# MSpec: A Design Pattern for Concurrent Data Structures [*]

Lingxiang Xiang     Michael L. Scott

Computer Science Department, University of Rochester

{lxiang, scott}@cs.rochester.edu

## Abstract

Speculation is a well-known means of increasing parallelism among concurrent methods that are usually but not always independent. Traditional nonblocking data structures employ a particularly restrictive form of speculation. Software transactional memory (STM) systems employ a much more general—though typically blocking—form, and there is a wealth of options in between.

We explore data-structure-specific speculation as a design pattern for concurrent data structures. Using several different structures as examples, we consider issues of safety (sandboxing), validation mechanism, and granularity of locking. We note that it can sometimes be useful to perform validation and locking at different granularities. Through experiments on UltraSPARC and x86 platforms, we demonstrate that MSpec can lead to highly efficient algorithms, particularly in methods with a significant search component.

## 1. Introduction

Concurrent data structures play a key role in multithreaded programming. Typical implementations use locks to ensure the atomicity of method invocations. Locks are often overly pessimistic: they prevent threads from executing at the same time even if their operations don't actually conflict. Finer grain locking can reduce unnecessary serialization at the expense of additional acquire and release operations. For data that are often read but not written, reader-writer locks will allow non-mutating methods to run in parallel. In important cases, RCU [16] may additionally eliminate all or most of the overhead of reader synchronization. Even so, the typical concurrent data structure embodies an explicit compromise between, on the one hand, the overhead of acquiring and releasing extra locks and, on the other hand, the loss of potential concurrency when logically nonconflicting method calls acquire the same lock.

In a different vein, *nonblocking* concurrent data structures are inherently optimistic. Their dynamic method invocations always include an instruction that constitutes their *linearization point*. Everything prior to the linearization point is (speculative) preparation, and can be repeated if necessary without compromising the correctness of other method invocations. Everything subsequent to the linearization point is "cleanup," and can typically be performed by any thread.

The usual motivation for nonblocking data structures is to avoid performance anomalies when a lock-holding thread is preempted or stalled. For certain important (typically simple) data structures, average-case performance may also improve: in the absence of conflicts (and consequent misspeculation), the reduction in serial work may outweigh the increase in (non-serial) preparation and cleanup work. Unfortunately, for more complex data structures the tradeoff tends to go the other way and, in any event, the creation of efficient nonblocking algorithms is notoriously difficult.

Transactional memory, by contrast, places the emphasis on ease of programming. Few implementations are nonblocking, but most are optimistic. With hardware support, TM may provide performance as good or better than that of the best-tuned fine-grain locking. For application code, written by non-experts, even software TM (STM) may outperform coarse-grain locking. For concurrent data structures in libraries, however, STM seems unlikely ever to be fast enough to supplant lock-based code written by experts.

But what about speculation? Recent work by Bronson et al. [2] demonstrates that hand-written, data-structure-specific speculation can provide a significant performance advantage over traditional pessimistic alternatives. Specifically, the authors describe a relaxed-balance speculative AVL tree that outperforms the `java.util.concurrent .ConcurrentSkipListMap` by 32–39%. Their code employs a variety of highly clever optimizations. While fast, it is very complex, and provides little guidance for the construction of other hand-written speculative data structures.

Our work takes a different tack. Drawing inspiration (loosely) from the NOrec STM algorithm [4], we present a general design pattern, Manual SPECulation(MSpec), for concurrent data structures. Following MSpec, programmers can, with modest effort, add speculation to existing lock-based code, or write new versions based on sequential code. While less efficient than the work of Bronson et al., MSpec is not limited to a particular data structure, and does not require dramatic code mutation.

Our intent is to establish manual speculation as a widely recognized complement to locking in the concurrent pro-

[*] This work was supported in part by the National Science Foundation under grants CCR-0963759, CCF-1116055, and CNS-1116109.

grammer's toolbox. Using four example data structures—*equivalence sets*, *$B^{link}$-trees*, *cuckoo hash tables*, and *linear bitmap allocators*—we show how to tune speculation and the related tasks of sandboxing and validation to obtain short critical sections and low overhead at different locking granularities. We note that in contrast to STM, MSpec makes it easy to perform locking and validation at different granularities. Subjectively, we also find that MSpec does not introduce significant code complexity relative to equivalent lock-based implementations, and it may even simplify the locking protocol for certain data structures.

In experiments on UltraSPARC and x86 platforms, we find that MSpec outperforms pessimistic code with similar or (in many cases) finer granularity locking. The advantage typically comes both from reducing the overall number of atomic (read-modify-write) instructions and from moving instructions and—more importantly—cache misses out of critical sections (i.e., off the critical path) and into speculative computation.

## 2. Motivating Example: Equivalence Sets

We illustrate the use of MSpec style with a concurrent implementation of equivalence sets. An instance of the data structure partitions some universe of elements into a collection of disjoint sets, where the elements of a given set have some property in common. For illustrative purposes, the code of Figure 1 envisions sets of integers, each represented by a sorted doubly-linked list. Two methods are shown. The `Move` method moves an integer from one set to another. The `Sum` method iterates over a specified set and returns some aggregate result. We have included a `set` field in each element to support a constant-time `MemberOf` method (not shown). Conventional lock-based code is straightforward: each method comprises a single critical section protected (in our code) by a single global lock.

### 2.1 A Speculative Implementation

The speculative version of `Sum` exploits the fact that the method is read only. By adding a version number to each set, we obtain an obstruction-free [10] `SpecSum`. If we iterate over the entire set without observing a change in its version number, we know that we have seen a consistent snapshot. As in most STM systems, failed *validation* aborts the loop and starts over. In MSpec, however, the programmer is responsible for any state that must be restored (in this case, none).

The baseline critical section in `Move` begins by removing element `e` from its original list (lines 15–16). An appropriate (sorted) position is then found in the target list `s` (lines 17–19). Finally, `e` is inserted into `s` at the chosen position (lines 20–23).

The position-finding part of `Move` is read only, so it can be performed speculatively in `SpecMove`—that is, before entering the critical section. The validation at line 51 ensures that the `next` element remains in the same set and no new element has been inserted between `prev` and `next` since line 49. To reflect changes to the circular list, a set's version number is increased both before and after a modification. An odd version number indicates that an update is in process, preventing other threads from starting new speculations (line 30).

### 2.2 Performance Results

### 2.2.1 Experimental Platforms

We tested our code on a Sun Niagara 2 and an Intel Xeon E5649. The Sun machine has two UltraSPARC T2+ chips, each with 8 in-order, 1.2 GHz, dual-issue cores, and 8 hardware threads per core (4 threads per pipeline). The Intel machine also has two chips, each with 6 out-of-order, 2.53 GHz, hyper-threaded cores, for a total of 24 hardware thread contexts. Code was compiled with `gcc` 4.4.5 (`-O3`).

To measure throughput, we arrange for a group of worker threads to repeatedly call randomly chosen methods of the data structure for a fixed period of time (1 second). We bind each thread to a logical core to eliminate thread migration, and fill all thread contexts on a given chip before employing multiple chips.

Unless otherwise specified, all mutex locks used in this paper are test-and-`test_and_set` locks with exponential back-off, tuned individually for the two experimental machines.

### 2.2.2 Test Configurations

We compare six different implementations of equivalence sets.

**CGL** is the coarse-grained "baseline" code of Figure 1. The single global lock is shared by all sets, and becomes a bottleneck at high thread counts.

**Spec-CGL** is the speculative code of Figure 1. Its critical sections are shorter than those of CGL.

**FGL** is a fine-grained locking version with a single lock per set. It is slightly more complex than CGL because the critical section in `Move` must acquire two locks (in canonical order, to avoid deadlock).

**Spec-FGL** is a speculative implementation of FGL. It performs locking and validation at the same granularity, so we combine the lock and version number into a single field, eliminating the four version number increments in the critical section.

**TinySTM** and **$\varepsilon$-STM** are transactional analogues of CGL. TinySTM employs the RSTM [1] OrecEager back end, which emulates TinySTM [6]. $\varepsilon$-STM employs elastic transactions [7], which are optimized for search structures. Loads and stores of shared locations were hand-annotated.

### 2.2.3 Scalability

Performance results appear in Figures 2 and 3. We used 50 equivalence sets in all cases, with either 500 or 5000 elements in the universe (10 or 100 per set, on average).

```
1   int Sum (Set *s) {
2     int sum = 0;
3     globalLock.acquire();
4     Element *next = s→head→next;
5     while (next != s→head) {
6       sum += next→value;
7       next = next→next;
8     }
9     globalLock.release();
10    return sum;
11  }

13  void Move (Element *e, Set *s) {
14    globalLock.acquire();
15    e→prev→next = e→next;
16    e→next→prev = e→prev;
17    Element *next = s→head;
18    while (next→value < e→value)
19      next = next→next;
20    next→prev→next = e;
21    next→prev = e;
22    e→next = next;
23    e→set = s;
24    globalLock.release();
25  }
```

```
26  int SpecSum (Set *s) {
27   again:
28    int sum = 0;
29    int ver = s→ver;
30    if (ver & 1) goto again;
31    Element *next = s→head→next;
32    while (next != s→head && ver == s→ver) {
33      sum += next→value;
34      next = next→next;
35    }
36    if (ver != s→ver) goto again;
37    return sum;
38  }

40  void SpecMove (Element *e, Set *s) {
41   again:
42    Element *prev = s→head;
43    Element *next = prev→next;
44    while (next→value < e→value) {
45      prev = next;
46      next = next→next;
47      if (prev→set != s)
48        goto again;
49    }
```

```
50    globalLock.acquire();
51    if (next→set != s || next→prev != prev) {
52      globalLock.release();
53      goto again;
54    }
55    e→set→ver++;
56    e→prev→next = e→next;
57    e→next→prev = e→prev;
58    e→set→ver++;
59    s→ver++;
60    next→prev→next = e;
61    next→prev = e;
62    e→next = next;
63    e→set = s;
64    s→ver++;
65    globalLock.release();
66  }
```

**Figure 1.** Coarse-grained lock implementation of concurrent equivalence sets. `Sum` and `Move` are baseline code, while `SpecSum` and `SpecMove` are code written in MSpec style. The `value` field of each set's head is initialized to $+\infty$ to avoid loop bound checks in lines 18 and 44. For simplicity, memory fences are omitted.

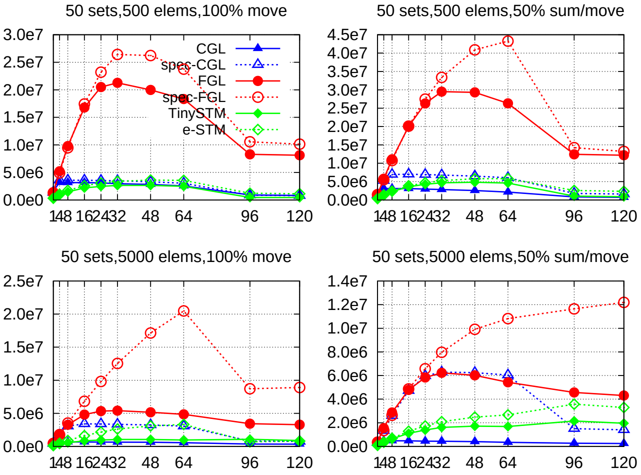


**Figure 2.** Throughput of concurrent equivalence sets on Niagara 2. The X axis is the number of concurrent threads; the Y axis is method invocations per second.

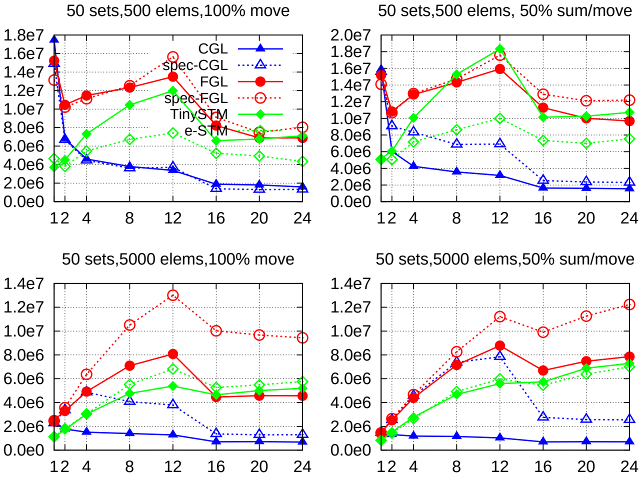


**Figure 3.** Throughput (invocations/s) of concurrent equivalence sets on Intel Xeon E5649.

We use 100% `Move` operations to simulate a write-dominant workload, and a 50/50 mix of `Move` and `Sum` operations to simulate a mixed but higher-contention workload. The number of elements per set determines the amount of work that can be moved out of the critical section in `SpecMove`.

Figure 2 illustrates the scalability of our five implementations for different workloads on Niagara 2. As expected, FGL outperforms CGL in all tests, and spec-FGL outperforms spec-CGL. Since an invocation of the `Move` method holds 2 locks simultaneously, FGL reaches its peak throughput when the thread count is around 32. The sharper performance drop after 64 threads is due to cross-chip communication costs.

MSpec improves scalability for FGL, and throughput for both CGL and FGL. In the 5000-element, 50/50 sum/move case, Spec-CGL even outperforms FGL by a tiny margin, out to 64 threads. This suggests that simple coarse-grained locking with speculation could be an attractive alternative to fine-grained locking for workloads with significant work amenable to speculation.

The baseline overhead of Spec-FGL, measured by comparing to FGL on a single thread, is less than 10%. Therefore, even without contention, Spec-FGL can deliver competitive performance. By contrast, both TinySTM and $\varepsilon$-STM have significant baseline overhead (2–4× slower than CGL on a single thread), and outperform only CGL.

Results on the Intel machine resemble those on Niagara 2, with a couple of interesting exceptions. First, single-thread performance is significantly higher in the 500-element case, where the data set can almost fit in the 32KB L1 data cache. Second, STM, while still slower than spec-FGL in most cases, performs much better on the Intel machine than it does on Niagara 2. We attribute the difference to the lower cost of atomic (CAS) instructions, which reduces the penalty for locking ownership records, and to out-of-order execution, which allows much of the STM instrumentation to execute in parallel with “ordinary” computation.

# 3. Principles of Coding in MSpec Style

The previous section showed how to convert a particular lock-based data structure to the MSpec style. This section generalizes on the example, and discusses three key design principles.

## 3.1 Code Skeleton

The code skeleton for a generic MSpec critical section appears in Figure 4. The key idea is to move work from the original critical section into speculation. Once the speculative phase has “figured out what it wants to do,” the actual writeback occurs under the protection of one or more locks. To avoid erroneous write-back, the critical section must begin by *validating* its view of memory (ensuring it was consistent). If inconsistency can lead to erroneous behavior during the speculative phase, additional validation operations must be inserted there as well. Exactly what must be checked to ensure consistency—and which locks acquired to ensure serializability—is the programmer’s responsibility.

```
function SpecOp
  again:
    if (++ try_counter > MAX_TRY_NUM)
      return NonSpecOp
    ... // speculative phase
    if any validation in speculation phase fails
      goto again
    acquire (locks)
    if ( validation fails )
      restore any necessary state
     goto again
    ... // nonspeculative phase
    release (locks)
```

**Figure 4.** Code skeleton of concurrent operation in MSpec style.

This “parallel work, serial writeback” idiom is characteristic of many STM systems. MSpec differs from these systems in two key ways. First, in most STM systems, programmers cannot control the division between speculative and nonspeculative work. In MSpec, programmers can move work that is unlikely to cause conflicts into the speculative section and keep the rest in the critical section. Second, validation in STM systems, because it must be provably safe in the general case, is generally both costly and frequent. In MSpec, the validation mechanism is invoked only when necessary, and can exploit the programmer’s understanding of the specific data structure at hand.

If a method has no side effects (does not update the data structure), it may be possible to move everything into the speculative phase, and elide the critical section altogether. We saw an example of this with the Sum method in Section 2. If all read-only methods become speculative, we may be able to replace a reader-writer lock with a simple mutual exclusion lock.

## 3.2 What can be done in speculative code?

Generally, any work that does not move shared data into an inconsistent state can be speculatively executed. Since the speculative work typically comes from the original critical section, intuitively, the more we are able to do in speculation, the shorter the critical path should be at run time. The principal caveat is that too large an increase in total work—e.g., due to misspeculation or extra validation—may change the critical path, so that the remaining critical section is no longer the application bottleneck. (Speculation may also have an impact on overall system throughput or energy consumption, but we do not consider those issues here.)

In general, a to-be-atomic method may consist of several logical steps. These steps may have different probabilities of conflicting with the critical sections of other method invocations. The overall conflict rate (and hence abort rate) for the speculative phase of a method is bounded below by the abort rate of the most conflict-prone step. Steps with a high conflict rate may therefore best be left in the critical section.

There are several common code patterns in concurrent data structures. Collection classes, for example, typically provide *lookup*, *insert*, and *remove* methods. *Lookup* is typically read-only, and may often be worth executing entirely in speculation. *Insert* and *remove* typically start with a search to see whether the desired key is present. The search is independent of other computation and thus can be speculative as well. In resource managers, an *allocate* method typically searches for free resources in a shared pool before actually performing allocation. The searching step can be moved to speculation. In other data structures, time-consuming logical or mathematical computations, such as compression and encryption, are also good candidates for speculation.

At least three factors at the hardware level can account for a reduction in execution time when speculation is successful. First, MSpec may lead to a smaller number of instructions on the program’s critical path, assuming this consisted largely of critical sections. Second, since the speculative phase and the following critical section usually work on similar data sets, speculation can serve as a data prefetcher, effectively moving cache misses off the critical path. This can improve performance even when the total number cache misses per method invocation stays the same (or even goes up). Within the limits of cache capacity, the prefetching effect increases with larger working sets. Third, in algorithms with fine-grain locks, speculation may reduce the number of locks that must be acquired, and locks are quite expensive on many machines. We will return to these issues in more detail in Section 4.

### 3.3 How do we validate?

Validation is the most challenging and flexible part of MSpec. Most STM systems validate after every shared-memory load, to guarantee *opacity* (mutual consistency of everything read so far) [8]. Heuristics such as a global commit counter [22] or per-location timestamps [5, 20] may allow many validation operations to complete in constant time, but the worst-case cost is typically linear in the number of shared locations read so far. (Also: per-location timestamps aren't *privatization safe* [15].) As an alternative to opacity, an STM system may *sandbox* inconsistent transactions by performing validation immediately before any "dangerous" instruction, rather than after every load [3], but for safety in the general case, a very large number of validations may still be required.

In MSpec, we can exploit data-structure-specific programmer knowledge to minimize both the number of validations and their cost. Determining when a validation is necessary is a tricky affair; we consider it further in the following subsection. To minimize the cost of individual validations, we can identify at least two broadly useful idioms.

***Version Numbers (Timestamps):*** While STM systems typically associate version numbers with individual objects or ownership records, designers of concurrent data structures know that they can be used at various granularities [2, 12]. Regardless of granularity, the idea is the same: if an update to location $l$ is always preceded by an update to the associated version number, then a reader who verifies that a version number has not changed can be sure that all reads in between were consistent.

It is worth emphasizing that while STM systems often conflate version numbers and locks (to minimize the number of metadata updates a writer must perform), versioning and locking serve different purposes and may fruitfully be performed at different granularities. In particular, we have found that the number of locks required to avoid over-serialization of critical sections is sometimes smaller than the number of version numbers required to avoid unnecessary aborts. The code of Figure 1, for example, uses a single global lock, but puts a version number on every set. With a significant number of long-running readers (the lower-right graphs in Figures 2 and 3), fine-grain locking provides little additional throughput at modest thread counts, but a single global version number would be disastrous. For read-mostly workloads (not shown), the effect is even more pronounced: fine-grain locking can actually hurt performance, but fine-grain validation is essential.

***In-place Validation:*** In methods with a search component, the "right" spot to look up, insert, or remove an element is self-evident once discovered: *how* it was discovered is then immaterial. Mechanisms like "early release" in STM systems exploit this observation [11]. In MSpec, we can choose to validate simply by checking the local context. An example appears at line 51 of Figure 1, where `next→set` and `next→prev` are checked to ensure that the two key nodes are still in the same set, and adjacent to one another. When it can be used, in-place validation has low overhead, a low chance of aborts, and zero additional space overhead.

### 3.4 What can go wrong and how do we handle it?

Speculative loads, almost by definition, constitute data races with stores in the critical sections of other threads. At the very least, speculative code must be prepared to see shared data in an inconsistent state. On some machines, with some compilers, it may also see "out of thin air" values or other violations of sequential consistency. Guaranteeing safety in the face of data races is a significant challenge for MSpec, but one that is shared by various other concurrent programming tasks—notably, the implementation of synchronization itself. Informally, we expect to be safe if validation operations include compiler and memory fences that serialize them with respect to other code in the same thread and that guarantee, when the validation succeeds, that no *dynamic* data race occurred. The precise specification of sufficient conditions is a subject of future work.

In general, our approach to safety is based on sandboxing rather than opacity. It requires that we identify "dangerous" operations and prevent them from doing any harm. Potentially dangerous operations include the use of incorrect data values, incorrect or stale data pointers, and incorrect indirect branches. Incorrect data can lead to faults (e.g., divide-by-zero) or to control-flow decisions that head into an infinite loop or down the wrong code path. Incorrect data pointers can lead to additional faults or, in the case of stores, to the accidental update of nonspeculative data. Incorrect indirect branches (e.g., through a function pointer or the vtable of a dynamically chosen object) may lead to arbitrary (incorrect) code.

An STM compiler, lacking programmer knowledge, must be prepared to validate before every dangerous instruction—or at least before those that operate on values "tainted" by speculative access to shared data. In a few cases (e.g., prior to a division instruction or an array access) the compiler may be able to perform a value-based sanity check that delays the need for validation. In MSpec, by contrast, we can be much more aggressive about reasoning that the "bad cases" can never arise (e.g., based on understanding of the possible range of values stored to shared locations by other threads). We can also employ sanity checks more often, if these are cheaper than validation (after following a child pointer in a tree, for example, we might check to see whether the target node's parent field points back to where we came from). Both optimizations may be facilitated by using a *type-preserving allocator*, which ensures that deallocated memory is never reused for something of a different type [18].

One final issue, shared with STM, is the possibility of starvation in the face of repeated misspeculation. This is most easily addressed by reverting to a nonspeculative version of the code after some maximum number of aborts.

## 4. Additional Case Studies

This section outlines the use of MSpec in three additional concurrent data structures, and summarizes performance results.

### 4.1 B$^{\text{link}}$-tree

B$^{\text{link}}$-trees [14, 21] are a concurrent enhancement of B$^{+}$-trees, an ordered data structure widely used in database and file systems. The main difference between a B$^{+}$-tree and a B$^{\text{link}}$-tree is the addition of two fields in each node: a *high key* representing the largest key among this node and its descendants, and a *right pointer* linking the node to its immediate right sibling. A node's high key is always smaller than any key of the right sibling or its descendants, allowing fast determination of a node's key range. The right pointer facilitates concurrent operations.

The original disk-based implementation of a B$^{\text{link}}$-tree uses the atomicity of file operations to avoid the need for locking. Srinivasan and Carey describe an in-memory version with a reader-writer lock in every node [23]. To perform a lookup, a reader descends from the root to a leaf node, then checks the node's high key to see if the desired key is in that node's key range. If not (in the case that the node has been split by another writer), the reader follows right pointers until an appropriate leaf is found. During this process, the reader holds only one reader lock at a time. When moving to the next node, it releases the previous node's lock *before* acquiring the new one. In an insert/remove operation, a writer acts like a reader to locate a correct leaf node, then releases that leaf's reader lock and acquires the same leaf's writer lock. Because a node split may occur during the lock switch, the writer starts another round of search for the proper leaf using writer locks.

A full node `A` is split in three steps. First, a new node `B` is allocated with its right pointer linking to `A`'s right sibling, and half the elements from `A` are moved to `B`. Second, `A`'s right pointer is redirected to `B`, and `A`'s writer lock is released. Third, `A`'s new high key is inserted into its parent node. For simplicity, we employ the remove algorithm of Lehman and Yao [14], which does not merge underflowed leaf nodes; this means there is no node deallocation in our code.

***Speculation:*** The B$^{\text{link}}$-tree algorithm already uses fine-grained locking. Its *lookup*, *insert* and *remove* operations contain two kinds of critical sections: (1) Critical sections protected by reader locks check a node's key range for a potential right move, or search for a key within a node. In MSpec code, these are all replaced with equivalent speculation. (2) Critical sections protected by writer locks perform actual insertion and removal. If a node is full, a split occurs in the critical section. The result of the MSpec transformation is that *lookup* becomes entirely speculative, and *insert* and *remove* start with a speculative search to check the presence/absence of the key to be inserted/removed. By performing searches in speculative mode, MSpec eliminates the need for reader-writer locks. Simpler and cheaper mutex locks suffice for updates, and *lookup* operations become nonblocking.

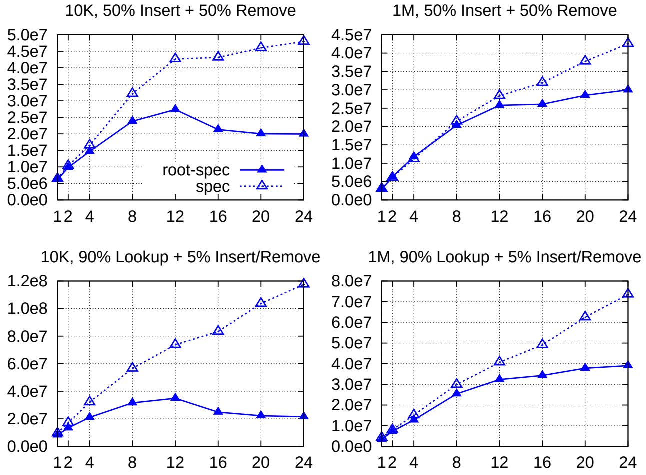


**Figure 5.** Throughput of B$^{\text{link}}$-tree methods on Intel Xeon E5649, for different tree sizes and method ratios.

***Validation:*** Validation in a B$^{\text{link}}$-tree is relatively easy. Every speculation works on a single node, to which we add a version number. If (type-preserving) node deallocation were added to *remove*, we would use one bit of the version number to indicate whether the corresponding node is in use. By setting the bit, deallocation would force any in-progress speculation to fail its validation and go back to the saved parent (note: *not* to the beginning of the method) to retry.

***Performance:*** Figure 5 compares the original and MSpec versions of B$^{\text{link}}$-tree on the Intel Xeon. Results on the Niagara 2 machine are qualitatively similar (Figure 6). The non-MSpec code uses a simple, fair reader-writer lock [17]. To avoid experimental bias, the MSpec code uses the same lock's writer side. Each node contains a maximum of 32 keys in both algorithms, and occupies about 4 cache lines. To avoid a performance bottleneck at the top of the tree, the original algorithm uses speculation at the root node (only).

We ran the code with two different sizes of trees and two different mixes of methods. Small trees (10K elements) are more cache friendly than larger trees (1M elements), but suffer higher contention because of fewer nodes. The 90% *lookup*, 5% *insert* and 5% *remove* method mix simulates read-dominant workloads, and 0%:50%:50% simulates write-dominant workloads. Before each throughput test, a warm-up phase inserts an appropriate number of elements into the tree, randomly selected from a double-sized range

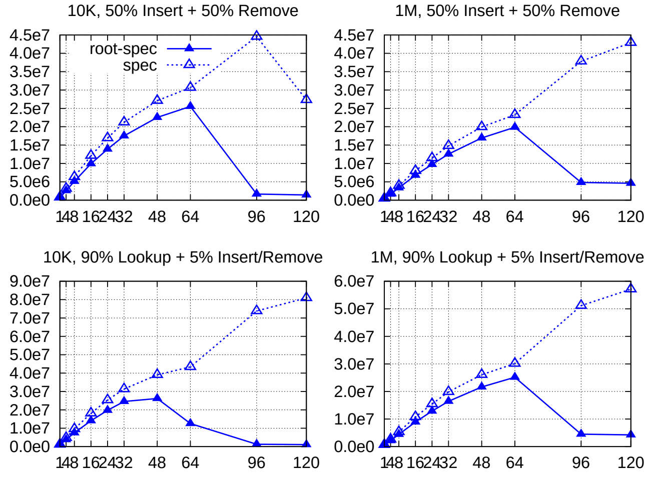
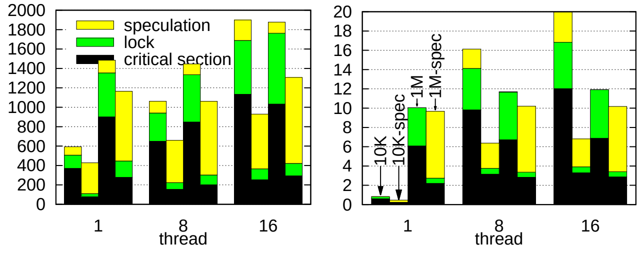


**Figure 6.** Throughput of B$^{\text{link}}$-tree methods on Niagara 2.

(e.g., $[0, 20000)$ for small trees). Keys used in the timing test are randomly selected from the same range.

MSpec provides both greater throughput and greater scalability in all experiments, even with speculation at the non-MSpec root node. MSpec scales very well even when the benchmark is running across chips ($>$12 threads on the Xeon; $>$64 on the Niagara 2). Comparing the left-hand and right-hand graphs in Figures 5 and 6, we can see that the advantage of MSpec increases with higher contention (smaller trees).

Figure 7 presents hardware profiling results to help understand the behavior of speculation. Clearly, the critical path is shorter in MSpec code. As a consequence, lock contention is significantly reduced.[1] In separate experiments (not shown) we began with originally-empty trees, and ran until they reached a given size. This, too, increased the advantage of MSpec, as the larger number of node splits led to disproportionately longer critical sections in the non-MSpec runs.



(a) CPU cycles/op (b) L2 cache misses/op

**Figure 7.** Hardware profiling of CPU cycles and L2 load misses per operation for "50% Insert/Remove" on Intel Xeon.

## 4.2 Cuckoo Hash Table

Cuckoo hashing [19] is an open-addressed hashing scheme that uses multiple hash functions to reduce the frequency of collisions. With two functions, each key has 2 hash values and thus 2 possible bucket locations. To insert a new element, we examine both possible slots. If both are already occupied, one of the prior elements is displaced and then relocated into its alternative slot. This process repeats until a free slot is found.

Concurrent cuckoo hashing was proposed by Herlihy and Shavit [9]. It splits the single table in two with each having its own hash function. In addition, each table becomes an array of *probe sets* instead of elements. A probe set is used to store elements with the same hash value. To guarantee constant time operation, the number of elements in a probe set is limited to a small constant CAPACITY. One variant of the data structure (a *striped cuckoo hash table*) uses a constant number of locks, and the number of buckets covered by a lock increases if the table is resized. In an alternative variant, (a *refinable cuckoo hash table*) the number of locks increases with resizing, so that each probe set retains an exclusive lock. The refinable variant avoids unnecessary serialization, but its code is more complex.

Since an element E may appear in either of two probe sets—call them A and B—an atomic operation in the concurrent cuckoo hash table has to hold two locks simultaneously. Specifically, when performing a *lookup* or *remove*, the locks for both A and B are acquired before entering the critical section. In the critical section of the *insert* method, if both A and B have already reached CAPACITY, then a resize operation must be done. Otherwise, E is inserted into one probe set. If that set contains more than THRESHOLD $<$ CAPACITY elements, then after the critical section, elements will be relocated to their alternative probe sets to keep the set's size below THRESHOLD.

***Speculation:*** As in the B$^{\text{link}}$-tree, MSpec makes *lookup* lock-free, and moves the presence/absence check at the beginning of *insert*/*remove* out of the critical section. In addition, *insert* executes the code that decides which probe set the new element should be added to in the speculative phase. If the element to remove is speculatively found in probe set A, *remove* needs to acquire only A's lock instead of both A's and B's.

***Validation:*** A version number is added to each probe set to enable validation. To minimize space overhead, we embed a probe set's size field (a few bits are sufficient) in its version number. In *lookup*, to validate the presence of an element in a set, we only need to compare the set's latest version number to the version seen during speculation. To validate the absence of an element, we compare two probe sets' latest version numbers with the versions seen during speculation. If either has changed, a retry is performed. In most cases, only one is changed, and we can skip the unchanged set

[1] We modified the OS kernel so that hardware performance counters could be read in user mode with low cost. Due to profiling overhead (20-30% in B$^{\text{link}}$-tree), the throughput of profiled code (Figure 7) does not exactly match the un-profiled case (Figure 5).

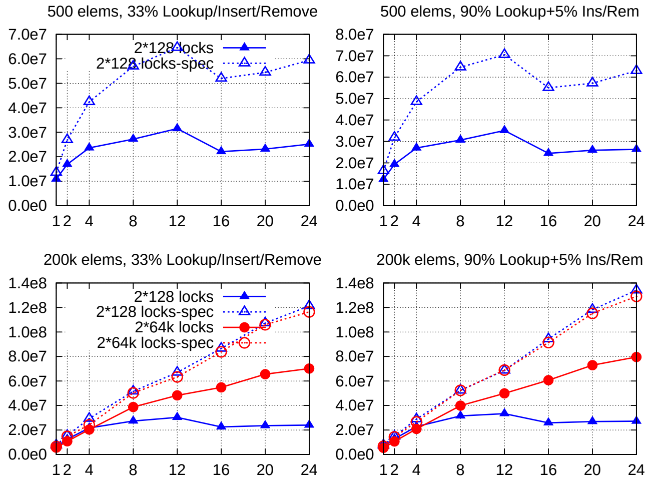


**Figure 8.** Throughput of cuckoo hash table on Intel Xeon E5649, for different data-set sizes and method ratios. The 2*128 curves use striped locking; the 2*64K curves are refinable.

in the next try. If both are unchanged, *lookup* returns false. Though the two sets may be checked at different times, their version numbers ensure that the two histories (in each of which the element is not in the corresponding set) overlap, so their exists a linearization point [13] in the overlapped region when the element was in neither set. To support concurrent *resize*, a resize version number is associated with the whole data structure. At any validation point, that number is also checked to detect a *resize* during speculation.

***Performance:*** Our experiments (Figures 8 and 9) employ a direct (by-hand) C++ translation of the Java code given by Herlihy and Shavit [9]. We use a CAPACITY of 8 and a THRESHOLD of 4; this means a probe set usually holds no more than 4 elements and a half full table's final size is #elements/4. We ran our tests with two different data set sizes: the smaller (∼500 elements) can fit completely in the shared on-chip cache of either machine; the larger (∼200K elements) is cache averse. For the striped version of the table, we use 128 locks. For the refinable version, the number of locks grows to 64K. As in the B$^{\text{link}}$-tree experiments, we warm up all tables before beginning timing.

For striped tables with 128 locks, MSpec is 10%-20% faster than the baseline with 64 threads on the Niagara 2, and more than 50% faster with 120 threads. The gap is significantly larger on the Xeon. Scalability in the baseline suffers from lock conflicts with increasing numbers of threads. MSpec overcomes this problem with fine-grain speculation, a shorter critical path, and fewer lock operations (in *lookup* and *remove*). For the same reason, MSpec is also useful for refinable tables in all configurations ("2*128 locks-spec" vs "2*128 locks" in the first row, "2*64K locks-spec" vs "2*64K locks" in the second row).

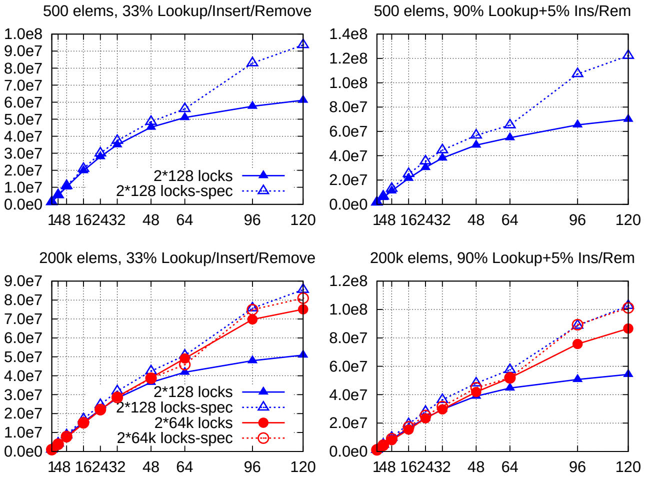
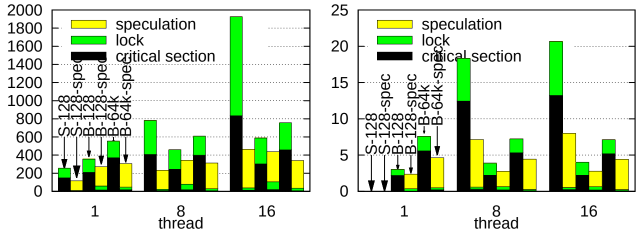


**Figure 9.** Throughput of cuckoo hash on Niagara 2.



(a) CPU cycles/op (b) L2 cache misses/op

**Figure 10.** Hardware profiling for "33% Lookup/Insert/Remove" cuckoo hash on Intel Xeon. In labels, prefix "S" means small table with 500 elements; "B" means big table with 200k elements.

For small data sets (upper graphs in Figures 8 and 9), refinable locking offers no advantage: there are only 128 buckets. For large data sets (lower graphs), non-MSpec refinable tables ("2*64K locks") outperform non-MSpec striped tables ("2*128 locks") as expected. Surprisingly, striped MSpec tables ("2*128 locks-spec") outperform both non-MSpec refinable tables and MSpec refinable tables ("2*64K locks-spec"), because the larger lock tables induce additional cache misses (compare the four "B-*" bars in Figure 10b).

This example clearly shows that fine-grained locking is not necessarily best. The extra time spent to design, implement and debug a fine-grained locking algorithm does not always yield the best performance. Sometimes, a simpler coarse-grained algorithm with speculation can be a better choice.

## 4.3 Bitmap Allocator

Bitmaps are widely used in memory management [24] and file systems. They are very space efficient: only one bit is required to indicate the use of a resource. A bitmap allocator may use a single flat bitmap or a hierarchical collection of bitmaps of different sizes. We consider the simplest

| Data Structure | Locks | Speculation | Validation | Safety Issues | Performance Wins | LOC (base:MSpec) |
|---|---|---|---|---|---|---|
| equivalence sets | CG, FG | key search, set iteration | VN, in-place | stale pointer | cache misses, atomic ops | |
| B$^{link}$-tree | FG | lookup, key range check, pointer chasing, node allocation | VN | stale pointer | cache misses, atomic ops, lock protocol | 775 : 809 |
| cuckoo hash | CG, FG | lookup | VN, in-place | stale pointer | cache misses, atomic ops, fewer locks | 530 : 636 |
| bitmap allocator | CG | bits ops | in-place | none | computation | 76 : 93 |

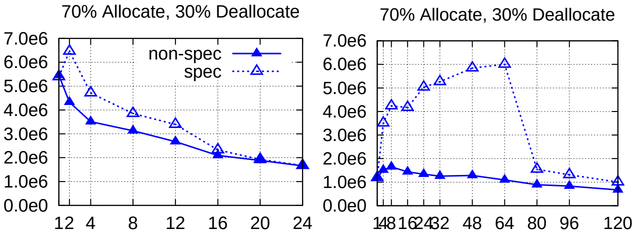
**Table 1.** A summary of the application of MSpec. CG/FG = coarse-/fine-grained, VN = version number, LOC = lines of code for concurrent methods.

case, where a next-fit algorithm is used to search for available slots. In the baseline algorithm, a global lock protects the entire structure. The data structure supports concurrent *allocate* and *deallocate* operations. To find the desired number of adjacent bits, *allocate* performs a linear search in its critical section, then sets all the bits to indicate they are used. *Deallocate* takes a starting position and size as parameters, and resets the corresponding bits in its critical section.

***Speculation:*** Most of execution time is spent searching the bitmap for free bits. This process can easily be moved to a speculative phase.

***Validation:*** After the lock is acquired, an in-place validation checks to make sure the bits found during speculation are still available. If any of them has been taken by a concurrent thread, we abort and perform a new speculative search starting from the current position. No safety issues arise, as the bitmap always exists.

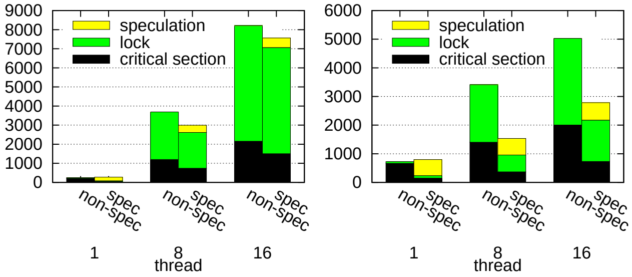
***Performance:*** Our experiments (Figure 11) employ an array of 256K bits with a time-limited scenario in which a 70% *allocate* / 30% *deallocate* mix consumes the bitmap gradually so it gets harder and harder to find a free slot. Allocation requests have the distribution 20% 1 bit, 30% 2 bits, 30% 4 bits, 15% 8 bits, and 5% 16 bits.



(a) Intel Xeon E5649 (b) Niagara 2

**Figure 11.** Throughput of linear bitmap allocator.

On Niagara 2, both implementations reach their peak throughput at low thread count. However, with all threads running on one chip, MSpec code's peak throughput is considerably higher as a result of a much shorter critical section in `Allocate`. By contrast, the benefit of MSpec is much more modest on the Intel Xeon. This is because the Xeon CPU can execute bit operations much faster than the simpler cores of the Niagara 2 machine, leaving less work available to be moved to speculation, as shown in Figure 12.



(a) Intel Xeon E5649 (b) Niagara 2

**Figure 12.** Hardware profiling for bitmap allocator CPU cycles distribution.

## 5. Summary

We have described the application of MSpec to four different concurrent data structures: equivalence sets, a B$^{link}$-tree, a cuckoo hash table, and a bitmap allocator. Each benefits from the MSpec design pattern. Table 1 summarizes these data structures, comparing the baseline locking policies, the work done in speculation, the validation methods, safety issues, the source of performance wins, and programming complexity as measured by lines of code. We see that MSpec does not increase the code complexity dramatically.

We encourage the designers of concurrent data structures to consider more aggressive use of manual speculation. In most cases, the principal design challenge will be to identify the minimal set of places in the code requiring memory fences and validation. Tools or programming disciplines to simplify this task are an appealing topic for future research.

## References

[1] RSTM: Reconfigurable Software Transactional Memory, v.7, 2011. http://code.google.com/p/rstm/.

[2] N. G. Bronson, J. Casper, H. Chafi, and K. Olukotun. A practical concurrent binary search tree. In *Proc. of the 15th ACM SIGPLAN Symp. on Principles and Practice of Parallel Programming*, PPoPP '10, pages 257–268, 2010.

[3] L. Dalessandro, M. L. Scott, and M. F. Spear. Transactions as the foundation of a memory consistency model. In *Proc. of the 24th Intl. Symp. on Distributed Computing*, DISC '10, pages 20–34, Sept. 2010.

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

[5] D. Dice, O. Shalev, and N. Shavit. Transactional locking II. In *Proc. of the 20th Intl. Symp. on Distributed Computing*, DISC '06, pages 194–208, Sept. 2006.

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

[7] P. Felber, V. Gramoli, and R. Guerraoui. Elastic transactions. In *Proceedings of the 23rd International Conference on Distributed Computing*, DISC'09, pages 93–107, 2009.

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

[9] M. Herlihy and N. Shavit. *The Art of Multiprocessor Programming*. Morgan Kaufmann Publishers Inc., 2008.

[10] M. Herlihy, V. Luchangco, and M. Moir. Obstruction-free synchronization: Double-ended queues as an example. In *Proc. of the 23rd Intl. Conf. on Distributed Computing Systems*, ICDCS '03, pages 522–529, 2003.

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

[12] M. Herlihy, N. Shavit, and M. Tzafrir. Hopscotch hashing. In *Proc. of the 22nd Intl. Symp. on Distributed Computing*, DISC '08, pages 350–364, 2008.

[13] M. P. Herlihy and J. M. Wing. Linearizability: A correctness condition for concurrent objects. *ACM Trans. Program. Lang. Syst.*, 12:463–492, July 1990.

[14] P. L. Lehman and S. B. Yao. Efficient locking for concurrent operations on B-trees. *ACM Trans. Database Syst.*, 6:650–670, Dec. 1981.

[15] V. J. Marathe, M. F. Spear, and M. L. Scott. Scalable techniques for transparent privatization in software transactional memory. In *Proc. of the Intl. Conf. on Parallel Processing*, ICPP '08, Sept. 2008.

[16] P. E. McKenney, J. Appavoo, A. Kleen, O. Krieger, R. Russel, D. Sarma, and M. Soni. Read-copy update. In *Proc. of the Ottawa Linux Symp.*, July 2001.

[17] J. M. Mellor-Crummey and M. L. Scott. Scalable reader-writer synchronization for shared-memory multiprocessors. In *Proc. of the 3rd ACM SIGPLAN Symp. on Principles and Practice of Parallel Programming*, PPOPP '91, pages 106–113, 1991.

[18] M. M. Michael and M. L. Scott. Simple, fast, and practical non-blocking and blocking concurrent queue algorithms. In *Proc. of the 15th ACM Symp. on Principles of Distributed Computing*, PODC '96, pages 267–275, May 1996.

[19] R. Pagh and F. F. Rodler. Cuckoo hashing. *J. Algorithms*, 51: 122–144, May 2004. ISSN 0196-6774.

[20] T. Riegel, C. Fetzer, and P. Felber. Time-based transactional memory with scalable time bases. In *Proc. of the 19th ACM Symp. on Parallel Algorithms and Architectures*, SPAA '07, pages 221–228, 2007.

[21] Y. Sagiv. Concurrent operations on b-trees with overtaking. In *Proc. of the 4th ACM SIGACT-SIGMOD Symp. on Principles of Database Systems*, PODS '85, pages 28–37, 1985.

[22] M. F. Spear, V. J. Marathe, W. N. Scherer III, and M. L. Scott. Conflict detection and validation strategies for software transactional memory. In *Proc. of the 20th Intl. Symp. on Distributed Computing*, DISC '06, pages 179–193, Sept. 2006.

[23] V. Srinivasan and M. J. Carey. Performance of B+ tree concurrency control algorithms. *The VLDB Journal*, 2:361–406, Oct. 1993.

[24] P. R. Wilson, M. S. Johnstone, M. Neely, and D. Boles. Dynamic storage allocation: A survey and critical review. In *Proc. of the Intl. Workshop on Memory Management*, IWMM '95, pages 1–116, 1995.