Lock-free internal binary search trees with memory management

Shane Valentine Howley

October 2012

Declaration

This thesis has not been submitted as an exercise for a degree at this or any other university. It is entirely the candidate's own work. The candidate agrees that the Library may lend or copy the thesis upon request. This permission covers only single copies made for study purposes, subject to normal conditions of acknowledgement.

Shane Valentine Howley

Abstract

The current design trend in microprocessor systems is to add more CPU cores thus increasing parallel execution resources. Although this allows many sequential programs to be run in parallel, it can be challenging to use multiple cores to speed up a single program. Concurrent algorithms have traditionally used mutual exclusion to synchronise access to data structures shared between executing threads. This limits access to the data structure to a single thread at any one time. The pitfalls of using mutual exclusion are well known, deadlock, livelock, convoying and priority inversion to name a few. Mutual exclusion inherently limits parallelism. Non-blocking algorithms are optimistic and can enable much greater levels of parallelism by allowing all threads access to shared data concurrently, resolving contention when detected.

This thesis presents a new non-blocking internal binary search tree algorithm. Previous binary search tree algorithms either rely on mutual exclusion or require modifications to the tree structure that result in suboptimal memory use. The new algorithm does not suffer from these limitations and closely mirrors the standard implementation of a sequential binary tree. Furthermore, the new algorithm is based on standard atomic instructions which comprise single-word reads, writes and compare-and-swap. The new algorithm is also informally proven to be lock-free and linearisable.

Extensive experimental measurement and analysis demonstrates that the new internal binary search tree algorithm has a higher throughput than previously published concurrent dynamic search structure algorithms for a range of workload patterns and data structure sizes. Tests were initially performed without memory management and then the effect of applying hazard pointer memory management was analysed and shown to impose minimal overhead. The memory footprint, per key, is shown to be significantly smaller than that of the other algorithms. Finally, a new technique for avoiding the ABA problem in descriptor-based concurrent algorithms is described and used to ensure all retired objects can be reclaimed.

Acknowledgements

I would first like to thank my supervisor, Dr. Jeremy Jones, without whose encouragement this thesis would not have come together at the end. Your advice and guidance has kept me on track over the last few years and you have always acted as a great sounding board for new ideas.

I would like to thank my parents, Patricia and Cyril, for their unrelenting support and for instilling me with the determination (and at times, stubbornness!) to see this through.

Thanks to Warren for sharing in the Ph.D. experience with many technical (and not so technical) conversations over coffee, these were an essential part of the day in the final few weeks.

Last, but not least, I would like to thank Di for bearing with me through the postgraduate work (which I believe I had initially estimated taking 3 years). Living with a Ph.D. student can't have been easy at times, but your love and support helped me push through to the end.

Shane Howley

Dublin, October 2012.

Publications Related to this Thesis

- Shane V. Howley and Jeremy Jones. A non-blocking internal binary search tree. In *Proceedings of the 24th ACM Symposium on Parallelism in Algorithms and Architecture*, SPAA '12, pages 161–171, Pittsburgh, PA, USA, June 2012. ACM.

Contents

Chapter 1

Introduction

This thesis is concerned with concurrent algorithms for shared memory multiprocessors. Such systems have multiple CPUs which communicate indirectly by reading and writing to shared memory locations in a unified address space. A sequence of instructions (or computer program) executed on a CPU is known as a *thread* of execution or a *process*. A typical system will have many more threads running than CPUs available so an operating system requires a scheduler component that decides which threads are run, for how long and on which CPU. These decisions are usually made based on fairness (all waiting threads should get a chance to run) and priority (some more important threads should run before others). A scheduling decision to stop a thread running in order to allow another thread use of the CPU is known *preemption*. This makes threads naturally asynchronous because they could be scheduled to run at any time as well as preempted at any time when running.

1.1 Current design trends

While CPU designers have been able to fit ever more transistors on a single chip with every new generation of manufacturing processes, increases in single-threaded throughput of CPUs has slowed significantly. The performance gained by raising clock speed alone can no longer be justified because of the associated increase in power consumption. CPU designers have also created more complex chips in order to extract more performance per clock by adding and enhancing features such as pipelining, branch prediction, caches and out of order instruction execution. Performance gains in these

areas have slowed in recent years due to diminishing returns from further optimisations.

Hardware architects have instead focussed on raising the number of processing cores per CPU package and the number of thread contexts per core while considering power efficiency. This has not had a huge impact on general purpose computing performance outside of some niche applications as traditional software is sequential. By adding more CPUs, the effect is that more sequential programs can be run in parallel. Concurrent algorithms that can effectively use these extra processing resources are required, therefore, in order to achieve higher performance from computer applications. Concurrent algorithms are an active research area, but their composition is not well understood by programmers who have learned to program writing sequential code. Herlihy and Shavit have attempted to bring concurrent programming principles into the mainstream with their well-received book *The Art of Multiprocessor Programming* [Herlihy and Shavit 2008].

1.2 Mutual exclusion

Traditionally, synchronising access to a shared resource is performed via mutual exclusion where a single thread has exclusive access to the resource. This type of synchronisation clearly limits parallelism, however, as concurrent threads requiring access to the resource must wait until it becomes available. Mutual exclusion is a type of *blocking* synchronisation – other threads are blocked from progressing if the resource is unavailable. Programs that are sharing data between threads will often contain blocks of code where access must be serialised so that updates cannot be processed concurrently and other threads do not read data that is undergoing updates. Access to these *critical sections* can be controlled via mutual exclusion.

Mutual exclusion is commonly enforced using a lock variable in shared memory. For instance, if the lock is free, a thread can acquire the lock by writing 1 to the memory location and can later release the lock by writing 0. If the lock is not free, it is possible that another thread is blocked from doing any useful work and will repeatedly check the status of the lock until it becomes free which is known as *spinning*. The assumption is that under normal circumstances the lock will not be held for very long.

Optimisations can be added to mutex locks in order to extract better parallelism by having information about the environment in which they will be used. A readers-writer lock (RW lock) allows concurrent access to multiple reader threads but sole access

to any writer thread. The writer thread must wait until all readers have released the RW lock before it can acquire it and process its updates to the shared data. A common problem with RW locks is writer starvation, which can happen if at least one reader thread continuously holds the lock which can be the case if there are many overlapping readers. A solution to this would be to not allow any new readers to acquire the lock while a writer is waiting, but this will cause reader threads to block until the current readers finish and then the writer releases the lock.

There are many well-known problems that can occur when dealing with mutual exclusion. *Deadlock* can occur when two or more threads must acquire more than one lock before they can proceed. One lock is acquired by one thread and another lock is acquired by a different thread but both then attempt to wait for the other lock they require. Both threads are waiting for the other to finish and release their lock and so neither progresses. One possible solution to deadlock is to have a waiting thread take some action in order to recover from the deadlock by perhaps releasing their held lock and retrying after some period of time. Unfortunately, this can give rise to another negative property called *livelock*. If the two or more threads involved require resources locked by another thread and attempt take some action to recover, it is possible that these threads become caught in a sequence of deadlocking and then backing off from one another. In this case, any livelocked threads never make progress. Lock *convoying* is another possible problem when using mutual exclusion and is caused by thread scheduling. If a thread holding a lock is preempted by the scheduler and other threads of equal priority require the lock, a queue of threads forms all waiting on the same lock to become free before the thread holding the lock is scheduled again. This is closely related to *priority inversion* which is caused when a lower priority thread acquires a lock and is preempted by a higher priority thread that requires the lock before it can proceed. The high priority thread yields in order to allow the low priority thread to complete its critical section and release the lock but the operating system schedules some medium priority thread instead. In this case, the waiting thread has had its priority inverted because it cannot progress until the low priority thread runs again.

1.3 Non-blocking synchronisation

Non-blocking synchronisation seeks to eliminate these concurrency problems. As the name suggests, the actions of any one thread do not block the entire system from making any progress. Threads are not required to wait before attempting to make

progress which has the effect of reducing latency, and most parts of the algorithm are designed to be executed concurrently. Allowing many threads concurrent access to a data structure also means that algorithms can achieve large speedups relative to versions protecting the data structure via mutual exclusion.

Unfortunately, non-blocking algorithms have their own set of associated problems. The complexity of their design means that even minor improvements to existing algorithms warrant publication as an academic paper accompanied with a detailed correctness proof. This makes it difficult for programmers to implement or modify these algorithms as subtle changes can affect their correctness. Adding complexity to a sequential algorithm in order to make it concurrent will usually mean adding a performance overhead for single threaded or lightly contended executions limiting the cases where the non-blocking algorithm is useful. This means that some lock-based algorithms that simplify access via mutual exclusion will perform better than non-blocking algorithms employing complex synchronisation in order to allow concurrent access. Some universal techniques exist that allow almost mechanical creation of non-blocking implementations of data structures but these tend to perform relatively poorly compared with concurrent algorithms tailored to the specific data structure. Non-trivial memory management issues also arise when access to shared memory is not exclusive. Memory use cannot follow the usual allocation and deallocation pattern of sequential programs because it is often difficult to determine a safe time for memory to be deallocated. Shared memory which is concurrently accessible can be removed from the visibility of any new thread accessing the shared data, but threads executing while the shared data was still visible could still attempt to access it.

1.4 Binary search trees

Trees are one of the fundamental types of data structure in computer science, particularly suited to storing hierarchical data and allowing efficient searching, and have applications in many areas. Binary search trees represent the simplest form of the tree type of data structure by allowing a maximum of two children per node and they have a straightforward ordering layout. Binary search trees are the basis for many more specialised types of tree suited to particular tasks, and while they are not commonly used directly, any augmentations to the basic structure can yield improvements in the derived data structures.

1.5 Contribution

This thesis makes the following contributions to the existing body of knowledge:

- Two non-blocking algorithms for internal binary search trees. The first uses a simplified construction in order to demonstrate the correctness condition. The second algorithm is a refinement of the first in terms of performance and resource use. Updates to nodes in the trees are serialised via a single field per node and this also enables a technique for detecting concurrent changes upstream in the tree structure that could affect the validity of searches. Empirical analysis of the refined design shows it outperforms previously published concurrent dynamic search structure algorithms while maintaining a significantly smaller memory footprint.

- A technique is described for avoiding the ABA problem in descriptor-based concurrent algorithms by generating unique null pointer values to reside in descriptor pointers when they are between periods of use. This improves on previous techniques which can block the most recently used descriptor from being reclaimed, thus consuming memory.

- A detailed measurement and analysis of the performance and memory use of the best known concurrent algorithms for dynamic sets. Each algorithm is tested using a range of common workload patterns and structure sizes. The performance of the algorithms that support hazard pointer memory reclamation is also investigated in order to show the effect an active memory reclamation technique has on the concurrent throughput.

1.6 Outline of the thesis

The rest of the thesis is outlined as follows:

- Chapter 2 describes the background, theory and related work in the field of non-blocking data structures. The architectural features of microprocessor systems that are leveraged in order to provide synchronisation between multiple threads are outlined along with some of the problems that must be dealt with. The following section then looks at properties that have been defined in order to make the behaviour of concurrent algorithms easier to reason about. The chapter concludes

by examining the previously published works on concurrent data structures, algorithmic techniques and memory management.

- Chapter 3 starts by describing the implementation issues that arise when enabling concurrent access to a binary search tree. Two algorithms for concurrent internal binary search trees are described, one building on and optimising the design of the other. A solution is then proposed for solving the ABA problem in descriptor-based concurrent algorithms, which is then used to ensure all retired objects in the new algorithms can be reclaimed.

- In chapter 4, a detailed analysis is performed on a number of concurrent dynamic data structures with a focus on throughput and memory use. The effect of integrating a memory management system to algorithms that can support it is also investigated.

- Chapter 5 concludes the thesis and suggests some future work.

Chapter 2

Background

This chapter discusses the hardware and software mechanisms that are used to synchronise threads in a multiprocessor system – and their associated problems. Some desirable properties for concurrent algorithms are then defined and outlined. Finally, a thorough review is performed covering: universal constructions for converting sequential data structures to non-blocking forms, the techniques that have been used previously to implement specific non-blocking data structures, and memory reclamation schemes for shared data.

2.1 Architecture

2.1.1 CPU cache subsystem

A modern CPU design will include two or more levels of cache to reduce average memory access times by providing an order of magnitude faster access to recently loaded memory locations. Taking a CPU using the Intel Nehalem architecture as an example, each CPU core has its own 64KB L1 and 256KB L2 cache and all cores on a single chip share an 8MB L3 cache. These cache levels form a hierarchy from fastest to slowest, with the most recently accessed memory locations in the L1 cache. When main memory is accessed, a fixed size contiguous block of memory (known as a cache line) must be read into the cache. This is the unit with which cache states are associated.

A cache coherency protocol ensures that the data stored in any of the caches is coherent with the data stored in the corresponding locations in main memory. When

considering memory locations that are unshared between CPU cores, the basic operations of the cache coherency protocol will be similar to, or a variation of, the following: When a memory location is read, each cache level in succession is checked to see if it contains that memory location, otherwise it is read into the cache from main memory and the L1 cache entry will be set in an exclusive or unshared state. Successive reads by the same CPU core can be serviced directly from its cache. If this memory location is written to by the same CPU core, the write will only update the cache line and it will be set in a modified or dirty state. When the cache line is evicted from the cache, its updated data must be written back to main memory.

When a CPU core attempts to access a memory location contained in the cache of another core, the coherency protocol must ensure that the most up to date version of the data is read. The core with exclusive access to the cache line will share it with the requesting core and both cache entries will be in the shared state. Both cores would now have low latency access to that memory location if it is read again. If either core was to write to that memory location while the cache line was a shared state, it would need to invalidate the other's cache line and set its own to be in a modified state. If the other CPU core were to then read the memory location again, the dirty cache line must be shared via the lowest common cache level between the cores (and possibly written back to main memory).

The cache coherency protocol is enforced even across physical CPU sockets in a multi-socket system. In these types of system, messages are passed across the system bus in order to ensure that all caches in the system remain coherent.

These cache interactions can have a significant impact on the performance of algorithms that involve sharing memory locations between multiple CPU cores. Programs in which multiple cores are writing to memory locations which cause invalidation of each other's cache lines suffer a penalty when rereading those memory locations compared to if it were just a single core performing the accesses. This inherently limits the ability for the program to scale as more execution resources are added, compared to running on a single core.

It is even possible for this cycle of cache line invalidations to occur when only private variables are being accessed. *False sharing* will happen when distinct memory locations which share the same cache line are being updated by different CPUs. The example in Figure 2.1 illustrates this if, for example, a reader thread executes concurrently with a writer thread and both have thread IDs that would make their thread

```
1  struct threadLocalStorage
2  {
3    volatile int locals[NUM_THREADS];
4  };
5
6  threadLocalStorage tls;
7
8  int readerThreadFunction(int threadID)
9  {
10   int x = 0;
11   for (int i = 0; i < 1000; i++)
12   {
13     x = x + tls.locals[threadID];
14   }
15   return x;
16 }
17
18 void writerThreadFunction(int threadID)
19 {
20   for (int i = 0; i < 1000; i++)
21   {
22     tls.locals[threadID] = i;
23   }
24 }
```

Figure 2.1: Demonstration of false sharing. `tls.locals[0...`$k-1$`]` is assumed to be stored in the same cache line where k is the number of ints in a single cache line. A concurrent reader and writer thread with thread IDs in the range 0 to $k-1$ will falsely share data even though they both operate on private memory locations.

local storage variables be in the same cache line. The reader will need to read its thread local storage variable at each iteration of the `for` loop, but the cache line containing it will be repeatedly invalided by the writer thread writing to its own thread local storage variable. This example is quite contrived in order to demonstrate a point, but a more realistic case where this is encountered would be with naturally disjointed objects such as dynamically allocated nodes in a linked list.

2.1.2 Read-modify-write

When a memory location is to be updated by multiple writers, it is often useful for a writer thread to know if no other writers have updated the location since it had last been read by that thread. Consider the pseudocode example of a naive attempt at implementing mutual exclusion in Figure 2.2: in the case that more than one CPU reaches line 29 while the lock is free, each CPU will write to the lock variable and think itself the owner of the lock. An atomic read-modify-write (RMW) operation can be

```
25 while (lock != 0)
26 {
27   // wait...
28 }
29 lock = 1;

31 // process critial section

33 lock = 0;
```

Figure 2.2: Critial section protected by a naive (and incorrect) mutual exclusion lock

```
34 bool CAS(Value* address, Value expected, Value swap)
35 {
36   atomic
37   {
38     if (*address == expected)
39     {
40       *address = swap;
41       return true;
42     }
43     else
44     {
45       return false;
46     }
47   }
48 }
```

Figure 2.3: Pseudocode operation of Compare-and-Swap

used to alleviate this issue by enabling a CPU to both check the state of the lock and, if available, obtain the lock in a single atomic step.

Modern microprocessor support for atomic RMW operations are divided into two types of implementations for historical reasons. Complex Instruction Set Computer (CISC) architectures such as Intel x86 or Motorola 68000 are generally able to perform both a load and store as part of a single instruction[1]. The most powerful instruction for solving synchronisation problems (as defined in terms of Herlihy's consensus numbers [Herlihy 1991]) is Compare-And-Swap (CAS)[2]. A CAS operation (Figure 2.3) does the following in a single atomic step: loads a value from a memory location, compares it to an expected value and conditionally stores a new value back to the same memory location. The operation then returns a boolean to indicate whether the values were successfully swapped or not.

[1] Although, the operations are also supported by the non-CISC SPARC architecture.

[2] Other, more specialised instructions like Fetch-and-Add, Swap, and Test-and-Set are generally also available but these can all be implemented using CAS.

```
49 Value LoadLinked(Value* address)
50 {
51   return *address;
52 }

54 bool StoreConditional(Value* address, Value store)
55 {
56   if (address has not been written to since matching LoadLinked)
57   {
58     *address = store;
59     return true;
60   }
61   else
62   {
63     return false;
64   }
65 }
```

Figure 2.4: Pseudocode operation of Load-Linked and Store-Conditional

The Reduced Instruction Set Computer (RISC) design philosophy normally restricts memory access to a single load or store per instruction meaning an ideal atomic RMW is not possible. The same effect can be achieved, however, using the instruction pair Load-Link and Store-Conditional (LL/SC) which is supported, for example by the Alpha, ARM, MIPS and PowerPC architectures. For ideal LL/SC semantics (Figure 2.4), Load-Link is used to read the value from a memory location after which arbitrary transformations may be applied to the value and a Store-Conditional can then be executed which will store the modified value in the same memory location if, and only if, no other writes were performed on that memory location since the Load-Link. Unfortunately, none of the architectures mentioned support the ideal semantics and are restricted usually by Store-Conditional failing in the case of the following: other loads to the same cache line containing the Load-Linked address (because the cache line state would be updated), nested LL/SC pairs (because only a single location can be Load-Linked at a time), context switches (because the thread which executed the Load-Link is descheduled). This restriction limits the usefulness of LL/SC so is used generally to simulate CAS (Figure 2.5).

Most 32-bit architectures have the ability to atomically load and store 64-bit values which means they can also support a double-width compare-and-swap (DWCAS). The advantage of DWCAS is that it allows the updating of two adjacent pointer-sized values atomically. Unfortunately, this is less common in 64-bit architectures, where it is currently only supported by Intel x86-64 via its `CMPXCHG16B` instruction. This means that algorithms which make use of and require DWCAS are not considered portable –

```
66 bool CAS(Value* address, Value expected, Value swap)
67 {
68   do
69   {
70     Value seen = LoadLinked(address);
71     if (seen != expected)
72     {
73       return false;
74     }
75   } while (!StoreConditional(address, swap));
77   return true;
78 }
```

Figure 2.5: Pseudocode implementation of Compare-and-Swap using Load-Linked and Store-Conditional

although it is possible to emulate in software with a performance cost.

2.1.3 Memory ordering

In order to optimise memory accesses, most CPU architectures will not process loads and stores in the order they are received so as to decrease stalls in the pipeline and better utilise the cache and memory subsystems. This has no logical effect on single-threaded programs because all memory accesses will have the same result/effect as they would have had if executed in their absolute order but the implications for concurrent executions are greater. For example, a store used to release a lock could be reordered before a store inside a critical section leaving an inconsistency inside the data structure protected by the lock if another CPU concurrently acquired the lock and read from the data structure. Architectures that perform reordering have explicit instructions, called memory barriers, that can be used to enforce ordering before proceeding, along with certain instructions that contain implicit memory barriers, such as the x86 CAS instruction. For simplicity and clarity, the source code examples in this thesis assume a sequentially-consistent memory order as different architectures might require different numbers of memory barriers.

2.1.4 The ABA problem

The ABA problem is a commonly encountered problem in concurrent programs. It arises when a thread reads a shared memory location as having the value, A, then another thread updates this memory location to some other value, B before changing

```
79 Node* top;

81 void push(Node* n)
82 {
83   do
84   {
85     Node* oldTop = top;
86     n->next = oldTop;
87   } while (!CAS(&top, oldTop, n));
88 }

90 Node* pop()
91 {
92   do
93   {
94     Node* oldTop = top;
95     if (oldTop == NULL) return NULL;
96     Node* newTop = oldTop->next;
97   } while (!CAS(&top, oldTop, newTop));

99   return oldTop;
100 }
```

Figure 2.6: ABA prone implementation of a lock-free stack using CAS.

to back to A again. If the first thread then reads the memory location again and observes the value A, it could incorrectly infer that the memory location had not changed between the two reads. This is what gives rise to the name *ABA*. CAS is particularly susceptible to the ABA problem when it compares its expected value with the memory location.

The problematic side to this is that assumptions based on this value having not changed may no longer be true. The ABA problem was first reported in reference to the CAS instruction on the IBM System/370 [IBM 1983]. When CAS is used to implement a naive list-based stack, the `pop()` function suffers from the ABA problem (Figure 2.6). Take the example that the initial stack contains at least two objects: A at the top and B the next object in the stack. A thread performing a `pop()` operation is preempted before the CAS at line 97 occurs. Before it is resumed, another thread pops both A and B off the stack, and then pushes A back on. When the first thread continues executing, the CAS expecting to see A on top of the stack will succeed and incorrectly make B the new top of the stack. This is incorrect and can cause further problems if B's memory has been reclaimed and/or reused for some other purpose.

The report that first identified the ABA problem also proposed a simple solution. Each ABA prone field which is to be modified with CAS must also have an adjacent version tag field. The tag value is incremented atomically with any changes to the ABA

prone field using DWCAS. If the ABA sequence occurs and a DWCAS is attempted which depends on the memory location having not changed, the modified tag value will cause it to fail and the program can proceed to handle that. In theory, LL/SC should avoid the ABA problem in these situations because SC will fail if any change is made to the memory location read by LL but because LL/SC is mostly used to implement CAS (again, due to the restricted implementations of LL/SC), it will encounter the same instances of the ABA problem.

2.2 Desirable properties

This section discusses some desirable properties of concurrent algorithms which make it easier to reason about the performance characteristics, correctness and behaviour of concurrent operations.

2.2.1 Non-blocking property

A non-blocking property defines a level of progress guarantee for algorithms in the face of thread stalls and concurrency conflicts. It is at odds with mutual exclusion which generally means that no progress is made or useful work is done by threads waiting for a lock to become free.

2.2.1.1 Obstruction-freedom

Obstruction freedom is the weakest of the three levels of non-blocking properties and the most recently defined [Herlihy et al. 2003a]. It guarantees that in the case there are n threads operating on a shared object, if any *n-1* threads are suspended the remaining thread will complete its operation in a bounded number of steps. Obstruction freedom is considered the simpler non-blocking property to adhere to and it can be reasoned that the overhead would be minimal when the chance of conflicts is low. It is often implemented by allowing threads to abort and roll back operations which were started by other threads in an attempt to complete their own operations. While this level of non-blocking behaviour eliminates deadlock, it is still susceptible to livelock because it allows for the situation where concurrently executing threads make no progress toward completing their operations.

2.2.1.2 Lock-freedom

Lock-freedom[3] is a superset of obstruction-freedom, offering the stronger guarantee that for any finite number of threads concurrently operating on a shared object, at least one thread will make overall system progress. In order to achieve this, the usual approach is to ensure information is made available to all threads so that they may cooperate in the completion of each others operations. Even in the event that all threads have conflicted and are all cooperating in the processing of the same operation, the guaranteed completion of this operation means system throughput will be achieved. Careful consideration must be made to the design of these 'helping' algorithms because help can potentially be given at any time after the attempted operation has been made visible, even after it has run its course. The design must ensure that this can have no unintended effect on the concurrent object, such as successful CAS operations on ABA prone fields. While progress is guaranteed for at least one thread (meaning livelock is eliminated), individual threads can still suffer from starvation if, for example, they are endlessly helping other operations.

2.2.1.3 Wait-freedom

Wait-freedom is the strongest non-blocking property, guaranteeing that every thread concurrently executing on a shared object will complete its operation in a finite number of steps. Wait-free algorithms are generally considered more difficult to write than other non-blocking equivalents and often carry a significant overhead in space requirements and/or sequential execution throughput. Because of this, wait-free algorithms tend to find more use in real-time systems where their guaranteed response times are required. The time guarantees of this type of non-blocking property mean that it is immune to thread starvation.

2.2.2 Linearisability

Linearisability is a correctness condition for concurrent objects, proposed by Herlihy and Wing [1990]. An operation is defined as being linearisable if it appears to take effect instantaneously at a point between its invocation and response. This can be viewed

[3] In the literature, the terms 'lock-free' and 'non-blocking' are sometimes used synonymously or with their meanings swapped due to a historical lack of formal definition. Furthermore, 'lock-free' is sometimes informally used to mean without explicit locking.

as similar to performing updates on a data structure which is protected by mutual exclusion lock – other threads attempting to access the data structure must wait for the lock to become free and can never observe it in a mid-update state. By ensuring linearisation points exist for the concurrent operations which can affect a shared object, it is possible to mirror the behaviour of the sequential definition of the object which makes it easier to reason about. For any possible concurrent execution, the absolute ordering of the linearisation points can be viewed as a sequential history which matches the expected outcome of serially executing the operations. This vastly simplifies the interactions that can occur between concurrent threads and makes algorithms more intuitive from a practical point of view.

2.2.3 Disjoint-access parallelism

Disjoint-access parallelism means that two threads can operate on separate sections of the same data structure without interfering with each other and do not have dependencies on any common set of objects. This means that disjoint-access parallel algorithms allow for greater levels of parallelism through reduced contention on shared memory locations. A simple example of disjoint-access parallelism is two threads operating at either end of a concurrent linked list – each can operate in isolation of the other provided the list is not empty.

2.3 Related work

2.3.1 Universal techniques

Initially, a series of universal techniques are introduced which are designed to make it easier to design non-blocking data structures. These techniques are designed such that they can be applied in a wide variety of circumstances, generally to convert existing sequential data structures to a concurrent non-blocking form.

2.3.1.1 Root swapping

Herlihy [1990, 1993] was the first to propose a universal method to convert a sequential data structure to a concurrent lock-free or wait-free version. For small structures, a thread attempting to update shared data makes a private copy of the entire structure

– to which the updates are then applied. The single root pointer to the data structure is then atomically modified to point to the private updated version using CAS, which will succeed if the structure had not been updated since the copy was made. This algorithm serialises all updates at a single point, the root pointer, so disjoint access parallelism is precluded. For larger objects, however, the cost of creating a local copy of the data structure could become substantial, making this method impractical. For larger structures (which are defined as being a collection of linked objects) Herlihy suggests only copying the parts of the structure which are being updated in order to reduce the memory and copy overhead. This may not have a substantial impact of the number of objects to be copied, however, as replacing a linked object requires updates to all the objects that link to it, so they must also be copied and replaced, and then the objects linking to those must be replaced also, and so on.

2.3.1.2 Cooperation descriptors

Turek et al. [1992] described a technique for replacing locks in a deadlock free lock-based algorithm, in order to make it lock-free. To perform a concurrent update, first, the sequence of instructions that would have been performed – had a lock been acquired – is converted to a sequence of virtual instructions. These are stored in a descriptor along with a virtual program counter and a version counter. A pointer to the descriptor is then stored in the same object that would be locked in the original lock-based algorithm – a null value here representing an unlocked state and a pointer value representing a locked state. The thread then begins processing the virtual instructions, but, if another thread encounters a descriptor stored in one of these locations, it must assist in processing the list of virtual instructions before it can continue and acquire the object with its own update operation. After each virtual instruction is decoded and executed, a DWCAS is used to update the virtual program counter and version counter. This technique doesn not disallow disjoint access parallelism, but it is likely that the emulation required to fetch, decode and execute the sequence of virtual instructions is expensive in comparison to updating the structure with atomic instuctions.

Barnes [1993] independently defined a very similar technique to that of Turek, Shasha and Parkash, where locks are replaced with pointers to descriptors containing sequences of virtual instructions that must be interpreted and executed. But he goes on to refine the design into a simpler form. Updates to individual objects are serialised via a field in each object that is either empty or contains a pointer to a descriptor describing

a sequence of updates being applied to the data structure, during which time the object must not be concurrently updated. When a thread is attempting to update the shared data structure, it makes private copies of the objects affected and performs its operation on these. The copies are then validated by rereading and verifying that no updates have happened to the affected objects since the copies were made, otherwise it must restart. The thread attempts to acquire exclusive access to each of the objects to be updated by copying a pointer to a descriptor describing the sequence of updates into them, ordering the acquisitions in such a manner as to avoid livelock. If successful, it will go ahead to perform the updates on the shared data structure and then remove the descriptor pointers. If another ongoing operation is encountered on a required object, the thread must first cooperate by processing the updates listed in its descriptor and release the object before it can continue. Because threads will assist in each others' operations before reattempting their own, throughput of updates is guaranteed but the algorithm as described requires nested LL/SC (in order to acquire access to each of the objects and ensure no concurrent updates occurred) which has not been implemented on any hardware.

2.3.1.3 Multiple CAS

Multiple CAS (MCAS) (sometimes called CASn) is an operation that executes CAS on n memory locations atomically for some bounded value of n. The MCAS interface takes in an array of CAS tuples comprising a memory location, an expected value, and the new value to write. It is not available in hardware on any known CPU but a software emulated MCAS typically allows for simpler implementations of lock-free data structures [Fraser 2003] at the cost of poorer performance compared with lock-free implementations of specific data structures.

Israeli and Rappoport [1994] presented a design for MCAS built using nestable LL/SC. Because nestable LL/SC is typically not available, a software implementation is provided based on DWCAS. Each location that is accessed using MCAS is required to have an adjacent bitmask field, comprising a bit for each thread accessing the structure. When attempting an MCAS, a thread copies all the CAS tuples into a globally readable structure also containing an operation state defining whether the MCAS is in progress, successful, or failed. Once all of the locations have been acquired in order and verified to contain the expected values, the operation can be considered successful and can proceed to write the new values to the locations. Acquiring a location is performed by

atomically setting a bit in the bitmask and verifying the expected value at the memory location. Acquisition can fail if either the location does not contain the expected value, in which case the operation must be rolled back and restarted, or it contains a set bit from another thread, in which case it must assist in completing the other thread's MCAS before continuing with its own. This can be considered an extension of cooperation descriptors and is required to make the design lock-free. This MCAS design limits the amount of concurrent MCAS operations that can occur to the number of bits that can be contained in the bitmask and its reliance on the emulated nestable LL/SC typically makes it inefficient. Anderson and Moir [1995] extend the basic design of Israeli and Rappoport to create a wait-free implementation that instead requires $O(log(n))$ bits per MCAS-able location where n is the maximum number of concurrent MCAS operations.

Harris, Fraser and Pratt [Harris et al. 2002; Fraser 2003; Fraser and Harris 2007] presented the first practical implementation of MCAS, requiring single-word CAS and up to two bits per memory location. First, a conditional CAS (CCAS) operation type is described. The specification of this operation is that a compare-and-swap will occur if another memory location is atomically verified to contain some value, or it could also be considered a double compare and single swap. This is implemented by creating a descriptor to contain the variables necessary to perform the CCAS: the two addresses, the two expected values and the single value to swap. A pointer to the descriptor is CASed into the memory location to be updated if the expected value is seen. Then the compare-only address is compared to the other expected value and if they match, another CAS is used to remove the pointer to the descriptor and swap it with the updated value. The MCAS operation is then constructed as follows: the list of locations and values to be CASed is copied into a descriptor, also containing a status value indicating whether the operation is ongoing, successful or failed. CCAS is used to copy a pointer to the MCAS descriptor into each memory location to be MCASed, conditional on the MCAS operation being in the ongoing state. This stops cooperating threads from performing incorrect work by acquiring memory locations on behalf of a completed MCAS. Some operations are described to distinguish between pointers to descriptors and the data stored in MCASable memory locations. A run time type identification (RTTI) system could identify the object type, but not all languages support this. Alternatively, two bits could be reserved in each memory location as flags to describe the object type – or a single bit for both types – storing additional type information inside the descriptors (this technique is described in more detail in section 2.3.2.3).

2.3.1.4 Transactional memory

In order to simplify the software design of non-blocking concurrent data structures, Herlihy and Moss [1993] introduced the concept of a Transactional Memory (TM). A TM allows more complex concurrent operations to be composed without worrying about the synchronisation of the underlying objects being accessed. The high level operation of a transaction is that it comprises a series of instructions executed by a single thread. Transactions appear to execute serially, as if successive transactions began after the previous transaction committed, and they are never affected by interleaving with other transactions. As a transaction executes, it makes tentative changes to shared memory locations it needs to update. These are not visible to other threads unless the transaction commits at which point all the updates become visible instantaneously. Envisioned as existing in hardware (HTM), it could be implemented as a per-CPU cache along with some new transactional instructions. Normal reads and writes are replaced with transaction versions which load the memory locations exclusively in the transactional cache or make changes directly in the cache. Successfully committing the transaction makes the data in the transactional cache available to other CPUs and the transactional cache lines are then considered the most up to date version of the data. Aborting the transaction causes the contents of the cache to be invalidated. Competing transactions requesting any of the same memory locations must have their conflict resolved by some other mechanism in order to avoid livelock and deadlock. HTM can be considered an extension of strong LL/SC working on multiple memory locations. The first availability of HTM in a commodity CPU is expected in the upcoming Intel Haswell microarchitecture [Intel].

Shavit and Touitou [1995, 1997] presented Software Transactional Memory (STM) as a means to emulate the functionality of HTM. Their implementation is somewhat restrictive, however, as the memory locations to be operated on during a transaction must be known in advance. It relies on using nestable LL/SC to acquire the transactional memory locations but suggests implementing this with DWCAS given its lack of support in hardware. They diverge from the cooperative methodology of Barnes – which can cause redundant recursive helping – and suggest aborting contending transactions rather than helping. An aborted transaction would then help the transaction that caused it to abort before retrying, maintaining the lock-free property. Less restrictive implementations have been developed by Fraser and Harris [Fraser 2003; Fraser and Harris 2007] (building on their MCAS design) and Herlihy et al. [2003b]. These support transactions that can be composed at runtime and can operate on an arbitrary

```
101 void addHead(Node* newNode)
102 {
103   newNode->prev = NULL;
105   atomic
106   {
107     if (head != NULL)
108     {
109       newNode->next = head;
110       head->prev = newNode;
111     }
112     else
113     {
114       newNode->next = NULL;
115       tail = newNode;
116     }
118     head = newNode;
119   }
120 }
```

Figure 2.7: Function to add a node to the head of a doubly-linked list. An atomic block is used so that threads which concurrently execute the function do not see the list in an intermediate state.

number of memory locations. They are also implemented directly using CAS.

Transactional memory enables significantly more straightforward atomic modification of data structures. Existing sequential algorithms can be adapted to make use of the 'atomic block' syntax to create lock-free concurrent algorithms without needing to make use of low-level atomic primitives directly. Figure 2.7 shows a function which adds a node to the head of a doubly-linked list and has been adapted to a concurrent form using an atomic block. The sequential implementation has its reads and writes to shared memory wrapped in the atomic block to enable concurrent execution via the transactional memory, avoiding a complicated and involved lock-free implementation built from CAS.

2.3.1.5 Performance

Fraser and Harris [Fraser 2003; Fraser and Harris 2007] performed detailed analyses of the performance characteristics of their MCAS and STM systems compared with lock-based and non-blocking algorithms. For large concurrent skip lists, their results show that CAS and MCAS based designs outperform those using STM due to the high overhead of STM operations versus the simpler accesses of CAS and MCAS. A skip-list built using STM is also shown to be significantly slower than lock-based skip list

implementations. When a smaller skip list structure was used, MCAS performance fell below that of the lock-based algorithms, while CAS still performed the best.

A binary search tree algorithm built using MCAS was compared to lock-based algorithms and was shown to outperform them when the set size is large. But, again, contention at smaller structure sizes caused it to suffer. STM performance when applied to a red-black tree was compared with lock-based designs which required either extensive locking or whole-structure locking due to the complexity of the red-black specification. STM was found to perform significantly better with higher levels of concurrency. The conclusion gathered from these experiments was that, from a programming perspective, it is simpler to design algorithms using STM than it is with MCAS and, similarly, it is simpler to design algorithms using MCAS than it is with CAS. In terms of run time performance, however, the algorithms built using CAS performed the best at both low and high contention levels. MCAS performed similarly to CAS at low contention but suffered at higher contention and STM tended to only perform better than RW locks.

2.3.2 Data structures

The following section reviews concurrent implementations of a number of common dynamic data structures including the current state of the art. These algorithms are implemented directly using hardware synchronisation primitives rather than the universal constructions mentioned in the previous section.

2.3.2.1 Stacks

Treiber [1986] described a very straightforward algorithm for a lock-free LIFO stack based on a singly linked list. Because stacks are only effected at one single point, a stack can be simply controlled by ensuring modifications to its top are made using CAS. Pushing a node onto the stack is done by first linking the node to the current top node, then using CAS to update the stack head pointer to point to the new node. Similarly, popping a node is done by using CAS to update the head pointer to point to the next node after the node it is currently pointing to. Treiber also identified that this approach is susceptible to the ABA problem during popping and suggested adding an ABA tag to the head pointer and modifying it with DWCAS.

2.3.2.2 Queues

Treiber [1986] presented a lock-free queue algorithm for the IBM System/370 based on a linked list and using tagged pointers. Enqueuing elements is done with a single-word CAS at the front of the list, but in order to dequeue an element, the entire queue must be traversed and the last element removed with a DWCAS. Treiber acknowledged the performance penalty of doing this when the queue is large and suggests an approximate FIFO algorithm as an alternative. In this design, the queue is essentially treated as two LIFO stacks. Enqueuing elements happens by adding them to the top of the enqueue stack. A dequeuing thread removes the entire list of added elements from the enque stack, reverses the order and adds them to the front of a dequeue stack. Any other dequeuing threads can then remove the first element from the dequeue stack if it is not empty, otherwise they must transfer another portion of the enqueue stack. A thread reordering a chain of queued items could be considered blocking here since other threads cannot access the elements in the chain until they are placed on the dequeue stack.

Another list-based queue described by Prakash et al. [1994] maintains pointers to both the head and tail of the queue in order to allow more efficient queue operations. A consistent snapshot of the head and tail must be acquired before any updates can be applied in order to determine the state of any concurrent operations. These must be helped to complete before the thread can progress. Head, tail and next pointers are all tagged in order to deal with the ABA problem. Enqueuing nodes happens at the tail and requires one DWCAS to add the new node to the back of the queue and a second DWCAS to swing the global tail pointer. Dequeuing is a much simpler operation and is performed at the head as per a LIFO stack with a single DWCAS.

Valois [1994] presented a simplification to Prakash, Lee and Johnson's queue by keeping the most recently dequeued item at the head of the list-structure making the next node in the list the logical queue head. This greatly simplifies the number of special cases that must be dealt with (when the queue is empty or contains one element) and eliminates the need for obtaining a snapshot of the head and tail before an operation can progress. Because of the ordering of adding a new node into the queue and then updating the tail pointer, it becomes possible for the tail pointer to lag behind the head pointer. This could break the queue if the node pointed to by the tail pointer was then reclaimed. This issue, and avoidance of the ABA problem occurring at the head and tail (which are updated with a single-width CAS), are dealt with by implementing a reference counting system. While the described implementation contained a race

condition, this was corrected later by Michael and Scott [1995]. In testing of their own queue algorithm [Michael and Scott 1996], they also identified a case where the memory requirements of Valois' queue could become unbounded. If a thread dequeues a node and stalls while still holding a pointer to the node, that node and every node linked after it (which is every node enqueued since the thread stalled) cannot decrease their reference counts and cannot be reclaimed.

The queue presented by Michael and Scott [1996] represents a safer version than that presented by Valois. The tail pointer can lag, at most, one step behind the end of the queue which ensures that the tail node cannot be dequeued. This avoids the issue by which the node currently pointed to by the tail pointer can be reclaimed. In order to remove the issue where memory use could be unbounded, they make the head, tail and the queue links tagged pointers (which get updated with DWCAS) and store removed nodes on a freelist. Michael [2002b] later did away with the need for tagged pointers using his SMR memory management system.

Ladan-Mozes and Shavit [2004] focussed on reducing the cost associated with using at least two CAS operations to enqueue a node. As a CAS requires memory barriers, requires exclusive cache line access, and flushes the processor's write-buffer; they state that a simple store can be done an order of magnitude faster. At a structural level, they reverse the order of the linked list so that nodes can be enqueued with a single CAS at the head of the list. This means that when dequeuing a node, it would not be able to see the previous node in the list, therefore the list is made doubly-linked. The normal next pointers are guaranteed to maintain a correct structure and the previous pointers are optimistically set with a store when a node is enqueued. If a thread is delayed or stalled before writing the previous pointer and the tail of the list catches up, the dequeuing operation must walk the length of the queue, fixing any inconsistent previous pointers along the way. In order to simplify the cases where the queue is empty or has a single element, if the last element is removed, a dummy node is enqueued to retain the list structure. This dummy node is skipped over if encountered by any dequeuing operation. Tagged pointers are used in order to avoid the ABA problem and to test for consistency in the optimistically set previous pointers.

2.3.2.3 Linked lists

Concurrently adding new nodes into to an ordered linked list is a simpler operation than removing because it can be achieved with a single CAS at the insertion point. If

another thread attempts to perform a concurrent insertion at the same point in the list, its CAS will fail when it finds the node's next pointer has changed. Considering a sequential implementation of a linked list for a moment, removing a node is usually completed with a single write – updating the next pointer of the predecessor node to point to the successor node. This technique cannot be directly applied to a concurrent implementation due to a race condition between one thread updating the next pointer of the predecessor and another thread updating the next pointer of the node to be removed. If the second thread concurrently inserts a node positioned directly after the removed node, it will assume the operation succeeded but the inserted node will be inaccessible. Alternatively, if a concurrent operation succeeds in removing the successor to the removed node from the list, the successor node will be incorrectly reintroduced to the list structure. Along with violating the specification of the linked list, this second case can cause further problems if the successor node was then reclaimed.

The first effective design of a lock-free linked list is credited to Valois [1995]. His design places intermediate 'auxiliary node' objects in between the logical nodes in the list. The auxiliary nodes act as a layer of indirection between successive nodes and simplify concurrent interactions between competing operations. Removals from the list can then be safely performed by joining two auxiliary nodes together, but this can then allow chains of auxiliary nodes to form. Each node also contains a backlink pointer to the previous valid node when it is removed to aid in the removal of its chained auxiliary.

Harris [2001] presented a solution to implementing a lock-free linked list by performing the node removal operation in two stages. The first stage is to 'mark' the next pointer of the node to be removed. This process leaves the pointer data intact but it signals that this node's key is now logically removed from the list. It also prevents concurrent updates to the node because by changing the data contained in the next pointer, any CAS expecting the unmodified pointer data will fail. The second stage is the same as for the sequential list, where the node is physically excised from the list. Concurrent operations can safely traverse over these marked nodes by decoding the pointer data and moving on. But, if there is a conflict, that thread must first assist in the removal of the marked node before it can attempt its own operation.

The manner in which the pointers are marked makes use of the fact that when memory is allocated, the memory address returned will be aligned on a boundary matching at least the size of a register[4]. This means that the least significant bits of

[4]The POSIX standard guarantees pointer alignment will be a power of two multiple of `sizeof(void *)`. The GNU C library guarantees at least double the register size alignment for calls

a pointer are guaranteed to be zero and so can be used to store auxiliary data (in this case a bit to signify a marked state) – provided they are properly masked away when the value needs to be treated as a pointer.

Michael [2002a] refined the design of Harris' linked list algorithm in order to make it better compatible with general memory management methods. He also provided specific implementations supporting IBM freelists [IBM 1983; Treiber 1986] as memory management (using tagged pointers) and his own Safe Memory Reclamation (SMR) method. These changes require forcing traversals to remove marked nodes encountered on the search path as they progress.

Another extension to the Harris and Michael algorithms was presented by Fomitchev and Ruppert [Fomitchev 2003; Fomitchev and Ruppert 2004]. Their modification involves converting Harris' two-step removal to a three-step process. The predecessor node first has its next pointer flagged using another low-order bit to avoid concurrent updates, then the node to be removed has a backlink pointer set to the predecessor and it is marked. The final stage is to physically excise the node from the list. The non-blocking property is maintained because threads observing flagged or marked nodes can help the operation to complete. The motivation behind this modification is to reduce the penalty of failing to successfully help an observed operation. In the Harris and Michael algorithms this would cause the operation to restart its traversal from the head node, but this algorithm allows the backlink pointers to be followed until an unmarked node is reached.

Heller et al. [2005] argue that most accesses of concurrent sets would be membership tests and therefore having a wait-free and read-only version of this method is preferable, at the cost of not having non-blocking updates. Traversals progress without acquiring any locks, but a physical insertion or removal requires blocking updates to a node and its predecessor.

2.3.2.4 Deques

Most of the early lock-free deque (double-ended queue) implementations [Greenwald 1999; Agesen et al. 2000; Detlefs et al. 2000] rely on double compare-and-swap (DCAS or also called CAS2). Double compare-and-swap differs from double-width compare-and-swap because it can operate on two non-contiguous memory locations as opposed

to `malloc` and `realloc`.

to being restricted to two adjacent memory locations. DCAS is supported by certain members of the Motorola 68000 family of processors but is currently not available on any modern processor. In the literature, its implementation was considered slow [Greenwald and Cheriton 1996] therefore designs that could work around a DCAS requirement were preferred. Doherty et al. [2004] argued that supporting DCAS in hardware could not be justified as it does not provide enough additional flexibility over CAS.

Michael [2003] provided the first list-based lock-free deque algorithm that can be implemented efficiently on modern hardware. The nodes are stored internally as a consistent doubly-linked list. The anchor point of the deque stores both the front and back pointers in a single block so they are always updated together along with some state information. This allows the algorithm to have a clear linearisation point for all operations, but means that it is not disjoint access parallel when operations are acting on different ends of the deque. Modifying both pointers in a single operation requires DWCAS, but Michael also describes a statically sized deque that is allocated from a single block of memory. This would allow more redundant bits of a pointer to be repurposed, enabling the anchor to be packed into a single word and to be updated with CAS. Dequeuing nodes from either end of the deque can be done with a single atomic operation, by moving the front or back pointer along one node, however, enqueuing must be done in two stages. The first stage is that the new node is linked into the list structure and the anchor pointer is updated to point to it. When the anchor is updated, a state bit is also set to signify to other threads that the list structure must be made consistent before the operation can be cleared and the next operation can occur. This is because the list structure could be broken if a concurrent enqueuing operation also linked its node into the list, in which case CAS is used to repair the link. Once this has been verified, the anchor is atomically updated again to remove the state flag and allow the next operation to proceed.

Sundell and Tsigas [2004b, 2008] proposed a deque algorithm that is similarly based around a doubly-linked list but with physically separate sentinel head and tail nodes in order to allow disjoint access parallelism when operations access opposing ends of the deque. The consistency of the list is maintained in a single direction only, using the linked list design of Harris [2001]. The previous pointer in a node is only updated once the change has succeeded on the next pointer. To enqueue a node, its links are initialised and CAS is used to link it into the next chain of the list. Then a further CAS is used to link it into the previous chain. Dequeuing nodes is similar, the next pointer of the node to be removed is marked, then its previous pointer. The excision of the node

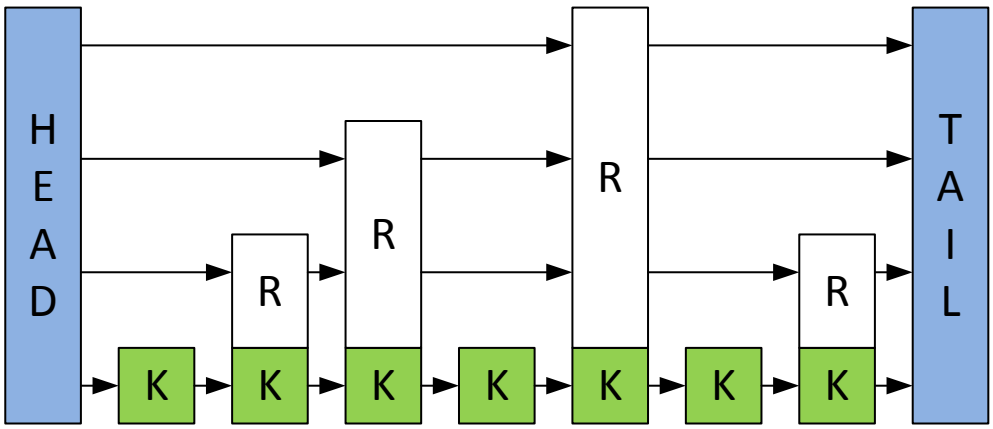

Figure 2.8: Structure of a skip list. Every key K in the skip list is present in the base level. The upper routing levels R allow for more efficient searches by skipping large portions of the list.

from the list structure then follows a similar pattern of using CAS to remove the node from the next chain, and another CAS to remove it from the previous chain. Sometimes, the previous links will not resemble the same structure as the next links if threads are delayed or stalled while inserting or removing a node. If this state is encountered by another thread, it must follow the links down the list until an unmarked node is found and then repair the previous chain so that it mirrors the structure of the next chain. This list structure was also extended [Sundell 2004] to allow adding and removing at arbitrary points making it a complete lock-free doubly-linked list algorithm.

2.3.2.5 Skip lists

Skip lists were first presented by Pugh [1990b] as an alternative to balanced trees. The structure of a skip list is that of multiple levels of linked lists where every element in the data structure is present in the base level and for every level that a node is present in, it is probabilistically present in the next higher level up to some maximum level (Figure 2.8). This allows searches to use these higher levels of links to 'skip' over large chunks of the base level list during a traversal. The randomised nature of the skip levels makes the structure probabilistically balanced giving it an average *O(log n)* time complexity for locating a node so they are often used in place of binary search trees. Pugh also presented the first concurrent version [Pugh 1990a] of a skip list using per

node locks to make updates. Nodes being removed are reduced by one level at a time until just the base level node remains. After being excised, the next pointer of the node is set to the previous node so that removed nodes can be detected by concurrent searches.

A number of similar lock-free skip-list algorithms based around Harris' linked list were developed independently and published in the same time frame [Sundell and Tsigas 2003, 2005, 2004a; Fraser 2003; Fomitchev 2003; Fomitchev and Ruppert 2004]. In order to maintain the consistency of the key set, the bottom level linked list operates in the same way as the Harris list where nodes are marked before they are removed. Once the base level has been marked, the skip links inside the node are then marked consecutively. If other threads encounter the marked links during a traversal, they will also attempt the excise the node from the level currently being searched. Adding nodes also happens at the base level first in order to linearise the insertion and the node is then linked into the higher levels while checking that the node has not suffered a concurrent removal during the process. Marked nodes receive backlink pointers to the previous node so that concurrent updates to a node can be helped or recovered from without restarting a search from the head. Fomitchev and Ruppert's design diverges slightly from that of Pugh in that the skip links are not stored in the base node objects – each layer in the skip list is a self-contained linked list with its own node objects. These then contain links to traverse forward or downward through the skip list. However, they do suggest that their algorithm could be adapted to store the links inside the nodes if optimal use of space is required.

The skip list *implementation* most commonly referenced in the literature [Herlihy et al. 2006; Bronson et al. 2010; Ellen et al. 2010] is the `ConcurrentSkipListMap` class written by Doug Lea [Lea] and released as part of the `java.util.concurrent` package in the Java SE 6 platform. The implementation is similar to the above algorithms but uses individual node objects at each level in the tree, as Fomitchev and Ruppert do, rather than an array of skip links. Lea argued that this would be a more optimal solution due to reduced contention on the index lists containing the skip links and simplified traversal operations.

2.3.2.6 Binary search trees

Binary search tree (BST) algorithms are typically divided into either external trees or internal trees. An external tree only stores keys in leaf nodes (Figure 2.9) as opposed

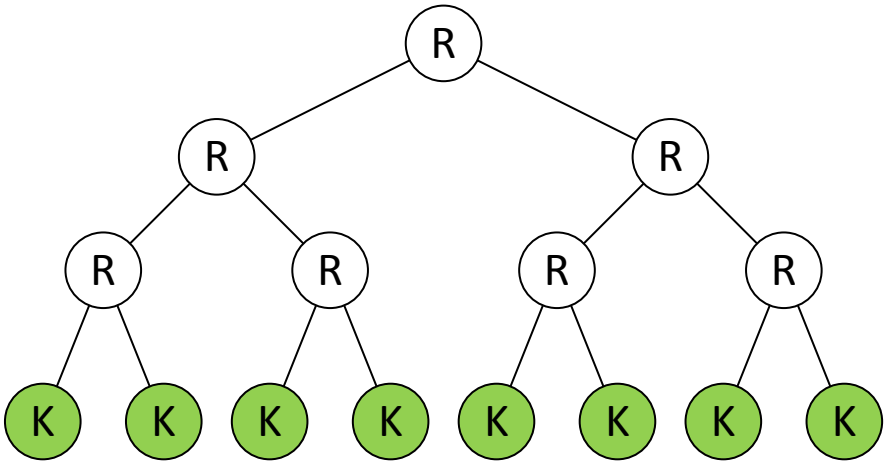

Figure 2.9: Structure of an external binary search tree. Every key K in the tree must be contained in a leaf node. The upper routing levels R allow searching for a key but their key values do not indicate a leaf with that key exists in the tree.

to an internal tree which stores a key in every node. The higher up nodes in an external tree also contain keys but these are only used for comparison and routing, a key contained in a routing node is not necessarily an element in the tree's set. The main advantage to using an external tree is that it simplifies concurrent node removal because only consecutive nodes at the edge of the tree are affected.

Fraser [2003], described a lock-free implementation of an internal BST which uses multiple CAS (MCAS) to update multiple parts of the tree structure atomically. Using his software-based MCAS algorithm, if all the locations required for an update are acquired, they can be updated with the new values, otherwise the operation must be rolled back. In order to detect concurrent restructuring when a node is repositioned in the tree, a 'threaded' tree representation is used. In this schema, instead of leaf nodes containing null links, they are set to point to their immediate successor and predecessor in the tree ordering. When a search terminates at a leaf, it can validate what it had observed the successor and predecessor nodes to be during its search against the data contained in the threaded links in order to detect a concurrent modification which might invalidate its search result. The threaded leaf structure and the design of the algorithm means that removing a node with two children requires up to eight memory locations to be updated atomically, which adds an appreciable overhead to the BST algorithm given its implementation using software MCAS.

Ellen et al. [2010] have developed the first specific lock-free external BST algorithm. Changes to the routing nodes are performed by applying Barnes [1993] method of cooperative updates. To remove a key, for example, a pointer to a structure containing all the information necessary to complete the update is copied into the two routing nodes which are to be modified and, if successful, the routing node pointing to the leaf node to be removed can be marked. The parent of this routing node then has its pointer switched to point to the remaining child of the marked routing node, thus excising both a routing node and leaf node. The final clean-up stage is to remove the pointer of the cooperation structure from the parent routing node. Any thread which is obstructed by the ongoing update can read the information in the cooperation descriptor and attempt to complete the operation on behalf of the initialising thread. The use of an external tree means that the worst case scenario is just two consecutive routing nodes being obstructed in this way. The algorithm contains an interesting memory management issue whereby a discarded operation descriptor must leave a valid pointer to itself in the node after the operation is complete in order to avoid the ABA problem, the assumption being that this value is unique and will not be used in this field again. This system does not permit arbitrary reuse of the descriptor object unless it is guaranteed that its pointer data is not contained in any nodes, otherwise it is susceptible to the ABA problem.

The concurrent AVL tree designed by Bronson et al. [2010] uses per-node locks to serialise concurrent updates and a relaxed balancing property which does not enforce the strict AVL property, but results in a tree which is approximately balanced and allows for future repair. The tree uses a 'partially external' design such that by default nodes behave as per an internal tree – where they represent a key's membership in the set – but in order to simplify the removal of nodes with two children, the defunct node objects are allowed to remain in the structure and act as routing nodes. This helps to avoid the problem of locking large parts of the tree if a remove-and-replace operation was implemented as per the sequential BST specification. These routing nodes can then be either excised during a later rebalancing operation or, if the node's key is reinserted, the node can be 'reactivated'. This partially external tree design was experimentally shown to have a negligible increase (0-20%) on the total number of nodes required to contain the key set. Searches acquire no locks and optimistically traverse the tree, but must take precautions due to the possibility of concurrent rebalancing operations. A search can be invalidated if the tree is restructured, so updates which would affect the search range in a node's subtrees must write to a version counter in the node so that it can be detected and the search is rolled back. Searches are read-only and can generally

traverse over locked nodes except in the case of certain rebalancing node rotations when they become blocked. The experimental evidence in the paper show this algorithm to significantly outperform a red-black tree implemented using STM.

2.3.3 Memory management

Most programming languages require explicit operations to manage the dynamic allocation and releasing of memory in software. For example in the C programming language, the `malloc()` function is used to allocate a specific number of contiguous bytes and to later release this memory, an explicit call to `free()` must be made. Accessing memory which has been deallocated is considered an error and causes undefined behaviour (such as a segmentation fault). If all the pointers to some allocated memory are discarded and it has not been released, it is referred to as a memory leak because the program has no way of accessing it and it remains resident in memory. Some runtime environments provide automatic garbage collection, for example the Java Virtual Machine, which can identify objects which are not reachable any more and reclaim them, but these frequently require stopping all execution so that the in-use memory can be scanned. Concurrent programming presents a further complication: determining when a memory object can be safely released if it is possible that it could be accessed by another thread at some arbitrary time in the future. The memory management systems outlined in this section describe solutions to determining a safe time to reclaim shared memory objects that are no longer globally visible.

2.3.3.1 Reference counting

Valois [1995] proposed a solution to the memory management problem encountered with his linked list algorithm using per object reference counts. Reference counters count the number of references there are to an object in the system. When a new pointer is created, the counter is incremented and when a pointer is destroyed or overwritten, the counter is decremented. When the counter reaches zero, there are no more pointers referencing the object which means it will never be accessed again, so the object can be safely reclaimed. Valois' lock-free algorithm uses atomic operations to update the reference counter to ensure that the counter is synchronised between threads. When the counter reaches zero, an atomic test-and-set operation is performed on a bit flag which, if successful, allows a thread to take ownership of the node in order to reclaim it. This accounts for the case where another operation is trying to increment the reference count

concurrently and also observes a zero reference count when it releases the node. The algorithm contains a race condition, however, as reported by Michael and Scott [1995] if the node is being reused while a concurrent operation is removing an incorrectly incremented reference count. In this case, the concurrent operation would observe the reference count becoming zero and attempt to reclaim the node. Because it is possible for reclaimed nodes to have their reference counts accessed, it means that the object's memory cannot be arbitrarily reused (by reclaiming, for example) but can be reused only as a new node object. Reference counting is a relatively straightforward memory management system but is expensive if many objects must be accessed in succession due to requiring two atomic writes per object – to acquire and release. This introduces additional contention to the data structure as accessing a reference counted object will cause cached copies to become invalidated, even in the case of read-only operations.

Detlefs et al. [2001, 2002] presented a method of performing lock-free reference counting that ensures nodes that may have been reclaimed will not have updates made to their reference counts. Their solution requires the use of DCAS when a node's reference counter is to be updated. The second memory location that is accessed as part of the DCAS is the location that led to this node being accessed. In this case, this location is not modified, but is used to verify that a pointer to the node still exists so the node could not possibly have been reclaimed since the pointer was first read. This method generally allows arbitrary reuse of reclaimed memory, but certain cases can cause it to fail. For example, if the virtual memory address that was associated with a reclaimed node is remapped for some other purpose; performing a read from that address during the DCAS could cause an error.

2.3.3.2 Deferred reclamation

The concept of retiring an object from use, which was previously concurrently accessible, by placing it in some auxiliary storage area (usually a list or queue) and deferring reclaiming the memory until some later safe time is a widely used technique. The idea was first described by Kung and Lehman [1980] who used two such lists. Instead of being reclaimed or put onto a freelist, defunct objects are put onto the first retire list. Periodically, a garbage collector thread locks this list and transfers the objects to the second list, leaving the first list empty. Once all concurrently running threads have terminated (or the less restrictive condition proposed by Pugh [1990a]; that all threads have indicated they have left the critical region where they could have acquired

a pointer to one of the retired objects), all the objects on the second list can be reclaimed or put onto a freelist and the process repeated. It is suggested that this could be implemented via globally readable timestamps posted by threads when they start and terminate. Harris [2001] describes a slight modification of this technique as a solution to manage the reclamation of nodes in his linked list algorithm. Harris' system uses a global timestamp counter which threads copy into their own globally readable variable before starting an operation. He reduces contention and the need for a global worker thread by using two per-thread retire lists. Retired nodes are added to the first list and periodically this list is swapped to the second list and stored with a timestamp. By checking the per-operation snapshots of the other threads, once they have been observed to all be greater than the second retire list's timestamp, the entire list can be reclaimed and the whole scheme is restarted. Harris noted that doing this added a appreciable 52% overhead when performed after each operation and advised amortising the cost over more operations. This modification reduced the measured overhead to 5% if amortised over 100 operations and 1% if amortised over 1000 operations.

The Read-Copy-Update (RCU) mechanism [McKenney and Slingwine 1998] (which is used extensively in the Linux kernel [Arcangeli et al. 2003] since version 2.5.42) also heavily relies on these same retire list ideas. RCU is used to replace reader-writer locks that allow either multiple readers or a single writer to access shared data. RCU allows readers access to the shared data without acquiring any locks, removing contention, and writers must perform their updates such that any concurrent readers are guaranteed to see consistent data at all times. This then faces the same issue, as with concurrent data structures, of determining when retired objects can be reclaimed. The method used in RCU is, again, to store information on a list and at some time in the future, once it is guaranteed that no concurrent threads could access the shared memory location, reclaim it. In order to determine the point at which shared memory can be reclaimed, the concept of a 'quiescent state' is used. A quiescent state is defined as being a point at which a process is guaranteed to be holding no references to a RCU-protected structure. A 'quiescent period' is then any time period for which all threads have passed through at least one quiescent state. Any RCU-protected memory that has been removed from the visibility of other threads can be safely reclaimed once a quiescent period has passed. Most systems will have easily and cheaply tracked natural quiescent states that apply to all RCU structures in the system such as a context switch (it is forbidden for Linux kernel threads to context switch while holding a lock or while in an RCU-protected region), otherwise artificial quiescent states can be inserted.

Fraser [2003] describes a similar system that does not depend on the tracking of external states called 'epoch-based reclamation'. This system uses a global epoch counter or timestamp that can be seen by all threads. When a thread wishes to enter the critical section and access the shared data structure, it must store a globally readable copy of the current epoch. At this point, if all other threads working inside a critical section are also operating in the current epoch, the epoch counter is incremented. This means that, at any point, all threads will always be operating in either the current epoch e or the previous epoch $e-1$. These two epochs each have a retire list associated with them and threads which are retiring objects add them to the retire list of the epoch they are operating in. It is therefore possible that a thread operating in epoch e could access an object that was placed on the $e-1$ retire list and even if the epoch were to then change, moving the thread to epoch $e-1$, the object would still be unsafe to reclaim from the $e-2$ retire list. At the point of an epoch change, once all threads enter the current epoch e, it is not possible that any thread could be holding a pointer to an object on the $e-2$ retire list and so the entire $e-2$ list can be reclaimed before the epoch is incremented again. This method of reclamation can be built into the data structure and ignored by the programmer (as opposed to the RCU implementation), but the reading and writing of epoch counters on a per-operation basic could carry an appreciable overhead. The RCU implementation relies on the fact that kernel threads in an RCU region cannot be preempted to avoid reclamation becoming blocked, but a user mode thread stalling inside a critical region would cause a situation where no memory can be reclaimed and dynamic memory use becomes unbounded.

2.3.3.3 Protected objects

Michael [2002b, 2004] presented another retire list based algorithm which he calls Safe Memory Reclamation (SMR) or Hazard Pointers. Instead of waiting until a grace period has passed to guarantee other threads have no references to a defunct object, this algorithm directly reads a list of references that other threads have indicated they might access. The idea is that each thread has a small fixed number of pointers which are globally readable. Hazard pointers are acquired by a thread by copying an object pointer into its private hazard pointer array. Normally, a check is then performed to make sure the object was not removed before the hazard pointer was made visible. Once acquired, the object can be safely read and is guaranteed not to be reclaimed for the duration of its acquisition. When an object is retired, it is placed onto a per-thread retire list which can either be processed immediately or allowed to grow to an arbitrary

size. Michael suggests allowing it to grow to twice the number of hazard pointers in the system, which would guarantee at least half can be reclaimed, reducing the cost per reclamation but this size must be chosen to balance amortising the reclamation cost over more operations and increased memory use. The reclamation process is performed by taking a snapshot of all the hazard pointers and comparing them to all the objects in the retire list. Any matches that cannot be reclaimed are left on the retire list for the next scan and all other objects can be safely reclaimed. This algorithm has a higher traversal cost than the other retire-list based algorithms because it requires a write and a potentially costly memory barrier for each object read. The only time shared data is accessed is when the entire hazard pointer array is read, and because this is a constant size it means that the algorithm is wait-free. A stalled thread can only ever block reclamation of the objects pointed to by its hazard pointers and so it avoids the unbounded memory usage of the previous retire list algorithms. Michael suggests extensions to the base algorithm to make it more efficient and useful, such as: reducing memory usage by using old data fields inside objects to link the retire list together, cache aligning the per-thread hazard pointer arrays to avoid false sharing between CPUs, and the ability to pass hazard pointer descriptor structures (hazard pointer array, retire list, etc.) between threads when they are finished operating on a hazard pointer protected data structure.

Herlihy et al. [2002, 2005] independently developed a memory management system quite similar to hazard pointers. They begin by defining the Repeat Offender Problem (ROP), an abstract problem that embodies the general problem of memory management of dynamic data structures. Any solution to ROP can be also used to solve the memory management problem. The proposed solution to ROP is called Pass The Buck (PTB). Before a memory location is accessed, a guard must be posted on that location to guarantee that the location will not be reclaimed for the duration of time it is guarded. Guards are dynamically assigned from a pool of available guards and are globally readable. At the end of an operation, if a thread needs to retire objects, it will pass a list of defunct objects to a *Liberate* function which iterates over the list of guards comparing the objects to the guarded references and returns a list of objects that can be safely removed. Each guard has a handoff pointer associated with it and if, when *Liberate* is scanning, it finds that one of the objects in its list is guarded, it copies a pointer to the object to the handoff pointer and discards the object from its list. This hands off the responsibility of reclaiming the object to a successive call to `Liberate`. As well as scanning for guarded objects, the `Liberate` function also collects handed off objects that are no longer guarded. The algorithm is wait-free but relies on DWCAS

when modifying the handoff pointers to avoid the ABA problem when they are set to null. A slight adaptation is outlined, however, that removes this requirement but makes the algorithm just lock-free. PTB has a much smaller memory footprint than hazard pointers (assumed using the suggested tuning parameters) and has the ability to offload guard scanning from application threads, but its operations are slightly more complex and Michael [2004] reports that his solution outperforms it.

2.3.3.4 Performance comparison

Hart et al. [2006, 2007] performed a detailed comparison of lock-free reference counting (LFRC), the quiescent-state-based reclamation (QSBR) used in RCU, Fraser's epoch-based reclamation (EBR) and hazard-pointer-based reclamation (HPBR). They argue that due to the similarity between PTB and hazard pointers, any analysis could be applied to both algorithms. In terms of throughput, LFRC was found to perform poorly compared with the other algorithms when traversing a linked data structure. The three other algorithms performed similarly on single element or queue based structures with QSBR shown to have slightly lower cost than EBR per operation due to reclamation being amortised over more operations. The single-threaded throughput of HPBR fell to half that of the other two when accessing a list-based structure that requires object traversal due to the per-object hazard pointer acquisition. The experimental analysis is quite limited however, focussed mainly on workloads experienced in the Linux kernel; read-mostly lists and queues and single-threaded overhead – with no experiments examining highly threaded or update heavy workloads.

2.4 Summary

This chapter has outlined all of the background information and related work that has influenced the designs presented in the next chapter. The lack of an existing algorithm for a lock-free internal binary search tree led to the decision to investigate that data structure. The main work that is presented advances from the lock-free external BST of Ellen et al. (described in section 2.3.2.6), and particularly incorporates the pointer marking (section 2.3.2.3) and cooperation descriptor (section 2.3.1.2) techniques.

Chapter 3

Design and Implementation

3.1 Binary search tree

3.1.1 Description

The main work of this thesis focusses on creating a concurrent binary search tree (BST). A BST implements the *set* abstract data type (ADT). A set comprises a collection of unique key values, of no strict ordering, with functions: `add(k)` to add the key k to the set if it is not already a member of the set; `remove(k)` to remove the key k from the set if it is a member of the set; and `contains(k)` to test for membership in the set.

A binary tree is structured from nodes such that each node has a single parent node (with the exception of the root node) and, at most, two child nodes. The *search* portion of a BST comes from the ordering rule that for any node in the tree, every node in its left subtree must have a key which is less than its own key and every node in its right subtree must have a key which is greater than its own key. Due to the fact that the search space is halved with each traversal of a node, a BST takes *O(log n)* time for add, remove and contains; giving it an advantage over sorted linked lists which require *O(n)* time. The worst case *O(n)* time can occur when keys are inserted in order which causes each node to only have a single subtree and the tree resembles a linked list.

Adding a new node to a BST always takes effect at the leaf nodes of the tree which are nodes with fewer than two children. The appropriate leaf is found by traversing nodes down through the tree, comparing each node's key with the key being added and either following the left or right child path. The ordering of the tree enforces strict

placement on newly inserted nodes – there is only a single possible insertion point that does not violate the ordering rules (Figure 3.1).

Two different methods are used to remove a node depending on whether it has two children or fewer than two. The more straightforward operation is removing a node with fewer than two children, whereby the node is unlinked and the child (if any) takes the position of the removed node in the tree (Figure 3.2). When removing a node with two children, it cannot be directly replaced by one of these children since, for example, each may already have 2 children and be unable to take a third. The algorithm used to process this removal is to first find the node with either the next highest or next lowest key in the tree. This can be found by continuously traversing either the right child links in the left subtree or the left child links in the right subtree until the path ends. The significance of these keys is that by replacing the removed node and/or its key with that of the found node, the ordering rules are preserved (Figure 3.3). Once a replacement node is found, the replacement is excised from the tree (as before – given that it cannot possibly have 2 children) and it is either repositioned to take the place of the to-be-removed node or its key is swapped into the node whose key is being removed.

3.1.2 Concurrency issues

The design of a BST presents several problems when creating a concurrent implementation. Starting the adaptation by considering a BST that supports the simplest of operations, `contains(k)` the current sequential design can already support concurrent readers. This is because the structure would effectively be read-only and concurrent searches have no effect on one another. An operation to add a node could be incorporated by using CAS to modify the null pointers of leaf nodes; any concurrent adds attempted at the same point in the tree would fail to complete their CAS. The effect on concurrent readers would be that a failure to locate a key would linearise at the point at which they read the null leaf pointer on their search path.

Removing a node, taking first the simpler case when it has fewer than two children, presents the first obstacle to a lock-free implementation. The sequential algorithm requires just a single write to excise the node from the tree, but applying this via a CAS to the concurrent design would introduce a number of race conditions. Concurrent removes could be undone, as in Figure 3.4, or concurrent adds and removes could result in children being added to a node during removal. These problems are similar to those encountered when removing nodes from a linked list with CAS. Considering

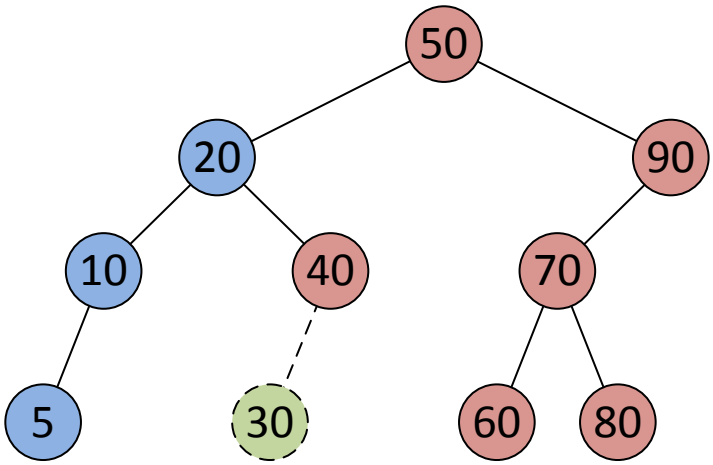

Figure 3.1: There is only a single possible leaf insertion point when adding the key 30. Blue nodes denote keys less than 30 and red nodes denote keys greater than 30.

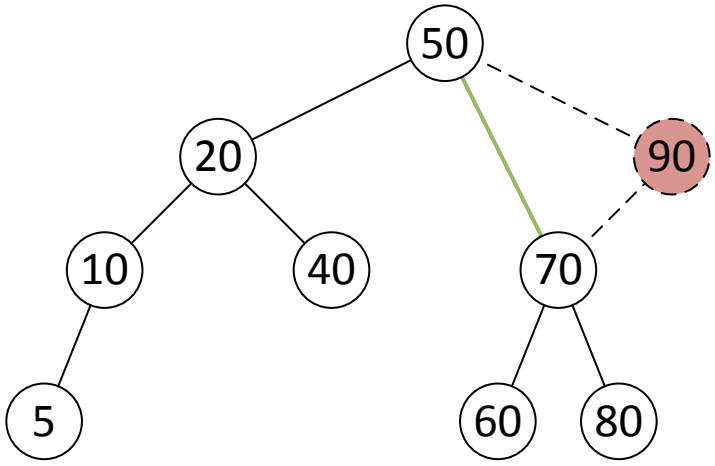

Figure 3.2: Removing node with key 90 from the tree. Because this node has only one child, its parent can simply swing its pointer to the child of the removed node, 70.

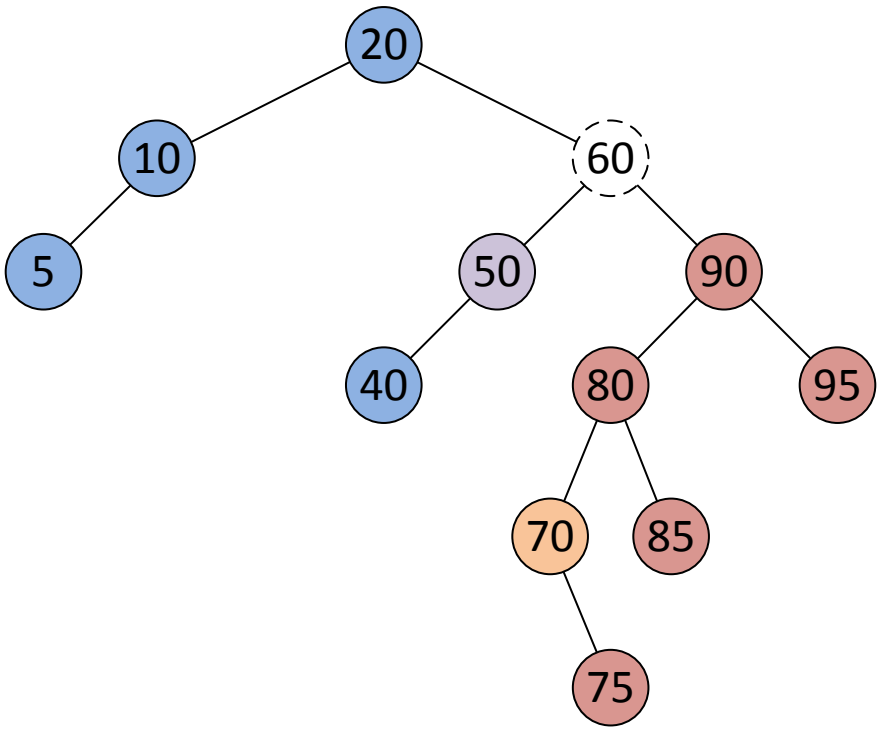

Figure 3.3: Removing node with key 60 from the tree. The only nodes which could take its position while still obeying the ordering conditions are 50 or 70 – the next lower and higher keys respectively.

the two-stage deletion of nodes from a lock-free linked list by marking the next pointer of the node to be removed, this technique would not solve the problem in the BST because both child pointers would need to be marked simultaneously. DWCAS could potentially be used to mark both pointers atomically but its relative lack of support makes it undesirable. Once these race conditions are resolved, however, concurrent `contains` operations will perform correctly as before.

Composing a set of steps to remove a node with two children presents further problems. The approach of relocating the replacement node would not be preferable for a lock-free design as it requires four pointer modifications on three distinct nodes (Figure 3.5). In contrast, updating the node's key and removing the replacement node (which is assumed a solved operation at this point) is conceptually more straightforward. The difficulty in achieving this is that both parts must appear to occur atomically. Key relocation will have an effect on in-progress searches that have passed through the updated node because the key range in that branch of the tree could have expanded or contracted. Figure 3.6 demonstrates an example where key relocation results in the

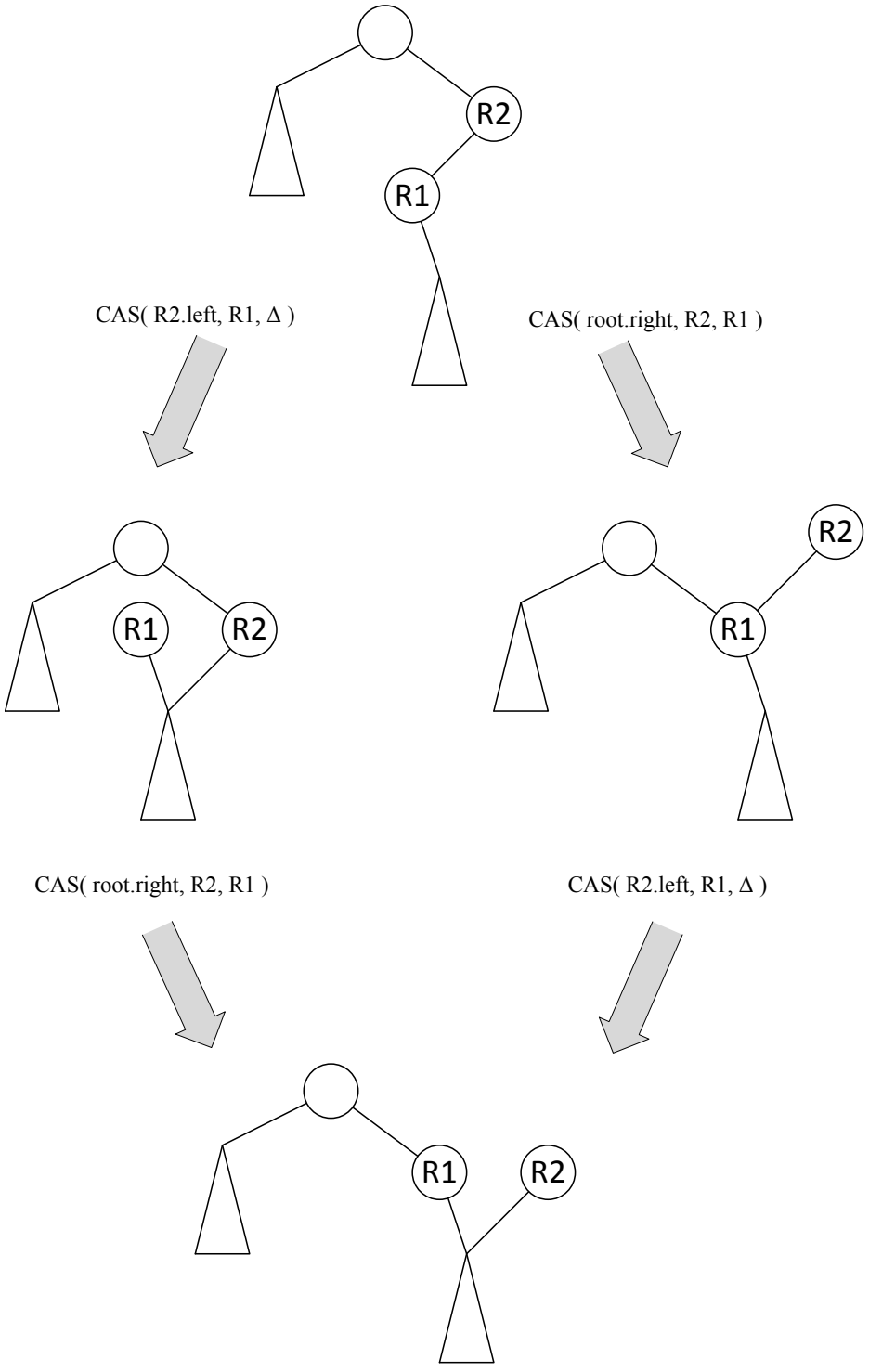

Figure 3.4: Parent and child node being concurrently removed via CAS. Regardless of which CAS takes effect first, each thread believes it has performed a successful removal and the end result is that R1 incorrectly remains in the tree.

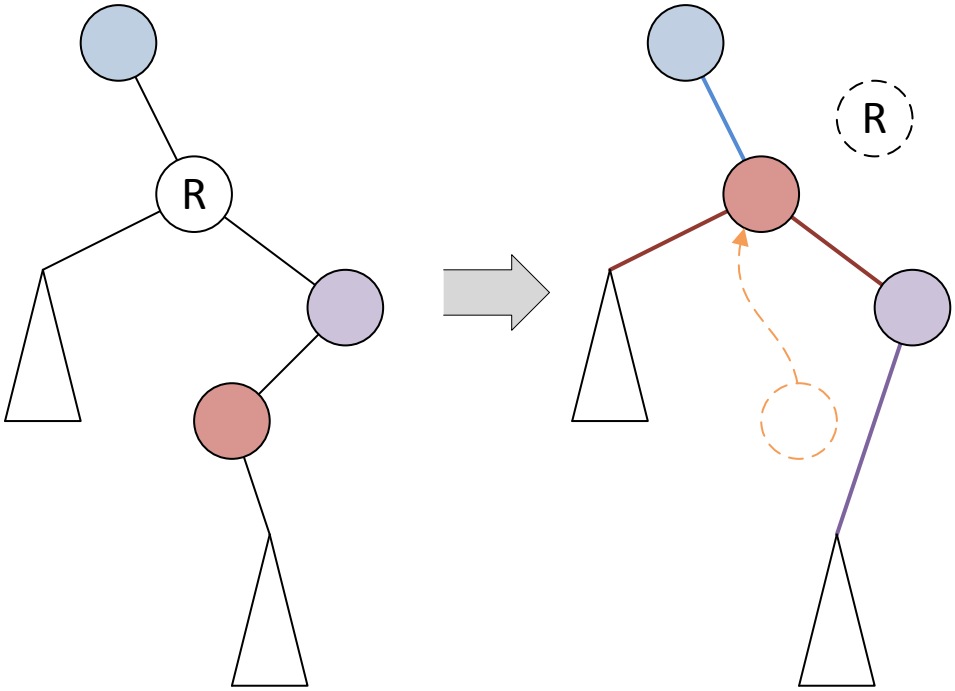

Figure 3.5: In order to reposition the replacement red node to take the place of the removed node R, three separate nodes and four child pointers must be updated.

incorrect insertion point being chosen in the tree. A similar problem exists for contains whereby it could determine a key is not in the set if the key range has contracted in its search path. A means of detecting upstream changes is therefore required in order to compose a correct BST algorithm.

3.2 Split-node binary search tree

3.2.1 High-level overview

This design simplifies synchronising concurrent changes to a node by performing them via a single pointer. This is done by splitting nodes into two objects: an immutable `Contents` object which contains the key and the child pointers and a `Node` object which contains a pointer to the node's `Contents` object. The main purpose of the `Node` object is to provide an additional layer of indirection between nodes. Adding new nodes to

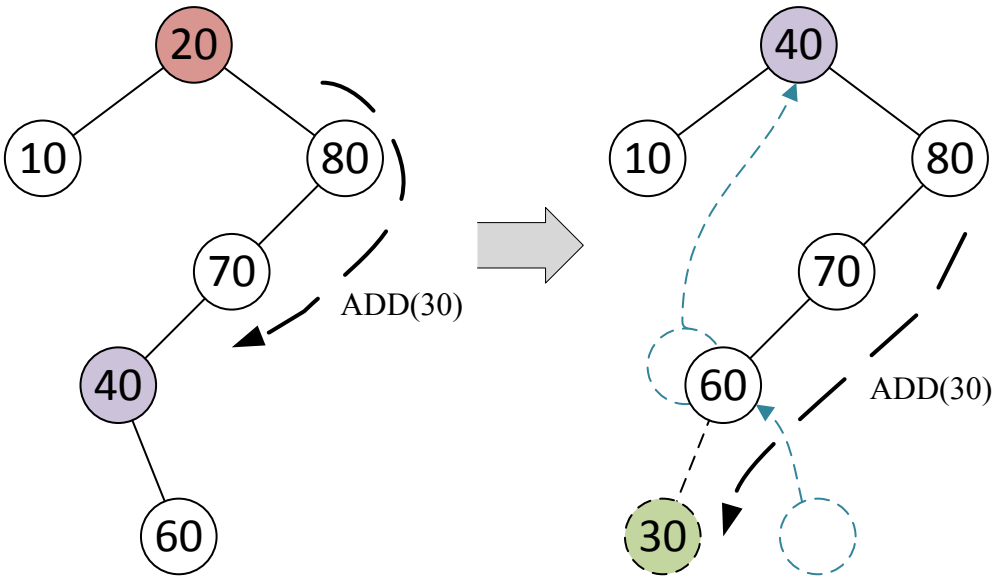

Figure 3.6: In the left hand tree, the key 20 is being replaced by key 40 while a concurrent operation to add key 30 to the tree is searching for its insertion point. On the right, the key relocation has taken place and the concurrent thread inserts a new node at where it determined to be the correct insertion point, left of node 60. The key range in this subtree had contracted due to the relocation so the correct insertion point should have been as the right child of node 10.

the tree is performed by first identifying the correct insertion point, and then copying the entire contents of the leaf node into a new `Contents` object – updated to include the newly inserted node. The physical insertion is executed by using CAS to change the contents pointer of the leaf node's `Node` object as in Figure 3.7. A consistent view of the contents is guaranteed due to the fact that `Contents` objects are immutable and a successful CAS means that the contents had not been updated since the node was read.

Removing a node with fewer than two children is performed in two stages. Once the node is found, CAS is used to mark its contents pointer using Harris' pointer marking technique [Harris 2001]. As with the insert, a successful CAS means that the node has not gained a second child in the meantime. The mark ensures that no further updates can be performed on this node and it cannot be unmarked at this point. Because of this, the node is now considered logically removed. The second

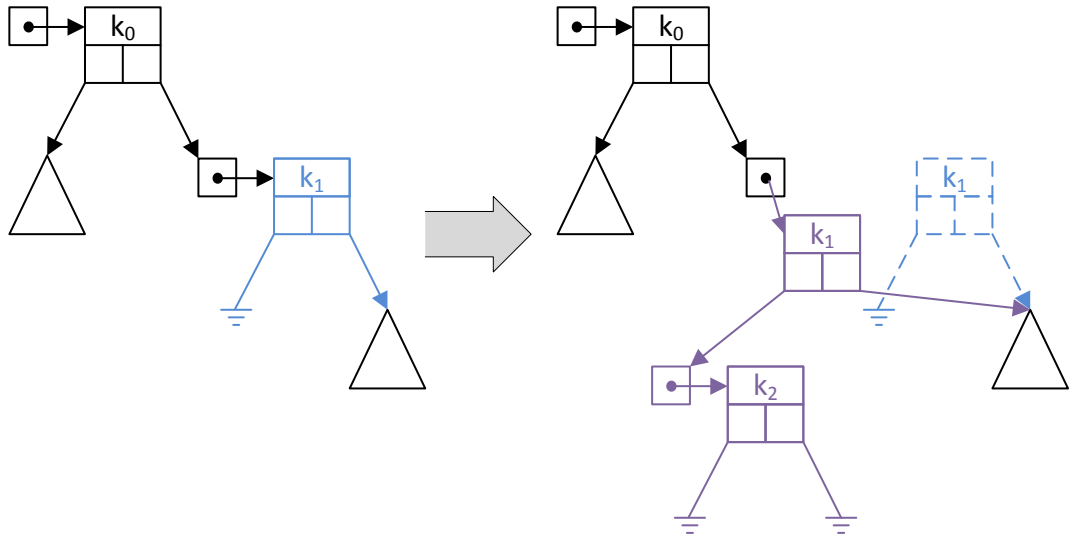

Figure 3.7: Split-node BST adding the key k_2 to the left of node with key k_1. The dashed blue contents object has been removed and can now be retired.

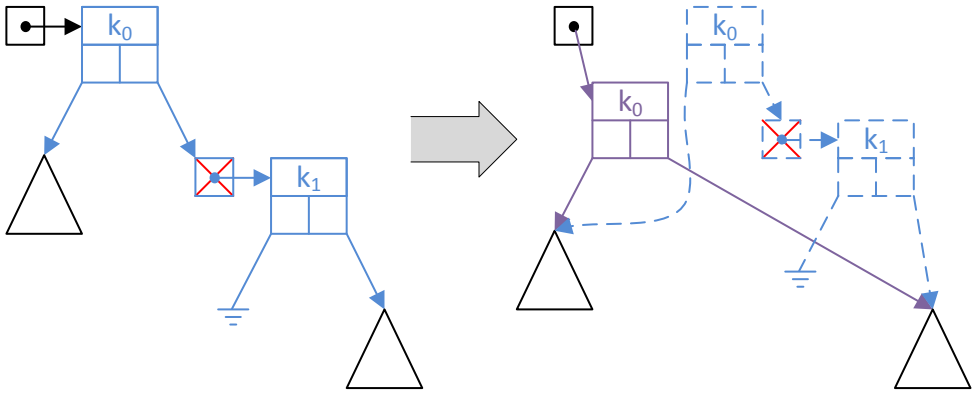

Figure 3.8: Removing the key k_1 from the tree. The red X denotes the node has been marked. The dashed blue contents objects and node have been removed and can now be retired.

stage, to physically remove the node, is performed by modifying the contents of the parent node to point around the removed node (Figure 3.8). The node can be excised by any concurrently executing thread that encounters the mark while traversing the tree.

In order to remove a node with two children, the node with the next largest key is located. A descriptor is used to store all the data necessary for another thread to complete the key relocation (or roll-back the changes if necessary) and a pointer to the descriptor is CASed in place of the contents of the node to be relocated. This is done to

prevent any further updates to this node while the operation is in progress. If the CAS was successful, a second CAS is executed to write the descriptor in place of the contents of the node to be removed. This ensures that the operation will succeed and the node's contents pointer is then switched to those containing the updated key. The final stage is to remove the relocated node which is done as described above. It is possible that the node to be removed could be updated before the second CAS is performed which will cause attempts of the CAS to fail. In this case, the operation must roll back by using CAS to restore the relocation node with its old contents and then the operation must restart. An example of the relocation process is shown in Figure 3.9.

When a search is in progress, the key range in the subtree currently being traversed is defined in terms of the last nodes for which the left and right subtree were taken (lastRight.key $<$ keyRange $<$ lastLeft.key). In the split-node tree implementation, key replacements are only done with the next largest key, meaning only the lower portion of the key range can contract. By checking for changes to the last node for which the right path was taken, searches can detect contractions of the search key range and trigger a restart. While key replacements can be performed with either of the next largest or next smallest key, the implementation is simplified by always choosing the next largest.

The choice to arbitrarily choose the next largest key for replacement will have an effect on the balance of the tree when random keys are inserted and removed, however. By always relocating nodes from the right subtree, the height of the left subtrees will generally be greater than that of the right subtrees. This can be mitigated by randomly choosing to perform a key replacement with the next largest or smallest key in the tree, but the search function must be augmented to monitor both the node for which the last right and last left path were taken, as changes to either one could cause contraction of the key range. This would introduce more retry paths to the search function which, under heavy update contention, would likely be hit more often.

3.2.2 Detailed implementation

3.2.2.1 Structures

The declarations of the classes used in this implementation are listed in Figure 3.10. The tree class needs only to contain a sentinel root element of the tree which is initialised with a contents object containing some dummy key (which is never actually read). The

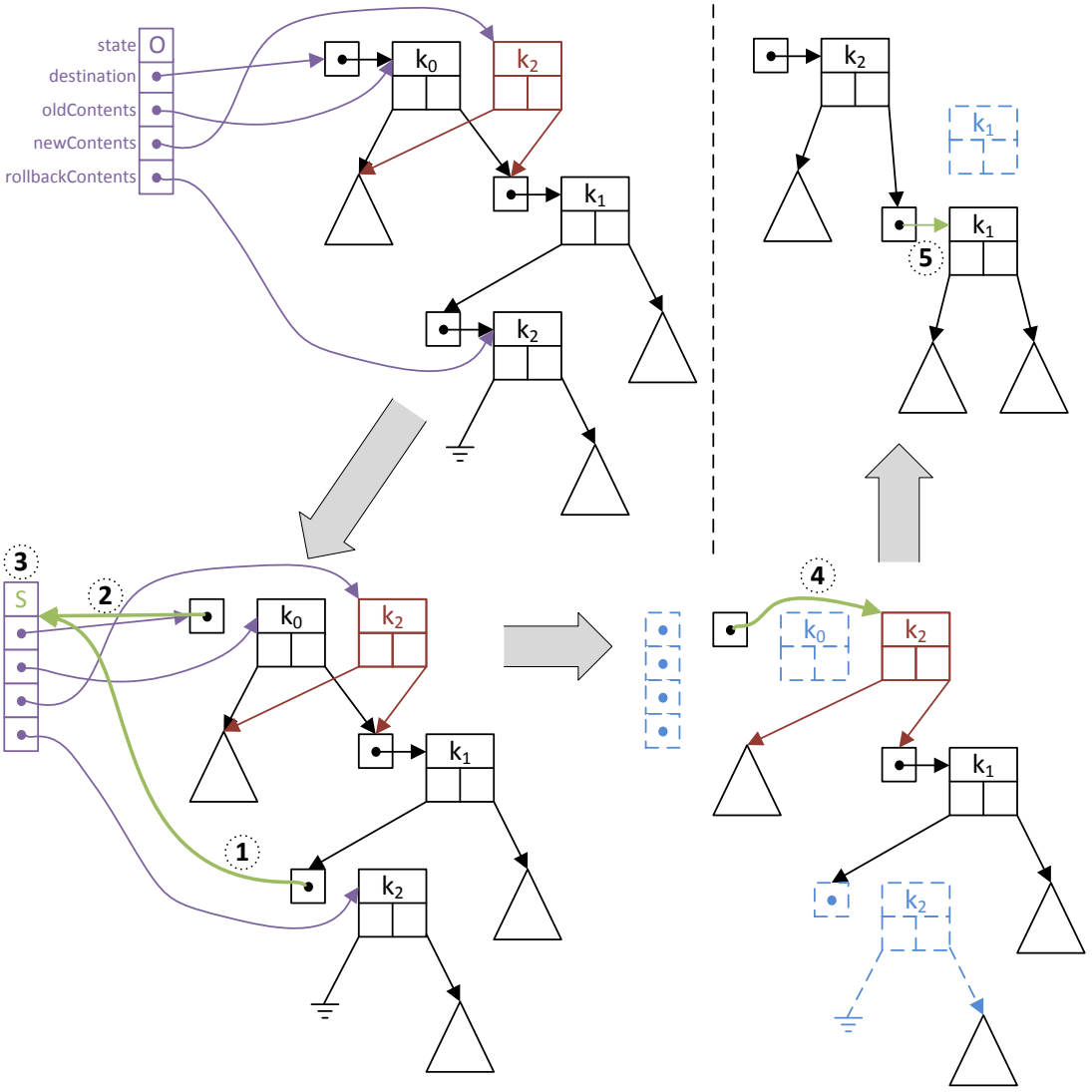

Figure 3.9: Removal of key k_0 from the tree by relocating key k_2 in its place. The first stage (top left) creates the purple `Relocation` structure and red `Contents` object. The CAS denoted 1 is used to modify the contents pointer of the node containing the relocating key, k_2, to point to the `Relocation` structure, then the CAS denoted 2 does the same for the node containing the removal key, k_0. Once these two CAS's have completed, the operation state is set to `SUCCESSFUL` (CAS 3). The 4th CAS is to point the node with the removed key to its new contents containing the updated key. The node with relocated key can then be safely excised by creating a new `Contents` objects for the node containing key k_1 and inserting it with the CAS denoted 5.

```
1  class SplitNodeBST
2  {
3    Node root;
4  }

6  class Node
7  {
8    Contents* volatile contents;
9  }

11 class Contents
12 {
13   int      key;
14   Node*    left;
15   Node*    right;
16 }

18 class Relocation
19 {
20   int volatile  state = ONGOING;
21   Node*         destination;
22   Contents*     oldContents;
23   Contents*     newContents;

25   Contents*     rollbackContents;
26 }
```

Figure 3.10: SplitNodeBST class declarations.

tree is initially considered empty and the first node to be added will be the real or logical root of the tree. It will always be positioned to the right of the sentinel node. This sentinel is added to simplify the implementation of various functions.

The `Node` class is a result of splitting tree nodes in two and is used to provide a level of indirection between successive nodes. It is also used to provide state information which is encoded in the contents pointer. The node can be in one of three states:

- `CONTENTS` – The pointer leads to a set of contents, the default unmodified state.
- `MARK` – The node's key has been logically removed from the set and the node is awaiting excision. The pointer data references the contents before removal.
- `RELOCATE` – The node is undergoing a key relocation operation, and the pointer data leads to a `Relocation` object describing the operation.

The following macros are also defined to aid in the modification of this data:

- `SETMARK(ptr)` / `ISMARKED(ptr)` – The first sets the state of the contents pointer `ptr` to indicate it is marked and the other resolves to a boolean which indicates if a pointer is marked.

- `SETRELOCATE(ptr)` / `ISRELOCATING(ptr)` – Performs a similar function but for the `RELOCATE` state.
- `GETREF(ptr)` – Strips away the state bits and returns the pointer data intact.
- `ISFLAGGED(ptr)` – Returns a boolean `true` if the contents pointer is in any other state than the default.

If the platform does not support encoding extra information into object pointers (e.g. Java), then the state can be stored inside the operation objects if they are rewritten to inherit from a common base class.

The contents class contains the data each node is required to store: the key, left child and right child. The objects are immutable so all updates to nodes must happen through the pointer contained in the `Node` class.

A `Relocation` object contains enough information for another thread to attempt to complete an operation which removes the key of a node with two children and replaces it with the next largest key in the set. The following information is required: A pointer to the `Node` whose key is to be removed, the data stored in that node's `contents` field when it was read, a pointer to a newly created contents object which is copied from the current (but with an updated key), and a pointer to the contents of the node whose key is to be relocated. Any inconsistencies in this snapshot are safe because they would have the effect of causing a CAS to fail and the operation would restart. The destination must be stored in this descriptor because a pointer to the structure is stored in the `contents` field of both the node whose key is to be removed and the node whose key will be relocated and it is necessary to differentiate between the two nodes when their `contents` fields are read. The `rollbackContents` are only used if the relocation fails and the changes already made must be undone. The purpose of the state field is covered in the explanation of the remove operation in section 3.2.2.4.

3.2.2.2 `contains(k)`

The code listing for the contains operation can be seen in Figure 3.11. It determines membership in the set via the return value from the `find` function (Figure 3.12 and Figure 3.13) which is a searching function used by all the operations to locate the position or potential position of a key from a specified start point `auxRoot` (which is normally passed as `root`) in the tree. The position and its predecessor's position are

```
27 bool contains(int key)
28 {
29   Node* pred, * curr;
30   Contents* predData, * currData;

32   return find(key, pred, predData, curr, currData, &root) ==
         FOUND;
33 }
```

Figure 3.11: SplitNodeBST `contains` function.

returned in the variables `pred` and `curr` and the values of their `contents` fields when traversed are returned in variables `predData` and `currData`. The result of the search can be one of three values:

- `FOUND` – Indicates `k` is a member of the set.
- `NOTFOUND` – Indicates `k` is not in the set but would be positioned as a child of `curr` if it were inserted.
- `ABORT` – Indicates the search of a subtree cannot return a usable result and must abort.

`ABORT` cannot be returned by a search that starts from the root and is only seen when searching for the next largest key during a remove operation.

The search begins by initialising the search variables `curr`, `currData`, `next`, `lastRight` and `lastRightData` before the traversing loop starts. `next` is used to point to the next node along the search path, after `curr`. `lastRight` and `lastRightData` are used to keep a record of the last node (and its `contents` pointer) for which the right child path was taken. The check at line 44 is to catch an invalid state when the search begins for a next largest node, but is never triggered under a normal search since `root` can never be flagged.

The search loop traverses nodes one at a time until either the key is found or a null node is reached. The checks at lines 60 and 88 deal with the situations in which a contents pointer is not in the `CONTENTS` state and has an ongoing operation attached to it. This code will be examined in the explanation of the remove operation. When the traversal has completed, the check on line 129 is to verify that, having searched `lastRight`'s right subtree, `k` cannot have been concurrently added to `lastRight`'s left subtree. If it has the same contents as when originally read, it confirms that no updates have been performed on `lastRight`. This is true because the `Contents` objects are

```
34 int find(int key, Node*& pred, Contents*& predData, Node*& curr,
       Contents*& currData, Node* auxRoot)
35 {
36   int result, currKey;
37   Node* next, * lastRight;
38   Contents* lastRightData;

40 retry:
41   result = NOTFOUND;
42   curr = auxRoot;
43   currData = curr->contents;
44   if (ISFLAGGED(currData))
45   {
46     return ABORT;
47   }

49   next = currData->right;
50   lastRight = curr;
51   lastRightData = currData;

53   while (next != NULL)
54   {
55     pred = curr;
56     predData = currData;
57     curr = next;
58     currData = curr->contents;

60     if (ISRELOCATING(currData))
61     {
62       // Node is involved in relocation
63       Relocation* op = (Relocation*)GETREF(currData);
64       if (curr == op->destination)
65       {
66         // Node contents are being replaced
67         processRelocation(op);
68         currData = op->newContents;
69       }
70       else
71       {
72         // Node is being relocated
73         if (processRelocation(op))
74         {
75           currData = SETMARK(op->rollbackContents);
76         }
77         else
78         {
79           if (!CAS(&(curr->contents), currData, op->
                rollbackContents))
80           {
81             goto retry;
82           }
83           currData = op->rollbackContents;
84         }
85       }
```

Figure 3.12: SplitNodeBST `find` function (part 1 of 2).

```
86      }
88      if (ISMARKED(currData))
89      {
90        // Node is marked
91        currData = GETREF(currData);
92        currData = exciseNode(pred, predData, curr, currData, next)
              ;
94        if (currData != NULL)
95        {
96          // Excision was processed by this thread
97          // Step back one node and continue
98          curr = pred;
99          continue;
100       }
101       else
102       {
103         goto retry;
104       }
105     }
107     // Normal traverse
108     currKey = currData->key;
109     if (key == currKey)
110     {
111       result = FOUND;
112       break;
113     }
115     if (key < currKey)
116     {
117       next = currData->left;
118     }
119     else
120     {
121       next = currData->right;
122       lastRight = curr;
123       lastRightData = currData;
124     }
126   }
128   // check lastRight
129   if ((result != FOUND) && (lastRight->contents != lastRightData)
        )
130   {
131     goto retry;
132   }
134   return result;
135 }
```

Figure 3.13: SplitNodeBST `find` function (part 2 of 2).

```
136 bool add(int key)
137 {
138   Node* pred, * curr, * newNode;
139   Contents* predData, * currData, * newContents;

141   while (true)
142   {
143     if (!find(key, pred, predData, curr, currData, &root) ==
          NOTFOUND)
144     {
145       break;
146     }

148     newNode = new Node(new Contents(key, NULL, NULL));

150     if (key < currData->key)
151     {
152       newContents = new Contents(currData->key, newNode, currData
            ->right);
153     }
154     else
155     {
156       newContents = new Contents(currData->key, currData->left,
            newNode);
157     }

159     if (CAS(&(curr->contents), currData, newContents))
160     {
161       return true;
162     }
163   }

165   return false;
166 }
```

Figure 3.14: SplitNodeBST `add` function.

immutable and are never reused once removed from a node due to a successful update.

3.2.2.3 `add(k)`

The function for adding a key to the set is included in Figure 3.14. Once the key is verified not to be in the tree and the insertion point is found, a new split node is created with the key to be inserted and a new `Contents` object is created. The contents of the leaf node at the insertion point are copied into this new object and the appropriate child pointer is set to the new split node. A CAS is then used to replace the contents of the leaf node. If the CAS succeeds, the new node and key are now part of the tree, but if it fails it means another thread has updated the leaf node and the search for the insertion point must be restarted.

```
167 bool remove(int key)
168 {
169   Node* pred, * curr, * replace;
170   Contents* predData, * currData, * replaceData, *
        relocationContents;
171   Relocation* relocateOp;

173   while (true)
174   {
175     if (!find(key, pred, predData, curr, currData, &root)== FOUND
          )
176     {
177       break;
178     }

180     if ((currData->right == NULL) || (currData->left == NULL))
181     {
182       // Node has < 2 children
183       if (CAS(&(curr->contents), currData, SETMARK(currData)))
184       {
185         exciseNode(pred, predData, curr, currData, replace);
186         return true;
187       }
188     }
189     else
```

Figure 3.15: SplitNodeBST `remove` function (part 1 of 2).

3.2.2.4 `remove(k)`

The code for removing keys is listed in Figure 3.15 and Figure 3.16. Once the node containing `k` has been found, one of two paths is taken. The path starting line 182 is taken if the node has fewer than two children or the path starting line 191 is taken otherwise. A case in which there are fewer than two children is much simpler to deal with as the node can be simply excised from the tree. CAS is used to mark the contents pointer and at this point, the key can be considered logically removed from the set.

The `exciseNode` function (Figure 3.17) is used to perform the physical removal. This makes a copy of the parent node's contents and replaces the child pointer to the marked node with a pointer to the marked node's child, if it has one. Another CAS is then performed to attempt to replace the parent's contents with this new object. After the node has been marked, it is possible that another thread could encounter the flag while traversing (line 88 of `find`) and attempt to assist in the excision of the node. If the thread succeeds in performing the excision, it can use the return values from the `exciseNode` function to allow it to continue its traversal without restarting.

If the initial search finds the key in a node with two children, an additional search

```
190     {
191       // Node has 2 children
192       if ((find(key, pred, predData, replace, replaceData, curr)
              == ABORT) || (curr->contents != currData))
193       {
194         continue;
195       }
196       relocationContents = new Contents(replaceData->key,
              currData->left, currData->right);
197       relocateOp = new Relocation(curr, currData,
              relocationContents, replaceData);

199       if (CAS(&(replace->contents), replaceData, SETRELOCATE(
              relocateOp)))
200       {
201         if (processRelocation(relocateOp))
202         {
203           // Update predData if pred is the parent of replace
204           if (pred == curr) predData = relocationContents;
205           exciseNode(pred, predData, replace, replaceData, curr);
206           return true;
207         }
208         else
209         {
210           CAS(&(replace->contents), SETRELOCATE(relocateOp),
                  replaceData);
211         }
212       }
213     }
214   }

216   return false;
217 }
```

Figure 3.16: SplitNodeBST `remove` function (part 2 of 2).

must be done to locate the node with the next largest key. This is done by searching for the same key in `curr`'s right subtree. The result is deemed invalid if the search aborts by finding `curr`'s `contents` field has an ongoing update (line 44), in which case the entire remove operation must be restarted. A valid search will return the node with the next highest key in the variable `replace`, its contents object in `replaceData` and its predecessor data in `pred` and `predData`. If `curr` has been updated at any point during the search, it will be detected before continuing by checking `curr->contents` at the end of the search (line 192) and a restart will be triggered. This is purely an optimisation, however, because the conflict will still be detected when `curr` is CASed during `processRelocation`. If no conflicts are detected, a replacement contents object is created for `curr` and a `Relocate` object is created containing all the information required for another thread to complete the operation. A pointer to this object is

```
218 Contents* exciseNode(Node* pred, Contents* predData, Node*
      removeNode, Contents* removeData, Node*& next)
219 {
220   Node* left, * right;
221   Contents* replaceData;
222   Relocation* op;

224   next = removeData->left == NULL ? removeData->right :
        removeData->left;
225   if (predData->left == removeNode)
226   {
227     left = next;
228     right = predData->right;
229   }
230   else
231   {
232     left = predData->left;
233     right = next;
234   }
235   replaceData = new Contents(predData->key, left, right);

237   if (CAS(&(pred->contents), predData, replaceData))
238   {
239     return replaceData;
240   }
241   else
242   {
243     return NULL;
244   }
245 }
```

Figure 3.17: SplitNodeBST `exciseNode` function.

inserted into `replace`'s `contents` field with the `RELOCATE` flag to ensure that `replace` cannot be removed or added to while this operation is in progress.

The first step in `processRelocation` (Figure 3.18) uses CAS to insert the relocation operation into the node whose key is to be removed, on line 251. A successful CAS means the relocate operation cannot fail and at this point, `k` can be considered logically removed from the set[1]. Given that any thread has the potential to determine the success or failure of the operation, a mechanism is required to relay the result back to the thread conducting the removal. This is implemented in a similar fashion to an MCAS descriptor operating on a single memory location. The initial state is set to `ONGOING` until the result of the operation is known. If any thread either completes the CAS or observes the completed CAS, it will set the operation state to `SUCCESSFUL`. If any other value is seen it will attempt to CAS the state from `ONGOING` to `FAILED`. Note

[1]The VCAS (or Value CAS) operation used here is a common variant of CAS that returns the data observed at the memory location used for comparison, rather than returning success or failure. In this instance, it would be sufficient to replace VCAS with CAS followed by a read if necessary.

```
246 bool processRelocation(Relocation* op)
247 {
248   int seenState = op->state;
249   if (seenState == ONGOING)
250   {
251     Contents* seenContents = VCAS(&(op->destination->contents),
          op->oldContents, SETRELOCATE(op));
253     if ((seenContents == op->oldContents) || (seenContents ==
          SETRELOCATE(op)))
254     {
255       seenState = VCAS(&(op->state), ONGOING, SUCCESSFUL) !=
            FAILED ? SUCCESSFUL : FAILED;
256     }
257     else
258     {
259       seenState = VCAS(&(op->state), ONGOING, FAILED) !=
            SUCCESSFUL ? FAILED : SUCCESSFUL;
260     }
261   }
263   if ((seenState == SUCCESSFUL) && (op->destination->contents ==
        SETRELOCATE(op)))
264   {
265     CAS(&(op->destination->contents), SETRELOCATE(op), op->
          newContents);
266   }
268   return seenState == SUCCESSFUL;
269 }
```

Figure 3.18: SplitNodeBST `processRelocation` function.

the use of VCAS in line 259 to return the actual value of `state` even if the thread has determined the operation has failed (because, if the operation has succeeded, the node may have had further operations applied to it since success was determined). If successful, a CAS is performed on the `contents` field to update it to the new object and remove the pointer to the ongoing relocation operation.

If the relocation was successful, `exciseNode` is called to remove the node that was relocated, as if it had been marked. Alternatively, if the operation failed, the changes to `replace` must be rolled back by restoring its contents to the original object reference. A concurrently traversing thread, executing `find` that observes an ongoing relocation operation will attempt to process the operation. In the case that the operation is observed in the node whose contents are being replaced (determined by examining the contents of the descriptor), the operation has already succeeded so after `processRelocation` returns, the `currData` search parameter can be updated and the search can continue. If the node whose key is to be relocated is observed instead, the

operation is still ongoing and has a chance of failure. In the case that the operation has failed, the `rollbackContents` field is accessed in order to restore the node to its original state. But, if the operation has succeeded, the `MARK` flag is set in the local search variable `currData` so that the excision code will be executed once the thread reaches line 88.

The act of writing a pointer to a `Relocate` object into `replace` but then restoring its old `Contents` object if the operation is rolled back can be seen as an instance of the ABA problem. A thread which read the contents of `replace` and did not observe the attemped relocation operation would assume that no writes were performed on `replace` if it was read after the rollback occurred. In this instance, however, this can be safely ignored because by restoring the old `Contents` object, the state of the node is identical to the state it was in before the concurrent `remove` operation was attempted.

3.2.3 Memory management

This algorithm is compatible with the memory management systems described in the previous chapter. When CAS operations are performed which replace or remove objects from the tree, the thread which executed the successful CAS can add the object to a retire list. If a marked node is removed, the thread which executed the CAS to replace the parent's contents is responsible for retiring the old contents, and both objects comprising the removed node. In the case of a relocation, the final pointer to the operation descriptor to be removed is that contained in the node whose key was relocated. Once that node is excised, the descriptor, the contents containing the removed key, and the `rollbackContents` can be retired by the thread completing the excision. In the cases where a thread fails its initial attempt to start an operation, it can retire or reuse these objects on its next attempt. If the initial CAS inserting a `Relocate` object into a node succeeds, but the second CAS cannot, the thread which successfully performs the rollback can retire the descriptor and the replacement contents object contained within. This is because those objects are globally visible and can be read by other threads until the `Relocate` object is removed from the tree.

3.2.4 Correctness

3.2.4.1 Non-blocking

The lock-free property is proved informally by examining each situation that can cause a thread to process its operation in an unbounded number of steps. Traversal of the tree is performed as part of all operation types via the `find` function. It is at least possible that a thread could continue to search indefinitely in the case that other threads continuously add new nodes along its search path, but this is allowed for by the lock-free property because these other threads are performing successful insertions. There are also places where condition checks are performed during the traversal that can trigger a restart from the root. The final check which verifies that the subtree key range has not contracted forces a retry if `lastRight`'s contents were changed. There are a small number of reasons that can cause this to occur: an insertion giving the node a new child, an excision of one of the node's children, the node itself becoming marked, or the node being involved in a relocation. All but the update involving relocation directly imply a successful operation has taken place on another thread, meaning system-wide throughput has been achieved. If a `RELOCATE` flag is seen, and the node is the destination of the key relocation, the operation has already succeeded, meaning progress has been made but if the node in question is having its key relocated, there is a possibility the operation could be rolled back. The other two retry points in `find` are triggered when helping to complete other operations, one when a thread attempts to assist with an excision and fails (which can only happen after a success remove performed by another thread) and the other when a thread fails in its attempt to roll back a relocation (again meaning another thread completed the roll back).

If a roll back must occur, it can still be reasoned that the system as a whole has made progress. When the node containing the key to be removed was read, it did not have any ongoing operation, but the failure of the CAS to replace its contents with the `Relocation` descriptor means that it has been updated. It is known that the node has two children, so only two possible updates could have occurred: one of its children was excised or another thread removed the same key. Both of these cases imply the progress of other threads.

An add operation will fail if the contents of the node receiving a new child were modified since being read. This node had fewer than two children before it was updated meaning it could have gained a new child via a concurrent insertion, it could

have become marked or it could have had a child excised – all indicating a successful operation by another thread. It is also possible that its key may have been relocated and, regardless of success or failure, it has previously been proven that it implies the system as a whole has made progress.

3.2.4.2 Linearisability

The contains operation directly uses the results of `find` therefore its linearisation points for determining if a key is or is not in the set occur there. Successfully finding a key linearises at the point where the contents object of the node with that key is read (line 58) because the key value of the object can never be modified. It will linearise here if the value read has the `CONTENTS` flag and the contents object contains the key being searched for. The linearisation point for failing to find a key occurs when the next path of the traverse is read and it happens to be null (line 117 or 121), but it will only successfully linearise once it verifies that the subtree being searched cannot have contracted by checking the contents of `lastRight` (line 129). There is also a special case when the tree is empty for which failure to find will linearise at line 49 by reading a null value.

An unsuccessful `add` will linearise at the same point as a successful `contains` because the key already exists in the set. If the key is not a member of the set, in order for an add to be successful, it must apply its update to the tree by CASing a new contents object into the leaf node at the insertion point. Any concurrent searches that read the contents field before the linearisation point would have concluded that the key was not in the set, whereas searches that read the contents after the linearisation point will see the new contents object.

For a remove operation that fails to find the key it wishes to remove from the set, the linearisation point will be the same as for an unsuccessful contains. If the node containing the key to be removed has fewer than two children, the linearization point for successfully removing will occur when the `MARK` flag is set inside the node. From that point on, any other threads that read the contents of the node will see the flag and attempt excision. The removal of a node with two children takes effect once the `Relocation` object is CASed into both nodes involved (because the operation cannot be rolled back at that point). Because the second CAS (line 251) can be executed by any thread encountering the ongoing operation, it is possible that the successful remove can be linearised by a thread other than the one which created the operation.

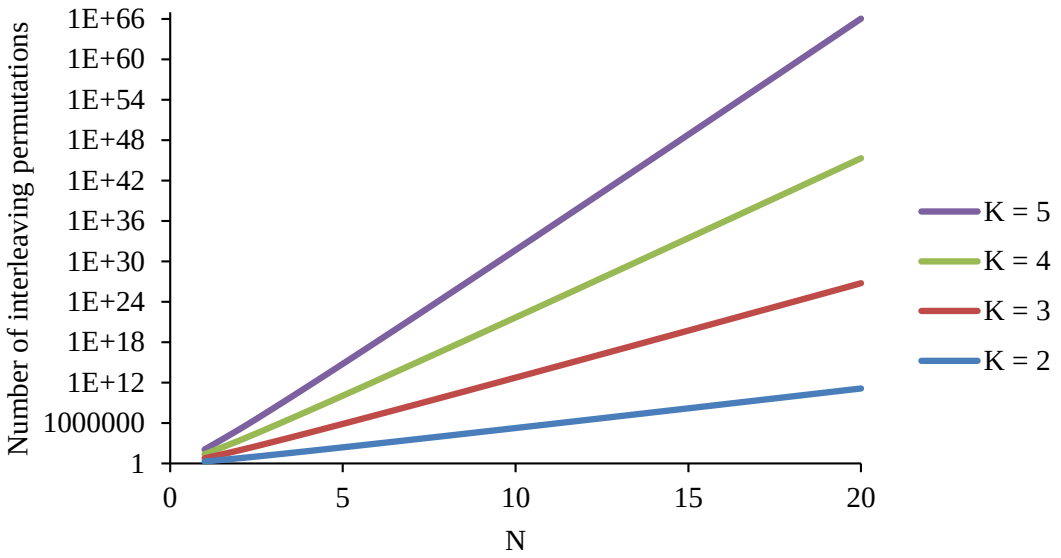

Figure 3.19: Graph showing the number of possible permutations of instruction interleavings for K threads executing N instructions for varying N and K.

3.2.4.3 Model checking

The correctness and linearisation points were proved by creating a model of the tree algorithm in the SPIN model checker [SPIN]. SPIN can be used to verify user-defined correctness conditions in a multi-threaded model by simulating every possible interleaving of the constituent threads' operations. Simulations quickly become unfeasible to run, however, as the number of concurrent threads and complexity of the threads increase. If each of K threads will finish in a finite number of steps N, the total number of combinations of interleavings of the steps increases exponentially in the number of steps and factorially in the number of threads as given by the equation (visualised in Figure 3.19):

$$\frac{(NK)!}{(N!)^K}$$

At each step in a simulation, SPIN calculates a state descriptor and compares it with already seen states in order to eliminate portions of the state space which it has already simulated. This helps to reduce the number of states as calculated above, but generally only enough to allow marginal increases to N and K. SPIN will pre-merge pairs of states that it determines have no dependency on ordering, but this was manually performed in order to further reduce N. This was done by merging states

which only operated on local variables with up to one state that reads or writes global memory (variables).

Due to the limitations on the size of a simulation in SPIN, the correctness of the algorithm could not be proved exhaustively but a small subset of cases that should result in a variation of every concurrent interaction were chosen. Because there is no state brought forward between a thread executing an (add, remove or contains) operation and the next, each thread was set to execute a single operation. Sequences where differing keys resulted in the same set of interleavings were eliminated as were sequences where particular operations could not possibly succeed and modify the data structure. Initial states and combinations of operations were pre-generated for individual SPIN verification runs rather than allowing SPIN to generate them in its own state space in order to allow many simulations to be run in parallel. The linearisation points were proved by using a bitmap to represent key membership in the set and verifying and/or updating the bitmap at the linearisation points.

SPIN requires that models are described in the high-level modelling language PROMELA in which each line of code represents a single state. The modelling code used to test the split-node tree is listed in Appendix A.

3.3 Single-node binary search tree[2]

3.3.1 High-level overview

The performance cost of having to access two disjoint memory locations when traversing a single node in the split-node tree design motivated the pursuit of a design in which each node could be composed as a single object, as in the sequential design. This necessitated abandoning the use of immutable data fields inside nodes because node data would have to be modified in-place. The split-node algorithm ensured that no concurrent updates had been applied to a node if its `contents` pointer had not changed between successive reads, but it would be unfeasible to read all the fields in a node atomically to check for updates. Furthermore, the ability to block concurrent updates to different fields in the node by serialising them via a CAS operation performed on the `contents` pointer is another property lost by moving to a single object per key in the BST.

[2] A version of this algorithm was presented at SPAA 2012 [Howley and Jones 2012].

The first step to overcoming these issues was to add an extra field to the unified node object through which all updates could be serialised. All update operations are required to acquire the right to perform an update by writing to this field, much in the same way as the split-node tree's relocation and marking operations writing to the `contents` field. Operations requiring further steps must maintain the lock-free property by providing enough information for other threads which might encounter the ongoing operation to complete it. Threads can compare the value seen in this operation field at different times and so determine if any updates have been applied to the node. This fundamental change to the node structure changed how each of the update operations could be implemented.

An insert is performed by copying all the data needed for another thread to complete the operation and writing it into a structure, then copying a pointer to the structure into the operation field of the node gaining the new child (Figure 3.20). Once the operation field has been updated, the operation cannot fail and therefore the key can be considered logically inserted at this point. The operation field will only be modified with CAS and can only have operations placed into it if it is clear. Although the physical insertion is performed using a single CAS on the child pointer, the two-stage insertion in this algorithm is required in order to serialize updates to the node, via the aforementioned operation field, ensuring that contending operations cannot interfere with each other.

Removing a node with fewer than two children is similarly straightforward. A key is logically removed by marking the operation field of the node containing the key. Once marked, the removal cannot be undone. An operation similar to insertion can then be applied to the parent of this node, completing the physical removal (Figure 3.21). Removing a node with two children is handled in a very similar way to the split-node tree: by creating a descriptor containing enough information for another thread to complete the key relocation and inserting it into the operation field of each node involved.

The operations for the single-node tree require more CAS instructions to be executed than the similar operations of the split-node tree. For example, the operation to add a new key could be performed with a single CAS (swapping a new `Contents` object into a node) but now requires three: one to write a pointer to the operation descriptor into the operation field of the node, one to update the child pointer of the node, and a final CAS to clear the pointer in the operation field. These extra CAS instructions should not cause a significant overhead when compared to the previous

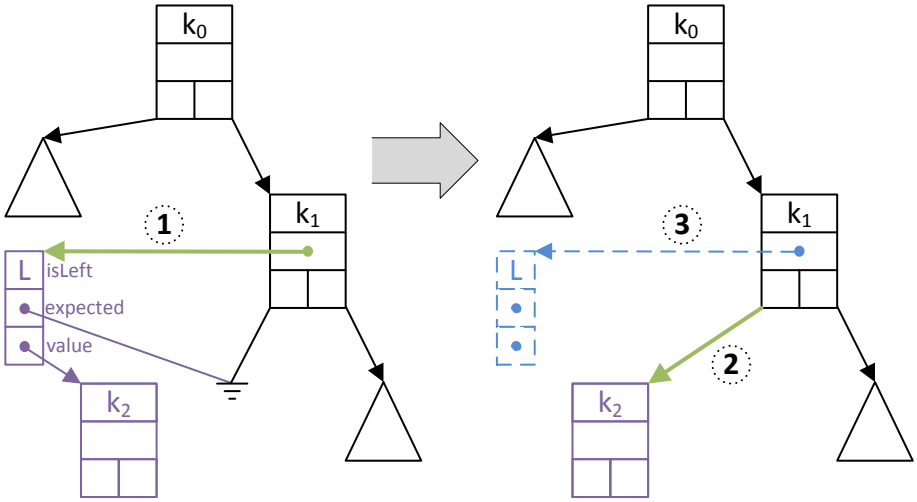

Figure 3.20: Single-node BST adding the key k_2 to the left of node with key k_1. The first CAS writes a pointer to the `ChildCASOp` structure into the `op` field of the node containing k_1. The second CAS adds the new node to the left child, and the final CAS clears the operation from the `op` field.

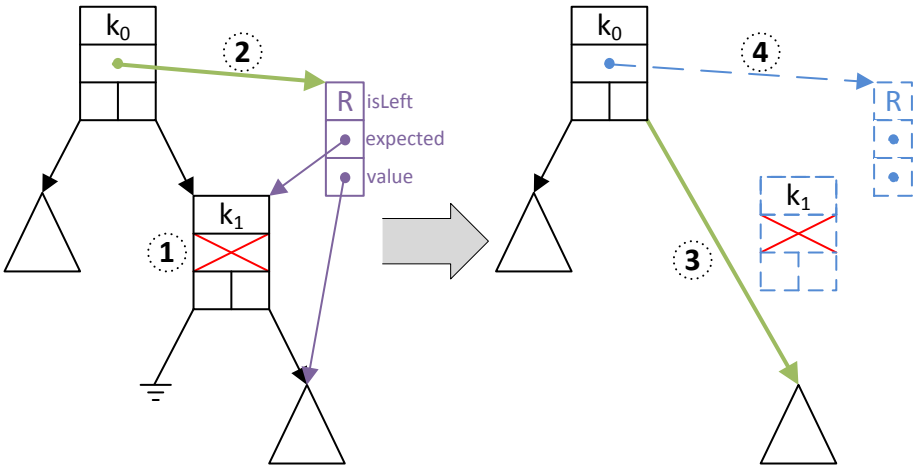

Figure 3.21: Excising the node with key k_1 from the tree. The red X denotes the node has been marked. The first CAS puts a pointer to the `ChildCASOp` structure into the parent node's `op` field. The second CAS replaces the right child pointer with a pointer to the removed node's only child, and the final CAS clears the operation from the `op` field.

```
1  class Node
2  {
3    int volatile         key;
4    Operation* volatile  op;
5    Node* volatile       left;
6    Node* volatile       right;
7  }

9  class Operation {}

11 class ChildCASOp : Operation
12 {
13   bool   isLeft;
14   Node* expected;
15   Node* update;
16 }

18 class RelocateOp : Operation
19 {
20   int volatile  state = ONGOING;
21   Node*         dest;
22   Operation*    destOp;
23   int           removeKey;
24   int           replaceKey;
25 }
```

Figure 3.22: Single-node BST class declarations.

single CAS, however, because they will all operate on the same cache line and they are executed in quick succession. The first CAS will cost the most because it will bring the cache line into the L1 cache (if not already) in a dirty or exclusive state. This means that further uncontended CAS operations will not cause any cache coherency traffic, and will complete relatively quickly.

3.3.2 Detailed implementation

3.3.2.1 Structures

The declarations of the classes used in this implementation are listed in Figure 3.22. As with the split-node tree, the tree class just contains a sentinel root element initialised with a dummy key, included to simplify the traversal code. The insertion or removal of the logical root will always occur at the right child of this sentinel node.

The node class closely resembles that of a sequential BST but with the addition of the `op` field which is used to signify if changes are being made which affect the node. The field is used to store a pointer to a descriptor containing information about an ongoing update affecting this node. Four flag values are also encoded into this field in

unused pointer bits, giving information about the state of the node. The four different states of change a node can be in are:

- `NONE` – No ongoing operation.
- `MARK` – The node's key is no longer in the set and the node should be physically removed.
- `CHILDCAS` – One of the node's child pointers is being modified.
- `RELOCATE` – The node is one of two affected by a key relocation.

The following macros are also defined to aid in modifying this data:

- `FLAG(ptr, state)` – Sets the state of the Operation pointer `ptr` to one of the above states.
- `GETFLAG(ptr)` – Strips away the pointer data to leave just the state information.
- `UNFLAG(ptr)` – Strips away the state bits leaving the pointer intact.

The left and right child pointers use special values to denote a null pointer. This is done to avoid the ABA problem occurring when a child pointer is initially null, is then set to point to a valid node, that node is removed and the pointer is set back to null again. The mechanism that is used to store null values retains the pointer data, but sets an unused low order bit to signify if the pointer is null. This guarantees that every null pointer is unique for that field. The macros `SETNULL(ptr)` and `ISNULL(ptr)` are used to set the null bit of a node pointer and to check if a pointer is null respectively. A node is initialized with null left and right pointers. All the fields of a node are mutable but `key`, `right`, and `left` can only be modified if the appropriate operation has been written to the `op` field first.

A `ChildCASOp` object contains enough information for another thread to complete an operation which modifies one of a node's child pointers. The following information is required: whether the change is to be made to the left or right child pointer, the expected data at that location, and the updated value it should be changed to. There is no need to store the destination node in the structure because the destination is the same node where the pointer to the operation object is found. An active `ChildCASOp` operation is flagged with the `CHILDCAS` state in a node's `op` field.

A `RelocateOp` object contains enough information for another thread to attempt to complete an operation to remove the key of a node with two children and replace

```
26 bool contains(int key)
27 {
28   Node* pred, * curr;
29   Operation* predOp, * currOp;

31   return find(key, pred, predOp, curr, currOp, &root) == FOUND;
32 }
```

Figure 3.23: Single-node BST `contains` function.

it with the next largest key. The following information is required: a pointer to the node whose key is to be removed, the data stored in that node's `op` field, the key being removed, and the key to attempt replacement with. The destination must be stored in this descriptor because a pointer to the structure is stored in the `op` field of both the node whose key would be removed and the node whose key will be relocated and it is necessary to be able to distinguish between them. An active `RelocateOp` operation is flagged with the `RELOCATE` state in a node's `op` field. The `state` field is used to store the status of the operation, serving the same purpose as it did with the split-node tree algorithm.

3.3.2.2 contains(k)

The `contains` function (Figure 3.23) comprises of a single call to `find` (Figure 3.24 and Figure 3.25) which does the actual work of determining if a key is contained in the subtree of the supplied node parameter, `auxRoot`. Alternatively, if a node containing the key is not found, the position where it would have been expected to be is returned. In either case, the position is returned in the variable `curr` and its parent in `pred`. The values read in their `op` fields are returned in `currOp` and `predOp`. The result of the returned search is encoded as one of four values:

- `FOUND` – If `k` is a member of the set.
- `NOTFOUND_L` – If `k` is not in the set but would be positioned at the left child of `curr` if it were inserted.
- `NOTFOUND_R` – Similar to `NOTFOUND_L` but for the right child.
- `ABORT` – If the search of a subtree cannot return a usable result (again, this can only occur when beginning a search from a position other than the root).

```
33 int find(int key, Node*& pred, Operation*& predOp, Node*& curr,
       Operation*& currOp, Node* auxRoot)
34 {
35   int result, currKey;
36   Node* next, * lastRight;
37   Operation* lastRightOp;
38 retry:
39   result = NOTFOUND_R;
40   curr = auxRoot;
41   currOp = curr->op;
42   if (GETFLAG(currOp) != NONE)
43   {
44     if (auxRoot == &root)
45     {
46       helpChildCAS((ChildCASOp*)UNFLAG(currOp), curr);
47       goto retry;
48     }
49     return ABORT;
50   }

52   next = curr->right;
53   lastRight = curr;
54   lastRightOp = currOp;

56   while (!ISNULL(next))
57   {
58     pred = curr;
59     predOp = currOp;
60     curr = next;
61     currOp = curr->op;

63     if (GETFLAG(currOp) != NONE)
64     {
65       // Help to complete an observed ongoing operation
66       help(pred, predOp, curr, currOp);
67       goto retry;
68     }

70     currKey = curr->key;
71     if (key < currKey)
72     {
73       result = NOTFOUND_L;
74       next = curr->left;
75     }
76     else if (key > currKey)
77     {
78       result = NOTFOUND_R;
79       next = curr->right;
80       lastRight = curr;
81       lastRightOp = currOp;
82     }
83     else
84     {
85       result = FOUND;
86       break;
87     }
88   }
```

Figure 3.24: Single-node BST `find` function (part 1 of 2).

```
89    if ((result != FOUND) && (lastRightOp != lastRight->op))
90    {
91      goto retry;
92    }
93    if (curr->op != currOp)
94    {
95      goto retry;
96    }

98    return result;
99  }
```

Figure 3.25: Single-node BST `find` function (part 2 of 2).

Before the traversal loop begins, the search variables `curr`, `currOp` and `next` are initialised. `lastRight` and `lastRightOp` are used to keep a record of the last node (and its `op`) for which the right child path was taken. The check at lines 42-44 is to catch the special case when the root has an ongoing operation to either add to the empty tree or remove the logical root.

The search loop continues traversing nodes one at a time until either the key is found or a null node is reached. The check at line 63 represents a simplified version of the code which attempts to help complete any ongoing operations observed during the traverse and then restart the search. This is presented here purely as a means to reduce the code complexity of the core algorithm and the number of special cases which must be dealt with. An optimised version of `find` has the ability to traverse over ongoing operations that do not impact on the results of the search (outlined in section 3.3.2.5). When the search loop has completed, the key range in the current subtree is validated by verifying that no updates have occurred to `lastRight` on line 89. The final check at line 93 is to ensure that `curr` had not been modified between reading its `op` field and reading its key.

3.3.2.3 add(k)

The functions required for adding a key to the set are listed in Figure 3.26. If the key is not found in the tree, the correct insertion point is returned and a new node and `ChildCASOp` object are created with the information necessary to perform the insertion. CAS is used to insert the operation into `curr`'s `op` field at which point `k` can be considered logically inserted. The CAS succeeding means that `curr`'s `op` field has not been modified since it was first read. This in turn means that all of `curr`'s other

```
100 bool add(int key)
101 {
102   Node* pred, * curr, * newNode;
103   Operation* predOp, * currOp, * casOp;
104   int result;
105
106   while (true)
107   {
108     result = find(key, pred, predOp, curr, currOp, &root);
109     if (result == FOUND)
110     {
111       return false;
112     }
113
114     newNode = new Node(key);
115     bool isLeft = (result == NOTFOUND_L);
116     Node* old = isLeft ? curr->left : curr->right;
117     casOp = new ChildCASOp(isLeft, old, newNode);
118     if ( CAS(&(curr->op), currOp, FLAG(casOp, CHILDCAS)))
119     {
120       helpChildCAS(casOp, curr);
121       return true;
122     }
123   }
124 }
125
126 void helpChildCAS(ChildCASOp* op, Node* dest)
127 {
128   Node** address = op->isLeft ? &(dest->left) : &(dest->right);
129   CAS(address, op->expect, op->update);
130   CAS(&dest->op, FLAG(op, CASCHILD), FLAG(op, NONE));
131 }
```

Figure 3.26: Single Node BST `add` and `helpChildCAS` helper functions.

fields are also unchanged, and so the operation to CAS one of `curr`'s child pointers cannot fail to complete – whether performed by the thread which created the operation or another. The `helpChildCAS` function does the main work of physically adding the new node to the tree and cleaning up after the operation. This can be called by any thread that encounters the ongoing operation in `curr`'s `op` field. Executing either of the two CAS's performed by `helpChildCAS` at any time after the operation has been completed is safe as they will be guaranteed to fail at that point. This is because `curr`'s child pointer and `op` pointer can never return to the values expected by the CAS.

3.3.2.4 remove(k)

The code for `remove` is listed in Figure 3.27 and its associated helper functions in Figure 3.28 and Figure 3.29. If `k` is found in the tree, the removal is handled differently

```
132 bool remove(int key)
133 {
134   Node* pred, * curr, * replace;
135   Operation* predOp, * currOp, * replaceOp, * relocOp;

137   while (true)
138   {
139     if (find(key, pred, predOp, curr, currOp, &root) != FOUND)
140     {
141       return false;
142     }

144     if (ISNULL(curr->right) || ISNULL(curr->left))
145     {
146       // Node has < 2 children
147       if (CAS(&curr->op, currOp, FLAG(currOp, MARK)))
148       {
149         helpMarked(pred, predOp, curr);
150         return true;
151       }
152     }
153     else
154     {
155       // Node has 2 children
156       if ((find(key, pred, predOp, replace, replaceOp, curr) ==
             ABORT) || (curr->op != currOp))
157       {
158         continue;
159       }

161       relocOp = new RelocateOp(curr, currOp, key, replace->key);
162       if (CAS(&(replace->op), replaceOp, FLAG(relocOp, RELOCATE))
             )
163       {
164         if (helpRelocate(relocOp, pred, predOp, replace))
165         {
166           return true;
167         }
168       }
169     }
170   }
171 }
```

Figure 3.27: Single-node BST `remove` function.

```
172 void helpMarked(Node* pred, Operation* predOp, Node* curr)
173 {
174   Node* newRef;
175   if (ISNULL(curr->left))
176   {
177     if (ISNULL(curr->right))
178     {
179       newRef = SETNULL(curr);
180     }
181     else
182     {
183       newRef = curr->right;
184     }
185   }
186   else
187   {
188     newRef = curr->left;
189   }

191   Operation* casOp = new ChildCASOp(curr == pred->left, curr,
        newRef);
192   if (CAS(&(pred->op), predOp, FLAG(casOp, CHILDCAS)))
193   {
194     helpChildCAS(casOp, pred);
195   }
196 }

198 void help(Node* pred, Operation* predOp, Node* curr, Operation*
      currOp)
199 {
200   switch (GETFLAG(currOp))
201   {
202   case CHILDCAS:
203     {
204       helpChildCAS((ChildCASOp*)UNFLAG(currOp), curr);
205       break;
206     }
207   case RELOCATE:
208     {
209       helpRelocate((RelocateOp*)UNFLAG(currOp), pred, predOp,
            curr);
210       break;
211     }

213   case MARK:
214     {
215       helpMarked(pred, predOp, curr);
216       break;
217     }

219   case NONE:
220   default:
221     break;
222   }
223 }
```

Figure 3.28: Single-node BST `helpMarked` and general `help` functions.

depending on the number of children the node has. The path starting line 146 is taken if the node has fewer than two children. CAS is used to change `curr`'s operation state from `NONE` to `MARK` at which point `k` can be considered logically removed from the set. `helpMarked` is used to perform the physical removal. It uses a `ChildCASOp` operation to replace `pred`'s child pointer to `curr` with either a null pointer if `curr` is a leaf node, or a pointer to `curr`'s only child. Note that `predOp` may have changed since it was read inside `find`, and so removing the marked node may fail. If it is required that a marked node is definitely excised before the marking operation returns, another call to `find(k)` can be made if the CAS at line 192 fails – which will ensure removal via the operation helping mechanism.

If the node containing `k` is found to have two children, the code path starting line 155 is taken instead. The next largest key is located by making an additional call to `find` starting in the node's right subtree. The result is deemed invalid if the search finds `curr`'s `op` field has been modified since the first search and aborts (line 49). The entire operation is then restarted. If the search succeeds, a `RelocateOp` object is created to attempt to replace `curr`'s key with the key of the node returned, `replace`. This operation is inserted into `replace`'s `op` field to ensure that `replace`'s key cannot be removed while this operation is in progress.

The first step in `helpRelocate` is the CAS to insert the `RelocateOp` into the `op` field of the node whose key is to be removed (line 229). If this CAS succeeds, it means the relocate operation cannot fail and now `k` can be considered logically removed from the set. The operation state is updated in order to relay the result back to the thread which created the operation, in the case that it is not the thread that determined the success or failure of the operation. As with the split-node tree, the initial state is set to `ONGOING` until the result of the operation is known. The thread that executes the CAS or any other thread that observed the operation field as containing a pointer to this ongoing operation will determine that the operation has succeeded and attempt to update the state to `SUCCESSFUL`. If any other value is observed by a thread, it will attempt to CAS the state from `ONGOING` to `FAILED`. If the operation has succeeded, a CAS is performed on the key to update it to the new value and a second CAS is used to remove the ongoing `RelocateOp` from the node. Note that the CAS to replace the key cannot suffer from the ABA problem because it can only possibly be replaced with a larger key, otherwise it is possible the key could return to a previously used value.

The second part of `helpRelocate` (from line 247 onward) performs cleanup on `replace` by either marking and excising it if the relocation was successful, or clearing

```
224 bool helpRelocate(RelocateOp* op, Node* pred, Operation* predOp,
      Node* curr)
225 {
226   int seenState = op->state;
227   if (seenState == ONGOING)
228   {
229     Operation* seenOp = VCAS(&(op->dest->op), op->destOp, FLAG(op
          , RELOCATE));
230     if ((seenOp == op->destOp) || (seenOp == FLAG(op, RELOCATE)))
231     {
232       CAS(&(op->state), ONGOING, SUCCESSFUL);
233       seenState = SUCCESSFUL;
234     }
235     else
236     {
237       seenState = VCAS(&(op->state), ONGOING, FAILED);
238     }
239   }

241   if (seenState == SUCCESSFUL)
242   {
243     CAS(&(op->dest->key), removeKey, replaceKey);
244     CAS(&(op->dest->op), FLAG(op, RELOCATE), FLAG(op, NONE));
245   }

247   bool result = (seenState == SUCCESSFUL);

249   if (op->dest != curr)
250   {
251     CAS(&(curr->op), FLAG(op, RELOCATE), FLAG(op, result ? MARK :
          NONE));
252     if (result)
253     {
254       if (op->dest == pred)
255       {
256         predOp = FLAG(op, NONE);
257       }
258       helpMarked(pred, predOp, curr);
259     }
260   }

262   return result;
263 }
```

Figure 3.29: Single-node BST `helpRelocate` function.

its `op` field if it failed. The check in line 249 is to ensure that `curr` is the node which potentially had its key relocated and not the node whose key was to be removed, which can be the case if `helpRelocate` is called from `find`. If the operation was successful and `curr` is marked, `curr` is excised from the list using `helpMarked` as above. The code in line 256 takes into account the special case where the node whose key was removed is the parent of `curr` and so, `predOp` must be updated to the value it was changed to in line 244.

3.3.2.5 Optimising traversal

The optimised version of `find` ignores ongoing operations during its search, but still must adhere to the requirement that it return a node without an operation attached to it. This is done by performing a `help` and retry only if the node found at the end of a traverse contains an active operation. Because `help` requires that `pred` has no ongoing operation, it is necessary for progress to back-track to the last node seen during the traversal which had no operation attached and perform the `help` from that point in the tree. This back-track point will have been explicitly stored during the traversal. It is at least possible that during a traversal, chains of marked nodes could be traversed. Therefore, depending on how the memory management system handles physically removed nodes, it may be necessary to verify physical deletion has not yet occurred at each traversal step over logically deleted nodes. This can be done safely because the physical removal of a chain of marked nodes only happens at the highest unmarked node in the tree, which can be monitored for updates. The optimisations violate the guarantee that when `find` returns, `pred` was observed without an operation and while this would affect a remove operation excising a node, it does not stop the `remove` from marking `curr` and thus successfully linearising. If there is a requirement to not leave marked nodes in the tree, a subsequent call to `find` can be made. The `contains` function can be made completely read-only by using a specialised version of the optimised `find`. If an ongoing operation is detected at the end point of the traversal, instead of helping to update the tree it is possible to determine if the searched key is in the set by examining the state and contents of the observed operation. These optimised implementations are listed in Appendix C.

A search which passes a single marked node or a `ChildCASOp` is naturally safe because these types of ongoing operations cannot affect the validity of a search result. Furthermore, care must be taken when passing multiple contiguous marked nodes, as

mentioned in the previous paragraph. What is not immediately clear is the correctness of ignoring a key relocation. In the case where a `RelocateOp` has been observed and passed before the key change has occurred, the search is only affected if this was the last node for which the right child path was taken. A problem would exist if the node which was used for replacement was removed before any changes to `lastRightOp` were made, but `helpRelocate` guarantees that this can never be the case by ensuring `lastRight`'s operation is cleared before this node is marked. This would then be detected at the end of a search, triggering a restart.

3.3.3 Memory management

The algorithm avoids the ABA problem occurring at the child or operation fields by using bit flags (`NULL` flag for children or `NONE` flag for operations) to essentially create unique null pointers (as also used in the external BST of Ellen et al. [2010]). In order to ensure the uniqueness of these pointers and avoid the ABA problem, the memory which they reference cannot be reused until the null pointer has been replaced. The consequences of this is that the memory usage of the algorithm increases as the majority of nodes will contain a unique null operation pointer and leaf nodes at the edge of the tree can contain up to two unique null child pointers. Once the unique null is removed (by adding a child node or updating the `op` field), the reuse of these objects could be handled by a garbage collector or a memory reclamation system such as hazard pointers.

This situation is not ideal as it would be useful to be able to reuse or reclaim these defunct objects as soon as they are removed from the logical tree. If the system supports DWCAS, the problem can be solved by adding tags to the `op` field and child pointers (at the expense of object size and cache density). If the tag is incremented each time the pointer is modified, it ensures that the ABA problem cannot occur even if the pointers being DWCASed are the same. Because DWCAS does not have wide support across modern architectures, a solution that uses single word CAS would be more desirable. A solution to this problem is proposed in section 3.4.

3.3.4 Correctness

3.3.4.1 Non-blocking

The lock-free property of the algorithm is proved by describing the interactions that can occur between reading and writing threads. A thread traversing the tree will eventually either locate the key it is searching for or stop on a leaf node, assuming nodes are not repeatedly added in its path ad infinitum. A retry can be triggered in the following cases: if an ongoing operation is encountered, `help` is called in an attempt to complete it, if `lastRight` has been updated since its `op` field was read, or if `curr` was updated since its `op` field was read. In each of these cases, the `find` has been restarted due to a node on the search path being updated. If these `op` fields contain a `CHILDCAS` or `MARK` flag, this means another thread has applied an update to the tree and therefore system-wide throughput was achieved. For a `RELOCATE` flag, if the node encountered is the node being replaced, this implies a successful remove operation. Otherwise, there is potential for the relocate operation to be rolled back which means the thread performing the removal did not achieve progress. Finally, if the modified `op` field has the `NONE` flag, a successful `ChildCAS`, relocation or rollback relocation must have occurred.

A relocation being forced to roll back still implies the progress of another thread, as follows: the `find`s which are part of a remove are governed by the same properties as previously mentioned, but the second `find` can abort and trigger a retry at line 49 if the node to be removed is seen to have an ongoing operation. The change can also be detected by a failure of the CAS which installs the `RelocateOp` link into the node whose key is to be removed (line 229). Because this node has two children, this can only either be a `ChildCAS`, if one of the child nodes is being excised, or a `RelocateOp` if another thread is removing the same key. In either case, another thread has successfully performed an operation on the node.

An add is always performed on a leaf node so if a valid insertion point is returned by `find`, a retry will only occur if the leaf's `op` field has been modified. This can only occur in the case that: another thread has successfully added a new child to the node, another thread has marked the node, or another thread is using the node as a replacement for a removal higher up in the tree. In each of these cases it has been previously demonstrated that system-wide progress was achieved.

3.3.4.2 Linearisability

The linearisability of the algorithm is proved by defining the linearisation points of the add, remove and contains operations. The contains operation has two possible outcomes: that the key was found in the tree or not. The linearization point for finding a key is the point at which an atomic read of the key has occurred at line 70. However, this result will only be linearised providing the check at line 93 passes. This is done to verify that the node did not change between its `op` field being read and the linearisation point. Failure to `find` will either linearise when the end of a search path is reached by reading a null link at line 74 or line 79. Again, this result is subject to checks to verify the result. The `lastRight` node must not have been concurrently modified or the key range of its right subtree may have changed during the search – this is confirmed in line 89. Node `curr` must also be verified at this point to ensure the node did not undergo a change (such as adding a `ChildCASOp` operation) between reading its `op` field and the linearisation point. In the special case that the tree is empty, the `contains` operation linearises at line 52 when `next` is assigned a null value.

The optimised version of the `find` function has the same linearisation points as the simplified version – providing that, at the end of the search, `curr` was observed to have no ongoing operation. Most of the cases for the read-only `contains` are similar to this with the following exceptions: if the traversal ends on a marked node, the failure to find linearises when the marked node has been verified not to have been excised, again subject to verification by checking `lastRight` has not been modified; if the key is found in a node being used for relocation, a successful search will linearise if the key exchange is verified not to have completed yet; if the key is not found but a `ChildCASOp` is seen, the search will linearise once the key of the node to be added is read and subject to a subsequent verification that the `ChildCASOp` is still ongoing.

An insert fails if it is determined that the key is already contained in the set and therefore has the same linearization point as a successful contains. A successful insertion occurs when the `ChildCASOp` operation is inserted into the node `curr`, returned by `find`. This is true as any other thread that were to search for `k` at this point would encounter the `CHILDCAS` flag in `curr`'s `op` field and must either help complete the operation or, in the case of read-only searches, must conclude that the key is in the set based on the `ChildCASOp` contents.

A remove will fail, similarly to insert, if `find` determines that the key is not in the set and, as before, has the same linearization point as an unsuccessful `contains`. The

removal of a node with fewer than two children will linearise when the node is set as marked in line 147. Removing a node with two children requires two CASs to succeed: one on the node which has the next largest key, and one on the node containing the key to be removed. A failure of either CAS will cause changes to be rolled back and the creating thread will retry its search. A successful remove can be linearised by a thread other than the one which created the operation as it occurs once the `RelocateOp` is installed in the second node which is performed in the `helpRelocate` function, line 229.

3.3.4.3 Model checking

A model of the single-node tree was verified in much the same manner as the split-node tree using SPIN. For reference, the PROMELA language code is listed in Appendix B.

3.4 Unique null pointers for descriptor based designs

3.4.1 Motivation

The nodes in the single-node BST design have multiple fields that are susceptible to the ABA problem: `op`, `left` and `right`. The correctness of the algorithm relies on having a consistent view of a node before updates are applied. The operation field is always cached in a local variable before any other fields are read and, by rereading the field at any later point and confirming it has not changed, it proves that the node has not undergone any updates. The reading of the fields of the object can then essentially be considered atomic.

In the current design, when an operation completes and wishes to remove itself from visibility in the node it operated on, it sets a flag in the `op` field but leaves its pointer data intact. In an ideal situation, this field would simply be set to null to signify no ongoing operation but doing this would make the algorithm susceptible to the ABA problem. A node to be removed could be observed to only have one child, then another thread could add a second with a `ChildCASOp` and set the `op` field back to null on completion. The thread that wishes to remove the node could then successfully (and incorrectly) CAS a `MARK` flag into the node. The same ABA effect can occur in the child pointers when multiple threads are helping to add a new node into the tree by CASing away a null pointer.

By leaving the pointer data intact, this ABA issue could only occur if the same pointer value is reused in the field, but by guaranteeing that it will not be reused, it can be considered a unique null pointer. This complicates the use of traditional memory management algorithms because there are now a set of memory locations that cannot be reused even after the threads complete their operations and leave the tree. Every internal node will have had an operation applied to it at one time and some leaf nodes may have had children removed so there are potentially a lot of unique nulls and, therefore, objects that could be precluded from reuse.

It is possible to solve the problem by using tagged pointers for these ABA prone fields, in which case null can simply be used in place of unique nulls and the defunct objects can be reused safely as per the memory management algorithm. But, the requirement of DWCAS makes these solutions undesirable due to lack of platform support. A CAS solution for this issue could potentially benefit any algorithm that relies on unique null pointers in order to avoid the ABA problem, e.g. [Ellen et al. 2010].

3.4.2 Description

This method borrows from the ideas behind using pointers protected by version tags [IBM 1983]. It would be somewhat impractical to allow a tag and pointer data to be concatenated into the same memory location. 32-bit systems could guarantee at most two or three bits which would lead to the tag wrapping over much too quickly, 64-bit systems could potentially offer more unused bits[3], but this will not be true in the future as virtual address ranges increase. Instead, the memory location only contains a tag when it needs to represent a null value and contains a valid pointer when the memory location points to a descriptor. A technically similar technique is seen in [Fraser and Harris 2007] to store versioned STM descriptor ownership records.

Unique nulls need to be distinguished from normal pointer data which is done by flagging an unused pointer bit (similar to pointer marking). When a new descriptor is to be placed into a memory location containing a unique null, a copy of the unique null is stored in the descriptor before CAS is used to update the memory location. Once the useful work of the descriptor is completed and the pointer to it must be removed from the data structure, CAS is used to replace its pointer with an incremented copy

[3]Current CPU designs based on the Intel x86-64 architecture have their virtual address space limited to 48-bits.

of the unique null. This process guarantees that unique nulls cannot overlap until the tag values wrap around – which can be ignored for 32-bit integers for most practical purposes. If a guarantee is required that the same tag cannot be encountered after having wrapped around, perhaps a system similar to hazard pointers could be employed to ensure a held tag value is not reused.

Other fields in the object can share the unique null values provided that they are only updated in tandem with a descriptor being inserted into the object. This means that there is no need to add additional fields to an object to cache the previous unique nulls of fields that currently contain pointer data.

Unfortunately, the fact that the application of this technique relies on having access to auxiliary bits inside pointers limits its usefulness to programming languages that can modify bare pointers. This means that if the technique were to be applied to Java, for example, it would require support for unique nulls to be built into the language and/or JVM.

3.4.3 Application to single-node BST

In order to integrate this with the single-node BST design, some modifications were required (code is present in Appendix D). The number of auxiliary bits required inside a pointer was reduced by storing the type of operation (`ChildCASOp` or `RelocateOp`) in the `Operation` class. The `MARK` flag never has any associated pointer data nor can it ever be modified once marked, so it can be encoded via a static `Operation` object or a pointer value which is guaranteed to be unused (e.g. address 0). This way, the only states the `op` field and child pointers can have are either null or a valid operation/node.

When a `ChildCASOp` operation is to be applied to a node, the unique null contained in `curr` will have been cached in `currOp`. A copy of the unique null is added to the operation descriptor so that it can be incremented to the correct value when the operation is completed. If a node with no children is being excised by the operation, a null pointer is required to replace it in the child pointer. This is generated by incrementing the unique null to the next unique null in the sequence – the same value that is CASed into the node's `op` field when the operation is removed. Similarly, for a `RelocateOp` operation, the unique nulls contained in both nodes involved are added to the descriptor so that when the operation pointer is removed, the nodes' `op` fields can be updated to the next unique null in the sequence, if necessary.

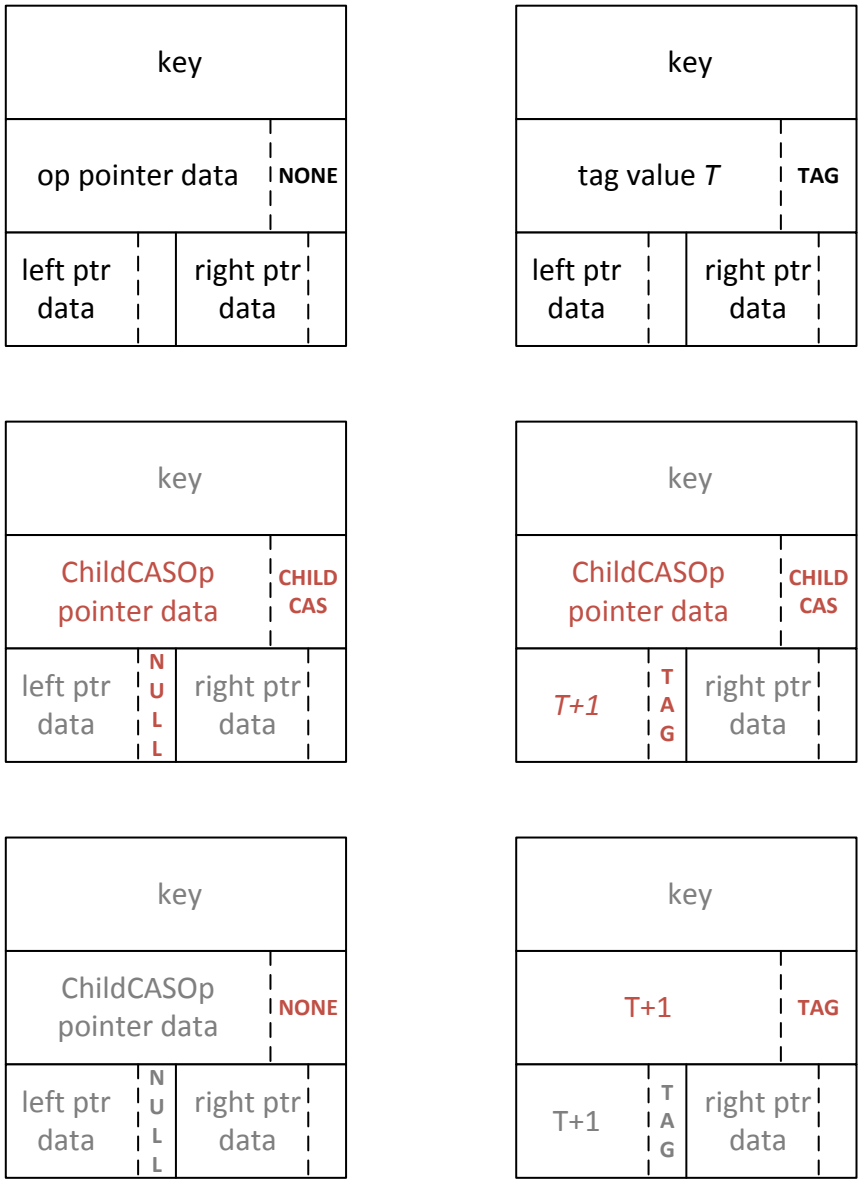

Figure 3.30: Comparing the original unique null technique (left) with the new technique (right). The object being examined is a node from the single-node BST algorithm undergoing the excision of a left child leaf node. Top: initial state. Middle: operation added to node and applied. Bottom: operation removed from node.

A comparison between the original unique null technique (as employed by Ellen et al. [2010]) and the new technique when applied to the single-node BST is outlined in Figure 3.30. The figure shows the update stages of a node having a leaf node left child removed. Both techniques start from a similar state except the new technique contains a tag value in the `op` field rather than some retired object pointer data. In both cases, adding a `ChildCASOp` pointer is CASed into the `op` field to begin the operation. The left child is then excised by converting the old pointer to a unique null in the original technique or writing the next tag value to the left child field in the new technique. The final stage is to remove the active `ChildCASOp` from the node. This is performed using the same process, converting the `ChildCASOp` pointer to a unique null by changing its state flag in the original technique or writing the next tag value to the `op` field in the new technique. The completed state for the original technique now still contains pointer data for two retired objects which cannot be arbitrarily used due to the ABA problem (or else the excised node could be reinserted as the left child of the node in the example), whereas the new technique has no such issue.

Although this technique adds slightly to the memory requirements of the operation structure types, it is a smaller overhead than would be required by tagging each of the pointers (which would require both larger nodes and larger operation descriptors). It is also expected that the overhead of applying this technique to the BST would be negligible compared with the original algorithm given that no further logic was required in the traversal loop as opposed to using tagged pointers which would require explicit reading of tag values at each iteration step.

Chapter 4

Results and Evaluation

The new algorithms were evaluated by comparing their throughput with that of the best known concurrent dynamic set algorithms, on a multi-CPU system. Hazard pointer memory management was then applied to the algorithms which could support it and the performance was examined. Finally, the memory footprint and memory usage of each algorithm were compared.

4.1 Raw performance benchmarks

4.1.1 Experimental setup

Five concurrent ordered set algorithms were compared:

- **Split-node BST** – The newly developed lock-free internal BST with node data spread across two objects (section 3.2).
- **Single-node BST** – The newly developed lock-free internal BST with node data contained in a single object (section 3.3) including optimisations outlined in section 3.3.2.5.
- **External BST** – The lock-free external BST of Ellen et al. [2010].
- **Balanced BST** – The optimistic relaxed balanced AVL tree of Bronson et al. [2010].
- **Skip list** – The Java library's `ConcurrentSkipListMap`, written by Doug Lea [Lea].

All the algorithms were implemented in C++ and benchmarked using a common test harness. The pseudocode provided for the external tree translated to C++ relatively easily, with the exception of the type identification of Node subclasses needed in the tree traversal. This was made possible by adding a field to the Node class to store the derived object type.

The implementation of the balanced tree was adapted partially from the Java code listed in the paper and partially from a code repository listed in the appendix (due to functions omitted from the original paper). In the original Java, updates are only made to the tree after entering Java `synchronized` blocks which are implemented in current versions of Java using a mutex field in every object. These blocks were translated to C++ using mutexes available from the POSIX threads (pthread) library, given the similarity of features. The code for both the balanced tree and Skip list implemented the `java.util.Map` interface and these were simplified to work as a set instead, while maintaining their linearisability.

The skip list was tuned as per the Java implementation such that each node in the base level had a 0.25 probability of having higher level indexing nodes and each level of indexing had a 0.5 probability of having higher level indices. The maximum height of the skip list was initialised to an optimal level given the structure size in each experiment. This was chosen to maximising the probability that the structure would contain a single node at the top level once the set contained every key.

Experiments were run on a dual-processor system with two six-core Intel Xeon E5645 CPUs and 64GB of RAM. Each CPU has a 12MB shared L3 cache and 12 hardware thread contexts available. The operating system was the x86_64 version of Ubuntu 11.04 with Linux kernel 2.6.38-13-generic. The benchmarks were compiled using gcc 4.5.2 with optimisation level O2.

A pseudo-random number generator instance for each thread was used to generate key values and operation types. The generators were seeded with unique values for each benchmark run. In order to get more consistent results, especially at larger key ranges, a *steady-state* number of random keys were added to the set before results were gathered for each run. When running a simulation with random operations with an $A : R$ ratio of adds to removes and random keys from a key range 0 to $K - 1$, if the data structure is initially empty the probability of an add operation succeeding is 1 and the probability of a remove operation succeeding is 0. As keys are added, the probability of successful adds drops and the probability of successful removes increases.

The steady-state number of keys

$$K\left(\frac{A}{A+R}\right)$$

is achieved when the probability of an add operation occurring multiplied by the probability of an add operation succeeding becomes equal to the probability of a remove operation occurring multiplied by the probability of a remove operation succeeding. So, for example, in an experiment with 9% add, 1% remove and 90% contains with a 2,000,000 key range, the steady-state number of nodes is 1,800,000.

In order to eliminate any contention caused by requesting memory from the operating system, each thread was allowed to allocate itself a pool of memory for each experiment before timing measurements were taken. This memory pool was allocated to be large enough to service all memory allocations required by the thread for the duration of the experiment. Objects created by threads were aligned to cache line boundaries (64-byte boundaries for the CPU tested) in order to eliminate the effects of 'false sharing' – where writes to two separate objects can contend if they happen to reside in the same cache line. Memory was not reused by operations or reclaimed to the operating system until the benchmark completed.

All threads were started simultaneously and allowed to run for 15 seconds – this was limited by the amount of memory available in the system given that memory was not reused but increasing this number did not produce results with differing throughput or have an effect on standard deviation. The threads periodically checked their own run time after every 10,000 operations – this number gave the best balance between overhead of checking runtime and synchronising the stopping of all threads at the end of the run. When the last thread finished, the runtime of that thread was divided into the total number of operations completed to yield the total throughput for that experiment. Each experiment was run a minimum of 30 times which was enough to expose the standard deviation and the average throughput was reported.

Experimental parameters that were varied were: the number of threads concurrently executing operations on the structure; the range of keys that could be selected for operations (and hence controlling the total number of nodes in the structure); and the ratio of add, remove and contains operations. The selection of these parameters is influenced by the experiments performed in previous literature comparing concurrent sets [Herlihy et al. 2006; Bronson et al. 2010]. The thread numbers were chosen to demonstrate varying degrees of concurrency starting with 1, 2, 4 & 6 threads running

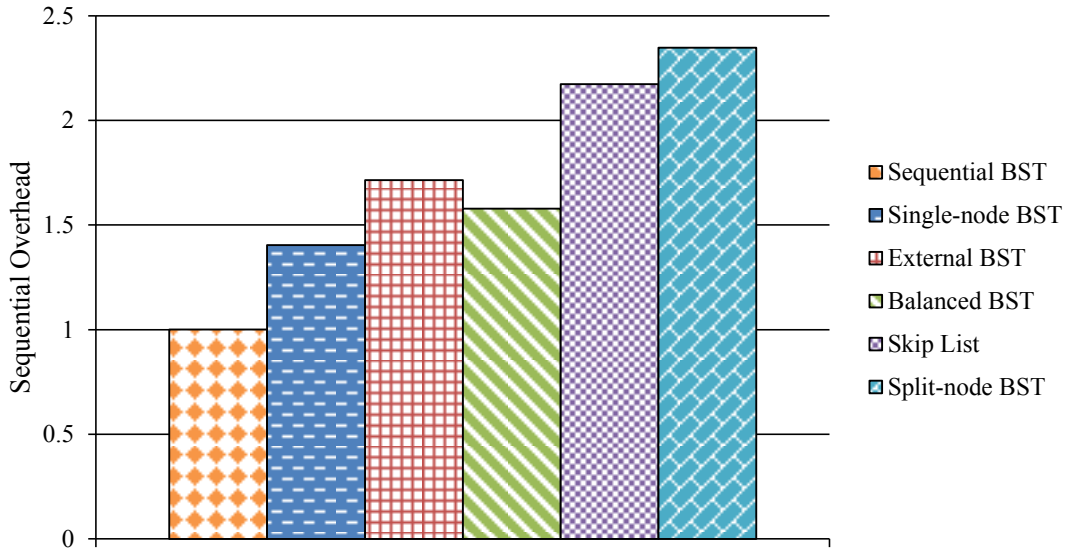

Figure 4.1: Single-threaded overhead versus a sequential BST algorithm for a 2,000 key range and a 50% add, 50% remove and 0% contains workload.

on a single CPU and then increasing in powers of two up to 192 (or 8 threads per hardware thread context available). Four different key ranges, from 2,000 to 2,000,000 increasing in powers of 10, were used to illustrate performance for varying contention levels and data structure sizes. The workloads were varied from being write dominated (50% add, 50% remove, 0% contains) to read dominated (9% add, 1% remove, 90% contains) with a balanced workload (20% add, 10% remove, 70% contains) in between.

4.1.2 Throughput

The single threaded performance of each concurrent algorithm was compared with that of a sequential BST in Figure 4.1. This was done to investigate the overhead associated with parallelising the data structure and also gives insight into the numbers of concurrently operating threads required to match sequential throughput. This was tested for a write heavy workload and averaged across all the key ranges tested. Of the five concurrent algorithms, the single-node tree has the lowest sequential overhead with a 39% increase in execution complexity. The overhead associated with the external tree was higher at 72% which can be attributed to the necessity for all searches to traverse to the leaf nodes before terminating. The balanced tree overhead of 58% was higher than the single-node tree due to the rebalancing functions which are called when nodes are inserted and deleted. The skip list and split-node tree were the poorest showing

here (117% and 134% overhead respectively) despite having a similar *O(log n)* search time for randomly inserted keys because they must read the contents of many more objects during a search.

The graphs in Figure 4.2 and 4.3 show the throughput of the various algorithms in millions of operations per second on the y axes and number of concurrently executing threads on the x axes. Each column of graphs in the figures represents a different key range and each graph in the column represents a different access pattern. Performance was compared for highly-parallel experiments (averaged using results from experiments with $\geq$24 threads) by normalising throughput to that of the single-node tree (in order to directly compare performance against the single-node tree) and combining the results which used the same access patterns (Figure 4.4) and the same key ranges (Figure 4.5).

Overall, the new single-node BST algorithm scales very well – showing a linear increase in throughput as more execution resources are made available. It does not appear to be particularly negatively affected by the high contention (update heavy, small key range) scenarios and offers the best all-round throughput of the algorithms tested when all the results are considered. As expected, the split-node BST algorithm generally performs less well than the algorithms that use a single object to contain node data because each iteration of the search loop requires two disparate objects to be read.

Single-node tree and balanced tree, both being internal trees, have a consistent performance profile when an access pattern is compared across the different key ranges (with the exception of the 2,000 key range). Performance at heavily-threaded workloads shows that the single-node tree has a constant 15-30% higher throughput for the 50-50-0 workload and 5-15% higher throughput for the 20-10-70 workload. Similar performance is exhibited under the read dominated 9-1-90 access pattern which is expected given the read-only `contains` function and structural similarity of each set. The inconsistent results for the 2,000 key range are attributable to the balanced tree's per-node mutual exclusion locks inhibiting progress of other threads.

The simpler traversal loop of the external tree means that its performance is good compared with the single-node tree for small key ranges despite the larger number of nodes which must be traversed to find a key. The external tree also tends to have more competitive performance for write dominated workloads given its much simpler remove routine. It performed on average 5% better for the 2,000 key range for highly-threaded workloads when compared with the single-node tree. However, it is let down by its

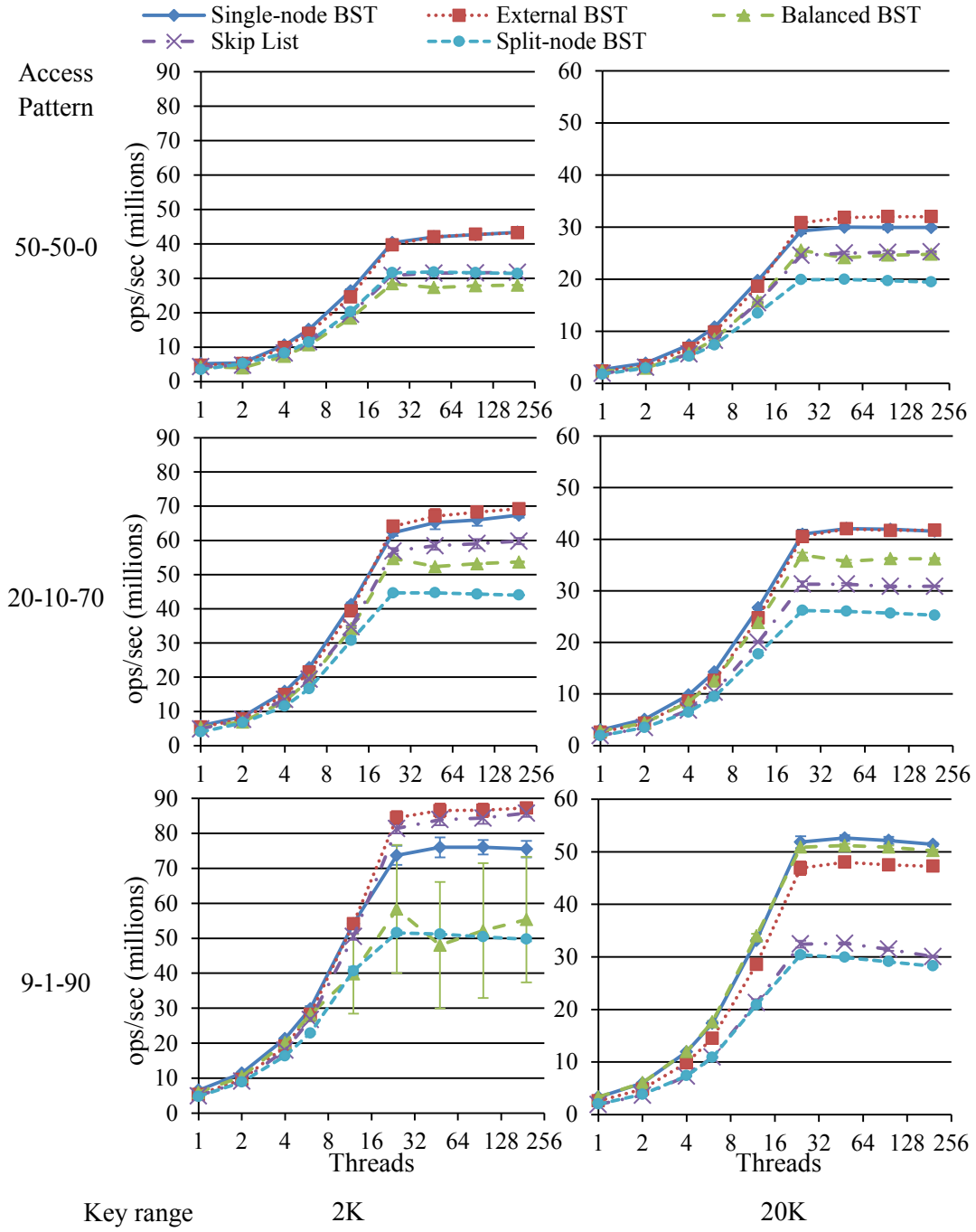

Figure 4.2: Throughput of different algorithms for varying experimental parameters with no memory management. Each row of graphs represents a different access pattern (%add-%remove-%contains), and each column represents a different key range. Each graph plots throughput in millions of operations per second against the number of threads concurrently executing. Error bars show one standard deviation.

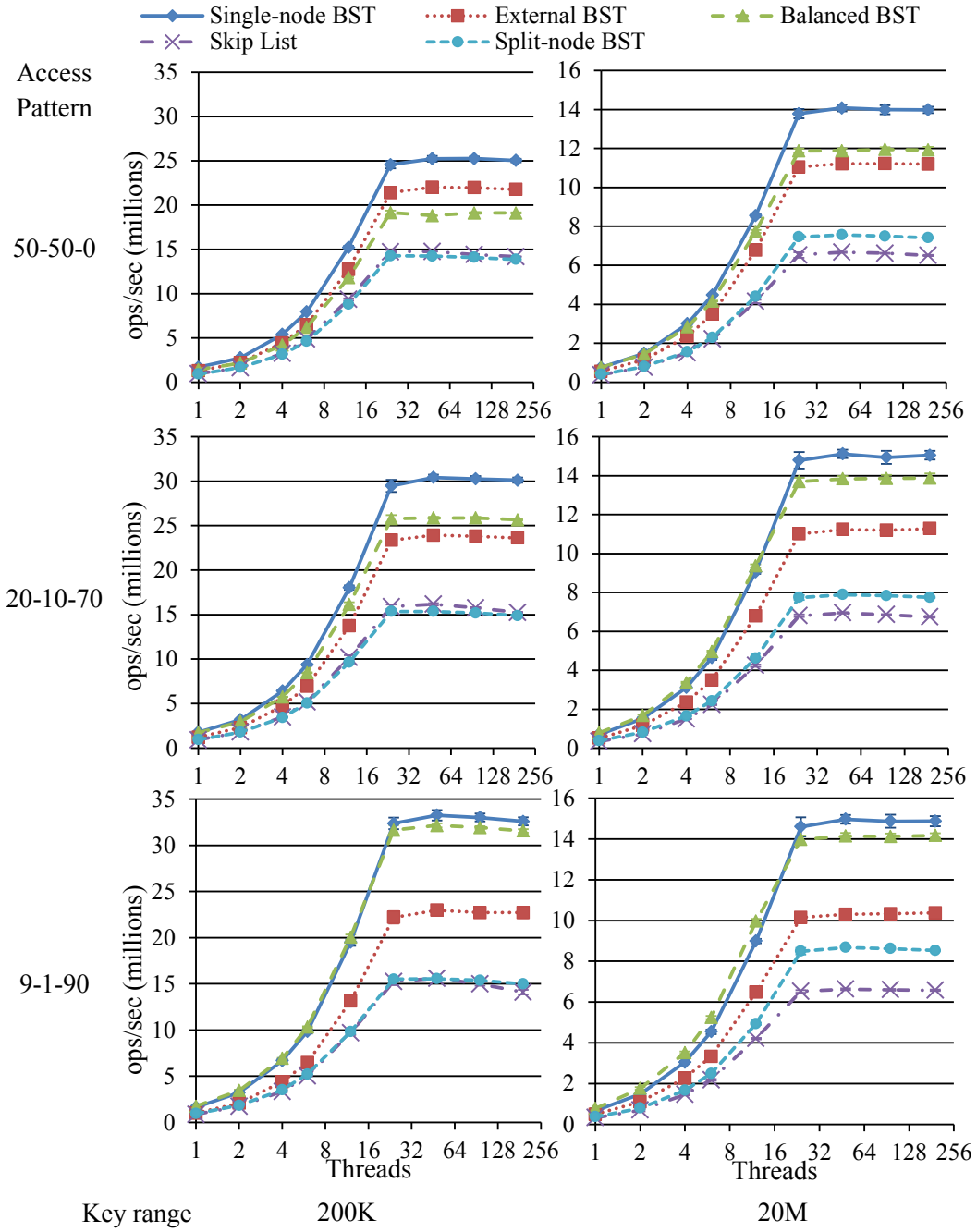

Figure 4.3: Throughput of different algorithms for varying experimental parameters with no memory management. Each row of graphs represents a different access pattern (%add-%remove-%contains), and each column represents a different key range. Each graph plots throughput in millions of operations per second against the number of threads concurrently executing. Error bars show one standard deviation.

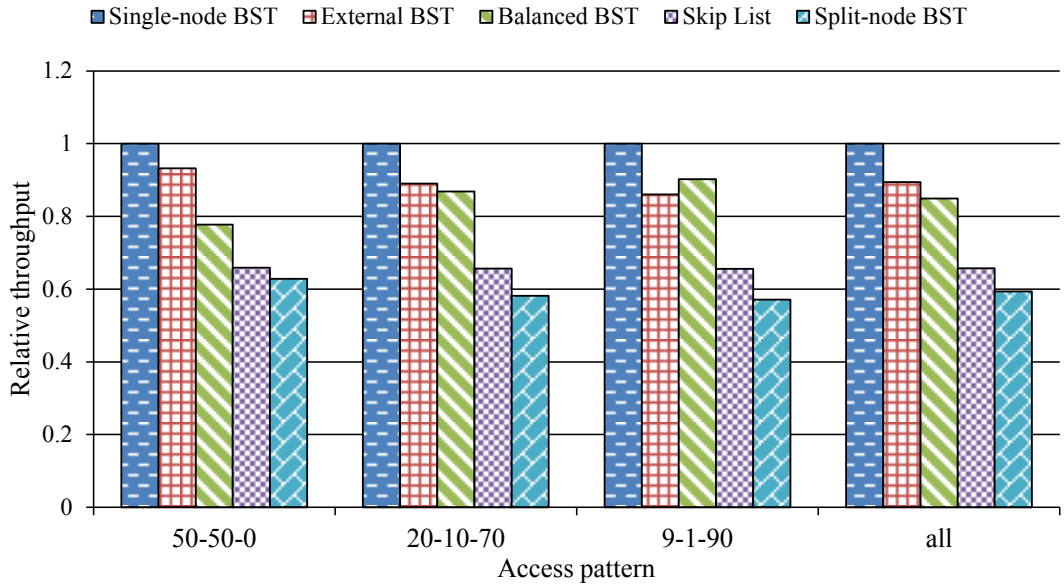

Figure 4.4: Throughput of each algorithm normalised to that of the single-node BST for experiments with ≥24 threads. Each set of bars shows the combined result for a tested access pattern (%add-%remove-%contains).

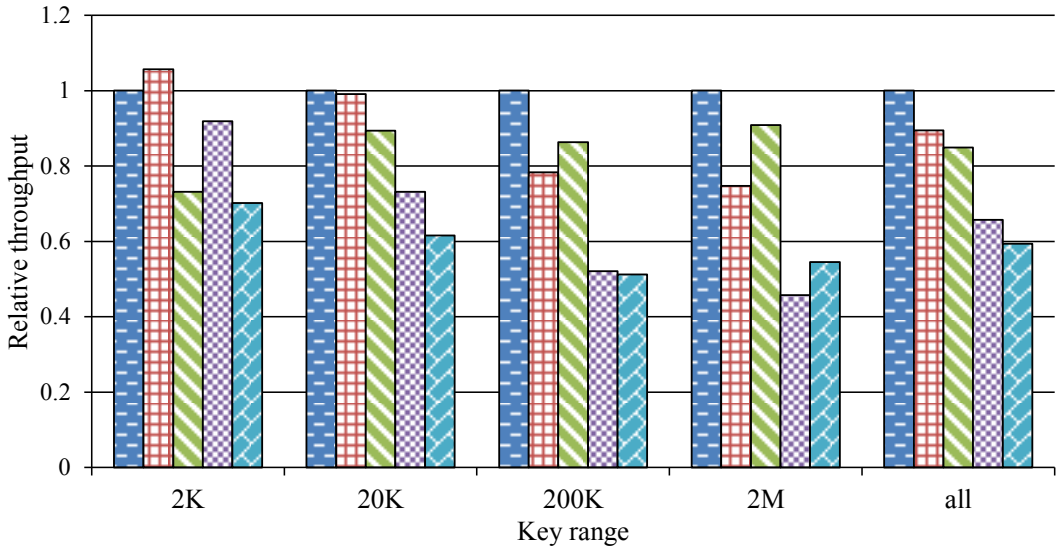

Figure 4.5: Throughput of each algorithm normalised to that of the single-node BST for experiments with ≥24 threads. Each set of bars shows the combined result for a tested key range.

external tree structure when the other results are considered: its average search time is longer than an internal tree as each search must traverse all the way to the leaf nodes – this is reflected in the results for the read dominated workloads. Larger key ranges compound this effect because the height of the tree is greater. The 2,000,000 key range showed the single-node tree to perform on average 34% better and the 9-1-90 workload showed it to be 16% better. The peak average performance differential for a benchmark was tied between the 200,000 and 2,000,000 key ranges with a 9-1-90 workload in which the single-node tree outperformed the external tree by 45%.

The skip list shows very competitive performance for the smallest key range tested and even displayed 10% higher throughput than the single-node tree for a heavily-threaded read-dominated workload. As the key ranges increase, however, it is not able to keep up with the tree-based structures due to its longer traversal (in terms of objects visited). The performance begins to plateau across the various workloads sooner than the other algorithms as the key ranges increase, which would suggest that the search uses a bigger proportion of execution time in an operation than the other algorithms. The largest key range tested shows the single-node tree having a consistent doubling of throughput or better and the trend would suggest that this gap would widen if structure size was increased even further.

The split-node tree performance does not climb as fast as the other algorithms when moving from a write heavy to a read heavy workload due in large part to the fact that most of the execution time is spent in the traversal loop. For the smaller key ranges, it performs closer to but consistently behind the other algorithms, with its throughput between 0% and 25% behind the nearest rival. As the tree sizes increase beyond this, the cost of reading both the `Node` and `Contents` objects at each traversal dominates the execution time of the algorithm and shows the single-node design performing 80-120% faster than it. At these larger key ranges, the split-node tree does begin to perform better than the skip list as the skip list would require visiting at least two index node objects in each skip level as it searches for a key.

4.2 Hazard pointer memory management

4.2.1 Experimental setup

The application of hazard pointers to most concurrent data structures is non-trivial. The interactions with objects in the structure must be closely examined to ensure that the hazard pointers are acquired, released and copied in a safe and correct manner and order. For example, to ensure that a hazard pointer to an object has been correctly acquired, it must be guaranteed to have not been retired from the structure. Care must also be taken when determining the point at which objects should be retired to ensure that no other threads can proceed to acquire a hazard pointer to the object or can incorrectly determine a successful acquisition.

The single-node BST code, after having adding hazard pointers, is listed in Appendix D. Support code for the hazard pointer system, and the object allocation and retirement functions is omitted but was implemented in the same manner as Michael [2004]. With the addition of the unique null code, very few changes were required to the fundamental algorithm to make it compatible with hazard pointers. The most significant change was to add a fourth state to a `RelocateOp` – which is reached after the operation has been removed from the destination node. This was modified to ensure that helping operations executing `helpRelocate` that cannot verify a successful acquisition of a hazard pointer to the destination node do not access the node if there is a possibility it has been retired (and possibly reclaimed).

The external tree also required some modification in order to make it compatible with hazard pointers. Because this algorithm has the same requirement for unique null pointers when descriptor objects are cleared from the structure, the new scheme for generating non-object backed unique null pointers was applied to it. The helping function – which is normally called after find returns – was integrated into the search loop to remove cases where retired leaf nodes could be accessed through a marked internal node protected by a hazard pointer. Because of this change, a separate read-only contains function was created in order to preserve the performance of the unmodified algorithm.

The balanced tree implementation was unsuitable for benchmarking because it is incompatible with hazard pointers (except without significant modification). The recursive search loop means that each node object traversed would require an additional hazard pointer whereas the memory management system requires a fixed number of

hazard pointers per-thread to be known in advance. It would be possible to replace this with an iterative search, however, similar to the internal and external trees that trigger a restart from the root when concurrent modifications are detected rather than stepping backward to a consistent state. A more fatal issue is that rebalancing requires that the parent pointers inside nodes can be followed, but these are not guaranteed to point to nodes that still exist in the tree. This is not a problem in the Java implementation because the parent pointers will prevent the removed nodes from being garbage collected.

The skip list code that was adapted for use in the first set of experiments was also heavily reliant on garbage collection memory management which made it incompatible with hazard pointers. The index objects which allow searching threads to skip over parts of the structure are optimistically followed even when they could be already removed from the structure. This makes it difficult to tell whether a hazard pointer acquisition was successful or not and is the case for both the right and the down links. Index objects are optimistically inserted and can be inserted after the node they refer to has been retired without error, which could cause a memory leak. Also, because index objects are pointed to by both down and right pointers belonging to other indices, it is non-trivial to determine when they may be safely retired once unlinked from their index level. Other less optimised skip list implementations do support hazard pointers but results from benchmarking these could not be directly compared with the previous experiments.

While configuring this experiment it became apparent that when a large number of threads were executing, the performance was becoming bottlenecked. This was being caused by calls to glibc malloc/free when some threads had empty reuse lists and were allocating new objects, and other threads had excess pooled nodes and were reclaiming them. Due to this limitation in the version of glibc used, the memory allocator was switched to a concurrency friendly alternative: Thread-Caching Malloc (TCMalloc) [TCMalloc]. TCMalloc was originally developed by Google as a drop-in replacement for glibc malloc. It uses per-thread caches of objects to satisfy memory allocations of varying sizes and these caches are periodically scanned for unused objects which are then returned to a central heap to allow for allocation by other threads. Moving to this allocator yielded a performance profile closer to what would be expected as more threads were concurrently executing.

The benchmark testing setup was otherwise configured identically to that of the previous experiments with no memory management. The hazard pointer system was

tuned to reduce contention by halving the suggested optimal frequency of hazard pointer scanning so that it would only execute once the retire list reached $4NH$ objects where N is the number of threads and H is the number of hazard pointers per thread. The number of objects allowed to be stored by each thread before reclaiming them was set to 4 times the scan threshold. These changes increased the per-thread memory footprint of the hazard pointer system but reduced the frequency of scans to the entire hazard pointer table and reduced calls to the memory allocator.

4.2.2 Throughput

The throughput results of the two algorithms using hazard pointer memory management are shown in a similar graph array to the previous throughput results in Figure 4.6 and 4.7. The results from the experiments run with no memory management are included for comparison. When hazard pointers are implemented, the single-node BST has consistently higher throughput than the external BST – even in test configurations where the external tree previously performed better.

The profile of throughput from both algorithms follows a similar rise and fall pattern which would indicate that it is as a result of the common hazard pointer system rather than a change in caching behaviour. The drop-off of performance, as the thread count increases beyond the number of available hardware thread contexts, is eliminated as the structure size increases to the largest level with mostly read-only operations which would suggest that it is due to contention in the memory management system. These experiments would involve the least amount of interaction with the memory management system, given their lower throughput.

When looking at the smaller sized trees and the read-focussed workloads, the performance of the single-node tree surpasses the results with no memory management. This can be attributed to the fact that the all the memory in use can be contained in the cache. When there is no memory management, a new cache line sized object must be read into the cache from memory every time a new object is created whereas with hazard pointers, a defunct object's memory is recycled so it could already be contained in the cache before reallocation occurs. This effect can be seen to taper off as the size of the key set increases.

The cases where the external tree performs worse than the experiments with no memory management tend to be toward smaller structure size and update-centric

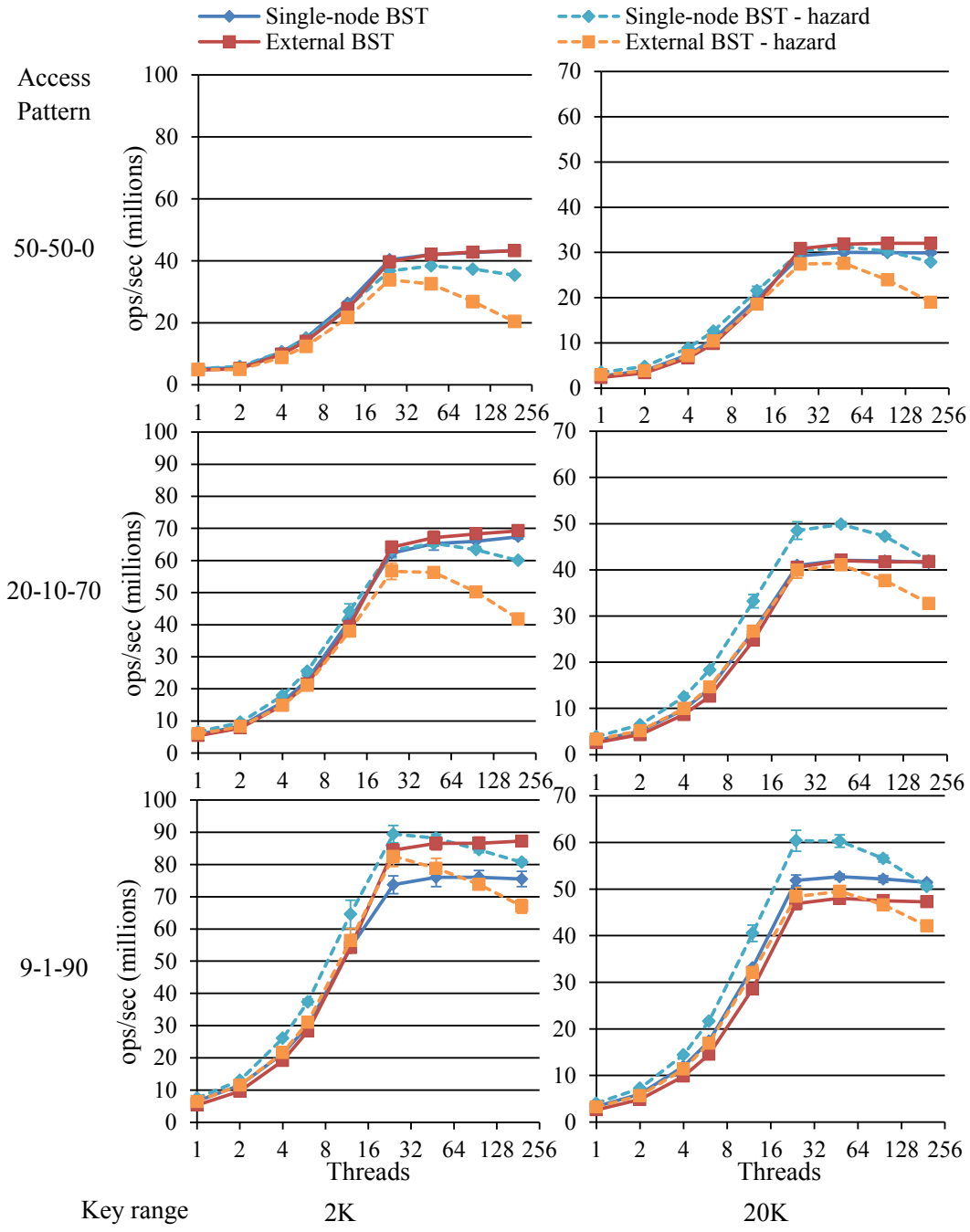

Figure 4.6: Throughput of single-node BST and external BST algorithms for varying experimental parameters with hazard pointer memory management. Results with no memory management are also shown for comparison. Each row of graphs represents a different access pattern (%add-%remove-%contains), and each column represents a different key range. Each graph plots throughput in millions of operations per second against the number of threads concurrently executing. Error bars show one standard deviation.

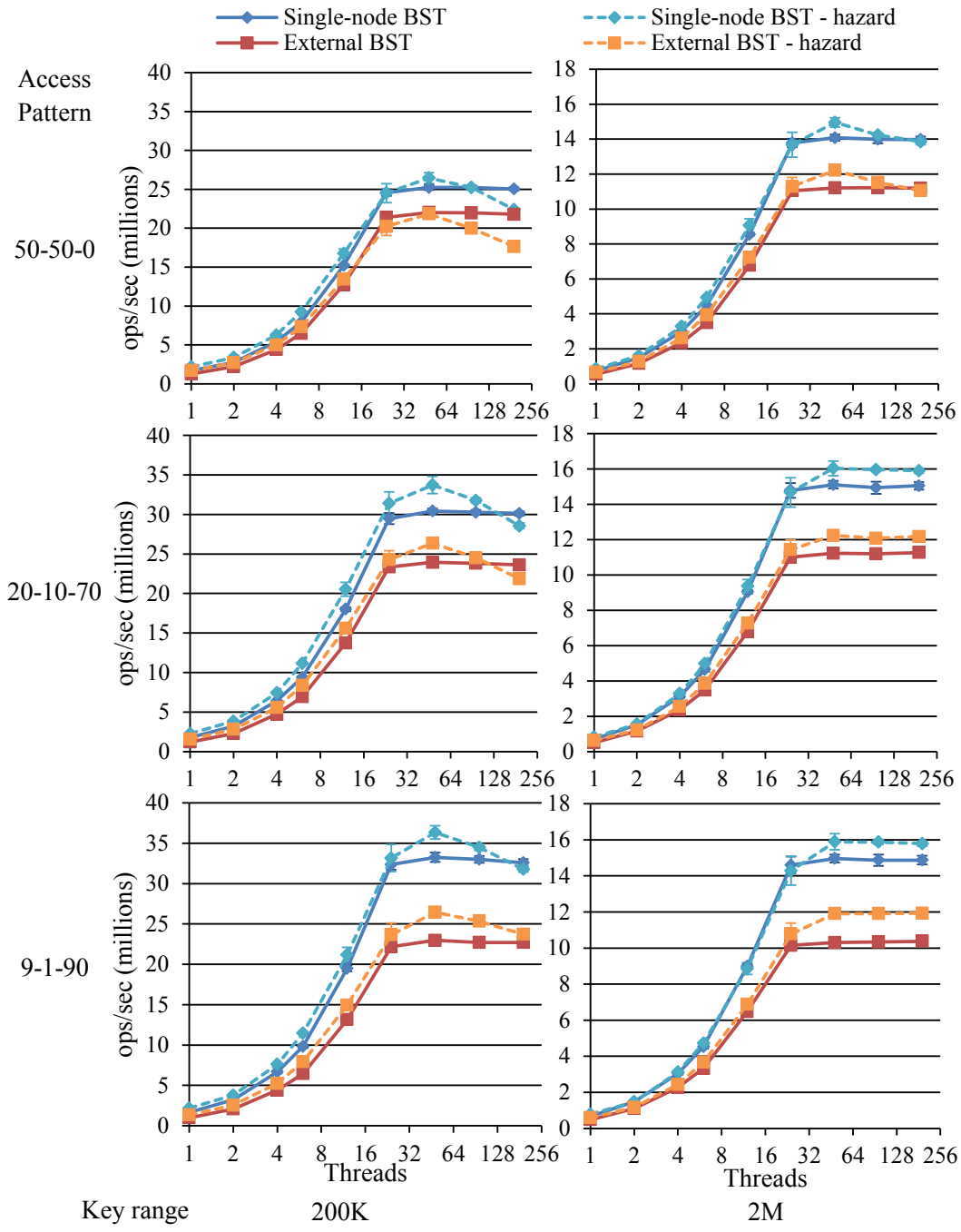

Figure 4.7: Throughput of single-node BST and external BST algorithms for varying experimental parameters with hazard pointer memory management. Results with no memory management are also shown for comparison. Each row of graphs represents a different access pattern (%add-%remove-%contains), and each column represents a different key range. Each graph plots throughput in millions of operations per second against the number of threads concurrently executing. Error bars show one standard deviation.

workloads. These configurations would already represent the highest-throughput and highest-contention of the set of benchmarks and the added contention of the memory management system is what causes the performance to drop. This drop is more severe for the external tree because of the heavier burden it puts on the memory management system compared to the single-node tree. The external tree allocates four objects for an insertion and one object for a removal, compared with the single-node tree which allocates two objects for an insertion, and either one or two for a removal (depending on the number of children).

4.3 Memory efficiency

The memory usage of each algorithm was also compared. These results can be calculated by examining the structure of each algorithm and summing the sizes of the objects used, for example an external tree containing N keys will contain N leaf node objects (8 bytes), $N - 1$ routing node objects (32 bytes) and some auxiliary objects totalling approximately 40 bytes per key. These were verified experimentally for each algorithm. Results were gathered using the same benchmark as for the throughput results with a 2,000 key range and a 50-50-0 workload and no memory management. Figure 4.8 shows the steady state memory footprint per key of the five algorithms tested. The absolute memory footprint (the memory being actively used in each object) was calculated by iterating over the number of objects in use by each structure at the end of a run and dividing by the number of keys contained in the set. The `sizeof` operator was used to total the amount of memory required to store these objects. In reality, however, the memory allocator will generally not allocate memory for these objects in the most efficiently sized blocks. A more realistic memory usage figure was totalled by calculating the memory footprint based on the assumption that all objects are aligned on 64-byte blocks, as in our benchmark.

The dynamic memory consumption of each algorithm was also examined (Figure 4.9). This refers to the average amount of memory allocated during an operation when all operations are taken into account. For the 50-50-0 workload, the completed operations will consist of 25% successful add, 25% unsuccessful add, 25% successful remove and 25% unsuccessful remove. This does not factor into the steady-state size of the data structure but represents the dynamic load imparted on the memory allocator or memory management subsystem.

Examining the results presented in Figure 4.8, it is apparent that despite only requiring a single node object per key, the balanced BST has significantly higher memory usage per key than the other algorithms. This is down to the extra fields in its node class, combined with the `pthread_mutex_t` structure which itself amounts to 40 bytes. The leaf node objects of the external tree have a relatively low absolute memory size because they are only used to store a key, but in the real-world memory usage scenario this bears the full cost of an aligned object. This is similarly seen with the index objects of the skip list and the node objects of the split-node tree. The tuning factor used for the skip list meant that there should be approximately half as many index objects as there were nodes, but varying this would increase or decrease its memory efficiency while also affecting performance [Pugh 1990b]. The balanced tree also suffers an overhead beyond the memory use for a single node here because the 50-50-0 workload causes it to retain an extra 20% superfluous routing nodes. This means that its real memory footprint is the least efficient at 144% higher than that of the single-node tree. The extra objects required for the external tree and skip list give them a memory-footprint 100% and 52% higher respectively. The split-node BST also has a 100% larger memory footprint than the single-node tree because it requires exactly twice as many aligned objects, given the structure similarity between the two.

The experimentally determined memory consumption rates (Figure 4.9) can be accounted for by adding up the amount of memory allocated for each of the possible operations. Taking the external tree as an example, an unsuccessful add or remove will only perform a read-only search – a successful add requires the creation of four objects and a successful remove requires the creation of one object. There is also a small overhead of wasted memory if the objects are created, but then the operation is ultimately unsuccessful. The memory consumption of the balanced tree is quite low here because it only ever allocates memory for an add operation and it is guaranteed not to go to waste because the insertion point is locked. It also has the ability to 'reactivate' routing nodes, therefore requiring no memory allocation for those particular add operations. The single-node, split-node and external BSTs allocate more memory per operation than the two others here because they all use operation structures to allow helping, but with the external tree using 49% more and the split-node tree using 40% more than the single-node tree. The balanced tree and skip list are more efficient here, only needing 50% and 45%, respectively, of the requirement of the single-node tree. In real-world terms, higher figures here could result in reduced performance if the memory allocator, garbage collector or similar is causing a bottleneck.

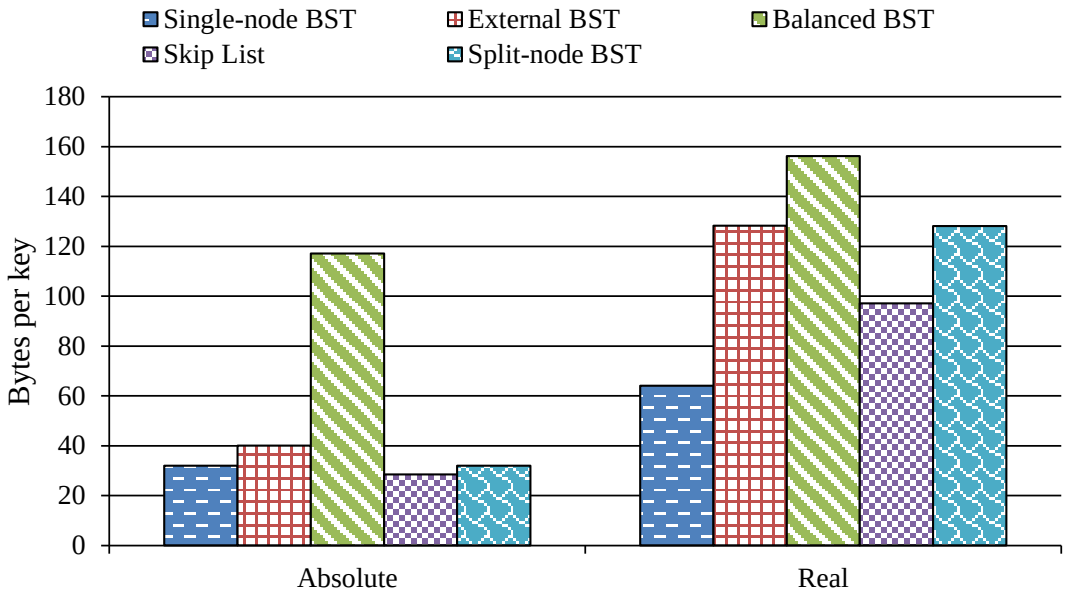

Figure 4.8: Memory footprint per key of the tested algorithms for a 2,000 key range and a 50% add 50% remove 0% contains workload.

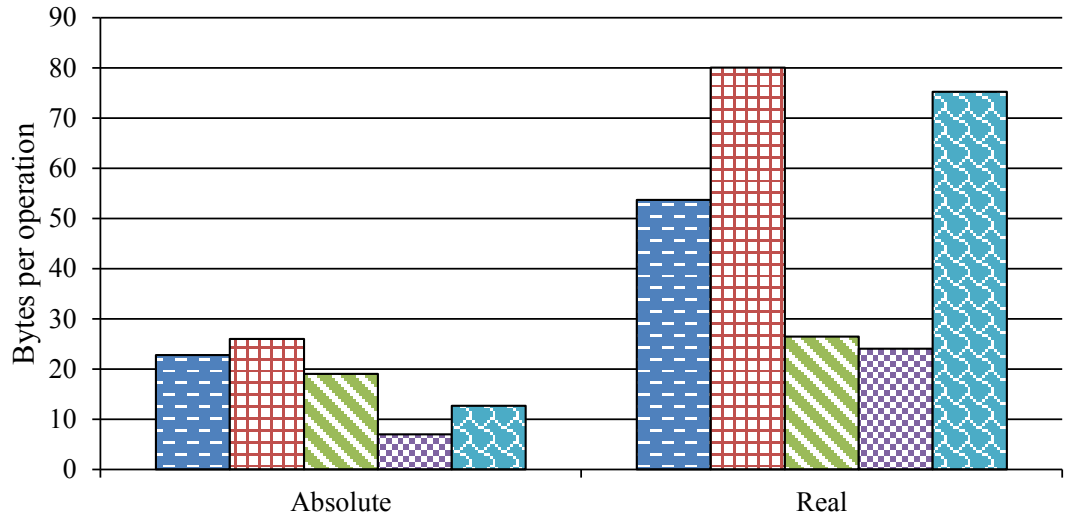

Figure 4.9: Memory allocation averaged per operation of the tested algorithms for a 2,000 key range and a 50% add 50% remove 0% contains workload.

4.4 Summary

In this chapter, the performance and memory usage of the new binary search tree algorithms were compared with that of other concurrent set algorithms. The single-node tree was demonstrated to improve throughput significantly from that of the split-node tree, while halving the amount of memory required. The external tree performed well when the structure size was low, but the single-node tree was shown to perform significantly better at large structure sizes due to its internal BST design. This was achieved while using just half of the memory required by the external tree. The balanced tree algorithm performed well although it did not perform consistently under high contention. Unfortunately, it requires mutual exclusion to synchronise updates and rebalancing operations and requires double the memory of the new single-node tree algorithm. The skip list performed well compared with the other algorithms for small structure sizes and can have its memory requirement tuned, but its throughput was not competitive when it contained large numbers of keys. Hazard pointer memory management was applied to the single-node tree and external tree and was shown not to impart any significant overhead to the algorithms.

Chapter 5

Conclusion

5.1 Contribution

This thesis presents two new algorithms for lock-free internal binary search trees. They have been informally proven to be both lock-free and linearisable, fulfilling the design goals. The techniques described and used in order to make the algorithm linearisable could be applied to future concurrent data structure designs. The design of the single-node binary search tree achieved the goal of implementing a data structure with higher throughput than previously published concurrent dynamic set algorithms.

The extensive experimentation which was carried out showed how the algorithm performs under a wide range of scenarios: under high and low contention, in update heavy and read heavy workloads, and in single-socket and multi-socket systems. It was found to perform well in all of these cases compared with competing algorithms for concurrent dynamic sets. When examined against a recent lock-free external binary search tree design, the new algorithm outperformed it in almost every metric. A concurrent AVL tree does hold the advantage of being self-balanced, but at the cost of structure size and undesirable concurrency behaviour due to mutual exclusion. Lock-free skip lists were found to lose competitiveness at larger structure sizes compared with tree-based structures, with the presented algorithm showing 65% higher throughput when all the performance differentials are averaged. The new algorithm was also shown to have the single-threaded performance closest to that of a sequential binary search tree implementation.

The new internal binary search tree algorithm was also designed to be more

memory efficient and has a considerably smaller memory footprint compared with the other algorithms tested. When very large data structures are considered, this results in a significant memory saving. By minimising the number of objects on the search path, this also reduced the number of memory accesses and had the effect of increasing performance significantly, as was seen by comparing the results of the two presented internal binary search tree algorithms.

It was shown that reclaiming shared memory from the data structure could be performed in an efficient manner without negatively affecting performance. The presented algorithms are compatible with commonly used efficient memory reclamation systems. A technique for generating ABA-preventing unique null pointers was also described and used to reduce the memory consumption of the algorithm. These unique null pointers were applied to solve a similar problem with the lock-free external tree, which was included in the evaluation, and could be applied to other descriptor-based algorithms.

5.2 Future work

- This thesis includes informal proofs and model checking as a means to prove the correctness and the non-blocking property of the presented algorithms. The validity of this method is argued, but in order to solidify these, formal proofs would be desirable.

- The experiments performed on the set algorithms were all run using randomly generated keys. This would typically have the effect of keeping trees approximately balanced. In the worst case scenario, with all keys inserted in ascending order, a tree would resemble a linked list and it is expected that the explicitly balanced data structures would have a significant advantage. Future work could investigate a lock-free, approximate and low overhead balancing algorithm to ensure performance cannot be degraded in this way.

- The new tree algorithm, as described, implements the set abstract data type but it is more common for programmers to use the (similar) *map* abstract data type. A map stores key-value pairs as opposed to sets (which only store keys). The current tree design requires that keys are modifiable to allow key relocation, but both key and value would have to relocate in a map. A possible solution could be to require that the value is an object pointer residing in an adjacent memory location to the

key – this would allow DWCAS to be used to update both atomically. Another possible solution could be to store the key and value in a separate key-value pair object and replace the key field of nodes with a single pointer to this – but doing this would double the number of memory accesses during a traversal, negatively affecting performance. Ideally a solution to this problem would not introduce extra per-node memory accesses and be implemented using single-word CAS.

Appendix A

SPIN Model Checking Code for Split-Node BST

```
/* Parameters */
#define INSERTS     2
#define DELETES     2
#define KEY_RANGE   8

/* Constants */
#define NULL        63
#define ROOT        0

#define NOTFOUND    0
#define FOUND       1
#define ABORT       2
#define CONTINUE    3

#define ONGOING     0
#define FAILED      1
#define SUCCESSFUL  2

/* Structures */
typedef TreeNode
{
  byte contents;
}

typedef Contents
{
  byte key;
  byte left;
  byte right;
}

typedef Relocation
{
  byte state;
  byte destination;
  byte oldContents;
  byte newContents;
```

```
38   byte failContents;
39 }

41 /* Macros */
42 #define ISMARKED( n )          ( ( n & 64 ) != 0 )
43 #define ISRELOCATING( n )      ( ( n & 128 ) != 0 )
44 #define ISFLAGGED( n )         ( ( n & 192 ) != 0 )
45 #define GETMARKEDREF( n )      ( n | 64 )
46 #define GETRELOCATIONREF( n )  ( n | 128 )
47 #define GETREF( n )            ( n & 63 )

49 /* Global variables */
50 TreeNode    memNodes[ 16 ];
51 byte        memNodesUsed = 0;
52 Contents    memContents[ 16 ];
53 byte        memContentsUsed = 0;
54 Relocation  memRelocations[ 4 ];
55 byte        memRelocationsUsed = 0;
56 bool        setupDone = false;
57 hidden byte threadDone = 0;
58 short       keyMask = 0;

60 /* Variable Aliases */

62 /* Helper functions */

64 inline find( _key, _pred, _predData, _curr, _currData, _auxRoot,
       _result )
65 {
66   atomic
67   {
68     _result = NOTFOUND;
69     _curr = _auxRoot;
70     _currData = memNodes[ _curr ].contents;

72     if
73     :: ( ISFLAGGED( _currData ) ) ->
74       _result = ABORT;
75       next = NULL;
76     :: else ->
77       next = memContents[ _currData ].right;
78       lastRight = _curr;
79       lastRightData = _currData;
80     fi;
81   }

83   do
84   :: ( next != NULL ) ->
85     atomic
86     {
87       _pred = _curr;
88       _predData = _currData;
89       _curr = next;
90       _currData = memNodes[ _curr ].contents;
91     }

93     // Node is relocating
94     if
95     :: ( ISRELOCATING( _currData ) ) ->
96       if
```

```
97        :: ( _curr == memRelocations[ GETREF( _currData ) ].
              destination ) ->
98          processOp( GETREF( _currData ), opSuccess );
99          if
100         :: ( opSuccess ) -> _currData = memRelocations[ GETREF(
                _currData ) ].newContents;
101         :: else -> _currData = memRelocations[ GETREF( _currData
                ) ].oldContents;
102         fi;
103       :: else ->
104         processOp( GETREF( _currData ), opSuccess );
105         if
106         :: ( opSuccess ) -> _currData = GETMARKEDREF(
                memRelocations[ GETREF( _currData ) ].failContents );
107         :: else ->
108           /* CAS( &( _curr->contents ), _currData, op->
                failContents ) */
109           atomic
110           {
111             if
112             :: ( memNodes[ _curr ].contents == _currData ) ->
113               memNodes[ _curr ].contents = memRelocations[ GETREF
                    ( _currData ) ].failContents;
114               _currData = memRelocations[ GETREF( _currData ) ].
                    failContents;
115             :: else ->
116               _result = ABORT;
117               break;
118             fi;
119           }
120         fi;
121     fi;
122   :: else -> skip;
123   fi;

125   // Node is marked
126   if
127   :: ( ISMARKED( _currData ) ) ->
128     temp = GETREF( _currData );
129     exciseNode( _pred, _predData, _curr, temp, next, _currData
          );

131     if
132     :: ( _currData != NULL ) ->
133       _curr = _pred;
134       _result = CONTINUE;
135     :: else ->
136       _result = ABORT;
137       break;
138     fi;
139   :: else -> skip;
140   fi;

142   // Normal traverse
143   atomic
144   {
145     if
146     :: ( _result == NOTFOUND ) ->
147       if
148       :: ( _key == memContents[ _currData ]._key ) ->
```

```
149             _result = FOUND;
150             break;
151           :: ( _key < memContents[ _currData ]._key ) ->
152             next = memContents[ _currData ].left;
153           :: else ->
154             next = memContents[ _currData ].right;
155             lastRight = _curr;
156             lastRightData = _currData;
157           fi;

159           /* Theory checking */
160           if
161           :: ( next == NULL ) && ( memNodes[ lastRight ].contents
                == lastRightData ) ->
162             assert( ( keyMask & ( 1 << _key ) ) == 0 );
163           :: else -> skip;
164           fi;

166         :: else -> _result = NOTFOUND;
167         fi;
168       }

170     :: else -> break;
171     od;

173     if
174     :: ( memNodes[ lastRight ].contents != lastRightData ) ->
          _result = ABORT;
175     :: else -> skip;
176     fi;
177   }

179   inline exciseNode( _pred, _predData, _removeNode, _removeData,
        _next, _replaceData )
180   {
181     /* allocate replacement contents */
182     atomic
183     {
184       _replaceData = memContentsUsed;
185       memContentsUsed++;
186       memContents[ _replaceData ].key = memContents[ _predData ].
            key;

188       if
189       :: ( memContents[ _predData ].left == _removeNode ) ->
190         memContents[ _replaceData ].right = memContents[ _predData
              ].right;
191         if
192         :: ( memContents[ _removeData ].left == NULL ) ->
193           memContents[ _replaceData ].left = memContents[
                _removeData ].right;
194         :: else ->
195           memContents[ _replaceData ].left = memContents[
                _removeData ].left;
196         fi;
197         _next = memContents[ _replaceData ].left;
198       :: else ->
199         memContents[ _replaceData ].left = memContents[ _predData
              ].left;
200         if
```

```
201         :: ( memContents[ _removeData ].left == NULL ) ->
202           memContents[ _replaceData ].right = memContents[
                _removeData ].right;
203         :: else ->
204           memContents[ _replaceData ].right = memContents[
                _removeData ].left;
205         fi;
206         _next = memContents[ _replaceData ].right;
207       fi;
208     }

210   /* CAS( &( _pred->contents ), _predData, _replaceData ) */
211   atomic
212   {
213     if
214     :: ( memNodes[ _pred ].contents == _predData ) ->
215       memNodes[ _pred ].contents = _replaceData;
216     :: else -> _replaceData = NULL;
217     fi;
218   }

220 }

222 inline processOp( this, result )
223 {
224   seenState = memRelocations[ this ].state;

226   if
227   :: ( seenState == ONGOING ) ->
228     /* VCAS( destination, oldContents, GETRELOCATIONREF( this ) )
            */
229     atomic
230     {
231       seenContents = memNodes[ memRelocations[ this ].destination
              ].contents;
232       if
233       :: ( memNodes[ memRelocations[ this ].destination ].
              contents == memRelocations[ this ].oldContents ) ->
234         memNodes[ memRelocations[ this ].destination ].contents =
              GETRELOCATIONREF( this );
235       :: else -> skip;
236       fi;
237     }

239     if
240     :: ( ( seenContents == memRelocations[ this ].oldContents )
           || ( seenContents == GETRELOCATIONREF( this ) ) ) ->
241       /* VCAS( &state, ONGOING, SUCCESSFUL ) */
242       atomic
243       {
244         if
245         :: ( memRelocations[ this ].state != FAILED ) ->
246           if
247           :: ( memRelocations[ this ].state == ONGOING ) ->
248             assert( ( keyMask & ( 1 << memContents[
                    memRelocations[ this ].oldContents ].key ) ) != 0
                    );
249             keyMask = keyMask ^ ( 1 << memContents[
                    memRelocations[ this ].oldContents ].key );
```

```
250             :: else -> skip;
251             fi;

253             memRelocations[ this ].state = SUCCESSFUL;
254             seenState = SUCCESSFUL;
255           :: else -> seenState = FAILED;
256           fi;
257         }

259       :: else ->
260         /* VCAS( &state, ONGOING, FAILED ) */
261         atomic
262         {
263           if
264           :: ( memRelocations[ this ].state != SUCCESSFUL ) ->
265             memRelocations[ this ].state = FAILED;
266             seenState = FAILED;
267           :: else -> seenState = SUCCESSFUL;
268           fi;
269         }
270       fi;

272     :: else -> skip;
273     fi;

275     /* CAS( &( destination->contents ), GETRELOCATIONREF( this ),
           seenState == SUCCESSFUL ? newContents : oldContents ); */
276     atomic
277     {
278       if
279       :: ( seenState == SUCCESSFUL ) ->
280         if
281         :: ( memNodes[ memRelocations[ this ].destination ].
                contents == GETRELOCATIONREF( this ) ) ->
282           memNodes[ memRelocations[ this ].destination ].contents =
                memRelocations[ this ].newContents;
283         :: else -> skip;
284         fi;
285       :: else ->
286         if
287         :: ( memNodes[ memRelocations[ this ].destination ].
                contents == GETRELOCATIONREF( this ) ) ->
288           memNodes[ memRelocations[ this ].destination ].contents =
                memRelocations[ this ].oldContents;
289         :: else -> skip;
290         fi;
291       fi;
292     }

294     result = seenState == SUCCESSFUL;
295 }

297 /*+------------------------+*/
298 /*| Setup process          |*/
299 /*+------------------------+*/
300 active proctype setup()
301 {
302   if
303   :: ( setupDone == false ) ->
304     atomic
```

```
305     {
306       memNodesUsed++;
307       memNodes[ ROOT ].contents = ROOT;

309       memContentsUsed++;
310       memContents[ ROOT ].key = 0;
311       memContents[ ROOT ].left = NULL;
312       memContents[ ROOT ].right = NULL;

314       /* Insert some nodes */

316       memContents[ ROOT ].right = 1;
317       memNodesUsed++;
318       memContentsUsed++;
319       memNodes[ 1 ].contents = 1;
320       memContents[ 1 ].key = 1;
321       memContents[ 1 ].left = NULL;
322       memContents[ 1 ].right = 2;
323       keyMask = keyMask | ( 1 << 1 );

325       memNodesUsed++;
326       memContentsUsed++;
327       memNodes[ 2 ].contents = 2;
328       memContents[ 2 ].key = 3;
329       memContents[ 2 ].left = 3;
330       memContents[ 2 ].right = 4;
331       keyMask = keyMask | ( 1 << 3 );

333       memNodesUsed++;
334       memContentsUsed++;
335       memNodes[ 3 ].contents = 3;
336       memContents[ 3 ].key = 2;
337       memContents[ 3 ].left = NULL;
338       memContents[ 3 ].right = NULL;
339       keyMask = keyMask | ( 1 << 2 );

341       memNodesUsed++;
342       memContentsUsed++;
343       memNodes[ 4 ].contents = 4;
344       memContents[ 4 ].key = 6;
345       memContents[ 4 ].left = NULL;
346       memContents[ 4 ].right = NULL;
347       keyMask = keyMask | ( 1 << 6 );

349       setupDone = true;
350     }
351   :: else -> skip;
352   fi;

354   /* Wait for all threads */
355   threadDone == ( INSERTS + DELETES );
356 }


359 /*+-----------------------+*/
360 /*| Insert process        |*/
361 /*+-----------------------+*/
362 active [INSERTS] proctype insert()
363 {
```

```
364   /* Declare local variables */
365   byte key = 1, pred, curr, predData, currData, next, result,
          newNode = NULL, newContents = NULL, seenState, seenContents,
           lastRight, lastRightData, temp;
366   bool opSuccess;

368   /* Wait for setup to complete before beginning */
369   setupDone == true;

371   /* Randomly choose a key within the KEYRANGE */
372   atomic
373   {
374     do
375     :: ( key < KEY_RANGE ) ->
376       if
377       :: key++;
378       :: break;
379       fi;
380     :: else -> break;
381     od;
382   }

384   /* Find the insertion point */
385   find( key, pred, predData, curr, currData, ROOT, result );

387   if
388   :: ( result != NOTFOUND ) -> goto doneI;
389   :: else -> skip;
390   fi;

392   /* allocate a node */
393   atomic
394   {
395     newNode = memNodesUsed;
396     memNodesUsed++;
397     memNodes[ newNode ].contents = memContentsUsed;
398     memContentsUsed++;
399     memContents[ memNodes[ newNode ].contents ].key = key;
400     memContents[ memNodes[ newNode ].contents ].left = NULL;
401     memContents[ memNodes[ newNode ].contents ].right = NULL;
402   }

404   /* allocate replacement contents for the parent */
405   atomic
406   {
407     newContents = memContentsUsed;
408     memContentsUsed++;
409     memContents[ newContents ].key = memContents[ currData ].key;
410     if
411     :: ( key < memContents[ currData ].key ) ->
412       memContents[ newContents ].left = newNode;
413       memContents[ newContents ].right = memContents[ currData ].
              right;
414     :: else ->
415       memContents[ newContents ].left = memContents[ currData ].
              left;
416       memContents[ newContents ].right = newNode;
417     fi;
418   }
```

```
420   /* CAS( &( curr->contents ), currData, newContents ) */
421   atomic
422   {
423     if
424     :: ( memNodes[ curr ].contents == currData ) ->
425       memNodes[ curr ].contents = newContents;
426       assert( ( keyMask & ( 1 << key ) ) == 0 );
427       keyMask = keyMask | ( 1 << key );
428     :: else -> goto doneI;
429     fi;
430   }

432   /* key successfully inserted */

434 doneI:
435   threadDone++;
436 }


439 /*+------------------------+*/
440 /*| Delete process         |*/
441 /*+------------------------+*/
442 active [DELETES] proctype delete()
443 {
444   /* Declare local variables */
445   byte key = 1, pred, curr, predData, currData, next, result,
        relocationContents, relocateOp, seenState, seenContents,
        replace, replaceData, lastRight, lastRightData, temp;
446   bool exciseDone, opSuccess;

448   /* Wait for setup to complete before beginning */
449   setupDone == true;

451   /* Randomly choose a key within the KEYRANGE */
452   atomic
453   {
454     do
455     :: ( key < KEY_RANGE ) ->
456       if
457       :: key++;
458       :: break;
459       fi;
460     :: else -> break;
461     od;
462   }

464   /* Find the node to be deleted */
465   find( key, pred, predData, curr, currData, ROOT, result );

467   if
468   :: ( result != FOUND ) -> goto doneD;
469   :: else -> skip;
470   fi;

472   if
473   :: ( ( memContents[ currData ].right == NULL ) || ( memContents
        [ currData ].left == NULL ) ) ->
474     /* Node has < 2 children */

476     /* CAS( &( curr->contents ), currData, GETMARKEDREF( currData
```

```
        ) ) */
477     atomic
478     {
479       if
480       :: ( memNodes[ curr ].contents == currData ) ->
481         memNodes[ curr ].contents = GETMARKEDREF( currData );
482         assert( ( keyMask & ( 1 << key ) ) != 0 );
483         keyMask = keyMask ^ ( 1 << key );
484       :: else -> goto doneD;
485       fi;
486     }

488     exciseNode( pred, predData, curr, currData, next, temp );

490   :: else ->
491     /* Node has 2 children */

493     /* Find the next largest node */
494     find( key, pred, predData, replace, replaceData, curr, result
          );

496     if
497     :: ( result != NOTFOUND ) || ( curr == replace ) -> goto
          doneD;
498     :: else -> skip;
499     fi;

501     /* allocate replacement contents for the node to be removed &
           a relocation structure for the operation */
502     atomic
503     {
504       relocationContents = memContentsUsed;
505       memContentsUsed++;
506       memContents[ relocationContents ].key = memContents[
            replaceData ].key;
507       memContents[ relocationContents ].left = memContents[
            currData ].left;
508       if
509       :: ( memContents[ currData ].right == replace ) ->
510         memContents[ relocationContents ].right = memContents[
              replaceData ].right;
511         exciseDone = true;
512       :: else ->
513         memContents[ relocationContents ].right = memContents[
              currData ].right;
514         exciseDone = false;
515       fi;

517       relocateOp = memRelocationsUsed;
518       memRelocationsUsed++;
519       memRelocations[ relocateOp ].state = ONGOING;
520       memRelocations[ relocateOp ].destination = curr;
521       memRelocations[ relocateOp ].oldContents = currData;
522       memRelocations[ relocateOp ].newContents =
            relocationContents;
523       memRelocations[ relocateOp ].failContents = replaceData;
524     }

526     /* CAS( &( replace->contents), replaceData, GETRELOCATIONREF(
           relocateOp ) ) */
```

```
527     atomic
528     {
529       if
530       :: ( memNodes[ replace ].contents == replaceData ) ->
531         memNodes[ replace ].contents = GETRELOCATIONREF(
              relocateOp );
532       :: else -> goto doneD;
533       fi;
534     }

536     processOp( relocateOp, opSuccess );

538     if
539     :: ( opSuccess ) ->
540       /* key successfully removed */
541       if
542       :: ( !exciseDone ) -> exciseNode( pred, predData, replace,
              replaceData, curr, temp );
543       :: else -> skip;
544       fi;
545     :: else ->
546       /* attempt to roll back changes */

548       /* CAS( &( replace->contents ), GETRELOCATIONREF(
              relocateOp ), replaceData ); */
549       atomic
550       {
551         if
552         :: ( memNodes[ replace ].contents == GETRELOCATIONREF(
                relocateOp ) ) ->
553           memNodes[ replace ].contents = replaceData;
554         :: else -> skip;
555         fi;
556       }
557     fi;
558   fi;

560 doneD:
561   threadDone++;
562 }
```

Appendix B

SPIN Model Checking Code for Single-Node BST

```
/* Parameters */
#define INSERTS        2
#define DELETES        2
#define KEY_RANGE      8
//#define FIXED_KEYS

/* Constants */
#define NULL           63
#define ROOT           0

#define FOUND          0
#define NOTFOUNDLEFT   1
#define NOTFOUNDRIGHT  2
#define ABORT          3

#define ONGOING        0
#define FAILED         1
#define SUCCESSFUL     2

#define NONE           0
#define MARKED         1
#define RELOCATE       2
#define HELPCAS        3

/* Structures */
typedef TreeNode
{
  byte key;
  byte op;
  byte left;
  byte right;
}

typedef HelpCASOp
{
  byte destination;
  bool isLeft;
```

```
38   byte oldVal;
39   byte newVal;
40 }

42 typedef RelocateOp
43 {
44   byte state;
45   byte destination;
46   byte destOp;
47   byte destKey;
48   byte newKey;
49 }

51 /* Macros */
52 #define GETFLAG( n )     ( ( ( n ) & 192 ) >> 6 )
53 #define SETFLAG( n, m ) ( ( n ) | ( ( m ) << 6 ) )
54 #define UNFLAG( n )      ( ( n ) & 63 )

56 #define ISNULL( n )      ( ( ( n ) & 64 ) != 0 )
57 #define SETNULL( n )     ( ( n ) | 64 )

59 /* Global variables */
60 TreeNode    memNodes[ 7 ];
61 byte        memNodesUsed = 0;
62 RelocateOp  memRelocate[ 2 ];
63 byte        memRelocateUsed = 0;
64 HelpCASOp   memHelpCAS[ 4 ];
65 byte        memHelpCASUsed = 0;
66 bool        setupDone = false;
67 hidden byte threadDone = 0;
68 short       keyMask = 0;

70 /* Variable Aliases */

72 /* Helper functions */

74 inline find( _key, _pred, _predData, _curr, _currData, _auxRoot,
       _result )
75 {
76   atomic
77   {
78     _result = NOTFOUNDRIGHT;
79     _curr = _auxRoot;
80     _currData = memNodes[ _curr ].op;
81   }

83   atomic
84   {
85     if
86     :: ( GETFLAG( _currData ) != NONE ) ->
87       assert( ( _auxRoot != ROOT ) || ( GETFLAG( _currData ) ==
              HELPCAS ) );
88       _result = ABORT;
89       next = SETNULL( NULL );
90     :: else ->
91       next = memNodes[ _curr ].right;
92       lastRight = _curr;
93       lastRightData = _currData;
94       lastClean = _curr;
95       lastCleanData = _currData;
```

```
96      fi;
97    }

99    do
100   :: ( !ISNULL( next ) ) ->
101     atomic
102     {
103       _pred = _curr;
104       _predData = _currData;
105       _curr = next;
106       _currData = memNodes[ _curr ].op;
107     }
108     currKey = memNodes[ _curr ].key;

110     atomic
111     {
112       if
113       :: ( _key == currKey ) ->
114         _result = FOUND;
115         break;
116       :: ( _key < currKey ) ->
117         _result = NOTFOUNDLEFT;
118         next = memNodes[ _curr ].left;
119       :: else -> /* ( key > currKey ) */
120         _result = NOTFOUNDRIGHT;
121         next = memNodes[ _curr ].right;
122         lastRight = _curr;
123         lastRightData = _currData;
124       fi;

126       if
127       :: ( GETFLAG( _currData ) == NONE ) ->
128         lastClean = _curr;
129         lastCleanData = _currData;
130       :: else -> skip;
131       fi;

133       /* Theory checking */
134       if
135       :: ( ISNULL( next ) && ( memNodes[ lastRight ].op ==
              lastRightData ) && ( GETFLAG( _currData ) == NONE ) && (
               memNodes[ _curr ].op == _currData ) && ( _auxRoot ==
              ROOT ) ) ->
136         assert( ( keyMask & ( 1 << _key ) ) == 0 );
137       :: else -> skip;
138       fi;
139     }

141   :: else -> break;
142   od;

144   if
145   :: ( memNodes[ lastRight ].op != lastRightData ) -> _result =
        ABORT;
146   :: else -> skip;
147   fi;

149   if
150   :: ( ( GETFLAG( _currData ) != NONE ) && ( _result != ABORT ) )
        ->
151     if
```

```
152     :: ( lastClean != _pred ) ->
153       currKey = memNodes[ lastClean ].key;
154       _curr = ( ( _key < currKey ) -> memNodes[ lastClean ].left
              : memNodes[ lastClean ].right );
155       if
156       :: ( memNodes[ lastClean ].op != lastCleanData ) -> _result
               = ABORT;
157       :: else -> _currData = memNodes[ _curr ].op;
158       fi;
159     :: else -> skip;
160     fi;

162     if
163     :: ( _result != ABORT ) ->
164       help( lastClean, lastCleanData, _curr, _currData, _result )
              ;
165       _result = ABORT;
166     :: else -> skip;
167     fi;
168   :: else -> skip;
169   fi;
170 }

172 inline HelpCASProcessOp( this )
173 {
174   /* CAS( isLeft ? &( destination->left ) : &( destination->right
          ), oldVal, newVal ) */
175   atomic
176   {
177     if
178     :: ( memHelpCAS[ this ].isLeft ) ->
179       if
180       :: ( memNodes[ memHelpCAS[ this ].destination ].left ==
              memHelpCAS[ this ].oldVal ) ->
181         memNodes[ memHelpCAS[ this ].destination ].left =
              memHelpCAS[ this ].newVal;
182       :: else -> skip;
183       fi;
184     :: else ->
185       if
186       :: ( memNodes[ memHelpCAS[ this ].destination ].right ==
              memHelpCAS[ this ].oldVal ) ->
187         memNodes[ memHelpCAS[ this ].destination ].right =
              memHelpCAS[ this ].newVal;
188       :: else -> skip;
189       fi;
190     fi;
191   }

193   /* CAS( &( destination->op ), SETFLAG( this, HELPCAS ), SETFLAG
         ( this, NONE ) ) */
194   atomic
195   {
196     if
197     :: ( memNodes[ memHelpCAS[ this ].destination ].op == SETFLAG
           ( this, HELPCAS ) ) ->
198       memNodes[ memHelpCAS[ this ].destination ].op = SETFLAG(
              this + 16, NONE );
199     :: else -> skip;
200     fi;
```

```
201   }
202 }

204 inline RelocateProcessOp( this, _result )
205 {
206   seenState = memRelocate[ this ].state;

208   if
209   :: ( seenState == ONGOING ) ->
210     /* VCAS( &( destination->op ), destOp, SETFLAG( this,
          RELOCATE ) ) */
211     atomic
212     {
213       seenOp = memNodes[ memRelocate[ this ].destination ].op;
214       if
215       :: ( memNodes[ memRelocate[ this ].destination ].op ==
            memRelocate[ this ].destOp ) ->
216         memNodes[ memRelocate[ this ].destination ].op = SETFLAG(
              this, RELOCATE );
217       :: else -> skip;
218       fi;
219     }

221     atomic
222     {
223       if
224       :: ( ( seenOp == memRelocate[ this ].destOp ) || ( seenOp
            == SETFLAG( this, RELOCATE ) ) ) ->
225         /* VCAS( &state, ONGOING, SUCCESSFUL ) */
226         if
227         :: ( memRelocate[ this ].state != FAILED ) ->
228           if
229           :: ( memRelocate[ this ].state == ONGOING ) ->
230             memRelocate[ this ].state = SUCCESSFUL;
231             /* key logically deleted */
232             printf( "Key␣%d␣logically␣deleted\n", memRelocate[
                  this ].destKey );
233             assert( ( keyMask & ( 1 << memRelocate[ this ].
                  destKey ) ) != 0 );
234             keyMask = keyMask ^ ( 1 << memRelocate[ this ].
                  destKey );
235           :: else -> skip;
236           fi;

238           seenState = SUCCESSFUL;
239         :: else -> seenState = FAILED;
240         fi;
241       :: else ->
242         /* VCAS( &state, ONGOING, FAILED ) */
243         if
244         :: ( memRelocate[ this ].state != SUCCESSFUL ) ->
245           memRelocate[ this ].state = FAILED;
246           seenState = FAILED;
247         :: else -> seenState = SUCCESSFUL;
248         fi;
249       fi;
250     }

252   :: else -> skip;
253   fi;
```

```
255   if
256   :: ( seenState == SUCCESSFUL ) ->
257     /* CAS( &( destination->key ), destKey, newKey ) */
258     atomic
259     {
260       if
261       :: ( memNodes[ memRelocate[ this ].destination ].key ==
              memRelocate[ this ].destKey ) ->
262         memNodes[ memRelocate[ this ].destination ].key =
              memRelocate[ this ].newKey;
263       :: else -> skip;
264       fi;
265     }

267     /* CAS( &( destination->op ), SETFLAG( this, RELOCATE ),
           SETFLAG( this, NONE ) ) */
268     atomic
269     {
270       if
271       :: ( memNodes[ memRelocate[ this ].destination ].op ==
              SETFLAG( this, RELOCATE ) ) ->
272         memNodes[ memRelocate[ this ].destination ].op = SETFLAG(
                this + 32, NONE );
273       :: else -> skip;
274       fi;
275     }
276   :: else -> skip;
277   fi;

279   _result = seenState == SUCCESSFUL;
280 }

282 inline help( _pred, _predData, _curr, _currData, _result )
283 {
284   _result = false;

286   if
287   :: ( GETFLAG( _currData ) == RELOCATE ) ->
288     helpRelocate( _pred, _predData, _curr, _currData, _result );
289   :: ( GETFLAG( _currData ) == MARKED ) ->
290     helpMarked( _pred, _predData, _curr, _currData, _result );
291   :: ( GETFLAG( _currData ) == HELPCAS ) ->
292     HelpCASProcessOp( UNFLAG( _currData ) );
293   :: else -> skip;
294   fi;
295 }

297 inline helpRelocate( _pred, _predData, _curr, _currData, _result
      )
298 {
299   temp = UNFLAG( _currData );
300   RelocateProcessOp( temp, _result );

302   if
303   :: ( memRelocate[ temp ].destination != _curr ) ->
304     /* CAS( &( curr->op ), currData, SETFLAG( op, result ? MARKED
            : NONE ) ) */
305     atomic
```

```
306     {
307       if
308       :: ( memNodes[ _curr ].op == _currData ) ->
309         memNodes[ _curr ].op = SETFLAG( temp, _result -> MARKED :
              NONE )
310       :: else -> skip;
311       fi;
312     }
313
314     if
315     :: ( _result ) ->
316       if
317       :: ( memRelocate[ temp ].destination == _pred ) ->
318         _predData = SETFLAG( temp, NONE );
319       :: else -> skip;
320       fi;
321
322       helpMarked( _pred, _predData, _curr, SETFLAG( temp, MARKED
            ), _result );
323       _result = true;
324     :: else -> skip;
325     fi;
326   :: else -> skip;
327   fi;
328 }
329
330 inline helpMarked( _pred, _predData, _curr, _currData, _result )
331 {
332   if
333   :: ( GETFLAG( _predData ) == NONE ) ->
334     if
335     :: ( ISNULL( memNodes[ _curr ].left ) ) ->
336       if
337       :: ( ISNULL( memNodes[ _curr ].right ) ) ->
338         temp = SETNULL( _curr );
339       :: else ->
340         temp = memNodes[ _curr ].right;
341       fi;
342     :: else ->
343       temp = memNodes[ _curr ].left;
344     fi;
345     atomic
346     {
347       /* allocate helpCAS operation structure */
348       newOp = memHelpCASUsed;
349       memHelpCASUsed++;
350       memHelpCAS[ newOp ].destination = _pred;
351       memHelpCAS[ newOp ].isLeft = memNodes[ _pred ].left ==
            _curr;
352       memHelpCAS[ newOp ].oldVal = _curr;
353       memHelpCAS[ newOp ].newVal = temp;
354
355       /* verify */
356       if
357       :: ( ( memNodes[ _pred ].op == _predData ) && !memHelpCAS[
            newOp ].isLeft ) ->
358         assert( memNodes[ _pred ].right == _curr );
359       :: else -> skip;
360       fi;
361     }
```

```
363     _result = false;
364     /* CAS( &( pred->op ), predData, SETFLAG( newOp, HELPCAS ) )
           */
365     atomic
366     {
367       if
368       :: ( memNodes[ _pred ].op == _predData ) ->
369         memNodes[ _pred ].op = SETFLAG( newOp, HELPCAS );
370         _result = true;
371       :: else -> skip;
372       fi;
373     }

375     if
376     :: ( _result ) ->
377       HelpCASProcessOp( newOp );
378     :: else -> skip;
379     fi;

381   :: else -> skip;
382   fi;
383 }

385 /*+------------------------+*/
386 /*| Setup process          |*/
387 /*+------------------------+*/
388 active proctype setup()
389 {
390   if
391   :: ( setupDone == false ) ->
392     atomic
393     {
394       memNodesUsed++;
395       memNodes[ ROOT ].op = SETFLAG( NULL, NONE );
396       memNodes[ ROOT ].key = 0;
397       memNodes[ ROOT ].left = SETNULL( NULL );
398       memNodes[ ROOT ].right = 1;

400       /* Insert some nodes */

402       memNodesUsed++;
403       memNodes[ 1 ].op = SETFLAG( NULL, NONE );
404       memNodes[ 1 ].key = 1;
405       memNodes[ 1 ].left = SETNULL( NULL );
406       memNodes[ 1 ].right = 2;
407       keyMask = keyMask | ( 1 << 1 );

409       memNodesUsed++;
410       memNodes[ 2 ].op = SETFLAG( NULL, NONE );
411       memNodes[ 2 ].key = 3;
412       memNodes[ 2 ].left = 3;
413       memNodes[ 2 ].right = 4;
414       keyMask = keyMask | ( 1 << 3 );

416       memNodesUsed++;
417       memNodes[ 3 ].op = SETFLAG( NULL, NONE );
418       memNodes[ 3 ].key = 2;
419       memNodes[ 3 ].left = SETNULL( NULL );
```

```
420         memNodes[ 3 ].right = SETNULL( NULL );
421         keyMask = keyMask | ( 1 << 2 );

423         memNodesUsed++;
424         memNodes[ 4 ].op = SETFLAG( NULL, NONE );
425         memNodes[ 4 ].key = 6;
426         memNodes[ 4 ].left = SETNULL( NULL );
427         memNodes[ 4 ].right = SETNULL( NULL );
428         keyMask = keyMask | ( 1 << 6 );

430         setupDone = true;
431       }
432     :: else -> skip;
433     fi;

435     /* Wait for all threads */
436     threadDone == ( INSERTS + DELETES );
437 }


440 /*+------------------------+*/
441 /*| Insert process         |*/
442 /*+------------------------+*/
443 active [INSERTS] proctype insert()
444 {
445     /* Declare local variables */
446     byte key = 1, pred, curr, predData, currData, currKey, next,
           result, newNode = NULL, newOp = NULL, seenState, seenOp,
           lastRight, lastRightData, lastClean, lastCleanData, temp;

448     /* Wait for setup to complete before beginning */
449     setupDone == true;

451     /* Randomly choose a key within the KEYRANGE */
452     atomic
453     {
454 #ifndef FIXED_KEYS
455       do
456       :: ( key < KEY_RANGE ) ->
457         if
458         :: key++;
459         :: break;
460         fi;
461       :: else -> break;
462       od;
463 #else
464       if
465       :: ( _pid == 1 ) -> key = 5;
466       :: ( _pid == 2 ) -> key = 4;
467       :: else -> skip;
468       fi;
469 #endif
470     }

472     /* Find the insertion point */
473     find( key, pred, predData, curr, currData, ROOT, result );

475     if
476     :: ( ( result != NOTFOUNDLEFT ) && ( result != NOTFOUNDRIGHT )
           ) -> goto doneI;
```

```
477   :: else -> skip;
478   fi;

480   /* allocate a node */
481   atomic
482   {
483     newNode = memNodesUsed;
484     memNodesUsed++;
485     memNodes[ newNode ].op = SETFLAG( NULL, NONE );
486     memNodes[ newNode ].key = key;
487     memNodes[ newNode ].left = SETNULL( NULL );
488     memNodes[ newNode ].right = SETNULL( NULL );

490   /* allocate helpCAS operation structure */
491     newOp = memHelpCASUsed;
492     memHelpCASUsed++;
493     memHelpCAS[ newOp ].destination = curr;
494     memHelpCAS[ newOp ].isLeft = result == NOTFOUNDLEFT;
495     memHelpCAS[ newOp ].oldVal = ( ( result == NOTFOUNDLEFT ) ->
          memNodes[ curr ].left : memNodes[ curr ].right );
496     memHelpCAS[ newOp ].newVal = newNode;
497   }

499   /* CAS( &( curr->op ), currData, SETFLAG( newOp, HELPCAS ) ) */
500   atomic
501   {
502     if
503     :: ( memNodes[ curr ].op == currData ) ->
504       memNodes[ curr ].op = SETFLAG( newOp, HELPCAS );
505       /* key logically inserted */
506       printf( "Key␣%d␣logically␣inserted\n", key );
507       assert( ( keyMask & ( 1 << key ) ) == 0 );
508       keyMask = keyMask | ( 1 << key );
509     :: else -> goto doneI;
510     fi;
511   }

513   HelpCASProcessOp( newOp );

515 doneI:
516   threadDone++;
517 }


520 /*+------------------------+*/
521 /*| Delete process         |*/
522 /*+------------------------+*/
523 active [DELETES] proctype delete()
524 {
525   /* Declare local variables */
526   byte key = 1, pred, curr, predData, currData, currKey, next,
        result, relocOp, seenState, seenOp, newOp = NULL, replace,
        replaceData, lastRight, lastRightData, lastClean,
        lastCleanData, temp;

528   /* Wait for setup to complete before beginning */
529   setupDone == true;

531   /* Randomly choose a key within the KEYRANGE */
532   atomic
```

```
533   {
534 #ifndef FIXED_KEYS
535     do
536     :: ( key < KEY_RANGE ) ->
537       if
538       :: key++;
539       :: break;
540       fi;
541     :: else -> break;
542     od;
543 #else
544     if
545     :: ( ( _pid - INSERTS ) == 1 ) -> key = 3;
546     :: ( ( _pid - INSERTS ) == 2 ) -> key = 6;
547     :: else -> skip;
548     fi;
549 #endif
550   }

552   /* Find the node to be deleted */
553   find( key, pred, predData, curr, currData, ROOT, result );

555   if
556   :: ( result != FOUND ) -> goto doneD;
557   :: else -> skip;
558   fi;

560   if
561   :: ( ISNULL( memNodes[ curr ].right ) || ISNULL( memNodes[ curr
        ].left ) ) ->
562     /* Node has < 2 children */

564     /* CAS( &( curr->op ), currData, SETFLAG( currData, MARKED )
        ) */
565     atomic
566     {
567       if
568       :: ( memNodes[ curr ].op == currData ) ->
569         memNodes[ curr ].op = SETFLAG( currData, MARKED );
570         /* key logically deleted */
571         printf( "Key␣%d␣logically␣deleted\n", key );
572         assert( ( keyMask & ( 1 << key ) ) != 0 );
573         keyMask = keyMask ^ ( 1 << key );
574       :: else -> goto doneD;
575       fi;
576     }

578     help( pred, predData, curr, SETFLAG( currData, MARKED ),
          result );

580   :: else ->
581     /* Node has 2 children */

583     /* Find the next largest node */
584     find( key, pred, predData, replace, replaceData, curr, result
          );

586     if
587     :: ( ( result != NOTFOUNDLEFT ) && ( result != NOTFOUNDRIGHT
        ) ) || ( curr == replace ) -> goto doneD;
```

```
588       :: else -> skip;
589       fi;

591       /* allocate a relocation structure for the operation */
592       atomic
593       {
594         relocOp = memRelocateUsed;
595         memRelocateUsed++;
596         memRelocate[ relocOp ].state = ONGOING;
597         memRelocate[ relocOp ].destination = curr;
598         memRelocate[ relocOp ].destOp = currData;
599         memRelocate[ relocOp ].destKey = key;
600         memRelocate[ relocOp ].newKey = memNodes[ replace ].key;
601       }

603       /* CAS( &( replace->op ), replaceData, SETFLAG( relocOp,
              RELOCATE ) ) */
604       atomic
605       {
606         if
607         :: ( memNodes[ replace ].op == replaceData ) ->
608           memNodes[ replace ].op = SETFLAG( relocOp, RELOCATE );
609         :: else -> goto doneD;
610         fi;
611       }

613       help( pred, predData, replace, SETFLAG( relocOp, RELOCATE ),
              result );
614     fi;

616   doneD:
617     threadDone++;
618   }
```

Appendix C

Optimised Traversal Code for Single-Node BST

```
1  int find( int key, Node*& pred, Operation*& predOp, Node*& curr,
       Operation*& currOp, Node* auxRoot )
2  {
3    int result, currKey;
4    Node* next, * lastRight, * lastClean;
5    Operation* lastRightOp, * lastCleanOp;

7  retry:
8    result = NOTFOUND_R;
9    curr = auxRoot;
10   currOp = curr->op;
11   if ( GETFLAG( currOp ) != NONE )
12   {
13     if ( auxRoot == &root )
14     {
15       helpChildCAS( ( ChildCASOp* )UNFLAG( currOp ), curr );
16       goto retry;
17     }
18     return ABORT;
19   }

21   next = curr->right;
22   lastRight = lastClean = curr;
23   lastRightOp = lastCleanOp = currOp;

25   while ( !ISNULL( next ) )
26   {
27     pred = curr;
28     predOp = currOp;
29     curr = next;
30     currOp = curr->op;
31     currKey = curr->key;

33     if ( key == currKey )
34     {
35       result = FOUND;
```

```
36        break;
37      }
38      else if ( key < currKey )
39      {
40        result = NOTFOUND_L;
41        next = curr->left;
42      }
43      else // ( key > currKey )
44      {
45        result = NOTFOUND_R;
46        next = curr->right;
47        if ( GETFLAG( currOp ) != MARK )
48        {
49          lastRight = curr;
50          lastRightOp = currOp;
51        }
52      }

54      if ( GETFLAG( currOp ) == NONE )
55      {
56        lastClean = curr;
57        lastCleanOp = currOp;
58      }
59    }

61    // check lastRight
62    if ( ( result != FOUND ) && ( lastRightOp != lastRight->op ) )
63    {
64      goto retry;
65    }

67    if ( GETFLAG( currOp ) != NONE )
68    {
69      if ( lastClean != pred )
70      {
71        curr = key < lastClean->key ? lastClean->left : lastClean->
            right;
72        if ( ISNULL( curr ) || ( lastCleanOp != lastClean->op ) )
73        {
74          goto retry;
75        }
76        currOp = curr->op;

78      }
79      help( lastClean, lastCleanOp, curr, currOp );
80      goto retry;
81    }

83    return result;
84  }

86  bool contains( int key )
87  {
88    Node* curr, * next, * lastRight, * lastUnMarked;
89    Operation* currOp, * lastRightOp, * lastUnMarkedOp;
90    int currKey;
91    bool found;

93  contains_retry:
94    found = false;
```

```
95    curr = &root;
96    currOp = root.op;
97    lastUnMarked = lastRight = curr;
98    lastUnMarkedOp = lastRightOp = currOp;
99    next = root.right;

101   while ( !ISNULL( next ) )
102   {
103     curr = next;
104     currOp = curr->op;
105     currKey = curr->key;

107     if ( key < currKey )
108     {
109       next = curr->left;
110     }
111     else if ( key > currKey )
112     {
113       next = curr->right;
114       lastRight = curr;
115       lastRightOp = currOp;
116     }
117     else
118     {
119       found = true;
120       break;
121     }

123     if ( GETFLAG( currOp ) != MARK )
124     {
125       lastUnMarked = curr;
126       lastUnMarkedOp = currOp;
127     }
128     else
129     {
130       if ( lastUnMarked->op != lastUnMarkedOp ) goto
              contains_retry;
131     }
132   }

134   if ( GETFLAG( currOp ) == MARK )
135   {
136     if ( lastUnMarked->op != lastUnMarkedOp ) goto contains_retry
          ;
137     else return false;
138   }

140   if ( found )
141   {
142     if (   ( GETFLAG( currOp ) == RELOCATE ) &&
143            ( ( ( RelocateOpPtr )currOp )->state == RelocateOp::
                 SUCCESSFUL ) &&
144            ( ( ( RelocateOpPtr )currOp )->destination != curr ) &&
145            ( ( ( RelocateOpPtr )currOp )->destination->key >= key
                 ) )
146     {
147       goto contains_retry;
148     }

150     return true;
```

```
151   }
152   else
153   {
154     if ( lastRight->op != lastRightOp ) goto contains_retry;

156     if ( GETFLAG( currOp ) == CHILDCAS )
157     {
158       Node* n = ( ( ChildCASOp* )UNFLAG( currOp ) )->newVal;
159       if ( ISNULL( n ) ) return false;
160       else return n->key == key;
161     }

163     return false;
164   }
165 }
```

Appendix D

Single-Node BST with Hazard Pointers

```
1  class Node
2  {
3    int volatile         key;
4    Operation* volatile  op;
5    Node* volatile       left;
6    Node* volatile       right;
7  }

9  class Operation
10 {
11   enum types
12   {
13     CHILDCAS,
14     RELOCATE
15   } type;
16 }

18 class ChildCASOp : public Operation
19 {
20   Operation*  tag;
21   bool        isLeft;
22   Node*       oldVal;
23   Node*       newVal;
24 }

26 class RelocateOp : public Operation
27 {
28   Operation*  sourceTag;
29   Node*       destination;
30   Operation*  destTag;
31   int         removeKey;
32   int         replaceKey;

34   enum relocateState
35   {
36     ONGOING,
37     SUCCESSFUL,
```

```
38      SUCCESSFUL_CLEAN,
39      FAILED
40    } volatile  state;
41  }

43  int find( int key, Node*& pred, Operation*& predTag, Node*& curr,
        Operation*& currTag, Node* auxRoot, void* volatile* hp )
44  {
45    int result;
46    int currKey;
47    Node* volatile* next;
48    Node* n, * lastRight;
49    Operation* lastRightTag;

51  retry:
52    result = NOTFOUNDRIGHT;
53    curr = auxRoot;
54    currTag = auxRoot->op;
55    if ( !ISTAG( currTag ) )
56    {
57      hp[ 0 ] = currTag;
58      if ( currTag != auxRoot->op ) goto retry;
59      if ( auxRoot == &root )
60      {
61        helpChildCAS( ( ChildCASOp* )currTag, curr );
62        goto retry;
63      }
64      return ABORT;
65    }

67    next = &curr->right;
68    lastRight = curr;
69    lastRightTag = currTag;

71    while ( true )
72    {
73      n = *next;
74      if ( ISNULL( n ) ) break;

76      hp[ 0 ] = n;
77      if ( curr->op != currTag ) goto retry;

79      hp[ 2 ] = curr;
80      pred = curr;
81      predTag = currTag;
82      hp[ 1 ] = n;
83      curr = n;
84      currTag = curr->op;

86      if ( !ISTAG( currTag ) )
87      {
88        if ( !ISMARKED( currTag ) )
89        {
90          hp[ 0 ] = currTag;
91          if ( currTag != curr->op ) goto retry;
92        }

94        help( pred, predTag, curr, currTag, hp + 3 );
95        goto retry;
96      }
```

```
98      currKey = curr->key;

100     if ( key < currKey )
101     {
102       result = NOTFOUNDLEFT;
103       next = &curr->left;
104     }
105     else if ( key > currKey )
106     {
107       result = NOTFOUNDRIGHT;
108       next = &curr->right;
109       hp[ 3 ] = curr;
110       lastRight = curr;
111       lastRightTag = currTag;
112     }
113     else // ( key == currKey )
114     {
115       result = FOUND;
116       break;
117     }

119   }

121   // check lastRight
122   if ( ( result != FOUND ) && ( lastRightTag != lastRight->op ) )
123   {
124     goto retry;
125   }

127   if ( curr->op != currTag )
128   {
129     goto retry;
130   }

132   hp[ 3 ] = NULL;
133   return result;
134 }

136 bool contains( int key, void* volatile* hp )
137 {
138   Node* volatile* next;
139   Node* curr, * n, * lastRight, * lastUnMarked;
140   Operation* currTag, * lastRightTag, * lastUnMarkedTag;
141   int currKey;
142   bool found, result;

144 contains_retry:
145   found = false;
146   curr = &root;
147   currTag = root.op;
148   lastUnMarked = lastRight = curr;
149   lastUnMarkedTag = lastRightTag = currTag;
150   next = &root.right;

152   while ( true )
153   {
154     n = *next;
155     if ( ISNULL( n ) ) break;
156     hp[ 0 ] = n;
```

```
157     if ( curr->op != currTag ) goto contains_retry;

159     hp[ 1 ] = n;
160     curr = n;
161     currTag = curr->op;
162     currKey = curr->key;

164     if ( key < currKey )
165     {
166       next = &curr->left;
167     }
168     else if ( key > currKey )
169     {
170       next = &curr->right;
171       if ( !ISMARKED( currTag ) )
172       {
173         hp[ 3 ] = curr;
174         lastRight = curr;
175         lastRightTag = currTag;
176       }
177       if ( hp[ 2 ] == NULL ) hp[ 2 ] = curr;

179     }
180     else // ( key == currKey )
181     {
182       found = true;
183       break;
184     }

186     if ( ISMARKED( currTag ) )
187     {
188       if ( lastUnMarked->op != lastUnMarkedTag ) goto
              contains_retry;
189     }
190     else
191     {
192       hp[ 2 ] = curr;
193       lastUnMarked = curr;
194       lastUnMarkedTag = currTag;
195     }
196   }

198   if ( ISMARKED( currTag ) )
199   {
200     if ( lastUnMarked->op != lastUnMarkedTag ) goto
          contains_retry;
201     else result = false;
202   }
203   else if ( found )
204   {
205     if ( !ISTAG( currTag ) )
206     {
207       hp[ 1 ] = currTag;
208       if ( curr->op != currTag ) goto contains_retry;

210       if ( currTag->type == Operation::RELOCATE )
211       {
212         if ( ( ( RelocateOp* )currTag )->destination != curr )
213         {
```

```
214             if ( ( ( RelocateOp* )currTag )->state == RelocateOp::
                    SUCCESSFUL_CLEAN )
215             {
216               goto contains_retry;
217             }
218           }
219           else if ( ( ( RelocateOp* )currTag )->removeKey == key )
220           {
221             return false;
222           }
223         }
224       }

226       result = true;
227     }
228     else
229     {
230       if ( lastRight->op != lastRightTag ) goto contains_retry;

232       if ( !ISTAG( currTag ) && ( currTag->type == Operation::
             CHILDCAS ) )
233       {
234         hp[ 1 ] = currTag;
235         if ( curr->op != currTag ) goto contains_retry;

237         n = ( ( ChildCASOp* )currTag )->newVal;
238         if ( !ISNULL( n ) )
239         {
240           hp[ 2 ] = n;
241           if ( curr->op != currTag ) goto contains_retry;

243           result = n->key == key;
244         }
245         else result = false;
246       }
247       else result = false;
248     }

250   hp[ 3 ] = hp[ 2 ] = hp[ 1 ] = hp[ 0 ] = NULL;
251   return result;
252 }

254 bool add( int key, void* volatile* hp )
255 {
256   Node* pred, * curr;
257   Operation* predTag, * currTag;
258   Node* newNode = NULL;
259   ChildCASOp* casOp = NULL;
260   int result;

262   while ( true )
263   {
264     result = find( key, pred, predTag, curr, currTag, &root, hp )
          ;
265     if ( result == FOUND )
266     {
267       break;
268     }
```

```
270     if ( !newNode ) newNode = allocNode( key );
271     if ( !casOp )   casOp = allocChildCASOp( currTag, result ==
            NOTFOUNDLEFT, result == NOTFOUNDLEFT ? curr->left : curr->
            right, newNode );
272     else
273     {
274       casOp->tag = currTag;
275       casOp->isLeft = result == NOTFOUNDLEFT;
276       casOp->oldVal = result == NOTFOUNDLEFT ? curr->left : curr
              ->right;
277     }

279     hp[ 0 ] = casOp;
280     if ( CASP( &( curr->op ), currTag, casOp ) )
281     {
282       helpChildCAS( casOp, curr );
283       hp[ 2 ] = hp[ 1 ] = hp[ 0 ] = NULL;
284       return true;
285     }
286   }

288   if ( newNode )  retire( newNode );
289   if ( casOp )    retire( casOp );

291   hp[ 2 ] = hp[ 1 ] = hp[ 0 ] = NULL;
292   return false;
293 }

295 bool remove( int key, void* volatile* hp )
296 {
297   Node* pred, * curr, * replace;
298   Operation* predTag, * currTag, * replaceTag;
299   RelocateOp* relocOp = NULL;

301   while ( true )
302   {
303     if ( find( key, pred, predTag, curr, currTag, &root, hp ) !=
          FOUND )
304     {
305       break;
306     }

308     if ( ISNULL( curr->right ) || ISNULL( curr->left ) )
309     {
310       // Node has < 2 children
311       if ( CASP( &( curr->op ), currTag, MARK ) )
312       {
313         helpMarked( pred, predTag, curr, hp + 3 );
314         hp[ 3 ] = hp[ 2 ] = hp[ 1 ] = hp[ 0 ] = NULL;
315         if ( relocOp ) retire( relocOp );
316         return true;
317       }
318     }
319     else
320     {
321       // Node has 2 children
322       if (  ( find( key, pred, predTag, replace, replaceTag, curr
              , hp + 1 ) != ABORT ) &&
323         ( curr->op == currTag ) )
```

```
324       {
325         if ( !relocOp ) relocOp = allocRelocateOp( replaceTag,
              curr, currTag, key, replace->key );
326         else
327         {
328           relocOp->sourceTag = replaceTag;
329           relocOp->destination = curr;
330           relocOp->destTag = currTag;
331           relocOp->removeKey = key;
332           relocOp->replaceKey = replace->key;
333         }

335         hp[ 1 ] = relocOp;
336         if ( CASP( &( replace->op ), replaceTag, relocOp ) )
337         {
338           if ( helpRelocate( relocOp, pred, predTag, replace, hp
                + 4 ) )
339           {
340             hp[ 3 ] = hp[ 2 ] = hp[ 1 ] = hp[ 0 ] = NULL;
341             return true;
342           }
343           relocOp = NULL;
344         }
345       }
346     }
347   }

349   hp[ 3 ] = hp[ 2 ] = hp[ 1 ] = hp[ 0 ] = NULL;
350   if ( relocOp ) retire( relocOp );
351   return false;
352 }

354 void help( Node* pred, Operation* predTag, Node* curr, Operation*
      currOp, void* volatile* hp )
355 {
356   if ( !ISTAG( currOp ) )
357   {
358     if ( currOp == MARK )
359     {
360       helpMarked( pred, predTag, curr, hp );
361     }
362     else
363     {
364       if ( currOp->type == Operation::CHILDCAS )
365       {
366         helpChildCAS( ( ChildCASOp* )currOp, curr );
367       }
368       else // ( currOp->type == Operation::RELOCATE )
369       {
370         RelocateOp* op = ( RelocateOp* )currOp;
371         Node* dest = op->destination;
372         if ( dest != curr )
373         {
374           hp[ 0 ] = dest;
375         }

377         helpRelocate( op, pred, predTag, curr, hp );
```

```
379         hp[ 0 ] = NULL;
380       }
381     }
382   }
383 }

385 void helpChildCAS( ChildCASOp* op, Node* destination )
386 {
387   Node* old = op->oldVal;
388   CASP( op->isLeft ? &( destination->left ) : &( destination->
        right ), old, op->newVal );

390   if ( CASP( &( destination->op ), op, INCREMENT_TAG( op->tag ) )
         )
391   {
392     if ( !ISNULL( old ) )
393     {
394       retire( old );
395     }

397     retire( op );
398   }
399 }

401 bool helpRelocate( RelocateOp* op, Node* pred, Operation* predTag
      , Node* curr, void* volatile* hp )
402 {
403   RelocateOp::relocateState seenState = op->state;
404   Node* dest = op->destination;
405   Operation* destTag = op->destTag;

407   if ( seenState == RelocateOp::ONGOING )
408   {
409     Operation* seenTag = ( Operation* )VCASP( &( dest->op ),
          destTag, op );

411     if ( ( seenTag == destTag ) || ( seenTag == op ) )
412     {
413       seenState = ( RelocateOp::relocateState )VCASL( &( op->
            state ), RelocateOp::ONGOING, RelocateOp::SUCCESSFUL );
414       if ( seenState == RelocateOp::ONGOING ) seenState =
            RelocateOp::SUCCESSFUL;
415     }
416     else
417     {
418       seenState = ( RelocateOp::relocateState )VCASL( &( op->
            state ), RelocateOp::ONGOING, RelocateOp::FAILED );
419     }
420   }

422   if ( seenState == RelocateOp::SUCCESSFUL )
423   {
424     CASL( &( dest->key ), op->removeKey, op->replaceKey );
425     CASP( &( dest->op ), op, INCREMENT_TAG( destTag ) );
426     CASL( &( op->state ), RelocateOp::SUCCESSFUL, RelocateOp::
          SUCCESSFUL_CLEAN );
427     seenState = RelocateOp::SUCCESSFUL_CLEAN;
428   }
```

```
430   bool success = seenState == RelocateOp::SUCCESSFUL_CLEAN;

432   if ( dest != curr )
433   {
434     if ( CASP( &( curr->op ), op, success ? MARK : INCREMENT_TAG(
          op->sourceTag ) ) )
435     {
436       retire( op );
437     }

439     if ( success )
440     {
441       if ( dest == pred ) predTag = ( Operation* )INCREMENT_TAG(
            destTag );
442       helpMarked( pred, predTag, curr, hp );
443     }
444   }

446   return success;
447 }

449 void helpMarked( Node* pred, Operation* predTag, Node* curr, void
     * volatile* hp )
450 {
451   Node* newRef = ISNULL( curr->left ) ? ( ISNULL( curr->right ) ?
        ( Node* )INCREMENT_TAG( predTag ) : curr->right ) : curr->
       left;
452   ChildCASOp* casOp = allocChildCASOp( predTag, curr == pred->
       left, curr, newRef );

454   hp[ 0 ] = casOp;
455   if ( CASP( &( pred->op ), predTag, casOp ) )
456   {
457     helpChildCAS( casOp, pred );
458   }
459   else
460   {
461     retire( casOp );
462   }

464   hp[ 0 ] = NULL;
465 }
```

References

Ole Agesen, David L. Detlefs, Christine H. Flood, Alexander T. Garthwaite, Paul A. Martin, Nir N. Shavit, and Guy L. Steele Jr. DCAS-based concurrent deques. In *Proceedings of the 12th ACM symposium on Parallel Algorithms and Architectures*, SPAA '00, pages 137–146, Bar Harbor, ME, USA, July 2000. ACM.

James H. Anderson and Mark Moir. Universal constructions for multi-object operations. In *Proceedings of the 14th ACM Symposium on Principles of Distributed Computing*, PODC '95, pages 184–193, Ottowa, ON, Canada, August 1995. ACM.

Andrea Arcangeli, Mingming Cao, Paul E. McKenney, and Dipankar Sarma. Using read-copy-update techniques for system v ipc in the linux 2.5 kernel. In *USENIX Annual Technical Conference, FREENIX Track*, pages 297–309, San Antonio, TX, USA, June 2003.

Greg Barnes. A method for implementing lock-free shared-data structures. In *Proceedings of the 5th ACM Symposium on Parallel Algorithms and Architectures*, SPAA '93, pages 261–270, Velen, Germany, June 1993. ACM.

Nathan G. Bronson, Jared Casper, Hassan Chafi, and Kunle Olukotun. A practical concurrent binary search tree. In *Proceedings of the 15th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '10, pages 257–268, Bangalore, India, January 2010. ACM.

David L. Detlefs, Christine H. Flood, Alexander T. Garthwaite, Paul A. Martin, Nir N. Shavit, and Guy L. Steele Jr. Even better DCAS-based concurrent deques. In *Proceedings of the 14th International Conference on Distributed Computing*, DISC '00, pages 59–73, Toledo, Spain, October 2000. Springer-Verlag.

David L. Detlefs, Paul A. Martin, Mark Moir, and Guy L. Steele Jr. Lock-free reference

counting. In *Proceedings of the 20th ACM Symposium on Principles of Distributed Computing*, PODC '01, pages 190–199, Newport, RI, USA, August 2001. ACM.

David L. Detlefs, Paul A. Martin, Mark Moir, and Guy L. Steele Jr. Lock-free reference counting. *Distributed Computing*, 15(4):255–271, December 2002.

Simon Doherty, David L. Detlefs, Lindsay Groves, Christine H. Flood, Victor Luchangco, Paul A. Martin, Mark Moir, Nir Shavit, and Guy L. Steele Jr. DCAS is not a silver bullet for nonblocking algorithm design. In *Proceedings of the 16th ACM Symposium on Parallelism in Algorithms and Architectures*, SPAA '04, pages 216–224, Barcelona, Spain, June 2004. ACM.

Faith Ellen, Panagiota Fatourou, Eric Ruppert, and Franck van Breugel. Non-blocking binary search trees. In *Proceedings of the 29th ACM SIGACT-SIGOPS Symposium on Principles of Distributed Computing*, PODC '10, pages 131–140, Zurich, Switzerland, July 2010. ACM.

Mikhail Fomitchev. *Lock-free linked lists and skip lists.* PhD thesis, Department of Computer Science and Engineering, York University, Toronto, ON, Canada, November 2003.

Mikhail Fomitchev and Eric Ruppert. Lock-free linked lists and skip lists. In *Proceedings of the 23nd ACM Symposium on Principles of Distributed Computing*, PODC '04, pages 50–59, St. John's, NL, Canada, July 2004. ACM.

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

Keir Fraser and Tim Harris. Concurrent programming without locks. *ACM Transactions on Computer Systems*, 25(2), May 2007.

Michael Greenwald and David Cheriton. The synergy between non-blocking synchronization and operating system structure. In *Proceedings of the 2nd USENIX symposium on Operating Systems Design and Implementation*, OSDI '96, pages 123–136, Seattle, WA, USA, October 1996. ACM.

Michael B. Greenwald. *Non-blocking synchronization and system design.* PhD thesis, Department of Computer Science, Stanford University, CA, USA, August 1999.

Timothy L. Harris. A pragmatic implementation of non-blocking linked-lists. In *Pro-*

ceedings of the 15th International Conference on Distributed Computing, DISC '01, pages 300–314, Lisbon, Portugal, October 2001. Springer-Verlag.

Timothy L. Harris, Keir Fraser, and Ian A. Pratt. A practical multi-word compare-and-swap operation. In *Proceedings of the 16th International Conference on Distributed Computing*, DISC '02, pages 265–279, Toulouse, France, October 2002. Springer-Verlag.

Thomas E. Hart, Paul E. McKenney, and Angela D. Brown. Making lockless synchronization fast: Performance implications of memory reclamation. In *Proceedings of the 20th International Parallel and Distributed Processing Symposium*, IPDPS '06, page 10, Rhodes Island, Greece, April 2006. IEEE Computer Society.

Thomas E. Hart, Paul E. McKenney, Angela D. Brown, and Jonathan Walpole. Performance of memory reclamation for lockless synchronization. *Journal of Parallel and Distributed Computing*, 67(12):1270–1285, December 2007.

Steve Heller, Maurice Herlihy, Victor Luchangco, Mark Moir, William N. Scherer III, and Nir Shavit. A lazy concurrent list-based set algorithm. In *Proceedings of the 9th International Conference on Principles of Distributed Systems*, OPODIS '05, pages 3–16, Pisa, Italy, December 2005. Springer-Verlag.

Maurice Herlihy. A methodology for implementing highly concurrent data structures. In *Proceedings of the 2nd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPOPP '90, pages 197–206, Seattle, WA, USA, March 1990. ACM.

Maurice Herlihy. Wait-free synchronization. *ACM Transactions on Programming Languages and Systems*, 13(1):124–149, January 1991.

Maurice Herlihy. A methodology for implementing highly concurrent data objects. *ACM Transactions on Programming Languages and Systems*, 15(5):745–770, November 1993.

Maurice Herlihy and J. Eliot B. Moss. Transactional memory: architectural support for lock-free data structures. In *Proceedings of the 20th Annual International Symposium on Computer Architecture*, ISCA '93, pages 289–300, San Diego, CA, USA, May 1993. ACM.

Maurice Herlihy and Nir Shavit. *The Art of Multiprocessor Programming*. Morgan Kaufmann, 2008. ISBN 0123705916.

Maurice Herlihy, Victor Luchangco, and Mark Moir. The repeat offender problem: A mechanism for supporting dynamic-sized, lock-free data structures. In *Proceedings of the 16th International Conference on Distributed Computing*, DISC '02, pages 339–353, Toulouse, France, October 2002. Springer-Verlag.

Maurice Herlihy, Victor Luchangco, and Mark Moir. Obstruction-free synchronization: Double-ended queues as an example. In *Proceedings of the 23rd International Conference on Distributed Computing Systems*, ICDCS '03, pages 522–529, Providence, RI, USA, May 2003a. IEEE Computer Society.

Maurice Herlihy, Victor Luchangco, Mark Moir, and William N. Scherer III. Software transactional memory for dynamic-sized data structures. In *Proceedings of the 22nd ACM Symposium on Principles of Distributed Computing*, PODC '03, pages 92–101, Boston, MA, USA, July 2003b. ACM.

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

Maurice Herlihy, Yossi Lev, Victor Luchangco, and Nir Shavit. A provably correct scalable concurrent skip list (brief announcement). In *Proceedings of the 10th International Conference On Principles of Distributed Systems*, OPODIS '06, Bordeaux, France, December 2006. Springer-Verlag.

Maurice P. Herlihy and Jeannette M. Wing. Linearizability: a correctness condition for concurrent objects. *ACM Transactions on Programming Languages and Systems*, 12(3):463–492, July 1990.

Shane V. Howley and Jeremy Jones. A non-blocking internal binary search tree. In *Proceedings of the 24th ACM Symposium on Parallelism in Algorithms and Architectures*, SPAA '12, pages 161–171, Pittsburgh, PA, USA, June 2012. ACM.

IBM. *IBM System/370 Extended Architecture, Principles of Operation*. IBM Publication No. SA22-7085, 1983.

Intel. Intel Architecture Instruction Set Extensions Programming Reference, Chapter 8. http://software.intel.com/sites/default/files/319433-014.pdf.

Amos Israeli and Lihu Rappoport. Disjoint-access-parallel implementations of strong shared memory primitives. In *Proceedings of the 13th ACM Symposium on Principles*

of Distributed Computing, PODC '94, pages 151–160, Los Angeles, CA, USA, August 1994. ACM.

H. T. Kung and Philip L. Lehman. Concurrent manipulation of binary search trees. *ACM Transactions on Database Systems*, 5(3):354–382, September 1980.

Edya Ladan-Mozes and Nir Shavit. An optimistic approach to lock-free FIFO queues. In *Proceedings of the 18th International Conference on Distributed Computing*, DISC '04, pages 117–131, Amsterdam, The Netherlands, October 2004. Springer-Verlag.

Doug Lea. Concurrency JSR-166 Interest Site. http://gee.cs.oswego.edu/dl/concurrency-interest/.

Paul E. McKenney and Jack D. Slingwine. Read-copy update: Using execution history to solve concurrency problems. In *Proceedings of the 10th IASTED International Conference on Parallel and Distributed Computing and Systems*, ICPDCS '98, pages 509–518, Las Vegas, NV, USA, October 1998. IAENG.

Maged M. Michael. High performance dynamic lock-free hash tables and list-based sets. In *Proceedings of the 14th ACM Symposium on Parallel Algorithms and Architectures*, SPAA '02, pages 73–82, Winnipeg, MB, Canada, August 2002a. ACM.

Maged M. Michael. Safe memory reclamation for dynamic lock-free objects using atomic reads and writes. In *Proceedings of the 21st ACM Symposium on Principles of Distributed Computing*, PODC '02, pages 21–30, Monterey, CA, USA, July 2002b. ACM.

Maged M. Michael. CAS-based lock-free algorithm for shared deques. In *Proceedings of the 9th International Conference on Parallel and Distributed Computing*, Euro-Par '03, pages 651–660, Klagenfurt, Austria, August 2003. Springer.

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

Maged M. Michael and Michael L. Scott. Correction of a memory management method for lock-free data structures. Technical Report UR CSD / TR599, Department of Computer Science, University of Rochester, NY, USA, December 1995.

Maged M. Michael and Michael L. Scott. Simple, fast, and practical non-blocking and blocking concurrent queue algorithms. In *Proceedings of the 15th ACM Symposium on Principles of Distributed Computing*, PODC '96, pages 267–275, Philadelphia, PA, USA, May 1996. ACM.

Sundeep Prakash, Yann Hang Lee, and Theodore Johnson. A nonblocking algorithm for shared queues using compare-and-swap. *IEEE Transactions on Computers*, 43 (5):548–559, May 1994.

William Pugh. Concurrent maintenance of skip lists. Technical Report CS-TR-2222, Department of Computer Science, University of Maryland, College Park, MD, USA, June 1990a.

William Pugh. Skip lists: a probabilistic alternative to balanced trees. *Communications of ACM*, 33(6):668–676, June 1990b.

Nir Shavit and Dan Touitou. Software transactional memory. In *Proceedings of the 14th ACM Symposium on Principles of Distributed Computing*, PODC '95, pages 204–213, Ottowa, ON, Canada, August 1995. ACM.

Nir Shavit and Dan Touitou. Software transactional memory. *Distributed Computing*, 10(2):99–116, February 1997.

SPIN. SPIN model checker. http://spinroot.com.

Håkan Sundell. *Efficient and Practical Non-Blocking Data Structures*. PhD thesis, Department of Computing Science, Chalmers University of Technology, Gothenburg, Sweden, November 2004.

Håkan Sundell and Philippas Tsigas. Fast and lock-free concurrent priority queues for multi-thread systems. In *Proceedings of the 17th International Parallel and Distributed Processing Symposium*, IPDPS '03, page 84, Nice, France, April 2003. IEEE Computer Society.

Håkan Sundell and Philippas Tsigas. Scalable and lock-free concurrent dictionaries. In *Proceedings of The 19th ACM Symposium on Applied Computing*, SAC '04, pages 1438–1445, Nicosia, Cyprus, March 2004a. ACM.

Håkan Sundell and Philippas Tsigas. Lock-free and practical doubly linked list-based deques using single-word compare-and-swap. In *Proceedings of the 8th International Conference on Principles of Distributed Systems*, OPODIS '04, pages 240–255, Grenoble, France, December 2004b. Springer-Verlag.

Håkan Sundell and Philippas Tsigas. Fast and lock-free concurrent priority queues for multi-thread systems. *Journal of Parallel and Distributed Computing*, 65(5):609–627, May 2005.

Håkan Sundell and Philippas Tsigas. Lock-free deques and doubly linked lists. *Journal of Parallel and Distributed Computing*, 68(7):1008–1020, July 2008.

TCMalloc. Thread-caching malloc documentation. http://goog-perftools.sourceforge.net/doc/tcmalloc.html.

R. Kent Treiber. Systems programming: Coping with parallelism. Technical Report RJ 5118, IBM Almaden Research Center, San Jose, CA, USA, April 1986.

John Turek, Dennis Shasha, and Sundeep Prakash. Locking without blocking: making lock based concurrent data structure algorithms nonblocking. In *Proceedings of the 11th ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems*, PODS '92, pages 212–222, San Diego, CA, USA, June 1992. ACM.

John D. Valois. Implementing lock-free queues. In *Proceedings of the 7th International Conference on Parallel and Distributed Computing Systems*, PDCS '94, pages 64–69, Las Vegas, NV, USA, October 1994.

John D. Valois. Lock-free linked lists using compare-and-swap. In *Proceedings of the 14th ACM symposium on Principles of Distributed Computing*, PODC '95, pages 214–222, Ottowa, ON, Canada, August 1995. ACM.