הטכניון
מכון טכנולוגי לישראל
TECHNION
Israel Institute of Technology

ספריות הטכניון
The Technion Libraries

בית הספר ללימודי מוסמכים ע"ש ארווין וג'ואן ג'ייקובס
Irwin and Joan Jacobs Graduate School

Multi-Threaded Coordination Methods for Constructing Non-blocking Data Structures

Anastasia Braginsky

Multi-Threaded Coordination Methods for Constructing Non-blocking Data Structures

Research Thesis

In Partial Fulfillment of the Requirements
for the Degree of Doctor of Philosophy

Anastasia Braginsky

Submitted to the Senate of the Technion —
Israel Institute of Technology

Iyar 5775 Haifa May 2015

The Research Thesis Was Done Under The Supervision of Prof. Erez Petrank
in the Faculty of Computer Science at the Technion

Acknowledgements

First and foremost, I wish to thank my advisor, Prof. Erez Petrank for his guidance, support, patience and fruitful conversations about my work, during all the stages of this research. Erez gave me the freedom and confidence to explore my own ideas and always was around to keep me on track with wise guidance. All these qualities made my years as a PhD student so enjoyable. An encouraging, cooperative and truly interested advisor is something that every PhD student wants, whereof I am one of the privileged.

I am grateful to my co-authors Dr. Alex Kogan and Nachshon Cohen for being a great collaborators and for their important role in the research.

Through this years, I was fortunate to be teaching assistant in charge in Operation Systems course. I have to thank the entire course staff for being such a great team to work with and specially Dr. Leonid Raskin for being a good friend and excellent lecturer to work with. I thank Mika Shapira for being a great teaching coordinator, who was always flexible enough for my needs.

This dissertation is dedicated to my parents Tatiana and Gennady Braginsky, for everything they gave me and and made possible for me. I would like to thank you for setting high goals for me. I would also like to thank my parents in law Valentina and Zalman Pevzner, for helping to take care of our beautiful daughter, Sarah, while I was finishing this dissertation.

But above all, I want to thank my dear husband Alexander Pevzner, for being such a devoted spouse, for bringing light and laugh into my life. Especially, I am grateful to Alex for reminding me to enjoy the journey thereof. Dear Alex, I could not have done this without you.

The generous financial help of the Technion and the Ministry of Science is gratefully acknowledged.

Publication List

1. A. Braginsky and E. Petrank. Locality-Conscious Lock-Free Linked Lists. *Proc. ICDCN 2011* [5]
2. A. Braginsky and E. Petrank. Lock-Free B+tree. *Proc. SPAA 2012* [6]
3. A. Braginsky, A. Kogan and E. Petrank. Drop the Anchor: Lightweight Memory Management for Non-Blocking Data Structures *Proc. SPAA 2013* [4]
4. A. Braginsky, N. Cohen and E. Petrank. CBPQ: High Performance Lock-Free Priority Queue. *Submitted. Not-yet published.*

Dedicated with love to my dear parents
Tatiana and Gennady

Contents

Abstract **1**

1 Introduction **3**
1.1 Overview . . . 3
1.2 Background . . . 4
1.2.1 Non-blocking synchronization . . . 4
1.2.2 Linearizability . . . 5
1.2.3 Non-blocking primitives . . . 6
1.2.4 Universal constructions . . . 7
1.2.5 Ad hoc data structures . . . 7
1.2.6 Memory management . . . 9

2 Locality-Conscious Lock-Free Linked Lists **11**
2.1 Introduction . . . 11
2.2 Preliminaries and Data Structure . . . 12
2.3 Using a Freeze to Retire a Chunk . . . 14
2.4 The List Operations: Search, Insert and Delete . . . 15
2.4.1 The insert operation . . . 15
2.5 The Freeze Procedure . . . 18
2.5.1 The initiation of a freeze . . . 20
2.5.2 The stabilization phase . . . 20
2.5.3 The decision and the recovery . . . 21
2.5.4 Managing the external freeze activities . . . 23
2.6 The Details of the Additional Chunk-level Methods . . . 25
2.6.1 The search operation . . . 25
2.6.2 The delete operation . . . 27
2.6.3 Counter Functionalities . . . 30
2.7 The Upper-Level List Operations . . . 31
2.8 Supporting functionalities . . . 36
2.9 Linearization Points . . . 36
2.10 The intuition behind the design considerations . . . 38
2.11 Lock-Freedom . . . 39

3 A Lock-Free B$^+$tree **46**
3.1 Introduction . . . 46
3.2 Preliminaries and Data Structure . . . 48
3.2.1 The B$^+$tree . . . 49
3.2.2 The structure of the proposed B$^+$tree . . . 49
3.2.3 Memory Management . . . 50
3.2.4 The Basic B$^+$tree Operations . . . 51
3.3 Splits and Joins with Freezing . . . 51
3.4 Balancing the B$^+$tree . . . 53
3.4.1 Node Split . . . 53
3.4.2 Nodes Join . . . 54
3.4.3 Two Invariants . . . 56
3.4.4 Extensions to the Chunk Mechanism . . . 57
3.5 Implementation and Results . . . 57
3.6 Linearization Points . . . 59
3.7 B$^+$tree supporting methods . . . 60
3.8 Code and Detailed Explanations for Split, Merge and Borrow . . . 62
3.8.1 Node Splits . . . 62
3.8.2 Merges . . . 64
3.8.3 Borrow . . . 69
3.9 Redirection of the call for an update . . . 69
3.10 Root boundary conditions . . . 71
3.10.1 Splitting the root . . . 71
3.10.2 Root Merge . . . 72
3.11 Minor Modifications to the Chunk Interfaces . . . 74
3.11.1 The addition of replace interface to the list . . . 74
3.11.2 The insert and delete operations . . . 75
3.11.3 Freeze Functionality Code . . . 76

4 *Drop the Anchor*: Lightweight Memory Management for Non-Blocking Data Structures **82**
4.1 Introduction . . . 82
4.2 Related Work . . . 84
4.3 An Overview of Drop the Anchor . . . 85
4.4 Detailed Description . . . 87
4.4.1 Auxiliary fields and records . . . 87
4.4.2 Anchor maintenance . . . 88
4.4.3 Node reclamation . . . 89
4.4.4 Recovery procedure . . . 89
4.4.5 The refined reclamation procedure . . . 92
4.5 Performance Evaluation . . . 93
4.6 Pseudo-code . . . 96
4.6.1 Correctness argument . . . 96

5 CBPQ: High Performance Lock-Free Priority Queue 107
5.1 Introduction . . . 107
5.2 A bird-Eye Overview . . . 109
5.3 The Full CBPQ Design . . . 110
5.3.1 Data Structures . . . 110
5.3.2 Operations Implementation . . . 111
5.3.3 Split and Merge Algorithms . . . 112
5.4 Optimizations . . . 118
5.5 Performance Evaluation . . . 120
5.6 Correctness . . . 122
5.7 Pseudo-code . . . 124

6 Discussion and Conclusions 128

References 130

List of Figures

2.1 The entry structure. . . . 13
2.2 The chunk structure. . . . 13
2.3 The list structure. . . . 14
2.4 The COPY recovery in list of chunks. . . . 33
2.5 The MERGE recovery in list of chunks. . . . 34
2.6 The specification of (simple) supporting functions. . . . 45

3.1 The structure of a chunk. The allocated grey entries present the ordered linked list. 50
3.2 The state transitions of the freeze state of a node. The initial states are presented in the boxes with the double border. . . . 52
3.3 The empirical results. . . . 58
3.4 The diagram presenting the merging of the root. The initial B^+tree is marked in grey. Node R_o is an old root that initially had two children C_1 and C_2. Node C_2 is frozen as a master. Node C_1 is frozen as a slave for C_2. . . . 73

4.1 Transition diagram for possible states of the thread t . . . 85
4.2 Recovery phases. Nodes marked with 'x' are deleted, i.e., the delete-bit of their `next` pointer is turned on [22]. Shaded nodes are frozen, i.e., the freeze-bit of their `next` pointer is turned on. . . . 90
4.3 Drop the Anchor vs. Hazard Pointers for lists with the initial size of 100k keys, the mixed workload results. . . . 94
4.4 Drop the Anchor vs. Hazard Pointers for lists with the different initial sizes and the recovery performance impact. . . . 95
4.5 Drop the Anchor vs. Hazard Pointers for lists with the initial size of 100k keys, the read-only workload results. . . . 101

5.1 Overview of the CBPQ data structure for $N = 5$ . . . 111
5.2 Throughput in delete and mixed workloads. . . . 120
5.3 CBPQ vs. Lock-Free and Lock-Based Mound in insert workload. . . . 120

Abstract

Shared-memory multiprocessors concurrently execute multiple threads of computation that communicate and synchronize through shared memory. Typically, this communication and synchronization is done via concurrent data structures, whose efficiency is crucial to performance. Furthermore, new challenges arise in designing scalable concurrent data structures that can perform well with an increasing number of concurrent threads. Non-blocking data structures are scalable and provide a progress guarantee. If several threads attempt to concurrently apply an operation on the structure, it is guaranteed that one of the threads will eventually make progress. Consequently, the popular mutual exclusion synchronization protocol cannot be used in non-blocking data structures.

The first goal of this dissertation is to address the design of high-performance, concurrent, non-blocking data structures for shared memory multi-processor platforms. Our intention is that the non-blocking data structure will become the primary choice for a concurrent data structure. To this end, we first present a high performance linked list. We extend state-of-the-art lock-free linked lists by building linked lists with special attention to locality of traversals. These linked lists are built of sequences of entries that reside on consecutive chunks of memory. When traversing such lists, subsequent entries typically reside on the same chunk and are thus close to each other, e.g., in same cache line or on the same virtual memory page. Such cache-conscious implementations of linked lists are frequently used in practice, but making them lock-free requires care. The basic component of this construction is a chunk of entries in the list that maintains a minimum and a maximum number of entries. This basic chunk component is an interesting tool on its own and is used to build the other lock-free data structures that we present (Chapter 2).

Another high-performance, non-blocking data structure presented in this dissertation is a priority queue. Priority queues are an important algorithmic component and are ubiquitous in systems and software. With the rapid deployment of parallel platforms, concurrent versions of priority queues are becoming increasingly important. In this dissertation, we present a novel, concurrent, lock-free linearizable algorithm for priority queues that significantly outperforms all known (lock-based or lock-free) priority queues. Our algorithm employs recent advances, including lock-free chunks and the use of the efficient fetch-and-increment atomic instruction. Measurements demonstrate a performance improvement by a factor

of up to 2 over existing approaches to concurrent priority queues (Chapter 5).

The second goal of this dissertation is to expand the coverage of existing lock-free variants for the data structures whose lock-free implementations have not yet been discovered. This is because the lock-free data structures provide a progress guarantee and are known for facilitating scalability, avoiding deadlocks and live-locks, and providing guaranteed system responsiveness. In this dissertation we present a design for a lock-free balanced tree, specifically, a B$^+$tree. The B$^+$tree data structure has an important practical applications, and is used in various storage-system products. To the best of our knowledge, this is the first design of a lock-free, dynamic, and balanced tree that employs standard compare-and-swap operations (Chapter 3).

The third and final dissertation goal is to support the efficient memory management for non-blocking data structures. Efficient memory management of dynamic non-blocking data structures remains an important open question. Existing methods either sacrifice the ability to deallocate objects or reduce performance notably. In this dissertation, we present a novel technique, called *Drop the Anchor*, which significantly reduces the overhead associated with the memory management while reclaiming objects even in the presence of thread failures. We demonstrate this memory management scheme on the common linked list data structure. Using extensive evaluation, we show that Drop the Anchor significantly outperforms Hazard Pointers, the widely used technique for non-blocking memory management (Chapter 4).

All the above-mentioned algorithms were evaluated empirically and shown to provide high throughput and performance. They all utilize our new method of thread coordination for the case when threads need to be redirected from an obsolete part of the data structure to a new one. We denote this technique *freezing*. The freezing technique supports the restructuring of the lock-free data structures, in order to notify threads to move to another part of the data structure, usually because the part they are currently using is obsolete. This is done by setting a special *freeze-bit* on data or pointer words in the obsolete part, making the data unsuitable for updating. A thread that fails in its attempt to use a frozen pointer or data realizes that this part of the data structure is obsolete and restarts its operation. For performance optimization we later batch the freeze-bits into separate *freeze-words* (Section 2.3).

Chapter 1

Introduction

1.1 Overview

Processor technology has advanced to a point where the processor clock speed, or the running frequency, can no longer be improved. This once driving factor of the computational throughput of a processor is now kept at a steady rate. Instead of increasing the processor clock speed, the processor vendors have shifted their focus towards providing multiple computational units as part of the same processor, and named the new family of central processing units *multicore processors*. These computer systems rely on the concept of *shared-memory*, in which every processor has the same read and write access to the part of the computer called the main memory.

This shift in processor technology has resulted in a plethora of novel and original research works in recent decades. The data structures can now be seen from a different angle, as a space for different processors to communicate, in order to read and update the data simultaneously. The driving force behind parallelism is the quest for performance. Each separate computation unit is called a *thread*. The initial idea behind the parallelism is: the more threads that can access the data in parallel and perform their operations simultaneously, the better the performance will be. However, the idea doesn't always make it into practice, because those accesses now need to be synchronized in order to keep the structure correct, where the synchronization may decrease the performance.

A popular way to synchronize thread accesses is to use *mutual exclusion*, where only one thread can access the entire (or part of) data structure. Other threads, needing the same (or nearby) access, are delayed until the first thread finishes. This simplifies programming, but has many disadvantages, mostly because the thread that has the access delays all the others that need the same data. This dissertation focuses on the *non-blocking* data structures (NBDS), in which no thread can delay the progress of others.

The dissertation addresses three aspects of non-blocking programming: (1) **Performance** – we believe that non-blocking algorithms are not yet performing at full capacity. We suggest more efficient NBDSs for the existing lock-free algorithms

for the linked list and the priority queue. Our linked list and priority queue outperform the existing equivalents by a factor of up to 2. (2) **Coverage** – we would like to extend the coverage of the existing non-blocking variants for the lock-based concurrent data structure. To this end, we suggest a non-blocking variant for a balanced search tree, for which a non-blocking algorithm was unknown until we presented our work (to the best of our knowledge). The lock-free B^+tree shows scalability and performance improvement over the lock-based invariant. (3) **Memory Management** – if they are to be adopted for widespread use, the non-blocking data structures need an efficient and dynamic memory management. We suggest a novel technique that manages the memory much faster than the previous techniques, without loss of any of their capabilities. Finally, all of our works employ a *frezing* technique for thread coordination, to be explained in Section 2.3. This technique allows many threads to be notified that part of the data structure they are working on is obsolete.

The rest of the introduction is devoted to describing the background and our contributions.

1.2 Background

1.2.1 Non-blocking synchronization

Concurrent data structures are more difficult to design than sequential ones, because threads executing concurrently may interleave their steps in many ways, each with a different and possibly unexpected outcome. A straightforward way to limit such interleavings is to use a mutual exclusion lock. However, locking may introduce a host of problems related both to performance and to software engineering. A single lock or just a few of them might cause a major bottleneck to the access of a data structure. Thus it is better to reduce the lock granularity, i.e., reduce the number of instructions executing while holding a lock – *fine-grained* locking. Concurrent data structures based on fine-grained locking are commonly used, but suffer from deadlock potential, priority inversion, the convoying effect, and other problems. The problems in the lock-based implementations arise because, if the thread that currently holds the lock is delayed, then all other threads attempting to take the lock are also delayed. This is called blocking. To solve this problem, several variants of progress guarantees have been proposed. They typically provide progress guarantees by precluding mutual exclusion. The non-blocking properties guarantee that a stalled process cannot cause all the other processes to stall indefinitely. The tradeoff they explore is the range of assurances which may be provided to groups of conflicting non-stalled processes.

Lock-freedom: An algorithm is lock-free if and only if some operation completes after a finite number of steps have been executed system-wide on the structure. This guarantee of system-wide progress is usually satisfied by making sure that if one process fails to make progress, then another process is guaranteed to make progress. This is a very different approach to that taken by lock-based algo-

rithms, in which a process will either spin or block until the contending operation is completed [26, 30].

Wait-freedom: Although lock-freedom guarantees system-wide progress it does not ensure that individual operations eventually complete since, in theory, an operation may continually be deferred while its process yields to a never-ending sequence of contending operations. In some applications a fairer condition such as wait-freedom may be desirable. An algorithm is wait-free if every operation on the structure completes after it has executed a finite number of steps. This condition ensures that no operation can experience permanent live-lock and, in principle, a worst-case execution time can be calculated for any operation [26].

In the past, it was decided that it is difficult to implement efficient wait-free algorithms on commodity hardware since fair access to memory is usually not guaranteed. Extensive, sophisticated, algorithmic-based synchronization was usually required to ensure that no process is starved. This was typically achieved by requiring each process to announce its current operation in a single-writer memory location. Processes which successfully made forward progress were required to periodically scan the announcements of other processes and help their operations to complete. Over time, the scanning algorithm checks every process in the system. However, Kogan and Petrank recently showed a methodology for creating fast wait-free data structures from lock-free data structures [34]. The methodology employs the lock-free algorithm, unless a rare case of starvation is encountered. If that happens, the algorithm switches to run in wait-free mode. Shortly thereafter, Timnat and Petrank presented a practical wait-free simulation for lock-free data structures [51]. Nowadays, the greatest interest is therefore in the creation of lock-free data structures, as efficient wait-free data structures can be created from them almost automatically.

Obstruction-freedom: Herlihy et al. have suggested a weak non-blocking property called obstruction-freedom, which they believe can provide many of the practical benefits of lock-freedom, but with reduced programming complexity and the potential for more efficient data-structure designs [29]. Since efficiently allowing operations to help each other to complete is a major source of complexity in many lock-free algorithms, and excessive helping can generate harmful memory contention, obstruction-freedom can reduce overheads by allowing a conflicting operation to instead be aborted and retried later. More formally, an algorithm is obstruction-free if and only if every operation on the structure completes after executing a finite number of steps that do not contend with any concurrent operation for access to any memory location.

1.2.2 Linearizability

Besides performance, algorithms can also be evaluated by a behavior metric: does the algorithm behave as expected when it is deployed in an application? One property which is commonly considered desirable in concurrency-safe algorithms is linearizability [31] (a variation on serializability). This property is defined in

terms of requests to and responses from a compliant operation: if the operation is implemented as a synchronous procedure, then a call to that procedure is a request and the eventual return from that procedure is a response. An operation is *linearizable* if and only if it appears to execute instantaneously at some point between its request and response. Linearizability ensures that operations have intuitively "correct" behavior. Concurrent invocations of a set of linearizable operations will have a corresponding sequence which could be executed by just one processor with exactly the same outcome. Another way of thinking of this condition is that it requires us to be able to identify a distinct point within each operation's execution interval, called its *linearization point*, such that if we order the operations according to the order of their linearization points, the resulting order obeys the desired sequential semantics.

1.2.3 Non-blocking primitives

An early paper by Herlihy demonstrates that various classic atomic primitives, e.g. fetch-&-add ($(F\&I)$) and test-&-set, have differing levels of expressiveness [25]. Specifically, a hierarchy is constructed in which primitives at a given level cannot be used to implement a wait-free version of any primitives at a higher level. Only a few of the well-known primitives discussed in the paper are universal in the sense that they can be used to solve the n-process consensus problem in its general form. One such universal primitive is compare-&-swap (CAS), which is usually used to build the non-blocking algorithms, together with atomic reads and writes. Originally implemented in the IBM System/370, many modern multiprocessors support this operation in hardware.

Rather than implementing the read-modify-write instructions directly, some processors provide separate load-linked and store-conditional (LL/SC) operations. Unlike the strong LL/SC operations sometimes used when describing algorithms, the implemented instructions must form non-nesting pairs and SC can fail "spuriously" [27]. Methods for building read-modify-write primitives from LL/SC are known: for example, how to use them to construct atomic single-word sequences such as CAS. Such constructions, based on a simple loop that retries a LL/SC pair, are non-blocking under a guarantee that there are not infinitely many spurious failures during a single execution of the sequence. In [42] Michael presents lock-free and wait-free implementations of LL/SC that require 64-bit CAS in 64-bit programs.

The design of efficient non-blocking algorithms is much easier if more expressive operations such as DCAS (double CAS) [11], MCAS (multiple-word CAS) [23] are supported. DCAS (MCAS) takes two (multiple) not necessarily contiguous memory locations and writes new values into them only if they match pre-supplied "expected" values. Unfortunately, only the obsolete Motorola 680x0 family of processors supports DCAS directly in hardware. As far as we know, MCAS is not implemented in hardware. Harris and Fraser describe how to implement MCAS using a single-word CAS command [23]. In addition, there is a so called WCAS (wide

CAS), a double-width CAS operation which acts on an adjacent pair of memory locations. This instruction is supported by the majority of modern multiprocessor architectures, while it requires more extensive synchronization by hardware than that required for the single-word CAS instruction. Of course, atomic reads and writes are also used.

It should be noted that non-universal ($F\&I$) instructions can be a very effective means of attaining better performance, can be applied to the operation with a single point of memory contention. This in comparison with stronger, but less performant compare-and-swap (CAS) atomic primitive instructions, as also noted in [17, 43]. A use of the ($F\&I$) instruction has provided nice performance improvements over the CAS instruction for incrementing of a contended counter on a modern x86 system [43]. There, the elimination of the retries incurred by the CAS implementation led to a 4-to-6 times performance improvement.

1.2.4 Universal constructions

Universal constructions are a class of lock-free and wait-free techniques that can be straightforwardly applied to a wide range of sequential programs to make them safe in parallel-execution environments. Indeed, most of these constructions are intended to be applied automatically by a compiler or run-time system. For example, Herlihy describes a universal construction for automatically creating a non-blocking algorithm from a sequential specification [27]. This requires a snapshot of the entire data object to be copied to a private location where shadow updates can safely be applied: these updates become visible when the single "root" pointer of the structure is atomically checked and modified to point at the shadow location. Although Herlihy describes how copying costs can be greatly reduced by replacing only those parts of the object that are modified, the construction still requires atomic update of a single root pointer. This means that concurrent updates will always conflict, even when they modify disjoint sections of the data structure. Many follow-up studies further extended the idea of the universal construction. They made it more efficient, but all these constructions are typically not efficient and scalable enough to be used in practice.

1.2.5 Ad hoc data structures

Although there are many universal constructions and programming abstractions that seek to facilitate the implementation of complex data structures, practical concerns have caused most designers to resort to building non-blocking algorithms directly from machine primitives, e.g., CAS, $F\&I$, and LL/SC. Consequently, a large body of work describes ad hoc designs for data structures such as stacks, queues, lists, skip-lists, hash-tables, and unbalanced binary search trees. In this dissertation we work mainly with ad hoc data structures. Here we present related work for the lock-free singly-linked list (which has received much attention and has many implementations), for the search tree, and for the priority queue. Additional

data structures will be presented when required, close to the relevant section.

Linked Lists: The first design of lock-free linked lists was presented by Valois [52]. He maintained auxiliary nodes in between the list's normal nodes, in order to resolve intererence among concurrent operations. A different lock-free implementation of linked lists was given by Harris [22]. His main idea was to mark a node before deleting it in order to prevent concurrent operations from changing its next-entry pointer. Harris's algorithm is simpler than Valois's, and his experimental results are generally better. Michael [41] proposed an extension to Harris's algorithm that did not assume garbage collection but reclaimed entries of the list explicitly. To this end, he developed an underlying mechanism of hazard pointers that was later used for explicit reclamation in other data structures as well. An improvement in complexity was achieved by Fomitchev and Rupert [18]. They use a smart retreat upon CAS failure, rather than the standard restart from scratch.

The high performance, lock-free, linked list that we propose also does not traverse the list from the beginning upon the CAS failure. In addition, our list attains high performance by employing locality and by using skips (over chunks of the list) during traversal. More details about this list can be found in Chapter 2. The first implementation of a wait-free linked list was independently proposed by Timnat et al. [50].

Trees: Previous works on lock-free trees include Fraser's construction [19] of a lock-free balanced tree. Their construction is simplified significantly by building on an underlying transactional memory system. Fraser also presents a construction of a lock-free tree based on multiple-word CAS [19], but this construction offers no balancing and in the worst case may require a linear complexity for the tree operations. Ellen et al. [16] presented the first lock-free tree using single-word CAS, but their tree offers no balancing. Bender et al. [3] described a lock-free implementation of a cache-oblivious B-tree from LL/SC operations. For comparison, our B+tree construction uses single-word CAS operations. Moreover, a packed-memory cache-oblivious B-tree is not equivalent to the traditional B+tree data structure. First, it only guarantees amortized time complexity (even with no contention), as the data is kept in an array that needs to be extended occasionally by copying the entire data structure. Second, it does not keep the shallow structure and it is less suitable for file systems. Finally, a full version of this paper has not yet appeared and details of the lock-free implementation are not specified. As part of this thesis we present the first lock-free, linearizable, dynamic B^+tree implementation supporting searches, insertions, and deletions. It is dynamic in the sense that there is no (static) limit to the number of nodes that can be allocated and put in the tree. The construction employs only reads, writes, and (single-word) CAS instructions. Searches are not delayed by rebalancing operations. The construction employs the lock-free chunk mechanism that fits naturally with a node of the B+tree that is split and joined, keeping the number of elements within given bounds, and thus maintaining the balance of the tree. More details

about the B$^+$tree can be found in Chapter 3. After our work was published, the research around non-blocking unbalanced binary search has continued [7, 44, 45]. An AVL tree with lock-free look up was presented in [13].

Priority Queues: Various constructions for the concurrent PQ exist in the literature. Hunt et al. [32] used a fine-grained lock-based implementation of a concurrent heap. Dragicevic and Bauer presented a linearizable heap-based priority queue that used lock-free software transactional memory (STM) [14]. Their algorithm was intended to improve performance by splitting critical sections into small atomic regions, but the overhead of the STM resulted in low performance. A quiescently consistent skip-list based priority queue was first proposed by Lotan and Shavit [37] using fine-grained locking, and was later made lock-free [19]. Another skip-list based priority queue was proposed by Sundell and Tsigas [49]. While this implementation is lock-free and linearizable, it required reference counting, which compromises disjoint-access parallelism and degrades performance.

Liu and Spear [36] introduced two concurrent versions of a data structure called *mounds* (one is lock-based and the other is lock-free). The mounds data structure is a rooted tree of sorted lists that relies on randomization for balance. It supports $O(\log(\log(N)))$ `insert` operations and $O(\log(N))$ `deleteMin` operations. Mounds perform well in practice (with high probability) and their `insert` operation is currently the most performant among concurrent implementations of the PQ. Linden and Jonsson [35] presented a skip-list based PQ. Deleted elements are first marked as deleted in the `deleteMin` operation. Later, they are actually disconnected from the PQ in batches when the number of nodes marked as deleted exceed a given threshold. Their construction outperforms previous algorithms by $30 - 80\%$. Recently, Calciu et al. [8] introduced a new lock-based, skip-list-based PQ that uses elimination and flat combining techniques to achieve high scalability at high thread counts. Their elimination mechanism is of independent interest and can be added to our mechanism to achieve even better performance. We have proposed a lock-free data structure for a high performance priority queue. It is called *CBPQ* for Chunked Based Priority Queue and it outperforms the presented above implementations; details are provided in Chapter 5.

1.2.6 Memory management

Many non-blocking algorithms in the literature are presented in pseudo-code which assumes that automatic garbage collection is provided as a run-time service. This ignores the problem that many languages do not provide this support and, furthermore, that none of the general-purpose garbage collectors are non-blocking. To deal with this, a range of non-blocking memory-management techniques have been suggested. Some works attempt to describe a lock-free garbage collector; however, a truly full lock-free garbage collector has not yet been achieved (to the best of our knowledge). Apart from garbage collection, another problem related to memory reclamation is the ABA problem. Affecting almost all lock-free algorithms, it was first reported in the documentation of CAS on the IBM System

370 [33]. It occurs when a thread reads a value A from a shared location, and then other threads change the location to a different value, say B, and then back to A again. Later, when the original thread checks the location, e.g., using read or CAS, it erroneously proceeds under the assumption that the location has not changed since the last reading because the compared values are the same. As a result, the thread may corrupt the object or return a wrong result. So in addition to the lock-free GC, many safe memory reclamation methodologies for specific data structures have been presented. Valois [52], followed by Michael and Scott [40], use reference counts to ensure that an object (fixed-size memory segment) is not reused while any thread still holds a pointer to it [52]. As there may be an arbitrary delay between obtaining a reference to an object and incrementing the reference count, objects reclaimed via reference counts must retain their type forever. Detlefs et al. solve this by using DCAS to increment the counter while simultaneously checking that the object remains globally accessible [12]. However, maintaining the reference counts can be quite costly in all reference counting schemes. This is particularly true for operations which read many objects: updates to reference counts may cause read-only objects to become a contention bottleneck. Michael [41] proposed the hazard pointer (HP) method, which focuses on local references: each thread maintains a list of pointers (HPs) of the nodes the thread may later access; when a node is removed from the data structure, the HP lists of all threads must be checked before the node is reclaimed. However, the cost of updating a hazard pointer when traversing objects can be high. On modern processors, a memory barrier must be executed after updating a hazard pointer: implementing this barrier may increase execution time by around 25% (according to Fraser [19]). A similar scheme, called Pass the Buck, was independently proposed by Herlihy et al. [28]. In its original form, it uses unbounded tags and is based on the double-width CAS atomic primitive, a CAS operation that can atomically update two adjacent memory words. This operation is available in some 32-bit architectures but only in very few of the current 64-bit architectures. More recently, Herlihy et al. [28] showed how to remove the tags from their method, to allow the later scheme to be implemented using single-width CAS.

We present a novel technique, called *Drop the Anchor*, which significantly reduces the overhead associated with memory management while reclaiming objects even in the presence of thread failures. We demonstrate this memory management scheme on the common linked-list data structure. Using extensive evaluation, we show that Drop the Anchor significantly outperforms Hazard Pointers, the widely used technique for non-blocking memory management. Our memory management technique is discussed in Chapter 4.

Chapter 2

Locality-Conscious Lock-Free Linked Lists

2.1 Introduction

In this chapter we further extend the linked list's design to allow cache-conscious linked lists. Our implementation partitions the linked list into sub-lists that reside on consecutive areas in the memory, denoted *chunks*. Each chunk contains several consecutive list entries. For example, setting each chunk to be one virtual page, causes list traversals to form a page-oriented memory access pattern. This partition of the list into sub-lists, each residing on a small chunk of memory is often used in practice (e.g., [20]), but there is no lock-free implementation for such a list. Breaking the list into chunks can be trivial if there is no restriction on the chunk size. In particular, if the size of each chunk can decrease to a single element, then clearly, each chunk can trivially reside in a single memory block, Michael's implementation will do, but no locality improvement will be obtained for list traversals. The sub-list's chunk that our design provides maintains upper and lower bounds on the number of elements it has. The upper bound simply follows from the size of the memory block on which the chunk is located, and a lower bound is provided by the user. If a chunk grows too much and cannot be held in a memory block, then it is *split* (in a lock-free manner) creating two chunks, each residing at a separate location. Conversely, if a chunk shrinks below the lower bound, then it is *merged* (in a lock-free manner) with the previous chunk in the list. In order for the split to create acceptable chunks, it is required that the lower bound (on the number of objects in a chunk) does not exceed half of the maximum number of entries in the chunk. Otherwise, a split would create two chunks that violate the lower bound.

A natural optimization of search for such a list is to quickly jump to the next chunk (without traversing all its entries), if the desired key is not within the key-range of this chunk. This gives us additional performance improvement since the search progress is done in skips, where the size of each skip is at least the chunk's minimal boundary. Furthermore the retreat upon CAS failure, in the majority of

the cases is done by returning to beginning of the chunk, rather than the standard restart from the beginning of the list.

To summarize, the contribution of this chapter is the presentation of a lock-free linked list, based on single word CAS commands, were the keys are unique and ordered. The algorithm does not assume a assume no lock-free garbage collector. The list design is locality conscious. The design poses a restriction on the keys and data length. For 64bit architecture the key is limited to 31 bit, and the data is limited to 32 bit.

2.2 Preliminaries and Data Structure

A linked list is a data structure that consists of a sequence of data records. Each data record contains a key by which the linked list is ordered. We denote each data record *an entry*. We think of the linked list as representing a set of keys, each associated with a data part. Following previous work [19, 22], a key cannot appear twice in the list. Thus, an attempt to insert a key that exists in the list fails. Each entry holds the key and data associated with it. Generally, this data is a pointer, or a mapping from the key to a larger piece of data associated with it. Next, we present the underlying data structure employed in the construction. We assume a 64-bit platform in this description. A 32-bit implementation can easily be derived, by either cutting each field in half, or by keeping the same structure, but using a wide compare-and-swap, which writes atomically to two consecutive words.

The structure of an entry: A list entry consists of a *key* and a *data* fields, and the *next* pointer (pointing to next entry). These fields are arranged in two words, where the *key* and *data* reside in the first word and the *next* pointer in the second. Three more bits are embedded in these two words. First, we embed the *delete* bit in the least bit of the *next* pointer, following Harris [22]. The *delete* bit is set to mark the logical deletion of the entry. The *freeze* bits are new in this design. They take a bit from each of the entry's words and their purpose is to indicate that the entire chunk holding the entry is about to be retired. These three flags consume one bit of the key and two bits from the *next* pointer. Notice that the three LSBs of a pointer do not really hold information on a 64-bit architecture. The entry structure is depicted in Figure 2.1. In what follows, we refer to the first word as the *keyData* word, and the second word as the *nextEntry* word.

We further reserve one key value, denoted by $\perp$ to signify that the entry is currently not allocated. This value is not allowed as a key in the data structure. As will be discussed in Section 2.4, an entry is available for allocation if its key is $\perp$ and its other fields are zeroed.

The structure of a chunk: The main support for locality stems from the fact that consecutive entries are kept on a *chunk*, so that traversals of the list demonstrate better locality. In order to keep a substantial number of entries on each chunk, the linked list makes sure that the number of entries in a chunk is

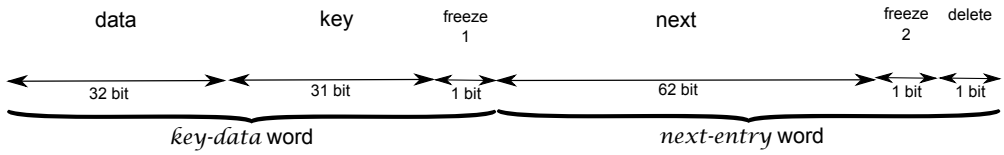


Figure 2.1: The entry structure.

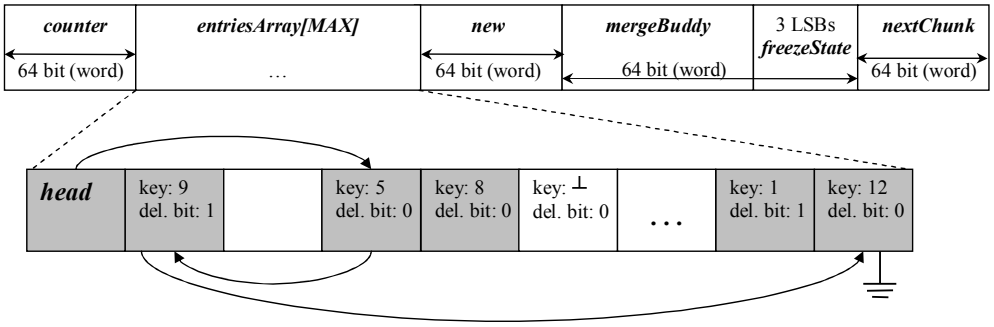


Figure 2.2: The chunk structure.

always between the parameters MIN and MAX. The main part of a chunk is an array that holds the entries in a chunk and may hold up to MAX entries of the linked list. In addition, the chunk holds some fields that help manage the chunk. First, we keep one special entry that serves as a dummy header entry, whose *next* pointer points to the first entry in this chunk. The dummy header is not a must, but it simplifies the algorithm's code. To identify chunks that are too sparse, each chunk has a counter of the number of entries currently allocated in it. In the presence of concurrent mutations, this counter will not always be accurate, but it will always hold a lower bound on the number of allocated entries in the chunk. When an attempt is made to insert too many entries into a chunk, the chunk is split. When it becomes too small due to deletions, it is merged with a neighboring chunk. We require MAX $> 2\cdot$MIN$+1$, since splitting a large chunk must create two well-formed new chunks. In practice MAX will be substantially larger than $2\cdot$MIN to avoid frequent splits and merges. Additional fields (*new*, *mergeBuddy* and *freezeState*) are needed for running the splits and the merges and are discussed in Section 2.5. The chunk structure is depicted in Figure 2.2.

The structure of entire list: The entire list consists of a list of chunks. Initially we have a HEAD pointer pointing to an empty first chunk. We let the first chunk's MIN boundary be set to 0, to allow small lists. The list grows and shrinks due to the splitting and merging of the chunks. Every chunk has a pointer *nextChunk* to the next chunk, or to NULL if it is the last chunk of the list. The keys of the entries in the chunks never overlap, i.e., each chunk contains a consecutive subset of keys in the set, and a pointer to the next chunk, containing the next subset (with strictly higher keys) in the set. The entire list structure is depicted in Figure 2.3. We set the first key in a chunk as its lowest possible key. Any smaller key is inserted in the previous chunk (except for the first chunk that can also get keys smaller than its first one.)

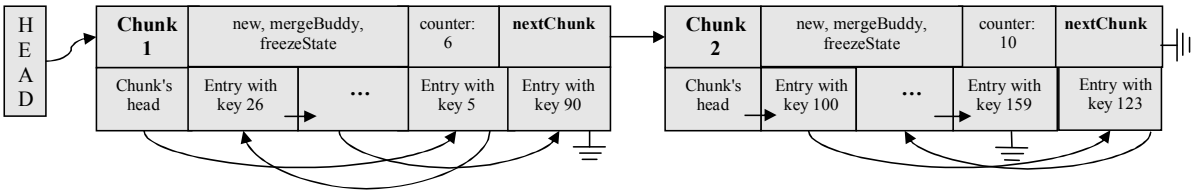


Figure 2.3: The list structure.

Hazard pointers: Whole chunks and entries inside a chunk are reclaimed manually. Note that garbage collectors do not typically reclaim entries inside an array. To allow safe (and lock-free) reclamation of entries manually, we employ Michael's hazard pointers methodology [39, 41]. While a thread is processing an entry - and a concurrent reclamation of this entry can foil its actions - the thread registers the location of this entry in a special pointer called a *hazard pointer*. Reclamation of entries that have hazard pointers referencing them is avoided. Following Michael's list implementation [41], each thread has two hazard pointers, denoted *hp0* and *hp1* that aid the processing of entries in a chunk. We further add four more hazard pointers *hp2*, *hp3*, *hp4*, and *hp5*, to handle the operations of the chunk list. Each thread only updates its own hazard pointers, though it can read the other threads' hazard pointers.

2.3 Using a Freeze to Retire a Chunk

In order to maintain the minimum and maximum number of entries in a chunk, we devised a mechanism for *splitting* dense chunks, and for *merging* a sparse chunk with its predecessor. The main idea in the design of the *split* and *merge* lock-free mechanisms is the *freezing* of chunks. When a chunk needs to be split or merged, it is first frozen. No insertions or deletions can be executed on a frozen chunk. To split a frozen chunk, two new chunks are created and the entries of the frozen chunk are copied into them. To merge a frozen chunk with a neighbor, the neighbor is first frozen, and then one or two new chunks are allocated and the relevant entries from the two merging chunks are copied into them. Details of the freezing mechanism appear in Section 2.5. We now review this mechanism in order to allow the presentation of the list operations.

The freezing of a chunk comprises three phases:

Initiate Freeze: When a thread decides a chunk should be frozen, it starts setting the freeze bits in all its entries one by one. During the time it takes to set all these bits, other threads may still modify the entries not yet marked as frozen. During this phase, only part of the chunk is marked as frozen, but this freezing procedure cannot be reversed, and frozen entries cannot be reused.

Stabilizing: Once all entries in a chunk are frozen, allocations and deletions can no longer be executed. At this point, we link the non-deleted entries into a list. This includes entries that were allocated, but not yet connected to the list. All

entries that are marked as deleted are disconnected from the list.

Recovery: The number of entries in the stabilized list is counted and a decision is made whether to split this chunk or merge it with a neighbor. Sometimes, due to changes that happen during the first phase, the frozen chunk becomes a good one that does not require a split or a join. Nevertheless, the retired chunk is never resurrected. We always allocate a new chunk to replace it and copy the appropriate values to the new chunk. Whatever action is decided upon (*split*, *join*, or *copy chunk*) must be carried through.

Any thread that fails to insert or delete a key due to the progress of a freeze, joins in helping the freezing of the chunk. However, threads that perform a search, continue to search in frozen chunks with no interference.

2.4 The List Operations: Search, Insert and Delete

We now turn to describe the basic linked list operations. The high-level code for an insertion, deletion, or search of a key is very simple. Each of this operations starts by invoking *FindChunk* method to find the relevant chunk. Then they call *SearchInChunk*, or *InsertToChunk*, or *DeleteInChunk* according to the desired operation, and finally, the hazard pointers *hp2*, *hp3*, *hp4*, and *hp5* are nullified, to release the hazard pointers set by the *FindChunk* method and allow future reclamation. The main challenge is in the work inside the chunk and the handling of the freeze process, on which we elaborate below.

Turning to the operations inside the chunks, the delete and search methods are close to the previous design [41], except for the special treatment of the chunk bounds and the freeze status. However, the insert method is quite different, because it must allocate an entry in a shared memory (on the chunk), whereas previously, it was assumed that the insert allocates a local space for a new entry and privately prepares it for insertion in the list.

For the purpose of handling the entries list in the chunk, we maintain five variables that are global and appear in all the code below. These variables are global for each thread's code, but are not shared between threads, and all of them follow Michael's design [41]. The first three per-thread shared variables are (*entry** *prev*), (*entry** *cur*), and (*entry** *next*). The other two are the two pointers (*entry** *hp0*) and (*entry** *hp1*) that point to the two hazard pointers of the thread. All other variables are local to the method that mentioned them.

2.4.1 The insert operation

The *InsertToChunk* method inserts a key with its associated data into a chunk. It first attempts to find an available entry and allocate it with the given key. If no available entry exists, a split is executed and the operation is retried. If an entry is obtained, the *InsertEntry* method is invoked to insert the entry into the

list. The insertion will fail if the key already exists in the chunk. In this case *InsertToChunk* clears the entry to free it for future allocations.

The *InsertToChunk* code is presented in Algorithm 1. It starts by an attempt to find an available entry for allocation. A failure occurs when all entries are in use and in this case a freeze is initiated. The *Freeze* method gets the key and data as an input, and also an input indicating that it is invoked by an insertion operation. This allows the *Freeze* method to try to insert the key to the newly created chunk. When successful, it returns a NULL pointer to indicate the completion of the insertion. It also sets a local variable *result* to indicate whether the completed insertion actually inserted the key or it completed by finding that the key already existed in the list (which is also a legitimate completion of the insertion operation). If the insertion is not completed by the *Freeze* method, then it returns a pointer to the chunk on which the insertion should be retried.

Connecting the entry to the list is done by *InsertEntry*. If the entry gets allocated and linked to the list, then the chunk counter is incremented only by the thread that linked the entry itself. If the key already existed in the list, then *ClearEntry* attempts to clear the entry for future reuse. However, a rare scenario may foil clearing of the entry. This happens when the other occurrence of the key (which existed previously in the list) gets deleted before our entry gets cleared. Furthermore, a freeze occurs, in which the semi-allocated entry gets linked by other threads into the new chunk's list. At this point, clearing this entry is avoided, and *ClearEntry* returns FALSE. In such a scenario, clearing the entry fails and the insert operation succeeds.

At the end of *InsertToChunk*, all hazard pointers are cleared and we return with a code specifying if the insert was successful, or the key previously existed in the list.

The allocation of an available entry is executed using the *AllocateEntry* method, depicted in Algorithm 2. An available entry contains $\perp$ as a key and zeros otherwise. An available entry is allocated by assigning the key and data values in the *keyData* word in a single atomic compare-and-swap (CAS) that assumes this word has the $\perp$ symbol and zeros in it. An entry whose *keyData* has the freeze bit set cannot be allocated as it is not properly zeroed. Note also that once an entry is allocated, all the information required for linking it to the list is available to all threads. Thus, if a freeze starts, then all threads may create a stabilized list of the allocated entries in a chunk. The *AllocateEntry* method searches for an available entry. If no free entry can be found, NULL is returned.

Next, comes the *InsertEntry* method, which takes an allocated entry and attempts to link it to the linked list. The *InsertEntry* code is presented in Algorithm 3. The input parameter *entry* is a pointer to an entry that should be inserted. It is already allocated and initiated with key and data.

Before searching for the location to which to connect this entry, we memorize this entry's next pointer. Normally, this should be a NULL, but in the presence of concurrent executions of *InsertEntry* (which may happen during a freeze), we

Algorithm 1: Insert a key and its associated data into a chunk

```
Bool InsertToChunk (chunk* chunk, key, data) {
 1: current = AllocateEntry(chunk, key, data);                    // Find an available entry
 2: while ( current == NULL ) {                                   // No available entry. Freeze and try again
 3:    chunk = Freeze(chunk, key, data, INSERT, &result);
 4:    if ( chunk == NULL ) return result;                        // Freeze completed the insertion.
 5:    current = AllocateEntry(chunk, key, data);                 // Otherwise, retry allocation
 6: }
 7: returnCode = InsertEntry(chunk, current, key);
 8: switch ( returnCode ) {
 9:    case SUCCESS_THIS:
10:       IncCount(chunk); result = TRUE; break;                  // Increase the chunk's counter
11:    case SUCCESS_OTHER:                                        // Entry was inserted by other thread
12:       result = TRUE; break;                                   // due to help in freeze
13:    case EXISTED:                                              // This key exists in the list. Reclaim entry
14:       if ( ClearEntry(chunk, current) )                       // Attempt to clear the entry
15:                result = FALSE;
16:       else                                                    // Failure to clear the entry implies that a freeze thread
17:                result = TRUE;                                 // eventually inserts the entry
18:       break;
19: } // end of switch
20: *hp0 = *hp1 = NULL;
21: return result;
}
```

Algorithm 2: Entry allocation

```
entry* AllocateEntry(chunk* chunk, key, data) {
1: keyData = combine(key, data);                 // Combine into the structure of a keyData word
2: expEnt = combine(⊥, 0);
3: foreach entry e                                                // Traverse entries in chunk
4:    if ( e→keyData == expEnt )
5:       if ( CAS(&(e→keyData), expEnt, keyData) ) return e;      // Try to allocate
6: return NULL;                                                   // No free entry was found
}
```

must make sure later that the entry's next pointer was not modified before we atomically wrote it in Line 10. After saving the current next pointer, we search for the entry's location via the *Find* method. If the key already exists in the list, *InsertEntry* checks whether the returned entry is the same as the one it is trying to insert (by address comparison). The result determines the return code: either the key existed and we failed, or the key was inserted, but not by the current thread. (This can happen during a freeze when all threads attempt to stabilize the frozen list.) Otherwise, the key does not exist, and *Find* sets the global variable *cur* with a pointer to the entry that should follow our entry in the list, and the global variable *prev* with the pointer that should reference our entry. The *Find* method protects the entries referenced by *prev* and *cur* with the hazard pointers *hp1* and *hp0*, respectively. There is no need to protect the newly allocated entry because it cannot be reclaimed by a different thread.

If any to-be-modified pointer is marked as frozen, we make sure that its re-

Algorithm 3: Connecting an allocated entry into the list

```
returnCode InsertEntry (chunk* chunk, entry* entry, key) {
 1: while ( TRUE) {
 2:    savedNext = entry→next;
 3:    // Find insert location and pointers to previous and current entries (prev, cur)
 4:    if ( Find(chunk, key) )                          // This key existed in the list
 5:      if ( entry == cur ) return SUCCESS_OTHER; else return EXISTED;
 6:    // If neighborhood is frozen, keep it frozen
 7:    if ( isFrozen(savedNext) ) markFrozen(cur);      // cur will replace savedNext
 8:    if ( isFrozen(cur) ) markFrozen(entry);          // entry will replace cur
 9:    // Attempt linking into the list. First attempt setting next field
10:    if ( !CAS(&(entry→next), savedNext, cur) ) continue;
11:    if ( !CAS(prev, cur, entry) ) continue;          // Attempt linking
12:    return SUCCESS_THIS;                             // both CASes were successful
13: }
 }
```

placement is marked as frozen well. An allocation of an entry can never occur on a frozen entry. However, once the allocation is successful, the new entry may freeze and still *InsertEntry* should connect it to the list. Finally, two CASs are used to link the entry to the list. Whenever a CAS fails, the insertion starts from scratch.

2.5 The Freeze Procedure

We now provide more details about the freeze procedure. The freezing process occurs when the number of entries in a chunk exceeds its boundaries. At this point, splitting or merging happens by copying the relevant keys (and data) into a newly allocated chunk (or chunks). This process comprises three phases: *initiation*, *stabilization* and *recovery*.

A thread will initiate a freeze when the MIN or MAX are exceeded or when there is an external request to freeze the current chunk in order for it to serve in a merge procedure with a neighbor. Once a freeze process has started, other threads may join and help the freeze since they need the resulting chunks to proceed in their activities. We distinguish between an external freeze (imposed upon a chunk by a sparse neighbor) and a regular freeze, using the freeze state field in the chunk. The main goal is to avoid the possibility that one thread plans to use this chunk for merging with a small neighbor, while another thread is splitting it because it is full. We let one type of freeze terminate before executing a different one.

The code for the *Freeze* method is presented in Algorithm 4. The input parameters are the chunk that needs to be frozen, the key, the data, and the event that triggered the freeze: INSERT, DELETE, ENSLAVE (if the freeze was called to prepare the chunk for merge with a neighboring chunk), or NONE (if the freeze is called while clearing an entry). The freeze will attempt to execute the insertion, deletion, or enslaving and will return a NULL pointer when successful. It will also set an input boolean flag to indicate the return code of the relevant operation. When

Algorithm 4: The main freeze method.

```
chunk* Freeze(chunk* chunk, key, data, triggerType tgr, Bool* res) {
 1: CAS(&(chunk→freezeState), NO_FREEZE, INTERNAL_FREEZE);
 2: // At this point, the freeze state is either INTERNAL_FREEZE or EXTERNAL_FREEZE
 3: MarkChunkFrozen(chunk);
 4: StabilizeChunk(chunk);
 5: if ( chunk→freezeState == EXTERNAL_FREEZE ) {
 6:    // This chunk was marked EXTERNAL_FREEZE before Line 1 executed.
 7:    master = chunk→mergeBuddy;                          // Get the master chunk
 8:    // Fix the buddy's mergeBuddy pointer.
 9:    masterOldBuddy = combine(NULL, INTERNAL_FREEZE);
10:    masterNewBuddy = combine(chunk, INTERNAL_FREEZE);
11:    CAS(&(master→mergeBuddy), masterOldBuddy, masterNewBuddy);
12:    return FreezeRecovery(chunk→mergeBuddy, key, data, MERGE, chunk, tgr, res);
13: }
14: decision = FreezeDecision(chunk);                    // The freeze state is INTERNAL_FREEZE
15: if ( decision == MERGE ) mergePartner = FindMergeSlave(chunk);
16: return FreezeRecovery(chunk, key, data, decision, mergePartner, trigger, res);
 }
```

unsuccessful, it will return a pointer to the new chunk on which the operation should be retried.

The *Freeze* method starts with an attempt to atomically change the freeze state from NO_FREEZE to INTERNAL_FREEZE. This freeze state of the chunk is normally NO_FREEZE and is switched to INTERNAL_FREEZE when a freeze process of this chunk begins. But it can also be EXTERNAL_FREEZE when a neighbor requested a freeze on this chunk to allow a merge between the two. Thus, an external freeze can start even when no size violation is detected in this chunk. The location of the freeze state field is in the three LSBs of the merge buddy pointer (see Figure 2.2). In the discussion below we assume we are dealing with an internal freeze. An external freeze is discussed separately in Section 2.5.4.

Whether or not the modification succeeds, we know that the freeze state can no longer be NO_FREEZE. It can be either INTERNAL_FREEZE or EXTERNAL_FREEZE. The *Freeze* method then calls *MarkChunkFrozen* to mark each entry in the chunk as frozen and *StabilizeChunk* to finish stabilizing the entries list in the chunk. At this point, the entries in the chunk cannot be modified anymore. *Freeze* then checks if the freeze is external or internal.

An external freeze can occur when a freeze is concurrently executed on the next chunk, and it has already enslaved the current chunk as its merge buddy. In this case, we cooperate with the joint freeze and joint recovery. When the state of the freeze is external, then the current chunk must have its *mergeBuddy* pointer already pointing to the chunk that initiated the merge, denoted the *master* chunk. To finish this freeze, we make sure that the master chunk has its merge buddy properly pointing back at the current chunk. The master chunk's *mergeBuddy* pointer must be either NULL or already pointing to the buddy we found. Thus it is enough to use one CAS command to verify that it is not NULL. Finally, we

Algorithm 5: Freezing all entries in a chunk

```
void MarkChunkFrozen(chunk* chunk) {
 1: foreach entry e {
 2:    savedWord = e→next;
 3:    while ( !isFrozen(savedWord) ) {                    // Loop till the next pointer is frozen
 4:       CAS(&(e→next), savedWord, markFrozen(savedWord));
 5:       savedWord = e→next;                                // Reread from shared memory
 6:    }
 7:    savedWord = e→keyData;
 8:    while ( !isFrozen(savedWord) ) {                    // Loop till the keyData word is frozen
 9:       CAS(&(e→keyData), savedWord, markFrozen(savedWord));
10:       savedWord = e→keyData;                          // Reread from shared memory
11:    }
12: } // end of foreach
13: return;
}
```

execute the recovery phase on the master chunk and return its output. We do not need to check the decision about the freeze of the buddy. It must be a merge.

If the freeze is internal, then we invoke *FreezeDecision* to see what should be done next (Line 14). If the decision is to merge, then we find the previous chunk and "enslave" it for a joint merge using the *FindMergeSlave* method. Finally, the *FreezeRecovery* method is called to complete the freeze process. Next, we explain each of the stages.

2.5.1 The initiation of a freeze

After changing the chunk's state, the initiation invokes *MarkChunkFrozen*, which goes through the chunk's entries one by one and marks them as frozen by setting the freeze bit first on the *nextEntry* word and then on the *keyData* word. The setting of these flags is atomic and it is retried repeatedly until successful. By the end of this process all entries (including the free ones) are marked as frozen. The freeze bit of the *head* entry is set last and at this point in time we consider the initiation phase to be completed. The pseudo code of *MarkChunkFrozen* is presented in Algorithm 5.

2.5.2 The stabilization phase

After all the entries in the chunk are marked as frozen, new entries cannot be allocated and existing entries cannot be marked as deleted. However, the frozen chunk may contain allocated entries that were not yet linked, and entries that were marked as deleted, but which have not yet been disconnected and reclaimed. The stabilization operation disconnects all deleted entries and links all allocated ones. The pseudo-code of the *StabilizeChunk* method appears in Algorithm 6. It starts by running *Find* on the maximal possible key value. This is done because the *Find* method (described in Section 2.6.1) always disconnects all entries that are marked as deleted (even when frozen). Such entries do not need to be reclaimed

Algorithm 6: Freeze stabilization.

```
void StabilizeChunk(chunk* chunk) {
1: maxKey = ∞;
2: Find(chunk, maxKey);                                   // Implicitly remove deleted entries
3: foreach entry e {
4:     key = e→key; eNext = e→next;
5:     if ( (key != ⊥) && (!isDeleted(eNext)) )          //This entry is allocated and not deleted
6:         if(!Find(chunk,key)) InsertEntry(chunk, e, key);   //This key is not yet in the list
7: } // end of foreach
8: return;
}
```

(when marked as frozen), but they should not be copied to the new chunk. Next, *StabilizeChunk* attempts to connect entries. It goes over all entries and searches for ones that are disconnected, but neither reclaimed nor deleted. Each such entry is linked to the list by invoking *InsertEntry*, which will only fail if the key already exists in a different entry in the chunk's list. In this case, this entry should indeed not be connected to the stabilized list.

2.5.3 The decision and the recovery

After stabilizing the chunk, everything is frozen, the list is completely connected, and nothing changes in the chunk anymore. At this point, we need to decide whether or not splitting or merging is required. Recall that the decision to freeze is initiated in the presence of many concurrent updates. It is possible that one thread could not find an entry to allocate and initiated a freeze for the purpose of splitting the chunk, but many other threads deleted entries concurrently and when the chunk actually stabilized, there was no need to split. There may even be a need for a merge. Thus, we make the decision on which operation to execute only after the chunk has stabilized and cannot change anymore.

At this point, we count the number of entries in the frozen chunk, and decide if a split or a merge is required according to the count. If the resulting count equals MIN we run a merge, and if it equals MAX, we run a split (in the recovery phase). The resulting count can never exceed the bounds, because there is no space to allocate more than MAX entries in the chunk, and since the chunk counter that is maintained during the run holds a lower bound on the actual number of entries and can never reach a value below MIN. If the resulting count is higher than MIN and lower than MAX, then no operation is required. Nevertheless, the frozen chunk is never resurrected. Otherwise, correctness cannot be guaranteed when a long-sleeping thread wakes to find a chunk that was resurrected. Instead, we copy the chunk to a new chunk in the (upcoming) recovery stage.

The *FreezeDecision* method is presented in Algorithm 7. It computes the number of entries and returns the recovery code: SPLIT, MERGE, or COPY.

The recovery procedure allocates a chunk (or two) and copies the relevant information into the new chunk (or chunks). If a merge is involved, the previous

Algorithm 7: Determining the freeze action.

```
recovType FreezeDecision (chunk* chunk) {
1: entry* e = chunk→head→next;
2: int cnt = 0;
3: while (clearFrozen(e)!=NULL) { cnt++; e=e→next; }        // Going over the chunk's list
4: if ( cnt == MIN) return MERGE;
5: if ( cnt == MAX) return SPLIT;
6: return COPY;
}
```

chunk in the list is first frozen (externally) and both chunks bring entries for the merge. Several threads may perform the freeze procedure concurrently, but all of them will make the same recovery decision about the freeze, as the frozen stabilized chunk looks the same to all threads. A thread that performs the recovery creates a local chunk into which it copies the relevant entries. At this point all threads create the same new chunk (or chunks). But now, each thread performs the operation with which it initiated the freeze on the new chunks. It can be an insert, delete, or enslave. Performing the operation is easy because the new chunks are local to this thread and no race can occur. (Enslaving a chunk is simply done by modifying its freeze state from NO_FREEZE to EXTERNAL_FREEZE and registration of the merge buddy.) But the success of making the local operation visible in the data structure is determined by whether the thread succeeds in creating a link to its new chunks in the frozen chunk, as explained next.

After creating the new chunks locally and executing the original operation on them, there is an attempt to atomically insert the address of its local chunk into a dedicated pointer in the frozen chunk (*new*). When two chunks are created, the second one is locally linked to the first one by the *nextChunk* field. If the insertion is successful, then this thread has also completed the the operation it was performing (insert, delete, or enslave). If the insertion is unsuccessful, then this means that a different thread has already completed the installation of new chunks and this thread's local new chunks will not be used (i.e., can be reclaimed). In this case, the thread must try its operation again from scratch.

The code for the recovery is presented in Algorithm 8. If a merge occurs, the merging chunk is supplied as a parameter. According to the number of (live) entries on the frozen chunk there are three ways to recover from the freeze.

Case I: MIN$<$ *count* $<$ MAX**.** In this case, the required action is to allocate a new chunk and copy all of the entries that reside on the frozen chunk's list to the new chunk (which is only locally visible and requires no synchronization). We do not specify the copying routine (in this case, as well as in the other cases) since the copy is from a frozen chunk that does not change, to a local chunk. This means that no concurrency is involved and the implementation is simple. The new chunk becomes the replacement of the old chunk when the pointer *new* in the old chunk points to it. An upper-level routine that handles the chunked list *ListUpdate* is then invoked to replace the frozen chunk with the chunk that is referenced by *new*.

The new chunk that holds the input key (after the freeze is completed) is then returned.

Case II: *count* == MIN**.** In this case we need to merge the old chunk with its previous chunk supplied through *mergeChunk*. We assume that the supplied chunk has already been frozen by an external freeze before the recovery is executed. Finally, we assume that the freeze states are properly set to internal on the old chunk and external on the previous chunk (so that no thread can interfere with the freeze process), and the *mergeBuddy* pointers on these two chunks point to each other.

We start by checking the overall number of entries in these two chunks, to decide if the merged entries will fit into one of two chunks. We then allocate a second new chunk, if needed, and perform the (local) copy to the new chunk or chunks. When copying into two new chunks, we split the entries evenly, and return the smallest key in the second chunk as the *separating key*. As before, we try to create a link from the old chunk to the new chunk or chunks. Next, the new chunk that holds the input key is determined according to the *separating key*, and finally, the *ListUpdate* method is called to replace the frozen chunk in the list with the two new chunks. This completes the recovery for the merge case.

Case III: *count* == MAX**.** In this case we need to split the old chunk into two new chunks. The basic operations of this case resemble those of the previous cases. We allocate a new chunk, perform the split locally, attempt to link the new chunks to the old one, update the list, and return the chunk holding the key.

2.5.4 Managing the external freeze activities

An external freeze can occur when a freeze is concurrently executed on the next chunk, and it has already enslaved the current chunk as its merge buddy. In this case, we cooperate with the joint freeze and joint recovery Lines 5-13 of the *Freeze* method. When the state of the freeze is external, then the current chunk must have its *mergeBuddy* pointer already pointing to the chunk that initiated the merge, denoted the *master* chunk. To finish this freeze, we make sure that the master chunk has its merge buddy properly pointing back at the current chunk. The master chunk's *mergeBuddy* pointer must be either NULL or already pointing to the buddy we found. Thus it is enough to use one CAS command to verify that it is not NULL. Finally, we execute the recovery phase on the master chunk and return its output. We do not need to check the decision about the freeze of the buddy. It must be a merge.

Algorithm 8: The freeze recovery.

```
chunk* FreezeRecovery(chunk* oldChunk, key, input, recovType, chunk* mergeChunk,
triggerType trigger, Bool* result) {
 1: retChunk=NULL; newChunk2=NULL; newChunk1=Allocate();                    // Allocate a new chunk
 2: newChunk1→nextChunk = NULL';
 3: switch ( recovType ) {
 4:    case COPY:
 5:      copyToOneChunk(oldChunk, newChunk1); break;
 6:    case MERGE:
 7:      if ( (getEntrNum(oldChunk)+getEntrNum(mergeChunk))≥MAX ) {
 8:        // The two neighboring old chunks will be merged into two new chunks
 9:        newChunk2 = Allocate();                                          // Allocate a second new chunk
10:        newChunk1→nextChunk = newChunk2;                                 // Connect two chunks together
11:        newChunk2→nextChunk = NULL';
12:        separatKey=mergeToTwoChunks(oldChunk,mergeChunk,newChunk1,newChunk2);
13:      } else mergeToOneChunk(oldChunk,mergeChunk,newChunk1);
14:      break;
15:    case SPLIT:
16:      newChunk2 = Allocate();                                            // Allocate a second new chunk
17:      newChunk1→nextChunk = newChunk2;                                   // Connect two chunks together
18:      newChunk2→nextChunk = NULL';
19:      separatKey = splitIntoTwoChunks(oldChunk, newChunk1, newChunk2); break;
20: } // end of switch
21: // Perform the operation with which the freeze was initiated
22: switch ( trigger ) {
23:    case DELETE:                                  // If key will be found, decrement counter has to succeed
24:      *result = DeleteInChunk(newChunk1, key);
25:      if ( newChunk2 != NULL ) *result = *result || DeleteInChunk(newChunk2, key);
26:      break;
27:    case INSERT:                                  // input should be interpreted as data to insert with the key
28:      if((newChunk2!=NULL)&&(key<separatKey)) result=InsertToChunk(newChunk2,key,input);
29:      else *result = InsertToChunk(newChunk1, key, input);
30:      break;
31:    case ENSLAVE:                        // input should be interpreted as pointer to master trying to enslave
32:      if (newChunk2!=NULL) newChunk2→mergeBuddy=combine(input, EXTERNAL_FREEZE);
33:      else newChunk1→mergeBuddy = combine(input, EXTERNAL_FREEZE);
34: } // end of switch
35: // Try to create a link to the first new chunk in the old chunk.
36: if ( !CAS(&(oldChunk→new), NULL, newChunk1) ) {
37:    RetireChunk(newChunk1); if (newChunk2) RetireChunk(newChunk2);
38:    // Determine in which of the new chunks the key is located.
39:    if ( key<separatKey ) retChunk=oldChunk→new; else retChunk=FindChunk(key);
40: } else { retChunk = NULL; }
41: ListUpdate(recovType, key, oldChunk);                                  // User defined function
42: return retChunk;
 }
```

If the freeze is internal, then we invoke *FreezeDecision* to see what should be done next (Line 14). If the decision is to merge, then we find the previous chunk and "enslave" it for a joint merge using the *FindMergeSlave* method (explained below). Finally, the *FreezeRecovery* method is called to complete the freeze process.

Let us now explain the *FindMergeSlave* method, which is presented in Algo-

Algorithm 9: Setting a chunk partner for a merge.

```
chunk* FindMergeSlave(chunk* master) {
 1: while ( TRUE ) {                  // Find a slave and set its freeze state & mergeBuddy pointer
 2:    slave = listFindPrevious(master);          // upper-level funciton returning previous chunk.
 3:    // Set slave's mergeBuddy pointer and freeze state (both reside on the same word).
 4:    expected = combine(NULL, NO_FREEZE);
 5:    new = combine(master, EXTERNAL_FREEZE);
 6:    if ( !CAS(&(slave→mergeBuddy), expected, new) ) {
 7:       if ( slave→mergeBuddy == new ) break;              // Someone else has set it right.
 8:       Freeze(chunk,0,master,ENSLAVE,&result);      //The slave is under a different freeze, help
 9:    } else break;
10: }                                                              // end of while
11: MarkChunkFrozen(slave);
12: StabilizeChunk(slave);
13: // slave is externally frozen - make sure the master's mergeBuddy points to the slave.
14: expected = combine(NULL, INTERNAL_FREEZE);          // Combine two values in one word
15: new = combine(slave, INTERNAL_FREEZE);
16: CAS(&(master→mergeBuddy), expected, new);
17: return slave;
}
```

rithm 9. This method finds the previous chunk, sets its freeze state and *mergeBuddy* pointer, initiates its freeze, stabilizes it, and sets the current *mergeBuddy* to point at the obtained chunk. This method starts by invoking the (upper-level) *listFindPrevious* method in order to find the chunk that precedes the current chunk. Sometimes, because of concurrent activity, *listFindPrevious* does not find its input chunk in the list (since it was already frozen and disconnected from the list of chunks). In this case, it cannot identify the previous chunk, and instead, it just returns the *mergBuddy* pointer, which properly points to its slave for the merge (that was already completed in a concurrent manner).

We denote the previous chunk *a slave* as it joins the merge initiated by the input chunk, which is the *master*. After identifying the slave, we attempt to atomically modify its freeze state and merge buddy to indicate an external freeze joint with the master chunk. Once the slave is marked with an external freeze, the two chunks are destined for a joint freeze and no chunk can come between them. (New chunks are only added as a result of a split.) If the change in the slave state fails, a search for a new slave is attempted, after making sure that the current one is out of the way, by participating in completing its current freeze. Next, we ensure that the master's chunkBuddy pointer points to the slave and then a pointer to the slave is returned to the caller.

2.6 The Details of the Additional Chunk-level Methods

2.6.1 The search operation

The search operation, implemented in the *searchInChunk* method of Algorithm 10, uses the *Find* method, described hereafter. The *searchInChunk* method starts by

Algorithm 10: Searching for data associated with the key

```
Bool SearchInChunk (chunk* chunk, key, *data) {
1: if ( Find(chunk, key) ) { data = cur→data; result = TRUE; } else result = FALSE;
2: *hp0 = *hp1 = NULL; return result;
}
```

Algorithm 11: Find the location of an entry in the chunk's list

```
Bool Find (chunk* chunk, key) {
1: try_again: prev = &(chunk→head);                                   // Restart point
2: cur = *prev;
3: while ( clearFrozen(cur) != NULL ) {          // Ignore freeze bit when comparing to NULL
4:    *hp0 = cur;                                          // Progress to an unprotected entry
5:    if ( *prev != cur ) goto try_again;                  // Validate progress after protecting
6:    next = cur→next;
7:    if ( isDeleted(next) ) {                             // Current entry is marked deleted
8:       if ( isFrozen(cur) ) markFrozen(next);            // next replaces cur; save freeze bit
9:       // Disconnect current: prev gets the value of next with the delete bit cleared
10:      if ( !CAS(prev, cur, clearDeleted(next)) ) goto try_again;
11:      RetireEntry(cur);                                 // CAS succeeded - try to reclaim
12:      cur = clearDeleted(next);
13:   } else {
14:      ckey = cur→key;
15:      if (*prev!=cur) goto try_again;          // Check new insert between them or new delete
16:      if (ckey ≥ key) return (ckey == key);
17:      prev = &(cur→next);
18:      tmp = hp0; hp0 = hp1; hp1 = tmp;         // All private. hp0, hp1 are ptrs to hazard ptrs
19:      cur = next;
20:   }
21: }
22: return FALSE;
}
```

call for *Find*, which protects its output with hazard pointers. The *SearchInChunk* method finishes by clearing the hazard pointers and returning.

Finding the location in the chunk's list: the *Find* method. We now present the *Find* method, invoked by several other methods. The pseudo-code for *Find* appears in Algorithm 11. This method finds the location of a given key in the list. It returns FALSE if the key does not exist in the list, or TRUEotherwise. It also sets in a global (indirect) pointer ***prev* to the entry that contains the highest key value between all keys smaller than the input key, and in a global pointer **cur*, the entry with the minimal key value that is larger or equal to the input key. Finally, if the key is found, the entry that follows *cur* is returned in a global pointer **next*. *Find* is very similar to the Find method presented in [41] up to changes needed for dealing with the freeze bit.

The *Find* method protects the entries that it uses and returns using hazard pointers so they are not being concurrently reclaimed. This holds upon return from *Find* so the calling method may assume that the referenced entries could not

be reclaimed and re-allocated, until the calling method clears the thread's hazard pointers.

If the traversal of *Find* finds an entry that is marked for deletion (i.e., the delete bit is set on its *next* pointer), then it disconnects the entry from the list and attempts to recycle it. Recycling is executed via *RetireEntry*, which is explained later in Section 2.6.2. Disconnecting and recycling a deleted entry is a service of *Find* to the structure of the list that will be assumed in the rest of this chapter. The key of a deleted entry is not checked, and cannot influence the search for the input key.

Any failing CAS causes a restart of the search. Also, in general, whenever we replace a pointer by another, e.g., in Line 8, we first make sure that if the old pointer was marked as frozen, then the replacement pointer is marked as frozen as well. This way the freeze bits of an entry are preserved everywhere.

2.6.2 The delete operation

The deletion algorithm (inside a chunk) is similar to the well-known one for lock-free linked lists [22, 41]. The deletion operation is partitioned into a logical deletion, which marks the entry as deleted by setting the delete bit (LSB) in the entry's *next* pointer. Next, the physical deletion disconnects the entry from the list and reclaims its space. The difference between our deletion method and the standard one is the need to check if the chunk's counter has reached the lower threshold MIN and call *Freeze* when it does. Additionally, we do not let the delete bit be set on a frozen entry. A delete can only occur before an entry gets frozen. Notice that *Freeze* can also help this deletion and we check if help happened any time *Freeze* is invoked. Finally, we need to maintain the counter of entries allocated in the chunk. In order to make sure that the counter holds a lower bound on the number of entries in the presence of concurrent updates, we decrement the counter before we delete the entry. If the delete fails, we increment the counter to account for the failure. A failure to decrement the counter can only happen when the lower bound has been reached. In this case, we initiate a freeze, which returns with a new chunk (containing the range of values that includes our input key). The decrement attempt is then repeated and this loop repeats until the decrement succeeds on the current chunk.

The deletion algorithm (inside a chunk) is presented in Algorithm 12. It starts by decrementing the counter, *Find* is invoked to find the entry holding the key. If the key does not exist in the list, then the counter is incremented, hazard pointers zeroed and FALSE is returned to the caller. Otherwise, we attempt a CAS to mark the entry as deleted. The CAS assumes that the freeze bit and the delete bit are not set at the deletion time (for proper counter measurement, we should know exactly who sets the delete bit). If the CAS fails due to a freeze bit, then a freeze action must be executed, then either freeze succeeded to promote this deletion or the delete should restart on the newly obtained chunk. Otherwise, the CAS failed due to some other thread deleting the entry, or a pointer modification. In this

Algorithm 12: The pseudo-code of deletion of an entry in a chunk

```
    Bool DeleteInChunk (chunk* chunk, key) {
1:  try_ again:
2:  while ( !DecCount(chunk) ) {                          //If too few entries in chunk; call freeze
3:      chunk = Freeze(chunk, key, 0, DELETE, &result);
4:      if ( chunk == NULL ) return result; // If Freeze succeeded to proceed with deletion, return
5:  } // end of decrement counter while
6:  while ( TRUE ) {
7:      if ( !Find(chunk, key) ) {
8:          IncCount(chunk); *hp0 = *hp1 = NULL; return FALSE;        // No such entry was found
9:      }
10:     // Mark entry as deleted, assume entry is not deleted or frozen
11:     clearedNext = clearFrozen(clearDeleted(next));
12:     if ( !CAS(&(cur→next), clearedNext, markDeleted(clearedNext)) ) {
13:         if ( isFrozen(cur→next) ) {                                  // CAS failed due to freeze
14:                 IncCount(chunk); chunk = Freeze(chunk, key, 0, DELETE, &result);
15:                 if ( chunk == NULL ) return result;   // If Freeze succeeded to delete, return
16:                 goto try_ again;
17:         } else continue;
18:     }
19:     // Remove entry
20:     if ( isFrozen(cur) ) markFrozen(next);                  // next replaces cur; retain freeze bit
21:     if ( CAS(prev, cur, next) ) RetireEntry(addr); else Find(chunk, key);
22:     *hp0 = *hp1 = NULL; return TRUE;
23: }
    }
```

case, we should search for the entry again before deleting it. The *Find* method will not return this entry again if it has already been deleted. Furthermore, it will disconnect it from the list and reclaim it before returning.

After marking the entry as deleted, we attempt to disconnect it from the list. If the freeze bit is set, we keep it set. If the disconnect succeeds, we reclaim the entry via *RetireEntry*. Otherwise, we call *Find*, which repeatedly attempts to disconnect an entry that is marked deleted, until the disconnection is achieved. Finally, we clear the hazard pointers that are set by the *Find* method, to allow future reclamation of the involved entries.

Entry reclamation Special care is required for reclaiming an entry in the presence of hazard pointers. First, it must be clear that the reclamation is not being executed on an entry that has a hazard pointer, and second, if an entry cannot be reclaimed right now, it will be properly scheduled for future reclamation (in a non-blocking manner). We follow the scheme presented by Michael in [41]. In this scheme, an entry can be reclaimed only by the very same thread that disconnects it from the list. There can only be one such thread, as the disconnection is executed with a CAS.

Each thread has its own list of to-be-retired entries. After successfully disconnecting an entry, the thread invokes *RetireEntry* (depicted in Algorithm 13), which pushes the given entry into the list of entries waiting to be reclaimed, and

Algorithm 13: The reclamation code employs Michael's reclamation scheme

```
    void RetireEntry (entry* entry) {
 1: addToRetList(entry);                  // Add the entry to the (local) list of to-be-retired entries
 2: HandleReclamationBuffer();            // Scan the list and reclaim the entries if possible
    }
    void HandleReclamationBuffer() {
 1: plist = initializeList();             // Local list for recording current hazard pointers
 2: hprec = getHPhead();                  // Obtain head of hazard pointers array (HPA)
 3: //Stage 1: Save current hazard pointers in plist (locally)
 4: while ( hprec != NULL ) {
 5:    for ( i=0; i<2; ++i ) {                                  // 2 hazard pointers per thread
 6:       hptr = hprec→HP[i];
 7:       if ( hptr != NULL ) insertList(plist, hptr);
 8:    }
 9:    hprec = getNextHPrecord(hprec);
10: }
11: // Stage 2: Reclaim to-be-retired entries that are not protected by a hazard pointer
12: tmplist = popAllRetList();            // Copy all local to-be-retired entires and clear RetList
13: entry = popList(tmplist);
14: while ( entry != NULL ) {
15:    if ( lookUp(plist, entry) ) pushRetList(entry);      // Entry protected, push back to RetList
16:    else { if ( !isFrozen(entry) ) ClearEntry(entry); } // Reclaim unprotected (non-frozen) entry
17:    entry = popList(tmplist);
18: }
19: freeList(plist);
    }
```

then attempts to reclaim all entries in the list via the *HandleReclamationBuffer* method. The *HandleReclamationBuffer* method compares the entries in the to-be-retired list with the ones in the hazard pointers array (HPA) and reclaims the entries that do not appear in the HPA. Our adaptation to this scheme does not reclaim entries marked as frozen even when no hazard pointer points to them. The *HandleReclamationBuffer* method is invoked on every *RetireEntry* call, in order to make sure that an entry is reclaimed as soon as possible, when no more hazard pointers point to it. Michael's reclamation scheme, slightly modified to support our notations, is depicted in Algorithm 13. For further discussion on the reclamation scheme we refer the reader to the *RetireNode* method and the *Scan* method in [41].

The actual clearing of an entry in our list means zeroing the entry and assigning $\perp$ as key's value. This is executed in the *ClearEntry* method depicted in Algorithm 14. This method is invoked either in case of a trial to insert a key that already existed in the chunk's list (*InsertToChunk*, Line 14) or by *HandleReclamationBuffer* when a deleted and disconnected entry is found to be safe for reuse (in Line 16 there). We do not reclaim an entry when it is found frozen, because this reclamation is not needed anymore, and it complicates the code to reclaim it.

The entry clearance is executed by two CAS operations. When the *keyData* word is cleared, the entry might immediately be re-allocated. Therefore, we first zero the *nextEntry* word, and only then put $\perp$ on the *keyData* word. An entry

Algorithm 14: The pseudo-code for clearing an entry and reclaiming it's space/

```
Bool ClearEntry (chunk* chunk, entry* entry) {
 1: savedKeyData = clearFrozen(entry→keyData);
 2: savedNext = clearFrozen(entry→next);
 3: newKeyData = combine(⊥, 0);
 4: if( CAS(&(entry→next), savedNext, 0) )
 5:    if(CAS(&(entry→keyData), savedKeyData, newKeyData) )
 6:       return TRUE;                                   // Both CASes were successful
 7: Freeze(chunk, 0, 0, NONE, &result);                  // A CAS failure indicates a freeze, help freeze
 8: // Check whether the entry to be reclaimed was linked back by the freeze
 9: if ( Find(chunk,entry→key) )
10:    if ( entry == cur ) return FALSE;                // cur is global initiated by Find
11: return TRUE;
 }
```

that is marked frozen is not reclaimed and this is ensured by the atomic CAS. We claim that a CAS can only fail when the entry's freeze bit is marked. If the chunk is not being frozen, a cleared entry is handled only by the current thread. The reason is that the entry is already disconnected from the list and no other thread has a hazard pointer to it, neither can it find the entry at this point. Furthermore, only one thread holds it in his to-be-retired list. Therefore, the clearing can only fail when a freeze process is executing.

When *ClearEntry* is called from *HandleReclamationBuffer* the freeze cannot resurrect it. The entry is deleted, and after executing the freeze procedure to make sure that it is completed, we know that the entry cannot exist in the newly created chunk anymore. However, when *ClearEntry* is called by *InsertToChunk*, the entry is not deleted and a freeze process may resurrect it (as discussed in the description of *InsertToChunk*). In this case, *ClearEntry* discovers the resurrection and returns FALSE.

2.6.3 Counter Functionalities

Here we present the lock-free counter functionalities we use in the *InsertInChunk* and *DeleteInChunk* methods.

It may happen that a thread fails or stops just before or after updating the counter, thus an accurate count for the number of entries in the chunk cannot be expected (in a lock-free execution). Our counter only ensures that the counter value is always *less or equal* to the real number of entries in the chunk's list, which is what we actually need for keeping the number of entries in the chunk between MIN and MAX. Recall that MAX is the number of entries in a chunk, and so even if the counter did not exist, no more than MAX entries could be allocated on a chunk. In order to make sure that the number of entries does not go below MIN, we maintain the counter as a lower bound on the actual number of entries. If the counter drops below MIN, we try to merge the chunk with a neighboring chunk. Since the counter is a lower bound on the actual entry number, we may find that no merging is really needed after the freeze.

Algorithm 15: The increment and the decrement of the chunk's counter

```
   void IncCount (chunk* chunk) {
1: while ( TRUE ) {
2:    counter = chunk→counter;
3:    if (CAS(&(chunk→counter),counter,counter+1) ) return;
4: }
   }
   Bool DecCount (chunk* chunk) {
5: while ( TRUE ) {
6:    counter = chunk→counter;
7:    if ( counter == MIN ) return FALSE;          // comparison with minimal, MIN-1 illegal
8:    if (CAS(&(chunk→counter),counter,counter-1) ) return TRUE;
9: }
   }
```

To ensure that the counter is a lower bound on the number of entries, we apply a couple of rules. First, the counter is only incremented *after* an entry is successfully allocated. This means that the counter does not supersede the number of entries in the chunk (MAX). Second, we decrement the counter *before* we delete an entry. So that if the executing thread halts between the counter decrement and the deletion, we know that the counter is smaller than the actual number of entries. It is never larger than it.

The code for handling the counter appears in Algorithm 15. The increment is straightforward and it always succeeds. The decrement method returns a failure if an attempt is made to reduce the counter below the MIN value.

2.7 The Upper-Level List Operations

Let us now specify the upper-level list handling. When operations, such as split or merge, are executed on a chunk, they may sometimes cause a split or a merge of chunks. In this case, the list of chunks needs to be updated. Note that the list of chunks need not handle inserts or deletes. It only handles splits and merges that follow inserts and deletes of entries inside the chunks.

We start by presenting the *ListUpdate*() method. This method is called after a chunk has finished the freeze and recovery phases, at Line 49 of *FreezeRecovery*() method. The *ListUpdate*() gets as input the recovery type (SPLIT, MERGE, or COPY), an (arbitrary) key located or should be located on the frozen chunk, and a pointer to the frozen chunk. At this point, the list on the frozen chunk is stabilized and cannot be changed anymore. The *new* field in the frozen chunk points to a new chunk that contains some of the entries copied from the frozen chunk, and, when a second new chunk is required (for SPLIT or a MERGE that ended up with two new chunks), then the *nextChunk* field of first new chunk points to the new chunk with the higher key values, copied from the frozen chunk. We also make the new chunks sequence to be finished with a special NULLpointer - NULL', in order to distinguish between NULLthat comes at the end of the upper-level list.

Algorithm 16: Update chunk list at the end of a freeze.

```
void ListUpdate(recovType, key, chunk* chunk) {
 1: while ( TRUE ) {
 2:   if ( chunk→new→nextChunk == NULL' ) // // There is only one new chunk (ref'd by new)
 3:     // Memorize the next pointer of the last chunk in the linked sub-list of the new chunks,
 4:     expected = chunk→new→nextChunk;                    // for the further update
 5:   else expected = chunk→new→nextChunk→nextChunk;       // There are two new chunks
 6:   if ( FindChunk(key) != chunk ) return; // Find the frozen chunk (and set NEXT and PREV).
 7:   // Mark the next pointer of the frozen chunk as swapped.
 8:   if ( !CAS(&(chunk→nextChunk), NEXT, markSwapped(NEXT)) ) continue;
 9:   if ( !HelpSwap(expected) ) continue;
10:   return;
11: } // end of while
   }
   Bool HelpSwap(chunk* expected) {
12: if ( CUR→new→nextChunk == NULL' ) {      // There is only one new chunk (ref'd by new)
13:   addr = &(CUR→new→nextChunk);                  // address to insert pointer to next.
14: } else {                          // There are two new chunks (last ref'd by nextChunk of new)
15:   addr = &(CUR→new→nextChunk→nextChunk);
16: }
17: if ( !CAS( addr, expected, NEXT) ) return FALSE;
18: if ( CUR→mergeBuddy == NULL) {            // For COPY and SPLIT, there is one old chunk
19:   if ( !CAS(PREV, CUR, CUR→new) ) return FALSE; else RetireChunk(chunk);
20: } else {                                  // For MERGE, there are two old chunks
21:   if (!CAS(PRE_PREV,CUR→mergeBuddy,CUR→new)) return FALSE; else
   RetireChunk(chunk);
22: } // end of if there is one old chunk
23: return TRUE;
 }
```

This is needed in order to synchronize the concurrent insertions of the chunks into upper-level list.

In addition, the least-significant bit of the *nextChunk* pointer holds a *swapped* bit, which is very similar to the *delete* bit that marks the logical delete of an entry. The *swapped* bit, when set, signifies that the chunk is about to be swapped with new chunks, and its *nextChunk* pointer cannot be modified anymore.

The code for *ListUpdate*() is presented in Algorithm 16. First, we memorize the value of *nextChunk* field of last new chunk, to be used later. After, we search for the frozen chunk by invoking the *FindChunk*() method on the input key (in the range of the old chunk). If *FindChunk*() returns a chunk different from the frozen chunk, then it means that some other thread has already removed the frozen chunk from the list of chunks, and we can just return. The search for a chunk always succeeds since each chunk has a range of keys and one of these ranges contains the input key. The *FindChunk*() method is presented later in this section (in Algorithm 17). This method also sets the global variables PRE_PREV, PREV, CUR and NEXT to point at the previous to the previous chunk (or NULL), previous chunk (or HEAD), currently found chunk that is also returned; and the next chunk (or NULL, if none exists) in the list. One important property of *FindChunk*() is that it takes care of any encountered chunk that is marked "swapped" by replacing

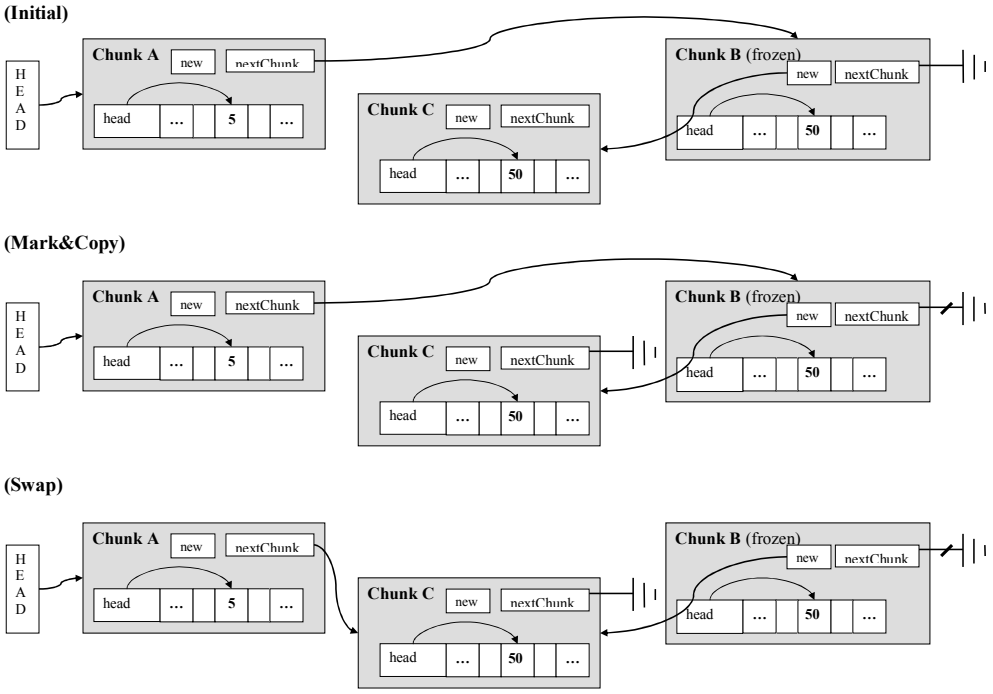


Figure 2.4: The COPY recovery in list of chunks.

it with the new chunks that should replace it.

Next, we attempt to mark the frozen chunk as swapped, by setting the least-significant bit on the *nextChunk* pointer. On failure, we start from scratch. After the *nextChunk* pointer is marked, it can not be modified anymore. Now, we attempt to link the new chunks into the list instead of the frozen chunk (and possibly a merge buddy in case of a merge). It is done in a supporting *HelpSwap*() method, also presented at Algorithm 16. In *HelpSwap*() we start by making the new chunk point to the next chunk in the chunk list. If a *nextChunk* field of *new* chunk is not NULL, then we have two chunks to insert and we make its *nextChunk* pointer point to the next chunk. Otherwise, we just have a single chunk to insert, which is pointed by *new*. In this case, we make its *nextChunk* pointer point at the next chunk in the list. The expected value is the one read before the *FindChunk*. If this setting of the *nextChunk* pointer fails, then we retry. Once the pointer to the next chunk is properly installed in the new chunks, we continue into linking it (or them) to the chunk list. In no merge is involved, we attempt to modify the previous chunk's pointer to point into the chunk referenced by *new*. If a merge is involved, then both the frozen chunk and its merge buddy (which is the chunk preceding the frozen chunk) need to be replace by the new chunks. *HelpSwap*() method make use of per thread global variables PRE_PREV, PREV, CUR and NEXT and assumes the CUR is marked as need to be swapped out.

We assume that the chunk list starts with a dummy record pointed by the global variable HEAD, and which also has a *nextChunk* field that can never be marked as swapped. We depict these steps for the COPY case of in Figure 2.4.

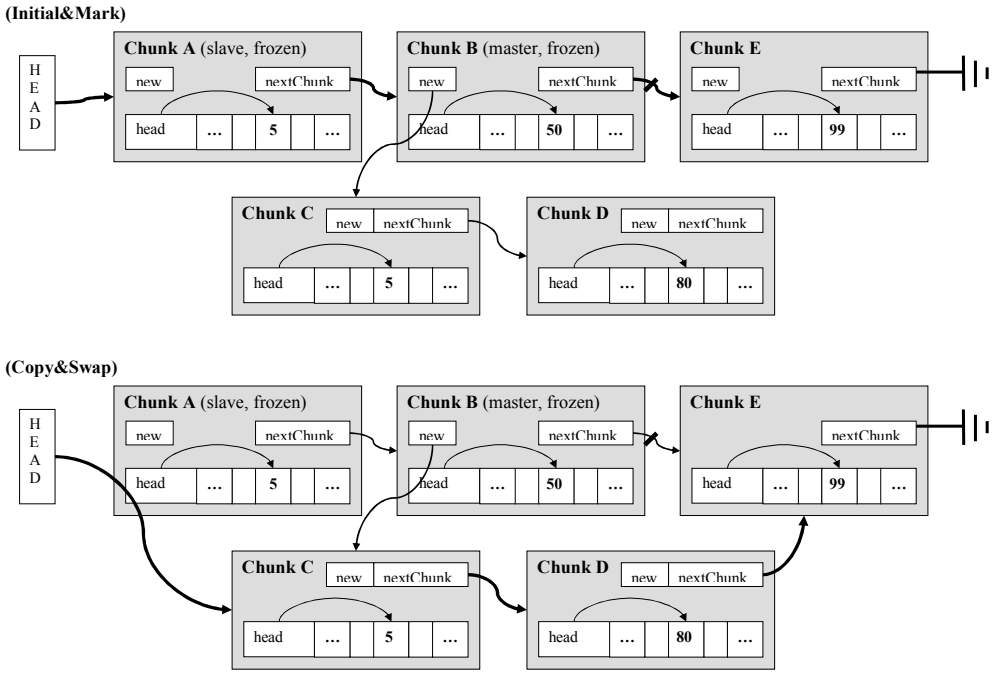


Figure 2.5: The MERGE recovery in list of chunks.

The only difference for the SPLIT case is that we have two new chunks to insert, instead of one. The MERGE case is depicted in Figure 2.5.

The *FindChunk*() method. We now proceed with describing the *FindChunk*() method, which is similar to the *Find*() method. The code for this method is presented in Algorithm 17. We use four global variables per thread. The CUR variable points to the chunk that is currently being inspected. The PREV variable points to the chunk that precedes the one pointed by CUR. (It may point to HEAD.) The PRE_PREV variable points to the chunk that precedes the chunk pointed by PREV, if one exists. Finally, the NEXT pointer points to the chunk that comes after the currently inspected one. Another hazard pointers are also here to provide the correct reclamation of the chunks. We assume another array of hazard pointers separate from one used for entries reclamations. Chunk's hazard pointers are *hp2, hp3, hp4, hp5* we need them to protect NEXT, CUR, PREV, PREV_PREV respectively. After initiation of the global variables and some hazard pointers we continue going over the list till the NULL pointer is encountered at the end of the list. For each inspected chunk we check whether it is marked as swapped. If it is, we replace it with the new chunks, similarly to the code of *ListUpdate*. If we help in merge that involves swapping out current chunk and the previous chunk we restart from the beginning of the list. This is done for simplicity of the presented code, since we can recover from the same place as well. We then check the next chunk and repeat swapping until we reach an unmarked chunk. When we have a current chunk that is not marked as swapped, we check whether we found the chunk holding a range of keys that contains the given key. If we are at the last chunk of the list, then it is the right one, as it is associated with all larger

Algorithm 17: Find the chunk whose associated range of keys contains the input key.

```
chunk* FindChunk (key) {
1: try_again:
2: PRE_PREV = NULL; PREV = &(HEAD→nextChunk); CUR = *PREV;
3: hp3* = CUR; if (*PREV != CUR) goto try_again;
4: while ( clearSwapped(CUR) != NULL ) {
5:    NEXT = CUR→nextChunk;
6:    hp2* = NEXT; if ((*PREV != CUR) || (*NEXT != CUR→nextChunk)) goto try_again;
7:    if ( isSwapped(NEXT) ) {                    // Perform swap for a logically-marked swap.
8:       if ( !HelpSwap() ) goto try_again;
9:       if ( CUR→mergeBuddy != NULL ) goto try_again;      // PREV & CUR were swapped out
10:      CUR = CUR→new;
11:      hp3* = CUR; if (*PREV != CUR) goto try_again;
12:   } else { // current chunk does not need to be swapped out
13:      if ( NEXT == NULL ) return CUR;
14:      nextKey = NEXT→head→next→key;
15:      if ((*PREV != CUR) || (*NEXT != CUR→nextChunk)) goto try_again;
16:      if ( nextKey > key ) return CUR;            // Next chunk's key range is too high.
17:      PRE_PREV = PREV;                            // Continue to next chunk
18:      PREV = &(CUR→nextChunk);
19:      tmp=hp5; hp5=hp4; hp4=hp3; hp3=hp2; hp2=tmp;      // promote the hazard
   pointers
20:      CUR = NEXT;
21:   }
22: } // end of while
}
```

Algorithm 18: Find previous chunk- High Level Method.

```
chunk* listFindPrevious (chunk* chunk) {
1: if ( FindChunk(chunk→head→next→key) != chunk )
2:    return chunk→mergeBuddy;
3: else return PREV;
}
```

keys. Otherwise, we check the smallest key in the next chunk. If the input key is smaller, then the current chunk is returned. Otherwise, we continue to check the next chunk. Implicitly, this means that a chunk is associated with the range of keys that start in its smallest key (of its first entry) and end in the smallest key of the following chunk. The last chunk in associated with a range whose highest value is ∞.

One last method to be specified that handles the list of chunks is the *listFind-Previous* method, predented at Algorithm 18 that finds the previous chunk to the input one, for use of the merge procedure. If it doesn't find its input chunk in the list, then this input chunk must have already been frozen and disconnected from the list of chunks. In this case, a previous chunk is not well defined, and this routine just returns the *mergBuddy* pointer of the input chunk.

2.8 Supporting functionalities

Several trivial low-level methods were not specified. For completeness, we provide a short specification for them in Figure 2.6. These functions are all local, and involve no concurrency (or contention) issues.

2.9 Linearization Points

When designing a concurrent data structure, it is important to spell out the linearization points for the different operations. This is done in this section. In particular, we specify the linearization points of the insertion, deletion and search operations.

The linearization point of insertion. We partition the insertion linearization point determination into two cases. If the insertion operation is successful, i.e., no other entries with same key are found, then the linearization point is the successful execution of the CAS instruction at Line 11 of *InsertEntry*, where we actually modify the previous entry to point at the newly inserted entry. This modification creates the linearization point, whether it is executed by the thread executing the insert or by a different thread that is helping it (during a freeze). However, when this modification is executed on a new local chunk that a thread is preparing to replace a frozen chunk, then the modification of the local chunk is not considered a linearization point. Instead, the linearization point of the insert becomes the point in which this chunk is successfully linked to the the frozen chunk (Lines 19 or 21 of the *HelpSwap*() method).

If the insertion is not successful, i.e., an entry with the same key is found in the list, then the linearization point is the linearization point of the successful *Find* that is invoked at Line 4 of *InsertEntry*. The linearization point of the *Find* method is specified below. And again, a special case is the one in which the freeze recovery is the one to find the key and decide on a failure. In this case, the finding of the key happens on a frozen chunk and it has a special linearization point: it is the time in which the chunk C is stabilized. This point is formally defined (as $SP(C)$) below.

The linearization point of deletion. Again, we start by considering the successful case, in which the entry is found in the list, then the linearization point is the successful mark of the entry with the deletion bit. This happens at the successful execution of the CAS instruction at Line 12 of *DeleteInChunk*. Note that sometimes we need to wait until a chunk is frozen and only then can we attempt a deletion on a new chunk; however, the actual delete only happens when we manage to set the delete bit on an unfrozen entry containing the key. When this modification of the delete bit is executed on a new local chunk that a thread is preparing to replace a frozen chunk, then the modification of the local chunk is not considered a linearization point. Instead, the linearization point of the delete becomes the point in which this chunk is successfully linked to the the frozen

chunk (Lines 19 or 21 of the *HelpSwap*() method).

When the deletion operation is not successful, i.e., an entry with the input key is not found, then the linearization point is the linearization point of the unsuccessful *Find* that is called on Line 7 of *DeleteInChunk*. And again, a special case is the one in which the freeze recovery is the one to not find the key and decide on a failure. In this case, the unsuccessful search of the key is executed on a frozen chunk and it has a special linearization point: the time in which the chunk C is stabilized. This point is formally defined (as $SP(C)$) below.

The linearization point of search is the linearization point of the *Find* method invoked at Line 1 of the *SearchInChunk* method.

The linearization point of the *Find* method is the most delicate one. The *Find* method may traverse a chunk while it is being frozen. At the same time, the freeze may terminate concurrently, and *inserts* and *deletes* may occur on a new chunk that is not accessed by the find. Therefore, the find may fail to find a key that is inserted before it terminates. We, therefore, set the linearization point of *Find* to be the minimum between its standard linearization point and the time in which a stabilizing operation terminates on the chunk. Thus, the linearization point of a *Find* may happen earlier than the actual time when the find locates (or fails to locate) the input key. The point at which the chunk stabilizes satisfies that a new chunk does not exist yet, but on the other hand, no changes can occur on the accessible entries in the chunk's list from this point and on. This discussion is formalized below.

Consider a *Find* operation on a chunk C, we define the *stabilization point* of the find operation on C, denoted $SP(C)$ to be ∞ if the *freezeState* of C is NO_FREEZE at the time that the *Find* method returns. Otherwise, $SP(C)$ is defined to be the time in which the first stabilization of C terminates. Namely, among all threads executing the *StabilizeChunk* method on chunk C, $SP(C)$ is determined to be the minimum time in which one of them started executing Line 7 (i.e., the return from the stabilization method). Now that we have defined $SP(C)$, we consider the normal operation of *Find*, set linearization points to it, and then select the minimum between them and $SP(C)$.

Again, we separate for successful and unsuccessful cases. When the *Find* is successful, i.e., it returns a non-NULL *cur* pointer, the linearization point of *Find* happens when the *cur* pointer successful passes the validation check in Line 15 in the *Find* method. (Note that the validation is successful when the condition in Line 15 is evaluated to FALSE.) And as explained earlier, if $SP(C)$ happens earlier, then $SP(C)$ is the linearization point.

The unsuccessful case is more involved. Consider an execution of *Find* with input key k. There are two failure possibilities.

1. The first possibility is that an entry with k existed in the list but was marked as deleted. In this case the execution of *Find* disconnects it and the linearization point is the successful removal of the entry from the list, i.e., the successful CAS in Line 10 of *Find*.

2. The second possibility is that the entry with k did not exist in the list when *Find* searched for it (even not with a deletion mark). In this case, we set the linearization point of the failing *Find* to be when the pointer to the entry with the smallest key higher than k was loaded into the local variable *cur* in Lines 2, 12 or 19 of the *Find* method.

Again, the above two linearization points are set only if they happen before $SP(C)$. Otherwise, $SP(C)$ is the linearization point.

2.10 The intuition behind the design considerations

In this section we explain the main idea behind the algorithm, which form the intuition for a correctness proof of this chapter. Various parts of the algorithm are not new. The use of hazard pointer is similar to previous work, and the synchronization operations are used in a standard manner. The main deviation from previous work is the use of the freeze process to avoid many of the concurrency problems that naturally arise without it. The main problem is that when many concurrent operations are run on a chunk, it is not easy to determine how many entries reside on it, and whether it requires a split or a merge or none. An attempt to decide on a split and then reverse the decision may run into serious synchronization difficulties. We therefore choose the freeze method to stabilize it and make all threads work in harmony on it afterwards.

When a thread fails to find space for allocation, or when the size of the the chunk appears to be too low for a delete, a freeze is initiated. The freeze process is not atomic. While entries are marked as frozen, more inserts and deletes may happen and the need for a split or a merge may change during the freeze process. However, the freeze process is irreversible. The thread that started it will go on marking entries as frozen whenever it gains CPU access, and other threads that fail to insert or delete will join and help freezing the chunk. When all entries in the chunk are frozen, no more updates can occur on this chunk the continuation of the recovery for this chunk is completely determined from that point on. Thus, even if many threads attempt to build new chunks to replace the frozen one, they will all build exactly the same replacements and it does not matter which thread will do the final action of swapping the old chunk out of the list replacing it with the newly prepared chunks. It doesn't even matter if some of the work is done by one thread and some by others, they are all guaranteed to create the same structure. Only after finishing with the replacement of the old chunk, will the threads re-attempt the operation that failed. The only difference in the results of newly created chunks can be in result of promoting the insertion, deletion or enslaving during the freeze recovery.

To summarize, there are two main strategies. The first says that once an entry is marked frozen it will not be modified again. Furthermore, when all entries in a chunk are marked as frozen, all entries in the chunk will not be modified anymore, making the chunk data stable. This ensure that two threads cannot disagree on

the frozen state of an entry. When a thread sees the entry marked as frozen, it knows that no other thread will see it not-frozen in the future of the execution. The second strategy says that once a chunk is stable, any thread can decide on what needs to be done with this chunk and any thread can actually do it. All threads must reach the same decisions exactly and they must all attempt to put exactly the same values in exactly the same format of new chunks. Therefore, it does not matter which of these thread does what. The outcome is determined when the chunk gets stable, and all races become benign.

From these two design points many of the invariants follow. For example, a thread can get inactive for as long as it wishes. When it wakes up, the chunk it is accessing may be frozen, but hazard pointers ensure that the chunk has not been reclaimed, and any attempt of this thread to modify the chunk will reveal the fact that all entries are frozen. The thread will then try to take part in the freeze process and will quickly discover the chunks that replaced the frozen one and apply its modifications to them.

2.11 Lock-Freedom

In this section we outline the proof of lock-freedom for the construction. We start with the operations inside a chunk and then discuss the high-level list operations. The proof is based on the following invariants, which can be verified to be correct by looking at the pseudo-code.

1. Once an entry becomes frozen it never resurrects to a non-frozen state again
2. An entry cannot be marked as deleted when it is marked as frozen

In order to show the lock-freedom of the entire implementation, let us start by claiming that the *Find*() method (Algorithm 11) is lock-free, because this method is used by all other operations.

Lemma 2.1. *The Find() method (presented in Algorithm 11) is lock-free.*

Sketch of proof: The *Find*() method is very much the same as the one presented by Michael in [41], which was proven to be lock-free. Our *Find*() method is that of Michael, adapted to allow for freezing. Therefore, we skip the proof that our *Find*() method is lock-free in the presence of concurrent insertions or deletions, but focus on how it interacts with a simultaneous freezing of the chunk. The *Find*() method may be restarted in Lines 5 and 15 due to the failure of the hazard pointers' verification check. It also may be restarted due to the CAS failure in Line 10 (when helping to remove deleted entry). From the perspective of the concurrent freeze, those restarts happen only if the previous entry of the chunk's linked list got marked as frozen after a pointer to the current entry was copied to *cur*. In this case *Find*() restarts, but the task of marking entries as frozen has made progress by at least one entry. As we show later in Lemma 2.6, a successful

freeze of the entire chunk implies the progress of at least one insert or delete operation. This analysis holds for all CAS instructions that may fail due to progress in freeze marking, and in particular it holds for the three CAS instructions in the *Find*() method. ■

Let us now look at insert and delete operations.

Lemma 2.2. *The InsertToChunk() method (presented in Algorithm 1) is lock-free.*

Sketch of proof: Obviously, the entry allocation (*AllocateEntry*() method, Algorithm 2) is lock-free, because there are no loops or other backward brunches in the code. Inspecting the *InsertToChunk*() method's code we can see that, if the *AllocateEntry*() method fails, then freeze is invoked and, according to Lemma 2.6, being involved in any freeze activity results in the progress of some thread.

When the *InsertToChunk*() method succeeds in allocating an entry, it invokes the *InsertEntry*() method. The *InsertEntry*() method looks for a suitable location for the entry in the list, via the *Find*() method, which is lock-free (Lemma 2.1). Later the *InsertEntry*() method uses two CAS instructions to insert the entry. The first CAS, in Line 10 of *InsertEntry*(), may fail only due to yet undetected freeze activity and the *InsertEntry*() method's loop is restarted. But on the next iteration of the *InsertEntry*() method's loop, the shared data is reread and the set freeze bit is detected; therefore this bit will not foil this CAS. If the same CAS fails again, it must be because the insertion of the new entry was executed by another thread and progress was obtained. The second CAS, in Line 11 of *InsertEntry*(), can also fail only because of a freeze or a successful insertion of a new entry by another thread.

After invoking the *InsertEntry*() method, the *InsertToChunk*() method's code is straight forward, with no loops or other backwards branches. Therefore it is lock-free. ■

Lemma 2.3. *The DeleteInChunk() method (presented in Algorithm 12) is lock-free.*

Sketch of proof: The deletion task is a variation of the Harris's deletion algorithm [22], known to be lock-free. The *DeleteInChunk*() method differs from the conventional lock-free delete task of Harris in the decrementing of the counter, which counts the number of the entries on the chunk. If the decrementing of the counter fails, then a freeze is invoked, which by Lemma 2.6 results in some thread's progress.

Now let us turn to a deletion task that succeeds in decrementing the counter. Another backwards branch of the *DeleteInChunk*() method can happen when setting the deletion bit fails, because a freeze bit has been concurrently set. This happens in Line 12 of the *DeleteInChunk*() method. Similarly to the analysis above, before branching back, a freeze activity is performed, which by Lemma 2.6 must result in some thread's progress.

Other than those potential backward branches, the deletion operation presented in the *DeleteInChunk*() method (Algorithm 12) is the same as conventional one and thus it is lock-free. ■

We next show that marking entries as frozen and stabilization tasks are lock-free.

Lemma 2.4. *The MarkChunkFrozen() and StabilizeChunk() methods (presented in Algorithm 5 and Algorithm 6 respectively) are lock-free.*

Sketch of proof: Two CAS instructions can cause a backwards branch in the task that marks entries as frozen. Let us investigate when the CAS in Line 4 of the *MarkChunkFrozen*() method can fail. This CAS tries to change the word which holds the next pointer p of an entry e. The CAS fails when p's value changes concurrently between Lines 2 and 4, as may occur in three cases: (1) when a new entry is inserted just after e (which implies progress with an insertion operation); (2) when entry e was marked as deleted (which implies progress with a deletion operation), or (3) when another task marked p as frozen (which implies progress with the marking task). There is progress in all three cases. Similarly, the second CAS, in Line 9 of *MarkChunkFrozen*() method, changes the *keyData* word of an entry e. This second CAS can fail if, between Lines 7 and 9, e was allocated (which implies progress for an insertion), reclaimed (which cannot happen twice unless insertion makes progress), or marked as frozen by another task (which implies progress for this task as well). Therefore, the task that marks entries as frozen is lock-free, because every repetition of the code execution involves some progress in the system. The lock-freedom of the stabilization task follows from the lock-freedom of the *Find*() method and the *InsertEntry*() method discussed earlier Lemmas 2.1 and 2.2. ■

For the next lemmas it is worth noting that there are exactly five tasks that can be executed inside a chunk: (1) insert, (2) delete, (3) search/find, (4) mark entries as frozen, and (5) stabilization. We next claim that freezing imply progress.

Lemma 2.5. *Assuming the freezing of a single chunk is lock-free, at least one insert or delete task progresses after at most k consecutive freezes of the adjacent chunks, where k is number of chunks in the list at the time when the first freeze starts.*

Sketch of proof: A freeze might be triggered by one of the following four cases:

1. A thread executes an insert of a key and finds no available space to insert a new entry on a chunk.
2. A thread executes a delete and is unable to decrease the counter or set the delete bit of the entry.
3. A thread needs to freeze a master and thus it helps to freeze the preceding chunk in the list in an attempt to find a slave for the master.

4. A thread reclaims an entry in chunk C and finds that C is in process of being frozen (Algorithm 14, Line 7).

We define a *freeze-purpose* of a freeze execution to be one of: insert, delete, enslaving, and reclamation, according to the trigger that started it. In the first and the second cases, presented above, a threads starts or helps the freeze in order to be able to insert or to delete a key. The freeze-purposes are insert in the first case and delete in the second case. In the third case, establishing master-slave relationship ensures the finish of a freeze of a single chunk. When the freeze is finished, an insert or a delete can occur on the new chunk, or the new chunk can serve as a slave to another master. The freeze-purpose in the third case is enslaving. Enslaving signifies progress, because (as explained in last paragraph of this proof) at most finite number of the enslaving actions cause an insert or a delete. In the last case, the freeze is triggered to finish reclamation of an entry in order to complete an unsuccessful insert or a successful delete. The freeze-purpose here is reclamation.

In Lines 22-34 of the *FreezeRecovery*() method, a thread that executes the freeze recovery tries to perform the freeze-purpose as part of this freeze. If the freeze-purpose is insert, then the thread adds its key and data to the local new chunk. Otherwise, if the freeze purpose is delete, the thread removes the key it is required to remove from the new chunk. In both the insert and the delete cases, the operation can make progress even when the operation returns $false$ upon completion, due to the existence or non-existence of the relevant key (e.g., an insertion may return $false$ because the key is already in the list).

If the freeze purpose is enslaving, then, as part of the freeze, the thread marks the new locally created chunk as the master's slave by marking its freeze state as EXTERNAL_FREEZE. Thus, if this new chunk is successfully inserted into the chunked list, it immediately results in an established master-slave bond, having a master and a slave chunks which point to each other. We explain later how enslaving implies progress. A thread that needs to freeze a chunk in order to reclaim an entry in the old chunk makes no progress during the freeze recovery. This case is discussed in the last paragraph of the proof. Each thread prepares its local chunk privately with the trigger purpose reflected in the local chunk that it creates. But eventually, only one thread t manages to connect its local chunk to the list replacing the frozen old chunk. We claim for the progress of thread t. This is obvious for insertions and deletions, and we need to discuss enslaving and reclamations. We start with the enslaving case.

If the freeze purpose is enslaving, the thread needs to freeze the previous chunk, in an attempt to create a slave for the master, but even if enslaving succeeds, enslaving by itself does not imply progress. The enslaving is required in order to establish the master-slave bond. After creating the bond, the thread needs to create the new chunks and connect the new chunks into the list, replacing the master and the slave. A thread t that manages to connect its new local chunk into the list has its freeze-purpose reflected in the new chunk. This can be a thread

that succeeded in enslaving or a different thread. If the freeze-purpose is insert or delete, the progress was made. But what if, in turn, t's freeze-purpose was enslaving? It is possible to imagine a chain of chunks where the last chunk is sparse and frozen, and is helping the previous chunk to freeze, in order to get it enslaved etc. Then, the previous chunk is sparse and frozen, and in turn helps the chunk that precedes it to freeze in order to get it enslaved. All chunks in such a chain must be sparse. In theory, this chain of events can continue until the beginning of the list. In this case we can see the insert or the delete operation make progress only after k freezes, where k is number of chunks in the list at the time when first freeze started. The k is a finite number, that can only be decreased. As increasing of the number of the chunks in such a chain of sparse chunks can only happen when sparse chunk becomes full, which implies progress with an insert operation.

The thread t that needs to proceed with freezing a chunk in order to reclaim an entry on the freezing chunk only helps the freeze in order to verify whether the entry to be reclaimed has finally found its way to the new chunk, due to deletion of another entry concurrently holding the same key. Thread t needs to help in freezing one single chunk, t is not required to freeze the next chunks if such a need occurs in freeze. Finishing the freeze for reclamation is directly followed by finishing of an entry reclamation, which is the last step in either a successful deletion or an unsuccessful insert. Thus, thread t makes progress in any case. ■

Lemma 2.6. *The entire freezing process is lock-free.*

Sketch of proof: We have already seen that the first two stages of the freeze, i.e., marking all entries as frozen and the process of stabilization of the chunk are lock-free (Lemma 2.4). The last freezing stage is the freeze recovery. When looking at the code of the *Freeze*() and the *FreezeRecovery*() methods we see that these methods have no backward branches and contain no loops so they are lock-free. Finally, in order to cover all freeze activities, it remains to show that the *FindMergeSlave*() method is lock-free. The only possible repetition in this method is the CAS in Line 6 of the *FindMergeSlave*() method, where there is an attempt to enslave the previous chunk. This CAS can only fail if the previous chunk is frozen. If the CAS of Line 6 fails, we first finish the freeze of the previous chunk. Therefore, freezing of a single specific chunk makes no progress only if a freeze of a previous chunk is required to be enslaved for a merge. Endless repeating the freeze of the chunks is impossible according to Lemma 2.5. From Lemma 2.5 the maximal number of consecutive freezes that may execute without progress is k. Where k is the number of chunks in the list when first one of the consecutive freezes starts. Thus, once a freeze is started, a progress (an insert or a delete) must occur in the system within k consecutive freezes. The freeze method returns to its caller only after the freeze is accomplished, including the enslaving chain of multiple chunks, if needed. Therefore, the end of a freeze implies progress for some insert or delete operation. ■

Lemma 2.7. *All operations on the entire list of chunks are lock-free*

Sketch of proof: The upper-level list of chunks operations include only finding the chunk and invoking the chunk's operation. Once the relevant chunk is found there are no restarts from the beginning of the upper-level list of chunks. Since the chunk's operations where proven to be lock-free in the previous lemmas, it remains to prove the lock-freedom for the *HelpSwap*(), *ListUpdate*() and *FindChunk*() methods. The *HelpSwap*() method is straight forward with no backward branches, and thus it is lock-free. In the *ListUpdate*() method, we have two backward branches, one in Line 8 and another one in Line 9. Denote the frozen chunk that is replaced by C.

In Line 8, we try to set the swapped bit on the *nextChunk* pointer of C. Failure occurs in two cases. The first case is when another new chunk was inserted just after C. Insertion of a chunk after C happens only upon a freeze having been made on the chunk following C. The second case is when some other thread has already set the swapped bit of C. In the first case, the freezing being completed on the next chunk implies progress (Lemma 2.6). In the second case we advance the list update, as what we attempted to execute was done by some other thread. In Line 9 of the *ListUpdate* method we try to replace C by swapping the next pointer of the chunk preceding C. This attempt can fail either if (1) some new chunk was inserted just after C, or if (2) some new chunk was inserted just before C. In both cases, similarly to the previous analysis, a new chunk is inserted into the upper-level chunk list only after some freeze completes. As shown in Lemma 2.6 every completion of a freeze implies progress. ■

Function's Signature	**Explanations**
word combine (Xbits x, Ybits y);	Concatenates two strings of bits into one machine word, when x comes goes to the most-significant bits, and y to the least-significant bits.
bool isFrozen (entry* p);	Checks if the frozen bit (second LSB) is set in a given pointer p and returns TRUE or FALSE accordingly.
entry* markFrozen (entry* p);	Returns the value of a pointer p with the frozen bit set to one; it doesn't matter if in initial p this bit was set or not.
entry* clearFrozen (entry* p);	Returns the value of a pointer p with the frozen bit reset to zero; it doesn't matter if in initial p this bit was set or not.
bool isDeleted(entry* p);	Checks if deleted bit (LSB) is set in given pointer p.
entry* markDeleted (entry* p);	Returns the value of a pointer p with the deleted bit set to one; it doesn't matter if in initial p this bit was set or not.
entry* clearDeleted (entry* p);	Returns the value of a pointer p with the deleted bit reset to zero; it doesn't matter if in initial p this bit was set or not.
bool isSwapped (chunk* c);	Checks if swapped bit (LSB) is set in given pointer to a chunk c.
chunk* markSwapped (chunk* c);	Returns the value of a pointer c with the swapped bit set to one; it doesn't matter if in initial c this bit was set or not.
chunk* clearSwapped (chunk* c);	Returns the value of a pointer c with the swapped bit set to zero; it doesn't matter if in initial c this bit was set or not.
void copyToOneChunk (chunk* *old*, chunk* *new*);	Goes over all reachable entries in the *old* chunk linked list and copies them to the *new* chunk linked list. It is assumed no other thread is modifying the *new* chunk, and that the old chunk is frozen, so it cannot be modified as well.
key mergeToTwoChunks (chunk* *old*1, chunk* *old*2, chunk* *new*1, chunk* *new*2);	Goes over all reachable entries in the *old*1 and *old*2 chunks linked lists (which are sequential), finds the median key (which is returned) and copies the bellow-median-value keys to the *new*1 chunk linked list and the above-median-value keys to the *new*2 chunk linked list. In addition it sets the *new*1 chunk's pointer *nextChunk* to point to the *new*2 chunk. It is assumed that no other thread modifies the *new*1 and *new*2 chunks, and that the old chunks are frozen and thus cannot be modified as well.
void mergeToOneChunk (chunk* *old*1, chunk* *old*2, chunk* *new*);	Goes over all reachable entries on the *old*1 and *old*2 chunks linked lists (which are sequential and have enough entries to fill one chunk's linked list) and copies them to the *new* chunk linked list. It is assumed that no other thread modifies the *new* chunk and that the old chunks are frozen and thus don't change.
key splitIntoTwoChunks (chunk* *old*, chunk* *new*1, chunk* *new*2);	Goes over all reachable entries on the *old* chunk linked list, finds the median key (which is returned) and copies the bellow-median-value keys to the *new*1 chunk and the above-median-value keys to the *new*2 chunk. In addition it sets the *new*1 chunk's pointer *nextChunk* to point at the *new*2 chunk. It is assumed that no other thread is modifying the *new*1 and *new*2 chunks, and that the old chunk is frozen and cannot be modified.
int getEntrNum (chunk* c);	Goes over all reachable entries in Chunk c, counts them, and returns the number of entries. Chunk c is assumed to be frozen and thus cannot be modified.
void Allocate();	Allocates a new chunk as a zeroed memory chunk. The freeze state is set to NO_FREEZE.

Figure 2.6: The specification of (simple) supporting functions.

Chapter 3

A Lock-Free B$^+$tree

3.1 Introduction

The growing popularity of parallel computing is accompanied by an acute need for data structures that execute efficiently and provide guaranteed progress on parallel platforms. Lock-free data structures provide a progress guarantee: if the program threads are run sufficiently long, then at least one of them must make progress. This ensures that the program as a whole progresses and is never blocked. Although lock-free algorithms exist for various data structures, lock-free balanced trees have been considered difficult to construct and as far as we know a construction for a lock-free balanced tree is not known.

In recent decades, the B-tree has been the data structure of choice for maintaining searchable, ordered data on disk. Traditional B-trees are effective in large part because they minimize the number of disk blocks accessed during a search. When using a B-tree on the computer memory, a reasonable choice is to keep a node on a single cache line. However, some studies show that a block size that is a (small) factor of the processor's cache line can deliver better performance if cache pre-fetching is employed by the hardware [9, 46]. Further details about the B-Tree structure and the B$^+$tree variant appear in Subsection 3.2.1.

This chapter presents the first lock-free, linearizable, dynamic B$^+$tree implementation supporting searches, insertions, and deletions. It is dynamic in the sense that there is no (static) limit to the number of nodes that can be allocated and put in the tree. The construction employs only reads, writes, and single-word CAS instructions. Searches are not delayed by rebalancing operations. The construction employs the lock-free chunk mechanism proposed in the previous chapter. The chunk mechanism provides a lock-free linked list that resides on a consecutive chunk of memory and maintains a lower- and upper-bound on the number of elements. The chunks are split or joined with other chunks to maintain the bounds in the presence of insertions and deletions. This lock-free chunk mechanism fits naturally with a node of the B$^+$tree that is split and joined, keeping the number of elements within given bounds, and thus maintaining the balance of the tree.

Our construction follows some basic design decisions that reduce the complexity

of the algorithm. First, a node marked by the need to join or split is frozen, and no more operations are allowed on it. It is never resurrected, and one or two nodes are allocated to replace it. This eliminates much of the difficulty with threads waking up after a long idle period and encountering an old node that has been split or joined. In general, a node begins its lifespan as an infant, proceeds to become a normal node, and remains so until frozen for a split or a join, after which it is eventually reclaimed. This monotonic progress, reflected in the node's state, simplifies the design. The replacement of old nodes with new ones is challenging as data may be held in both the old and the new nodes simultaneously. To allow lock-freedom, we let the search operation dive into old nodes as well as new ones. But to ensure linearizability, we only allow new nodes to be modified after the replacement procedure is completed. Additionally, we take special care in the selection of a neighboring node to join with, to ensure that it cooperates correctly. Finally, we enforce the invariant that two join nodes always have the same parent. Our construction follows important lock-free techniques that have been previously used. In particular, we mark pointers to signify deletion following Harris [22], we assign nodes with states similarly to Ellen et al. [16]. We also extend these techniques in ways that are useful for future work, e.g., we gradually move a node to the *frozen* state, by marking its fields one by one as frozen.

This design of the lock-free B$^+$tree is meant to show the feasibility of a lock-free balanced tree. It is quite complex and we have not added (even straightforward) optimizations. We implemented this design (as is) in C and ran it against an implementation of a standard lock-based B$^+$tree [47]. The results show that the lock-based version wins when no contention exists or the contention is very low. However, as contention kicks in, the lock-free B$^+$tree behaves much better than the lock-based version. The lock-free tree is highly scalable and allows good progress even when many threads are executing concurrently. Similarly to the lock-free algorithm of the linked-list, a wait-free variant of the search method (denoted *contains*) can be defined here as well and in the same manner. Again, to keep things simple, we do not spell it out.

Note that a balanced tree has a better worst-case behavior compared to regular trees. Ignoring concurrency, each operation has a worst-case complexity of $O(\log n)$ in contrast to a worst-case complexity of $O(n)$ for an imbalanced tree. Furthermore, in the presence of concurrent threads, we prove that progress must be made at worst-case within $O(T \log n + T^2)$ computational steps, where T is number of the concurrent running threads and n is number of keys in the B$^+$tree. (This means bounded lock-freedom with bound $O(T \log n + T^2)$.) Such guarantee can only be achieved with balanced trees, as computing a similar bound on the worst-case time to make progress in a non-balanced tree would yield $O(Tn)$[1].

Previous work on lock-free trees include Fraser's construction [19] of a lock-free balanced tree that builds on a transactional memory system. Our work does not

[1] Actually, we do not know how to show a lock-free bound which is lower than $O(T^2 n)$ for non-balanced concurrent trees.

require any special underlying system support. Fraser also presents a construction of a lock-free tree that uses multiple-word CAS [19], but this construction offers no balancing and at worst may require a linear complexity for the tree operations. Recently, Ellen *et al.* [16] presented a lock-free tree using a single-word CAS, but their tree offers no balancing. Bender *et al.* [3] described a lock-free implementation of a cache-oblivious B-tree from LL/SC operations. Our construction uses single-word CAS operations. Moreover, a packed-memory cache-oblivious B-tree is not equivalent to the traditional B$^+$tree data structure. First, it only guarantees amortized time complexity (even with no contention), as the data is kept in an array that needs to be extended occasionally by copying the entire data structure. Second, it does not keep the shallow structure and is thus not suitable for use with file systems. Finally, a full version of [3] paper has not yet appeared and some details of lock-free implementation are not specified.

In Section 3.2 we set up some preliminaries and present the B$^+$tree representation in the memory together with the basic B$^+$tree algorithms. In Section 3.3 we describe the B$^+$tree node's states and recall the lock-free chunk functionality from the previous chapter. Balancing functions are presented in brief in Section 3.4, and the implementation and results are described in Section 3.5. In Section 3.6 we describe the linearization points. Supporting B$^+$tree methods are presented in Section 3.7. Balancing code and all relevant details are presented in Section 3.8. In Section 3.9 the redirection and help methods are presented. Boundary conditions by which the root needs to be exchange are presented in Section 3.10, and minor modifications required of the original chunk mechanism are presented in Section 3.11.

3.2 Preliminaries and Data Structure

This section presents the data structures used to implement the lock-free B$^+$tree, starting with a review of the lock-free chunk mechanism presented in the previous chapter. A *chunk* is a (consecutive) block of memory that contains *entries*. Each entry contains a key and a data field, and the entries are stored in the chunk as a key-ordered linked list. A chunk consumes a fixed amount of space and has two parameters, determining the minimum and maximum entries that may reside in it. The chunk supports set operations such as *search*, *insert* and *delete*. When an insert of a new entry increases the number of entries above the maximum, a *split* is executed and two chunks are created from the original chunk. Similarly, when a delete violates the minimum number of entries, the chunk mechanism *joins* this chunk and another chunk, obtained from the data structure using the chunks (in particular the B$^+$tree). Therefore, the B$^+$tree implements a method that the chunk can call to obtain a partner to join with. A different B$^+$tree method is called by the chunk mechanism when the split or join are completed to ask that the tree replaces the frozen nodes with new ones. The chunk also supports an additional *replace* operation that allows replacing the data of an entry with a new

value atomically without modifying the entry's location in the list. This operation is useful for switching a descendant without modifying the key associated with it. All operations are lock-free.

3.2.1 The B$^+$tree

A B$^+$tree [10] is a balanced tree used to maintain a set of *keys*, and a mapping from each key to its associated *data*. Each node of the tree holds entries, each entry has a key and an auxiliary data. In contrast to a B-tree, only the leaves in a B$^+$tree hold the keys and their associated data. The data of the keys in the internal nodes is used to allow navigating through the tree. Thus, data in an internal node of the tree contains pointers to descendants of the internal node. The B$^+$tree structure simplifies the tree insertions and deletions and is commonly used for concurrent access. In our variant of a B$^+$tree, key repetition is not allowed.

Each internal node consists of an ordered list of entries containing keys and their associated pointers. A tree search starts at the root and chooses a descendant according to the values of the keys, the convention being that the entry's key provides the upper bound on the set of keys in its subtree. Each node has a minimum and maximum number of possible entries in it. In our B$^+$tree the maximum is assumed to be even and is denoted d. The minimum is set to $d/2 - 3$. For $d \geq 10$ this ensures the balance of the tree, and specifically that the number of nodes to be read before reaching a leaf is bounded by a logarithm of the tree size. All insertions and deletions happen at leaves. When an insert violates the maximum allowed number of entries in the node, a split is performed on that node. When a delete violates the minimum allowed number of entries, the algorithm attempts to join two nodes, resulting in borrowing entries from a neighboring node or merging the two nodes, if moving entries is not possible.

Splitting and joining leaves may, in turn, imply an insert or a delete to the parent, and such an update may roll up until the root. We ignore the minimum number of entries on the root, in order not to enforce a minimal number of entries in the tree. Note that splits and joins always create nodes with a legitimate number of entries. In practice, the minimum value is sometimes set to be smaller than $d/2 - 3$ to avoid frequent splits and joins.

3.2.2 The structure of the proposed B$^+$tree

For simplicity, our construction assumes the key and the data fit into a single word. This is the assumption of the chunk mechanism and it makes the allocation of a new entry easier. In practice, this means a word of 64 bits, with a key of 32 bits and data of 32 bits.[2] An architecture that provides a double-word compare-and-swap would allow using a full word for each of the fields, removing the restrictions, and

[2]Since a data field cannot hold a full pointer, we assume a translation table, or some base pointer to which the 32-bit address is added to create the real memory address. In the first case, this limits the number of nodes to 2^{32} nodes, and in the second case, it limits the entire tree space to 4GB, which is not a harsh constraint.

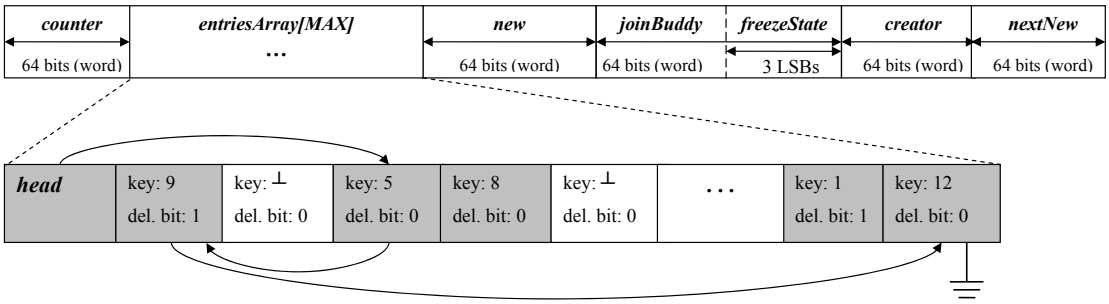


Figure 3.1: The structure of a chunk. The allocated grey entries present the ordered linked list.

simplifying the construction. The key values are taken from a finite set, bounded from above by a value that we denote ∞. The tree is represented by a pointer to the root node, initially set to an empty root-leaf node.

Our B$^+$tree node is built using the chunk structure of the previous chapter. The chunk's maximum and minimum number of entries are set to d and $d/2 - 3$ to satisfy the B$^+$tree node requirement (except for the zero minimum bound on the root). In addition to a chunk, the tree node contains two additional fields to support its management: a *height* field indicating the distance from the leaves and a *root* flag indicating whether the node is a root.

We briefly review the fields of a chunk (see Figure 3.1). A detailed discussion appears in the previous chapter. The main part of the chunk is an array that contains all the entries. The *counter* field counts the number of entries in a chunk. It is accurate during sequential execution and is always guaranteed to hold a lower bound on the real count, even in the presence of concurrent executions. The pointers *new*, *joinBuddy*, *nextNew* and *creator* point to nodes involved in the rebalancing, to be described below in Section 3.4. The split and join of a chunk requires a *freeze* of all operations on it, which imposes the *freeze state* of a chunk to be declared using *freezeState* field. The freezing mechanism will be explained later, in Sections 3.3 and 3.4.

3.2.3 Memory Management

To avoid some of the ABA problems, lock-free algorithms typically rely on garbage collection or use the hazard pointer mechanism of Michael [41]. To simplify the current presentation, we assume the existence of garbage collection for the nodes. This means that nodes are never reused unless they become unreachable from all threads. An extension of the same scheme to a use of hazard pointers is possible.[3]

[3] In the implementation we measured, we implemented hazard pointers inside the chunk and did not reclaim full nodes at all during the execution.

3.2.4 The Basic B$^+$tree Operations

The B$^+$tree interface methods: *SearchInBtree()*, *InsertToBtree()*, and *DeleteFromBtree()* are quite simple. The code of the basic B$^+$tree operations is presented in Algorithm 21 (relegated to Section 3.8). An insert, delete, or search operation first finds the leaf with the relevant key range, after which the appropriate chunk operation is run on the leaf's chunk. It either simply succeeds or a more complicated action of a split or a join begins. Some care is needed when the suitable leaf is a new one (an infant), whose insertion into the B$^+$tree is not yet complete. In that case, we must help finish the insertion of the new node before continuing to perform the operation on it. Further explanations on the freezing of a node, on the infant state, etc. appear in Section 3.3.

3.3 Splits and Joins with Freezing

Before it is split or joined, a node's chunk must be frozen. The complete details appear in the previous chapter. The freezing is executed by the chunk mechanism when its size limits are violated. This happens obliviously to the containing data structure, in this case, the B$^+$tree. Here we provide an overview on the chunk's freeze required to understand the B$^+$tree algorithm. To freeze a node, i.e., to freeze the chunk in it, all the chunk's entries are marked *frozen* (one by one) by setting a designated bit in each entry. After all the entries are marked frozen, no changes can occur on this node. A thread that discovers that a node needs to be frozen, or that a freeze has already begun, helps finish freezing the node. However, search operations do not need to help in freeze and can progress on the frozen nodes. Since changes may occur before all entries are marked frozen, the final state of the frozen node may not require a split or a join at the end of the freeze. Still a frozen node is never resurrected. After the freeze has been marked and the node can no longer be modified, a decision is made on whether it should be split, or joined with a neighboring node, or just copied into a single new node. If a join is required, then a neighboring node is found by the B$^+$tree. This communication between the chunk and the B$^+$tree is implemented using a predetermined method *FindJoinSlave()* that the tree supplies and the chunk mechanism uses. Then the neighboring chunk is frozen too. To recover from the node freeze, one or two nodes are allocated, and the live entries in the frozen node (or nodes) are copied into the new node (or nodes). Thereafter, a B$^+$tree method *CallForUpdate()* is called to let the tree replace the frozen nodes with the new ones. We focus in what follows on issues specific to the B$^+$tree, i.e., finding a neighbor, replacing the frozen nodes with the new ones in the B$^+$tree, and maybe rolling up more splits or joins.

Each tree node has a *freezeState* field, holding one of eight possible freeze states. Three bits are used to store the state. The freeze state is also a communication link between the B$^+$tree and the chunk mechanism, and so it can be read and updated both by the B$^+$tree and by the chunk. When a new node is created to replace a frozen node, and until it is properly inserted into the B$^+$tree, its freeze

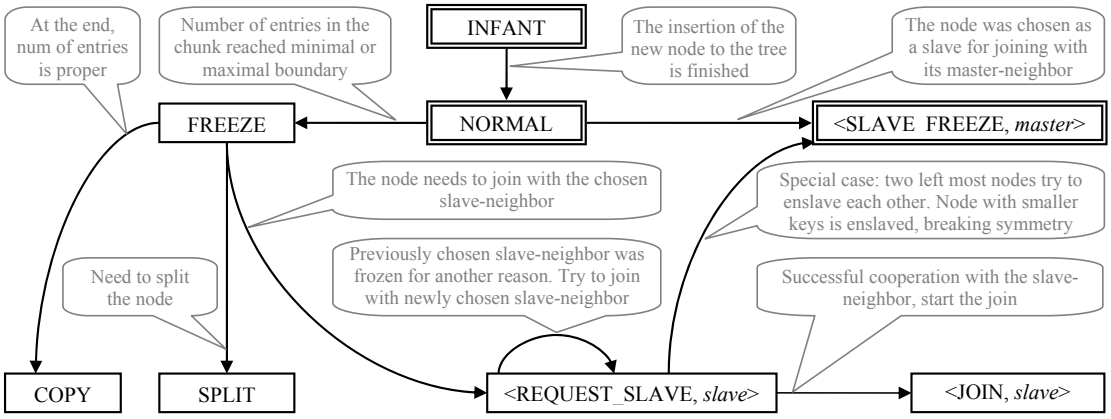


Figure 3.2: The state transitions of the freeze state of a node. The initial states are presented in the boxes with the double border.

state is marked as INFANT. No insertions or deletions are allowed on an infant node until the node's freeze state becomes NORMAL. Any thread that attempts an operation on such a node must first help move this node from the INFANT to the NORMAL state. A node that is properly inserted into the B$^+$tree and can be used with no restrictions has a NORMAL freeze state. When an insert or a delete operation violates the maximum or minimum number of entries, a freeze of that node is initiated and its freeze state becomes FREEZE. After the freezing process stabilizes and the node can no longer be modified, a decision is reached about which action should be taken with this node. This decision is then marked in its freeze state as explained below.

When neither split nor join is required (because concurrent modifications have resulted in a legitimate number of entries), the freeze state of the node becomes COPY, and the node is simply copied into a newly allocated node. By the end of the copy, the parent's pointer into the old node is replaced (using the chunk's replace operation) with the pointer to the new node, and the new node becomes NORMAL. When a split is required, the node's frozen state changes to SPLIT and all its live entries are copied into two new INFANT nodes. These nodes are then inserted into the tree in place of the frozen node, after which they can become NORMAL. A join is more complicated since a neighbor must be found and *enslaved* for the purpose of the join. Since only three bits are required to store the freeze state, we can use the freeze state to also store a pointer to a join buddy and modify the state and the pointer together atomically.[4] The join process starts by looking for a neighbor that can be enslaved for the join and then the freeze state of the join initiator is changed into REQUEST_SLAVE together with a pointer to a potential join buddy in the *joinBuddy* word. Thus, the freeze state is actually modified into a pair <REQUEST_SLAVE, *slave*>. At the enslaved node, its state is then modified from NORMAL into the pair <SLAVE_FREEZE, *master*>, where *master* is

[4] An 8-alignment of a node can be assumed in modern systems and the three redundant least-significant bits can hold the freeze state

a pointer to the node that initiated the join. (Upon failure, we try to resolve the contention and try again.) When the connection between the join initiator (the master) and the join slave is finalized, the freeze state of the master is modified into <JOIN, *slave*>, where *slave* points to the determined join buddy node. The node that is typically chosen for a join is the immediate left sibling of the current node, except for the leftmost node, which chooses its immediate right sibling for the join. A special boundary condition appears when the two leftmost children of a node try to enslave each other. In order to break the symmetry in this case, we take special care to identify this situation and then choose the leftmost sibling among the two to be the slave. Figure 3.2 presents the state transition diagram for the *freezeState* field.

3.4 Balancing the B$^+$tree

The basic methods for the tree operations have been discussed in Section 3.2.4. We now give a high-level description of how to balance the tree following a split or a join of a node, and discuss the interface between the chunk mechanism and the tree operations. As we said, upon a violation of the node size constraints, the chunk mechanism first freezes the node to prevent it from being modified and decides on the rebalancing action (split, join, or copy). In case of a join, the chunk mechanism invokes the B$^+$tree method *FindJoinSlave()*, which finds such a join buddy. Later, the chunk mechanism creates new node(s) and copies the relevant information into them. When this part is completed, the chunk calls B$^+$tree method *CallForUpdate()*. This method lets the B$^+$tree algorithm replace the frozen node (or nodes) with the newly created node (or nodes) in the tree. The *CallForUpdate()* method actually redirects the calls according to whether a split, a copy, or a join occurred. Let us examine each of these cases. The pseudo-code for *CallForUpdate()* is presented in Section 3.9.

Before diving into the details, note that in general, upon creation of a node due to a split, a join, or a copy, the new node's freeze state is initiated to INFANT, its root flag is initiated to FALSE, its height value is copied from the original node's height value, and its counter is initiated to the exact number of entries copied into it. Also, the *creator* pointer of a new node is initiated to point to the old node, initiated the split, join, or copy operation.

3.4.1 Node Split

After the chunk mechanism executes a split, the original node N is frozen, and N's *new* field points to the new node N_1 holding the lower half of the keys from the old node N. The field N_1.*nextNew* points to the second new node N_2 holding the higher half of the keys from the old node N. The two new nodes' freeze states are initiated to INFANT so that no updates can occur on these nodes until they are inserted into the tree. Given that the chunk split already completed, the *CallForUpdate()* method invokes the *InsertSplitNodes()* method, algorithm of

which we describe below. The code with all the details is provided in Section 3.8 (Algorithm 22).

Replacing split node N starts by searching for its parent P in the tree. If the parent cannot be found, then the input node is no longer in the tree. This happens if the new node N_1 was properly inserted by some other thread, and the node N was disconnected in the process. In this case, the splitting process continues with inserting N_2. Otherwise, and having found the parent, we modify it to point to the new node N_1. This is done by inserting a new entry to P. The new entry contains the maximal key from N_1 as key and the pointer to N_1 as data. If the insert fails, it means that someone else has inserted this entry to the chunk and it is fine to continue. Therefore, we do not check if the insert succeeded. Note the possibility that the parent's chunk insert will create a split in the parent, which will recursively cause a split and roll it up the tree.

After the first new node is in place, we replace the pointer in the parent node, that points to the frozen node N, with the pointer to the second new node N_2. Again, this can only fail if another thread has done this earlier. In order to replace the pointer on the correct parent, we search for the parent (in the tree) of the split node once again. The second parent search may yield a different parent if the original parent was concurrently split or joined. After making the parent point to the two new nodes N_1 and N_2, it remains to set their state to NORMAL and return. The splitting process is completed.

If the original node N was determined to be the root, then a new root R with two new children N_1 and N_2 is created. Next, the B$^+$tree's root pointer is swapped from pointing to N to point to R. The details of the root's split code are relegated to Section 3.10.1.

3.4.2 Nodes Join

Establishing master-slave relationship: We assume that the join is initiated by a sparse node N, denoted *master*. The chunk mechanism has frozen the node N and it has determined that this node has too few entries. To complete the join, the chunk lets the B$^+$tree find the *slave*. The B$^+$tree establishes a master-slave relationship and later the chunk mechanism joins the entries of both nodes. The B$^+$tree's *FindJoinSlave()* method is responsible for establishing master-slave relationship and returns the slave for the given master. Its code is presented in Algorithm 23 in Section 3.8.2. The master-slave relationship establishing is described below.

The search for the neighboring node starts by finding the master's parent node P together with the pointers to the master's and its neighbor's entries. The parent node search fails only if the node N has already been deleted from the tree, in which case a slave has already been determined and this step is completed. Otherwise, the parent and a potential slave node M were found. The left-side neighbor is returned for all nodes except the left-most node, for which a right-side neighbor is returned. In order to establish the relationship, we first change N's freeze state

from <FREEZE, NULL> to <REQUEST_SLAVE, M>. This CAS operation may fail if N's freeze state has been already promoted to JOIN, in which case N's final slave has already been set. This CAS operation may also fail if another slave was already chosen due to delay of this CAS command. In this case, we just use that slave.

After finding a potential slave, we attempt to set its freeze state to <SLAVE_FREEZE, N> and freeze it. For this purpose the *SetSlave()* method is invoked from the *FindJoinSlave()* method. The *SetSlave()* method's code and details are presented in Algorithm 24 in Section 3.8.2. The *SetSlave()* method's algorithm is described in the next paragraph. After succeeding in setting the slave's freeze state, we change the master's state from <REQUEST_SLAVE, M> to <JOIN, M>, to enable the actual join attempts.

In order to enslave the slave M, we first attempt to CAS M's freeze state from <NORMAL, NULL> to <SLAVE_FREEZE, N>. After the CAS of the freeze state in the slave is successful, slave is frozen and we may proceed with the join. But M's freeze state isn't necessarily NORMAL: if it is not, then M is either still an infant or it is already frozen for some other reason. In the first case, we help M to become NORMAL and retry to set its freeze state. In the second case, we help to complete M's freeze. After the freeze on M is complete, M is frozen forever and is not suitable to serve as a slave. Therefore, M's enslaving has failed and the process of finding another slave must be repeated. A special case occurs when the potential slave M has a master freeze state as well and is concurrently attempting to enslave N for a join. This case can only happen with the two leftmost nodes and if we do not give it special care, an infinite run may result, in which each of the two nodes repeatedly tries to enslave the other. In order to break the symmetry, we check explicitly for this case, and let the leftmost node among the two give up and become the slave, with a SLAVE_FREEZE state and a pointer to its master (which was originally meant to be enslaved for it). Finally, we finish M's enslaving by freezing M. After the freezing of the slave M is done, we continue with the join.

Merge: If the number of entries on the master and the slave is less than d, chunk mechanism creates a new single chunk to replace the master and the slave. We denote this operation *merge*. In this situation, the *InsertMergeNode()* method is called (via *CallForUpdate()*) by the chunk mechanism in order to insert the new node into the tree. At this point, a master-slave relationship has been established, both M and N have been frozen, and a new node N_1 has been created with the keys of both M and N merged. The code with all the details for the merge is presented in Algorithm 25 in Section 3.8.2.

The merge starts by checking which of the original nodes (master N or slave M) has higher keys. We denote this node by N_{high} and the other node by N_{low}. Next, we look for N_{high}'s parent. If the parent is not found, then N_{high} must have already been deleted and we need to handle the node with the lower keys, N_{low}. Otherwise, we replace the parent's pointer to N_{high}, with a pointer to the new

node N_1. Next, we handle the pointer to N_{low} at the parent by attempting to delete it. Finally, we turn the new node's freeze state to normal.

Special care is given to the root. We would like to avoid having a root with a single descendant, and that might happen when the two last descendants of a root are merged. In this case, we make the new merged node become the new root. (See the *MergeRoot()* method in Section 3.10.2.)

Borrow: If the keys of two join nodes cannot fit a single node, they are copied into two new nodes. This operation is called *borrow*. The code and all the details for the borrow process is presented in Algorithm 26 in Section 3.8.3. Recall that in the borrow case four nodes are involved: the master N, the slave M, the new node with the lower keys N_1 and the new node with the higher keys N_2. As in merge case, we start by finding the high and low keys' nodes, N_{high} and N_{low}, among the master and the slave.

We then take the following steps: (1) Insert a reference to N_1 to the parent node (with the maximal key on the N_1 as the key); (2) Change the parent entry pointing to N_{high} to point to N_2; (3) Delete the parent entry pointing to N_{low}.

3.4.3 Two Invariants

We mention two correctness invariants that may expose some of the correctness arguments behind the algorithm and help the reader understand the course of the algorithm.

Keys duplication. During the balancing operations described above, we sometimes make duplicates of keys appear in the tree, but at no time will a key appear to be absent. For example, after the first new node is inserted to the parent as part of the split, there are keys that reside simultaneously in two different nodes: all keys in the first new node are also still available in the old split node, which is still in the tree. Similarly, as part of the merge, when an old frozen node with higher keys is replaced with the new node, there are keys that appear twice: all keys in the old frozen node with lower keys now also appear in the new node. Recall that a search in our B$^+$tree is allowed to navigate through the tree and return the result, based on the data found on the frozen node.

This does not foil searches in the tree. Old searches may access keys in the old frozen node(s), but new searches can only access the new infant node(s). Furthermore, none of these nodes can be modified until the rebalance process terminates. The new node is an infant, which temporarily precludes modifications, and the old node is frozen, which precludes modifications permanently.

We should also note that the tree doesn't grow too big because of duplication. An invariant that we keep is that there can only be two copies of a key in the tree. Thus, even though we may increase the size of the tree during the balancing, the increase will be at most by a factor of two. The factor-two increase is theoretical. In practice, the increase in the tree size is negligible. More about correctness and progress guaranties can be found in Section 5.6.

Master-slave bond. We take special care to guarantee that the master and

the slave keep the same parent up to the end of their join. Initially, the master and the slave are children of the same node P. However, there is always the chance that the parent node P is split so that these two children nodes do not have a single common parent anymore. This can subsequently lead the algorithm to make the tree structure inconsistent. Therefore, we enforce an invariant that the master and slave nodes must remain on the same parent. Namely, we do not allow the parent entries that point to a master and to its slave be separated into different nodes due to a parent's split or borrow, until new nodes replace the frozen master and slave. To this end, we take special care when the values of a frozen parent are divided into two new parents, and make sure that two such entries reside on a single new parent. Ensuring this variant is executed both on the parent splitting algorithm as well as on the children joining algorithm. First, at the parent side we check whether the descendants form a master and a slave and if they do, they are not placed on different new nodes. But the descendants may later enter a master and slave relationship, after this check was executed. Therefore, on the descendants' side, after declaring the intent of a master to enslave its neighbor (setting the master's state to REQUEST_SLAVE), we check that the master's parent is not in a frozen state. If it is, we help the parent to recover before continuing the descendants' join. This ensures that the parent split (or borrow) does not occur obliviously and concurrently with its descendants' join (or borrow).

3.4.4 Extensions to the Chunk Mechanism

The chunk interface requires some minor modifications, over presented in previous chapter, to properly serve the B$^+$tree construction in this chapter. Probably the most crucial modification arises from the need to deal with an ABA problem that arises during insertions and deletions of entries to the chunk of an internal node in the tree. The concern is that an insert or a delete may succeed twice due to a helper thread that remains idle for a while. Consider, for example, a merge and a subsequent delete of an entry at the parent node. Suppose that one thread executes the delete, but a second thread attempts this delete later, after the same key (with a different descendant) has been entered to the parent again. Thus, a delete should only succeed when the entry still points to the frozen node. As for inserts, we need to avoid reentering a pointer to a child node that has actually been frozen and deleted while the updating thread was stalled. To solve such problems, we add versioning to the *nextEntry* word in the chunk's linked-list. This eliminates the ABA problem, as a delayed CAS will fail and make us recheck the node that we attempt to insert or delete and discover that it has already been frozen. The extensions to the chunk mechanism are described and discussed in Section 3.11.

3.5 Implementation and Results

We have implemented the lock-free B$^+$tree presented in this chapter as well as the lock-based B$^+$tree of [47] in the C programming language. The lock-free design in

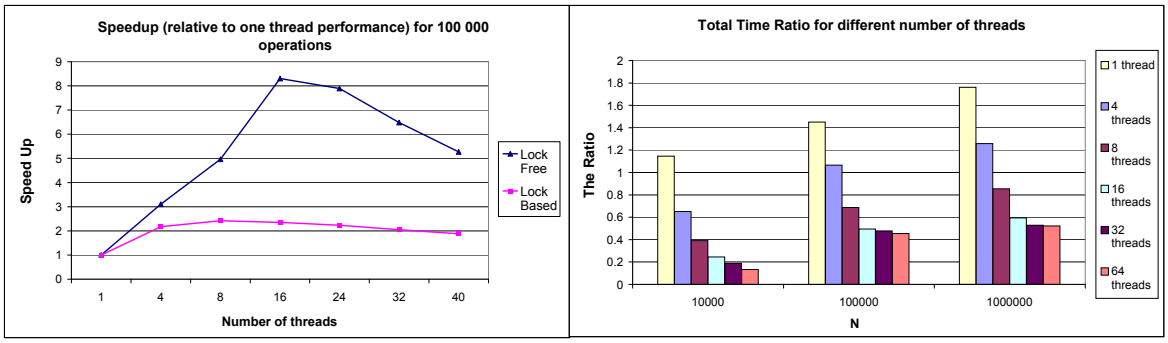


Figure 3.3: The empirical results.

this chapter can be optimized in many ways. However, we have implemented it as is with no further optimizations. The operations of the lock-based B+tree progress in a top-down direction. During the descent through the tree, *lock-coupling* [2] is employed, i.e., a child is locked before its parent is unlocked. Exclusive locks on the nodes are used for insert and delete operations, and shared locks are used for search operations. Deadlock-freedom is guaranteed by a proactive approach to rebalancing that splits full nodes or join sparse ones, while going down the path to the leaves.

We ran the experiments on the SUN FIRE machine with an UltraSPARC T1 8-core processor, each core running 4 hyper-threads, running Solaris 10. Overall, the eight cores, with quad hyper-threading simulates the concurrent execution of 32 threads. In both implementations the size of a B+tree node was set to the machine's virtual page size, i.e., 8KB. In each test we start with a tree with N random keys in the range $[0, 2^{18}]$ already inserted to it, and during the test, we apply N operations on it. If the test runs T threads, then each executes N/T operations. The parameter N was varied among 10^4, 10^5 and 10^6. The operations consisted of insertions, deletions and searches in parallel, out of which 20% were insertions, 20% were deletions, and the remaining 60% were searches. All the threads start simultaneously at the beginning and we measure the time it takes to complete all operations by all threads.

The left graph of Figure 3.3 depicts the ratio between the time it took to complete the runs on the lock-free implementation as compared to the lock-based implementation. A result higher than 1 means that the lock-free implementation is slower. Clearly, the lock-free implementation outperforms the lock-based implementation when contention is not low. Note that contention increases as the tree gets smaller and as the number of threads increases. Also, the results show that the average cost of an operation increases as the tree gets larger, because rebalancing may ascend to higher levels. Such costs are heavier for the lock-free tree, but this overhead is offset by lock-freedom efficiency when contention kicks in. The right graph of Figure 3.3 depicts the speedup, which clearly shows that the lock-free algorithm is more scalable.

The weaker performance of the lock-free tree for low contention can be ameliorated by simple optimizations. For example, during the split, each thread helping the split copies the entries from the old node to a newly created private node and only one of these new nodes eventually replaces the old node and joins the tree. While threads can cooperate to perform copying, we decided to avoid it in this version because it complicates the design.

3.6 Linearization Points

When designing a concurrent data structure, it is important to spell out the linearization points for the different operations. This is done in this section. The B$^+$tree methods all have a similar pattern of operation: they traverse the B$^+$tree to find the relevant leaf node, and then call the appropriate chunking methods on the leaf's chunk. Thus the linearization points of the B$^+$tree are typically based on the linearization points defined for the chunk in the previous chapter.

Search linearization point: The linearization point of the search operation is exactly the linearization point of the leaf's chunk search, as in the previous chapter. In particular, if the leaf is not frozen, then the linearization point follows that of the underlying linked-list in the leaf's chunk, and if the leaf is frozen then the linearization point is set to be the point in which the chunk became frozen. As the freezing mechanism is not instantaneous, we need to define a point in the freezing process more accurately for the linearization point. We follow the previous chapter and set the linearization point to be the point in the freeze process by which all the frozen bits have been set and also the internal list of the freezing node has been stabilized. Define this point as the *freezing point*. The freezing process of a chunk is explained in Section 3.11 and more thoroughly in the previous chapter. Formally, consider the linearization point of the search of the linked-list that is inside the chunk of the leaf (as defined by Harris [22]). If the chunk's linked-list search linearization point occurs before the freezing point, then that is also the linearization point of the overall tree search. If the chunk's linked-list linearization point happens after the freezing point, then we define the overall tree search linearization point to be the later point between the freezing point and the point in which the search started. The latter maximum makes sure that the linearization point happens during the execution of the search.

Justifying this choice for non-frozen node is straightforward. As for frozen nodes, we note that the frozen node may be replaced with a new node during the search execution and various actions may be applied on the new node. But at the freezing point, we know that the values of the frozen node exist only in the frozen node and are properly represented by the view of the frozen node.

The delicate case is when the search starts after the freezing point and still gets to the frozen leaf and completes the search there. In this case, since the search ends up in this leaf, we know that a new node that replaces this leaf (following the end of the freeze) has not yet been modified while the search traversed the tree,

Algorithm 19: Finding the relevant leaf node, given a key.

```
Node* FindLeaf (key) {
1: node = btree→root;
2: while ( node→height != 0 ) {                          // Current node is not leaf
3:     Find(&(node→chunk),key); node=cur→data;    //entry's data field is a pointer to the child
4: }
5: return node;                                          // current node is leaf
}
```

because the rebalancing operation has not yet terminated at that point. Therefore the new node has definitely not been modified when the search started, and the frozen values represent correctly the state of the tree at that point in time.

Insert and delete linearization points: Unlike the analysis of the search operation, frozen nodes are not hazardous for the insert's and delete's initial tree traversing. If an insert or delete arrive at a frozen leaf, than the *InsertToChunk()* or the *DeleteInChunk()* methods will redirect the operation (after helping the frozen node) to a non-frozen leaf node. Intuitively, the insert operation is assumed to be finished when a leaf including the new key is reachable from the root via data pointers. Similarly, the delete operation is assumed to be finished when a leaf excluding an old key is reachable from the root via data pointers. In a worst-case, this may require more than just handling a freeze.

There are three cases possible here. First, if the insert or delete operation doesn't cause a rebalancing activity (split, merge, borrow, or copy), than the linearization point is simply determined to be the leaf's chunk linearization point. Second, if a rebalancing (by freezing) occurs and if the thread performing the insert or delete operation has its operation executed in the node that replaces the frozen node, then the linearization point of the operation becomes the linearization point of the insert operation of the new node to the parent of the frozen node (replacing the frozen node with the new one). Note that this definition may be recursive if the parent requires rebalancing for the insertion. The third case is when the result of this operation is not reflected in the node that replaces the frozen one. In this case, we again define the linearization point recursively, setting it to be the linearization point of the re-attempted operation on the new node that replaced the frozen one.

3.7 B$^+$tree supporting methods

Before detailing the full B+ tree code in the following sections, we first describe in detail the supporting methods, *FindLeaf()* and *FindParent()*, which are used by other B+ tree methods. *FindLeaf()* finds the relevant leaf in which a given key may reside. This method is invoked by main B$^+$tree's interfaces: *SearchInBtree()*, *InsertToBtree()* and *DeleteFromBtree()*.

The *FindLeaf()* method is specified in Algorithm 19. Note that this search never fails, because each key may only belong to one leaf of the tree, depending

Algorithm 20: Find a parent of a node, given a key located on it and a pointer to the child

```
Node* FindParent (key, Node* childNode, Entry** prntEnt, Entry** slaveEnt) {
 1: node = btree→root;
 2: while (node→height != 0) {                                  // Current node is not leaf
 3:    Find(&(node→chunk), key);
 4:    if ( childNode == cur→data ) {      // Did we find exactly the entry pointing to the child?
 5:      *prntEnt = cur;
 6:      if( slaveEnt!=NULL ) { // Look for the child's neighbor, is the current entry the leftmost?
 7:                  if(prev==&(node→chunk→head) ) *slaveEnt=next;
 8:                  else *slaveEnt=EntPtr(prev);
 9:      } // end of if child neighbor was needed
10:      if (node→freezeState == INFANT) helpInfant(node);          // Help infant parent node
11:      return node;
12:    } // end of if child was found
13:    node = cur→data;
14: } // end of while current node is not leaf
15: return NULL;                                          // Current node is leaf, no parent found
 }
```

on the keys and pointers in the internal nodes. An appropriate leaf can always be returned, even if the key does not exist in it. The procedure starts from the root and ends when a leaf is found. At each step, it uses the *Find()* operation of the chunk mechanism to locate the appropriate pointer for the next descent. The *Find()* method of the chunk mechanism sets a global pointer **cur* to the entry with the minimal key value that is larger than or equal to the input key. This is exactly the entry whose pointer should be followed when descending the tree.

The second supporting method is *FindParent()*. When a split or a join occurs, we may need to find the parent of the current node in order to modify its pointers. Furthermore, we may need to find an adjacent node as a partner for a join, when a node becomes too sparse. The *FindParent()* method is presented in Algorithm 20. It is given a pointer to the child node, but also a key that exists on the child node, which allows navigation towards the child on the tree. It either finds the parent node, or returns NULL if the child can no longer be found in the tree (i.e., it was removed from the tree before this search was completed). A leaf node cannot be a parent; therefore we return NULL if we reach a leaf. Otherwise, we stop when we find an entry in a node whose descendant is the input node. At that point we know we found the parent node. The discovered entry in the parent node is returned, using the parameter *prntEnt*.

FindParent() may also provide a neighbor to be enslaved for a join. *FindParent()* looks for this neighbor when the parameter *slaveEnt* is not NULL. It always returns a pointer to the left neighbor (of the child node), unless the child is the leftmost child of its parent and then it returns the right-side neighbor. Technically, due to the uses of this method, it doesn't simply return the pointer to the neighbor. Instead, it returns a pointer to the parent entry that points to the neighbor, using the parameter *slaveEnt*. To find this neighbor, we further exploit

Algorithm 21: Search, Insert, and Delete – High Level Methods.

```
(a) BOOL SearchInBtree (key, *data) {
1: Node* node = FindLeaf(key);
2: return SearchInChunk(&(node→chunk), key, data);
}
(b) BOOL InsertToBtree (key, data) {
3: Node* node = FindLeaf(key);
4: if (node→freezeState == INFANT) helpInfant(node);          // Help infant node
5: return InsertToChunk(&(node→chunk), key, data);
}
(c) BOOL DeleteFromBtree (key, data) {
6: Node* node = FindLeaf(key);
7: if (node→freezeState == INFANT) helpInfant(node);          // Help infant node
8: return DeleteInChunk(&(node→chunk), key);
}
```

the method *Find()* of the chunk mechanism. The *Find()* method sets a global pointer **cur* to the entry with the minimal key value that is larger than or equal to the input key. But the *Find()* method also sets two more pointers: ***prev* and **next*. The global (indirect) pointer ***prev* points to the entry that precedes the entry pointed to by **cur* [5]. The entry that follows the one pointed to by **cur* is returned in a global pointer **next* (if such an entry exists). When *Find()* is used properly, the left neighbor of the child node will be point to from the global ***prev* pointer initiated by *Find()*, unless the child node is the leftmost child, in which case *prev* will point to the head entry. If it does point to the header, then we just return the pointer *next*.

If we find a parent, we also check whether its freeze state is INFANT; if it is, we help the infant parent to become a normal node before returning it. In most cases, the *FindParent()* is used to find a parent and then to apply insert, delete or replace operations on its chunk. Those operations are not allowed to be done on an infant parent.

3.8 Code and Detailed Explanations for Split, Merge and Borrow

The code for the B+tree interfaces is presented in Algorithm 21 and was described in Section 3.2.4. Here we describe the code and all the details for the balancing B+tree operations.

3.8.1 Node Splits

After the chunk mechanism executes a split, the original node N is frozen, and N's *new* field points to the new node N_1, which holds the lower half of the keys from the old node N. The field N_1.*nextNew* points to the second new node N_2,

[5] Eventually, ***prev* pointer points to the inner field *next* of the entry that precedes the entry pointed to by **cur*. We use *EntPtr()* to convert it to a normal pointer to the previous entry.

Algorithm 22: The split of a non-root node. The input parameter - sepKey - is the highest key in the low-values new node.

```
void InsertSplitNodes (Node* node, sepKey) {
 1: Entry* nodeEnt;          // Pointer to the parent's entry pointing to the node about to be split
 2: Node* n1 = node→new;                  // Pointer to the new node that holds the lower keys
 3: Node* n2 = node→new→nextNew;          // Pointer to the new node that holds the higher keys
 4: maxKey = getMaxKey(node);                    // Get maximal key on the given frozen node
 5: if ((parent = FindParent(sepKey, node, &nodeEnt, NULL)) != NULL) {
 6:    InsertToChunk(parent→chunk,sepKey,n1);   //Fails only if someone else completes before us
 7: }
 8: if ((parent = FindParent(maxKey, node, &nodeEnt, NULL)) != NULL) {
 9:    ReplaceInChunk(parent→chunk,              //Fails only if someone else completes before us
10:          nodeEnt→key, combine(nodeEnt→key, node), combine(nodeEnt→key, n2));
11: }
12: // Update the states of the new nodes from INFANT to NORMAL
13: CAS(&(n1→<freezeState, joinBuddy>), <INFANT, NULL>, <NORMAL, NULL>);
14: CAS(&(n2→<freezeState, joinBuddy>), <INFANT, NULL>, <NORMAL, NULL>);
15: return;
}
```

which holds the higher half of the keys from the old node N. The two new nodes' freeze states are initiated to INFANT so that no updates can occur on these nodes until they are inserted into the tree.

The split of the root is relegated to Section 3.10.1. The code for completing the split of a (non-root) node is presented in Algorithm 22. The *InsertSplitNodes()* method is invoked by the *CallForUpdate()* method. The *InsertSplitNodes()* method receives a pointer to the frozen node whose split needs to be completed, and the *sepKey* parameter. The *sepKey* parameter holds the middle key that "separates" the two new nodes. The *sepKey* key and all lower keys have been placed in the first new node (*n1*), and all keys higher than *sepKey* have been placed in the second new node (*n2*).

The split starts by searching for the parent (in the tree) of the split node, using *sepKey* for navigating in the tree. If the parent cannot be found, then the input node is no longer on the path for *sepKey* in the tree. This happens if the new node with the low level key was properly inserted by some other thread. Otherwise, and having found the parent P, we modify the parent to point to the new nodes. We want to insert a new link to the first new node (with the low level keys) into P. This is done by inserting a new entry to the parent node. If the insert fails, it means that someone else has inserted this entry to the chunk and it is fine to continue. Therefore, we do not check wether the insert succeeded.

Note that after the first new node is inserted to the parent, there are keys that reside simultaneously in two different nodes: all keys in the first new node are also still available in the old split node, which is still in the tree. This does not foil searches in the tree. Old searches may access keys in the old node, but new searches can only access the new node. Furthermore, none of these nodes can be modified until the split process terminates. The new node is infant, which

temporarily precludes modifications, and the split node is frozen, which precludes modifications permanently.

After the first new node is in place, we replace the pointer in the parent node, which points to the frozen node, with the pointer to the second new node (Line 9). Again, this can only fail if another thread has done this earlier. The *ReplaceInChunk()* method finds the entry with key and data as in its third argument and replaces it with key and data as in its last argument. (The *combine()* method syntactically combines the key and the data values into a single word.) In order to invoke *ReplaceInChunk()* on the correct parent, we search for the parent (in the tree) of the split node, using the maximal key of that node for navigating in the tree. The second parent search may yield a different parent if the original parent was concurrently split or joined. After making the parent point to the two new nodes, it remains to set their state to NORMAL and return. The split is done.

Note that the chunk insert in Line 6 might create a split in the parent, which will recursively call *CallForUpdate()*, and the splits may roll up the tree.

3.8.2 Merges

Establishing master-slave relationship: We assume that the merge is initiated by a node N, denoted *master*. The chunk mechanism has frozen the node N and it has determined that this node has too few entries. But the merge can not be completed by the chunk mechanism because the *slave* is required. Thus, we start by establishing a master-slave relationship, in order to share the entries with this node. To this end, the chunk mechanism calls the *FindJoinSlave()* method, presented in Algorithm 23. This method returns the slave for the given master.

The *FindJoinSlave()* method starts by calling the *FindParent()* method, which returns a pointer to the master's parent node together with the pointers to the master's and its potential slave's entries. The parent node search fails only if the node N has already been deleted from the tree, in which case a slave has already been determined and can be retrieved from the *joinBuddy* field of N. Otherwise, the parent and a potential slave node M were returned by *FindParent()*. In order to establish the relationship we first change N's freeze state from <FREEZE, NULL> to <REQUEST_SLAVE, M>. (Recall that the *joinBuddy* field and the freeze state field are located in one word.) If this is not our first try, the field may hold a previous slave pointer (*old_slave*) that we could not enslave. In this case, we change the value of N's freeze state from <REQUEST_SLAVE, *old_slave*> to <REQUEST_SLAVE, M>, where M is the new potential slave. The CAS operation in Line 10 may fail if N's freeze state has already been promoted to JOIN or became SLAVE_FREEZE due to swapping the master-slave roles as it will be explained below. In these cases N's final slave has already been set in the *joinBuddy* field of N. The CAS operation in Line 10 may also fail if another slave was already chosen due to delay of this CAS command. In this case, we just use that slave (Line 15).

For correctness, we enforce the invariant that the parent entries that point to

Algorithm 23: The code of finding a node partner for a join in the lock-free B$^+$tree.

```
Node* FindJoinSlave(Node* master) {
 1: Node* oldSlave = NULL;
 2: start: anyKey = master→chunk→head→next→key;          // Obtain an arbitrary master key
 3: if ( (parent = FindParent(anyKey, master, &masterEnt, &slaveEnt)) == NULL)
 4:    // If master is not in the B+tree; its slave was found and is written in the joinBuddy
 5:    return master→<*,joinBuddy>;
 6: slave=slaveEnt→data;                                 // Slave candidate found in the tree
 7: // Set master's freeze state to <REQUEST_SLAVE,slave>; oldSlave isn't NULL
 8: if (oldSlave==NULL) expState=<FREEZE,NULL>;          // when the code is repeated
 9: else expState=<REQUEST_SLAVE,oldSlave>;
10: if ( !CAS(&(master→<freezeState, joinBuddy>), expState, <REQUEST_SLAVE, slave>) ) {
11:    // Master's freeze state can only be REQUEST_SLAVE, JOIN or
12:    // SLAVE_FREEZE if the roles were swaped
13:    if ( master→<freezeState,*> == <JOIN,*> ) return master→<*, joinBuddy>;
14: }
15: slave = master→<*, joinBuddy>;          // Current slave is the one pointed by joinBuddy
16: // Check that parent is not in a frozen state and help frozen parent if needed
17: if ( (parent→<freezeState,*> != <NORMAL,*>) && (oldSlave == NULL) ) {
18:    Freeze(parent, 0, 0, master, NONE, &result); oldSlave = slave; goto start;
19: }
20: // Set slave's freeze state from <NORMAL, NULL> to <SLAVE_FREEZE, master>
21: if (!SetSlave(master,slave,anyKey,slave→chunk→head→next→key)) {
22:    oldSlave=slave; goto start;
23: }
24: // We succeed to get the slave update master
25: CAS(&(master→<freezeState, joinBuddy>), <REQUEST_SLAVE, slave>, <JOIN, slave>);
26: if (master→<freezeState,*> == <JOIN,*>) return slave; else return NULL;
 }
```

the master and the slave always reside on the same node. Namely, we do not allow entries that point to a master and to its slave be separated into different nodes due to a parent's split or borrow, until new nodes replace the frozen master and slave. To this end, we take special care when the values of frozen parent are divided into two new parents, and make sure that two such entries reside on a single new parent. A standard race occurs because after checking whether the descendants form a master and a slave and deciding that they are not, the descendants may later enter a master and slave relationship. A standard solution is to also check at the descendants' side. Namely, after declaring the intent of a master to enslave its neighbor (setting the state to REQUEST_SLAVE), we check that the master's parent is not in a frozen state. If it is, we help parent to recover before continuing the descendants' join (Lines 17, 18). This ensures that the parent split (or borrow) does not occur obliviously to and concurrently with its descendants' merge (or borrow).

After finding a potential slave, we attempt to set its freeze state to <SLAVE_FREEZE, N> and freeze it. This is done in the *SetSlave()* method presented in Algorithm 24 and explained in the next paragraph. If this action is not successful, we will try from scratch and look for another potential slave. After succeeding in setting the

Algorithm 24: Setting the slave's freeze state for a join in the lock-free B$^+$tree.

```
Bool SetSlave(Node* master, Node* slave, masterKey, slaveKey) {
 1: // Set slave's freeze state from <NORMAL, NULL> to <SLAVE_FREEZE, master>
 2: while
    (!CAS(&(slave→<freezeState,joinBuddy>),<NORMAL,NULL>,<SLAVE_FREEZE,master>)){
 3:   // Help slave, different helps for frozen slave and infant slave
 4:   if (slave→<freezeState,*>==<INFANT,*>) { helpInfant(slave); return FALSE; }
 5:   elseif (slave→<freezeState,*>==<SLAVE_FREEZE,master>) break;        //already done
 6:   else {                    // The slave is under some kind of freeze, help and look for new slave
 7:     // Special case check: two leftmost nodes try to enslave each other, break the symmetry
 8:     if ( slave→<freezeState, *> == <REQUEST_SLAVE, master> ) {
 9:       if (masterKey<slaveKey) {   //Current master is left sibling and should become a slave
10:         if (CAS(&(master→<freezeState,joinBuddy>),<REQUEST_SLAVE,slave>,
11:           <SLAVE_FREEZE,slave>) ) return TRUE; else return FALSE;
12:       else   // Current master node is right sibling and the other node should become a slave
13:         if ( CAS( &(slave→<freezeState,joinBuddy>), <REQUEST_SLAVE,master>,
14:           <SLAVE_FREEZE,master>) ) return TRUE; else return FALSE;
15:     } // end case of two leftmost nodes trying to enslave each other
16:     Freeze(slave, 0, 0, master, ENSLAVE, &result);          // Help in different freeze activity
17:     return FALSE;
18:   } // end of investigating the enslaving failure
19: } // end of while
20: MarkChunkFrozen(slave→chunk);              // Slave enslaved successfully. Freeze the slave
21: StabilizeChunk(slave→chunk);
22: return TRUE;
 }
```

slave's freeze state, we change the master's state from <REQUEST_SLAVE, M>, to <JOIN, M> to enable the actual join attempts.

The *SetSlave()* method attempts to CAS the freeze state of the slave M from <NORMAL, NULL> to <SLAVE_FREEZE, N>. If the CAS of the freeze state in the slave is successful, we may proceed with the join. But M's freeze state isn't necessarily NORMAL: if it is not, then M is either still an infant or it is already frozen for some other reason. In the first case, *SetSlave()* helps M to become NORMAL and retries to set M's freeze state. In the second case, it helps to complete M's freeze. After finishing the freeze on M, M is frozen forever and is not suitable to serve as a slave. Therefore, failure is returned by *SetSlave()* and another slave must be found. A special case occurs when the potential slave M has a master freeze-state as well and is concurrently attempting to enslave N for a join. This case can only happen with the two leftmost nodes and, if special care is not taken, an infinite run may result, in which each of the two nodes repeatedly tries to enslave the other. In order to break the symmetry, we check explicitly for this case, and let the leftmost node among the two give up and become the slave, with a SLAVE_FREEZE state and a pointer to its master (which was originally meant to be enslaved for it). The *FindJoinSlave()* checks for this case in its last line. If it is successful in turning the freeze state of the master into JOIN, then all is well. Otherwise, and given that *SetSlave()* completed successfully, then it must be the case that the master has become a slave. In this case, no slave is returned,

Algorithm 25: The merge of two old nodes to one new node

```
void InsertMergeNode (Node* master) {
 1: Node* new = master→new;                                   // Pointer to the new node.
 2: Node* slave = master→<*, joinBuddy>;
 3: maxMasterKey = getMaxKey(master);                         // Both master and slave are frozen
 4: maxSlaveKey = getMaxKey(slave);
 5: if ( maxSlaveKey < maxMasterKey ) {      // Find low and high keys among master and slave
 6:    highKey=maxMasterKey; highNode=master; lowKey=maxSlaveKey; lowNode=slave;
 7: } else {
 8:    highKey=maxSlaveKey; highNode=slave; lowKey=maxMasterKey; lowNode=master;
 9: }
10: if ((parent = FindParent(highKey, highNode, &highEnt, NULL)) != NULL) {
11:    highEntKey = highEnt→key;          // Change the highest key entry to point on new node
12:    ReplaceInChunk(parent→chunk,       // If fails, the parent chunk was updated by a helper
13:          highEntKey, combine(highEntKey,highNode), combine(highEntKey,new));
14: } // If high node cannot be found continue to the low
15: if ((parent = FindParent(lowKey, lowNode, &lowEnt, NULL)) != NULL) {
16:    if (parent→root) MergeRoot(parent, new, lowNode, lowEnt→key);
17:    else                                                  //lowNode is the expected data
18:       DeleteInChunk(&(parent→chunk),lowEnt→key,lowNode);
19: } // If also low node can no longer be found on the tree, then the merge was completed
20: // Try to update the new node state from INFANT to NORMAL
21: CAS(&(new→<freezeState, joinBuddy>), <INFANT, NULL>, <NORMAL, NULL>);
22: return;
}
```

and the returned NULL value tells the calling method (in the chunk mechanism) to treat the master as the slave.

Finally, we finish the *SetSlave()* by freezing the slave in Line 20, so that the join can continue. The method *MarkChunkFrozen()* marks all entries of a node frozen by setting a designated bit in each entry. After the entries are marked frozen, the *StabilizeChunk()* method ensures that no changes occur on this node. At this point the slave has been enslaved and frozen.

Merge: The *InsertMergeNode()* method is called (via *CallForUpdate()*) by the chunk mechanism in order to insert the new node into the tree. At this point, a master-slave relationship has been established, both M and N have been frozen, and a new node has been created with the keys of both M and N merged. (The case in which there are two new nodes is handled by a similar method called *InsertBorrowNodes()*, described in Section 3.8.3.) The code for completing of the merge is presented in Algorithm 25.

The *InsertMergeNode()* method's input parameter is a pointer to the master, this master's slave can be found in the *joinBuddy* field on the master. The *InsertMergeNode()* method starts by checking which of the original nodes (master and slave) has higher keys. Denote this node by *highNode*. Note that the master and the slave are frozen and thus immutable. Next, *FindParent()* is invoked on *highNode*. If the parent is not found, then *highNode* must have already been deleted and we need to handle the old node with the lower keys, *lowNode*. Otherwise,

Algorithm 26: The merge of two old nodes to two new nodes. The input parameter - sepKey - is the highest key in the low-values new node.

```
void InsertBorrowNodes (Node* master, sepKey) {
1: Node* n1 = node→new;                    // Pointer to the new node that holds the lower keys
2: Node* n2 = node→new→nextNew;            // Pointer to the new node that holds the higher keys
3: Node* slave = master→<*, joinBuddy>;
4: maxMasterKey = getMaxKey(master);                    // Both master and slave are frozen
5: maxSlaveKey = getMaxKey(slave);
6: if (maxSlaveKey<maxMasterKey) {  //Find low and high keys nodes among master and slave
7:    highKey = maxMasterKey; oldHigh = master; lowKey = maxSlaveKey; oldLow = slave;
8: } else { highKey=maxSlaveKey; oldHigh=slave; lowKey=maxMasterKey; oldLow=master; }
9: if ( lowKey < sepKey ) sepKeyNode = oldHigh;     // sepKey located on the higher old node
10: else sepKeyNode = oldLow;                       // sepKey located on the lower old node
11: if ((insertParent = FindParent(sepKey,sepKeyNode, &ent, NULL)) != NULL) {
12:    // Insert reference to the new node with the lower keys
13:    InsertToChunk(insertParent→chunk, sepKey,n1);
14: }
15: if ((highParent =FindParent(highKey,oldHigh,&highEnt, NULL))!=NULL) { //Find the parent
16:    ReplaceInChunk(highParent→chunk,highEnt→key,          //of the old node with the higher
17:          combine(highEnt→key,oldHigh),              //keys and change it to point to the new
18:          combine(highEnt→key,n2));                          //node with the higher keys
19: }
20: if ((lowParent =FindParent(lowKey, oldLow,&lowEnt, NULL)) != NULL) {          //Delete,
21:    DeleteInChunk(&(lowParent→chunk),lowEnt→key,                  //currently duplicated,
22:          oldLow);                                          // reference to the old low node
23: }
24: // Try to update the new children states to NORMAL from INFANT
25: CAS(&(n1→<freezeState, joinBuddy>), <INFANT, NULL>, <NORMAL, NULL>);
26: CAS(&(n2→<freezeState, joinBuddy>), <INFANT, NULL>, <NORMAL, NULL>);
27: return;
}
```

we adapt the parent's reference to *highNode*, to point to the new node. Next, we handle the pointer to *lowNode* at the parent by attempting to delete it. Finally, we turn the new node's freeze status to normal.

Special care is given to the root. We would like to avoid having a root with a single descendant, which can occur when the two descendants of a root are merged. In this case, we make the merged node become the new root. If merged node parent is found to be root, the *MergeRoot()* method is invoked from *InsertMergeNode()* instead of deleting the pointer to *lowNode* at the parent. This is so, because deleting an entry from the root may lead us to having a single root descendant. (See the *MergeRoot()* method in Section 3.10.2.)

Note that after the *highNode* has been replaced with the new node, the keys in the two nodes are repeated (the old lower frozen node and the new node), but any search will find the keys at the same state in both places, as when the merge began. The concurrent insertions and deletions will help the two nodes due to freezing or infancy (similar to the split case).

3.8.3 Borrow

In this Section we present the details of the *borrow* case, in which the keys in two join nodes are copied into two new nodes. The code for the *InsertBorrowNodes()* method responsible for the borrow is presented in Algorithm 26. As in merge case the *InsertBorrowNodes()* method is invoked from *CallForUpdate()* when a need for the borrow is encountered. The input parameters are a pointer to the master node and a separation key, which is the highest key on the new node that contains the lower keys. Recall that in the borrow case four nodes are involved: the master, the slave, the new node with the lower keys (in the code denoted *n1*) and the new node with the higher keys (in the code denoted *n2*). We start by finding the high and low keys' nodes, among master and slave (Lines 4-8), similar to the merge.

Next we are going to insert the new entry pointing to to the new node with the lower part of the keys. When we search for the parent for that, we carefully choose the child we supply for the search. The key of the new parent entry will be the separation key. This is the highest key on *n1* and therefore resides on the *n1* node. The data will be the pointer to *n1*. The separation key separates the two new nodes after the borrow. It may previously have been located either on the (old) high or the low keys' frozen node. It is important to supply to the *FindParent()* method the child on which the separation key was originally located. The check is done in Lines 9, 10.

We then follow these steps: (1) Insert a reference to *n1* to the parent node (2) Change the parent entry pointing to the high keys' frozen node to point to *n2* (3) Delete the parent entry pointing to the low keys' frozen node. During this execution, some entries will be duplicated. Namely, keys will appear twice in the tree, once in the old node and again in the new node (both times with same associated data). But none of the keys will appear to be absent, and therefore search correctness is maintained (similarly to the split and merge cases). Finally, we update the new nodes' freeze states.

3.9 Redirection of the call for an update

The B$^+$tree interfaces (discussed in Section 3.2.4) start by finding the relevant leaf of the tree and then executing the search, insert, or delete on the leaf's chunk . The insert or delete operations can cause the chunk to reach the minimal or maximal boundaries, after which the node will be marked frozen, stabilized, and a new node or nodes will be created according to the final (frozen) number of entries. Finally, the chunk mechanism invokes the *CallForUpdate()* function, which inserts the new nodes into the B$^+$tree instead of the old frozen node or nodes, to complete the rebalancing. The *CallforUpdate()* method is presented in Algorithm 27. It gets as input the pointer to the node that needs to be replaced, its freeze state and the separation key. Note that the freeze state was already determined by the chunk mechanism and is not going to be changed anymore. In addition, we assume the existence of a global pointer to the B$^+$tree named *btree*, which can be

Algorithm 27: Interface for updating the B+tree on frozen node recovery

```
void CallForUpdate (freezeState, Node* node, sepKey ) {
 1: Node* n1 = node→new;                    // Pointer to the new node that holds the lower keys.
 2: Node* n2 = node→new→nextNew;            // Pointer to the new node that holds the higher keys.
 3: switch ( freezeState ) {
 4:    case COPY:
 5:      if ( node→root ) { CAS(&(n1→root), 0, 1); CAS(&(btree→root), node, n1); }
 6:      else if ((parent=FindParent(node→chunk→head→next→key,node,&nodeEnt,NULL))
 7:        !=NULL)
 8:            ReplaceInChunk(parent→chunk,combine(nodeEnt→key,node),
 9:                combine(nodeEnt→key,n1));
10:      CAS(&(n1→<freezeState, joinBuddy>), <INFANT, NULL>, <NORMAL, NULL>);
11:      return;
12:   case SPLIT:
13:      if (node→root) SplitRoot(node,sepKey,n1,n2); else InsertSplitNodes(node,sepKey);
14:      return;
15:   case JOIN:
16:      if ( n2 == NULL) InsertMergeNode(node);   // If there is only one new node, then merge,
17:      else InsertBorrowNodes(node, sepKey);          // otherwise call the borrow operation.
18:      return;
19: }
 }
```

used by all the threads and provides the access to the shared B+tree structure. The global *btree* pointer is used here and also in other methods later.

The *CallforUpdate()* method actually redirects the calls according to the freeze state that it gets. Notice that for the copy case, we do not have a special method because we only need to replace the pointer to the old node with pointer to new one (using the chunk replace operation).

When a node is in an INFANT freeze state, its insertion or its sibling insertion to the B+tree is not yet complete. The insertion or deletion operations cannot be performed on an infant node, and it must be helped to become NORMAL before executing. Not only operations, but also balancing activities (split, join, copy), must be held until the insertion is completed. Thus, the *helpInfant()* method is called from various methods.

The code of *helpInfant()* is presented in Algorithm 28. First, we find the creator of the given infant node. The creator initiated the freeze due to which this infant node was inserted into the B+tree. The creator's freeze state reveals what the reason was for the freeze. To finish the freeze operation we often need the separation key, which is the highest key on the node with lower keys *in creation time*. But the highest key on the node with lower keys can be different now due to concurrent completion of the freeze activity. Thus we compute the key and then we check that the node with lower keys is still an infant. If it is not, we finish by changing the higher new node's freeze state to NORMAL (if the node exists). If the node with the lower keys is still an infant, than we have found the correct separation key.

If the infant has been inserted into the B+tree due to a COPY, its insertion is

Algorithm 28: Helping the infant node.

```
void helpInfant (Node* node) {
 1: creator = node→creator; creatorFrSt = creator→<freezeState, *>;
 2: Node* n1 = creator→new;                  // Ptr to the new node that holds lower keys from creator
 3: Node* n2 = creator→new→nextNew;    // Ptr to the new node, holding higher keys from creator
 4: sepKey = getMaxKey(n1);                                                    // n1 is never NULL
 5: if ( (n1→<freezeState,*>) != INFANT ) {                                  // Check low is still infant
 6:    if (n2) CAS(&(n2→<freezeState, joinBuddy>), <INFANT, NULL>, <NORMAL, NULL>);
 7:    return;
 8: }
 9: // If this is root split, only children's state correction is needed
10: if ( (creator→root) && (creatorFrSt == SPLIT) ) {
11:    CAS(&(n1→<freezeState,joinBuddy>), <INFANT, NULL>, <NORMAL, NULL>);
12:    CAS(&(n2→<freezeState,joinBuddy>), <INFANT, NULL>, <NORMAL, NULL>);
13:    return;
14: }
15: switch ( creatorFrSt ) {                                  // Can be only COPY, SPLIT or JOIN
16:    case COPY:
17:       CAS(&(node→<freezeState, joinBuddy>), <INFANT, NULL>, <NORMAL, NULL>); return;
18:    case SPLIT:
19:       InsertSplitNodes(creator, sepKey); return;
20:    case JOIN:
21:       if (n2 == NULL)          // If freeze state is JOIN and there is one new node, help in merge
22:          InsertMergeNode(creator);
23:       else InsertBorrowNodes(creator, sepKey);                          // help in borrowing
24:       return;
25: } // end of switch
}
```

almost done. It is completed by changing infant's freeze state to NORMAL. This operation can be unsuccessful only if it is concurrently completed by someone else.

In the cases of SPLIT and JOIN, we simply call the relevant method. Due to possible multiple invocations, the *InsertSplitNodes()*, *InsertMergeNode()* and *InsertBorrowNodes()* methods are idempotent. Making them such allows us to help the operation in its entirety, so that by its end, the insertion of the new nodes is complete.

3.10 Root boundary conditions

3.10.1 Splitting the root

The code for splitting the root in method *SplitRoot()* is provided in Algorithm 29. The *SplitRoot()* method is called by *CallForUpdate()*, Line 13, when there is a request to split a node whose root bit is set. The *SplitRoot()* method's input parameters are a pointer to the old root, pointers to the new nodes that hold the lower and higher keys of the old root, and the highest key in the new low-value node. The *SplitRoot()* method starts by allocating a new root, making it point to the new nodes N_1 and N_2 . The nodes N_1 and N_2 are created by the chunk's split of the old root. The thread that succeeds in replacing the B+tree root pointer

Algorithm 29: The code of the split of the root in the lock-free B$^+$tree.

```
29.1 void SplitRoot (Node* root, sepKey, Node* n1, Node* n2) {
 1: Node* newRoot = Allocate();          // Allocate new root with freeze state set to INFANT
 2: newRoot→<freezeState, joinBuddy> = <NORMAL, NULL>; newRoot→root=1;
 3: newRoot→height = root→height+1;                          // Update new root fields
 4: // Construct new root with old root's new nodes. Son with higher keys is pointed with ∞
    key entry
 5: addRootSons(newRoot, sepKey, n1, ∞, n2);
 6: CAS(&(btree→root), root, newRoot); // Try to replace the old root pointer with the new
 7: // If CAS is unsuccessful, then old root's new nodes were inserted under other new root,
 8: CAS(&(n1→<freezeState,joinBuddy>), <INFANT, NULL>, <NORMAL, NULL>);
 9: CAS(&(n1→<freezeState,joinBuddy>), <INFANT, NULL>, <NORMAL, NULL>);
10: return;
}
```

Algorithm 30: Check if the merge of the root is needed and perform if needed

```
30.1 void MergeRoot (Node* root, Node* posibleNewRoot, Node* c1, c1Key) {
 1: rootEntNum=GetEntNum(root→chunk,&firstEnt,&secondEnt);    // Count the entries in
    the list (do not use counter)
 2: if( rootEntNum > 2 ) { DeleteInChunk(&(root→chunk), c1Key, c1); return; }
 3: // rootEntNum is 2 here, check that first entry points to the frozen low node second on
    infant new possible root
 4: if((firstEnt→data == c1) && (secondEnt→data == posibleNewRoot)) {
 5:    CAS(&(posibleNewRoot→root), 0, 1);                          // Mark as root
 6:    CAS(&(btree→root), root, posibleNewRoot);     // Try to replace the old root pointer
    with the new
 7:    // If CAS is unsuccessful, then old root was changed by someone else
 8: }
 9: return;
}
```

from the old root to a new root (allocated by this thread), inserts the actual new root. Other allocated root's candidate are freed.

The actual root split occurs in a single CAS instruction. The split of the root doesn't prevent any other threads from going through the root (to perform operations on other nodes). In Algorithm 29 we use the method *addRootSons()*, which is local and sequential. It simply installs the new entries and we do not present its code. *SplitRoot()* uses this method to install two entries to the new root's chunk. The first entry holds the highest key in N_1 as key and a pointer to N_1 as data; the second entry holds the ∞ key and a pointer to node N_2.

3.10.2 Root Merge

Special care is required for handling the root merge. The root needs to be merged when the number of its children is reduced from two to one, that is, when its last two children are merged. To maintain tree balance, we do not allow a root node pointing to one single descendant. The *MergeRoot()* method (presented in Algorithm 30) is called on every merge of the root's children from the *Insert-*

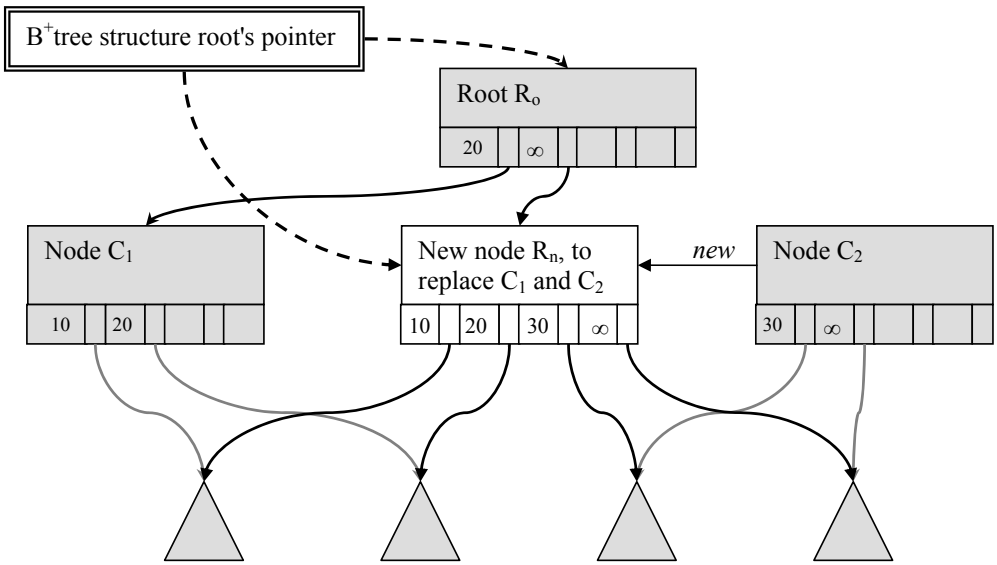


Figure 3.4: The diagram presenting the merging of the root. The initial B$^+$tree is marked in grey. Node R_o is an old root that initially had two children C_1 and C_2. Node C_2 is frozen as a master. Node C_1 is frozen as a slave for C_2.

MergeNode() method. A pictorial view of the root merge can be seen in Figure 3.4. We have an old root R_o having two children C_1 and C_2, while node C_2 was found as having too few entries and node C_1 was chosen as C_2's slave. Node R_n is created due to C_2's chunk merge and it needs to replace C_1 and C_2. In *MergeRoot()* method, as with every merge, the parent's entry pointer pointing to C_2 is first changed to point to R_n, after which C_2's parent is checked to determine whether it is the root.

Before an entry is deleted from the root node (due to a merge of the root's children), *MergeRoot()* is invoked and checks whether the root should be merged instead of just deleting one of its entries. If so, we recheck that root merge is needed and then set the root bit on new node R_n. Finally, we try to replace the pointer to the root. The *MergeRoot()* method's first input parameter is the current root; its second input parameter is the pointer to the new node, reference to which was already inserted into the root by replacing the old son pointer. This is the possible new root. The third input parameter is a pointer to the old low-value root child, which must be removed from the parent during a regular merge. We denote it *c1*. The key is the key located on the root's entry, whose data should point to the old low-value root child (*c1*).

We check whether the root has too few children by counting all reachable, non-deleted entries on the non-frozen root (the entries are pointing to the children on the root). This is done via the *GetEntNum()* method, which goes over the list and counts non-deleted entries. In addition, *GetEntNum()* copies the first and second entries that it finds to the given entry's pointers. If *GetEntNum()* finds more then two entries, the remaining reference to the frozen low node is deleted and we return to *InsertMergeNode()*. Otherwise we verify that the last two children

Algorithm 31: The replacing the key-data world value in the entry.

```
31.1 Bool ReplaceInChunk (chunk* chunk, key, exp, new) {
 1: while ( TRUE) {
 2:   if ( !Find(chunk, key) ) return FALSE;
 3:   if ( !CAS(&(cur→keyData), exp, new) ) {        // assume no freeze bit set in exp - no
      replace on frozen entry
 4:     if ( isFrozen(&(cur→keyData)) ) { // if CAS failed due to freeze help in freeze and try
      again
 5:       chunk = Freeze(chunk, key, exp, new, REPLACE, &result);
 6:       if ( chunk == NULL ) return result;       // Freeze completed the replace
      operation
 7:       continue;
 8:     }
 9:     return FALSE;              // CAS failed due to unexpected key or data, return FALSE
10:   } // end of if CAS wasn't successful
11:   return TRUE;                                   // CAS was successful, return TRUE
12: } // end of while
    }
```

are indeed those we received as input and try to replace the pointer to the root if necessary.

3.11 Minor Modifications to the Chunk Interfaces

In this section we present some minor changes to the chunk list interfaces as they appeared in the previous chapter. These changes are necessary to our B$^+$tree implementation.

3.11.1 The addition of replace interface to the list

Because the replace operation is not a standard interface for lists, it didn't appear in the previous chapter. It is, however, required for the implementation of the B$^+$tree we present it here. It allows the value of data in the key-data word of an entry, to be changed (replaced), without the need to remove the entire entry from the list and insert it back again. It is used to replace the data only (i.e., the pointer to the descendant), making the entry point to another node in the B$^+$tree.

The *ReplaceInChunk()* method gets as input the key, the expected key and data (in a single word), and the new key and data values where only the data is different (also in a single word). The code appears in Algorithm 31. It starts by finding the entry e that holds the input key and then uses a CAS to atomically replace the key-data value, assuming e currently holds the expected key-data value, that is given in the input. If the *ReplaceInChunk()* method fails to find the key in the list or to find the expected data, it returns with a FALSE. If the entry is frozen, *ReplaceInChunk()* first needs to help finishing the freeze process. It is a caller's responsibility to assure the replace doesn't wrong the order of the list and to provide the expected value without a frozen bit set.

3.11.2 The insert and delete operations

Insert: For the insert operation we add a versioning mechanism in order to ensure that splits are executed one at a time. We avoid the ABA problem and ensure that an insert executed by the *InsertSplitNodes()* method is done exactly once. The change is relevant only to insertions of internal nodes. When an insert determines that it is working on a leaf chunk, it works as usual (as presented in the previous chapter). The problematic case comes from the following scenario. Assume a process P is executing a split of a node A, replacing A with two new nodes B_1 and B_2 . When the split is almost done, P invokes the insertion of a new entry pointing to B_1 into A's parent C. Assume P is delayed just before the insert operation and in the meantime B_1's entry is inserted by a helper and then even deleted. If P wakes up and continues with previous insert, the frozen node B_1 can be erroneously inserted again into the B$^+$tree.

In order to solve this problem and a similar delete problem we describe below, we include another field in the *nextEntry* word. The *nextEntry* is a pointer to the next entry in the chunk's list with its two LSB bits used for deleted and frozen bits. It is reasonable to assume that a chunk will not include more than 1024 entries. So we can use the entry indexes inside chunk's entries array, instead of the pointers. Thus, in the *nextEntry* word, we leave 10 bits for next entry index, 2 bits for deleted and frozen bits, and the remaining bits (52) can be used to present a version number, which will be updated each time entry's next pointer is updated.

The check is performed in *InsertEntry()* method of chunk. This methods gets the possible new entry location using *Find()* method and *prev* and *curr* global pointers as explained in Section 3.7. After getting the possible new entry location (as part of it the version number of *prev*), we should check whether the data pointed by the new entry is still not a frozen node. Then we will be able to insert the new entry only if the version of *prev* wasn't changed. This solves the ABA problem because, after a node (referenced from the data field) is removed from the tree, it is always frozen.

Delete: The delete operation that we use in the given B$^+$tree is also slightly different from the one described in the previous chapter. When delete concludes it is working on a leaf chunk, it works as usual (similarly to the previous chapter). Otherwise, when delete is used on a chunk of an internal node, delete ensures that only the entry with given key *and* data is deleted. Usually a delete operation in a list is based only on a key. In our case delete should get not only the expected key, but also the expected data (i.e., a pointer to the descendant) that should be associated with this key. If the key is found, but the data is different, then the delete operation fails in the same way as when the key is not found. To avoid any kind of ABA problem and to ensure that delete (as part of a join) is applied only once, we use the versioning of the pointers. When an entry with an expected key and data is found, we also record the version of its *nextEntry* word where we are going to set the deleted bit. Next we check if data is pointing to the frozen node;

Algorithm 32: Freeze stabilization.

```
32.1 void StabilizeChunk(chunk* chunk) {
 1:  maxKey = ∞;  Find(chunk, maxKey);                  // Implicitly remove deleted entries
 2:  foreach entry e {
 3:    key = e→key; eNext = e→next;
 4:    if ( (key != ⊥) && (!isDeleted(eNext)) )        // This entry is allocated and not deleted
 5:      if ( !Find(chunk, key) ) InsertEntry(chunk, e, key);    // This key is not yet in the list
 6:  } // end of foreach
 7:  return;
     }
```

Algorithm 33: Freezing all entries in a chunk

```
33.1 void MarkChunkFrozen(chunk* chunk) {
 1:  foreach entry e {
 2:    savedWord = e→next;
 3:    while ( !isFrozen(savedWord) ) {                 // Loop till the next pointer is frozen
 4:      CAS(&(e→next), savedWord, markFrozen(savedWord));
 5:      savedWord = e→next;                            // Reread from shared memory
 6:    }
 7:    savedWord = e→keyData;
 8:    while ( !isFrozen(savedWord) ) {                 // Loop till the keyData word is frozen
 9:      CAS(&(e→keyData), savedWord, markFrozen(savedWord));
10:      savedWord = e→keyData;                         // Reread from shared memory
11:    }
12:  } // end of foreach
13:  return;
     }
```

if it is, we set the deleted bit if the version number is still the same.

This change is needed in order to ensure that a node is not removed twice from its parent, due to a delayed merge helper. Assume a process P is proceeding with a merge of nodes A and B. It invokes the deletion of A's entry on A's parent C. Let's assume that the key leading to node A is k. Assume P is delayed just before the delete operation, and in the meantime A's entry is deleted and key k is again inserted with an entry pointing to some new node N. If P continues with an unchanged version of delete, node N can be erroneously deleted from the B$^+$tree. With the new delete version, node N cannot be deleted since the pointer to node N is not the expected data that should be the pointer to A.

3.11.3 Freeze Functionality Code

For self-containment, in this section we present the code of the freeze related functions initially presented in the previous chapter. We avoid repeating the explanations of the code and only highlight the differences.

Initially, in the previous chapter, the *Freeze()* method for a chunk consisted of the following 5 steps: (1) change the freeze state to frozen; (2) mark all the entries as frozen and stabilize the chunk using *MarkChunkFrozen()* and *StabilizeChunk()*; and (3) decide whether the chunk should be split, joined, or copied, using the

Algorithm 34: Determining the freeze action.

```
34.1 recovType FreezeDecision (chunk* chunk) {
  1: entry* e = chunk→head→next;     int cnt = 0;
  2: while ( clearFrozen(e) != NULL) { cnt++; e = e→next; }     // Going over the chunk's list
  3: if ( cnt == MIN) return MERGE; if ( cnt == MAX) return SPLIT; return COPY;
 }
```

FreezeDecision() method. (If the chunk is to be merged, an additional step of finding the slave using the *FindJoinSlave()* method is required.) The two final steps are (4) creating new chunks according to the decision made and attaching them to the old chunk using *FreezeRecovery()*; and (5) calling the *CallForUpdate()* method to insert the new chunk(s) in place of the old one.

The *MarkChunkFrozen()*, *StabilizeChunk()* and *FreezeDecision()* methods used for the chunk lists and B$^+$tree are exactly the same as in the previous chapter. The modifications to the *Freeze()* and *FreezeRecovery()* methods (relative to their origin in the previous chapter) are mostly due to changes in the freeze states. For the chunk list we used fewer freeze states (only three), but for the B$^+$tree node we use eight. In this chapter the parameters for *Freeze()* and *FreezeRecovery()* methods are nodes instead of chunks; both are very similar. Finally, in B$^+$tree we have a design point that requires master and slave parent entries to be located on the single parent node (explained in Section 3.4).

The *FreezeRecovery()* method presented in Algorithm 36 is similar to one presented in the previous chapter. In the previous chapter the *FreezeRecovery()* method has the following structure. First, the required new node or nodes are prepared (first switch statement). Then (second switch statement), the thread proceeding with the freeze recovery tries to promote its initial purpose. In other words, the thread carrying on the freeze recovery of frozen node N, does it in order to progress with the initial purpose (i.e., a delete, insert, replace, etc. on N). Finally, the threads proceeding with *FreezeRecovery()* compete to attach new node(s) to the old frozen one. The thread that succeed to attach also promotes its initial purpose.

In the B$^+$tree freeze recovery an additional check is carried between the first and second switch-statements. Recall that we keep an invariant (for simplicity and correctness), by which the parent entries that point to the master and the slave always reside on the same node. Namely, we do not allow entries that point to a master and its slave to be separated into different nodes due to a parent's split or borrow, until new nodes replace the frozen master and slave. It is this case that we test for between the first and second switch. This check is relevant only if we have two new non-leaf nodes. If the last entry on *newNode1* or the first entry on *newNode2* is pointing to a frozen node that has a merge buddy, then we move both of these entries into *newNode2*, so that they reside together. Apart from the previously mentioned alteration, the *FreezeRecovery()* method for the B$^+$tree differs slightly from the one in the previous chapter in: updating the *creator* field,

taking care to promoting the replace operation, and the way of determining which new node is going to replace the frozen one. All these changes are to ensure that the chunk will work properly as a node in the tree.

Algorithm 35: The main freeze method.

```
35.1 chunk* Freeze(Node* node, key, expected, data, triggerType tgr,
     Bool* res) {
 1: CAS(&(node→<freezeState,mergeBuddy>), <NORMAL, NULL>, <FREEZE, NULL>);
 2: // At this point, the freeze state is neither NORMAL nor INFANT
 3: switch ( node→<freezeState,*> ) {
 4:   case COPY: decision = COPY; break;   // If the freeze state is already specifically set to
    split,
 5:   case SPLIT: decision = SPLIT; break;   // copy or merge, only freeze recovery is needed
 6:   case MERGE: decision = MERGE; mergePartner=node→<*,mergeBuddy>; break;      //
    mergePartner is already set
 7:   case REQUEST_SLAVE: decision = MERGE; mergePartner = FindJoinSlave(node);
 8:     if (mergePartner != NULL) break;              // If partner is NULL node was turned to
    SLAVE_FREEZE, continue
 9:   case SLAVE_FREEZE: decision = MERGE; mergePartner = node→<*,mergeBuddy>;
10:     // Swap between node and mergePartner, so node is always the master and
    mergePartner is the slave
11:     tmp = mergePartner; mergePartner = node; node = tmp;
12:     MarkChunkFrozen(mergePartner→chunk); StabilizeChunk(mergePartner→chunk); //
    Verify slave is frozen
13:     CAS(&(node→<freezeState,mergeBuddy>),          // Slave is set, verify master is in
    MERGE state
14:                 <REQUEST_SLAVE, mergePartner>, <MERGE, mergePartner>);
15:     break;
16:   case FREEZE: MarkChunkFrozen(node→chunk); StabilizeChunk(node→chunk);
17:   decision = FreezeDecision(node→chunk);
18:     switch (decision) {
19:         case COPY: CAS(&(node→<freezeState,mergeBuddy>), <FREEZE,
    NULL>, <COPY, NULL>,); break;
20:         case SPLIT: CAS(&(node→<freezeState,mergeBuddy>), <FREEZE,
    NULL>, <SPLIT, NULL>,); break;
21:         case MERGE: mergePartner = FindJoinSlave(node);
22:             if (mergePartner==NULL) {   // The node become slave, its merge
    body is master. The node need to be
23:                 mergePartner = node; node = node→<*,mergeBuddy>;   //
    the master and mergePartner - the slave
24:                 CAS(&(node→<freezeState,mergeBuddy>),
25:
    <REQUEST_SLAVE,mergePartner>,<MERGE,mergePartner>);
26:             } break;
27:     } // end of switch on decision
28: } // end of switch on freeze state
29: return FreezeRecovery(node, key, expected, data, decision, mergePartner, trigger, res);
}
```

The *Freeze()* method presented in the previous chapter works the same here, as

presented in Algorithm 35. But because here we have the possibility to determine the freeze decision from the freeze state of a node, we use it for some optimization. If the reason for the freeze is already known, located in *freezeState* field, and we can figure out all needed data for the freeze recovery, we skip some steps and invoke *FreezeRecovery()* method. This is done in the switch statement. A thread that enters the FREEZE case of the switch statement behaves according to the *Freeze()* method in the previous chapter, with modification that inside the FREEZE case statement we set the freeze state according to its new possible values. The new freeze state values didn't exist in the previous chapter. A final nuance is to check whether the *FindJoinSlave()* method returned a NULL value when the slave and master swapped their roles.

Algorithm 36: The freeze recovery.

```
Node* FreezeRecovery(Node* oldNode, key, expected, input, recovType, Node*
mergePartner, triggerType trigger, Bool* result) {
 1: retNode=NULL; sepKey=∞; newNode1=Allocate(); newNode1→creator=oldNode; newNode2=NULL;
 2: switch ( recovType ) {
 3:    case COPY: copyToOneChunkNode(oldNode, newNode1); break;
 4:    case MERGE:
 5:       if ( (getEntrNum(oldNode)+getEntrNum(mergePartner))≥MAX ) {          // Borrow, two new nodes
 6:         newNode2 = Allocate(); newNode1→nextNew = newNode2;          // Connect two new nodes together
 7:         newNode2→creator=oldNode; sepKey=mergeTo2Nodes(oldNode,mergePartner,newNode1,newNode2);
 8:       } else mergeToOneNode(oldNode,mergePartner,newNode1); break;          // Merge into single new node
 9:    case SPLIT:
10:       newNode2 = Allocate(); newNode1→nextNew = newNode2;          // Connect two new nodes together
11:       newNode2→creator=oldNode; sepKey=splitIntoTwoNodes(oldNode,newNode1,newNode2); break;
12: } // end of switch
13: // If there are two new non-leaf nodes check and do not separate master and slave entries
14: if ( (newNode2 != NULL) && (newNode2→height != 0) ) {
15:    leftNode = (Node*)(getMaxEntry(newNode1))→data; leftState = leftNode→<freezeState,*>;
16:    rightNode = (Node*)newNode2→chunk→head→next→data; rightState = rightNode→<freezeState,*>;
17:    if ( (rightState == <REQUEST_SLAVE, *>) && ((leftState == NORMAL) || (leftState == INFANT) ||
   (leftState == FREEZE) ||
18:          (leftState == COPY) || (leftState == <SLAVE_FREEZE, rightNode>))) {
19:       moveEntryFromFirstToSecond(newNode1, newNode2); sepKey = (getMaxEntry(newNode1))→key;
20:    } else if (rightState == <MERGE, leftNode>) {
21:       moveEntryFromFirstToSecond(newNode1, newNode2); sepKey = (getMaxEntry(newNode1))→key;
22:    } else if ( (rightState == INFANT) && (leftState == <SLAVE_FREEZE, *>) ) {
23:       if (rightNode→creator == leftNode→<*,mergeBuddy>)
24:          { moveEntryFromFirstToSecond(newNode1, newNode2); sepKey = (getMaxEntry(newNode1))→key; }
25:    }
26: }
27: switch ( trigger ) {                          // Perform the operation with which the freeze was initiated
28:    case DELETE:                                 // If key will be found, decrement counter has to succeed
29:       *result = DeleteInChunk(newNode1→chunk, key);
30:       if ( newNode2 != NULL ) *result = *result || DeleteInChunk(newNode2→chunk, key); break;
31:    case INSERT:                                 // input should be interpreted as data to insert with the key
32:       if ( (newNode2 != NULL) && (key<sepKey) ) *result = InsertToChunk(newNode2→chunk, key, input);
33:       else *result = InsertToChunk(newNode1→chunk, key, input); break;
34:    case REPLACE:
35:       if ((newNode2!=NULL)&&(key<sepKey)) *result=ReplaceInChunk(newNode2→chunk,key,expected,input);
36:       else *result = ReplaceInChunk(newNode1→chunk, key, expected, input); break;
37:    case ENSLAVE:     // input should be interpreted as pointer to master trying to enslave, only in case of COPY
38:       if ( recovType == COPY ) newNode1→<freezeState,mergeBuddy> = <SLAVE_FREEZE, (Node*)input>;
39: } // end of switch
40: // Try to create a link to the first new node in the old node.
41: if ( !CAS(&(oldNode→new), NULL, newNode1) ) {
42:    // Determine in which of the new nodes the destination key is now located.
43:    if ( key<=sepKey ) retNode=oldNode→new; else retNode=oldNode→new→nextNew;
44: }
45: if (newNode1→<freezeState,*> == <SLAVE_FREEZE, *>) {
46:    m = newNode1→<*,mergeBuddy>;          // If the new chunk is enslaved, correct its master to point on it
47:    CAS(&(m→<freezeState,mergeBuddy>),<REQUEST_SLAVE, oldNode>,<REQUEST_SLAVE, newNode1>);
48: }
49: CallForUpdate(recovType, oldNode, key);
50: return retNode;
}
```

Chapter 4

Drop the Anchor: Lightweight Memory Management for Non-Blocking Data Structures

4.1 Introduction

Non-blocking data structures [26, 30] are fast, scalable and widely used. In the last two decades, many efficient non-blocking implementations for almost any common data structure have been developed. However, when designing a dynamic non-blocking data structure, one must address the non-trivial issue of how to manage its memory. Specifically, one has to ensure that whenever a thread removes some internal node from the data structure, then (a) the memory occupied by this node will be eventually deallocated (i.e., returned to the memory manager for arbitrary reuse), and (b) no other concurrently running thread will access the deallocated memory, even though some threads might hold a reference to the node.

Previous attempts to tackle the memory management problem had limited success. Existing non-blocking algorithms usually take two standard approaches. The first approach is to rely on automatic garbage collection (GC), simply deferring the problem to the GC. By doing this, the designers hinder the algorithm from being ported to environments without GC [15]. Moreover, the implementations of these designs with currently available (blocking) GC's cannot be considered non-blocking.

The second approach taken by designers of concurrent data structures is to adopt one of the available non-blocking memory management schemes. The most common schemes are probably the Hazard Pointers technique by Michael [41] or the similar Pass the Buck method by Herlihy et al. [28]. In these schemes, each thread has a pool of global pointers, called *hazard pointers* in [41] or *guards* in [28], which are used to mark objects as "live" or ready for reclamation. When a thread t reclaims a node, t adds the node to a special local reclamation buffer. Once in a while, t scans its buffer and for each node it checks whether some other thread has

a hazard pointer[1] to the node. If not, that node can be safely deallocated. Special attention must be given to the time interval after a thread obtains a reference to an object and before it registers this object in a hazard pointer. During this time, the object may be reclaimed and reallocated. Thus, by the time it gets protected by a hazard pointer, it could have become a completely different entity. This delicate point enforces validation of the object's state after assigning it with a hazard pointer.

Although these techniques are not universal (i.e., there is no automatic way to incorporate them into a given algorithm), they are relatively simple. Moreover, a failure of one thread prevents only a small number of nodes (to which the failed thread has references in its hazard pointers) from being deallocated. The major drawback of these techniques, however, is their significant runtime overhead, caused mainly by the management and validation of the global pointers required before accessing *each* internal node for the first time [24]. Along with that, expensive instructions, such as memory fences or compare-and-swap (CAS) instructions [24, 28, 41], are required for correctness of those schemes. Moreover, if the validation fails, the thread must restart its operation on the data structure, harming the performance further.

Another known method for memory management uses per-thread timestamps, as in [22], which are incremented by threads before every access to the data structure. When a thread removes a node, it records the timestamps of other threads. Later, it can deallocate the node once all threads increase their timestamps beyond the recorded values. Although this method is very lightweight, it is vulnerable to thread delays and failures. In such cases, memory space of an unbounded size may become impossible to reclaim [41].

In this chapter, we concentrate on the linked list, one of the most fundamental data structures, which is particularly prone to the shortcomings of previous approaches [24]. The presented technique eliminates the performance overhead associated with the memory management without sacrificing the ability to deallocate memory in case of thread failures. The good performance of our technique stems from the assumption that thread failures are typically very uncommon in real systems, and if they do occur, this is usually indicative of more serious problems than being unable to deallocate some small part of memory. Our approach provides a flexible tradeoff between the runtime overhead introduced by memory management and the size of memory that might be lost when some thread fails.

Our memory management technique builds on a combination of three ideas: timestamps, anchors and freezing. As in [22], we use per-thread timestamps to track the activity of each thread on the data structure. Similarly to [41], we use global pointers, which we call anchors. Unlike [41], however, a thread drops the anchor (i.e., records a reference in the anchor) every bunch of node accesses, e.g., every one thousand nodes it traverses. As a result, the amortized cost of anchor management is spread across multiple node accesses and is thus very low. To

[1] In this dissertation we will use the term "hazard pointers", but guard pointers are equally relevant.

recover the data structure from a failure of a thread t, we apply freezing [5]. That is, using t's anchors, other threads mark nodes that t may hold a reference to, as frozen. Then they copy and replace the frozen part of the data structure, restoring the ability of all threads to deallocate memory. The recovery operation is relatively expensive, but it is required only in the uncommon case in which a thread fails to make progress for a long while. Thus, the overall cost of the memory management remains very low.

We have implemented our scheme in C and compared its performance to the widely used implementation of the linked list based on Hazard Pointers (HP) [41]. Our performance results show that the total running time, using the anchor-based memory management, is about 250–500% faster the one based on HP. We also discuss how to apply our technique on other data structures, where the use of other approaches for memory management is more expensive.

4.2 Related Work

Memory management can be fairly considered as the Achilles heel of many dynamically sized non-blocking data structures. In addition to the techniques mentioned in the introduction (that use per-thread timestamps [22] or global pointers [28, 41]), one can also find an approach based on reference counting [12, 21, 48, 52]. There, the idea is to associate a counter with every node, which is updated atomically when a thread gains or drops a reference to the node. Such atomic updates are typically performed with a fetch-and-add instruction, and the node can be safely removed once its reference count drops to zero. This approach suffers from several drawbacks, such as requiring each node to keep the reference count field even after the node is reclaimed [48, 52] or using uncommon atomic primitives, such as double compare-and-swap (DCAS) [12]. The major problem, however, remains performance [24, 41], since even when applying a read-only operation on the data structure, this approach requires atomic reference counter updates on every node access.

In a related work, Hart et al. [24] compare several memory management techniques, including hazard pointers, reference counters, and so-called quiescent-state-based reclamation. In the latter, the memory can be reclaimed when each thread passes through at least one quiescent state [38], in which it does not hold any reference to shared nodes, and in particular, to nodes that have been removed from the data structure. In fact, the timestamp-based technique [22] discussed in the introduction can be seen as a special case of the quiescent-state approach. Hart et al. [24] find that when using hazard pointers or reference counters, expensive atomic instructions, such as fences and compare-and-swaps (CAS) executed for every traversed node, dominate the performance cost. Quiescent-state reclamation usually performs better, but it heavily depends on how often quiescent states occur. Moreover, if a thread fails before reaching the quiescent state, no memory can be safely reclaimed from that point.

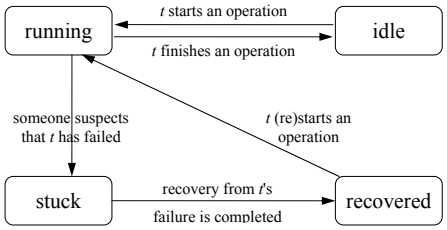


Figure 4.1: Transition diagram for possible states of the thread t

Dragojevic et al. [15] consider how hardware transactional memory (HTM) can help to alleviate the performance and conceptual difficulties associated with memory management techniques. In contrast to [15], our algorithm does not rely on special hardware support, such as HTM.

The freezing idea was previously used in the context of concurrent data structures by Braginsky and Petrank in their recent work on chunk-based linked lists [5]. There, list nodes are grouped into chunks for better cache locality and list traversal performance. The freezing technique is used in [5] for list restructuring to notify threads that the part of the data structure they are currently using is obsolete. This is done by setting a special freeze-bit on pointers belonging to nodes in the obsolete part, making the pointers/nodes unsuitable for traversing. A thread that fails to use a frozen pointer realizes that this part of the data structure is obsolete and it restarts its operation, usually after helping to accomplish the list restructuring procedure that froze that part.

4.3 An Overview of Drop the Anchor

As mentioned in the introduction, our technique relies on three building blocks, namely timestamps, anchors, and freezing. A thread t manages a monotonically increasing timestamp in the following way. When t starts its operation on a list data structure, it reads the timestamps of all threads and sets its timestamp to the maximal value it read plus one. When t finishes its operation, it simply marks its timestamp as idle.

The timestamp of t is associated with two flags, STUCK and IDLE. These flags specify one of the following states of the timestamp (and of the corresponding thread): *running* (both flags are turned off, meaning that t has a pending operation on the data structure), *idle* (only the IDLE flag is on, meaning that t does not have any pending operation on the data stucture), *running, but stuck*, which we call for brevity simply *stuck* (only the STUCK flag is on, meaning that t with a pending operation is suspected by other thread(s) to be stuck) and *recovered* (both flags are turned on, meaning that other threads have frozen and copied the memory that might be accessed by t). The transition between these states is captured in Figure 4.1. Normally, t moves between running and idle states. Once some thread suspects t to be stuck, t's timestamp is marked as stuck. The only way for t to return to the running state is to go through the recovered state (cf. Figure 4.1).

The timestamps are also used to mark the insertion and deletion times of list nodes. That is, each node in the list has two additional fields, which are set as follows. When t decides to insert (remove) a node into (from) the list, it sets the node's insertion (deletion, respectively) timestamp field to be higher by one from the maximal timestamp value that it observes among the timestamps of all threads.

The nodes deleted by t are stored in t's special reclamation buffer, which is scanned by t once in a while (as in [41]). During each scan, and for each deleted node n, t checks whether the deletion time of n is smaller than the current timestamps of other threads (plus an additional condition described later), and if so, deallocates the node. This check ensures that all threads have started a new list operation since the time this node was removed from the list, and therefore, no thread can be viewing this node at this time.

If threads did not fail, this would be everything needed to manage the memory of non-blocking lists by a traditional epoch-based approach [22]. Unfortunately, thread failures may happen. In the design described so far, if a thread fails during its operation on the list, no additional node can be deallocated, since the timestamp of the failed thread would not advance.

To cope with the problem of thread failures, we use two additional concepts, namely anchors and freezing. Anchors are simply pointers used by threads to point to list nodes. In fact, hazard pointers[41] can be seen as a special case of anchors. The difference between the two is that the anchor is not dropped (set) before accessing every internal node in the list, but rather every ten, one hundred, or several thousands of node accesses (the frequency is controlled by the ANCHOR_THRESHOLD parameter). As a result, the amortized cost of anchor management is significantly reduced and spread across the traversal of (controllably) many nodes in the list data structure. The downside of our approach, however, is that when a thread t is suspected of being stuck, other threads do not know for sure which object t may access when (and if) it revives. They only know a range of nodes where t might be, which includes the node pointed by t's anchor plus additional nodes reachable from that anchored node. The suspecting threads use this range to recover the list from the failure of t. Specifically, they freeze all nodes in the range by setting the special *freeze-bit* of all pointers in these nodes[2]. Next, they copy all frozen nodes into a new sub-list. Finally, they replace the frozen nodes with the new sub-list and mark t's timestamp as recovered. This mark tells other threads that the list was recovered from t's failure. In other words, threads may again deallocate nodes they remove from the data structure, disregarding t's timestamp.

The recovery procedure is relatively heavy performance-wise and has certain technical issues, but in return, the common path, i.e., the traversal of the data structure, incurs virtually no additional operations related to memory manage-

[2]The freeze-bit is one of the least significant bits of a pointer, which are normally zeroed due to memory alignment.

```
1  struct GlobalMemoryManagementRec{
2      uint128_t timeStampAndAnchor;
3      uint64_t lowTimeStamp;
4  };
5
6  struct LocalMemoryManagementRec{
7      list_t reclamationBuffer;
8      uint64_t minTimeStamp;
9      uint32_t minTimeStampThreadID;
10     uint32_t minTimeStampThreadCnt;
11 };
```

Listing 4.1: Auxiliary records

ment. Since the recovery is expected to be very infrequent, we believe (and show in our performance measurements) that the complication associated with the recovery procedure pays off by eliminating the overhead in the common path. In the next section, we provide technical details of the application of this general idea into the concrete non-blocking implementation of the linked list.

4.4 Detailed Description

4.4.1 Auxiliary fields and records

We use the singly linked list of Harris [22] as a basis for our construction. To support our scheme, each thread maintains two records where it stores information related to the memory management. The first record is global, i.e., it can be read and written by any thread (not just the owner of the record), and used to manage the thread's timestamp and anchor. The second record is local, and is used during object reclamations and for deciding whether the recovery procedure is necessary.

The structure of the records is given in Listing 4.1. The global record contains two fields, `timestampAndAnchor` and `lowTimeStamp`. The `timeStampAndAnchor` field contains the timestamp, the anchor, and the IDLE and STUCK bits of the thread, combined into one word so that all can be modified atomically. The width and the actual internal structure of the field depends on the underlying machine. In certain settings of 64-bit Linux-based architectures, the virtual memory addressing requires 48 bits; the two least significant bits in a pointer are typically zeroed due to memory alignment. Moreover, most existing architectures support wide-CAS instruction, which operates atomically on two adjacent memory words (i.e., 128 bits). In such settings, we allocate 64 bits for the timestamp and 64 bits for the anchor pointer, including two bits for two flags, which specify the state of the thread (i.e., running, idle, stuck and recovered). In the settings where only 64 bits can be a target for a CAS instruction, one can use 48 bits or fewer for the anchor pointer and, respectively, 16 bits for timestamp. However, different allocation techniques can be used to require fewer bits for the pointer.

When a thread t accesses the list, it reads the timestamps of all threads in the system and sets its own timestamp to one plus the maximum among

all the timestamp values that were read. It writes its new timestamp in the `timeStampAndAnchor` field, simultaneously setting the IDLE bit to zero. When t completes its operation, it simply turns the IDLE bit on (leaving the same timestamp value). The exact details of the manipulation of this field are provided in subsequent sections.

In addition to the `timeStampAndAnchor` field, the global record contains a field called `lowTimeStamp`. This field is set by t to the minimal timestamp observed by t when it starts an operation on the list. As described in Section 4.4.4, the `lowTimeStamp` field is used by other threads when they try to recover the list from the failure of t (to identify nodes that were inserted into the list before t started its current operation).

The local record has four fields. The description of their role is given in Section 4.4.3.

Along with adding auxiliary records for each thread, we also augment each node in the linked list with two fields having self-explanatory names, `insertTS` and `retireTS`. These fields are set to the current maximal timestamp plus one when a node is inserted into or deleted from the list, respectively.

4.4.2 Anchor maintenance

Anchor maintenance is carried out when threads traverse the list, looking for a particular key. The simplified pseudo-code for this traversal composes the `find` method given in Section 5.7.Recall that this method is used by all list operations in [22].

A thread counts the number of list nodes it has passed through and updates its anchor every ANCHOR_THRESHOLD nodes (where ANCHOR_THRESHOLD is some preset number). The anchor points to the first node in the list that can be accessed by the thread (which is the node pointed by `prev` in the `find` method). Anchor updates are made in the auxiliary `setAnchor` function also shown in Section 5.7. An anchor update may fail for thread t_i if some other thread t_j has marked the `timeStampAndAnchor` field of t_i as stuck, as explained in Section 4.4.4.

It is important to note that the actual update of the anchor is done with CAS (and not with a simple write operation) to avoid races with concurrently running threads that might suspect t_i being stuck and try to set the STUCK bit in t_i's `timeStampAndAnchor`. From a performance standpoint, however, the write operation of a hazard pointer, made on accessing every node, requires an expensive memory fence right after it [24, 41]. In contrast, the CAS in our approach is performed only every ANCHOR_THRESHOLD node accesses, and its amortized cost is negligible.

We note that the `find` function is allowed to traverse the frozen nodes of the list. A node is frozen if the second least significant bit in its `next` pointer is turned on (while the first least significant bit is used to mark the node as deleted [22]). If there is a need to update the `next` pointer of the frozen node, the update operation fails (as in [5]) and retries after invoking the `helpRecovery` method (pseudo-code

can be found in Section 5.7). As its name suggests, the latter method is used to help the recovery process of some stuck thread. This method is also called when a thread fails to update its anchor in `setAnchor`.

Finally, we note that at any time instant, list operations have references to at most two adjacent list nodes. (Recall that for the linked list data structure two hazard pointers are required [41]). As we require that a stuck thread will be able to access nodes only between its current anchor and (but not including) the next potential anchor, the ANCHOR_THRESHOLD parameter for the linked lists has to be at least 2.

4.4.3 Node reclamation

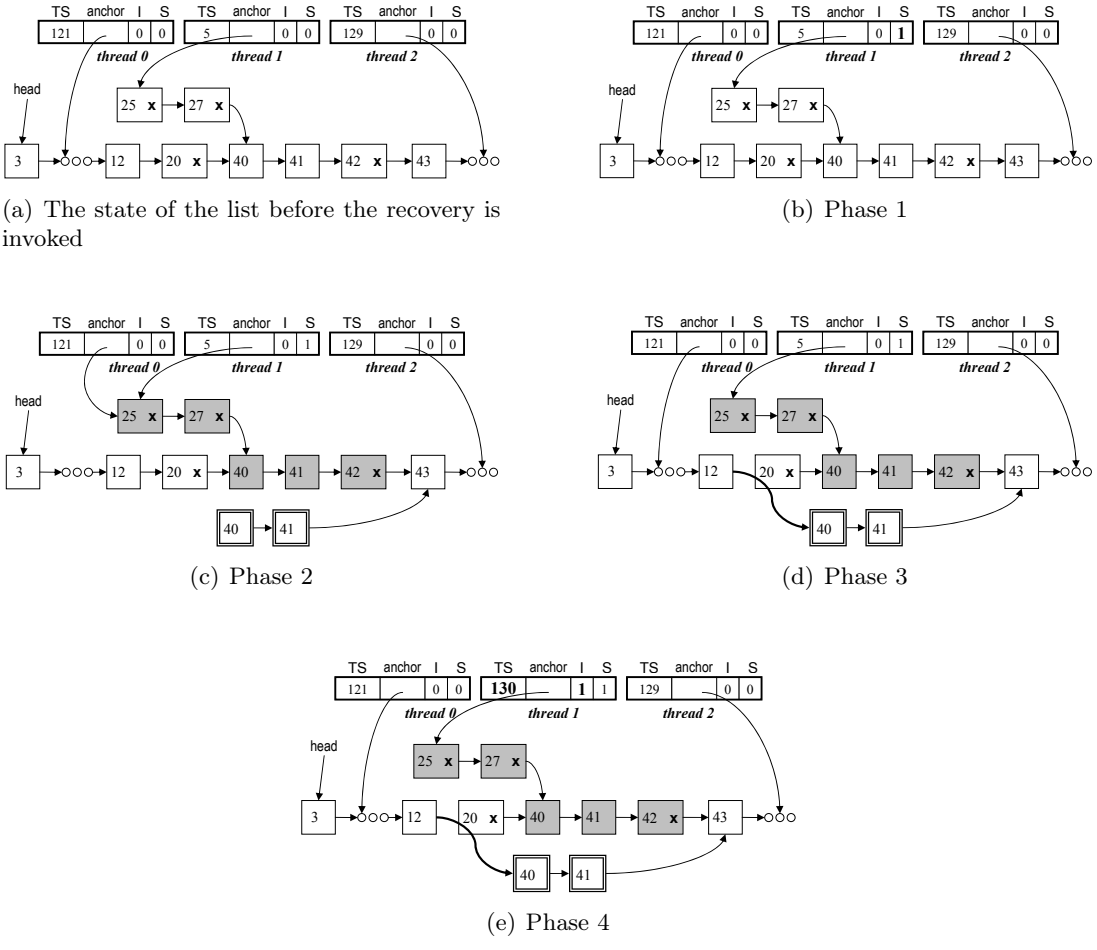
When a thread t_i removes a node from the list, it calls the `retireNode` method, which sets the deletion timestamp of the node (i.e., the `retireTS` field) to the current maximal timestamp plus one. Then, similarly to [41], the `retireNode` method adds the deleted node to a reclamation buffer. The latter is simply a local linked list (cf. Listing 4.1) where t_i stores nodes deleted from the list data structure, but not deallocated yet. When the size of the buffer reaches a predefined bound (controlled by the RETIRE_THRESHOLD parameter), t_i runs through the buffer and deallocates all nodes with the retire timestamp smaller than the current minimal timestamp (plus an additional condition elaborated in Section 4.4.4). Note that if the deletion time of a node n is smaller than the timestamp of a thread t_j, t_j started its last operation on the list after n was removed from the list; thus, t_j will never access n. Obviously, if this holds for any t_j, it is safe to deallocate n.

When t_i finds that some thread t_j exists such that the timestamp of t_j is smaller than or equal to the timestamp of one of the nodes in t_i's reclamation buffer, t_i stores the ID of that thread (i.e., j) in the `minTimeStampThreadID` field of its local memory management record (cf. Listing 4.1) and t_j's timestamp in the `minTimeStamp` field of that record. It also sets the `minTimeStampThreadCnt` field to 1. It is important to note that if several threads have the same minimal timestamp, t_i will store the smallest ID in `minTimeStampThreadID`. This will ensure that even if several threads are stuck with the same timestamp, all threads will consider the same thread in the recovery procedure.

On later scans of the reclamation buffer, if t_i finds that the thread t_j (whose ID is stored in t_i's `minTimeStampThreadID`) still has the same timestamp, t_i will increase the `minTimeStampThreadCnt` counter. Once the counter reaches the predefined RECOVERY_THRESHOLD parameter, t_i will suspect that t_j has failed and will start the recovery procedure described in Section 4.4.4.

4.4.4 Recovery procedure

The recovery procedure is invoked in one of the following three cases. First, it is invoked by a thread t_i that tries to deallocate an object n from its reclamation



(a) The state of the list before the recovery is invoked

(b) Phase 1

(c) Phase 2

(d) Phase 3

(e) Phase 4

Figure 4.2: Recovery phases. Nodes marked with 'x' are deleted, i.e., the delete-bit of their `next` pointer is turned on [22]. Shaded nodes are frozen, i.e., the freeze-bit of their `next` pointer is turned on.

buffer, but repeatedly finds a running thread t_j whose timestamp remains smaller than or equal to the timestamp of n (cf. Section 4.4.3). The second case is when a thread t_i tries to modify the `next` pointer of one of the nodes in the list, but finds that this node is frozen (cf. Section 4.4.2). Finally, the third case happens when a thread tries to update its anchor by modifying its `timeStampAndAnchor` field, but finds that some other thread turned the STUCK bit on in this field (cf. Section 4.4.2). In two last cases t_i invokes the `helpRecovery` method. There, t_i scans through global records of the threads, looking for a thread t_j with the STUCK bit in t_j's `timeStampAndAnchor` field turned on.

The recovery procedure consists of four phases (the code can be found in Section 5.7). We explain these phases using the example in Figure 5.4. Assume that at some point in time the list data structure is in the state depicted in Figure 5.4(a), and thread t_0 decides to recover the list from the failure of thread t_1. Before invoking the first phase of the recovery procedure, t_0 stores locally the current value of t_1's `timeStampAndAnchor` field. Then, in the first phase of the recovery procedure, t_0 attempts to modify t_1's `timeStampAndAnchor` field by turning the STUCK bit on using CAS operation (cf. Figure 5.4(b)). If this operation fails, t_0 rereads t_1's `timeStampAndAnchor` field and checks whether it was marked as stuck by some other thread. If not, it aborts the recovery procedure (since either t_1 is actually alive and has modified its `timeStampAndAnchor` field, or some other thread, i.e., t_2, has finished the recovery of t_1 and, as we will see later, turned both STUCK and IDLE bits on). Otherwise, if the CAS operation that turns the STUCK bit on succeeds, or if it fails, but t_1 is marked as stuck by another thread, t_0 proceeds to the second phase.

In the second phase of the recovery procedure, t_0 freezes and copies all nodes that t_1 might access if t_1 revived and traversed the list until realizing at the next anchor update that its anchor is marked as stuck. To identify such nodes, t_0 extracts t_1's anchor pointer out of the value stored in t_1's `timeStampAndAnchor` field (which points to node 25 in our example in Figure 5.4(a)). Then, t_0 starts setting the freeze-bit in the `next` pointers of reachable nodes, starting from node 25. It copies the frozen nodes (with the freeze-bit set off) into a new list. Note that some of the nodes may already be deleted from the list (e.g., node 25, 27 and 42 in Figure 5.4), but not disconnected or reclaimed yet. Such nodes are frozen, but they do not enter the new copied part of the list. The thread t_0 keeps freezing and copying until it passes through ANCHOR_THRESHOLD nodes having an insertion timestamp smaller than the value of t_1's `lowTimeStamp` field. In our example in Figure 5.4, let us assume that these are nodes 25, 27, 40, 41 and 42. Note that for traversing those nodes, t_0 had to update its own anchor to be the same as t_1's anchor in order to handle t_0's failure during the recovery procedure. At the end of the second phase the list looks as depicted in Figure 5.4(c). The pseudo-code for how we freeze the nodes and create the copies can be found in Section 5.7.

In the third phase, t_0 attempts to replace the frozen nodes with a locally copied part of the list. To this end, it runs from the beginning of the list data structure

and looks for the first (not-deleted) node m whose `next` pointer either points to the not-deleted frozen nodes or it is followed by a sequence of one or more deleted nodes such that the `next` pointer of the last node in the sequence points to a not-deleted frozen node (in Figure 5.4(c), m is the node 12). If such m were not found by reaching the end of the list data structure, t_0 would finish this phase, as it would assume that some other thread has replaced the frozen part of the list with the new list created by that thread. Otherwise, t_0 attempts to update m's `next` pointer to point to the corresponding copied node in the new list. If it fails, it restarts this phase from the beginning. Otherwise, t_0 inserts all nodes between m and the first frozen node (i.e., node 20 in Figure 5.4(c)) into its reclamation buffer in order to deallocate them later, bringing the list to the state exhibited in Figure 5.4(d). The code of the procedure for replacing frozen nodes can also be found in Section 5.7.

One subtlety that is left out of the code for lack of space, is the verification that the new local list indeed matches the frozen nodes being replaced. It is crucial to ensure that if a thread running the recovery procedure gets delayed, it does not replace another frozen part of the list when it resumes. To this end, we record the sources of the new nodes, when they are copied, and CAS the new list into the data structure only if it replaces the adequate original nodes that can be found in the recorded sources.

In the final, fourth phase, t_0 sets the IDLE bit in the t_1's `timeStampAndAnchor` field, marking t_1 as recovered. Additionally, t_0 promotes t_1's timestamp, recording the (logical) time when t_1 was recovered (cf. Figure 5.4(e)). Note that t_0 does not need to check whether its CAS has succeeded, since if it hasn't, some other thread has performed this operation. We denote a timestamp of a thread with IDLE and STUCK flags turned on as *recovery timestamp*.

4.4.5 The refined reclamation procedure

Thread t_1, considered stuck, might actually have a pointer to a node n which is already not a part of the list. For instance, in the state of the list shown in Figure 5.4(a), t_1 might be stopped while inspecting node 25 (or 27). If this node is currently in the reclamation buffer of some other thread t_k (i.e., t_0 or t_2), and if t_k does not consider t_1 after the recovery is done (i.e., t_k only checks that node 25's `retireTS` is smaller than the timestamp of any *running* thread), t_k might deallocate node 25 and t_1 might erroneously access this memory if and when it revives. Note that node 27 may already be unreachable from the node pointed by t_1's anchor by the time of t_1's recovery, if, e.g., the next pointer of node 25 was updated while t_1 was inspecting node 27. In this case, node 27 will not be frozen and copied at all. In order to cope with such situations, before deallocating a node we require its retire timestamp (`retireTS`) to be larger than the timestamp of any thread in the *recovered* state (in addition to being smaller than the timestamp of any running thread). This way we prevent nodes removed from the list before some thread got recovered from being deallocated, as the thread being recovered

might hold a pointer to such node. When (and if) the recovered thread becomes running again, it will be possible to reclaim those nodes. To summarize, when a thread wants to deallocate a node n, it checks that n is not frozen (i.e., the freeze-bit in its `next` pointer is not set) and that the following condition holds:

$$\text{MAX}(\{\text{timestamp of } t_x \mid t_x \text{ is recovered}\}) < \text{n.retireTS} < \text{MIN}(\{\text{timestamp of } t_x \mid t_x \text{ is running}\})$$

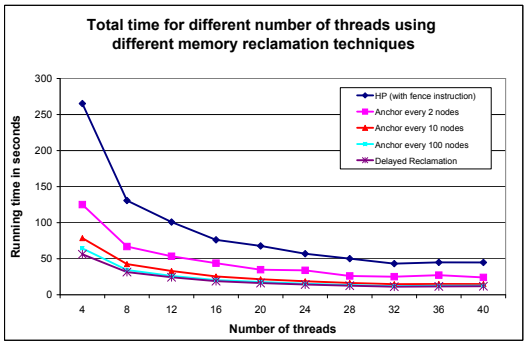
It should be noted that when calculations of the retire timestamp and the recovery timestamp are done simultaneously (by different threads), the retire timestamp can erroneously be higher than the recovery timestamp, and wrong reclamation can happen. Therefore, when calculating the retire timestamp for a node, we require a thread to pass twice over the timestamps of the threads verifying that no thread was marked stuck or recovered concurrently. If such thread(s) is found, the node is inserted into the reclamation stack as frozen.

For simplicity of presentation, in the algorithm described above frozen nodes are not reclaimed. Such nodes can only appear if threads fail and such a solution may be acceptable. However, frozen nodes can be easily reclaimed for recovered threads that have resumed operation. A recovered thread can reclaim nodes according to the recovery timestamps. Also, a frozen node that appears in a reclamation stack can be reclaimed using its `retireTS` field and the `lowTimeStamp` field of all stuck threads. Details are omitted.

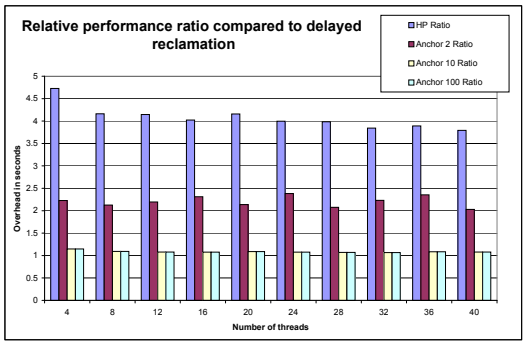
4.5 Performance Evaluation

We have implemented the non-blocking linked list data structure of Harris [22] with several memory management techniques. First, we have implemented the Hazard Pointers technique following the pseudo-code presented in [41], but with the additional memory fence instruction added just after the write of a new value to the hazard pointer of a thread [24]. Second, we have implemented our new Drop the Anchor technique presented in this paper. Finally, we have also implemented a simple technique, where nodes removed by a thread t from the list are added to t's reclamation stack and reclaimed later once 64 nodes are collected in the reclamation stack. We refer to this implementation as *delayed reclamation*. We note that this scheme is incorrect in a sense that it allows threads to access deallocated memory, but we used this implementation to represent a memory management scheme with a minimal performance impact. All our implementations were coded in C and compiled with -O3 optimization level.

We have run our experiments on the machine with two AMD Opteron(TM) 6272 16-core processors, operated by Linux OS (Ubuntu 12.04). We have varied the number of threads between 1 and 40, slightly above the number of threads that can run concurrently on this machine (32). If not stated otherwise, each test starts by building an initial list with 100k random keys. After that, we measure the total time of 320k operations divided equally between all threads. The keys for searches



(a) The total running time comparison for 20% inserts, 20% deletes and 60% searches.



(b) Memory management overhead referred to delayed reclamation for 20% inserts, 20% deletes and 60% searches.

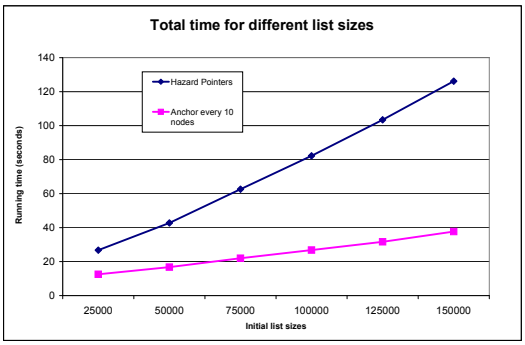
Figure 4.3: Drop the Anchor vs. Hazard Pointers for lists with the initial size of 100k keys, the mixed workload results.

and insertions are randomly chosen 20-bit sized keys. For deletion operations, we ensure that randomly chosen keys actually exist in the list in order to make the reclamation process substantial. The values of RECOVERY_THRESHOLD and RETIRE_THRESHOLD were always 64. All threads are synchronized to start their operations immediately after the initial list is built and we measure the time it takes to complete all operations by all threads. We run each test 10 times and present the average results. The variance of all reported results is below 1.5% of the average.

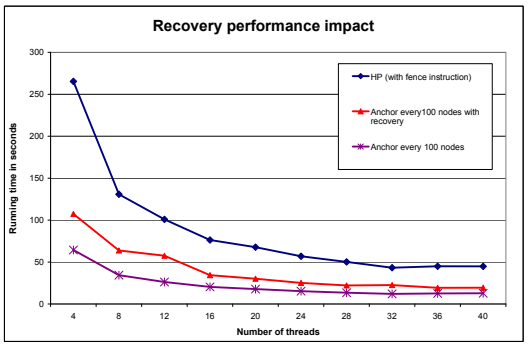
Figures 4.5 and 4.3 show the measurements of the total time required to complete our benchmark using the HP memory management, the delayed reclamation and the Drop the Anchor method. For the latter, we have used three versions with different values for the ANCHOR_THRESHOLD value. Specifically, in the first version the anchor is dropped every second node (which is the lowest legitimate value for ANCHOR_THRESHOLD in the case of the linked list), in the second version the anchor is dropped every 10 nodes, and in the third – every 100 nodes. We show the results for read-only workload where all operations are searches (Figure 4.5(a)) and for the mixed workload, where 20% of all operations are insertions, 20% are deletions, and the remaining 60% are searches (Figure 4.3(a)).

Our measurements show that in the mixed workload the Drop the Anchor-based implementation is faster in about 150–250% than the HP-based one, even if the anchor is dropped every second node. When increasing the ANCHOR_THRESHOLD parameter from 2 to 100, we get even higher improvement of 300–450% over the performance of HPs.

In read-only workload we can see even better performance improvement (400% on average) due to anchors usage compared to HP usage (cf. Figure 4.5(a)). Finally, we can see that for substantial amount of threads, the Drop the Anchor-based linked list performance is very close to the linked list implementation based on the simple delayed reclamation. This suggests that the amortized cost of the memory management in the Drop the Anchor technique is very small.



(a) Drop the Anchor vs. Hazard Pointers when 16 threads are run on lists with different initial sizes. In the Anchor-based implementation, the anchor is dropped every 10 nodes.



(b) The impact of the recovery on the performance of lists with the initial size of 100k keys.

Figure 4.4: Drop the Anchor vs. Hazard Pointers for lists with the different initial sizes and the recovery performance impact.

Additionally, Figures 4.5(b), 4.3(b) present the relative performance ratio of each memory management technique, explained above, compared to delayed reclamation. When the ratio is close to 1 it means that the memory management technique adds almost no overhead over the delayed reclamation. The HP memory management shows 400–550% slowdown, where Anchor-based implementation shows 7–10% slowdown for anchors dropped every 100 nodes, and 200–250% slowdown for anchors dropped every 2 nodes, all compared to delayed reclamation results.

In another set of experiments, we measure the impact of the initial size of the list on the performance of the HP-based and Drop the Anchor-based implementations, while the number of threads is constant (16) and the workload is mixed. The results are depicted in Figure 4.4(a)(a). It can be seen that the running time of both implementations increases linearly with the size of the list as threads need to traverse more nodes per operation on average. The slope of the HP-based implementation is much steeper, however, suggesting that the overhead introduced by fences is much more significant than the cost of the anchor management.

In Figure 4.4(b)(b) we can see the performance impact of the recovery procedure in the Drop the Anchor technique. We use the version of the technique with the ANCHOR_THRESHOLD value equals 100 for more significant impact. We explicitly delay one of the threads, thus causing this thread to be considered as stuck and recovered by other threads. The stuck thread returns to run after 2 seconds and the presented total time is measured until all threads finish their runs. The results show that the recovery procedure has 15–50% impact on the performance, even when the ANCHOR_THRESHOLD value is high. In any case, the Anchors-based implementation's performance (with the delay and recovery) is much better than the HP-based one.

4.6 Pseudo-code

In this section, we provide the implementation of the Drop the Anchor technique applied to the non-blocking linked list data structure. The implementation is provided in a pseudo-code, which follows closely the syntax of C. We start with the `find` method of the list (cf. Listing 4.2), which is used internally by all three methods of the list interface, namely `search`, `insert` and `remove`. The `find` method traverses list nodes looking for a node holding the given key. The description of this method is given in Section 4.4.2.

In Listing 4.3 we provide the details of the `setAnchor` method. As its name suggests, this method is called by a thread when it needs to update its anchor. The reader is referred to Section 4.4.2 for the description of this method. Next, in Listing 4.4 we give the implementation of the `helpRecovery` method called by a thread when it realizes that some recovery procedure is in progress.

Listing 4.5 shows the code of the `recover` method, which handles all four phases of the recovery procedure (cf. Section 4.4.4). The non-trivial phases of the recovery procedure are detailed in separate listings. Specifically, Listing 4.6 details the second phase of the recovery, where a recovering thread freezes all nodes that the thread being recovered might have a pointer to. Along with that, Listing 4.7 shows the details of the third phase of the recovery procedure, where a recovering thread replaces the frozen part of the list with the sub-list created during the second phase of the recovery.

4.6.1 Correctness argument

In this section we give the arguments and the proofs of correctness. We start by outlining the assumed memory model and defining linearization points for the modified list operations. Then we argue that any internal node deleted after the last recovery was finished (or deleted any time if no thread has been suspected being stuck) will be eventually reclaimed (we call this property *eventual conditional reclamation*). Next, we argue that our technique guarantees the safety of memory references. In other words, no thread t accesses the memory that has been reclaimed since the time t obtained a reference to it. Finally, we argue that our technique is non-blocking, meaning that whenever a thread t starts the recovery procedure, then after a finite number of t's steps either t completes the recovery, or some other thread completes an operation on the list. In addition, we show that the system-wide progress with respect to the list operations is preserved, that is after a finite number of completed recovery procedures there is at least one completed list operation.

Model and linearizability

Our model for concurrent multi-threaded computation follows the linearizability model of [31]. In particular, we assume an asynchronous shared memory system

```
1  Bool find(int key) {    // Global Node **prev, *cur, are used
2     int ckey, nodesAfterAnchor;
3
4  try_again:
5     prev = &head;  // Global head points to the first node
6     cur = *prev;    nodesAfterAnchor = 0;
7
8     while (cur != NULL) {
9        if (nodesAfterAnchor % ANCHOR_THRESHOLD == 0) {
10          if (!setAnchor(prev)) {    // save prev as anchor
11             helpRecovery();         // idle or stuck bit is set
12             goto try_again;
13          }
14       }
15       nodesAfterAnchor++;
16       Node *next = cur->next;
17
18       if (isDeleted(next)) {  // current node is deleted, try to
19          // finish deletion, but don't change frozen pointers
20          if (!CAS(prev, cur, clearDeleted(next))) {
21             if (isFrozen(*prev))   // if try failed due to freeze,
22                helpRecovery();     // first help recovery
23             goto try_again;         // retry
24          }
25          retireNode(cur);
26          cur = clearFrozen(clearDeleted(next));
27       } else {
28          ckey = getKey(cur);
29          if (ckey >= key)
30             return (ckey == key);   // compare search key
31          prev = &(cur->next);
32          cur = clearFrozen(next);
33       }
34    }
35    return FALSE;
36 }
```

Listing 4.2: List traversal with `find`. The code related to the new memory management technique is underlined.

```
1  Bool setAnchor (Node** prev) {
2     Node* newAnchor = nodePointer(prev);
3
4     // globalMMRecords is an array of
5     // GlobalMemoryManagementRecords
6     uint128_t localTsAndAnchor =
7        globalMMRecords[<my ID>].timeStampAndAnchor;
8
9     if (isSuspected(localTsAndAnchor)
10       return FALSE;
11
12    // extract the current time stamp
13    uint64_t localTs = extractTimestamp(localTsAndAnchor);
14
15    // combine the current time stamp,
16    // the new anchor and the IDLE and STUCK flags
17    uint128_t newTsAndAnchor =
18          <localTs | newAnchor | FALSE | FALSE>;
19
20    return
21    CAS(&(globalMMRecords[<my ID>].timeStampAndAnchor),
22       localTsAndAnchor, newTsAndAnchor);
23 }
```

Listing 4.3: The `setAnchor` function

```
1  void helpRecovery() {
2      uint128_t localTsAndAnchor =
3          globalMMRecords[<my ID>].timeStampAndAnchor;
4
5      if (isStuck(localTsAndAnchor)) {
6          // I was suspected to be stuck
7          recover(<my ID>, localTsAndAnchor);
8          setTimeStamp(); // Reset my timestamp
9          return;     // Continue after finishing self-recovery
10     }
11
12     // Otherwise, find who is stuck
13     // (a thread with bit IDLE reset and bit STUCK set)
14     for (int i = 0; i < threadNum; i++) {
15         uint128_t currTsAndAnchor =
16             globalMMRecords[i].timeStampAndAnchor;
17
18         if (!isIdle(currTsAndAnchor) &&
19             isStuck(currTsAndAnchor)) {
20                 recover( i, currTsAndAnchor);   // Help
21                 return;
22         }
23     }
24     return;
25 }
```

Listing 4.4: The `helpRecovery` function

where n deterministic threads communicate by executing atomic operations on some finite number of shared variables. Each thread performs a sequence of steps, where in each step the tread may perform some local computation or invoke a single atomic operation on a shared variable. The atomic operations allowed in our model are reads, writes, or compare-and-swaps (CAS). The latter receives a memory address of a shared variable v and two values, *old* and *new*. It sets the value of v to *new* only if the value of v right before CAS is applied is *old*; in this case CAS returns *true*. Otherwise, the value of v does to change and CAS returns *false*. We assume that each thread has an ID, denoted as tid, which is a value between 0 and $n-1$. In systems where tid may have values from arbitrary range, known non-blocking renaming algorithms can be applied (e.g., [1]). In addition, we assume each thread can access its tid and n.

The original implementation of all operations of the non-blocking linked list by Harris [22] is linearizable [31]. We argue that after applying Drop the Anchor memory management technique, all list operations remain linearizable. Recall that all list operations invoke the `find` method, which returns pointers to two adjacent nodes, one of which holds the value smaller than the given key. For further details, see [22]). Denote this node as *prev*. Furthermore, recall that list operations may invoke `find` several times. For instance, `insert` will invoke `find` again if the `next` pointer of *prev* has being concurrently modified (in particular, in our case, frozen). Thus, we define the linearization points for a list operation *op* with respect to the *prev* returned from the last invocation of `find` by *op*. If this *prev* node is not frozen (i.e., the freeze-bit of its `next` pointer is not set), the linearization point of *op* is exactly as in [22]. However, if this *prev* node is frozen, we set the linearization

```
1  void recover(int tID, uint128_t tsAndAnchor) {  // tID − id of the thread we are going to recover; tsAndAnchor − the recorded
2              // value of tID's timeStampAndAnchor field
3
4      // Phase 1: mark as stuck
5      if ( !DWCAS(&globalMMRecords[tID].timeStampAndAnchor, tsAndAnchor, markSuspected(tsAndAnchor) ) {
6          if ( &globalMMRecords[tID].timeStampAndAnchor != markSuspected(tsAndAnchor) ) {
7              // CAS failed and not because the same timestamp was marked stuck.
8              // So some progress was made by the thread to be recovered (tID) or other thread finished to recover it.
9              return;
10         }
11         tsAndAnchor = markSuspected(tsAndAnchor);
12     }
13
14     Node *anchor = getAnchor(tsAndAnchor);
15     if (anchor != NULL ) {
16         // Phase 2: Mark part of the list as frozen and create a copied list of not−deleted nodes
17         Node *newList = freezeNodes(tID, tsAndAnchor);
18
19         // Phase 3: Replace frozen nodes
20         if (newList != NULL) {
21             replaceFrozenNodes(tID,tsAndAnchor, newList);
22         }
23     }
24
25     // Phase 4: Set recovery timestamp and mark as recovered
26     uint128_t newTsAndAnchor = < calcMaxTimeStamp()+1 | anchor | TRUE | TRUE >;
27     // if the following double−wide CAS is not successful, some other thread succeeded to complete this phase
28     DWCAS(&globalMMRecords[tID].timeStampAndAnchor, tsAndAnchor, newTsAndAnchor);
29
30     return;
31 }
```

Listing 4.5: The `recover` function

point of *op* at the time instance defined as following. Consider the sequence of frozen nodes read by the corresponding `find` operation starting from a frozen node m and including the (frozen) node *prev* (where m and *prev* might be the same node). The linearization point of *op* is defined at the latest of the two events: (a) the corresponding `find` traversed m (i.e., read the `next` pointer of the node previous to m in the list) and (b) the latest time at which some node between (and including) m and *prev* was inserted or marked as deleted. The intuition is that when `find` returns a result from a frozen part of the list, this part no longer reflects the actual state of the list at the moment *prev* node is read. Thus, we have to linearize the corresponding operation at some earlier time instance, at which the nodes read by `find` are still consistent with the actual keys stored in the list.

Eventual conditional reclamation

Lemma 4.1. *Let T_s and T_f be the time when a thread t starts and finishes, respectively, the call to `retireNode(node)` and $T_r > T_f$ is the time when t finishes to scan its reclamation buffer. Then at least one of the following events occurs in the time interval $[T_s, T_r]$:*

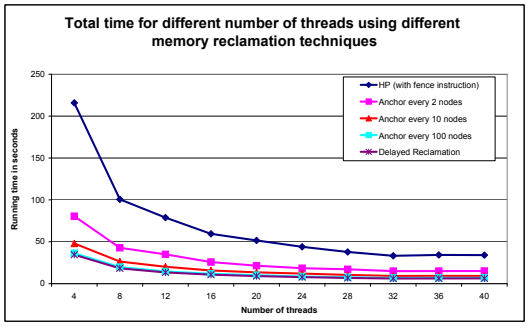
1. *Some thread remains running throughout $[T_f, T_r]$, and its timestamp changes at most once in $[T_f, T_r]$.*
2. *Some thread becomes recovered at some point in time in $[T_s, T_r]$.*

```
1  Node *freezeNodes(int tID, uint128_t tsAndAnchor) {
2     Node *localListHead = NULL;
3     int cnt = 0;
4     uint64_t maxTimeStamp = calcMaxTimeStamp();
5     uint64_t lowTimeStamp =
6           globalMMRecords[tID].lowTimeStamp;
7     Node *current = getAnchor(tsAndAnchor);
8
9     // update my anchor to enable list recovery from my
10    // failure at the time I try to recover the list from
11    // the failure of tID
12    if (<myID> != tID && !setAnchor(current)) {
13       // failure here means that the recovery I work on
14       // is done, and some other threads suspects me
15       // being failed
16       return NULL;
17    }
18
19    // iterate and freeze all the nodes, starting from the
20    // anchor until ANCHOR_THRESHOLD "old enough"
21    // nodes are found, or the end of the list is reached
22    while (cnt<ANCHOR_THRESHOLD && current!=NULL) {
23       Node *savedNext = current->next;%
24       // loop till the next pointer is frozen
25       while (!isFrozen(savedNext)) {
26          CAS(&(current->next), savedNext,
27                markFrozen(savedNext));
28          savedNext = current->next;
29       }
30
31       if (current->insertTS < lowTimeStamp) {
32          // this node is "old enough" (tID must traverse it)
33          cnt++;
34       }
35
36       if ( !isDeleted(current->next) ) {
37          // create a local copy of the frozen node
38          Node *newNode = (Node *)malloc(sizeof(Node));
39          newNode->key  =  current->key;
40          newNode->insertTS   = maxTimeStamp + 1;
41          newNode->next = clearFrozen(current->next);
42
43          <connect newNode to the local list using
44          localListHead>
45       }
46
47       // prepare to the next iteration
48       current = clearDeleted(clearFrozen(current->next));
49    } // end of while, enough entries are frozen or list ended
50
51    return localListHead;
52 }
```

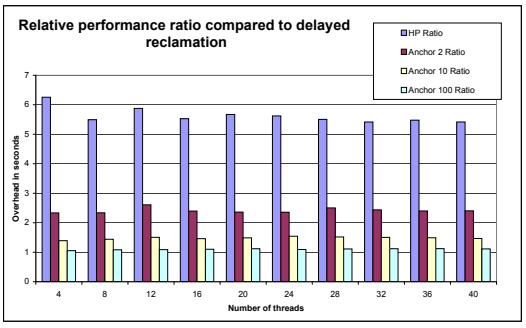
Listing 4.6: The `freezeNodes` function

```
1  void replaceFrozenNodes(int tID,
2                          uint128_t tsAndAnchor, Node *newList) {
3      while (1) {
4          Node **prev = NULL;
5          Node *curNext = NULL;
6          Node *newNext = NULL;
7
8          // Special version of find, which looks for the last not
9          // deleted node that is previous to a not deleted
10         // frozen node. It updates my anchors while running
11         // through the list.
12         if (!findInRecovery(tID, tsAndAnchor, newList,
13                 &prev, &curNext, &newNext)) {
14             // We did not find a place to insert the copied
15             // nodes, since the recovery was finished by
16             // another thread. Delete the local list of
17             // copied nodes.
18             deleteCopiedList(newList);
19             return;
20         }
21
22         if ( CAS(prev, curNext, newNext) ) {
23             // We have replaced all frozen nodes in the list.
24             return;
25         } // if CAS fails, try again
26
27     }  // end of the while
28 }
```

Listing 4.7: The `replaceFrozenNodes` function



(a) The total running time comparison for searches only.



(b) Memory management overhead referred to delayed reclamation for searches only.

Figure 4.5: Drop the Anchor vs. Hazard Pointers for lists with the initial size of 100k keys, the read-only workload results.

3. The memory allocated to `node` *is reclaimed by the time* T_r.

Sketch of proof: During the execution of `retireNode(node)`, t reads timestamps of all threads and sets the `retireTS` field of `node` to the maximal value it has read, plus one. When t runs through its reclamation buffer, it checks the reclamation condition (cf. Section 4.4.4) with respect to `n.retireTS`. If the condition holds, then t reclaims the memory allocated to `node`, satisfying the third clause of the Lemma. If the condition does not hold, then we have two cases. First, there might be a running thread t_x whose timestamp is smaller than `n.retireTS`. Assuming a timestamp does not overlap, the timestamp of any thread holds monotonically increasing values. This follows from the way the timestamp of each thread is calculated. Thus, t_x must stay running in $[T_f, T_r]$, since if it becomes idle after T_f and becomes running again, its timestamp will be equal to or larger than `n.retireTS`. From the same reason, t_x may update its timestamp in $[T_f, T_r]$ at most once (with a value which t_x started to calculate before T_f). Thus, the first clause of the Lemma holds.

The second case where the memory of n is not reclaimed by the time T_r occurs when there is some recovered thread t_x whose timestamp is larger than `n.retireTS`. But then this timestamp must be updated after T_s (since otherwise the value of `n.retireTS` would be larger) and before T_r, meaning that t_x must be recovered at some point in $[T_s, T_r]$, satisfying the second clause of the Lemma.

■

Based on the lemma above, we prove that when a thread removes a node from the list, as long as that tread keeps applying (delete) operations on the list and particularly "bad" things do not happen to other threads (e.g., they are not suspected to be failed), the memory of that node will be eventually reclaimed.

Lemma 4.2. *Let* T_s *be the time when a thread* t *starts the call to* `retireNode(node)`. *Then* `node` *will be eventually reclaimed as long as* t *keeps removing nodes from the list and there is no thread that is stuck or recovered at or after* T_s.

Sketch of proof: First, we note that just like in [41], every thread is solely responsible for reclaiming nodes in its reclamation buffer. Thus, in order for n to be eventually reclaimed, t must scan its reclamation buffer once in a while and this will happen only if t will keep removing nodes from the list.

Next, consider a node n that was removed from the list by a thread t and placed by t into its reclamation buffer at some time in $[T_s, T_f]$, as defined in Lemma 4.1. According to Lemma 4.1, there are two cases when the memory allocated to n will not be reclaimed by the time T_r. The first case occurs when some thread is recovered at some time in $[T_s, T_r]$, for which this Lemma holds. In the second case, there is a thread t_x that is in the *running* state throughout $[T_f, T_r]$. Then one of the following two events will occur with respect to t_x's execution: (a) t_x will either finish or keep restarting its current operation, promoting its timestamp before some thread will suspect that t_x might be stuck. Thus, eventually its timestamp

will become larger than $n.retireTS$ at some time instant T'. Thus, given the condition in Section 4.4.5 holds for all other threads, t will reclaim n on the first scan of its reclamation buffer started after T' (or even earlier, if T' occurs during the reclamation buffer scan by t). (b) t_x will not update its timestamp after T_r and before it gets suspected being stuck. Since we assume that t_i keeps removing nodes from the list, it will eventually suspect t_x (and recover it). Thus, the Lemma holds. ■

Note that even if some thread t_x gets stuck or recovered after T_s as above, it may have impact only on nodes being removed *before* (or concurrently to) t_x's recovery.

Safety of memory references

First, we prove that access to any node that can be reached from t's anchor (for any thread t) is safe, i.e., such node cannot be reclaimed.

Lemma 4.3. *No node reachable from an anchor of some thread can be reclaimed.*

Sketch of proof: Consider a thread t_a. Assume at some point in time t_a's timestamp is T_h and t_a's anchor refers a node N. If N is still in the list, it is not yet disconnected, thus N can not be reclaimed and it is safe to access N. So assume N is deleted, disconnected from the list and exists in the reclamation buffer of thread t_d with reclamation timestamp T_r. Note that if t_a is running or stuck $T_r > T_h$, otherwise t_a wouldn't find it. If t_a is in running or stuck state, N can not be reclaimed because there is a timestamp less than N's reclamation timestamp. If t_a is in idle state, its anchor pointer must be NULL. If t_a is in recovered state, its timestamp T_h states the time when t_a was recovered. Assuming N is deleted, the deletion must happen before the freeze of node N, because frozen node can not be marked as deleted. Therefore $T_h \geq T_r$ and N can not be reclaimed because there is a recovered timestamp larger than N's reclamation timestamp.

Let's look on a node N which is not an anchor, but reachable the t_a's anchor. Again, let's assume N is deleted, disconnected from the list and exists in the reclamation buffer of thread t_d with reclamation timestamp T_r (because the opposite case is trivial). Note that if t_a is running or stuck $T_r \geq T_h$, otherwise $T_r < T_h$, meaning that N was removed from the list before t_a has started, it contradicts the assumption that N is reachable from the anchor. Similarly to the previous case, if t_a is in recovered state, it implies that $T_h \geq T_r$ and N can not be reclaimed because there is a recovered timestamp larger than N's reclamation timestamp. ■

Using the lemma above, we show that with the Drop the Anchor memory management, no thread will access a reclaimed memory.

Lemma 4.4. *No thread t accesses the memory that has been reclaimed since the time t obtained a reference to it.*

Sketch of proof: A thread t can accesses nodes only in three occasions:

1. The nodes reachable from the list's head, as part of proceeding with specific operation
2. The nodes reachable from the thread's anchor, as part of recovery of a stuck thread
3. The frozen nodes, as part of recovery help or when a stuck thread itself wakes up

In first case, thread t can access nodes in the "live" list because t's timestamp must be greater then reclamation timestamp of any reachable node if it got disconnected from the list concurrently with t's progress. In second case, the safety of thread t accesses was proved in Lemma 4.3. Finally, we currently do not reclaim the frozen part, therefore it is always safe to access the frozen memory. ■

Progress guarantees

The original implementation of all operations of the non-blocking linked list by Harris [22] is lock-free [31] . We argue that after applying Drop the Anchor memory management technique, all list operations remain lock-free. We say that a thread t_i *starts the recovery* of a thread t_j when t_i sets the STUCK bit on in t_j's `timestampAndAnchor` field. Similarly, we say a thread t_i *completes the recovery* of a stuck thread t_j when t_i sets the IDLE bit on in t_j's `timestampAndAnchor` field.

Lemma 4.5. *If a thread t_i starts the recovery of t_j at T_s, then the recovery of t_j will be completed at $T_f > T_s$ (by possibly another thread t_k) and/or infinitely many list operations will be linearized after T_s.*

Sketch of proof: Consider the four phases of the recovery procedure (cf. Section 4.4.4). At the time T_s, t_i completes the first phase by marking t_j as *stuck*. Then, at the second phase of the recovery, t_i runs over the list starting from the node pointed by t_j's anchor and freezes each node. It was proved at [5] that freezing activity is lock free (obviously freezing can fail only if `next` pointer was modified after read and before freeze). If there are no infinitely many changes to the list (i.e., infinitely many nodes are not removed or inserted), t_j will either reach the end of the list or will traverse enough nodes to stop (for exact details, cf. Section 4.4.4). In both cases, t_i will finish the second phase in a finite number of steps. (Note that t_i may never finish this phase if the list gets changed infinitely many times, but this will satisfy the Lemma).

In the third phase of the recovery, t_i runs over the list looking for the node that precedes the frozen part of the list. Here, again, if the list is not changed infinitely many times during this phase, t_i will either find the relevant node, or reach the end of the list. Thus, after a finite number of steps, t_i will finish the third phase.

The last fourth phase consists of a CAS operation that does not require a retry in case of a failure. Thus, this phase will be completed by t_i in a constant number of steps.

To summarize, if the list does not change indefinitely many times after T_s, t_i will finish recovery of t_j after a finite number of steps at the time $T_f > T_s$. ■

Next, we show that despite recovery operations, the system-wide progress is preserved, i.e., threads never keep recovering one another forever without completing list operations.

Lemma 4.6. *Consider $n+1$ recovery operations completed at times $T_1 < T_2 < ... < T_{n+1}$. Then there must be at least one list (delete) operation linearized in the time interval $[T_1, T_{n+1}]$.*

Sketch of proof: Observe that a thread t might start a recovery of another thread only when t completes a delete operation on the list. Consider $n+1$ recovery operations completed at times $T_1 < T_2 < ... < T_{n+1}$. There must be at least one thread t_i that started at least two recovery operations at T_k and $T_m \in [T_1, T_{n+1}]$. But then based on the observation above, t_i has completed at least one delete operation between the time T_k and T_m. ■

Chapter 5

CBPQ: High Performance Lock-Free Priority Queue

5.1 Introduction

Priority queues serve as an important basic tool in algorithmic design. They are widely used in a wide variety of applications and systems such as simulation systems, job scheduling (computer systems), networking (e.g., routing and realtime bandwidth management), file compression, artificial intelligence, numerical computations, and more. With the proliferation of modern parallel platforms, the need for a high-performance concurrent implementation of the priority queue has become acute.

A priority queue supports two operations: `insert` and `deleteMin`. The abstract definition of a priority queue (PQ) provides a set of key-value pairs, where the key represents a priority. The `insert()` method inserts a new key-value pair into the set (the keys don't have to be unique), and the `deleteMin()` method removes and returns the value of the key-value pair with the lowest key (i.e., highest priority) in the set.

Lock-free (or non-blocking) algorithms [26, 30] guarantee eventual progress of at least one operation under any possible concurrent scheduling. Thus, lock-free implementations avoid dead-locks, live-locks, and priority inversions. Typically, they also demonstrate high scalability, even in the presence of high contention.

In this paper we present a design of a high performance lock-free linearizable priority queue (PQ). The design builds on a combination of two ideas. First, we use the chunk linked list [5] as the underlying data structure. This replaces the standard use of heaps, skip lists, linked lists, or combinations of them. Second, we use the fetch-and-increment ($F\&I$) instruction to obtain better performance for the `deleteMin` operation. This replaces the stronger, but less performant compare-and-swap (CAS) atomic primitive.

The `deleteMin` operation is a bottleneck for concurrent implementations of concurrent PQs, because it creates a single point of contention, and concurrent `deleteMin` invocations cause a performance degradation [35]. It is exactly at such

a point of contention that a $F\&I$ instruction is of great advantage over a use of a CAS instruction, as also noted in [17, 43].

In order to build a data structure that can use a $F\&I$ instruction, we employed the *chunk linked list* from [5], where each node (denoted *chunk*) in the list is an array of M elements. When the relevant chunk is found, an index of a location into which a new element can be inserted can be calculated concurrently using a $F\&I$ instruction. Following [5] the chunks are concurrently split or merged with other chunks to maintain predetermined size boundaries. This is done using a *freezing* mechanism, which notifies threads that old parts of the data structure (which they are currently accessing) have been replaced by new ones. We denote the obtained PQ design a *Chunk-Based Priority Queue (CBPQ)*.

Various constructions for the concurrent PQ exist in the literature. Hunt et. al. [32] used a fine-grained lock-based implementation of a concurrent heap. Dragicevic and Bauer presented a linearizable heap-based priority queue that used lock-free software transactional memory (STM) [14]. A quiescently consistent skip-list based priority queue was first proposed by Lotan and Shavit [37] using fine-grained locking, and was later made non-blocking [19]. Another skip-list based priority queue was proposed by Sundell and Tsigas [49]. While this implementation is lock-free and linearizable, it required reference counting, which compromises disjoint-access parallelism and degrades performance.

Liu and Spear [36] introduced two concurrent versions of data structure called *mounds* (one is lock-based and the other is lock-free). The mounds data structure is a rooted tree of sorted lists that relies on randomization for balance. It supports $O(\log(\log(N)))$ `insert` operations and $O(\log(N))$ `deleteMin` operations. Mounds perform well in practice (with high probability) and their `insert` operation is currently the most performant among concurrent implementations of the PQ. Recently, Linden and Jonsson [35] presented a skip-list based PQ. Deleted elements are first marked as deleted in the `deleteMin` operation. Later, they are actually disconnected from the PQ in batches when the number of nodes marked as deleted exceed a given threshold. Their construction outperforms previous algorithms by $30-80\%$. Very recently, Calciu et al. [8] introduced a new lock-based, skip-list-based PQ that uses elimination and flat combining techniques to achieve high scalability at high thread counts. We didn't manage to compare their scheme to ours in time for this submission, yet, their own measurements seem to suggest a smaller improvements over previous work than our own. In addition, their elimination mechanism is of independent interest and can be added to our mechanism to achieve even better performance.

We implemented CBPQ in C++ and compared its performance to the Linden's and Jonsson's PQ [35] and to the lock-free and the lock-based implementations of the Mounds PQ [36]. We evaluated the performance of our design using targeted micro-benchmarks: one micro-benchmarks runs only `insert` operations, a second micro-benchmark runs only `deleteMin` operations, and a third micro-benchmark runs a mix of `insert` and `deleteMin` operations, where each occurs with equal

probability. Measurements show that the CBPQ `deleteMin` operations run up to 4 times faster than deletions of the Linden's and Jonsson's PQ, which outperforms the deletions of Mound. Mounds have the fastest `insert` operations, outperforming the CBPQ `insert` by a factor of up to 2. But for a mix of `deleteMin` and `insert` operations CBPQ outperforms Mounds by a factor of up to 3, and Linden's and Jonsson's PQ by a factor of 1.5–2.

The rest of the paper is organized as follows. In Section 5.2 we provide a short overview of the CBPQ design. In Section 5.3 we present the full CBPQ algorithm. We discuss optimizations in Section 5.4 and report measurements in Section 5.5. In Section 5.6 we describe the linearization points and show lock-freedom.

5.2 A bird-Eye Overview

The CBPQ data structure is composed of a list of chunks. Each chunk has a range of keys associated with it, and all CBPQ entries with keys in this range are located in that chunk. The ranges do not intersect and the chunks are sorted by the ranges values. To improve the search of a specific range, an additional skip-list is used as a directory that allows navigating into the chunks, so inserting a key to the CBPQ is done by finding the relevant chunk using a skip-list search, and then inserting the new entry into the relevant chunk.

The first chunk is built differently from the rest of the chunks since the first chunk has the smallest keys and it therefore supports all `deleteMin` operations. We do not allow inserts to the first chunks. An insert of a key that is in the range of the first chunk goes through special handling and is discussed below. The other chunks (excluding the first) are used for insertions only.

The first chunk consists of an immutable ordered array of elements. To delete the minimum, a thread simply needs to atomically fetch and increment a shared index to the array of the first chunk. All other chunks consist of arrays with keys in the appropriate range, but are not ordered. So an insert can simply find a chunk using the skip list chunk directory, and then add the new element to the first empty slot in the array, again, simply by fetching and incrementing the index of the first available empty slot in the array.

When an insert operation finds that it needs to insert a key to the first chunk it registers this key in a special buffer. Then, a new first chunk with a new sorted array is created from the remaining keys in the first chunk, all keys registered in the buffer, and if needed, more keys from the second chunk. Typically, we let a thread wait for a limited time before making everybody cooperate on creating a new first chunk. During this limited wait, the buffer fills with more keys that justify the cost of the new first chunk construction. The creation of a new first chunk is also triggered when there are no more elements to delete in the first chunk. The creation of a new first chunk is made lock-free by letting any thread take part in the construction, and never making a thread wait for other threads to complete a task.

When an internal chunk becomes full due to insertions, it is split into two half-full chunks using the lock-free freezing mechanism of [5]. The full description of the algorithm is provided in Section 5.3 below.

5.3 The Full CBPQ Design

The CBPQ contains a set of keys (each associated with some value) that are partitioned into ranges with up to M keys in each. Each range is located in a chunk and the chunks themselves are ordered in a linked list. (In Section 5.4 below, we also add a skip-list to expedite finding a chunk with a desired range.) Each chunk has an array in which the contained keys are stored. The *first chunk* holds the range with the lowest M keys in the set, which are also ordered. From the first chunk's creation, its range is immutable, no keys can be added, and only deletions can be performed on it, by increasing the array index. The insertions are usually done on the remaining *internal* chunks, each holding an unordered set of the keys in its associated range. Insertions are done by increasing the array index and writing the key-value pair. When a key needs to be inserted into the range of the first chunk, it is instead placed in a special chunk called *buffer*. This buffer is pointed to by the first chunk and it holds all the keys that need to be inserted into the first chunk. Upon creation of a new first chunk (when it is empty or when enough keys have accumulated in the buffer), the buffer keys are inserted into the new first chunk.

5.3.1 Data Structures

Each chunk in CBPQ has a *status* word, which will be atomically updated and that signifies the chunk *state*; an array index (which we denote just *index*); and a *frozen index*. A chunk can be created in INSERT, DELETE, or BUFFER states, indicating that it was created for further insertions, for deletions, or for serving as a buffer for keys to be inserted into the first chunk, respectively. When a chunk has to be replaced by a new chunk, the old chunk enters the FREEZING state, indicating that it is in the process of being frozen. The FROZEN state indicates that the chunk is frozen and thus, obsolete. In this work we disregard memory management; the chunks are statically allocated and a frozen chunk is never reused. The frozen index is used only for freezing the first chunk, as will be explained in Section 5.3.3. In Listing 5.1, we give the `Status` and `Chunk` classes that we use. The relevant `state`, `index`, and `frIndex` fields are all held in a single machine word that is called the `Status`.

In addition to the status, each chunk consists of an array of keys (`entries` in the `Chunk` class), which holds the key-value pairs contained in the chunk. The entries are machine words, where bits can be differently divided between key and value. For simplicity, in what follows we will refer to the keys only. Each chunk has an immutable maximal value of a key that can appear on it, defined when the chunk is created (`max` in `Chunk`). A chunk holds keys less than or equal to its max

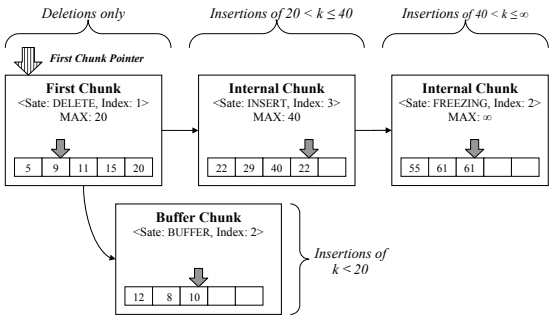


Figure 5.1: Overview of the CBPQ data structure for $N = 5$

```
1  class Status{
2      uint29_t frIndex;
3      uint3_t state;
4      uint32_t index;
5  }; // 64 bits, machine word
6
7  class Chunk{
8      Status status;
9      uint64_t entries[M];
10     uint32_t max;
11     uint64_t frozen[M/63+1];
12     Chunk *next, *buffer;
13 };
```

Listing 5.1: Status and Chunk records

value and greater than the max value of the previous chunk (if it exists). This max field is not relevant for the buffer chunk. Any chunk (except the buffer) uses a pointer to the next chunk (the `next` field). Finally, only the first chunk uses a pointer to a buffer chunk (the `buffer` field). The meaning of the `frozen` array in the `Chunk` is related to the freeze action and will be explained in Section 5.3.3. Finally, the CBPQ is a global pointer `head` to the first chunk. Fig. 5.1 illustrates the CBPQ scheme.

5.3.2 Operations Implementation

Insert: The insert pseudo-code is presented in the `insert()` method in Listing 5.2. In order to insert a key into CBPQ, we first need to find the relevant chunk – C. Because chunks are ordered in their ranges, a simple search can be used, skipping the chunks with smaller maximums (Line 5). If an insert must be performed to the first chunk, the `insert_to_buffer()` method is invoked, as explained in the next paragraph. Otherwise, C is not first. After C is found, its index is atomically increased (Line 13). The `aIncIdx()` method wraps an $F\&I$ instruction and returns the status with the new value of the chunk index. The index is increased regardless of the state of the chunk. The number of bits required to represent the chunk size (M) is much smaller than the number of bits in the index. Thus, an overflow does not happen. If C is not frozen and the increased index does not point beyond the chunk's capacity, we simply write the relevant key

to the array entry (Lines 17-21). The write is followed by a memory fence in order to ensure it will be visible to any other thread that may freeze C concurrently. As long as no freeze of the index was detected, the insert is finished. Otherwise (if the index was increased too much or a freeze was detected), the freeze is completed and C is split (Lines 23, 24), as will be explained later. After the chunks restructure, the insert is restarted.

Insert to the first chunk: The first chunk is immutable from its creation and is used only for deletions. However, it is still responsible for the range of the lowest keys. Such keys are inserted into the buffer pointed to from the first chunk. The pseudo-code of an insertion to the buffer chunk is presented in the `insert_to_buffer()` method in Listing 5.2. It starts by allocating a new buffer holding the relevant key, if needed (Line 37). The `create_buffer()` method returns *true* if the new buffer was successfully connected to the first chunk, or *false* if another thread had connected another buffer. In the latter case, a new pointer to the buffer is inserted into `curbuf`. The `create_buffer()` method's pseudo-code can be found in Appendix 5.7, Listing 5.6. Keys are inserted to the buffer in a manner similar to their insertion to the internal chunk: the index is increased and the key is placed. If this cannot be done because the buffer is full or frozen, the `insert_to_buffer()` method returns *false* (after the first chunk's freeze and recovery) to signal that the insert operation has to be retried. The insert to buffer operation cannot end until the new key is included in the first chunk and considered for deletion. So after a key is successfully inserted into a buffer, the freeze and merge of the first chunk is invoked. However, if the first chunk is already frozen, the insert to the first chunk can safely return (Line 53), because no deletion can now happen until the new key is taken into the new first chunk. After the first chunk is replaced, the insertion is considered done. The freeze and merge will be explained later.

Delete minimum: The deletion is very simple and usually very fast. It goes directly to the first chunk, which has an ordered array of minimal keys. The first chunk's index is atomically increased. Unless the need to freeze the first chunk is detected, we can just return the relevant key. The pseudo-code for the deletion operation is presented in the `deleteMin()` method in Listing 5.2.

5.3.3 Split and Merge Algorithms

Freeze: For splitting or merging chunks, a freeze is first applied on the chunks, indicating that new chunks are replacing the frozen ones. A frozen chunk is logically immutable. Then, a recovery process copies the relevant entries into new chunks that become active in the data structure. Threads that wake up after being out of the CPU for a while may discover that they are accessing a frozen chunk and they then need to take actions to move into working on the new chunks that replace the frozen ones. In [5], the freezing process of a chunk was applied by atomically setting a dedicated freeze bit in each machine word (using a CAS loop), signifying that the word is obsolete. Freezing was achieved after all words were marked in

```
1  void insert(int key) {
2      Chunk* cur = NULL, *prev = NULL;
3      while(1) {
4          // set the cur and prev pointers according to the key
5          getChunk(&cur, &prev, key);
6
7          if ( cur==head ) {   // first chunk
8              if ( insert_to_buffer(key, cur, head) ) return;
9              else continue;
10         }
11
12         // atomically increase the index in the status
13         Status s = cur->status.aIncIdx();
14         int idx = getIdx(s);
15
16         // insert into not full and non-frozen chunk
17         if ( idx<M && !s.isInFreeze() ) {
18             cur->entries[idx] = key;
19             memory_fence;
20             if ( !cur->entryFrozen(idx) ) return;
21         }
22
23         freezeChunk(cur);
24         freezeRecovery(cur, prev); //restructure the CBQP
25         // retry after restructure
26     }
27 }
28
29
30
31 bool insert_to_buffer(int key, Chunk* cur, Chunk* curhead) {
32     Chunk *curbuf = cur->buffer;
33     bool result = true;
34
35     // PHASE I: key insertion into the buffer
36     if( curbuf==NULL )  // the buffer is not yet allocated
37         if ( create_buffer(key,cur,&curbuf) )
38             goto phaseII;   // the key was added in creation
39
40     // atomically increase the index in the status
41     Status s = curbuf->status.aIncIdx();
42     int idx = getIdx(s);
43     if ( idx<M && !s.isInFreeze() ) {
44         curbuf->entries[idx] = key;
45         memory_fence;
46         if ( !curbuf->entryFrozen(idx) ) result = false;
47         else result = true;
48     } else { result = false; }
49
50     // PHASE II: first chunk is merged with the buffer
51     // before this insert ends
52 phaseII:
53     if (curhead->status.isInFreeze()) return result;
54     freezeChunk(cur);
55     freezeRecovery(cur, NULL);
56     return result;
57 }
58
59
60
61 int deleteMin() {
62     Chunk* cur, next;
63     while(1){
64         cur = head;
65         // atomically increase the index in the status
66         Status s = cur->status.aIncIdx();
67         int idx = getIdx(s);
68         // delete from not full and non-frozen chunk
69         if ( idx<M && !s.isInFreeze() )
70             return curr->entries[idx];
71
72         // First freeze, then remove frozen chunk from CBPQ
73         freezeChunk(cur);
74         freezeRecovery(cur, NULL); // retry after restructure
75     }
76 }
```

Listing 5.2: Common code path: insertion of a key and deletion of the minimum

this manner. Such a freezing process may be slow, applying a CAS instruction on each obsolete word, sometimes repeatedly. In the context of CBPQ it turns out that we can freeze a chunk in a faster way.

Assuming a 64-bit architecture, we group each 63 entries together and assign a *freeze word* of 64 bits to signify the freeze state of all 63 entries. MSB of the freeze word simply says that the word is set. Then, we associate a bit for each of the 63 entries. This bit signifies whether the key in the associated entry has been copied into the chunks that replace the current frozen chunk. For example, an insert operation may increment the array index reserving a location for the insert, but then be delayed for a long while and wake up to find that the chunk has been frozen and entries have been copied. If the entry content was not available during the copying of the old chunk to the new one, then this entry has not been copied, and the inserting thread should be able to determine that via the specific freezing bit that is associated with the frozen chunk entry.

All freeze bits are located separately from the data holding words, with a simple mapping from them to their freeze bits. In the CBPQ `Chunk` class, the data words are the M array entries that can be used to store the CBPQ keys. All freeze words are combined in the `frozen` array, which appears in Listing 5.1 as an array in the `Chunk` class.

During the freezing process, the 63 entries are read. If a zeroed entry is found, then the insert operation on it was not completed and we do not need to copy it into the new chunk. This processing happens in the `freezeKeys()` routine (Appendix 5.7, Listing 5.5). To update a freeze word, one needs to simply read the 63 entries, decide which of the entries is nullified, and create a word of bits signifying which of the entries are relevant for the copy and which are not. After creating such a freeze word, a CAS is used to try and place this word in the freeze array. Each freeze word is updated only once, due to MSB being set only once. All this allows the freezing process to be completed using fewer CAS instructions.

Following the pseudo-code in Listing 5.3, here is how we execute the freeze for a chunk C. In the first phase of the operation, we change C's status, according to the current status (Lines 6-28). Recall that the status consists of the state, the index and the frozen index. If C is not in the process of freezing or already frozen, then it should be in a BUFFER, an INSERT or a DELETE state, with a zeroed frozen index and an index indicating the current array location of activity. For insert or buffer chunks, we need only change the state to FREEZING; this is done by setting the bits using an atomic OR instruction (Line 11). The frozen index is only used for the first chunk, in order to mark the index of the last entry that was deleted before the freeze. Upon freezing, the status of the first chunk is modified to contain FREEZING as a state, the same index, and a frozen index that equals the index if the first chunk is not exhausted, or the maximum capacity if the first chunk is exhausted, i.e., all entries have been deleted before the freeze (Line 14). Let us explain the meaning of the frozen index.

As the deletion operation uses a $F\&I$ instruction, it is possible that concurrent

```
1  void freezeChunk(Chunk* c) {
2      int idx, frozenIdx = 0; Status localS; // locally copied status
3      // PHASE I : set the chunk status if needed
4      while(1){
5          // read the current status to get its state and index
6          localS = c->status; idx = localS.getIdx();
7
8          switch (localS.getState()){
9              case BUFFER : // in insert or buffer chunks
10             case INSERT : // frozenIdx was and remained 0
11                 c->status.aOr(MASK_FREEZING_STATE);
12                 break;
13             case DELETE :
14                 if (idx>N) frozenIdx=M; else frozenIdx=idx;
15                 Status newS; // set: state, index, frozen index
16                 newS.set(FREEZING, idx, frozenIdx);
17                 // CAS to the new state, it can be prevented
18                 // by a delete updating the index
19                 if ( c->status.CAS(localS, newS) ) break;
20                 else continue;
21             case FREEZING: // in process of being freezed
22                 frozenIdx = localS.frIndex; break;
23             case FROZEN :  // c was frozen by someone else
24                 c->markPtrs(); // set next chunk and buffer
25                 return;   // pointers as deleted
26         }
27         // break the loop reaching this line, continue only
28         break; // if CAS from DELETE state failed
29     }
30     // PHASE II: freeze the entries
31     freezeKeys(c, frozenIdx);
32     // move from FREEZING to FROZEN state using atomic OR
33     c->status.aOr(MASK_FROZEN_STATE);
34     c->markPtrs();  // set the chunk pointers as deleted
35 }
```

Listing 5.3: Freezing of a chunk

deletions will go on incrementing the index of the first array in spite of its status showing a frozen state. However, if a thread attempts to delete an entry from the first chunk and the status shows that this chunk has been frozen, then it will not use the obtained index. Instead, it will help the freezing process and then try again to delete the minimum entry after the freezing comlpetes. Therefore, the frozen index indicates the last index that has been properly deleted. All keys residing in locations higher than the frozen index must be copied into the newly created first chunk during the recovery of the freezing process. If all keys in the frozen first chunk have been deleted, then no key needs to be copied and we simply let the frozen index contain the maximum capacity M, indicating that all keys have been deleted from the first chunk.

In Line 19 the status is updated using a CAS to ensure that concurrent updates to the index due to concurrent deletions are not lost. If C is already in the FREEZING state because another thread has initiated the freeze, we can move directly to phase II. If C is in the FROZEN state, then the chunk is in an advanced freezing state and there is little left to do. It remains to mark the chunk pointers `buffer` and `next` so that they will not be modified after the chunk has been disconnected from CBPQ. These pointers are marked (in Line 24 or Line 34) as deleted (using the common Harris delete-bit technique [22]). At this point we can be sure that sleeping threads will not wake up and add a link to a new buffer chunk or a next chunk to C and we may return.

The second phase of the freeze assumes the frozen index and state has been properly set and it executes the setting of the words in the `frozen` array in method `freezeKeys()` (Line 31) as explained above. The pseudo-code of the `freezeKeys()` method can be found at Appendix 5.7, Listing 5.5. With the second phase done, it remains to change the state from FREEZING to FROZEN (using the atomic OR instruction in Line 33) and to mark the chunk's pointers deleted as discussed above. The atomic OR instruction is available on the x86 platform and works efficiently. However, this is not an efficiency-critical part of the execution as freezing happens infrequently, so working with a simple CAS loop to mark the pointers would be fine.

CBPQ recovery from a frozen chunk: Once the chunk is frozen, we proceed like [5] and replace the frozen chunk with one or more new chunks that hold the relevant entries of the frozen chunk. This is done in the `freezeRecovery()` method, presented in Listing 5.4. The input parameters are: `cur` – the frozen chunk that requires recovery, and `prev` – the chunk that precedes `cur` in the chunk list or NULL if `cur` is the first chunk. The `freezeRecovery()` method is never called with a `cur` chunk being the buffer chunk. The first phase determines whether we need to split or merge the frozen chunk, Lines 6-7. If `cur` is the first chunk (which serves the `deleteMin` operation), a merge has to be executed; as the first chunk gets frozen when there is need to create a new first chunk with other keys. If it is not the first chunk, then an internal chunk (which serves the `insert` operation) must have been frozen because it got full and we need to split

```
1  void freezeRecovery(Chunk* cur, Chunk* prev) {
2      bool toSplit = true; Chunk *local=NULL, *p=NULL;
3
4      while(1) {
5          // PHASE I: decide whether to split or to merge
6          if ( prev==NULL||(prev==head && prev->status.isInFreeze()) )
7              toSplit = false;
8
9          // PHASE II: in split, if prev is frozen, finish its recovery first
10         if ( toSplit && prev->status.isInFreeze() ){
11             freezeChunk(prev); // ensure prev freeze is done
12             if ( getChunk(&prev, &p) ){ // search the previous to prev
13                 // the frozen prev found in the list, p precedes prev
14                 freezeRecovery(prev, p); // invoke recursive recovery
15             }
16             // prev is already not in the list; re-search
17             // the current chunk and find its new predecessor
18             if ( !getChunk(&cur, &p) ) {
19                 return; // the frozen cur is not in the list
20             } else {prev = p; continue;}
21         }
22
23         // PHASE III: apply the decision locally
24         if (toSplit) local = split(cur);
25         else local = mergeFirstChunk(cur);
26
27         // PHASE IV: change the PQ accordingly to the
28         // decision, if decision was wrong - repeat
29         if (toSplit) {
30             if ( CAS(&prev->next, cur, local) ) return;
31         } else {
32             // When modifying the head, check if cur second or first
33             if(prev==NULL)
34                 if( CAS(&head, cur, local) ) return
35             else if( CAS(&head, prev, local) ) return;
36         }
37
38         // for the retry check for new location
39         if ( !getChunk(&cur,&p) )
40             return; // the frozen cur is not in the list
41         else prev = p;
42     }
43 }
```

Listing 5.4: CBPQ recovery from a frozen chunk

it into two chunks. There is also a corner case in which a merge of the first chunk happens concurrently with a split of the second chunk. This requires coordination that simply merges relevant values of the second chunk into the new first chunk. So if `cur` is the second chunk and the first chunk is currently freezing, then we know we should work on a merge.[1]

In order to execute the entire recovery we will need to place the new chunks in the list of chunks following the previous chunk. We therefore proceed by checking if the previous chunk is in the process of freezing and if it is, we help it finish the freezing process and recover. Namely, we freeze it, we look for its predecessor and then invoke the freeze recovery for it (Lines 10-21). This may cause recursive recovery calls until the head of the chunk list, Line 14. During this time, there is

[1] It is possible that we miss the freezing of the first chunk and start working on a split of the second chunk. In this case a later CAS instruction, in Line 30, will fail and we will repeat the recovery process with the adequate choice of a merge.

a possibility that some other thread has helped recovering our own chunk and we therefore search for it in the list, Line 18. If we can't find it, we know that we are done and can return.

In the third phase we locally create new chunks to replace the frozen one (Lines 24,25). In the case of a split, two new half-full chunks are created from a single full frozen chunk, using the `split()` method. The first chunk, with the lower-valued part of the keys, points to the second chunk, with the higher-valued part. In the case of a merge, a new first chunk is created with M ordered keys taken the from the frozen buffer and from the second chunk. This is done using the `mergeFirstChunk()` method (Appendix 5.7, Listing 5.7). If there are too many frozen keys, a new first chunk and new second chunk can be created. The new first chunk is created without pointing to a buffer. The buffer will be allocated when needed for insertion.

In phase IV, the thread attempts to attach its local list of new chunks to the chunk list. Upon success the thread can return. Otherwise, the recovery is tried again, but before that, `cur` is searched for in the chunk list. If it is not there, then other threads have completed the recovery and we can safely return. Otherwise, a predecessor has been found for `cur` in the search and the recovery is re-executed.

5.4 Optimizations

A Skip-List Optimization: The CBPQ algorithm achieves very fast deletions. Previous work required complexity O(log$n$), whereas the CBPQ deletions are executed in $O(1)$ (average) complexity, and hence CBPQ deletions are more efficient. However, as described in Section 5.3, insertions use linear search on the chunk list, which is slow compared to the logarithmic randomized search used in both the Mounds PQ and the skip-list-based PQ. Measurements showed that most of the insertion time is spent on reaching the relevant chunk. In order to improve the search time for the relevant chunk, we added a simple lock-free skip-list from [30] to index the chunks. We have made only two changes to the skip-list from [30]. We first turned it into a true dictionary, that holds a value in addition to the key. In our setting the key is the max value of a chunk and the value is a pointer to the chunk. Next, we modified it to return the value (associated with the key) as a result of a search for a key k. If k is in the set, then its associated value is returned. Otherwise, the value associated with (i.e., a pointer to the chunk whose max value is) the largest key smaller than k is returned.

Deletions are not affected by the skip-list. It is only used during insertions. When a key needs to be inserted, the relevant chunk is found using the skip-list. As the chunk list is modified by splits and merges, we modify the skip-list to reflect the changes, but we do not bother to keep a perfect synchronization between the currently available chunks and the chunks pointed by the skip-list as explained below.

Updates to the skip-list are executed as follows. After finding the appropri-

ate chunk, the `insert` operation is executed as specified in the previous section. When a chunk restructuring is needed, we start by executing a split or a merge as described in Section 5.3.3. Next, we let the (unique) thread that succeeded to insert its local new chunk into the chunk list and make it the replacement chunk (Lines 30,34 or 35 in the `freezeRecovery()` method) make the necessary updates to the skip-list. It removes the frozen, obsolete chunks from the skip-list and it adds the new chunks.

Since the skip-list is not tightly coupled with the insertions and deletions of chunks from the chunk list, it is possible that the skip-list will lead to a frozen chunk, or that it will not find a desired chunk. If, during execution of an `insert` operation, a search of the skip-list leads to a frozen chunk, a recovery of that chunk is invoked, the chunk is removed from the skip-list, and the search can be restarted. In contrast, if we can not find the chunk we need through the skip-list search, we choose a chunk that is closest to it and that precedes it in the skip-list. Next, we simply proceed by walking linearly on the actual chunk list to find the chunk we need.

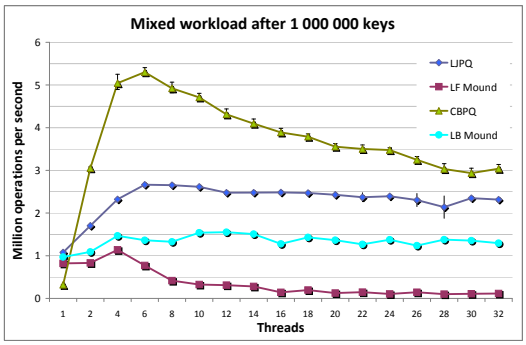
An Effective Backoff Optimization: When an `insert` operation needs to insert a key to the first chunk, it initiates a freeze of the first chunk. It starts by putting its item in a buffer, and then it checks if the freezing of the first chunk was initiated. If yes, it simply returns, knowing that the item will be inserted during the recovery from the freeze. Otherwise, it starts the freeze of the first chunk. Frequent freezing of the first chunk create a downgrade in performance. We found out that it is worth waiting a bit before executing the freeze. This lets several insert operations put their items into the buffer and complete fast, before the freezing begins. When the freezing begins, the buffer is blocked for further insertions and all current items in the buffer are processed.

We implemented this pause by adding a delay after a successful modification of the first chunk state to FREEZING (Line 19, Listing 5.3). Note that insertions to the buffer may continue, because the buffer itself becomes blocked (frozen) only after the freezing of the first chunk actually starts. Such a delay allows many fast insertions into first chunk. They insert their items into the buffer and return immediately (Line 53, Listing 5.2). A good wait-time turned out to be 5 microseconds.

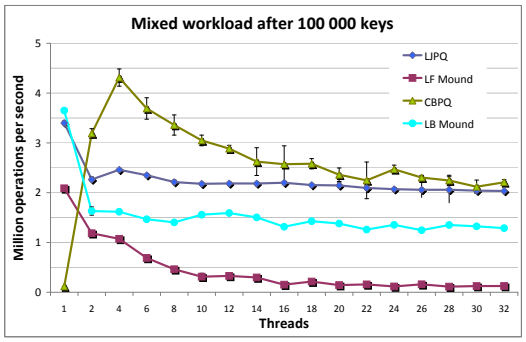
The problem of frequent insertions into the first chunk is not common with a workload executing random insertions only (with no deletes). In fact we measured a probability of 0.15% (about once every 600 inserts) for insertion to hit the first chunk. However, when we run a mix of (random) inserts and deletes, the probability of insertion to the first chunk increases significantly into 15%. The reason for this is that the deletes always remove the lowest keys, making the range of the first chunk cover the minimum possible value up to the current minimum in the PQ. Since this range increases with time and the insertions choose a uniformly distributed key, the chances of hitting the range of the first chunk increase substantially during the run.

5.5 Performance Evaluation

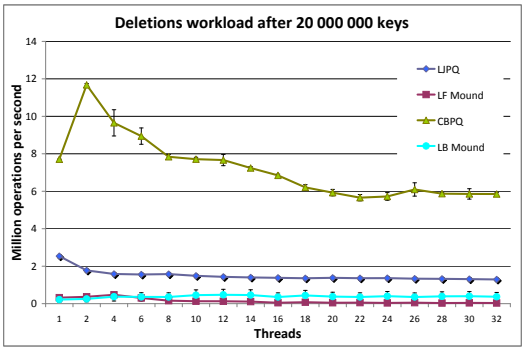
We implemented the CBPQ and compared it to the Linden's and Jonsson's PQ [35] (LJPQ) and to the lock-free and lock-based implementation of Mounds. We chose these implementations, because they are the best performing priority queues in the literature and they were compare to other PQ implementations in [35, 36]. We thank the authors of [36] and [35] for making their code available to us. All implementations were coded in C++ and compiled with a -O3 optimization level.



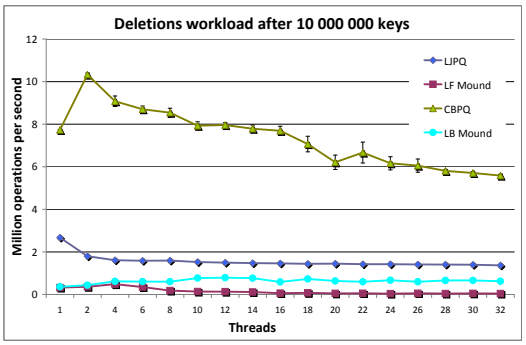
(a) Mixed throughput for a PQ with 1M keys



(b) Mixed throughput for a PQ with 100K keys

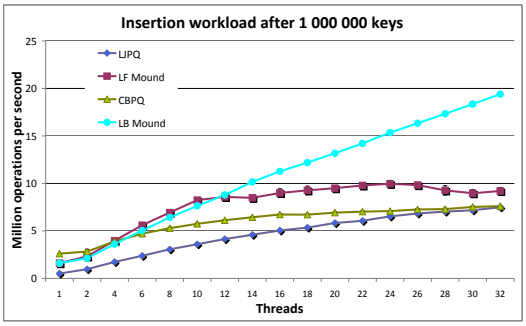


(c) Deletion throughput for a PQ with 20M keys

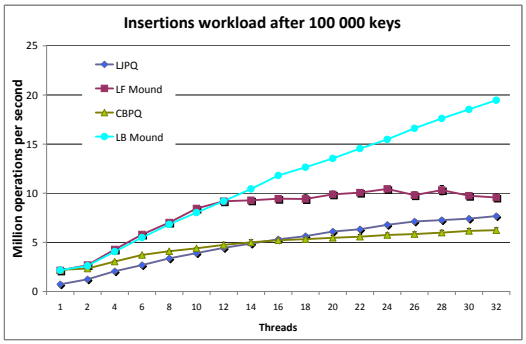


(d) Deletion throughput for a PQ with 10M keys

Figure 5.2: Throughput in delete and mixed workloads.



(a) Insert-only throughput for a PQ with 1M keys



(b) Insert-only throughput for a PQ with 100K keys

Figure 5.3: CBPQ vs. Lock-Free and Lock-Based Mound in insert workload.

We ran our experiments on a machine with two AMD Opteron(TM) 6272 16-core processors, overall 32 cores. The machine was operated by Linux OS (Ubuntu 14.04) and the number of threads was varied between 1 and 32. The chunk capacity (M) was chosen to be 928, so one chunk occupies a virtual page of size 4KB. The CBPQ implementation included the skip-list and backoff optimizations of Section 5.4. The performance was evaluated using targeted micro-benchmarks: insertion-only or deletion-only workloads and mixed workloads where deletions and insertions appear with equal probability. The keys for insertions were uniformly chosen at random among all 24-bit sized keys. We ran each test 10 times and report average results. Error bars on the graphs show 95% confidence level.

A Mixed workload: In Figure 5.2(a) we report the throughput of operations during one second, on a PQ that contains 1M entries before a measurement starts. It turns out that the CBPQ outperforms LJPQ by a factor of $1.5 - 2$. When comparing the CBPQ to the lock-based version of Mounds (which outperforms the lock-free version of Mounds), we see an improvement by a factor of 3 when many threads are active. The results of the measurements with a smaller number of keys can be seen in Figure 5.2(b) and they show that the CBPQ outperforms LJPQ by $20 - 80\%$ for shorter chunk lists. When running the CBPQ on a single thread, the obtained throughput was low. We suspect that the reason for that is that the added delay does not achieve a concurrency benefit, and only delays the execution.

Deletion-only workload: In order to make the deletion measurement relevant with deletion-only workload, we ensured that there are enough keys in the PQ initially so that deletions actually deleted a key and never operated on an empty PQ. As the CBPQ performs the deletes very fast, we upload the heap with 10M or 20M entries to ensure that all deletes operate on non-empty PQ. In a deletion-only workloads we see a huge performance improvement for the CBPQ. Results for the deletion-only workload are reported in Figure 5.2(c) for 20M and Figure 5.2(d) for 10M keys. The CBPQ deletion throughput is up to 4 times higher than LJPQ throughput, and up to 10 times higher than the lock-based Mounds throughput.

Insertion-only workload: As in the mixed workload, we start with a PQ that initially has 100K or 1M random keys in it. During the test, we let a varying number of concurrent threads run simultaneously for 1 second, and we measure the throughput. Figures 5.3(a) and 5.3(b) show the results. In both cases, the CBPQ throughput is about 3 times worse than that of Mound, for large numbers of threads. This follows from the difference in complexity. Inserts have complexity $O(\log(\log(N)))$ for the Mounds and $O(\log(N))$ for the CBPQ. Note that for smaller amount of threads, the advantage of Mounds deteriorates significantly. In spite of the advantage Mounds has with inserts, CBPQ significantly outperforms Mounds on a mixed set of operations. The CBPQ implementation outperforms LJPQ for inserts-only workloads. Note that CBPQ's skip-List contains only N/M entries (N being the number of items in the PQ), and not N like LJPQ.

5.6 Correctness

Let us first specify the CBPQ operations linearization points, and next show that CBPQ is lock-free. As a rule, CBPQ's linearization points happen only during modifications of non-frozen chunks that are either in an INSERT or a DELETE state. When an operation accesses a frozen chunk C, it recovers C, but a linearization point will not happen during the freeze or recovery.

First, we define the straightforward `deleteMin` operation's linearization point. It is set to the increase of the index on a non-frozen first chunk. In the `deleteMin()` method the linearization point is set to Line 66 (Listing 5.2), conditioned on the event that later in Line 69 the index is not detected as larger than the chunk capacity and the chunk is not detected as frozen. If the chunk is frozen, or the index is too high, then the operation will restart after the recovery and Line 66 will be executed again.

We divide the discussion for the `insert` operation's linearization point into two different cases. The first case is that the entry is inserted into the first chunk and the second case is that the entry is inserted into a non-first chunk. We start with the latter and select the linearization point of an insert to a non-first chunk to be at the write of a key-value pair into the appropriate array entry of the non-frozen chunk. In the `insert()` method this linearization point happens in Line 18, conditioned on the event that later in Line 20 the chunk is not detected as frozen.

An insertion into the first chunk is more complicated. First, the entry is put in the buffer chunk, then, the first chunk freezes, and then the buffer freezes, and during the recovery of the first chunk, the entry is added to it. We let the linearization point for this insertion be the point in which the first chunk freezes, given that the entry was successfully added to the buffer before the buffer froze. Formally, we condition on the entry being inserted into the buffer in Line 44 of the `insert_to_buffer()` method, and later in Line 46 it turns out that this entry has been properly admitted into the buffer before the buffer itself was frozen. Otherwise, if this entry was not admitted, its insertion will be retried after the first chunk (and the buffer) are recovered. Given the above condition, the linearization point is set to the time in which the first chunk's state becomes FREEZING. This happens with a successful CAS in Line 19 of the `freezeChunk()` method. If more than one insertion into the first chunk participate in the same freeze, then their linerization points happen simultaneously, as explained above, at the point in time when the first chunk became FREEZING. To set the order between these simultaneous inserts we use the increasing order of their keys. Their linearization points can be defined happening in the above mentioned order, in the first chunk freezing time, following one another with no other operation happening in between.

Lock-Freedom: In order to show that CBPQ is lock-free, we first consider each of the operations `insert` and `deleteMin` and then consider the freezing mechanism.

The `insert` operation has only two cases in which a backward branch occurs. One is in Line 9 of the `insert()` method in which the insertion to the buffer

returns failure. This can only happen if the buffer is frozen. The second case is when reaching Line 23, meaning that we fail to insert a key to a non-first chunk, because it is frozen. The `deleteMin` operation needs to restart only when reaching Line 73 of the `deleteMin()` method, where it discovers that the first chunk is under freeze.

We note that in each of the above cases, we find a chunk that needs to be frozen and we then invoke the `freezeChunk()` and the `freezeRecovery()` methods in order to replace the chunk with a new one. We will later show that the freezing mechanism guarantees progress, and we assume it now.

We start with a general claim saying that during the lifetime of a chunk some progress must be made.

Claim 5.1. *For any chunk C, during the time interval that starts with inserting C into the chunk list and ends freezing C progress must occur by some thread in the system.*

Proof. C's initial states can be one of INSERT, DELETE, and BUFFER. If C is created in the INSERT state, which means that C is not first chunk, then $N/2$ insertions to C must complete before triggering C's freeze. If C is created in the DELETE state, then either N deletions from C complete before C needs a freeze or at least one key is inserted into the associated buffer. In the latter case, when the state of C changes to the FREEZING state, a linearization point for an insert to the first chunk occurs, as specified above. Finally, if C is created in the BUFFER state, we get a very similar behavior. C can be frozen only after at least one key has been inserted into it. In addition, the buffer freeze happens only after first chunk freezes as part of the `mergeFirstChunk()` method and again, the first chunk's change to the FREEZING state is the linearization point for an insert into the first chunk, which means progress. □

Recall that, the `insert` and `deleteMin` operation follow the backward branch only when a freeze is encountered. Let C be the chunk that was frozen and into which we wanted to insert the entry, or from which we wanted to delete an entry. Let C' be the new chunk that replaced it during the execution of the recovery. Therefore, the `insert` or `deleteMin` operations started on C will retry on C'. According to Claim 5.1 another retry can happen only if some progress occurred on C' by some thread in the system.

We now move to proving that the freezing mechanism maintains lock-freedom. From code inspection of the freezing process, the only backward jumps (that may hinder progress) appear in two places: (1) when the first chunk's state is changed from DELETE to FREEZING in Line 19 of Listing 5.3 (CAS failure, because the expected status word has been changed) and (2) when an insertion of a new chunk to the chunk list (as part of recovery) fails since the expected previous chunk pointer has been concurrently modified. See Lines 30, 34 and 35 in Listing 5.4. In addition, there is also a recursion where `freezeRecovery()` calls itself in Line 14. The recursion can not lead to infinite help without progress, because the recursion

depth is bounded by the chunk list size. The list size growth also implies a progress of some insertions. In addition, because of the Claim 5.1, new chunks can not be repeatedly added to and removed from the list (causing infinite help loop) without some threads having a progress. All the above implies that new chunks can not repeatedly be added to the list without a progress.

Let's start with the first backward jump. Here, the status word has been modified by a concurrently operating thread. This word contains three fields: the state, the index, and the frozen index. A DELETE state can only change to a FREEZING or a FROZEN state which means an advance in the freezing process and then failing of the CAS cannot happen again in this loop due to a state change. If the index has been modified concurrently, and then a `deleteMin` operation has been linearized concurrently, which implies progress. Finally, the frozen index can only change simultaneously with the state change and this case is similar to the change of the state discussed first.

Moving to the second case, we know that the next pointer of the previous chunk has been modified concurrently. This can happen if a new chunk has been added to the chunk list after the previous one or the previous chunk has been frozen. The first can only happen if another thread has added a new chunk to replace the one that we are currently trying to recover. This means that the recovery is done, and then searching for this chunk in Line 39 of Listing 5.4 will not find the chunk and hence return from this function. Return from the `freezeRecovery()` method is the end of the freeze activity and thus, a progress of a freeze. The second (failure to insert a new chunk because the previous chunk is frozen) means that the previous chunk is going to be recovered and replaced by a new chunk. After that, either the insertion CAS succeeds, or a newer chunk was inserted again, and thus, according to the Claim 5.1, before the previous chunk is frozen again, progress must occur in the system.

5.7 Pseudo-code

In this section, we provide the remaining CBPQ implementation details. The main interfaces as `insert()` and `delMin()` methods were presented in Listing 5.2; here we present the rest of the code. The implementation is provided in a pseudo-code, which follows closely the syntax of C. We start with the `freezeKeys()` routine (cf. Listing 5.5), which is used internally by `freezeChunk()`. To update a freeze word, one needs to simply read the 63 entries, decide which of the entries is nullified, and create a word of bits signifying which of the entries are relevant for the copy and which are not. After creating such a freeze word, a CAS is used to try and place this word in the freeze array.

We continue to the `create_buffer()` method (cf. Listing 5.6), which is used in the `insert_to_buffer()` method and which returns *true* if the new buffer (with the given key) was successfully connected to the first chunk, or *false* if another thread had connected another buffer. In any case, at the end, the output parameter

```
1  int freezeKeys(Chunk chunk, int frIdx) {
2
3      // go over parts of the entries which are held by one freeze word
4      for(int k=0; k<PARTS_IN_CHUNK-1; ++k){
5          uint64_t frWord=INIT_FR_WD;  // init to 0s with MSB set to 1
6          uint64_t mask = 1;
7          // prepare a mask for one frozen word; if frIdx is higher than
8          // VALS_PER_WORD, then the entire part of the entries is skipped
9          for(int i=frIdx; i<VALS_PER_WORD; ++i, mask<<=1){
10             int curval = curr->vals[i+k*VALS_PER_WORD];    // read the entry
11             if(curval != EMPTY_ENTRY)                    // EMPTY_ENTRY==0
12                 // the value exists, mark the relevant entry (in local variable)
13                 frWord |= mask;
14         }
15
16         // put local variable into the chunk. After this cas, surely MSB is set
17         atomicCAS(&chunk->meta.frozen[k], 0, frWord);
18
19         // compute frIdx with respect to the next loop, which looks at entries
20         // k*VALS_PER_WORD... (k+1)*VALS_PER_WORD-1 as k grows
21         // by one in each iteration, frIdx decreases by VALS_PER_WORD
22         frIdx -= VALS_PER_WORD;
23         frIdx = (frIndex<0)?0:frIdx;
24     }
25 }
```

Listing 5.5: Freezing the entries

```
1  bool create_buffer(int key, Chunk chunk, Chunk* curbuf) {
2      Chunk *buf = alloc();
3      buf->vals[0] = key; // buffer is created with the intended value
4      bool result = CAS(&chunk->buffer,NULL,buf);
5      *curbuf = buf; // update the buffer pointer (ours or someone's else)
6      return result;
7  }
```

Listing 5.6: Creating the buffer chunk

`curbuf` holds the pointer to the buffer created by this or other thread.

Finally, in Listing 5.7 the `mergeFirstChunk()` method is presented. This method creates a new first chunk with frozen entries of the old first chunk, buffer, and other chunks if needed. Two arrays for keys are allocated locally, array `keysC` holds the entries to be copied and array `keysN` is used for attaching additional entries from other frozen chunks. Once enough entries are accumulated in array `keysC` it is passed to the `initFirstFromArray()` method, which sorts the array and creates a new first chunk (with second if needed). The new first chunk (or its new descendant) points back to the list, to the first not frozen chunk. Pointer to the new first chunk is returned.

```
1  Chunk* mergeFirstChunk(Chunk cur) {
2      int keysC[2*M]={0}, int keysN[M]={0};
3      int lenC=0, lenN=0, frEntNum=0;
4      Chunk* nextNF = NULL;   // next not frozen chunk
5
6      fillFromFrozen(cur, keyC, &lenC);
7
8      if(ifBufferExists(cur)) {   // freeze buffer chunk
9          freezeChunk(cur->buffer);
10         fillFromFrozen(cur->buffer, keyN, &lenN);
11     }
12
13     mergeArray(keyC, lenC, keyN, lenN);
14     lenC+=lenN;
15     nextNF = cur->next;
16
17     while(lenC < M){ // not enough entries, freeze 2nd chunk
18         if (!cur->next) break;
19         freezeChunk(cur->next);
20         fillFromFrozen(cur->next, keyN, &lenN);
21
22         mergeArray(keyC, lenC, keyB, lenB);
23         lenC+=lenN;
24         cur = cur->next;
25         nextNF = cur->next;
26     }
27
28     if (lenC == 0) // no more entries remain in the PQ
29         return initializePQ{};
30
31     return initFirstFromArray(keyC, lenC, nextNF);
32 }
```

Listing 5.7: Creating the new first chunk

Chapter 6

Discussion and Conclusions

The goal of this dissertation is to advance the state of the art for concurrent data structures in three dimensions. The first goal is to design lock-free algorithms for involved data structures to which lock-free algorithms did not previously exist, such as a B-tree. The second goal is to improve system support for lock-free data structures and especially memory management support. Finally, the third goal is to improve performance for important basic data structures such as the linked list and the priority queue.

Interestingly, two basic technical constructions turned out useful for several higher level algorithms. First, the lock-free chunk mechanism, that is able to merge and split, has been used with the linked-list, the B-tree, and the priority queue. Second, the freezing mechanism, that lets new constructs replace obsolete ones during the execution, was used with the linked-list, the B-tree, the priority queue, and the lock-free memory management support.

First, in this dissertation, we have presented a chunking and freezing mechanisms that build a cache-conscious lock-free linked list. Our list consists of chunks, each containing consecutive list entries. Thus, a traversal of the list stays mostly within a chunk's boundary (a virtual page or a cache line), and therefore, the traversal enjoys a reduced number of page faults (or cache misses) compared to a traversal of randomly allocated nodes, each containing a single entry. Maintaining a linked list in chunks is often used in practice (e.g., [20]) but a lock-free implementation of a cache-conscious linked list has not been available heretofore. The building blocks of this list, i.e., the chunks and the freeze operation, are used for building additional data structures in this dissertation.

Second, we presented a lock-free dynamic B^+tree, which builds on CAS synchronization. The construction is composed of a chunk mechanism that provides the low-level node implementation, including splitting and joining a node, and then a higher level mechanism which handles the operations at the tree level. The two mechanisms and their interface are lock-free. To the best of our knowledge, this is the first design of a lock-free balanced search tree for a general platform. Results indicate better handling of contention and higher scalability when compared to the lock-based version of the B^+tree. We have also proven the correctness (with

respect to linearizability) of the algorithm and its and lock-freedom property.

Third, we presented a new method for memory management of non-blocking data structures called *Drop the Anchor*. Drop the Anchor is a novel combination of the time-stamping method (which cannot handle thread failures) with the anchors and freezing techniques that provide a fallback allowing reclamation even when threads fail. Non-blocking algorithms must be robust to thread failures and so coping with thread failures in the memory manager is crucial. We have applied Drop the Anchor for the common non-blocking linked list implementation and compared it with the standard Hazard Pointers method. Measurements show that Drop the Anchor drastically reduces the memory management overhead, while robustly reclaiming objects in all executions.

We believe our technique can be applied for other non-blocking data structures. Specifically, assume a data structure represented by a directed graph, where vertices correspond to internal nodes and edges correspond to pointers between these nodes. When recovering a thread t, we need to freeze and copy the sub-graph containing all internal nodes at the distance that depends on the ANCHOR_THRESHOLD parameter, from the node pointed by t's anchor. Essentially, although the copying operation might be expensive and even involve the whole data structure, the scalability bottleneck associated with the memory fences will be removed from the common node access step.

Finally, we presented a novel concurrent, linearizable, and lock-free design of the priority queue data structure, called CBPQ. CBPQ cleverly combines the chunk linked list and the performance advantage of the $F\&I$ atomic instruction. We implemented CBPQ and measured its performance against high performance skip-list based PQ [35] and the Mounds [36] (lock-free and lock-based), which are the best performing priority queues available. Measurements with a mixed set of insert and delete operations show that CBPQ outperforms Mounds by a factor of 3 and high performance, skip-list based PQ by a factor of 2.

References

[1] H. Attiya and A. Fouren. Adaptive and efficient algorithms for lattice agreement and renaming. *SIAM J. Comput.*, 31(2):642–664, 2001.

[2] R. Bayer and M. Schkolnick. Concurrency of operations on b-trees. *Acta Informatica*, 1977.

[3] M. A. Bender, J. T. Fineman, S. Gilbert, and B. C. Kuszmaul. Concurrent cache-oblivious b-trees. *PODC*, 2005.

[4] A. Braginsky, A. Kogan, and E. Petrank. Drop the anchor: Lightweight memory management for non-blocking data structures. In *Proc. SPAA*, 2013.

[5] A. Braginsky and E. Petrank. Locality-conscious lock-free linked lists. In *Proc. ICDCN*, 2011.

[6] A. Braginsky and E. Petrank. Lock-free b+tree. In *Proc. SPAA*, 2012.

[7] T. Brown, F. Ellen, and E. Ruppert. A general technique for non-blocking trees. In *PPOPP*, 2014.

[8] I. Calciu, H. Mendes, and M. Herlihy. The adaptive priority queue with elimination and combining. *DISC*, 2014.

[9] S. Chen, P. B. Gibbons, T. C. Mowry, and G. Valentin. Fractal prefetching b+-trees: Optimizing both cache and disk performance. *SIGMOD*, 2002.

[10] D. Comer. The ubiquitous b-tree. 1979.

[11] D. Detlefs, C. H. Flood, A. Garthwaite, P. A. Martin, N. Shavit, and G. L. Steele. Even better dcas-based concurrent deques. *DISC*, 2000.

[12] D. Detlefs, P. A. Martin, M. Moir, and G. L. Steele. Lock-free reference counting. *Distributed Computing*, 15(4):255–271, 2002.

[13] D. Drachsler, M. Vechev, and E. Yahav. Practical concurrent binary search trees via logical ordering. *PPOPP*, 2014.

[14] K. Dragicevic and D. Bauer. Optimization techniques for concurrent stm-based implementations: A concurrent binary heap as a case study. In *Proc. IPDPS*, 2009.

[15] A. Dragojevic, M. Herlihy, Y. Lev, and M. Moir. On the power of hardware transactional memory to simplify memory management. In *Proc. ACM PODC*, pages 99–108, 2011.

[16] F. Ellen, P. Fatourou, E. Ruppert, and F. Breugel. Non-blocking binary search tree. *PODC*, 2010.

[17] P. Fatourou and N. D. Kallimanis. A highly-efficient wait-free universal construction. *SPAA*, 2011.

[18] M. Fomitchev and E. Rupert. Lock-free linked lists and skip lists. In *PODC*, 2004.

[19] K. Fraser. Practical lock-freedom. In *PhD dissertation, University of Cambridge*, 2004.

[20] L. Frias, J. Petit, and S. Roura. Lists revisited: Cache-conscious stl lists. *J. Exp. Algorithmics 14*, 2009.

[21] A. Gidenstam, M. Papatriantafilou, H. Sundell, and P. Tsigas. Efficient and reliable lock-free memory reclamation based on reference counting. *IEEE Transactions on Parallel and Distributed Systems*, 20(8):1173–1187, 2009.

[22] T. L. Harris. A pragmatic implementation of non-blocking linked-lists. In *PODC*, pages 300–314, 2001.

[23] T. L. Harris, K. Fraser, and I. Pratt. A practical multi-word compare-and-swap operation. In *DISC*, 2002.

[24] T. E. Hart, P. E. McKenney, A. D. Brown, and J. Walpole. Performance of memory reclamation for lockless synchronization. *Journal of Parallel and Distributed Computing*, 67(12):1270–1285, 2007.

[25] M. Herlihy. Impossibility and universality results for walt-free synchronization. 1988.

[26] M. Herlihy. Wait-free synchronization. *ACM Trans. Program. Lang. Syst.*, 13(1):124–149, 1991.

[27] M. Herlihy. A methodology for implementing highly concurrent data objects. *ACM Trans. Program. Lang. Syst.*, 1993.

[28] M. Herlihy, V. Luchangco, P. Martin, and M. Moir. Nonblocking memory management support for dynamic-sized data structures. *ACM Trans. Comput. Syst.*, 23(2):146–196, 2005.

[29] M. Herlihy, V. Luchangco, and M. Moir. The repeat offender problem: A mechanism for supporting dynamic-sized, lock-free data structures. 2002.

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

[31] M. P. Herlihy and J. M. Wing. Linearizability: a correctness condition for concurrent objects. *ACM Trans. Program. Lang. Syst.*, 12(3):463–492, 1990.

[32] G. Hunt, M. Michael, S. Parthasarathy, and M. Scott. An efficient algorithm for concurrent priority queue heaps. In *Information Processing Letters*, 1996.

[33] IBM and I. System/370. Extended architecture, principles of operation. *Publication no. SA22-7085*, 1983.

[34] A. Kogan and E. Petrank. A methodology for creating fast wait-free data structures. In *PPOPP*, 2012.

[35] J. Linden and B. Jonsson. A skiplist-based concurrent priority queue with minimal memory contention. In *OPODIS*, pages 206–220, 2013.

[36] Y. Liu and M. Spear. Mounds: Array-based concurrent priority queues. In *Proc. ICPP*, 2012.

[37] I. Lotan and N. Shavit. Skiplist-based concurrent priority queues. In *Proc. IPDPS*, 2000.

[38] P. E. McKenney and J. D. Slingwine. Read-copy update: Using execution history to solve concurrency problems. In *Proc. Parallel and Distributed Computing and Systems*, pages 509–518, 1998.

[39] M. Michael. High performance dynamic lock-free hash tables and list-based sets. *SPAA*, 2002.

[40] M. Michael and M. L. Scott. Correction of a memory management method for lock-free data structures. *Technical Report TR599; CS Dept. Univ. of Rochester*, 1995.

[41] M. M. Michael. Hazard pointers: Safe memory reclamation for lock-free objects. *IEEE Transactions on Parallel and Distributed Systems*, 15:491–504, 2004.

[42] M. M. Michael. Practical lock-free and wait-free ll/sc/vl implementations using 64-bit cas. *ICDCS*, 2004.

[43] A. Morrison and Y. Afek. Fast concurrent queues for x86 processors. In *Proc. PPOPP*, 2013.

[44] A. Natarajan and N. Mittal. Fast concurrent lock-free binary search trees. *PPOPP*, 2014.

[45] R. Oshman and N. Shavit. The skiptrie: low-depth concurrent search without rebalancing. *PODC*, 2013.

[46] J. Rao and K. A. Ross. Cache conscious indexing for decision-support in main memory. 1999.

[47] O. Rodeh. B-trees, shadowing, and clones. 2008.

[48] H. Sundell. Wait-free reference counting and memory management. In *Proc. IEEE IPDPS*, 2005.

[49] H. Sundell and P. Tsigas. Fast and lock-free concurrent priority queues for multi-thread systems. In *Journal of Parallel and Distributed Computing*, 2005.

[50] S. Timnat, A. Braginsky, A. Kogan, and E. Petrank. Wait-free linked lists. *OPODIS*, 2012.

[51] S. Timnat and E. Petrank. A practical wait-free simulation for lock-free data structures. In *PPOPP*, 2014.

[52] J. D. Valois. Lock-free linked lists using compare-and-swap. In *Proc. ACM PODC*, pages 214–222, 1995.

מאוזן ,וחסר-חסמים שמשתמש בפקודת החומרה CAS הסטנדרטית. הצמתים בעץ B+ מתפצלים ומתאחדים לצורך שמירת על האיזון של העץ. לכן טבעי היה להשתמש בגושים (שתוארו לעיל) לצורך מימוש הצמתים חסרי-חסימות.

המטרה השלישית והאחרונה של תזה זו היא לתמוך בניהול הזיכרון יעיל עבור מבנים חסרי-חסימות. ניהול זיכרון יעיל של מבנים חסרי-חסימות דינמיים הוא שאלה פתוחה וחשובה. השיטות הקיימות לניהול זיכרון חסר-חסימות מקריבות את היכולת לשחרר אובייקטים או מפחיתות את הביצועים באופן משמעותי. בעבודה זו, אנו מציגים שיטה חדשנית הנקראת "Drop the Anchor" המאפשרת להפחית את התקורה הקשורה לניהול הזיכרון באופן משמעותי, ותומכת בשחרור אובייקטים אפילו בנוכחות החוטים שנכשלים ומפסיקים לרוץ. אנו מדגימים את השיטה הזו לניהול זיכרון על מבנה הנתונים הפופולארי של הרשימה המקושרת. תוך הערכת ביצועים מקיפה, אנו מראים כי "Drop the Anchor" משפר באופן משמעותי את הביצועים לאומת השיטה המקובלת היום שהיא שימוש ב-"Hazard Pointers". הנכונות של השיטה שלנו הוכחה באופן פורמאלי.

כל האלגוריתמים הנ"ל הוערכו באופן אמפירי ומספקים תפוקה גבוהה וביצועים יעילים. כולם מנצלים את הטכניקה החדשה שלנו לתיאום חוטים למקרה שחוטים צריכים להיות מנותבים מחלק מיושן של מבנה הנתונים לחלק חדש. אנו קוראים לטכניקה זו - טכניקת ההקפאה (freezing). טכניקת ההקפאה תומכת בארגון מחדש של מבני נתונים חסרי-חסימות, והיא מתמקדת באיך להודיע לחוטים לעבור לחלק אחר של מבנה הנתונים, בדרך כלל בגלל שהחלק בו הם משתמשים כעת הוא מיושן. הדבר נעשה על ידי הגדרת סיבית-הקפאה (freeze-bit) מיוחדת על כל המילים או המצביעים בחלק המיושן, מה שהופך אותם ללא מתאימים לעדכון. חוט שנכשל בניסיונו להשתמש במצביע או נתון קפוא מבין שחלק זה של מבנה הנתונים הוא מיושן ומפעיל מחדש את פעולתו. לשיפור הביצועים באלגוריתם של תור עדיפויות שיפרנו את טכניקת ההקפאה ושם אנו אוספים את כל סיביות-ההקפאה למילים-ההקפאה (freeze-words) הנפרדות, כך אנו מקטינים את כמות הפקודות האטומיות הדרושות להקפאה.

מנסים ליישם במקביל פעולה כלשהי על המבנה, מובטח כי אחד החוטים בסופו של דבר מתקדם. כתוצאה מכך, פרוטוקול הסנכרון הפופולארי של מניעה הדדית אינו ניתן לשימוש במבני נתונים חסרי-חסימות.

לעבודת דוקטורט זו שלוש מטרות. המטרה הראשונה, היא לעצב מבני נתונים מקביליים וחסרי-חסימות בעלי ביצועים גבוהים במיוחד. אנו שואפים לכך שמבני נתונים חסרי-חסימות יהיו הבחירה העיקרית למבנה נתונים המקביליים. לשם כך, דבר ראשון, אנו מציגים אלגוריתם בעל ביצועים גבוהים לרשימה מקושרת חסרת-חסימות. אנו מרחיבים את האלגוריתם לרשימה מקושרת חסרת-חסימות הקיים בספרות, על ידי בניית רשימות מקושרות עם התייחסות מיוחדת לעקרון המקומיות (locality). הרשימה המקושרת שלנו בנויה מרצפים של ערכים שנמצאים בגושים רצופים של זיכרון (chunks). כאשר עוברים על רשימות כאלה, הערכים הקרובים בערכם בדרך כלל נמצאים באותו גוש ולכן הם קרובים זה לזה גם במיקומם בזיכרון, לדוגמא, נמצאים באותה שורת המטמון או על אותו דף זיכרון הווירטואלי. יישומים לרשימות מקושרות המתחשבים במטמון מופיעים לעתים קרובות בפרקטיקה, אבל הפיכתם לחסרי-חסימות דורש טיפול קפדני. המרכיב הבסיסי של בנייה זו הוא גוש של ערכים ברשימה ששומר על מספר מינימאלי ומספר מרבי של ערכים. הגושים הללו ניתנים לפיצול ואיחוד בדרך חסרת-חסימות. המרכיב הבסיסי הזה הוא כלי מעניין בפני עצמו, ששימש בהמשך העבודה לבניית מבני נתונים חסרי-חסימות אחרים שאנו מציגים.

מבנה נתונים אחר, גם כן חסר-חסימות ובעל ביצועים גבוהים, המוצג בתזה זו הוא תור עדיפויות. תור עדיפויות הוא מרכיב אלגוריתמי חשוב המופיע בהרבה מערכות תוכנה. עם הפריסה המהירה של החומרה המקבילית, גרסאות מקביליות של תורי עדיפויות הופכות לחשובות יותר ויותר. בתזה זו, אנו מציגים אלגוריתם חסר-חסימות חדש לתורי עדיפויות שביצועיו גוברים באופן משמעותי על הביצועים של התורים הידועים האחרים. האלגוריתם שלנו מתבסס על שימוש בגושים (שתוארו לעיל) ועל שימוש בפקודת החומרה האטומית fetch-and-increment היעילה. פקודת fetch-and-increment עדיפה על פקודת compare-and-swap (CAS) לשימוש בטיפול במצבים בהם הרבה חוטים צריכים לעדכן את אותו מקום בזיכרון המשותף, כמו למשל הוצאת איבר העדיף ביותר מתור עדיפויות. מהשוואת ביצועים אפשר לראות שיפור בפקטור של עד פי שתיים, על פני גישות הקיימות אחרות לתורי עדיפויות מקביליים.

המטרה השנייה של העבודה הזו, היא להרחיב את הכיסוי של גרסאות חסרי-חסימות הקיימות למבני נתונים שעדיין אין להם גרסא כזו. ההרחבה כזו מועילה משום שמבני נתונים חסרי-חסימות מספקים הבטחת התקדמות (scalability) וידועים ביכולתם להתרחב ולהימנע מקיפאון ומחוסר-ההתקדמות (livelock), ומתן היענות מובטחת. בעבודת דוקטורט זו אנו מציגים עיצוב לעץ מאוזן חסר-חסימות שהוא עץ B+. עץ B+ מיועד לעבודה יעילה במערכות שקוראות וכותבות בלוקים גדולים של מידע, השימוש בו שכיח במסדי נתונים ובמערכות קבצים. למיטב ידיעתנו, זה המבנה הראשון של עץ דינמי,

תקציר

טכנולוגיית יצור המעבדים התקדמה לנקודה שבה מהירות השעון של מעבד, או תדירותו, כבר אינן יכולות להשתפר. עליה במהירות השעון ובתדירותו של המעבד, שבעבר אפשרו עליה המתמדת של התפוקה החישובית של המעבד, כעת הם נשמרים בקצב קבוע. במקום להגדיל את מהירות השעון של מעבד, ספקי המעבדים שינו את המיקוד ומספקים כיום כמה יחידות חישוב הנקראות ליבות, כחלק מאותו המעבד. מעבדים מרובי ליבות מסתמכים על הרעיון של זיכרון המשותף, שבו לכל ליבה יש אפשרות גישה לקריאה ולכתיבה לחלק מהמחשב הנקרא הזיכרון הראשי.

שינוי זה בטכנולוגית המעבד הביא לשפע של מחקר חדשני ומקורי בעשורים האחרונים. כעת ניתן לראות את מבני הנתונים מזווית שונה, כמרחב לתקשורת לליבות שונות, המאפשרת לקרוא ולעדכן נתונים בו-זמנית. הכוח המניע מאחורי המיקבול הוא השאיפה לשיפור ביצועים. כל יחידת חישוב נפרדת נקראת חוט. המטרה היא שכמה שיותר חוטים יוכלו לגשת לנתונים במקביל ולבצע את הפעולות שלהם בו-זמנית. עם זאת, העדכונים של החוטים השונים צריכים להיות מסונכרנים על מנת לשמור על תקינות מבנה הנתונים.

הדרך הפופולארית לסנכרן החוטים היא להשתמש במניעה הדדית, המאפשרת רק לחוט אחד לגשת למבנה נתונים (או לחלק ממנו). חוטים אחרים, הזקוקים לאותם נתונים, מעוכבים עד שהחוט הראשון מסיים. שיטה זו מפשטת את תיכנות, אבל יש לה חסרונות רבים, בעיקר בגלל שהחוט הראשון מעכב גישות של כל שאר החוטים הזקוקים לאותם נתונים. עבודת דוקטורט זו מתמקדת במבנים חסרי-חסימות, שבהם אין חוט יכול לעכב את התקדמותם של אחרים.

מעבדים מרובי ליבות הינם הבחירה הנוכחית לשיפור הביצועים. חוטים מתקשרים ומסתנכרנים באמצעות הזיכרון המשותף. בדרך כלל, התקשורת והסנכרון מתבצעים באמצעות מבני נתונים מקביליים. היעילות של מבני נתונים אלה היא קריטית לביצועים. יתר על כן, אתגרים חדשים מתעוררים בעיצוב מבני נתונים מקביליים הניתנים להרחבה שיכולים לעבוד גם עם מספר הולך וגדל של חוטים במקביל (scalable). מבני נתונים מקביליים חסרי-חסימות הם ניתנים להרחבה וגם מספקים הבטחת התקדמות. אם כמה חוטים

המחקר נעשה בהנחיית פרופסור ארז פטרנק

בפקולטה למדעי המחשב

אני מודה לטכניון ולמשרד המדע על התמיכה הכספית הנדיבה בהשתלמותי.

שיטות לתיאום בין חוטים מרובים לצורך בנית מבני נתונים חסרי-חסימות

חיבור על מחקר

לשם מילוי חלקי של הדרישות לקבלת תואר
דוקטור לפילוסופיה

אנסטסיה ברגינסקי

הוגש לסנט הטכניון – מכון טכנולוגי לישראל

אייר תשע״ה חיפה מאי 2015

שיטות לתיאום בין חוטים מרובים לצורך בנית מבני נתונים חסרי-חסימות

אנסטסיה ברגינסקי