Compiler Aided Manual Speculation for High Performance 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.

Using several different concurrent data structures as examples, we show that manual addition of speculation to traditional lock-based code can lead to significant performance improvements. Successful speculation requires careful consideration of profitability, and of how and when to validate consistency. Unfortunately, it also requires substantial modifications to code structure and a deep understanding of the memory model. These latter requirements make it difficult to use in its purely manual form, even for expert programmers. To simplify the process, we present a compiler tool, CSpec, that automatically generates speculative code from baseline lock-based code with user annotations. Compiler-aided manual speculation keeps the original code structure for better readability and maintenance, while providing the flexibility to chose speculation and validation strategies. Experiments on UltraSPARC and x86 platforms demonstrate that with a small number annotations added to lock-based code, CSpec can generate speculative code that matches the performance of best-effort hand-written versions.

Categories and Subject Descriptors D.1.3 [*Programming Techniques*]: Concurrent Programming—Parallel Programming; D.3.4 [*Programming Languages*]: Processors—Code generation

General Terms Algorithm, Performance

Keywords Manual Speculation, Design Pattern, Concurrent Data Structure, Compiler Assistance

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 allow non-mutating methods to run in parallel. In important cases, RCU [17] 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 without compromising the correctness of other 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 as—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? Work by Bronson et al. [3] 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 attempts to regularize the notion of manual speculation (MSpec). Specifically, we characterize MSpec as a *design pattern* that transforms traditional, pessimistic code into optimistic, speculative code by addressing three key questions: (1) Which work should be moved out of a critical section and executed speculatively? (2) How does the remaining critical section validate that the speculative work was correct? (3) How do we avoid erroneous behavior when speculative code sees inconsistent data?

Using MSpec, we have constructed speculative versions of four example data structures: *equivalence sets*, *cuckoo hash tables*, B^{link}-*trees*, and a *linear bitmap allocator*. Our implementations outperform not only STM, but also pessimistic code with similar or finer

* This work was supported in part by National Science Foundation grants CCR-0963759, CCF-1116055, and CNS-1116109, and by an IBM Canada CAS Fellowship. Our Niagara 2 machine was provided by Oracle Corp.

PPoPP'13, February 23–27, 2013, Shenzhen.

granularity locking. The advantage typically comes both from reducing the overall number of atomic (read-modify-write) instructions and from moving instructions and cache misses out of critical sections (i.e., off the critical path) and into speculative computation. We note that unlike STM, MSpec does not require locking and validation to be performed at identical granularities.

In developing MSpec, our intent was to provide a useful tool for the construction (by experts) of concurrent data structures. We never expected it to be *easy* to use, but we expected the principal challenges to revolve around understanding the (data structure-specific) performance bottlenecks and speculation opportunities. As it turned out, some of the more tedious, mechanical aspects of MSpec were equally or more problematic. First, the partitioning of code into speculative and nonspeculative parts, and the mix of validation code and normal code in the speculative part, breaks natural control flows, making the source code difficult to read, understand, and debug. Second, since the original locking code must generally be preserved as a fallback when speculation fails, the programmer must maintain two versions, applying updates and bug fixes to both. Third, because naive speculation introduces data races, the programmer must deeply understand the memory model, and pepper the code with `atomic` annotations and/or memory fences.

To address these "mechanical" challenges, we have developed a set of program annotations and a source-to-source translator, CSpec, that generates speculative code automatically from annotated source. CSpec allows the programmer to continue to work on the (lightly annotated) original code. It eliminates the possibility of version drift between speculative and nonspeculative versions, and automates the insertion of fences to eliminate data races.

As a motivating example, we present our hand-constructed (MSpec) version of equivalence sets in Section 2, with performance results and implementation experience. We then turn to the CSpec language extensions and source-to-source translator in Section 3, and to guidelines for using these in Section 4. Additional case studies appear in Section 5, with performance results for both MSpec and CSpec versions. Our principal conclusion, discussed in Section 6, is that compiler-assisted manual speculation can be highly attractive option. While more difficult to use than STM or coarse-grain locking, it is substantially easier than either fully manual speculation or the creation of ad-hoc, data structure-specific nonblocking or fine-grain locking alternatives.

2. Motivating Example: Equivalence Sets

An instance of the data structure for equivalence sets 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 `Sum` method iterates over a specified set and returns some aggregate result. The `Move` method moves an integer from one set to another. We have included a `set` field in each element to support a constant-time `MemberOf` method (not shown).

2.1 Implementation Possibilities

A conventional lock-based implementation of equivalence sets is intuitive and straightforward: each method comprises a single critical section. Code with a single lock per set (lines 1–43 in Figure 1) provides a good balance between coding efficiency and performance. It is only slightly more complicated than a scheme in which all sets share a single global lock—the critical section in `Move` must acquire two locks (in canonical order, to avoid deadlock at line 25)—but scalability and throughput are significantly better. Given the use of doubly-linked lists, we do not see any hope for an efficient nonblocking implementation on machines with only single-word (CAS or LL/SC) atomic primitives.

STM can of course be used to create an optimistic version of the code, simply by replacing critical sections with transactions. The resulting performance (with the GCC default STM system—see Figures 2 and 3) is sometimes better than with a global lock, but not dramatically or uniformly so. Even a specialized STM with "elastic transactions" [7], which provide optimization for the search phase in `Move`, still lags behind per-set locking. STM incurs nontrivial overhead to initialize a transaction and to inspect and modify metadata on each shared memory access. Additional overheads stem from two major lost opportunities to exploit application semantics. First, to avoid proceeding on the basis on an inconsistent view of memory, STM systems typically *validate* that view on every shared memory read, even when the programmer may know that inconsistency is harmless. And while heuristics may reduce the cost of validation in many cases, the fallback typically takes time proportional to the number of locations read. Second, STM systems for unmanaged languages typically perform locking based on hash-based "slices" of memory [6], which almost never coincide with program-level objects. In our equivalence set example, the commit code for GCC STM must validate all `head` and `next` pointers read in the loop in `Sum`, and acquires 6 locks (one per pointer) for write back in `Move`. Hand-written code can do much better.

2.2 A Manual Speculative Implementation

Our MSpec version of `Sum` exploits the fact that the method is read only, that it traverses only one set, and that nodes encountered in that traversal will never have garbage `next` pointers, even if moved to another set. These observations allow us to write an obstruction-free [10] `MSpecSum` that validates using a single version number in each set. Because locking and validation are performed at the same granularity, we combine the lock and version number into a single *sequence lock* field (`lck.v`) [14]. If we iterate over the entire set without observing a change in this field, we know that we have seen a consistent snapshot. As in most STM systems, failed *validation* aborts the loop and starts over (line 67).

The baseline critical section in `Move` begins by finding an appropriate (sorted) position in the target list `s` (lines 27–32). The element `e` is then removed from its original list (lines 34–35) and inserted into `s` at the chosen position (lines 37–41). The position-finding part of `Move` is read only, so it can be performed speculatively in `MSpecMove`—that is, before entering the critical section. The validation at line 83 ensures that the `pnext` element remains in the same set and no new element has been inserted between `pprev` and `pnext` since line 81, so that it is correct to insert `e` before `pnext`. The other validation, at line 79, ensures that the while loop continues to traverse the same set, and will therefore terminate. A set's version number is increased both before and after a modification. The low bit functions as a lock: an odd version number indicates that an update is in process, preventing other threads from starting new speculations. Version number increments occur implicitly in lock acquisition and release.

Under the C++11 memory model, any class field that is read during speculation and written in a critical section (in our case, three fields of `Element`) must be declared as an `atomic` variable to avoid data races. For reads and writes to atomics inside the critical section, since locks already guarantee exclusive modification, we can specify relaxed memory order for best performance. For sequence-lock based speculative reads, depending on the hardware, different methods for tagging memory orders add different overhead [2]. Since our experiments employ total store order (TSO) machines like the x86 and SPARC, in which loads are not reordered by the processor, atomic loads incur no extra costs under the sequential consistency memory order. So we simply skip tagging speculative reads and let them use `memory_order_seq_cst` by default.

```
1  struct Element {
2    int key;
3    Element *next, *prev;
4    Set *set;
5  };
6  struct Set {
7    Element head;
8    Lock lck;
9  };

11 int ESets::Sum(Set *s) {
12   int sum = 0;
13   s->lck.lock();
14   Element *pnext = s->head->next;
15   while (pnext != s->head) {
16     sum += pnext->key;
17     pnext = pnext->next;
18   }
19   s->lck.unlock();
20   return sum;
21 }

23 void ESets::Move(Element *e, Set *s) {
24   Set *oset = e->set;
25   grab_unordered_locks(oset->lck, s->lck);
26   // find e's next element in s
27   Element *pprev = s->head;
28   Element *pnext = pprev->next;
29   while (pnext->key < e->key) {
30     pprev = pnext;
31     pnext = pprev->next;
32   }
33   // remove e from its original set
34   e->prev->next = e->next;
35   e->next->prev = e->prev;
36   // insert e before pnext
37   e->next = pnext;
38   e->prev = pprev;
39   e->set = s;
40   pprev->next = e;
41   pnext->prev = e;
42   release_locks(oset->lck, s->lck);
43 }

45 // manual speculative implementation
46 // for TSO machines
47 struct Element {
48   int key;
49   atomic<Element*> next, prev;
50   atomic<Set*> set;
51 };
52 struct Set {
53   Element head;
54   SeqLock lck;
55 };

57 int ESets::MSpecSum(Set *s) {
58  again:
59   int sum = 0;
60   int v = s->lck.v;
61   if (v & 1) goto again;
62   Element *pnext = s->head->next;
63   while (pnext!=s->head && v == s->lck.v) {
64     sum += pnext->value;
65     pnext = pnext->next;
66   }
67   if (v != s->lck.v) goto again;
68   return sum;
69 }

71 void ESets::MSpecMove(Element *e, Set *s) {
72   Set *oset = e->set;
73  again:
74   Element *pprev = s->head;
75   Element *pnext = pprev->next;
76   while (pnext->value < e->value) {
77     pprev = next;
78     pnext = pprev->next;
79     if (pnext->set != s)
80       goto again;
81   }
82   grab_unordered_locks(oset->lck, s->lck);
83   if (AL(pnext->set, MO_relaxed) != s || AL(
         pnext->prev, MO_relaxed) != pprev) {
84     release_locks(oset->lck, s->lck);
85     goto again;
86   }
87   AS(AL(e->prev, MO_relaxed)->next, AL(e->
         next, MO_relaxed), MO_relaxed);
88   AS(AL(e->next, MO_relaxed)->prev, AL(e->
         prev, MO_relaxed), MO_relaxed);
89   AS(e->next, pnext, MO_relaxed);
90   AS(e->prev, pprev, MO_relaxed);
91   AS(e->set, s, MO_relaxed);
92   AS(pprev->next, e, MO_relaxed);
93   AS(pnext->prev, e, MO_relaxed);
94   release_locks(oset->lck, s->lck);
95 }
```

Figure 1. Per-set lock implementation of concurrent equivalence sets. The `key` field of each set's head is initialized to $+\infty$ to avoid loop bound checks in lines 16. For code simplicity, it's assumed that for any element, there is at most one thread calling `Move` on it at any time. `AL=atomic_load, AS=atomic_store, MO_*=memory_order_*`.

2.3 Performance Results

We tested our code on an Oracle (Sun) Niagara 2 and an Intel Xeon E5649. The Niagara 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 Xeon 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` 4.7.1 (`-O3`).

To measure throughput, we arrange for worker threads to repeatedly call randomly chosen methods for a fixed period of time. 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. Our mutex locks, where not otherwise specified, use test-and-`test_and_set` with exponential back-off, tuned individually for the two machines.

2.3.1 Test Configurations

We compare six different implementations of equivalence sets. **FGL** and **SpecFGL** are the versions of Figure 1, with set-granularity locking. (Code for the `Move` operation includes an extra check to make sure that `e` is still in the same set after line 25.) **CGL** and **SpecCGL** are analogous versions that use a single global lock. **Gnu-STM** uses the default STM system that ships with GCC 4.7.1; **ε-STM** employs elastic transactions [7], which are optimized for search structures. Loads and stores of shared locations were hand-annotated in ε-STM.

2.3.2 Performance and Scalability

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

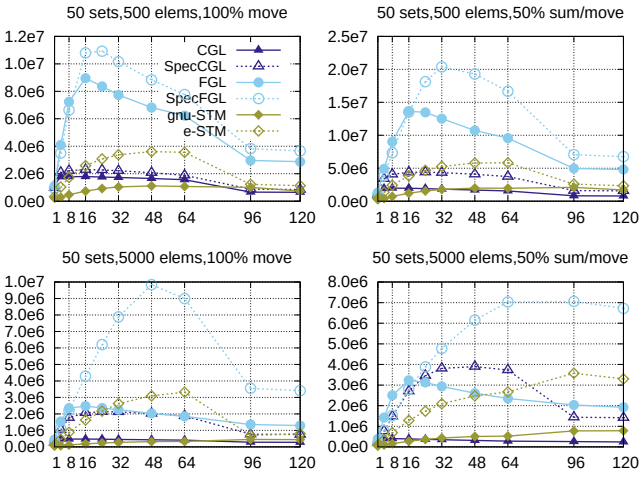


Figure 2. Throughput of concurrent equivalence sets on Niagara 2. The X axis is the number of concurrent threads; the Y axis is method invocations per second.

Figure 2 illustrates scalability on Niagara 2. As expected, FGL outperforms CGL in all tests, and SpecFGL outperforms SpecCGL. 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.

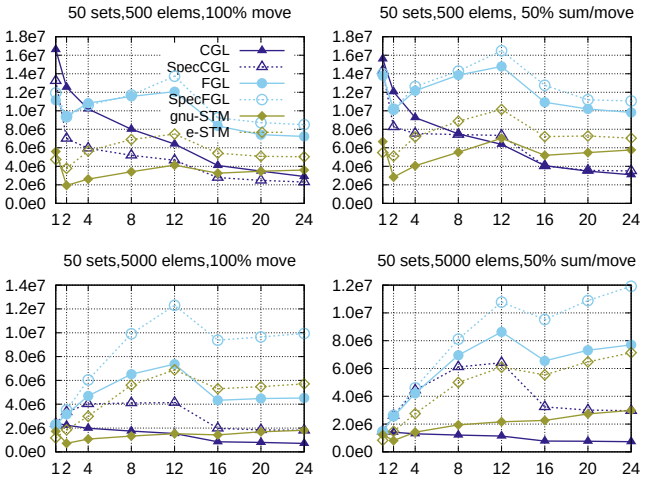


Figure 3. Throughput (invocations/s) of concurrent equivalence sets on Intel Xeon E5649.

Manual speculation improves scalability and throughput for both CGL and FGL. In the 5000-element, 50/50 sum/move case, SpecCGL even outperforms FGL, out to 80 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 SpecFGL, measured by comparing to FGL on a single thread, is less than 10%. Therefore, even without contention, SpecFGL can deliver competitive performance. By contrast, both gnu-STM and ε-STM have significant baseline overhead (2–4$\times$ slower than CGL on a single thread), and can outperform only CGL.

Results on the Xeon machine resemble those on Niagara 2, with the interesting exception that single-thread performance is significantly higher in the 500-element case, where the data set can almost fit in the 32KB L1 data cache.

2.4 Limitations of Manual Speculation

The above results show that hand-written speculation can limit the cost of validation and yield significant performance and scalability improvements not only over STM, but also over lock-based alternatives. Other examples, summaries of which appear in Section 5, confirm that these results are not limited to a single data structure. The presence of general principles suggests, in fact, that manual speculation be thought of as a first-class *design pattern* for concurrent data structures. At the same time, our experience suggests that this pattern is not very easy to use.

Some of the challenges are expected, and are discussed in more detail in Section 4. To extract concurrency, one must understand the workload characteristics and concurrency bottlenecks that suggest an opportunity for speculation. To control the cost of validation, one must identify any data-structure-specific indications of consistency. To maintain correctness, one must insert validation before any operation that might produce anomalous results in the wake of mutually inconsistent reads.

Several challenges, however, are more "mechanical," and suggest the need for automation. First, the changed code layout and the mix of validation code and normal code break natural control flows, making the source code difficult to read, understand and debug. Second, if the original locking code co-exists with its speculative version (either as a backup when speculation fails or as an alternative when speculation is unprofitable), the programmer will have to maintain two separate versions of the code, and make sure that they track each other in the face of updates and bug fixes. Third, manual speculation requires a deep understanding of the underlying memory model. Speculative loads, almost by definition, constitute data races with stores in the critical sections of other threads. In C++ 11, where data races are illegal, the programmer must identify and annotate all places where naive speculation would otherwise introduce a race. As discussed by Boehm [2], even the choice of annotation is nontrivial, and potentially architecture-dependent. In the equivalence sets example, the default sequentially consistent ordering is zero-cost for TSO machines, but has unacceptable overhead on PowerPC machines.

3. Compiler Aided Manual Speculation

We propose to overcome the difficulties in applying manual speculation with the aid of compiler. The key idea is to automatically generate speculative code from an annotated version of the lock-based code, allowing the programmer to focus on higher-level issues of what to do speculatively, and how and when to validate. This semi-automatic approach to speculation, which we call CSpec (*compiler aided manual speculation*), consists of (1) a simple but flexible set of annotations to specify the speculation method, and (2) a source-to-source compiler that transforms the annotated source into an explicitly speculative version.

3.1 Interface

The language interface is designed to be as simple as possible. It comprises the following directives:

- **#pragma spec**: tells the compiler to generate a speculative version for the following critical section. This directive should appear immediately before a lock acquisition statement, which must be the single entry point of its critical section.
- **#pragma spec consume_lock(lock0, lock1, ...)**: instructs the compiler to "consume" one or more locks (these should be a subset of the critical section's lock set) at a particular place in the code. Here, "consume" means that the lock is truly needed (to protect writes or to achieve a linearization point), and that speculation on data protected by the lock must end. Statements that cannot be reached from any `consume_lock` form the *speculative part* of a critical section. Multiple `consume_lock` directives are allowed in a critical section. It is also legal to consume a same lock more than once on a given code path: the compiler is responsible for acquiring each lock at the first `consume_lock` that names it.
- **#pragma spec set_checkpoint(id)**: marks a checkpoint with a specific integer id (id$>$ 0). A checkpoint is a place to which execution can roll back after a failed validation. During rollback, all local variables modified beyond the checkpoint will be reset to their snapshots at the checkpoint. The default checkpoint (id=0) is located at the critical section's entry point.
- **#pragma spec validate_ver(ver0, ver1, ...[, cp_id])**: does a version number-based validation. If any version number has changed since checkpoint `cp_id`, roll back to that checkpoint.
- **#pragma spec validate_val(val0, val1, ...[, cp_id])**: is similar to `validate_ver`, but does value-based validation.
- **#pragma spec validate_cond(cond_expr[, cp_id])**: evaluates the expression `cond_expr` and rolls back to checkpoint `cp_id` if it is false.
- **#pragma spec waive_rollback(cp_id, var0, var1, ...)**: waives value rollback for specified variables at checkpoint `cp_id`.

Figure 4 presents a re-implementation of the equivalence set data structure using CSpec annotations. This new version is almost identical to the original lock-based code, except for seven embed-

```
1  int ESets::Sum(Set *s) {
2      int sum = 0;
3      #pragma spec
4      s→lck.lock();
5      Element *pnext = s→head→next;
6      while (pnext != s→head) {
7          sum += pnext→key;
8          pnext = pnext→next;
9          #pragma spec validate_ver(s→lck.v)
10     }
11     #pragma spec validate_ver(s→lck.v)
12     s→lck.unlock();
13     return sum;
14 }

16 void ESets::Move(Element *e, Set *s) {
17     Set *oset = e→set;
18     #pragma spec
19     grab_unordered_locks(oset→lck, s→lck);
20     Element *pprev = s→head;
21     Element *pnext = pprev→next;
22     while (pnext→key < e→key) {
23         pprev = pnext;
24         pnext = pprev→next;
25         #pragma spec validate_cond(pnext→set==s)
26     }
27     #pragma spec consume_lock(oset→lck, s→lck)
28     #pragma spec validate_cond(pnext→set==s && pnext→prev==pprev)
29     e→prev→next = e→next;
30     ..... // same as L35−41 of Figure 1
31     release_locks(oset→lck, s→lck);
32 }
```

Figure 4. Concurrent equivalence sets using CSpec.

```
1  int ESets::Sum(Set *s) {
2      ..... // same as L12−20 of Figure 1
3  }

5  int ESets::SpecSum(Set *s) {
6      int sum = 0;
7      int SPEC_tmp_var_0 = sum;
8  SPEC_label_1:
9      sum = SPEC_tmp_var_0;
10     int SPEC_ver_0 = spec::wait_ver(s→lck.v);
11     Element *pnext = AL(s→head→next, MO_seq_cst);
12     while (pnext != s→head) {
13         sum += pnext→key;
14         pnext = AL(pnext→next, MO_seq_cst);
15         if (AL(s→lck.v, MO_seq_cst) != SPEC_ver_0)
16             goto SPEC_label_1;
17     }
18     if (AL(s→lck.v, MO_seq_cst) != SPEC_ver_0)
19         goto SPEC_label_1;
20     return sum;
21 }
```

Figure 5. Automatically generated code for the `Sum` method. Memory ordering is optimized for TSO machines.

ded directives. Compiler output for the `Sum` operation appears in Figure 5. It includes both the original locking version and a speculative version. The speculative version includes appropriate tags on all atomic accesses, optimized for the target machine.

Compared with the pure manual speculation code in Figure 1, the CSpec implementation has the following advantages: (1) The language interface is concise. Usually, only a small number of directives is needed to describe the speculation mechanism; often, the original control flow can be retained, with no code motion required. The resulting code remains clear and readable. (2) Code maintenance is straightforward: there is no need to manage separate speculative and nonspeculative versions of the source. Any updates will propagate to both versions in the compiler's output. (3) The programmer is no longer exposed to low-level details of the memory model, as the compiler will handle them properly.

3.2 Implementation

Our compiler support is implemented as a `clang` [1] front-end plugin. The plugin takes user-annotated source code as input and does source-to-source transformation to create a speculative version. A nonspeculative version is also generated, simply by discarding all embedded directives. Alternative implementations, such as IR-based transformation or static and run-time hybrid support would also be possible; we leave these for future exploration.

We define a lock's *live range* as the set of all statements that are reachable from its acquisition statement but not reachable from its release statement. A *speculative code region* is defined as the union of all live ranges of the locks that appear in a critical section's *entry point*—the lock acquisition call immediately following **#pragma spec**.

Lockset Inference. A key part of the CSpec transformation algorithm is to identify the lock set that each `consume_lock` should actually acquire. The intuition behind the analysis is straightforward: we want every critical section, at run time, to acquire locks exactly once (this marks the end of speculation), and to acquire all locks named in any `consume_lock` directive that may subsequently be encountered on any path to the end of the critical section. Conservatively, we could always acquire the entire lockset at the first dynamically encountered `consume_lock` directive. To the extent possible with static analysis, we wish to reduce this set when possible, so that locks that are never consumed are also never acquired. Pseudocode of the algorithm we use to accomplish this goal appears in Algorithm 1.

Algorithm 1: Translating `consume_lock` directives

```
   Input: a speculative code region R
1  // GetLiveRange(L, S): get L's live range assuming S is the
   acquisition statement;
2  C ← all consume_lock directives in R;
3  foreach consume_lock cl in C do
4      foreach Lock l in cl.LockArgList do
5          cl.LiveRange ← cl.LiveRange ∪ GetLiveRange(l, cl);
6          foreach Statement s in GetLiveRange(l, cl) do
7              s.LockSet ← s.LockSet ∪ cl.LockArgList;
8  foreach consume_lock cl in C do
9      foreach Statement s in cl.LiveRange do
10         cl.LockSet ← cl.LockSet ∪ s.LockSet;
11     if cl is unreachable from any other consume_lock in C then
12         replace cl with LockStmt(R, cl.LockSet);
13     else if cl is not dominated by another consume_lock in C then
14         remove cl;
15         foreach BasicBlock b in GetBasicBlock(cl).Parents do
16             if a path from R.entry to cl goes through b and the
               path does not contain any other consume_lock then
17                 emit LockStmt(R, cl.LockSet) at the end of b;
18     else //cl is dominated by other consume_lock
19         remove cl;
```

The algorithm works in two phases. The first phase (lines 3–7) calculates the `LockSet` of each statement in code region `R`. These are the locks that must be acquired before the statement executes at run time. The second phase first infers the lock set for each `consume_lock` (lines 9–10) so that the set will contain all locks required by the `consume_lock`'s live range. Then for each `consume_lock`, the algorithm decides how to handle it according to its reachability (lines 11–19). The generated lock acquisition calls use the same function name as the original call (the entry point of `R`) and the lock parameters appear in the same order as in the original call's parameter list. Thus, the output code is deadlock free if the original code was.

As an alternative to static lockset inference, we could allow locks to be acquired more than once at run time, and release each the appropriate number of times at the end of the critical section. This strategy, however, would be incompatible with source code using non-reentrant locks.

Checkpoint. A checkpoint serves as a possible rollback position. There are three kinds of variables a checkpoint may snapshot: (1) Live-in local variables. For a local variable that is declared before a checkpoint and may be modified after the point, our source transformation tool creates a local mirror for that variable and copies its value back when speculation fails (see `sum` in Figure 5). Nonlocal variables are assumed to be shared; if they need to be restored, the programmer will have to do it manually. (2) Version numbers. The tool finds `validate_ver`s that may roll back to this checkpoint and, for each, inserts a `wait_ver` call to wait until the number is unblocked (line 10 in Figure 5). The values read (e.g., `SPEC_ver_0`) are kept for validation. (3) Validation values. `Validate_val`s are handled in the same way as `validate_ver`s, except that no `wait_ver` is inserted.

Tagging Atomic Variables. After code transformation, our tool detects the class fields that are both written in write-back mode (after consuming a lock) and accessed in speculative mode, and re-declares them as atomic variables. Accesses to these variables are grouped into three categories: (1) accesses in write-back mode; (2) reads in validation; (3) other reads in a speculative phase. The tool selects an appropriate (target machine-specific) memory ordering scheme (including a fence if necessary) for each access category.

Switching Between Locking and Speculative Versions. The generation of two versions of a data structure opens the possibility of dynamically switching between them for the best performance. Since speculation doesn't win in all cases, it might sometimes be better to revert to the original locking code. The switch could be driven by user-specified conditions such as workload size or current number of threads. Switch conditions might also be automatically produced by the compiler based on profiling results, or on run-time information like the abort rate of speculation. These possibilities are all subjects for future work. In the experiments reported here, we use only the speculative version of the code.

3.3 Limitations

Our source-to-source tool currently faces three limitations, all of which suggest additional subjects for future work. First, the critical section must have a single entry point. In source programs with multiple entry points, it may be possible to merge these entry points by manually rearranging code structure. Second, we do not support nested speculation due to the complexity of nested rollbacks; inner directives will simply be discarded. Third, the analysis of locksets is static, and can be defeated by complicated data flow and re-assignment of lock variables. We argue that these are uncommon in concurrent data structures, and could be addressed through manual source code changes.

4. Principles of Coding with CSpec

Sections 2 and 3 presented an example of manual speculation and a compiler-based tool to assist in the process. In the current section we generalize on the example, focusing on three key questions that the user of CSpec must consider.

4.1 Where do we place `consume_lock` directives?

This question amounts to "what should be done in speculation?" because `consume_lock`s mark the division between the speculative and nonspeculative parts of the original critical section. Generally, `consume_lock(L)` is placed right before the first statement that modifies the shared data protected by `L` on a given code path. Sometimes a modification to shared data `A` can only be performed under the guarantee that shared data `B` won't be changed; in this case `B`'s lock should also be consumed. Since the speculative parts come from the original critical section, it may be profitable to rearrange the code to delay the occurrence of `consume_lock`s. 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.

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 after `consume_lock`.

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, so there's no need to place any `consume_lock` in it. *Insert* and *remove* typically start with a search to see whether the desired key is present; we can make this speculative as well, by inserting `consume_lock`s before the actual insertion/deletion. In resource managers, an *allocate* method typically searches for free resources in a shared pool before actually performing allocation. 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, speculation 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 of 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 5.

4.2 How do we validate?

Validation is the most challenging and flexible part of CSpec programming. 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 [23] or per-location timestamps [6, 21] 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* [16].) 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 [5], but for safety in the general case, a very large number of validations may still be required.

Using the three `validate_*` directives provided by compiler-aided manual speculation, 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 [3, 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. The `validation_ver` directive serves this purpose in CSpec.

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 SpecCGL code of Section 2, 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—even if it involved an inconsistent view of memory. Mechanisms like "early release" in STM systems exploit this observation [11]. In manual speculation, we can choose to validate simply by checking the local context. An example appears at line 28 of Figure 4, where `pnext→set` and `pnext→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. In CSpec, it is realized as value-based validation (`validate_val`) or condition-based validation (`validate_cond`).

4.3 What can go wrong and how do we handle it?

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 [4]. 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 CSpec, 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. 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 [19].

5. Additional Case Studies

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

5.1 Cuckoo Hash Table

Cuckoo hashing [20] is an open-addressed hashing scheme that uses multiple hash functions to reduce the frequency of collisions. With two functions, each key has two hash values and thus two 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 lock protocol is much 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: Speculation makes *lookup* obstruction-free. We place `consume_locks` only before a modification to a probe set. This moves the presence/absence check in *insert*/*remove* out of the critical section. We illustrate the CSpec implementation of *remove* in Figure 6. If the element to remove is speculatively found in probe set `A`, *remove* needs to consume only `A`'s lock instead of both `A`'s and `B`'s (case ①).

Validation: A version number is added to each probe set to enable `validate_ver`. In *remove* (and similarly in *lookup* and *insert*), to validate the presence of an element in a set (cases ①, ②), we only need to validate that set's version (lines 11, 16) after its lock is consumed. We don't do any validation in the `search` method, because linear search in a limited-sized ($<$ `CAPACITY`) probe set will terminate in `CAPACITY` steps regardless of any change in set elements. To validate the absence of an element (case ③), both probe sets' versions should be checked (line 20). 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 there 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 the checkpoint, that number is validated (line 7) to detect a *resize* which breaks the mapping between `key` and probe sets (line 5).

Performance: Our experiments (Figures 7 and 8) 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

```
1   bool CuckooHashMap::Remove(const KeyT& key) {
2       std::pair<Lock*, Lock*> lcks = map_locks(key)
3       #pragma spec
4       lock(lcks.first, lcks.second);
5       Bucket &setA = bucket0(key), &setB = bucket1(key)
6       #pragma spec set_check_point(1)
7       #pragma spec validate_ver(this→ver)
8       int idx = search(setA, key);
9       if (idx>=0) { // ① key is in setA
10          #pragma spec consume_lock(lcks.first)
11          #pragma spec validate_ver(setA.ver, 1)
12          setA.remove(idx);
13          unlock(lcks.first, lcks.second);
14      } else if ((idx = search(setB, key)) >=0) { // ② key is in setB
15          #pragma spec consume_lock(lcks.first, lcks.second)
16          #pragma spec validate_ver(setB.ver, 1)
17          setB.remove(idx);
18          unlock(lcks.first, lcks.second);
19      } else { // ③ key is not found
20          #pragma spec validate_ver(setA.ver, setB.ver, 1)
21          #pragma spec validate_ver(this→ver)
22          unlock(lcks.first, lcks.second);
23      }
24      return idx>=0;
25  }
```

Figure 6. CSpec tagged `Remove` method of cuckoo hash table.

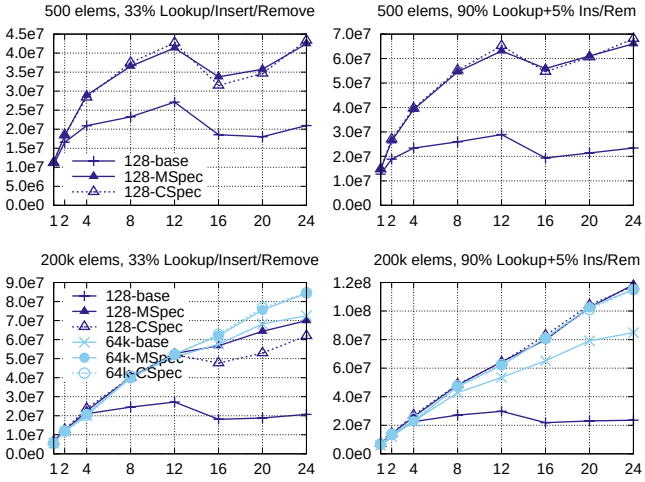


Figure 7. Throughput of cuckoo hash table on Intel Xeon E5649, for different data-set sizes and method ratios. The 128-* curves use striped locking; the 26k-* curves are refinable. *-MSpec are pure manual speculation implementations.

means a probe set usually holds no more than 4 elements. 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. Before each throughput test, a warm-up phase inserts an appropriate number of elements into the table, randomly selected from a double-sized range (e.g., $[0, 1000)$ for small tables). Keys used in the timing test are randomly selected from the same range.

For striped tables with 128 locks, CSpec code is 10%–20% faster than the baseline with 64 threads on the Niagara 2, and even better with 120 threads. The gap is significantly larger on the Xeon. Scalability in the baseline suffers from lock conflicts with increasing numbers of threads. CSpec overcomes this problem with fine-grain speculation, a shorter critical path, and fewer lock operations. For the same reason, CSpec is also useful for refinable tables in all configurations ("*-CSpec" vs "*-base"). For small data

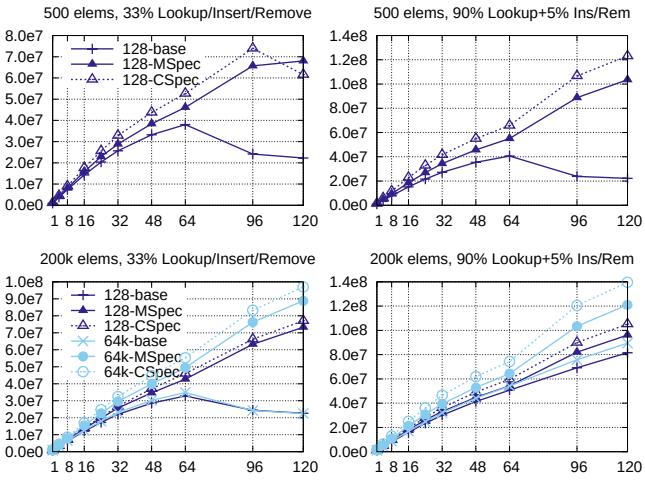


Figure 8. Throughput of cuckoo hash table on Niagara 2.

sets (upper graphs in Figures 7 and 8), refinable locking offers no advantage: there are only 128 buckets. For large data sets (lower graphs), baseline refinable tables ("64k-base") outperform baseline striped tables ("128-base") as expected. Surprisingly, CSpec striped tables ("128-CSpec") outperform baseline refinable tables and are comparable to CSpec refinable tables ("64k-CSpec"), because the larger lock tables induce additional cache misses.

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 superior performance. Sometimes, a simpler coarse-grained algorithm with CSpec can be a better choice.

A best-effort manual speculation version (MSpec) is also included in our experiments. For easy rollback, a critical section in MSpec is divided as a monotonic speculative phase and a non-speculative phase by reorganizing the code. Also, when a speculation fails, only one probe set has been changed in most cases, and we can skip the unchanged set in the next try. MSpec code, which adds 15% more lines to the baseline code, is considerably more complex than CSpec code. However, CSpec's performance matches MSpec's on Xeon ("*-CSpec" vs "*-MSpec"), and is even faster than the latter on Niagara 2, due to its simpler control flow and fewer instructions.

5.2 B^{link}-tree

B^{link}-trees [15, 22] 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^{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^{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 [24]. 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

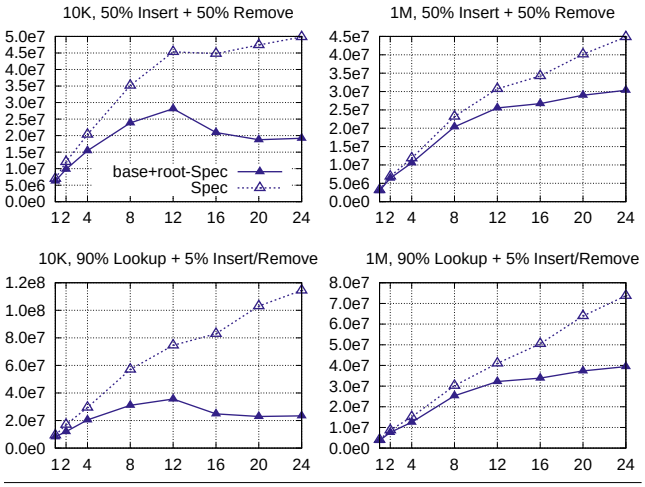
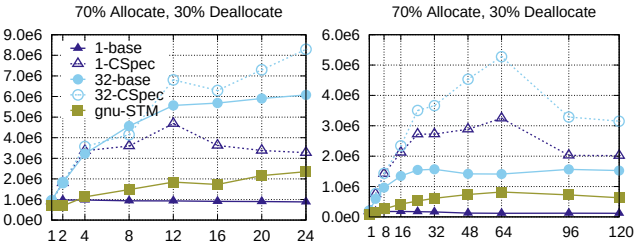


Figure 9. Throughput of B$^{\text{link}}$-tree methods on Intel Xeon E5649, for different tree sizes and method ratios.



(a) Intel Xeon E5649 (b) Niagara 2

Figure 10. Throughput of linear bitmap allocator. The lock array uses a single lock in 1-* curves, 32 locks in 32-* curves.

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 [15], 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. Not all of them can be transformed using CSpec, because the movement from one node to its right sibling or its child needs to hold two reader locks. CSpec cannot translate the overlapped critical section formed by the two locks. So, we manually convert the move step to 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. CSpec is able to handle these critical sections. The result of the 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, speculation eliminates the need for reader-writer locks. Simpler and cheaper mutex locks suffice for updates, and *lookup* operations become nonblocking.

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 (*not* the beginning of the method) to retry.

Performance: Figure 9 compares the original and CSpec versions of B$^{\text{link}}$-tree on the Xeon machine. Results on the Niagara 2 are qualitatively similar. The locking code uses a simple, fair reader-writer lock [18]. To avoid experimental bias, the CSpec 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. The 90% *lookup*, 5% *insert* and 5% *remove* method mix simulates read-dominant workloads, and 0%:50%:50% simulates write-dominant workloads. As in the cuckoo hash experiments, we warm up all trees before beginning timing.

CSpec provides both greater throughput and greater scalability in all experiments, even with speculation at the root node in baseline locking code. CSpec scales very well even when running across chips (>12 threads). Comparing the left-hand and right-hand graphs in Figure 9, we can see that the advantage of speculation increases with higher contention (smaller trees). 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 CSpec, as the larger number of node splits led to disproportionately longer critical sections in the baseline runs.

5.3 Bitmap Allocator

Bitmaps are widely used in memory management [25] 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 case, where a next-fit algorithm is used to search for available slots. In the baseline algorithm, an array of locks protects the entire structure (one lock protects one segment of the bitmap). 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. Since only one lock in the lock array is held at a time, we simply place a `consume_lock` directive before flipping the free bits.

Validation: After the lock is consumed, a `validate_cond` checks that the bits found during speculation are still available. No safety issues arise, as the bitmap always exists.

Performance: Our experiments (Figure 10) 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. Two lock array sizes, 1 (global lock) and 32, are used.

Data Structure	Locks	CSpec	Directives	Other changes
equivalence sets	CG, FG	2	5	No
cuckoo hash	CG, FG	3	23	No
B^{link}-tree	FG	3	17	hand-written speculative functions for moving between two nodes
bitmap allocator	CG	1	3	No

Table 1. A summary of the application of CSpec. CG/FG = coarse-/fine-grained.

On Niagara 2, CSpec-generated code ("*-CSpec") is significantly faster than the baseline as a result of a much shorter critical section in `Allocate`. Again, we see that coarse-grained locking with speculation ("1-CSpec") beats nonspeculative finer-grained locking ("32-base"). However, the benefit of CSpec is more modest on the Xeon machine. This is because the Xeon CPU can execute bit operations much faster than the simpler cores of the Niagara 2, leaving less work available to be moved to speculation. We also test a manual speculative version (not shown). Surprisingly, it is slightly slower than the CSpec version because its reorganized code contains more branch instructions. On both machines, the STM implementation is only faster than the single-lock baseline.

6. Conclusions

While fully automatic speculation, as provided by transactional memory, has the advantage of simplicity, we believe that manual speculation has a valuable role to play, particularly in the construction of concurrent data structure libraries. In support of this contention, we have presented four different structures—equivalence sets, a B^{link}-tree, a cuckoo hash table, and a bitmap allocator—in which speculation yields significant performance improvements. The principal challenges in their construction, as suggested in Section 4, were to identify the work that could profitably be moved to speculation, and to determine how and when to validate. To eliminate other, more mechanical challenges, we developed a set of compiler directives and a translation algorithm that partitions critical sections; identifies covering lock sets; and automates checkpointing, rollback, and the access tagging required to avoid data races and preserve sequential consistency.

Table 1 summarizes our four example data structures, comparing the baseline locking policies, the number of CSpec regions, the total number of added directives (excluding **#pragma spec**), and other changes to the source code. It clearly supports the claim that speculation can easily be added to existing lock-based code, with a small number of CSpec directives and few or no additional adjustments.

Our work suggests several avenues for future research, including a richer set of annotations, more sophisticated translation mechanisms, nested speculation, and dynamic selection among implementations with differing amounts of speculation.

References

[1] Clang: A C language family frontend for LLVM. http://clang.llvm.org.

[2] H.-J. Boehm. Can seqlocks get along with programming language memory models? *2012 ACM SIGPLAN Workshop on Memory Systems Performance and Correctness*, 2012.

[3] N. G. Bronson, J. Casper, H. Chafi, and K. Olukotun. A practical concurrent binary search tree. *15th ACM SIGPLAN Symp. on Principles and Practice of Parallel Programming*, Jan. 2010.

[4] L. Dalessandro and M. L. Scott. Sandboxing transactional memory. *21st Intl. Conf. on Parallel Architectures and Compilation Techniques*, Sep. 2012.

[5] L. Dalessandro, M. L. Scott, and M. F. Spear. Transactions as the foundation of a memory consistency model. *24th Intl. Symp. on Distributed Computing*, Sep. 2010.

[6] D. Dice, O. Shalev, and N. Shavit. Transactional locking II. *20th Intl. Symp. on Distributed Computing*, Sep. 2006.

[7] P. Felber, V. Gramoli, and R. Guerraoui. Elastic transactions. *23rd Intl. Conf. on Distributed Computing*, Sep. 2009.

[8] R. Guerraoui and M. Kapałka. On the correctness of transactional memory. *13th ACM SIGPLAN Symp. on Principles and Practice of Parallel Programming*, Feb. 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. *23rd Intl. Conf. on Distributed Computing Systems*, May 2003.

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

[12] M. Herlihy, N. Shavit, and M. Tzafrir. Hopscotch hashing. *22nd Intl. Symp. on Distributed Computing*, Sep. 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] C. Lameter. Effective synchronization on Linux/NUMA systems. *Gelato Federation Meeting*, San Jose, CA, May 2005.

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

[16] V. J. Marathe, M. F. Spear, and M. L. Scott. Scalable techniques for transparent privatization in software transactional memory. *Intl. Conf. on Parallel Processing*, Sep. 2008.

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

[18] J. M. Mellor-Crummey and M. L. Scott. Scalable reader-writer synchronization for shared-memory multiprocessors. *3rd ACM SIGPLAN Symp. on Principles and Practice of Parallel Programming*, Apr. 1991.

[19] M. M. Michael and M. L. Scott. Simple, fast, and practical non-blocking and blocking concurrent queue algorithms. *15th ACM Symp. on Principles of Distributed Computing*, May 1996.

[20] R. Pagh and F. F. Rodler. Cuckoo hashing. *J. Algorithms*, 51:122–144, May 2004.

[21] T. Riegel, C. Fetzer, and P. Felber. Time-based transactional memory with scalable time bases. *19th ACM Symp. on Parallel Algorithms and Architectures*, June 2007.

[22] Y. Sagiv. Concurrent operations on B-trees with overtaking. *4th ACM SIGACT-SIGMOD Symp. on Principles of Database Systems*, Mar. 1985.

[23] M. F. Spear, V. J. Marathe, W. N. Scherer III, and M. L. Scott. Conflict detection and validation strategies for software transactional memory. *20th Intl. Symp. on Distributed Computing*, Sep. 2006.

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

[25] P. R. Wilson, M. S. Johnstone, M. Neely, and D. Boles. Dynamic storage allocation: A survey and critical review. *Intl. Workshop on Memory Management*, Sep. 1995.