-data in a table of ownership records ("orecs" [17]), using a hash function to map from a location in the heap to the orec used for that location. This traditional design means fetching two cache-lines on each data access. It can also introduce false conflicts when distinct data items can hash to the same orec. Even with a few false conflicts, this traditional design can increase book-keeping overheads, as more complex data structures need to be used for handling updates to several locations that map to the same orec.

With SpecTM, we follow STM-Haskell [15] in using a `TVar` data type which encapsulates both (*i*) the meta-data required by the TM system, and (*ii*) a word of the application's data. If the `TVar` is aligned to a 2-word boundary, then the complete structure will be held on a single cache line.

`TVars` do not significantly change the short transaction API in Figure 2: the calls simply take addresses of `TVars` instead of memory words. However, using `TVars` reduces the cache miss rates thus improving STM's performance and scalability. Also, as each memory location maps to a single orec, the book-keeping overheads are reduced.

Figure 3 illustrates these different approaches. Figure 3(a) shows the use of a table of ownership records, with application data words `W1` and `W3` both mapping to the first ownership record, and `W2` and `W4` to the second. In contrast, in Figure 3(b), each `TVar` includes an application word and the ownership record.

Performance. The "SpecTM-Short-TVar" line in Figure 1 shows the impact on performance of using short transactions and `TVars` in the hash table workload. The difference between these results and the earlier "SpecTM-Short" results follows from the placement of the orecs within `TVars`, and the consequent changes in cache behavior.

2.4 Combined Meta-Data with Value-Based Validation

The final form of specialization in SpecTM is to combine the STM's meta-data with the application's own data. From the point of view of the SpecTM API, data structures are still maintained using `TVars`. However, rather than having each `TVar` include two words, each `TVar` now consists of a single word of the application's data, within which one bit is reserved for the use of the STM (Figure 3(c)). Eliminating the additional orec word further reduces STM overheads. For example, traditional STMs need to perform a sequence of three reads (orec, data word and then orec again) to get a correct snapshot of data and the corresponding orec [16]. When data and meta-data are held in the same word, this sequence becomes a single atomic read. Similarly, at commit-time, the entire `TVar` can be updated by an atomic write.

Reserving a bit restricts the programmer to storing pointers to aligned addresses (where the alignment guarantees that some low-order bits are spare), or storing small integer values (shorter than a full word). These restrictions would not be palatable in general-purpose code, but they can be accommodated in our pointer-based data structures.

Validation with Version Numbers. Before introducing how SpecTM operates using a single bit per word, we briefly summarize how BaseTM works when using full orecs (Figure 3(a)–(b)). With full orecs, as with TL2 and other STM systems [16], the orecs combine a lock-bit and either a version number or a reference to a transaction record. If the lock-bit is set then the orec is locked by a transaction that is writing to the orec's data (and the body of the orec points to this owning transaction). Otherwise, if the lock-bit is clear, then the body of the orec contains a version number that is incremented whenever a transaction commits an update to the orec's data. As in other STM systems, these version numbers are used for validating locations that are read by a transaction: a transaction records the version numbers of orecs when first reading from them, and it validates by checking that these version numbers are unchanged.

Version-Free Validation. We use the single bit of meta-data as a lock-bit: it is locked by the `Tx_RW_*` operations in read-write transactions, and during the commit phase of normal transactions. To acquire the lock, a transaction atom-

ically tries to set the lock bit and to replace the rest of the word with a pointer to the owner's transaction record. As in other STM systems, deadlock is avoided conservatively by aborting if the lock is not free.

However, having only a single bit of meta-data introduces a problem when handling transactions that read from a location but do not wish to write to it. The problem is that, without version numbers, a transaction cannot check for conflicting writes to the locations that it has read from. In general, it is incorrect for it to simply re-read the values in each of these locations and to check that each of these matches the values seen by the transaction. This is because there is no guarantee that these reads see a consistent view of memory, given that there may be concurrent writes in progress.

Our observation is that, although the general case of version-free validation is incorrect, we can identify a series of special cases in which it can nevertheless be employed safely:

- Many read-modify-write transactions update all of the locations that they read. In SpecTM these transactions are expressed using the `Tx_RW_*` API. Version numbers are not needed in these transactions because all of the orecs are locked before making the updates. For instance, a transaction to add a node into a doubly-linked list will lock and update all four of the locations involved.
- "Mostly-read-write" transactions read from only one location that they do not update—e.g., in the skip list in Section 3, an insertion must read from a location that records the maximum height of the list nodes. For these transactions it is correct to validate their single read-only access with a value-based comparison. The transaction proceeds by locking the locations being updated, checking that the value in the location read has not changed, and then making the updates and releasing the orecs. The single read forms the linearization point.
- Finally, some locations satisfy a "non-re-use" property in which a given value is not stored in a given location more than once. In this case, if a transaction reads values $v_1, v_2, \ldots$ from a series of locations $a_1, a_2, \ldots$, and then it subsequently sees these same values upon validation, then the non-re-use property means that the complete set of values was present at those locations at the instant of the start of the validation. In effect, the values themselves are taking the place of version numbers. This kind of non-re-use property often occurs when the values are pointers to dynamically allocated data managed by mechanisms such as those of Herlihy *et al.* [21] and Michael [25].

These three special cases cover many of the transactions used in shared memory data structures; indeed they cover all of the cases that occur in our hash table and skip list workloads. In fact, we initially identified these cases after we were surprised that value-based validation worked correctly for our workloads, even though it should not work in general.

Exploiting these special cases requires care on the part of the programmer—the first two cases can be checked automatically by the SpecTM implementation, but we have not yet investigated checking tools or proof methods for the non-re-use property. In order to support general-purpose transactions that do not fit in any of these special cases, a global version number can be used to track the number of transactions that have committed. This can be used, as Dalessandro *et al.* show [3], to make value-based validation safe without requiring a non-re-use property.

If this general-purpose case occurs rarely then, rather than having a single, shared, version number on which each thread contends, each thread can maintain a separate version number. These per-thread numbers are updated on each transactional commit on the given thread. This design makes it fast to (logically) increment the shared counter, at the cost of reading all of the threads' counters in the general case.

Performance. The "SpecTM-Short-TVar-Val" line in Figure 1 shows the impact on performance of using short transactions, with `TVars`, and with value-based validation exploiting the three special cases described above. For this workload, the additional performance above the "SpecTM-Short-TVar" line is slight; however, it closes the gap with the performance of the CAS-based implementation.

3. SpecTM Case Study

To illustrate the use of SpecTM in more detail, we now show how it can be used to implement a skip list (Figure 4). For brevity, we simplify the skip list to store only integer values and to provide `search`, `insert`, and `remove` operations (the pseudo-code shows only the former two). We also omit memory-management code—many now-conventional techniques can be used [11, 21, 25].

Each skip list node stores an integer value, and an array of forward pointers, with one pointer for each level of the skip list the node belongs to (line 2). The skip list is represented by a head node that points to the first node in each level of the list (line 7). To iterate the list, a window of pointers for all skip list levels is used (line 10).

The function for searching the skip list (line 15) traverses the nodes by reading their forward pointers (line 19). It starts at the highest level in the skip list, moving successively lower whenever the level would skip over the integer being sought. As in lock-free linked list and skip list implementations [11, 18], a "deleted" bit is reserved in all of a node's forward pointers to indicate that the node has been deleted. The search function ignores deleted notes (line 20). The search terminates once it reaches the bottom level.

Adding a new node (line 30) starts with a search for the value being inserted (line 35). The skip list does not permit duplicate elements, so `false` is returned if the value is found (line 36) Otherwise, the search returns an iterator that can be used for the insertion. The level of the new node is generated randomly, with the probability of node being

```
1  const int MAX_LEVEL = 32;
2  struct Tower {
3    int id;
4    TmPtr next[MAX_LEVEL]
5    int lvl;
6  };
7  struct Skiplist {
8    Tower head;
9  };
10 struct Iterator {
11   Tower *prev[MAX_LEVEL];
12   Tower *next[MAX_LEVEL];
13 };
14
15 Tower *Skiplist::Search(int id, Iterator *it, int lvl)
       {
16   Tower *curr, *prev = &head;
17   while(--lvl >= 0) {
18     while(true) {
19       curr = Tx_Single_Read(&(prev->next[lvl]));
20       curr = Unmark(curr);
21       if(curr == NULL || curr->id >= id)
22         break;
23       prev = curr;
24     }
25     it->prev[lvl] = prev;
26     it->next[lvl] = curr;
27   }
28   return curr;
29 }
30 bool Skiplist::Add(Tower *data) {
31   Iterator it;
32   bool restartFlag;
33 restart:
34   int headLvl = PtrToInt(Tx_Single_Read(&head.lvl));
35   Tower *curr = Search(data->id, &it, headLvl);
36   if(curr != NULL && curr->id == id)
37     return false;
38   data->lvl = GetRandomLevel();
39   if(data->lvl == 1)
40     restartFlag = !AddLevelOne(data, &it))
41   else
42     restartFlag = !AddLevelN(data, &it);
43   if(restartFlag)
44     goto restart;
45   return true;
46 }
47 bool Skiplist::AddLevelOne(Tower *data, Iterator *it) {
48   TmPtrWrite(&(data->next[0]), it->next[0]);
49   return Tx_Single_CAS(&iter->prev[0]->next[0],
50     it->next[0], data) == it->next[0];
51 }
52 bool Skiplist::AddLevelN(Tower *data, Iterator *it) {
53   bool ret;
54   STM_START_TX();
55   int headLvl = STM_READ_INT(&(head.lvl));
56   if(data->level > headLvl) {
57     STM_WRITE_INT(&(head.lvl), data->level);
58     for(int lvl = headLvl;lvl < data->level;lvl++) {
59       it->prev[lvl] = head;
60       it->next[lvl] = NULL;
61     }
62   }
63   for(int lvl = 0;lvl < data->level;lvl++) {
64     Ptr nxt = STM_READ_PTR(&win->prev[lvl]->next[lvl]);
65     if(nxt != it->next[lvl]) {
66       ret = false;
67       STM_ABORT_TX();
68     }
69     STM_WRITE_PTR(&(it->prev[lvl]->next[lvl]), data);
70     TmPtrWrite(&(data->next[lvl]), win->next[lvl]);
71   }
72   ret = true;
73   STM_END_TX();
74   return ret;
75 }
```

Figure 4. Skiplist implementation using SpecTM.

assigned a level l equal to $\frac{1}{2^l}$. The node is then inserted atomically into all of the lists up to this level. The nodes with level one are inserted using a short specialized transaction (lines 40) and the nodes with higher levels are inserted using an ordinary transaction (line 42). If the insertion does not succeed due to the concurrent changes to the skip list, the whole operation is restarted (line 44). Otherwise, the insert succeeds and `true` is returned to indicate its success.

Removals proceed in a similar manner to insertions. The node is first located using the search function. A single transaction is used to atomically mark the node at all levels, and to remove it from all of the lists it belongs too. Removal of nodes at level one is performed using a short specialized transaction, and the removal of nodes with higher levels is performed using ordinary transactions.

These insertion and removal operations typify the way we use SpecTM. The common cases are expressed using short transactions, and less frequent cases are expressed with more general, but slower, ordinary transactions. If developers see the need to further improve performance, they can further specialize the implementations. The skip list implementation used in our evaluations uses short transactions for levels 1–2, leaving only 25% of insert and remove operations to be executed using ordinary transactions.

Code Complexity. The code of the SpecTM skip list is clearly more complex than a traditional transactional implementation. However, it is simpler than the code of the lock-free skip list based on CAS, such as one by Fraser [11]. There are two main reasons for this simplicity:

First, because specialized transactions can co-exist with ordinary transactions, the more complex operations can still be expressed as ordinary transactions. This limits the amount of code that must be written using SpecTM.

Second, although the API for SpecTM is more complex than for traditional transactions, conceptually it still provides the abstraction of atomicity. Experience with the bounded-size Rock Hardware TM [5, 6, 9] has shown that short transactions are simpler to use than CAS—e.g., without multi-word atomic operations, Fraser's CAS-based skip list must handle nodes which are partially-removed and partially-inserted.

4. Evaluation

In this section we evaluate the performance of SpecTM. We first describe the details of the implementation (Section 4.1), and summarize the different STM systems that we use in our evaluation (Section 4.2). We then examine the performance of SpecTM in single-threaded executions (Section 4.3). This lets us assess the sequential overheads of different approaches. Then, we evaluate the performance and scalability of hash table and skip list data structures implemented using SpecTM, and compare it to lock-free implementations (Section 4.4). We do so on two systems that support 16 and 128 hardware threads respectively.

4.1 Implementation

Our implementation of SpecTM operates on 64-bit systems. Consequently, we ignore the possibility of version number overflow. Existing techniques could be used to manage overflow on 32-bit systems if needed [10, 14].

We use a conventional epoch-based system for memory management, based on that described by Fraser [11]. This mechanism ensures that a location is not deallocated by one thread while it is being accessed transactionally by another thread. Epoch-based reclamation works well in our setting. However, it would be straightforward to use alternatives such as tracing garbage collection, or lock-free schemes [21, 25].

Our baseline STM implementation uses the TL2 algorithm [4], timebase extension [27], and hash-based write sets [30]. The approach is typical of C/C++ STM systems [4, 7, 10, 28, 30]. It performs well for our workloads, matching the performance reported by Dragojević *et al.* [8].

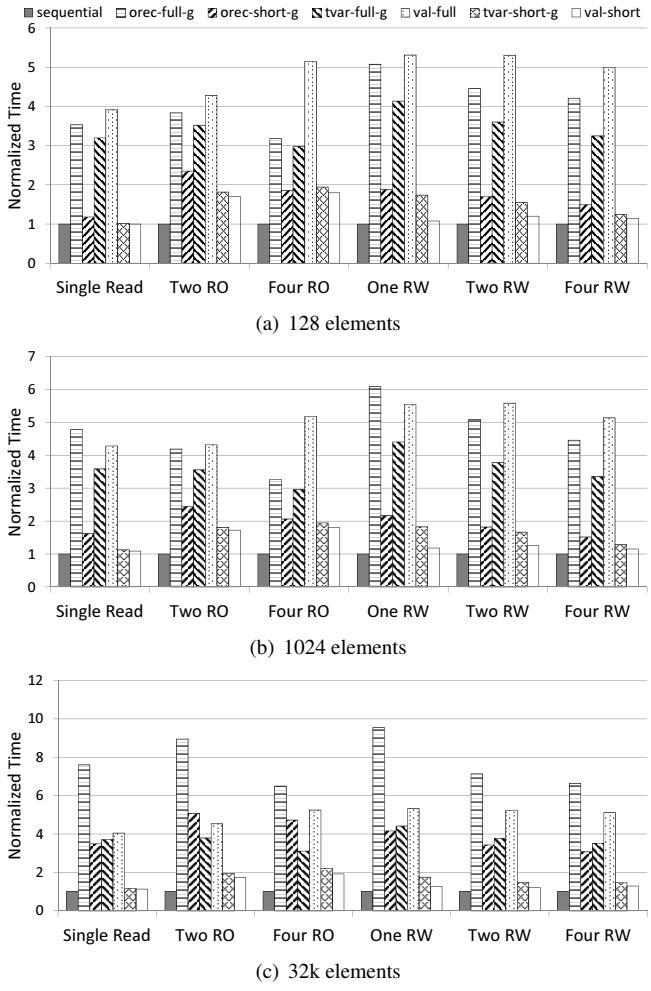
BaseTM provides opacity [12], guaranteeing that a running transaction sees a consistent view of the heap. BaseTM provides weak isolation, meaning that it does not detect conflicts between concurrent transactional and non-transactional accesses to the same location (if strong isolation is needed, then many implementation techniques exist [16]). BaseTM does not provide privatization safety (barriers such as those of Marathe *et al.* [24] could be added, if needed).

As illustrated in Figure 3(a), we use a global table of ownership records, indexed by a hash function. We use commit-time locking (CTL), where orecs are locked only during commit, rather than during a transaction's execution. We use invisible reads (so the implementation of `Tx_Read` does not write to any shared data). We use deferred updates (meaning that updates are held in a write log during execution, and flushed to the heap on commit). We use a simple contention manager: upon conflict, a transaction aborts itself, and waits for a randomized linear time before restarting (as in the first phase of SwissTM's two-phase contention manager [7]). With BaseTM, all transactions executed by the same thread use the same per-thread transaction descriptor that is allocated and initialized at thread start-up.

BaseTM can use two version management strategies. Both strategies are conventional, but they offer different performance characteristics:

Global version numbers. We can use a global version number, held in a 64-bit integer that is incremented by non-read-only transactions. As with TL2, we sample the global version number at the start of a transaction, and obtain opacity by ensuring that all orecs accessed by the transaction have versions no later than this starting number.

Local (per-orec) version numbers. Alternatively, we can use per-orec version numbers, without reference to a global version. This avoids contention on a shared global counter, but instead requires read-set validations after every read in order to ensure opacity.



(a) 128 elements

(b) 1024 elements

(c) 32k elements

Figure 5. Single thread performance of SpecTM.

4.2 STM Variants

We summarize the labels used on our graphs:

sequential is optimized sequential code; it is not safe for multi-threaded use, but it provides a reference point of the cost of an implementation without concurrency control.

lock-free are lock-free implementations of the data structures, based on the designs from Fraser's thesis [11].

*orec-** implementations use a shared table of orecs, as shown in Figure 3(a).

*tvar-** implementations use per-data-item ownership records, as shown in Figure 3(b).

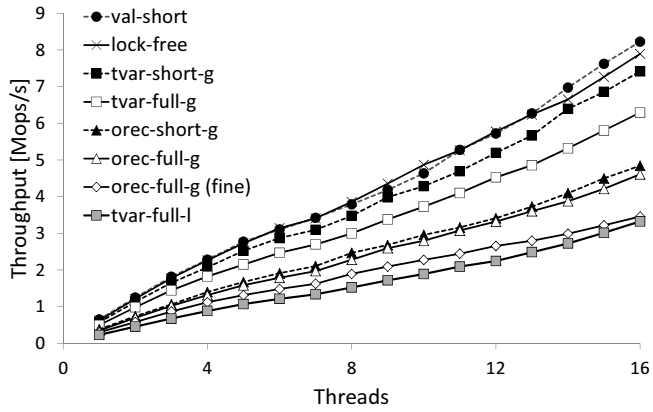
*val-** implementations use per-data-item lock-bits with value-based validation, as shown in Figure 3(c).

-full- implementations use the normal STM interface (Section 2.1).

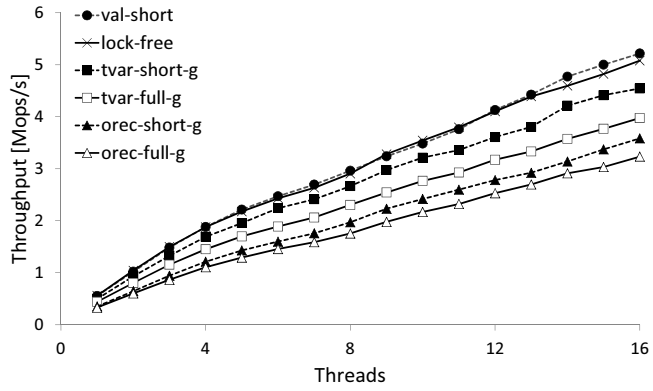
-short- implementations use specialized interfaces for short transactions (Section 2.2).

**-g* implementations use a global version number.

**-l* implementations use local (per-orec) version numbers.

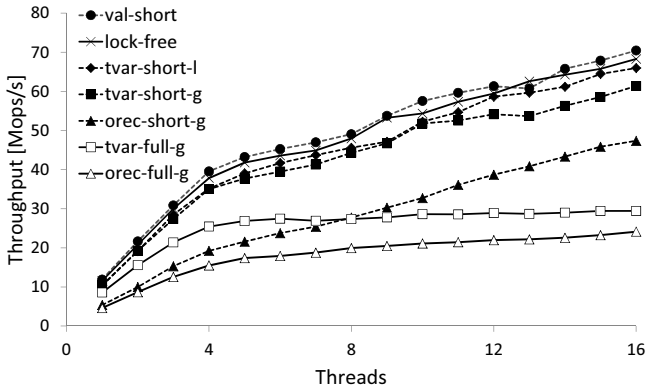


(a) 90% lookups

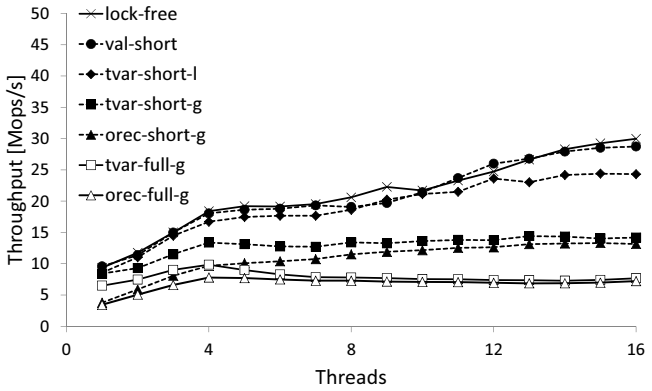


(b) 10% lookups

Figure 6. Skip list, 64k values, 16 cores.



(a) 90% lookups



(b) 10% lookups

Figure 7. Hash table, 64k values, 16k buckets, 16-cores.

For instance, *orec-full-g* denotes our BaseTM implementation using a TL2-style global version number, whereas *tvar-short-g* denotes an implementation using specialized short transactions, without an orec table. Our experimental harness explores the full set of variants. However, we omit some of the lines from the graphs for clarity.

4.3 Single-Threaded Performance

We evaluate single-threaded performance using a synthetic workload on a single-socket system with a 2.26GHz Intel Xeon E5520 quad-core CPU.

The synthetic workload allocates an array of pointers, with each pointer aligned to a L2 cache-line boundary. We then measure the time needed to execute a large number of short transactions on randomly chosen items in the array. We repeat the experiment for `Tx_Single_Read` transactions, read-only (RO) transactions that access 2 and 4 consecutive items in the array, and read-write (RW) transactions that access 1, 2, and 4 consecutive items. (The *val-full* RO transactions assume the non-re-use property from Section 2.4 and perform value-based validation.) We run the experiments with different sizes of the array, thereby controlling how much of the array fits in data caches, and consequently how frequently cache misses occur.

Figure 5 shows the normalized execution time of the different STM implementations for array sizes of 128 elements (half the size of 32KB L1 cache), 1024 elements (half the size of 256KB L2 cache) and 32 768 elements (half the size of 8MB L3 cache). We normalize the read-only results against sequential code that reads from 1, 2, and 4 items using ordinary load instructions. We normalize the RW results against sequential code that performs a single-word CAS instruction on each of the 1, 2, and 4 items. Comparison with these baselines illustrates the costs that SpecTM implementations add in order to obtain transactional guarantees.

Comparing the *sequential* bars with BaseTM (*orec-full-g*) shows a 3x–10x overhead for the baseline transactions compared with sequential code. Most variants of SpecTM out-perform BaseTM. The exception is *val-full* in which the read-set validation costs incurred on each transactional read dominate execution time.

The best performing SpecTM variants are those that use short transactions. Comparing *orec-short-g* (Figure 3(a)), *tvar-short-g* (Figure 3(b)), and *val-short* (Figure 3(c)), the value-based implementation is slightly faster because it avoids the atomic increment on a shared global timestamp.

For a single read, the fastest SpecTM variants are comparable to sequential code. For RW transactions, the overhead

of *val-short* is 10%–30% on 1-item, and 15%–30% on 2 or 4 items. This is substantially less than the 3x-10x overheads of BaseTM. The overheads of RO transactions are higher than those of RW transactions; we believe this is due to the more complex control flow in by validation.

When cache misses are rare (Figure 5(a)), the use of short transactions helps more than the use of `TVars` and value-based validation. This is because the meta-data remains resident in the L1 cache, and so the reduced number of instructions executed is significant. In contrast, for workloads which are not L1-resident, the use of more compact meta-data is as important as the use of short transactions.

4.4 Multi-Threaded Performance

We evaluate scalability using integer set benchmarks, with threads performing a random mix of lookups, insertions and removals. For each of the operations, threads pick a key uniformly at random from a predefined range. In the experiments, we used the range 0–65 535, and we varied the mix of operations to control the contention between threads. Before the experiment starts, the set is initialized by inserting half of the elements from the key range. In order to keep the size of the set roughly constant, the ratio of insert and remove operations is equal. During the execution, about half of the insert operations fail because the value they are trying to insert is already in the set. Similarly, about half of the remove operations fail because the value they are trying to remove is not in the set.

We use hash table and skip list integer set implementations in our experiments. By default, we set the number of bucket chains in the hash table to $16k$, which makes the average length of the bucket chain 2. We set the maximum height of the skip list nodes to 32.

We use two different multi-core systems. First, a machine with 4 quad-core AMD Opteron 8374HE CPUs clocked at 2.2GHz. Second, a machine with 8 Intel Xeon x7560 CPUs clocked at 2.26GHz. Each of the CPUs has 8 cores and each core can run 2 hardware threads; hence the entire system supports 128 hardware threads. Both systems run Windows Server 2008 R2 Enterprise operating system. Our results are the mean of 6 runs with the lowest and the highest discarded.

In each of the figures below, we omit the implementations that do not perform well to improve readability. In particular, we do not include results for all **-g* implementations where the impact of contention on the shared global version is high. This typically occurs with short transactions and systems with many hardware threads. Likewise, we do not include results for all **-l* implementations where the cost of incremental validation is high. This typically occurs with longer transactions and systems with fewer hardware threads.

4.4.1 Performance on the 16-way system.

Skip list. Figure 6 shows the throughput of the skip list experiment on the 16-way machine with a read-mostly workload (90% lookups) and write-heavy workload (10% lookups).

Figure 6(a) shows the read-mostly workload. The *val-short* implementation performs as well as the lock-free implementation, and it outperforms BaseTM *orec-full-g* by 60%–80%. The *tvar-short-g* implementation is slightly slower than the lock-free algorithm (7–10%). For this workload, the use of the *val-** and *tvar-** variants is more important than the use of short transactions (note that *orec-short-g* is only slightly faster than *orec-full-g*). The *tvar-full-l* implementation performs poorly because of the cost of incremental read-set validations to ensure opacity without a global version number.

Figure 6(a) also shows the performance of a skip list implementation using BaseTM, but splitting each lookup/insert/remove operation into a series of fine-grained transactions that are implemented over the ordinary STM interface rather than using short transactions. This is labeled *orec-full-g (fine)*. Comparing this fine-grained implementation with the ordinary *orec-full-g* implementation shows that we do not obtain performance benefits by using fine-grain transactions *without* the specialized implementation for them in SpecTM: without the specialized implementation, the overheads of the fine-grain transactions are prohibitive.

Figure 6(b) shows performance under the write-heavy workload. The overall performance is lower than the read-mostly workload. However, the relative performance of different variants is approximately the same (in the figure we omit some of these lines for clarity). The *val-short* results perform roughly the same as the lock-free implementation, and outperform base SpecTM by 60–70%.

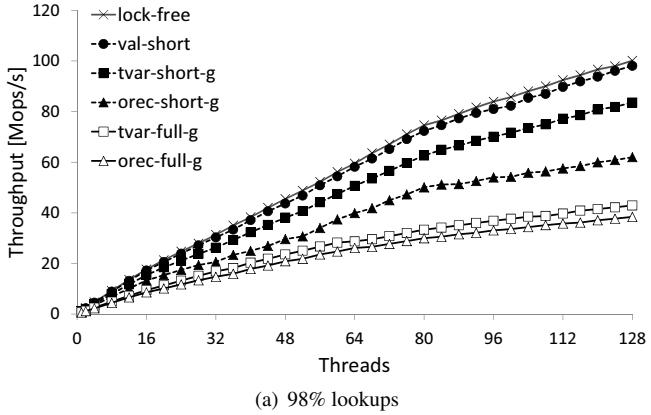
To summarize, we get a slight benefit from splitting larger transactions into fine-grain short transactions (*orec-full-g* to *orec-short-g*), but in doing so we enable the specializations from *orec-short-g* to *val-short*, that let us achieve the same level of performance as the CAS-based implementation.

Hash table. Figure 7 shows the throughput of various hash table implementations on the 16-way machine.

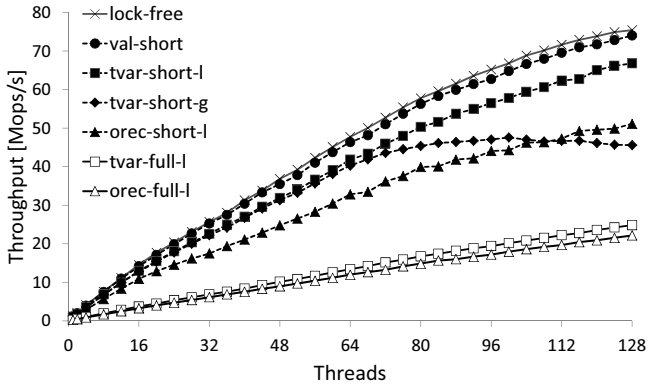
If a hash table is lightly loaded, as in our setup, then operations on it are much shorter than those on a skip list. Consequently, the hash table workload stresses SpecTM in a different way to the skip list because the use of centralized data has a higher impact on scalability.

The impact of the shared global counter in the **-g* variants is apparent in the read-mostly workload (Figure 7(a)), and even more so in the write-heavy workload (Figure 7(b)). On this machine, we see little scaling beyond 4 threads for the `*-g` variants without specialized short transactions. (With more than 4 threads, two or more CPUs must be used, which significantly increases the cost of cache-misses.)

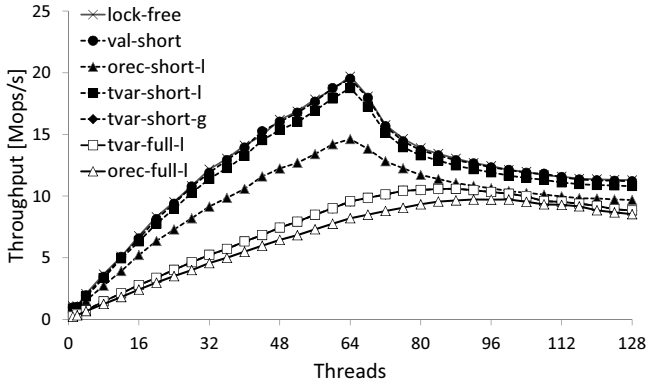
In the read-mostly workload (Figure 7(a)), the *val-short* results match the performance of the lock-free hash table. This implementation outperforms the baseline *orec-full-g* implementation between 2.5 and 3 times and the baseline



(a) 98% lookups

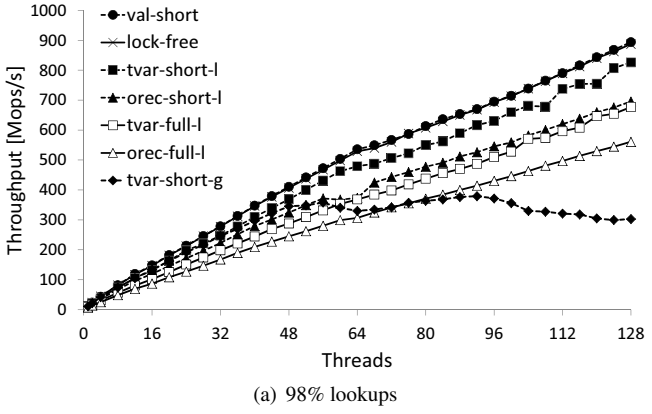


(b) 90% lookups

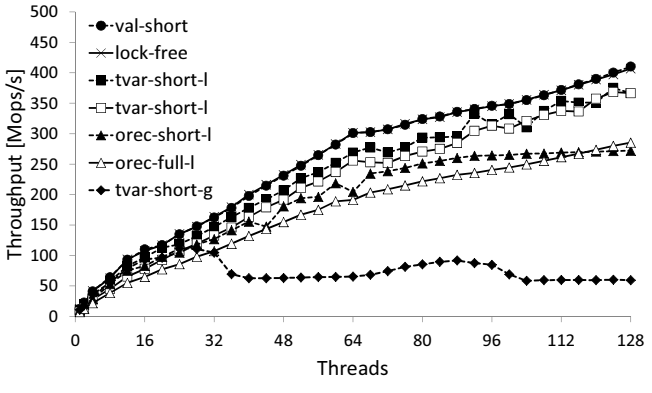


(c) 10% lookups

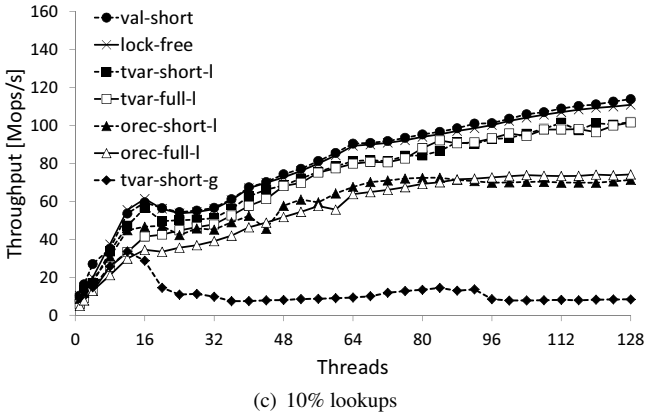
Figure 8. Skip list, 64k values, 128-way system.



(a) 98% lookups



(b) 90% lookups



(c) 10% lookups

Figure 9. Hash table, 64k values, 16k buckets, 128-way.

implementation with local per-orec timestamps *orec-full-l* between 1.6 and 3.5 times (not shown in the figure). The *tvar-short-** results are slightly lower than the *val-short* results. The other SpecTM variants perform and scale less well. The **-l* variants typically have lower single-threaded performance (due to incremental validation), but scale better than the **-g* variants.

In the write-heavy workload (Figure 7(b)) the *val-short* implementation matches the performance of the lock-free implementation, despite a much higher update rate. The main difference between the read-mostly and write-heavy results is that the *orec-** and *tvar-** implementations with shared timestamp scale poorly because of contention on the shared timestamp.

4.4.2 Performance on the 128-way system.

Skip list. Figure 8 shows the results of the skip list experiment on the 128-way system with 98%, 90% and 10% lookup operations.

With 98% lookups (Figure 8(a)), the rate of update operations is not high enough for contention on the shared timestamp counter to be significant. The most notable difference from the previous experiments is that *val-short* performs slightly less well than the lock-free implementation:

it achieves 95%–97% of the lock-free skip list's throughput, and it outperforms the baseline STM by 2–2.5 times. Unlike the 16-way case, the *tvar-short-g* is less competitive with the lock-free algorithm and with *val-short*. Interestingly, also unlike the 16-way case, the use of short transactions helps significantly, even when a shared table of orecs is used (comparing *orec-full-g* and *orec-short-g*).

With 90% lookups (Figure 8(b)), contention on the shared timestamp counter results in the **-l* variants outperforming **-g* variants. For this reason, we focus on **-l* variants in the figure. Once again, the *val-short* and *tvar-short-l* implementations have the best performance and scalability.

With 10% lookups (Figure 8(c)), all implementations scale poorly—including the lock-free implementation. Nevertheless, the relative performance of the SpecTM variants follows that of Figure 8(b).

Hash table. Figure 9 shows the hash table results on the 128-way machine with 98%, 90% and 10% lookups.

Figure 9(a) shows the performance with 98% lookups. The *val-short* implementation matches the performance of the lock-free hash table implementation, and it outperforms BaseTM between 60% and 70%. We see roughly equal gains from co-locating data and meta data, and from using the API for short transactions.

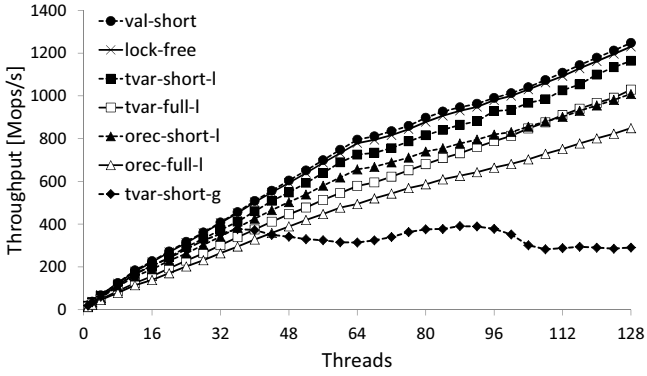
Figure 9(b) shows the performance with 90% lookups. With more update operations the scalability is slightly impacted for all hash table variants (including the lock-free variant). As in previous experiments, *val-short* matches the performance of the lock-free algorithm.

Figure 9(c) shows performance with only 10% lookups. As contention increases, the performance of *orec-short-l* drops to offer little or no benefit above *orec-full-l*. The main reason for the lower scalability of SpecTM variants with short transactions is the use of encounter-time locking (ETL) in short RW transactions, when compared with commit-time locking (CTL) in long transactions. As the contention increases, the ETL implementation leads to more locks being acquired by later aborted transactions, whereas the CTL implementation does not acquire the locks in the first place.

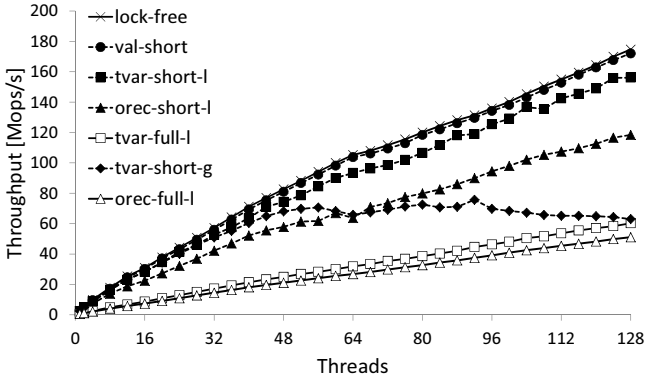
Finally, we examined the performance of the hash table under workload with short chains of buckets (0.5 entries per bucket on average), and with long chains of buckets (32 entries per bucket on average). Figure 10(a) shows the short-chain results, and Figure 10(b) shows the long-chain results. The overall result, that *val-short* matches the performance of the lock-free implementation, remains from the earlier experiments. With longer chains, the **-full-l* variants scale poorly: their read sets become large, increasing costs of incremental validation.

4.5 Summary

We evaluate a number of SpecTM variants across a range of workloads on two parallel systems. In all cases, the best performing SpecTM variant was *val-short*. It either matched



(a) 98% lookups, 64k buckets (0.5-entry chains)



(b) 90% lookups, 1k buckets (32-entry chains)

Figure 10. Hash table, with short and long chains in each bucket, 128-way system.

the performance of the lock-free implementations of the same data structure or was marginally slower (typically 3%–5%). It also outperformed BaseTM substantially across all workloads.

The results suggest that both the specialized API for short transactions, and the control over the STM meta-data, are very useful in achieving good performance. Exactly how much each type of specialization helps depends on the workload and on the target system. In addition, the performance gain of the variants that use both types of specialization is often higher than the simple combination of the performance gains by each type of specialization in isolation. This is because the co-location of STM meta-data with application data allows the implementation of short transactions to be further simplified.

5. Related Work

Transactional memory was first proposed by Herlihy and Moss [17] as a way to simplify lock-free programming by allowing programmers to define short, customized, read-modify-write operations. The first STM of Shavit and Touitou [29] had a similar aim, albeit a simpler, static interface. The first dynamic STM in which transactions do not have to access predetermined sets of objects was STM by

Herlihy *et al.* [20]. Dynamic STMs are more flexible than the static ones as the transactions can choose which object to access based on the values read in the current transaction; this flexibility motivates our use of a dynamic interface for short transactions. Harris *et al.* provide a recent survey of TM implementations [16]. Our BaseTM implementation builds on many techniques from previous work, particularly those of Dice *et al.* [4], Riegel *et al.* [27], and Spear *et al.* [30].

Current hardware typically implements single-word synchronization primitives, such as compare-and-swap (CAS), load-link/store-conditional (LL/SC), fetch-and-add, and similar. Lock-free algorithms are often much easier to develop if less restrictive forms of the synchronization primitives are available.

The restrictions of current hardware have led researchers to investigate software implementations of multi-word atomic primitives, such as multi-word compare-and swap (e.g. [13, 22]), and k-compare-single-swap primitive [23]. Our view is that it is easy to implement CASN over short transactions, but it is difficult to implement short transactions over CASN: Our interface for short transactions is dynamic (unlike typical interfaces for CASN), and it provides the programmer a consistent view of the heap during a transaction. Unlike CASN implementations, our implementations support inter-operation with general-purpose STM.

Recently, HTM systems have started to emerge from industry [5, 26]. Typically, these support limited forms of transactions to enable practical implementations in hardware. The HTM feature of the Rock CPU from Sun Microsystems [5] supports only "best-effort" transactions, where each transaction can fail repeatedly for implementation-dependent reasons. The recently announced Intel TSX [26] instructions support a "restricted transactional memory" mode that is subject to implementation-defined size constraints. Despite weak guarantees, short and simple transactions make it possible to use HTM to simplify and speed-up concurrent code [6, 9].

As with these HTM systems, SpecTM promotes the use of short transactions. Because of the short maximum size of SpecTM's transactions, most algorithms developed for SpecTM can be run on the bounded HTMs and vice-versa. This means that SpecTM allows us to better understand how to design algorithms for the future HTMs, but it also enables us to benefit from these new algorithms even before HTMs are widely available.

Attiya recently examined the complexity of TM systems from a theoretical viewpoint [1] and proposed the use of "mini-transactions" in which a data structure's implementation would be built over a series of small transactions. Attiya's arguments are motivated by complexity lower bounds on implementations of general purpose transactions. However, the proposal matches our practical experience that a series of optimized short transactions can perform better than a single, longer transaction.

6. Conclusions and Future Work

We developed SpecTM to explore the use of optimized short transactions and the co-location and specialization of STM meta-data with application data. These specializations trade the generality of traditional STMs for performance. The target users of SpecTM are experienced programmers that can exploit it in their algorithms to atomically access a handful of locations instead of developing more complex algorithms based on the synchronization primitives available today in hardware. Our experience shows that this ability of SpecTM can result in much simpler algorithms, and our results suggest that the most specialized version of SpecTM (*val-short*) lets us write concurrent hash table and skip list algorithms that perform and scale as well as the lock-free versions of the same algorithms on systems with up to 128 hardware threads.

Our work on SpecTM opens several interesting directions for future work. One direction is to use SpecTM to implement new, efficient, concurrent data structures—for instance, looking at structures such as B-Trees which are more complex than those studied in typical research on lock-free algorithms. This work would let us better understand the kinds of trade-offs STM designers can make when building support for efficient concurrent data structures.

It is possible that some of the techniques used in SpecTM could be employed automatically by STM compilers to optimize transactions, and that software checking tools could be used to ensure that programmers correctly follow the requirements for using SpecTM.

We certainly have not explored the full space of specialized STM designs. For instance, it might be beneficial to explore pointer-only STM designs which use additional spare bits in the pointers as orecs (typically, in 64 bit systems, the processor or OS does not support virtual address spaces that exploit the entire 64-bit space). In addition, a value-based STM that locks words when reading could be used to simplify the programming model in our designs which use value-based validation. Of course, more sophisticated contention managers could improve the performance in some workloads.

Finally, developing algorithms using SpecTM has important implications as HTM systems become available [26]. With the exploration of HTM support in CPUs, it is worthwhile considering the integration of SpecTM and HTMs—both to accelerate aspects of SpecTM, and to provide a software fall-back path for use with best-effort HTMs.

Acknowledgements

We would like to thank the anonymous reviews and our shepherd Bryan Ford along with Richard Black, Austin Donnelly, Miguel Castro, Cristian Diaconu, Steven Hand, Orion Hodson, Paul Larson, Dimitris Tsirogiannis, and Marcel van der Holst for their feedback on earlier drafts of this paper, and for their assistance with conducting the experiments.

References

[1] H. Attiya. Invited paper: The inherent complexity of transactional memory and what to do about it. In *Distributed Computing and Networking*, volume 6522 of *Lecture Notes in Computer Science*, pages 1–11. 2011.

[2] C. Cascaval, C. Blundell, M. Michael, H. W. Cain, P. Wu, S. Chiras, and S. Chatterjee. Software transactional memory: why is it only a research toy? *Communications of the ACM*, 51(11):40–46, Nov. 2008.

[3] L. Dalessandro, M. F. Spear, and M. L. Scott. NOrec: Streamlining STM by abolishing ownership records. In *PPoPP '10: Proc. 15th ACM Symposium on Principles and Practice of Parallel Programming*, pages 67–78, Jan. 2010.

[4] D. Dice, O. Shalev, and N. Shavit. Transactional locking II. In *DISC '06: Proc. 20th International Symposium on Distributed Computing*, pages 194–208, Sept. 2006. Springer-Verlag Lecture Notes in Computer Science volume 4167.

[5] D. Dice, Y. Lev, M. Moir, and D. Nussbaum. Early experience with a commercial hardware transactional memory implementation. In *ASPLOS '09: Proc. 14th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 157–168, Mar. 2009.

[6] D. Dice, Y. Lev, V. J. Marathe, M. Moir, D. Nussbaum, and M. Oleszewski. Simplifying concurrent algorithms by exploiting hardware transactional memory. In *SPAA '10: Proc. 22nd Symposium on Parallelism in Algorithms and Architectures*, pages 325–334, June 2010.

[7] A. Dragojević, R. Guerraoui, and M. Kapałka. Stretching transactional memory. In *PLDI '09: Proc. 2009 ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 155–165, June 2009.

[8] A. Dragojević, P. Felber, V. Gramoli, and R. Guerraoui. Why STM can be more than a research toy. *Communications of the ACM*, 54(4):70–77, April 2011.

[9] A. Dragojević, M. Herlihy, Y. Lev, and M. Moir. On the power of hardware transactional memory to simplify memory management. In *PODC '11: Proc. 30th ACM SIGACT-SIGOPS Symposium on Principles of Distributed Computing*, PODC '11, pages 99–108, June 2011.

[10] P. Felber, C. Fetzer, and T. Riegel. Dynamic performance tuning of word-based software transactional memory. In *PPoPP '08: Proc. 13th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 237–246, Feb. 2008.

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

[12] R. Guerraoui and M. Kapałka. On the correctness of transactional memory. In *PPoPP '08: Proc. 13th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 175–184, Feb. 2008.

[13] T. Harris, K. Fraser, and I. A. Pratt. A practical multi-word compare-and-swap operation. In *DISC '02: Proc. 16th International Symposium on Distributed Computing*, pages 265–279, Oct. 2002.

[14] T. Harris, M. Plesko, A. Shinnar, and D. Tarditi. Optimizing memory transactions. In *PLDI '06: Proc. 2006 Conference on Programming Language Design and Implementation*, pages 14–25, June 2006.

[15] T. Harris, S. Marlow, S. Peyton Jones, and M. Herlihy. Composable memory transactions. *Communications of the ACM*, 51(8):91–100, Aug. 2008.

[16] T. Harris, J. Larus, and R. Rajwar. *Transactional Memory, 2nd edition*. Morgan & Claypool, 2010.

[17] M. Herlihy and J. E. B. Moss. Transactional memory: architectural support for lock-free data structures. In *ISCA '93: Proc. 20th Annual International Symposium on Computer Architecture*, pages 289–300, May 1993.

[18] M. Herlihy and N. Shavit. *The art of multiprocessor programming*. Morgan Kaufmann, 2008.

[19] M. Herlihy and J. M. Wing. Linearizability: a correctness condition for concurrent objects. *TOPLAS: ACM Transactions on Programming Languages and Systems*, 12(3):463–492, July 1990.

[20] M. Herlihy, V. Luchangco, M. Moir, and W. N. Scherer III. Software transactional memory for dynamic-sized data structures. In *PODC '03: Proc. 22nd ACM Symposium on Principles of Distributed Computing*, pages 92–101, July 2003.

[21] M. Herlihy, V. Luchangco, P. Martin, and M. Moir. Nonblocking memory management support for dynamic-sized data structures. *TOCS: ACM Transactions on Computer Systems*, 23(2):146–196, May 2005.

[22] A. Israeli and L. Rappoport. Disjoint-access-parallel implementations of strong shared memory primitives. In *PODC '94: Proc. 13th ACM Symposium on Principles of Distributed Computing*, pages 151–160, Aug. 1994.

[23] V. Luchangco, M. Moir, and N. Shavit. Nonblocking k-compare-single-swap. In *SPAA '03: Proc. 15th Annual Symposium on Parallel Algorithms and Architectures*, pages 314–323, June 2003.

[24] V. J. Marathe, M. F. Spear, and M. L. Scott. Scalable techniques for transparent privatization in software transactional memory. In *ICPP '08: Proc. 37th International Conference on Parallel Processing*, Sept. 2008.

[25] M. M. Michael. Hazard pointers: safe memory reclamation for lock-free objects. *IEEE Transactions on Parallel and Distributed Systems*, 15(6):491–504, June 2004.

[26] J. Reinders. Transactional synchronization in Haswell, Feb. 2012. `http://software.intel.com/en-us/blogs`.

[27] T. Riegel, C. Fetzer, and P. Felber. Time-based transactional memory with scalable time bases. In *SPAA '07: Proc. 19th ACM Symposium on Parallelism in Algorithms and Architectures*, pages 221–228, June 2007.

[28] B. Saha, A.-R. Adl-Tabatabai, R. L. Hudson, C. Cao Minh, and B. Hertzberg. McRT-STM: a high performance software transactional memory system for a multi-core runtime. In *PPoPP '06: Proc. 11th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 187–197, Mar. 2006.

[29] N. Shavit and D. Touitou. Software transactional memory. In *PODC '95: Proc. 14th ACM Symposium on Principles of Distributed Computing*, pages 204–213, Aug. 1995.

[30] M. F. Spear, L. Dalessandro, V. J. Marathe, and M. L. Scott. A comprehensive strategy for contention management in software transactional memory. In *PPoPP '09: Proc. 14th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 141–150, Feb. 2009.