The Distributed Computing Column

BY

Panagiota Fatourou

Department of Computer Science, University of Crete
P.O. Box 2208 GR-714 09 Heraklion, Crete, Greece
and
Institute of Computer Science (ICS)
Foundation for Research and Technology (FORTH)
N. Plastira 100. Vassilika Vouton
GR-700 13 Heraklion, Crete, Greece
faturu@csd.uoc.gr

Programming with SpecTM

Tim Harris
Microsoft Research
tharris@microsoft.com

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

Abstract

In this paper we examine the use of "mini" transactions. An implementation of mini-transactions supports small sequences of memory accesses as atomic transactions (perhaps 1–4 accesses). When building a shared memory data structure using mini transactions, the programmer must either stay within the limits of a single mini transaction, or split the operation across a series

Bulletin of the EATCS no 107, pp. 72–94, June 2012

of mini transactions. Mini transactions therefore provide a greater degree of atomicity than single-word compare and swap (CAS), but they do not provide the full features of a general-purpose transactional memory (TM). We illustrate how hashtables and skip lists can be built over the SpecTM API for mini transactions. We discuss the advantages and disadvantages of the SpecTM API over a general-purpose TM. To address some of these limitations, we discuss techniques for integrating SpecTM with a general-purpose STM.

1 Introduction

In this paper we examine the use of "mini" transactions to implement shared-memory hashtables and skip lists. In this approach, an operation on a data structure is split across a series of short transactions, rather than using a single transaction encompassing the entire operation. There are several reasons for studying the use of mini transactions:

First, practical implementations of hardware transactional memory (HTM [17]) limit the size of transactions. Some proposals, such as AMD ASF [5], provide a guarantee that a transaction that follows various programming rules will be able to commit eventually if it accesses only 1–4 locations. Recent HTM implementations do not provide such a guarantee [7, 22]. However, irrespective of whether or not a guarantee is given, it is likely that shorter transactions will be more likely to commit than longer ones.

The second reason for studying mini transactions is that, in recent work, we showed that implementations of data structures using mini transactions can perform well [10]. We showed how parts of the implementation could be specialized in cases such as transactions that access a small fixed number of locations. Our preliminary results suggested that data structures built using this system were much faster and more scalable than those built using a general-purpose STM system (we used one based on SwissTM [9] and the STM described by Spear *et al.* [25]).

Finally, Attiya showed recently that, under various assumptions, a series of short transactions can be more efficient than a single long transaction [1]. Attiya's work derived lower bounds on the operations that need to be performed by an STM implementation, given assumptions about the progress-properties and safety-properties that the STM should satisfy.

In Section 2 we review the SpecTM API and describe, in outline, the implementation techniques that we use. We designed the SpecTM API to let us streamline much of a traditional STM system's book-keeping—the result is an API that is more cumbersome to use than a general-purpose STM, but which still provides

```
// Transaction management operations
void Tx_Start(TX_RECORD *t);
void Tx_Abort(TX_RECORD *t);
bool Tx_Commit(TX_RECORD *t);
bool Tx_Validate(TX_RECORD *t);

// Data access operations
Ptr Tx_Read(TX_RECORD *t, Ptr *addr);
void Tx_Write(TX_RECORD *t, Ptr *addr, Ptr val);
```

Figure 1: A traditional STM interface (pseudo-code).

the key abstraction of multi-word atomic memory accesses.

In Sections 3–4 we illustrate the use of SpecTM in practice in implementing a hashtable and a skip list. We take an informal approach in this paper. We aim to illustrate the use of SpecTM through examples, but we do not attempt to quantify exactly how much easier SpecTM is to use than single-word CAS, or exactly how much more difficult SpecTM is to use than a general-purpose TM. Based on these examples, we discuss the difficulties that we have encountered in using the SpecTM API (Section 5).

In Section 6 we discuss how SpecTM can be integrated with general-purpose TM. An advantage of such integration is that SpecTM can be used to accelerate the common cases in a data structure's implementation. General-purpose TM can be used as a fall-back for cases whose performance is not critical, or for cases which appear impractical to split into mini transactions. The main challenge is integrating the conflict detection algorithms used in different TM systems—some techniques from existing software-hardware hybrid TM systems can be applied.

2 Programming Models

Figure 1 sketches the kind of interface typically exposed by general-purpose STM systems. There are operations to start transactions, abort transactions, and to commit them. If `Tx_Commit` returns `true` then we say that the transaction has "succeeded", and its effects appear to take place atomically at some point during its execution. Otherwise, `Tx_Commit` returns `false`, we say that the transaction has "failed", and the transaction's effects are not made visible to other threads. There is a validation operation to detect whether or not a transaction has already experienced a conflict. There are operations to read a memory location, and to update a memory location with a new value. (For brevity we focus on an interface in which all of the locations read and written contain pointers.)

Note that, throughout the paper, we study implementations that provide only weak isolation [3], meaning that there is no conflict detection between transactional and non-transactional memory accesses. This is sufficient to implement

programming models that distinguish between transactional and non-transaction locations (e.g., STM-Haskell [14]). Alternative programming models are often proposed, such as ones that allow privatization idioms. Existing mechanisms could be used to support such models over the basic systems we describe here (Harris *et al.* survey many techniques [15]).

Although TM interfaces such as Figure 1 are widespread, they can be seen as one option in a broader set of alternative abstractions for writing atomic operations on shared memory. One possible characterization of these abstractions is in terms of the following properties:

- *Size*—Are operations of unbounded size permitted, or is there a maximum size? General-purpose TM is unbounded, as are multi-word compare-and-swap operations (CASN). "Strong" LL/SC is unbounded. A CAS operation has a bound of 1. Practical implementations of LL/SC have a bound of 1. DCAS has a bound of 2.

- *Dynamic access*—Can the locations to access be selected dynamically: i.e., selecting the next location to access based on the values seen in previous locations. Recent STM systems support dynamic accesses. CAS, DCAS, and CASN, support only static accesses—that is, the entire set of locations to access is supplied as a parameter to the operation. Shavit and Touitou's original STM supported only static accesses [24].

- *Inconsistency hidden*—Does the programmer have to consider the possibility of seeing a mutually inconsistent view of a set of locations? Alternatively, does the abstraction provide a property such as opacity [12] or TMS1 [8] that precludes this? HTM designs typically hide inconsistency. Many STM designs do, whereas others do not. The question does not arise with many CAS, DCAS, and CASN abstractions which provide only a success/failure response, rather than a snapshot of the locations accessed.

- *Fallback required*—Can the programmer use the abstraction without needing to write alternative code using a different abstraction? Best-effort HTM systems do not guarantee that any transaction will ever commit successfully (even if the transaction is short and does not experience contention). Consequently, an alternative code path is needed—e.g., based on locking, or based on STM. Typical STM systems do not need a fallback code path.

Figure 2 compares the properties of various practical implementations of programming abstractions along these axes. Concretely, for HTM, we consider a best-effort system. Note that, the size in this table is listed as unbounded (because the API does not prevent unbounded-size transactions from being written), but a

	CAS	DCAS	CASN	STM	Best-effort HTM	SpecTM
Size	1	2	n	n	n	4
Dynamic access	n/a	Static	Static	Dynamic	Dynamic	Dynamic
Inconsistency hidden	n/a	n/a	n/a	Usually	Yes	No
Fallback required	No	No	No	No	Yes	No

Figure 2: Comparison of typical implementations of different abstractions.

fallback is required because a best-effort implementation is not required to be able to run any specific size of transaction successfully. Figure 2 also characterizes the behavior of our SpecTM system for writing mini transactions.

Unlike general-purpose STM, SpecTM supports only a limited number of memory accesses within a single transaction (4, in the current implementation). Unlike CASN, it provides a dynamic interface.

Unlike many STM implementations, SpecTM exposes inconsistent views of memory to the programmer. To prevent possible problems that could result from execution with inconsistent reads, SpecTM includes functions that allow the programmer to explicitly check for the inconsistencies if needed. Furthermore, some implementation strategies, such as write-locking on reads in short read-write transactions, might guarantee consistency of reads for subsets of the SpecTM API [10].

Unlike best-effort HTM, or bounded-sized HTM, a program using SpecTM does not require a fallback path.

2.1 SpecTM

Our earlier paper expands on the rationale for designing SpecTM, and for providing this combination of features [10]. In outline, the overriding goal is to help us build high-performance implementations on current multi-socket multi-core shared-memory machines. Figure 3 shows the current SpecTM API:

Transactionally-managed locations are held in `TmPtr` structures. Section 2.2 discusses how different SpecTM implementations use different concrete representations for a `TmPtr`. However, from the programmer's viewpoint, a `TmPtr` encapsulates a pointer-typed value.

The `Tx_Single_*` functions perform transactions that access a single location: either read, write, or compare-and-swap. These accesses synchronize correctly with concurrent SpecTM transactions. Using a separate interface for these operations allows the implementation to optimize this frequent special case (e.g., avoiding initializing a transaction record).

The `Tx_RW_R*` operations are used for transactions that read from a series of locations, and then write new values to them all. `Tx_RW_R1` starts a trans-

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

// Read-write short transactions:
Ptr Tx_RW_R1(TX_RECORD *t, TmPtr *addr_1);
Ptr Tx_RW_R2(TX_RECORD *t, TmPtr *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, TmPtr *addr_1);
Ptr Tx_RO_R2(TX_RECORD *t, TmPtr *addr_2);
...
bool Tx_RO_1_Is_Valid(TX_RECORD *t);
bool Tx_RO_2_Is_Valid(TX_RECORD *t);
...
void Tx_RO_1_Abort(TX_RECORD *t);
void Tx_RO_2_Abort(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 3: SpecTM API for short transactions (pseudo-code).

action, and performs its first read. `Tx_RW_R2` performs its second read, and so on. Using explicit sequence numbers on the operations avoids the need for the SpecTM implementation to track the current size of the read-write set. The `Tx_RW_*_Is_Valid` functions validate a transaction that has performed the specified number of reads. The `Tx_RW_*_Commit` functions commit such a transaction, taking the new values to store at each of the locations accessed (e.g., taking 2 values in a 2-word transaction). This API forces all of a transaction's writes to be deferred until its commit point (allowing the implementation to be streamlined because a read does not need to consult a log of preceding writes).

The `Tx_RO_*` operations manage read-only transactions, in a similar manner to 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. Aside from locations that are upgraded, each `Tx_RW_R*` call and `Tx_RO_R*` call must access a distinct address.

To illustrate the use of these operations, a double-compare single-swap operation can be implemented as follows:

```
bool DCSS(TmPtr *a1, TmPtr *a2,
          Ptr o1, Ptr o2, Ptr 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;
}
```

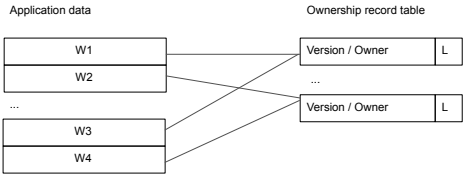
The `DCSS` function reads from the two locations supplied (`a1`, `a2`). If the values match those expected (`o1`, `o2`), then the first access is upgraded to a read-write access, and the new value (`n1`) written during commit. The transaction is repeated until either the commit succeeds, or a valid mismatch is seen.

2.2 SpecTM Implementations

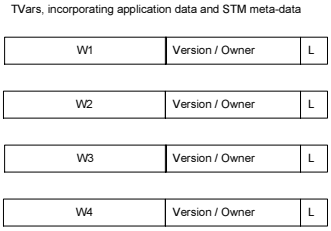
We have built three variants of SpecTM. They each implement the interface in Figure 3, but they differ in how a `TmPtr` is represented:

SpecTM-ORec. SpecTM-ORec follows the design of many general-purpose STMs in using a table of "ownership records" (ORecs) to hold the meta-data used by the STM system. A hash function maps heap addresses onto slots in a fixed-sized table of ORecs. This approach allows the STM's meta-data to be kept completely separate from the application's data. A `TmPtr` contains simply an ordinary pointer: the application's data structures do not need to be modified. A downside of this approach is that each transactional load will touch two cache lines: one to load the data, and a second to load the meta-data. Figure 4(a) illustrates this structure.

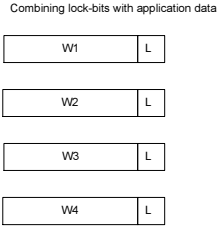
SpecTM-TVar. SpecTM-TVar follows the approach of STM-Haskell [14] by limiting the pointers passed to the STM functions to be references to specific "TVar" structures. Each `TmPtr` is a two-word TVar, holding a piece of STM meta-data alongside the piece of application data that it manages. With care, this allows



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



(b) SpecTM-TVar: Meta-data co-located with application data in `TVar`s.



(c) SpecTM-LB: One lock bit of meta-data held in each data item.

Figure 4: Organization of STM meta-data in variants of SpecTM.

both words to be held on the same cache-line. However, it requires that an application's data structure be changed to accommodate the extra words. Figure 4(b) illustrates this scheme.

SpecTM-LB. SpecTM-LB reduces the meta-data used by the STM down to a single "lock bit" held in the *same* memory word as the application data that it

controls[1]. A `TmPtr` is a single word, within which this lock bit is held as the LSB. Figure 4(c) illustrates this structure. Using a single lock bit, in place of a TVar or ORec, reduces the memory consumption of the system.

However, in order for an algorithm using SpecTM-LB to be correct, the programmer must be careful about the structure of the transactions that are used: (*i*) at most one "normal" location can be accessed by `Tx_RO_*` operations, (*ii*) any number of locations can be accessed via `Tx_RW_*` operations, and (*iii*) any number of additional read-only locations can be accessed by `Tx_RO_*` operations provided that these locations satisfy a "non-repeated value" property (NRV).

In order to satisfy NRV, the program must ensure that any particular value `v` is stored into a each location `x` at most once between the start and end of any transaction. This might be satisfied if `v` contains a sequence number that is incremented on each store to `x` (and is large enough to prevent wrapping), or if `v` is a pointer to a dynamically allocated object that is placed in a data structure, and then de-allocated using a mechanism such as those of Michael [21] and Herlihy *et al.* [16]. (Forms of NRV property have been used to support multi-word atomic snapshot algorithms, and to avoid A-B-A problems in lock-free algorithms. We use the name NRV from Lev and Moir [19].)

Our earlier paper [10] examines the performance of these different implementations of SpecTM. For the data structures we have studied, SpecTM-LB performs best, then SpecTM-TVar, then SpecTM-ORec. We believe this primarily reflects the decreasing storage requirements, with SpecTM-LB requiring the least storage, and requiring the fewest cache lines to be accessed. Conversely, SpecTM-LB places the greatest burden on the programmer by requiring the programmer to ensure that the NRV property is satisfied.

In the next two sections we illustrate the use of SpecTM to implement a hashtable and a skip list (Sections 3–4). These two designs are both correct with all of the SpecTM implementations, including SpecTM-LB. Then, in Section 5 we discuss some of the limitations of programming using SpecTM. In Section 6 we discuss how some of these limitations can be overcome by integrating SpecTM with an general-purpose STM system.

3 Hashtable

In this section we illustrate the use of SpecTM to build a hash table. For brevity, we simplify the data structure to store only integer values and to provide `search`, `insert`, and `delete` operations. We also omit memory-management code from

[1]Note that the SpecTM-LB implementation is called "version-free" or "value-based" in our earlier paper [10].

the pseudo-code—many conventional techniques are available, as discussed in our earlier paper [10].

The overall approach is based on Fraser's design [11]. We assume a fixed size array of buckets, each of which is the head of a singly linked list of elements. One value is stored in each element. The lists can be of unbounded size. Consequently, depending on the loading of the hash table, we cannot rely on using a single SpecTM transaction to traverse an entire list.

Figure 5 provides pseudo-code. The hash table is implemented as an array of `TmPtr`s (line 3) that point to the sorted lists of nodes (line 5). Each node stores a single integer value and the `TmPtr` to the next node in the list. The hash table code internally uses an iterator that stores the address of the pointer to the current node and the address of the current node during the iteration (line 9).

We structure each list using the "mark bit" technique [13]. With this technique, we reserve a bit within the "next" pointer of each node in the list, and if this bit is set then the node itself is considered to have been deleted from the lists. Comparing the SpecTM implementation of the hash table with Fraser's original lock-free design, the main benefit from using SpecTM is in the handling of deletions. SpecTM lets us simplify deletion by using a 2-word transaction to atomically (*i*) mark a node as deleted, and (*ii*) unlink the node from the list. This atomicity avoids the need for concurrent traversals of the list to consider nodes that have been marked as deleted, but not yet excised from the list. In more detail:

The main internal function of the hashtable is the function for searching for a node with a specified value (line 14). The search function is invoked by all three public hashtable functions. The arguments of the function are used to pass the identifier of the node that is being searched for and the reference of the iterator used to return the position of the node. The search first locates the bucket the element belongs to and stores the address of the bucket list head into the iterator (line 15). Next, it traverses the bucket list by following the forward pointers of the nodes in the list (line 17). While traversing the list, the search does not need to consider marked nodes (line 18). The search can only access a marked node if it already held the reference to the node before the node was marked because the node gets marked and removed in the same transaction. This means that it is safe to just read through the marked pointers as they can only be marked by the concurrent remove operation. The traversal of the list stops when either the end of the list is reached or the element with the higher or equal value is found (line 19).

To check whether the hash table contains a particular value (line 25) it is enough to search for the value (line 27). If the search stops before reaching the end of the list and the element has the value that is being looked for then the function returns `true`. Otherwise, the element is not in the hash table and it returns `false`. There is no need to check whether the node returned by the search is marked or not. If the node is marked and the lookup returns `true` it simply means that it is

```
1  const int BUCKETS = 1024;
2  struct Hashtable {
3      TmPtr buckets[BUCKETS];
4  };
5  struct Node {
6      int id;
7      TmPtr next;
8  };
9  struct Iterator {
10     TmPtr *prev;
11     Ptr curr;
12 };
13
14 void Search(Hashtable *htable, int id, Iterator *it) {
15     it->prev = GetBucket(htable, id);
16     while(true) {
17         it->curr = Tx_Single_Read(it->prev);
18         it->curr = Unmark(it->curr);
19         if(it->curr == NULL || it->curr->id >= id) {
20             break;
21         }
22         it->prev = &(it->curr->next);
23     }
24 }
25 bool Contains(Hashtable *htable, int id) {
26     Iterator it;
27     Search(htable, id, &it);
28     return (it.curr != NULL && it.curr->id == id);
29 }
30 bool Add(Hashtable *htable, Node *node) {
31     Iterator it;
32 retry:
33     Search(htable, data->id, &it);
34     if(it.curr != NULL && it.curr->id == node->id) {
35         return false;
36     }
37     TmPtrWrite(&(node->next), it.curr);
38     if(Tx_Single_CAS(it.prev, it.curr, node) != it.curr) {
39         goto retry;
40     }
41     return true;
42 }
43 bool Remove(Hashtable *htable, int id) {
44     Iterator it;
45     TX_RECORD t;
46 retry:
47     Search(htable, id, &it);
48     if(it.curr == NULL || it.curr->id != id) {
49         return false;
50     }
51 retry_tx:
52     Ptr prevNext = Tx_RW_R1(&t, it.prev);
53     if(prevNext != it.curr) {
54         Tx_RW_1_Abort(&t);
55         goto retry;
56     }
57     TmPtr *nextPtr = &(it.curr->next);
58     Ptr nextVal = Tx_RW_R2(&t, nextPtr);
59     if(!Tx_RW_2_Is_Valid(&t)) {
60         goto retry_tx;{
61     }
62     Tx_RW_2_Commit(&t, nextVal, Mark(nextVal));
63     return true;
64 }
```

Figure 5: Hashtable algorithm using SpecTM.

linearized before the concurrent remove operation.

To add a new element (line 30), a search is first performed for the node with the same value as the element being added (line 33). If the node with the same value is found (line 34), the new element is not added to the hash table. If the element is not found, the iterator points at the successor of the new element. The next pointer of the new node is updated (line 37) and the SpecTM compare-and-swap transaction is executed to try to link the new element into the list (line 38). If the compare-and-swap does not succeed the whole operation is retried. There is no need to explicitly check whether the state of the nodes where the search stopped has changed. If it has, the compare-and-swap will fail and the operation will be restarted.

To remove a node with a particular value (line 43), a search is first performed to find the node to remove (line 47). If the node is not in the hash table, the remove returns `false` immediately, indicating that the operation could not be performed. If the element is found, a SpecTM transaction is executed to unlink the node that is being removed and to mark it atomically (lines 51–62). The transaction first re-reads the pointer to the node to remove (line 52) and checks whether it has changed. If it has (line 53), that means that the state of the nodes has changed since the search and the whole operation is restarted (line 55). Otherwise, the transaction reads the next pointer of the node that is being removed (line 58). If the transaction aborts at this point (line 59) the SpecTM transaction is restarted (line 60). If the read is successful, then the transaction can commit the new values of the previous and removed nodes' next pointers (line 62).

4 Skip List

The pseudo-code of the skip list algorithm is shown in Figure 6. Each skip list node stores an integer value, and an array of forward pointers. The array holds 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 along the list, a window of pointers for all skip list levels is used (line 10).

Similarly to the hashtable, we structure skip list using the "mark bit" technique, as in Fraser's lock-free skip list [11]. When a node is removed from the list, "mark bits" at all levels of the node are set, indicating that the node is deleted. Comparing the SpecTM implementation of the skip list with Fraser's original lock-free design, the main benefit from using SpecTM is in the handling of deletions. SpecTM lets us simplify deletion by using a transaction to atomically (*i*) mark a node as deleted, and (*ii*) unlink it from the list. Similarly to the hashtable, this atomicity avoids the need for concurrent traversals of the list to

```
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(); // Using general-purpose STM
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 6: Skiplist implementation using SpecTM.

consider nodes that have been marked as deleted, but not yet excised from the list. The skip list algorithm is further simplified as the atomicity of insert and remove operations eliminates races between a concurrent insertion and removal of the same node: atomicity allows a node to be inserted/deleted at *all* levels as a single atomic action. In contrast, Fraser's lock-free skip list is further complicated by handling partially-inserted/partially-deleted nodes, and ensuring correctness when multiple operations are in progress on the same node at the same time. In more detail:

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. 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 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 SpecTM 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.

5 Limitations of SpecTM

Broadly speaking, there are two difficulties when using SpecTM. First, there is the difficulty of writing an operation using short transactions, as opposed to using transactions of arbitrary length. This is an algorithmic problem, and we do

not yet understand the trade-offs very well. Aspects of this problem might be interesting to consider from a theoretical viewpoint, as well as in terms of ease-of-programming.

Second, there is the difficulty of expressing short transactions correctly using the SpecTM API: even if an algorithm has been decomposed correctly into a series of short transactions, there is a risk that the more complex SpecTM API will admit new kinds of error when writing mini transactions. The main difficulties we have encountered are:

Sequencing. SpecTM requires operations to be invoked in the correct sequence—e.g., `Tx_RW_R1` should be called before `Tx_RW_R2`, and the `Tx_RW_*_Commit` function that is called should match the number of data accesses that have been made. To detect sequencing bugs we need only track the size of the transaction's read-set and write-set, to ensure that the addresses in different elements in the sets are disjoint, and to check that an "upgrade" operation is performed at most once on any location.

Note that the `Tx_RO_*_Abort` functions exist solely to enable this form of sequencing check. These functions are empty in non-debug builds. However, when debugging, they delimit the boundaries between SpecTM transactions and their implementation resets the statistics maintained for sequence checks.

We have not yet built a tool for checking the use of the SpecTM API statically. However, a number of aspects of the design of SpecTM should help here. First, the correctness of a series of calls to the SpecTM API depends primarily on the names of the function being called, and on the set of functions that have previously been called. This means that a simple intra-procedural forwards data-flow analysis should be sufficient for tracking most usage. Note that this tool would not check the disjointness between the items in the read-set and write-set.

Validation. A more difficult aspect of the SpecTM API is the requirement to include calls to the `Tx_*_Is_Valid` functions whenever it is necessary to ensure that a transaction has seen a consistent view of memory. This dangers of working with inconsistent data have been reported in many earlier STM systems [15]. In part, these dangers led to the proposal for opacity as a correctness criteria for transactional memory. Approaches taken in earlier systems have included implicit validation as part of every transactional read [15], timestamp-based mechanisms to eliminate some of these validation steps [23, 27], along with static analyses to identify "safe" regions during which validation can be deferred [26]. For instance, if a thread performs a series of independent reads then validation may be deferred to the end of the sequence.

When programming using SpecTM, we rely on the programmer placing vali-

dation calls where necessary. Unfortunately, characterizing precisely what "where necessary" means is not straightforward. There are two broad options:

- First, we could place a very strong requirement on the part of the programmer, and permit *all* `Tx_RW_*` and `Tx_RO_*` operations to return *any* value. Validation must return `false` if the values returned by these operations do not represent a consistent view of memory.

  This definition allows a debug build to return "spurious" values which, in the absence of validation, are likely to lead to crashes—for instance, pointer-typed operations could probabilistically return addresses that are not mapped to valid memory. This definition is reminiscent of "catch fire" semantics for programs with data races (as in, for instance, the C/C++ memory model [4]). An advantage of this approach is that it provides a clear definition to the programmer of the behavior of invalid programs, and it makes it likely that crashes in debugging implementations of SpecTM will identify missing validation operations.

  Even with this definition, a programmer using SpecTM can still optimize the placement of validation calls. For instance, a single validation call may be used after a series of unrelated memory reads. Validation must be performed before de-referencing a pointer read from within a transaction, or before access an address computed from a value read.

- Second, we could define the semantics of `Tx_RW_*` and `Tx_RO_*` more strongly, and constrain exactly what they should do in the presence of invalidity—for instance, we might require that a value that was present in the location at some time in the past is returned, or we might require that a value present during the current transaction is returned.

  This second style of definition may allow the programmer to use slightly fewer validation operations, and hence obtain some performance improvement.

  A disadvantage of this approach is that the requirements on programmers are less clear, and it seems more difficult to build checking tools. The crux of the problem is distinguishing between cases in which validation has been missed accidentally, from cases where validation has been omitted deliberately to exploit a particularly subtle optimization. It is unclear how to distinguish these cases without a specification of the intended higher-level behavior of the program.

  There are many plausible definitions for the behavior of reads in invalid transactions—in contrast, the extreme approaches of "catch fire" and "opacity" are both relatively straightforward.

We currently take the "catch fire" approach, and have transactions in debug builds probabilistically appear invalid spuriously. We are not currently aware of any optimization opportunities that we are missing through this approach.

Non-repeating values (NRV). The final difficulty we highlight with SpecTM is the NRV property required of some locations in transactions in SpecTM-LB. In effect, NRV is shifting the responsibility of managing version numbers from the STM system to the programmer using the STM. A direct way to support NRV would be for each value to include a version number field that is incremented upon every write. This is typical of most STM algorithms, and SpecTM-LB's support for general NRV locations can be seen as a mechanism to exploit other forms of non-re-use, rather than just using a version number.

We do not currently have a good way to test that a program's use of memory satisfies NRV. The approaches that we have considered seem prohibitively costly, even for use in debugging builds. For instance, one could adapt the transactional write operations to maintain a history log for each location, and arrange that each SpecTM-LB transaction checks for re-use of values in these logs for the locations that it has read from. The cost of logging and checking is likely to be very high.

It might be possible to adapt this approach to perform checking probabilistically—either in terms of whether or not to log an update, or in terms of whether or not to perform checks at commit-time. It is not yet clear if these reduced checks would be sufficient to catch re-use. Equally, it is not clear if even the full checking regime would catch re-use bugs—it relies on spotting an occurrence of re-use, and so will not be useful for detecting problems that occur rarely. A further alternative would be to log additional information about the synchronization between threads, and to use this to spot "near miss" occasions of re-use, where a program contains a re-use bug, but where this is not witnessed by a SpecTM-LB transaction in a given run.

6 Integration with General-Purpose TM

From a pragmatic viewpoint, the main way in which we address the limitations in Section 5 is to enable inter-operation between transactions written with SpecTM and transactions written through a general-purpose TM interface. This reduces the amount of code that must be written using SpecTM: the programmer can use SpecTM to optimize performance-critical transactions, and use the general-purpose interface for code that is more complex.

Integration between SpecTM and a general-purpose TM can be implemented either by having both TM systems manage disjoint sets of memory locations, or by designing mechanisms to allow the two types of transaction to access the same

data correctly. In the former case, transactions written with SpecTM and with the general-purpose STM can be composed in a manner similar to *two-phase commit*: both transactions are first prepared for commit and, only after the prepare is successful, they are committed together. If one of the transactions aborts, then the other transaction gets aborted as well. In this manner the composite transaction is also atomic. With this approach, the general-purpose transaction can benefit from optimized implementations of data structures with SpecTM, but the data structures cannot be implemented with a mix of specialized and general-purpose transactions. The latter approach enables the programmer to have the specialized and general-purpose transaction access the same data concurrently. For instance, one could use a general-purpose transaction to re-size a hash table, while using SpecTM transactions for the common cases of data accesses. We focus on the latter form of inter-operability as it is more general and more interesting when implementing concurrent data structures.

In SpecTM we use two kinds of specialization to attempt to streamline the implementation: (*i*) the SpecTM API requires more work on the part of the software using STM, in an attempt to reduce the work needed within the SpecTM implementation, (*ii*) the representation of `TmPtr` structures in SpecTM-TVar and SpecTM-LB attempts to reduce the space occupied by the TM meta-data.

The different `TmPtr` representations introduce different constraints on integration between SpecTM and a general-purpose STM. Concretely, we use a general-purpose STM we refer to as "BaseTM". This uses similar techniques to SwissTM [9] and the STM described by Spear *et al.* [25].

SpecTM-ORec. Our SpecTM-ORec implementation uses the same protocol for managing the ORecs as the BaseTM system. No additional work is required, either on the SpecTM-ORec transactions, or on the BaseTM transactions.

SpecTM-TVar. SpecTM-TVar changes the way in which transactional data is represented. This prevents BaseTM from being used directly on the same data: the SpecTM transactions would be using ORecs held alongside the data in transactional variables in `TmPtr` structures (Figure 4(b)), whereas the BaseTM transactions would be using ORecs held in the usual ORec table (Figure 4(a)).

There are two main approaches for integrating SpecTM-TVar with general-purpose transactions.

Hybrid-TM-style. The first approach is to build on earlier techniques for hybrid HTM/STM systems [18, 6]. In Hybrid-TM models, the HTM is used to provide good performance, while an STM serves as a backup to handle situations where the HTM could not execute the transaction successfully. This approach may reduce the pressure on HTM implementations to provide features such as

long-running transactions, or conditional blocking.

Unfortunately, many Hybrid-TM techniques are not good fits for SpecTM. The problem is that the SpecTM-TVar transactions would be required to monitor the BaseTM transactions for conflicts (because the BaseTM transactions are unaware of the TVars being used by SpecTM-TVar). This additional monitoring would harm the performance of SpecTM transactions.

One hybrid technique that *is* practical to use is Lev *et al.*'s Phased TM system (PhTM, [20]). Instead of requiring hardware transactions to check for conflicts with concurrent software transactions, PhTM prevents HW and SW transactions from executing concurrently. The PhTM system maintains a counter of currently-executing SW transactions. Every HW transaction checks this counter, and if non-zero, the HW transaction aborts itself. Since the counter is also read inside the HW transaction, any subsequent modifications to this counter also trigger an abort. This approach reduces overheads for HW transactions, but it results in increased aborts (as discussed by Baugh *et al.* [2]): an overflow of even a single HW transaction aborts *all* other concurrently executing HW transactions.

Haskell-STM-style. An alternative to a phased-TM system is to adapt the BaseTM interface to use TVars. Unlike SpecTM, these general-purpose transactions can access an unbounded number of locations, and they do not need to provide sequence numbers on their accesses, or to distinguish read-only locations from read-write locations. However, as with SpecTM-TVar, all of a transaction's data accesses must be to locations with associated TVars. With this interface, the meta-data used by the STM system is the same as the meta-data used by SpecTM-TVar.

Whether or not this approach is practical will depend on the setting. It seems most palatable when writing new data structures using transactions: it requires the representation of the data to be able to be adapted to include TVars, and so it would be inappropriate for existing data structures, or those which are sometimes used inside transactions and sometimes used outside.

SpecTM-LB. SpecTM-LB uses only a single lock bit within each of the locations managed by the STM. The problem now is not that SpecTM's meta-data is in a different place to BaseTM's, but that the actual format of the meta-data is different. Again, two approaches are possible:

Phased-TM. As with SpecTM-TVar, we can use the techniques of Lev *et al.* to ensure that SpecTM-LB and BaseTM transactions do not run concurrently [20]. An advantage of this approach is that the only overhead on SpecTM-LB transactions is to ensure that execution is in a "SpecTM phase". A disadvantage is that, if even a single thread wishes to use general-purpose transactions, then *all* SpecTM-LB transactions must be prevented from running, irrespective of the data that they

are accessing.

Locked-by-Software. The intuition behind Phased-TM is that two different TM systems can co-exist without shared conflict detection mechanisms so long as they are separate in time. An alternative form of STM integration for SpecTM-LB is to keep the locations managed by SpecTM separate in space from those managed by BaseTM. That is, both kinds of STM can co-exist, so long as they are accessing disjoint sets of locations.

If lock bits can be reserved in all transactional data, then these can be used to control the separation between SpecTM-LB and BaseTM. If the bit is set then either (*i*) the location is currently part of a current SpecTM transaction's write set, or (*ii*) the location is currently owned for writing by BaseTM. If the bit is clear, then the location is available for reading by either kind of transaction.

This approach avoids intruding on the fast path of SpecTM-LB transactions that run and commit without conflict—all of the locations that they encounter will have the lock bit clear, and all of the additional work to integrate with BaseTM will be on the existing slow-path for when the lock bit is set. Conversely, BaseTM must ensure that it has ownership of all of the locations it is accessing by setting the lock bit before accessing them.

The main complexity with this use of the lock bit is how to arrange for BaseTM to release the lock bit in order to allow SpecTM-LB transactions to acquire it. Our current design is:

- BaseTM eagerly acquires the lock bit when executing a transaction.
- BaseTM releases the lock bit only when requested by a SpecTM-LB transaction that wishes to access the same location.
- If a SpecTM-LB transaction finds that the lock bit is held by a BaseTM transaction, then the thread running the SpecTM-LB transaction executes a "dummy" BaseTM transaction on the location. The dummy transaction synchronizes with other BaseTM transactions (ensuring no other writers are present) before releasing the lock bit back to SpecTM-LB's use.

This approach avoids repeated updates to the lock bit when a location is accessed by a series of BaseTM transactions.

7 Discussion

In this paper we have discussed the design of the SpecTM interface, illustrated its use in constructing shared memory data structures, and discussed some of

the shortcomings of SpecTM, along with techniques for integrating the different forms of SpecTM with general-purpose STM systems.

The limitations in Section 5 all reflect additional requirements that are placed on the programmer when using the SpecTM API rather than when using a traditional TM API. We believe that two of these limitations are relatively straightforward for the programmer to handle: The problem of sequencing the invocation of SpecTM operations correctly is straightforward to check dynamically, and appears relatively amenable to static analysis. The problem of calling validation functions correctly follows the existing problem of correctly sandboxing programs using TM systems without opacity in languages such as C/C++. In addition, in each of these two cases, it seems likely that straightforward testing tools could check that a program is constructed correctly, or that a compiler could target the SpecTM API correctly for programs whose workloads are suitable.

However, it is unclear if the additional performance benefits of exploiting the non-repeating value property (NRV) are sufficient for the additional complexity in using SpecTM-LB. This is the one setting in which we do not have a good checking tool (static or dynamic), and in which there seems to be a great risk of programmers making accidental errors in their use of SpecTM. In future work we would like to study this problem more closely—e.g., is it possible to provide a sufficiently expressive set of "NRV-safe" abstractions that guarantee that the NRV property will be satisfied, and is it possible to develop checking techniques that are sufficiently lightweight to be used in practice?

References

[1] Hagit 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] Lee Baugh, Naveen Neelakantam, and Craig Zilles. Using hardware memory protection to build a high-performance, strongly atomic hybrid transactional memory. In *ISCA '08: Proc. 35th Annual International Symposium on Computer Architecture*, pages 115–126, June 2008.

[3] Colin Blundell, E. Christopher Lewis, and Milo M. K. Martin. Subtleties of transactional memory atomicity semantics. *Computer Architecture Letters*, 5(2), November 2006.

[4] Hans-J. Boehm and Sarita V. Adve. Foundations of the C++ concurrency memory model. In *PLDI '08: Proc. 2008 ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 68–78, June 2008.

[5] Dave Christie, Jae-Woong Chung, Stephan Diestelhorst, Michael Hohmuth, Martin Pohlack, Christof Fetzer, Martin Nowack, Torvald Riegel, Pascal Felber, Patrick

Marlier, and Etienne Riviere. Evaluation of AMD's advanced synchronization facility within a complete transactional memory stack. In *EuroSys '10: Proc. 5th ACM European Conference on Computer Systems*, April 2010.

[6] Peter Damron, Alexandra Fedorova, Yossi Lev, Victor Luchangco, Mark Moir, and Dan Nussbaum. Hybrid transactional memory. In *ASPLOS '06: Proc. 12th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 336–346, October 2006.

[7] Dave Dice, Yossi Lev, Mark Moir, and Daniel 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, March 2009.

[8] Simon Doherty, Lindsay Groves, Victor Luchangco, and Mark Moir. Towards formally specifying and verifying transactional memory. *Formal Aspects of Computing*, pages 1–31, 2012.

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

[10] Aleksandar Dragojević and Tim Harris. STM in the small: trading generality for performance in software transactional memory. In *EuroSys '12: Proc. 7th European conference on Computer systems*, April 2012.

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

[12] Rachid Guerraoui and Michał 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, February 2008.

[13] Tim Harris. A pragmatic implementation of non-blocking linked-lists. In *DISC '01: Proc. 15th International Conference on Distributed Computing*, 2001.

[14] Tim Harris, Maurice Herlihy, Simon Marlow, and Simon Peyton Jones. Composable memory transactions. In *PPoPP '05: Proc. ACM Symposium on Principles and Practice of Parallel Programming*, June 2005. A shorter version appeared in *CACM* 51(8):91–100, August 2008.

[15] Tim Harris, James Larus, and Ravi Rajwar. *Transactional Memory, 2nd edition*. Morgan & Claypool, 2010.

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

[17] Maurice Herlihy and J. Eliot 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] Sanjeev Kumar, Michael Chu, Christopher J. Hughes, Partha Kundu, and Anthony Nguyen. Hybrid transactional memory. In *PPoPP '06: Proc. 11th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, March 2006.

[19] Yossi Lev and Mark Moir. Lightweight parallel accumulators using C++ templates. In *IWMSE '11: Proc. 4th International Workshop on Multicore Software Engineering*, 2011.

[20] Yossi Lev, Mark Moir, and Dan Nussbaum. PhTM: Phased transactional memory. In *TRANSACT '07: 2nd Workshop on Transactional Computing*, August 2007.

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

[22] James Reinders. Transactional synchronization in Haswell, February 2012. `http://software.intel.com/en-us/blogs`.

[23] Torvald Riegel, Pascal Felber, and Christof Fetzer. A lazy snapshot algorithm with eager validation. In *DISC '06: Proc. 20th International Symposium on Distributed Computing*, volume 4167 of *Lecture Notes in Computer Science*, pages 284–298, September 2006.

[24] Nir Shavit and Dan Touitou. Software transactional memory. In *PODC '95: Proc. 14th ACM Symposium on Principles of Distributed Computing*, pages 204–213, August 1995.

[25] Michael F. Spear, Luke Dalessandro, Virendra J. Marathe, and Michael 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, February 2009.

[26] Michael F. Spear, Virendra J. Marathe, William N. Scherer III, and Michael L. Scott. Conflict detection and validation strategies for software transactional memory. In *DISC '06: Proc. 20th International Symposium on Distributed Computing*, September 2006.

[27] Cheng Wang, Wei-Yu Chen, Youfeng Wu, Bratin Saha, and Ali-Reza Adl-Tabatabai. Code generation and optimization for transactional memory constructs in an unmanaged language. In *CGO '07: Proc. International Symposium on Code Generation and Optimization*, pages 34–48, March 2007.