Practical Parallel Data Structures

Shahar Timnat

Practical Parallel Data Structures

Research Thesis

Submitted in partial fulfillment of the requirements
for the degree of Doctor of Philosophy

Shahar Timnat

Submitted to the Senate
of the Technion — Israel Institute of Technology
Sivan 5775 Haifa June 2015

This research was carried out under the supervision of Prof. Erez Petrank, in the Faculty of Computer Science.

Some results in this thesis have been published as articles by the author and research collaborators in conferences during the course of the author's doctoral research period, the most up-to-date versions of which being:

Keren Censor-Hillel, Erez Petrank, and Shahar Timnat. Help! In *Proceedings of the 34th Annual ACM Symposium on Principles of Distributed Computing, PODC 2015, Donostia-San Sebastian, Spain, July 21-23, 2015*.

Erez Petrank and Shahar Timnat. Lock-free data-structure iterators. In *Distributed Computing - 27th International Symposium, DISC 2013, Jerusalem, Israel, October 14-18, 2013. Proceedings*, pages 224–238, 2013.

Shahar Timnat, Anastasia Braginsky, Alex Kogan, and Erez Petrank. Wait-free linked-lists. In *Principles of Distributed Systems, 16th International Conference, OPODIS 2012, Rome, Italy, December 18-20, 2012. Proceedings*, pages 330–344, 2012.

Shahar Timnat, Maurice Herlihy, and Erez Petrank. A practical transactional memory interface. In *Euro-Par 2015 Parallel Processing - 21st International Conference, Vienna, Austria, August 24-28, 2015. Proceedings.*

Shahar Timnat and Erez Petrank. A practical wait-free simulation for lock-free data structures. In *ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPOPP)*, pages 357–368, 2014.

Acknowledgements

I wish to thank my advisor, professor Erez Petrank, for the guidance and support he has given me. Throughout my studies, I have always been fully persuaded that he was the right advisor for me. Erez has always received me with his good-hearted smile, and he always had excellent advice to offer for my research and for life in general. For me, he is much more than a Ph.D. advisor.

I would also like to extend a special word of thanks to professor Keren Censor-Hillel. The work with her has not only been highly productive, but also a lot of fun. Finally, I wish to thank professor Maurice Herlihy for his wise ideas which pointed me to the right directions, and for his kind attitude which made the experience very pleasant.

The Technion's funding of this research is hereby acknowledged. I would also like to thank Mr. and Mrs. Jacobs for the Irwin and Joan Jacobs Fellowship, which I received in 2012 while I was still an M.Sc. student. My work was also supported by the Israeli Science Foundation grant No. 283/10, and by the United States - Israel Binational Science Foundation (BSF) grant No. 2012171.

Contents

List of Figures

Abstract **1**

1 Introduction **3**
1.1 Wait-Free Linked-Lists 4
1.2 A General Wait-Free Simulation for Lock-Free Data Structures 5
1.3 On the relations between Wait-Freedom and Help 6
1.4 Iterator 8
1.5 Harnessing HTM to Support Data Structures with Progress Guarantees 10
1.6 Outline of this Thesis 12

2 Wait-Free Linked-Lists **13**
2.1 Introduction 13
2.2 An Overview of the Algorithm 13
2.3 The Algorithm 15
2.3.1 The Underlying Data Structures 16
2.3.2 The Helping Mechanism 17
2.3.3 The Search Methods 17
2.3.4 The Insert Operation 18
2.3.5 The Delete Operation 21
2.3.6 The Contains Operation 24
2.3.7 Memory management 24
2.4 A Correctness Proof 24
2.4.1 Highlights 25
2.4.2 General 26
2.4.3 Definitions 26
2.4.4 General List Invariants 29
2.4.5 The Insert Operation 40
2.4.6 The Delete Operation 46
2.4.7 Wait-Freedom 51
2.4.8 Final Conclusion 54

2.5 Linearization Points . . . 54
2.6 A Fast-Path-Slow-Path Extension . . . 55
2.6.1 overview . . . 55
2.6.2 The Delayed Help Mechanism . . . 56
2.6.3 The Search Method . . . 57
2.6.4 The Insert Operation . . . 57
2.6.5 The Delete Operation . . . 57
2.6.6 Linearization Points . . . 60
2.6.7 The Contains Operation and Handling Infinite Insertions . . . 60
2.7 Performance . . . 61
2.8 Conclusion . . . 63

3 A Practical Wait-Free Simulation for Lock-Free Data Structures 65
3.1 Introduction . . . 65
3.2 Additional Related Work . . . 66
3.3 Transformation overview . . . 66
3.4 Model and General Definitions . . . 68
3.5 Typical Lock-Free Algorithms . . . 69
3.5.1 Motivating Discussion . . . 69
3.5.2 Notations and Definitions Specific to the Normalized Form. . . . 70
3.6 Normalized Lock-Free Data Structures . . . 73
3.6.1 The Normalized Representation . . . 74
3.7 Transformation Details . . . 76
3.7.1 The Help Queue and the Operation Record . . . 77
3.7.2 Giving Help . . . 78
3.8 Correctness . . . 82
3.8.1 Correctness of the executeCASes Implementation . . . 83
3.8.2 Linearizability and WaitFreedom . . . 99
3.9 On the Generality of the Normalized Form . . . 109
3.10 Examples: the Transformation of Four Known Algorithms . . . 110
3.10.1 Harris's linked-list . . . 111
3.10.2 Binary Search Tree . . . 112
3.10.3 Skiplist . . . 115
3.10.4 The Linked-List of Fomitchev and Ruppert . . . 117
3.11 Optimizations . . . 119
3.11.1 Using the Original Algorithm for the Fast Path . . . 119
3.11.2 Avoiding versions . . . 120
3.12 Performance . . . 121
3.12.1 Memory Management . . . 121
3.12.2 Our Wait-Free Versions vs. the Original Lock-Free Structures . . . 121
3.12.3 Our Wait-Free Transformation vs. a Universal Construction . . . 122

4 Help! 127
4.1 Introduction . . . 127
4.2 Additional Related Work . . . 128
4.3 Model and Definitions . . . 128
4.4 What is Help? . . . 130
4.4.1 Intuitive Discussion . . . 130
4.4.2 Help Definition . . . 131
4.4.3 General Observations . . . 133
4.5 Exact Order Types . . . 133
4.5.1 Generalizing the Proof To Cover the Fetch&Add Primitive . . . 140
4.6 Global View Types . . . 142
4.6.1 From Single Scanner Snapshot to Global View Types . . . 151
4.7 Max Registers . . . 153
4.8 Types that Do Not Require Help . . . 157
4.8.1 A Help-Free Wait-Free Set . . . 157
4.8.2 A Help-Free Wait-Free Max Register . . . 158
4.9 A Universality of Fetch-And-Cons . . . 159
4.10 Discussion . . . 160

5 Lock-Free Data-Structure Iterators 163
5.1 Introduction . . . 163
5.2 Goals and Limitations . . . 164
5.3 Jayanti's Single Scanner Snapshot . . . 165
5.4 From Single Scanner Snapshot to Multiple Data Structure Snapshots . . . 167
5.4.1 Reporting the Operations of Other Threads . . . 167
5.4.2 Supporting Multiple Snapshots . . . 167
5.5 The Data Structure Snapshot Algorithm . . . 168
5.5.1 The Reporting Mechanism . . . 168
5.5.2 Performing a Data Structure Snapshot . . . 171
5.5.3 Memory Reclamation . . . 172
5.6 The Snap-Collector Object . . . 174
5.6.1 The Snap-Collector Implementation . . . 176
5.6.2 Some Simple Optimizations . . . 178
5.7 Proof . . . 179
5.7.1 Overview . . . 180
5.7.2 Definitions . . . 181
5.7.3 Constructing the Whole-Order . . . 186
5.7.4 Visibility Properties . . . 186
5.7.5 Sequential and Real-Time Consistency of the Whole-Order. . . . 198
5.7.6 Adjusting the Proof for Multiple Scanners . . . 202
5.7.7 Linearizability of the Snap-Collector . . . 203

5.8 Performance 204
5.9 Conclusion 206

6 A Practical Transactional Memory Interface 209
6.1 Introduction 209
6.2 Additional Related Work 209
6.3 The MCMS Operation 210
6.3.1 Implementing MCMS with Hardware Transactional Memory . . 211
6.3.2 Implementing MCMS without TM support 212
6.4 The Linked-List Algorithm 212
6.5 The Binary Search Tree Algorithm 214
6.6 Fall-back Execution for Failed Transactions 215
6.6.1 Using Locking for the Fall-back Path 217
6.6.2 Non-Transactional MCMS Implementation as a Fall-back Path . 217
6.6.3 A Copying-Based Fall-back path 218
6.7 Performance 218
6.8 Conclusions 220

7 Conclusions 223

A A Full Java Implementation of the Wait-Free Linked List 225

B Avoiding Versioned Pointers for the Wait-Free Linked List 233

C The Full Code of the Fast-Path-Slow-Path Extension for the Wait-Free Linked-List 239

D The Wait-Free Queue Used in the Wait-Free Simulation 251

E Implementing a Contention Failure Counter in the Presence of Infinite Insertions 255

Hebrew Abstract i

List of Figures

2.1 Insert and Delete Overview 14
2.2 General structure 16
2.3 The Help and Search methods 18
2.4 The insert operation 20
2.5 The delete operation 23
2.6 The *contains* and *helpContains* methods 25
2.7 The delayed help mechanism 57
2.8 The FPSP fastSearch method 58
2.9 The FPSP insert method 58
2.10 The FPSP slowInsert method 59
2.11 The FPSP delete method 59
2.12 The FPSP slowDelete method 60
2.13 The number of operations done in two seconds as a function of the number of threads 63

3.1 Operation Record 77
3.2 The help method 78
3.3 The helpOp method 79
3.4 The preCASes method 80
3.5 The executeCASes Method 82
3.6 The postCASes Method 83
3.7 Lock-Free versus Wait-Free algorithms, 1024 keys. Left: AMD. Right: IBM 123
3.8 Number of Operation Completed in the Slow Path., 1024 keys. Left: AMD. Right: IBM 123
3.9 Lock-Free versus Wait-Free algorithms, 64 keys. Left: AMD. Right: IBM 123
3.10 Number of Operation Completed in the Slow Path, 64 keys. Left: AMD. Right: IBM 124
3.11 Our Wait-Free List against a Universal Construction List 125
3.12 Ratio between Our List and a Universal Construction List 125

4.1 The algorithm for constructing the history in the proof of Theorem 4.3. 137
4.2 The algorithm for constructing the history in the proof of Theorem 4.7. 144
4.3 The algorithm for constructing the history in the proof of Theorem 4.11. 154

4.4 A help-free wait-free set implementation 158
4.5 A help-free wait-free max register implementation 159

5.1 Jayanti's single scanner snapshot algorithm 166
5.2 Adding a Snapshot Support to an Underlying Set Data Structure 173
5.3 An Implementation of the Snap-Collector 176
5.4 Generating The Whole-Order for an Execution 187
5.5 Results for 32 possible keys (left) 128 possible keys (middle) 1024 possible keys (right) 207

6.1 The MCMS Semantics (left) and its HTM Implementation (right) . . . 211
6.2 The List and Tree Algorithms 214
6.3 The Tree Algorithm 216
6.4 MCMS-based lists vs. Harris's linked-list. The x-axis represents the number of threads. The y-axis represents the total number of operations executed in a second (in millions key ranges 32 and 1024, in thousands for key range 1048576. 220
6.5 MCMS-based trees vs. the BST of Ellen et al. The x-axis represents the number of threads. The y-axis represents millions of operations executed per second. 221

Abstract

In today's world, where nearly every desktop and laptop contains several cores, parallel computing has become the standard. Concurrent data structures are designed to utilize all available cores to achieve faster performance. In this thesis we design new concurrent data structures, we provide techniques for improving the guarantees of concurrent data structures, we propose efficient iterators for concurrent data structures, we propose new programming techniques, and we formally prove some inherent limitations of concurrent data structures.

In particular, we study data structures that offer progress guarantees. Wait-freedom, which is the strongest progress guarantee by the standard definitions, is a central concept in this thesis. We start by designing the first wait-free linked-list with practical performance. We then generalize the technique, and offer an automatic transformation that allows even a non-expert to design efficient wait-free data structures. We use the proposed transformation to obtain fast wait-free skiplist, and binary search tree.

Our study continues with an investigation of the concept of help in wait-free algorithms. The wait-free progress guarantee is often achieved by allowing some threads to help other threads complete their own work. We propose a formal definition for the notion of help, and prove that many wait-free data structures cannot be implemented without using help.

Our next step is to design an iterator that can be used in concurrent wait-free data structures. An iterator is an interface which allows a traversal of all of the nodes that belong to a certain data structure. Until recently, no wait-free data structures offered support for an iterator. Finally, we propose a programming paradigm that facilitates the use of hardware transactional memory (HTM) with concurrent data structures, and particularly with concurrent data structures that provide a progress guarantee.

Chapter 1

Introduction

The era of multi-core architectures has been having a huge impact on software development: exploiting parallelism has become the main challenge of today's programming. With multiple processors communicating by accessing shared memory, the behavior of concurrent algorithms is measured by both *safety/correctness* and *progress* conditions. Typically, the stronger the progress guarantee is, the harder it is to design the algorithm, and often, stronger progress guarantees come with a higher performance cost.

Most of the code written today is lock-based, but this is shifting towards codes without locks [HS08]. Standard progress guarantees include *obstruction-freedom*, *lock-freedom* (a.k.a. *non-blocking*), and *wait-freedom*. The strongest among these is wait-freedom. A wait-free algorithm guarantees that *every* thread makes progress (typically completing a method) in a finite number of steps, regardless of other threads' behavior. The holy grail of designing concurrent data structures is in obtaining efficient wait-free implementations, with research dating back to some of the most important studies in distributed computing [Lam74, FLP85, Her88].

This worst-case guarantee has its theoretical appeal and elegance, but is also critical in practice for making concurrent data structures useable with real-time systems. Even when run on a real-time platform and operating system, a concurrent application must ensure that each thread makes its deadlines, i.e., has a bounded worst-case response time in worst-case scenarios. Furthermore, wait-freedom is a desirable progress property for many systems, and in particular operating systems, interactive systems, and systems with service-level guarantees. For all those, the elimination of starvation is highly desirable. However, until recently, only few wait-free data structures were known, as they are considered notoriously hard to design, and largely inefficient.

The weaker lock-freedom guarantee is more common. A lock-free algorithm guarantees that at least *one* thread makes progress in a finite number of steps. The downside of the lock-free guarantee is that all threads but one can starve in an execution, meaning that lock-freedom cannot suffice for a real-time scenario. As lock-free data structures are easier to design, constructions for many lock-free data structures are available in the literature, including the stack [HS08], the linked-list [Har01], the skiplist [HS08], and

the binary search tree [EFRvB10]. Furthermore, practical implementations for many lock-free algorithms are readily available in standard Java libraries and on the Web.

Recently, wait-free designs for the simple stack and queue data structures appeared in the literature [KP11, FK11]. Wait-free stack and queue structures are not easy to design, but they are considered less challenging as they present limited parallelism, i.e., a limited number of contention points (the head of the stack, and the head and the tail of the queue).

The existence of wait-free data structures has been shown by Herlihy [Her90] using universal simulations. Universal simulation techniques have evolved dramatically since then (e.g., [Her93, AM99, ADT95, Gre02, FK09, CER10, CIR12]), but even the state of the art universal construction [CER10] is too slow compared to the lock-free or lock-based implementations and cannot be used in practice[1]. Universal constructions achieve a difficult task, as they go all the way from a sequential data structure to a concurrent wait-free implementation of it. It may therefore be hard to expect that the resulting wait-free algorithm will be efficient enough to become practicable.

This thesis provides new and efficient wait-free data structures, better techniques to design them, and better understanding of the nature of wait-freedom. We offer a rigorous study of the concept of *help*, which we prove to be essential in many wait-free structures; We enhance both wait-free and lock-free structures by adding a fast wait-free iterator to them; and we suggest a programming paradigm that can be used to harness transactional memory (TM) to be used in data structures, even if a progress guarantee is required.

1.1 Wait-Free Linked-Lists

A linked-list is one of the most commonly used data structures. The linked-list seems a good candidate for parallelization, as modifications to different parts of the list may be executed independently and concurrently. Indeed, parallel linked-lists with various progress properties are abundant in the literature. Among these are lock-free linked-lists.

A lock-free linked-list was first presented by Valois [Val95]. A simpler and more efficient lock-free algorithm was designed by Harris [Har01], and Michael [Mic04] added a hazard-pointers mechanism to allow lock-free memory management for this algorithm. Fomitchev and Ruppert achieved better theoretical complexity in [FR04].

The first contribution of this thesis is a practical, linearizable, fast and wait-free design and implementation of a linked-list. Our construction builds on the lock-free linked-list of Harris [Har01], and extends it using a helping mechanism to become wait-free. The main technical difficulty is making sure that helping threads perform each operation correctly, apply each operation exactly once, and return a consistent result (of success or failure) according to whether each of the threads completed the

[1]The claim for inefficiency of universal constructions has been known as a folklore only. In Section 3.12.3 we provide the first measurements substantiating this claim.

operation successfully. This task is non-trivial and it is what makes wait-free algorithms notoriously hard to design.

Next, we extend our design using the fast-path-slow-path methodology [KP12], in order to make it even more efficient, and achieve performance that is almost equivalent to that of the lock-free linked-list of Harris. Here, the idea is to combine both lock-free and wait-free algorithms so that the (lock-free) fast path runs with (almost) no overhead, but is able to switch to the (wait-free) slow path when contention interferes with its progress. It is also important that both paths are able to run concurrently and correctly.

The fast-path-slow-path method attempts to separate slow handling of difficult cases from the fast handling of the more typical cases. This method is ubiquitous in systems in general and in parallel computing particularly [Lam87, MA95, AK99, AK00]. It has been adapted recently in [KP12] for creating fast wait-free data structures.

According to the fast-path-slow-path methodology of [KP12], an operation starts executing using a fast lock-free algorithm, and only moves to the slower wait-free path upon failing to make progress in the lock-free execution. It is often the case that an operation execution completes in the fast lock-free path, achieving good performance. But some operations fail to make progress in the fast path due to contention, and in this case, the execution moves to the slower wait-free path in which it is guaranteed to make progress. As many operations execute on the fast (lock-free) path, the performance of the combined execution is almost as fast as that of the lock-free data structure. It is crucial to note that even the unlucky threads, that do not manage to make progress in the fast path, are guaranteed to make progress in the slow path, and thus the strong wait-free guarantee can be obtained. Thus, we obtain the best of both worlds: the performance and scalability of the lock-free algorithm combined with the wait-free guarantee.

1.2 A General Wait-Free Simulation for Lock-Free Data Structures

Our next step is to examine the design process we did to obtain the fast wait-free linked-list, and try to generalize it for a wide range of data structures. The design process of our wait-free linked-list, and also of the wait-free queue presented in [KP11, KP12], is to start with a lock-free data structure, work (possibly hard) to construct a correct wait-free data structure by adding a helping mechanism to the original data structure, and then work (possibly hard) again to design a correct and efficient fast-path-slow-path combination of the lock-free and wait-free versions of the original algorithm. Designing a slow-path-fast-path data structure is non-trivial. One must design the lock- and wait-free algorithms to work in sync to obtain the overall combined data structure with the required properties.

We ask whether this entire design can be done mechanically, and so also by non-

experts. More accurately, given a lock-free data structure of our choice, can we apply a generic method to create an adequate helping mechanism to obtain a wait-free version for it, and then automatically combine the original lock-free version with the obtained wait-free version to obtain a fast, practical wait-free data structure?

We answer this question in the affirmative. Thus, the second major contribution of this thesis is an automatic transformation that takes a linearizable lock-free data structure in a normalized representation (that we define) and produces a practical wait-free data structure from it. The resulting data structure is almost as efficient as the original lock-free one.

We claim that the normalized representation we propose is meaningful in the sense that important known lock-free data structures can be easily specified in this form. In fact, all linearizable lock-free data structures that we are aware of in the literature can be stated in a normalized form. We demonstrate the generality of the proposed normalized form by stating several important lock-free data structures in their normalized form and then obtaining wait-free versions of them using the mechanical transformation. In particular, we transform the linked-list [Har01, FR04], the skiplist [HS08], and the binary search tree [EFRvB10], and obtain practical wait-free designs for them all.

1.3 On the relations between Wait-Freedom and Help

When designing our wait-free linked-list, and also when presenting our general transformation, we employed a *helping* mechanism. This approach is frequently used in wait-free designs in literature [Plo89, Her88, Her90, Her91, HS08, KP11, FK11, KP12]. Loosely speaking, in helping mechanisms, apart from completing their own operation, processes perform some additional work whose goal is to facilitate the work of others. Curiously, despite being a crucial ingredient, whether explicitly or implicitly, in many implementations of concurrent data structures, the notion of helping has been lacking thorough study as a concept.

Intrigued by the tools needed in order to achieve wait-freedom, we offer a rigorous study of the interaction between the helping paradigm and wait-freedom. In particular, we are interested in the following question: Does wait-freedom require help? To this end, we start by proposing a formal definition of help. The proposed definition is based on linearization order of histories of an implementation rather than on a semantic description. We give evidence that the proposed definition matches the intuitive notion. We then present and analyze properties of types for which any wait-free implementation necessitates help. Such types include popular data structures such as the stack and the queue. In contrast, we show that other types can be implemented in a wait-free help-free manner. A natural example is an implementation of a set (with the INSERT, DELETE, and CONTAINS operations) with a bounded range of possible values.

We note that there is some ambiguity in the literature regarding the concept of help; it is used informally to describe two different things. One usage of help is in the common

case where processes of lock-free algorithms coordinate access to a shared location. Here, one process p_1 completes the (already ongoing) operation of another process p_2 in order to enable access to shared data and to allow p_1 to complete its operation. Barnes [Bar93] uses this practice as a general technique to achieve lock-freedom. This is also the case for the queue of [MS96], where processes sometimes need to fix the tail pointer to point to the last node (and not the one before last) before they can execute their own operation. Loosely speaking, the purpose of the above practice is not "altruistic". A process fixes the tail pointer because otherwise it would not be able to execute its own operation.

This is very different from the usage of help in, e.g., UPDATE operations in [AAD+93], which perform embedded scans for the sole "altruistic" purpose of enabling concurrent SCAN operations. It also differs from reading a designated announcements array, whose sole purpose is to allow processes to ask other processes for help, such as in [Her88]. In [Her88], a process could have easily completed its operation without helping any other operation (by proposing to the consensus object used in this build a value that consists only the process's own value, without values of other processes viewed in the announcements array). Our definition of help deliberately excludes the former concept (where a process simply enables data access for its own operation), and captures only the latter "altruistic" form of help.

Having a formal notion of helping, we turn to study the interaction between wait-freedom and help. We look into characterizing properties of types that require help in any wait-free implementation. We define and analyze two general types of objects. The first type, which we call *Exact Order Types*, consists of types in which the order of operations affects the result of future operations. That is, for some operations sequences, every change in the order of operations influences the final state of the object. Natural examples of exact order types are FIFO queues and LIFO stacks.

We note that exact order types bare some similarity to previously defined objects, such as *perturbable objects* [JTT00] and *class G objects* [EHS12], since all definitions deal with an operation that needs to return different results in several different executions. However, these definitions are not equivalent. For example, queues are exact order types, but are not perturbable objects, while a max-register is perturbable but not exact order. We mention perturbable objects in Section 4.10.

The second type, which we call *Global View Types*, consists of types which support an operation that obtains the entire state of the object. Examples of global view types are snapshot objects, increment objects, and fetch&add. For instance, in an increment object that supports the operations GET and INCREMENT, the result of a GET depends on the exact number of preceding INCREMENTS. However, unlike the queue and stack, the result of an operation is not necessarily influenced by the internal order of previous operations. Notice that global view types are not equivalent to *readable objects* as defined by Ruppert [Rup00], since for some global view types any applicable operation must change the state of the object. For example, a fetch&increment object is a global

view type, but is not a readable object.

We prove that every wait-free implementation of any exact order type and any global view type requires help. Furthermore, when the CAS primitive is not available, we show that a max register [AACH12] requires help even in order to provide lock-freedom.

Theorems 4.3, 4.8, 4.11 (rephrased): A linearizable implementation of a wait-free exact order type or a wait-free global view type using READ, WRITE, and CAS, or a lock-free max register using READ and WRITE, cannot be help-free.

We prove the above by constructing infinite executions in which some operation never completes unless helping occurs. This is done by carefully combining the definition of help with the attributes of the type.

We then show positive results, i.e., that some types can be implemented in a wait-free help-free manner. This is trivial for a *vacuous* type whose only operation is a NO-OP, but when CASes are available this also holds for a max register and for a *set* type, which supports the operations INSERT, DELETE and CONTAINS[2].

The proof that these types have wait-free help-free implementations can be generalized to additional types, provided they have an implementation in which every operation is linearized in a specific step of the same operation. Intuitively, these are implementations in which the result of an operation "does not depend too strongly" on past operations.

Naturally, the characterization of types which require help depends on the primitives being used, and while our results are generally stated for READ, WRITE, and CAS, we discuss additional primitives as well. In particular, we show that exact order types cannot be both help-free and wait-free even if the FETCH&ADD primitive is available, but the same statement is not true for global view types. Finally, we show that a fetch&cons primitive is universal for wait-free help-free objects. This means that given a wait-free help-free fetch&cons object, one can implement any type in a wait-free help-free manner.

1.4 Iterator

Almost none of the designs of wait-free, or even of lock-free structures, support operations that require global information on the data structure, such as counting the number of elements in the structure or iterating over its nodes. In general, operations such as these will be trivially enabled if snapshot operations are supported because snapshot operations enable a thread to obtain an atomic view of the structure. But creating a "consistent" or linearizable snapshot without blocking simultaneous updates to the data structure is a difficult task.

[2]A degenerated set, in which the INSERT and DELETE operations do not return a boolean value indicating whether they succeeded can also be implemented without CASes.

The next major contribution of this thesis, is the design of wait-free, highly efficient iterators for concurrent data structures that implement sets. We use this design to implement iterators for the linked-list and skiplist. The iterator is implemented by first obtaining a consistent *snapshot* of the data structure, i.e., an atomic view of all the nodes currently in it. Given this snapshot, it is easy to provide an iterator, or to count the number of nodes in the structure.

A well-known related problem is the simpler *atomic snapshot object* of shared memory [AAD+93], which has been extensively studied in the literature. An atomic snapshot object supports only two types of operations: UPDATE and SCAN. An UPDATE writes a new value to a register in the shared memory, and a SCAN returns an atomic view of all the registers.

Unfortunately, existing snapshot algorithms cannot easily be extended to support a (practical) data structure iterator. One problem is that atomic snapshot objects are designed for pre-allocated and well-defined memory registers. Therefore, they are not applicable to concurrent data structures that tend to grow and shrink when nodes are added or removed. Still, one could imagine borrowing ideas from snapshot objects, generalizing them, and building a snapshot algorithm for a memory space that grows and shrinks.

A more substantial problem is that data structures require both fast read and fast write (update) operations. The UPDATE operation in classic snapshot object algorithms [AAD+93, And94] requires O(n) steps (n is the number of threads), which is too high an overhead to impose on all operations that modify the data structure. Later snapshot algorithms support UPDATE in O(1) steps. Examples are the coordinated collect algorithm of Riany et al. [RST95], subsequently leading to the interrupting snapshots algorithm [AST09], and the time optimal snapshot algorithms of Fatourou and Kallimanis [FK07].

The simple nature of a READ operation, i.e., reading a memory register, might at first glance suggest that implementing it in O(1) should be easy. However, this is not the case. State of the art algorithms that support UPDATE in O(1) steps employ non-trivial linearization properties. Some of them ([RST95, AST09, FK07]) even allow the linearization point of an UPDATE to occur before the new value has actually been written to *any* register in the memory. Thus, a simple READ has no way of retrieving the value it is supposed to return, since this result is not available anywhere. Consequently, there is no available snapshot object algorithm that supports both fast reads and fast writes. One might think of it as a tradeoff: the complex linearization properties that are used to enable UPDATE in O(1), are precisely those that prevent an implementation of a READ operation. Section 5.3 specifies Jayanti's algorithm, which is an example for an algorithm that uses such an unusual linearization point.

The algorithm of Jayanti [Jay05] is wait-free and supports UPDATE operations in O(1) steps. Jayanti's algorithm does not support a READ operation, and it is not trivial to add an efficient READ to it, but our work builds on ideas from this algorithm. An

UPDATE operation of Jayanti's algorithm first executes the required update, and then checks whether a SCAN is currently being taken. If so, the update operation announces the update again in a designated memory register. In this work we extend this basic idea to provide a snapshot that supports an efficient READ as well as the INSERT, DELETE, and CONTAINS operations, which are more complex than the simple UPDATE operation. This facilitates the desirable iterator operation for the data structure.

Although most lock-free data structures do not provide iterators, one notable exception is the recent CTrie of Prokopec et al. [PBBO12]. This lock-free CTrie efficiently implements the creation of a snapshot in constant time, but the performance of updates deteriorates when concurrent snapshots are being taken, because each updated node must be copied, together with the path from the root to it. An additional recent work presenting a concurrent data structure that supports snapshot operations is the practical concurrent binary search tree of Bronson et al. [BCCO10]. But their work uses locks, and does not provide a progress guarantee.

This thesis presents a wait-free snapshot mechanism that implements an $O(1)$ UPDATE and READ operations. We have implemented a linked-list and skiplist that employ the snapshot and iterator and measured the performance overheads. In our implementation we made an effort to make updates as fast as possible, even if iterations take a bit more time. The rationale for this design is that iterations are a lot less frequent than updates in typical data structures use. It turns out that the iterator imposes an overhead of roughly 15% on the INSERT, DELETE, and CONTAINS operations when iterators are active concurrently, and roughly 5% otherwise. When compared to the CTrie iterator of [PBBO12], our iterator demonstrates lower overhead on modifications and read operations, whereas the iteration of the data structure is faster with the CTrie iterator.

1.5 Harnessing HTM to Support Data Structures with Progress Guarantees

Transactional memory (TM) is becoming an increasingly central concept in parallel programming. Recently, Intel introduced the TSX extensions to the x86 architecture, which include RTM: an off-the-shelf hardware that supports hardware transactional memory. There are practical reasons for a developer to avoid using hardware transactional memory. First, HTM is only available for some of the computers in the market. Thus, a code that relies on HTM only suits a fraction of the available computers and must be accompanied with a different code base for the other platforms. Second, RTM transactions are "best effort" and are not guaranteed to succeed. Thus, to work with HTM, a *fall-back* path must also be provided and maintained, in case transactions repeatedly fail.

The final contribution of this thesis is a new programming discipline for highly-

concurrent linearizable objects that takes advantage of HTM when it is available, and still performs reasonably when it is not available. For this purpose, we suggest to encapsulate the HTM inside an intermediate level operation. The intermediate operation is compiled to an HTM implementation on platforms that support HTM, and to a non-transactional implementation otherwise. To a certain extent, our intermediate operation can even be implemented with an "out of the box" fall-back path for failing transactions. This fall-back path can be made lock-free, or even wait-free, thus rendering our operation a valid alternative for designing lock-free operations.

The intermediate operation we find best suited for this purpose is a slight variation of the well-known MCAS (Multi-word Compare And Swap) operation. The MCAS operation executes atomically on several shared memory addresses. Each address is associated with an *expected-value* and a *new-value*. An execution of MCAS succeeds and returns true iff the content of each specified address equals its expected value. In this case, the data in each address is replaced with the new value. If any of the specified addresses contains data that is different from the expected value, then false is returned and the content of the shared memory remains unchanged.

We propose an extended interface of MCAS called MCMS (Multiple Compare Multiple Swap), in which we also allow addresses to be compared without being swapped. The extension is functionally redundant, because, in effect, comparing an address without swapping it is identical to an MCAS in which this address' expected value equals its new value. However, when implementing the MCMS using transactional memory, it is ill-advised to write a new (identical) value to replace an old one. Such a replacement may cause unnecessary transaction aborts.

In order to study the usability of the MCMS operation, we designed two algorithms that use it. One for the linked-list data structure, and one for the binary search tree. The MCMS tree is almost a straightforward MCMS-based version of the lock-free binary search tree by Ellen et al. [EFRvB10]. But interestingly, attempting to design a linked-list that exploits the MCMS operation yielded a new algorithm that is highly efficient. The main idea is to mark a deleted node in a different and useful manner. Instead of using a mark on the reference (like Harris [Har01]), or using a mark on the reference and additionally a backlink (like Fomitchev and Ruppert [FR04]), or using a separate mark field (like the lazy linked-list [HHL$^+$05]), we mark a node deleted by setting its pointer to be a back-link, referencing the previous node in the list. This approach works excellently with transactions.

We present three simple fall-back alternatives to enable progress in case RTM executions of MCMS repeatedly fail. The simplest way is to use locks, in a similar manner to *lock-elision* [RG01]. The second approach is to use CAS-based MCMS ([HFP02]) as a fall-back. The third alternative is a copying scheme, where a new copy of the data structure is created upon demand to guarantee progress. Both the linked-list and tree algorithm outperform their lock-free alternatives when using either a lock-based fall-back path or a copying fall-back path. The list algorithm performs up to X2.15

faster than Harris's linked-list, and the tree algorithm performs up to X1.37 faster than the tree of Ellen et al. A fall-back path (that does not use transactions) is at times a bit faster (up to X1.1) and at times a bit slower than the lock-free alternatives, depending on the specific benchmark and configuration.

Another important advantage of programming with MCMS is that the resulting algorithms are considerably simpler to design and debug compared to standard lock-free algorithms that build on the CAS operation. The stronger MCMS operation allows lock-free algorithms to be designed without requiring complicated synchronization mechanisms that facilitate lock-freedom.

1.6 Outline of this Thesis

The main contributions of this thesis appear in Chapters 2 – 6. Chapter 2 gives the design of our wait-free linked-list, and the extension of it using the fast-path-slow-path technique to match the performance of the lock-free linked-list of Harris. In Chapter 3 we present our general technique to transform lock-free data structures into wait-free ones, and use it to obtain four new wait-free data structures. In Chapter 4 we explore the interaction between help and wait-freedom. We give a formal definition of the concept of help, and show that many wait-free data structures cannot be implemented without employing a help mechanism. In Chapter 5 we present a design of an efficient iterators for lock-free and wait-free data structures that implement sets. In Chapter 6 we discuss our proposed MCMS operation, and show how to use it to obtain faster lock-free data structures, and still keep a simple design.

Each chapter starts with a small introduction, which adds details and related work, and also connects the chapter to the previous chapters of this thesis. Excluding Chapter 4, which is more theoretical, all chapters present new designs and implementations of data structures, and end with a performance measurements section that compare the new designs to previous alternatives. The individual chapters also include correctness proves for the main implementations and techniques presented in them.

Chapter 2

Wait-Free Linked-Lists

2.1 Introduction

As discussed in Section 1.1, this chapter gives the first practical implementation of a wait-free linked-list. A (shorter) version of the work presented here was published in [TBKP12]. Our build starts from the lock-free list of Harris [Har01], and add a helping mechanism to obtain wait-freedom. We then use the fast-path-slow-path technique of [KP12] to achieve faster performance. This chapter includes the design of the linked-list, correctness proof, and performance measurements.

We compared our wait-free linked-list's efficiency with that of Harris's lock-free linked-list. Our first design(slightly optimized) performs worse by a factor of 1.5 when compared to Harris's lock-free algorithm. This provides a practical, yet not optimal, solution. However, the fast-path-slow-path extension reduces the overhead significantly, bringing it to just 2-15 percents. This seems a reasonable price to pay for obtaining a data structure with the strongest wait-free guarantee, providing non-starvation even in worst-case scenarios, and making it available for use with real-time systems.

We begin in Section 2.2 with an overview of the algorithm and continue in Section 2.3 with a detailed description of it. The correctness proof appears in Section 2.4. The linearization points of the algorithm are specified in Section 2.5. We give describe the fast-path-slow-path extension of the algorithm in Section 2.6, Section 2.7 presents the performance measurements, and we conclude in Section 2.8. Java implementations of the wait-free algorithm and of the fast-path-slow-path extension are given in Appendices A and C respectively. The basic algorithm uses versioned pointers (pointers with a counter associated with them). Appendix B gives a Java implementation of a variation of the algorithm that eliminates their need, and uses only regular pointers.

2.2 An Overview of the Algorithm

Before getting into the technical details (in Section 2.3) we provide an overview of the design. The wait-free linked-list supports three operations: INSERT, DELETE, and

CONTAINS. All of them run in a wait-free manner. The underlying structure of the linked-list is depicted in Figure 2.2. Similarly to Harris's linked-list, our list contains sentinel `head` and `tail` nodes, and the `next` pointer in each node can be marked using a special `mark bit`, to signify that the entry in the node is logically deleted.

To achieve wait-freedom, our list employs a helping mechanism. Before starting to execute an operation, a thread starts by publishing an *Operation Descriptor*, in a special `state` array, allowing all the threads to view the details of the operation it is executing. Once an operation is published, all threads may try to help execute it. When an operation is completed, the result is reported to the state array, using a CAS which replaces the existing operation descriptor with one that contains the result.

A top-level overview of the insert and delete operations is provided in Figure 2.1. When a thread wishes to INSERT a key k to the list, it first allocates a new node with

```
1: boolean insert(key)
2:     Allocate new node (without help)
3:     Publish the operation (without help)
4:     Search for a place to insert the node
5:         If key already exists, return with failure
6:     Direct the new node next pointer
7:     Insert the node by directing its predecessor next pointer
8:     Return with Success
```

```
1: boolean delete(key)
2:     Publish the operation (without help)
3:     Search for the node to delete
4:         If key doesn't exist, return with failure
5:     Announce in the state array the node to be deleted
6:     Mark the node next pointer to make it logically deleted
7:     Physically remove the node
8:     Report the node has been removed
9:     Compete for success (without help)
```

Figure 2.1: Insert and Delete Overview

key k, and then publishes an operation descriptor with a pointer to the new node. The rest of the operation can be executed by any of the threads in the system, and may also be run by many threads concurrently. Any thread that executes this operation starts by searching for a place to insert the new node. This is done using the search method, which, given a key k, returns a pair of pointers, *prev* and *curr*. The prev pointer points to the node with the highest key smaller than k, and the curr pointer points to the node with the smallest key larger than or equal to k. If the returned curr node holds a key equal to the key on the node to be inserted, then failure is reported. Otherwise the node should be inserted between prev and curr. This is done by first updating the new node's `next` pointer to point to curr, and then updating prev's `next` field to point to it. Both of these updates are done using a CAS to prevent race conditions, and the failure of any of these CASes will cause the operation to restart from the search method. Finally, after that node has been inserted, success is reported.

While the above description outlines the general process of inserting a node, the actual algorithm is a lot more complex, and requires care to avoid problematic races that can make things go wrong. For example, when two different threads help insert the same node, they might get different prev and curr pointers back from the search method, due to additional changes that are applied concurrently on the list. This could lead to various problems, such as one of the threads reporting failure (since it sees another

node with the same key) while the other thread successfully inserts the node (since it doesn't see the same node, which has been removed). In addition to these possible inconsistencies, there is also a potential ABA problem that requires the use of a version mark on the `next` pointer field[1]. We discuss these and other potential races in Section 2.3.4.

When a thread wishes to DELETE a key k from the list, it starts by publishing the details of its operation in the `state` array. The next steps can be then executed by any of the threads in the system until the last step, which is executed only by the thread that initiated the operation, denoted the *owner thread*. The DELETE operation is executed (or helped) in two stages. First, the node to be deleted is chosen. To do this, the search method is invoked. If no node with the key k is found, failure is reported. Otherwise, the node to be deleted is *announced* in the `state` array. This is done by replacing the state descriptor that describes this operation to a state descriptor that has a pointer to the specific node to be deleted. This announcement helps to ascertain that concurrent helping threads will not delete two different nodes, as the victim node for this operation is determined to be the single node that is announced in the operation descriptor. In the second stage, deletion is executed similarly to Harris's linked-list: the removed node's `next` field is marked, and then this node is physically removed from the list. The node's removal is then reported back to the `state` array.

However, since multiple threads execute multiple operations, and as it is possible that several operations attempt to DELETE the same node, it is crucial that exactly one operation be declared as successfully deleting the node's key and that the others return failure. An additional (third) stage is required in order to consistently determine which operation can be considered successful. This step is executed only by the owner threads, and is given no help. The threads that initiated the concurrent delete operations compete among themselves for the ownership of the deletion. To this end, an extra `success-bit` designated for this purpose is added to each node in the list. The thread that successfully CASes this bit from false to true is the only one that reports success for this deletion. We believe that using an extra bit to determine an ownership of an operation is a useful mechanism for future wait-free constructions as well. The full details of the DELETE operation are given in Section 2.3.5.

The CONTAINS operation is much simpler than the other two. It starts by publishing the operation. Any helping thread will then search for it in the list, reporting success (on the operation record) if the key was found, or failure if it was not.

2.3 The Algorithm

In this section we present the details of the algorithm.

[1]The versioning method provides a simple solution to the ABA problem. A more involved solution that does not require a versioned pointer appears in Appendix B.

2.3.1 The Underlying Data Structures

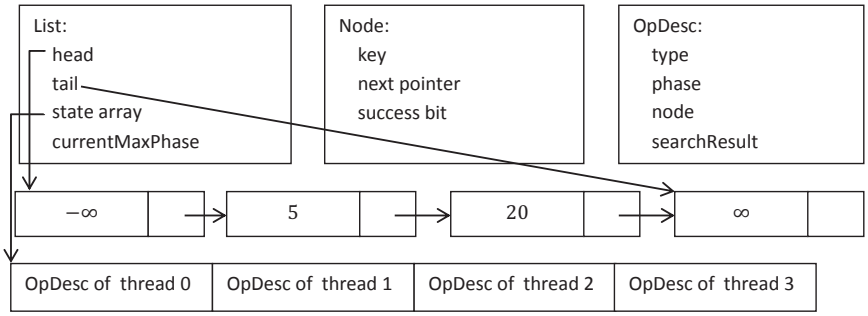

Figure 2.2: General structure

The structure of the linked-list is depicted in Figure 2.2. A node of the linked list consists of three fields: a key, a `success bit` to be used when deleting this node, and a special pointer field. The special pointer field has its least significant bit used by the algorithm for signaling between threads. In addition, this pointer is versioned, in the sense that there is a counter associated with it (in an adjacent word) and each modification of it (or of its special bit) increments the counter. The modification and counter increment are assumed to be atomic. This can be implemented by squeezing all these fields into a single word, and limiting the size of the counter and pointer, or by using a double-word compare-and-swap when the platform allows. Alternatively, one can allocate a "pointer object" containing all these fields and bits, and then atomically replace the existing pointer object with a new one. The latter approach is commonly used with Java lock-free implementations, and we use it as well.

In addition to the nodes of the list, we also maintain an array with an operation-descriptor for each thread in the system. The OpDesc entry for each thread describes its current state. It consists of a phase field `phase`, the `OpType` field signifying which operation is currently being executed by this thread, a pointer to a node, denoted `node`, which serves the insert and delete operations, and a pair of pointers $(prev, curr)$, for recording the result of a search operation. Recall that the result of a SEARCH operation of a key, k, is a pair of pointers denoted *prev* and *curr*, as explained in Section 2.2 above.

The possible values for the operation type (OpType) in the operation descriptor state are:

insert	asking for help in inserting a node into the list.
search_delete	asking for help in finding a node with the key we wish to delete.
execute_delete	asking for help in marking a node as deleted (by tagging its next pointer) and unlinking it from the list.
contains	asking for help in finding out if a node with the given key exists.
success	operation was completed successfully.
failure	operation failed (deletion of a non-existing key or insertion of an existing key).

determine_delete decide if a delete operation completed successfully.

The first four states in the above list are used to request help from other threads. The last three states indicate steps in the executions in which the thread does not require any help. The linked-list also contains an additional long field, `currentMaxPhase`, to support the helping mechanism, as described in Subsection 2.3.2.

2.3.2 The Helping Mechanism

Before a thread starts executing an operation, it first selects a phase number larger than all previously chosen phase numbers. The goal of assigning a phase number to each operation is to let new operations make sure that old operations receive help and complete before new operations are executed. This ensures non-starvation. The phase selection mechanism ensures that if operation O_2 arrives strictly later than operation O_1, i.e., O_1 receives a phase number before O_2 starts selecting its own phase number, then O_2 will receive a higher phase number. The phase selection procedure is executed in the maxPhase method depicted in Figure 2.3. Note that although a CAS is used in this method, the success of this CAS is not checked, thus preserving wait-freedom. If the CAS fails, it means that another thread increased the counter concurrently, which is sufficient for the phase numbering. After selecting a phase number, the thread publishes the operation by updating its entry in the `state` array. It then goes through the array, helping all operations with a phase number lower than or equal to its own. This ensures wait-freedom: a delayed operation eventually receives help from all threads and soon completes. See Figure 2.3 for the pseudo-code.

2.3.3 The Search Methods

The contains method, which is part of the data structure interface, is used to check whether a certain key is a part of the list. The search method is used (internally) by the insert, delete, and contains methods to find the location of a key and perform some maintenance during the search. It is actually nearly identical to the original lock-free search method. The search method takes a key and returns a pair of pointers denoted *window*: `pred`, which points to the node containing the highest key less than the input key, and `curr`, which points to the node containing the lowest key higher than or equal to the requested key. When traversing through the list, the search method attempts to physically remove any node that is logically deleted. If the remove attempt fails, the search is restarted from the head of the list. This endless attempt to fix the list seems to contradict wait-freedom, but the helping mechanism ensures that these attempts eventually succeed. When an operation delays long enough, all threads reach the point at which they are helping it. When that happens, the operation is guaranteed to succeed. The search operation will not re-iterate if the operation that executes it has completed, which is checked using the isSearchStillPending method.

```
 1: private long maxPhase() {
 2:     long result = currentMaxPhase.get();
 3:     currentMaxPhase.compareAndSet(result, result+1);
 4:     return result;
 5: }
 6:
 7: private void help(long phase) {
 8:     for (int i = 0; i < state.length(); i++) {
 9:         OpDesc desc = state.get(i);
10:         if (desc.phase <= phase) {       ▷ help older perations
11:             if (desc.type == OpType.insert) {
12:                 helpInsert(i, desc.phase);
13:             } else if (desc.type == OpType.search_delete
14:                 || desc.type == OpType.execute_delete) {
15:                 helpDelete(i, desc.phase);
16:             } else if (desc.type == OpType.contains) {
17:                 helpContains(i, desc.phase);
18:             } } } }
19:
20: private boolean isSearchStillPending(int tid, long ph) {
21:     OpDesc curr = state.get(tid);
22:     return (curr.type == OpType.insert ||
23:             curr.type == OpType.search_delete ||
24:             curr.type == OpType.execute_delete ||
25:             curr.type==OpType.contains) &&
26:         curr.phase == ph;
27: }
```

```
28: private Window search(int key, int tid, long phase) {
29:     Node pred = null, curr = null, succ = null;
30:     boolean[] marked = {false}; boolean snip;
31:     retry : while (true) {
32:         pred = head;
33:         curr = pred.next.getReference();       ▷ advancing curr
34:         while (true) {
35:                     ▷ Reading both the reference and the mark:
36:             succ = curr.next.get(marked);
37:             while (marked[0]) {        ▷ curr is logically deleted
38:                         ▷ Attempt to physically remove curr:
39:                 snip = pred.next.compareAndSet
40:                         (curr, succ, false, false);
41:                 if (!isSearchStillPending(tid,phase))
42:                     return null;             ▷ to ensure wait-freedom.
43:                 if (!snip) continue retry; ▷ list has changed, retry
44:                 curr = succ;                          ▷ advancing curr
45:                 succ = curr.next.get(marked);   ▷ advancing succ
46:             }
47:             if (curr.key >= key)           ▷ The window is found
48:                 return new Window(pred, curr);
49:             pred = curr; curr = succ;    ▷ advancing pred & curr
50:         }
51:     }
52: }
53:
54:
55:
```

Figure 2.3: The Help and Search methods

If the associated operation is complete, then the SEARCH method returns a null. The pseudo-code for the search method is depicted in Figure 2.3.

2.3.4 The Insert Operation

Designing operations for a wait-free algorithm requires dealing with multiple threads executing each operation, which is substantially more difficult than designing a lock-free operation. In this section, we present the insert operation and discuss some of the races that occur and how we handle them. The basic idea is to coordinate the execution of all threads using the operation descriptor. But more actions are required, as explained below. Of-course, a proof is required to ensure that all races have been handled. The pseudo-code of the INSERT operation is provided in Figure 2.4. The thread that initiates the operation is denoted *the operation owner*. The *operation owner* starts the INSERT method by selecting a phase number, allocating a new node with the input key, and installing a link to it in the `state` array.

Next, the thread (or any helping thread) continues by searching the list for a location where the node with the new key can be inserted (Line 17 in the method HELPINSERT). In the original lock-free linked-list, finding a node with the same key is interpreted as failure. However, in the presence of the helping mechanism, it is possible that some other thread that is helping the same operation has already inserted the node but has not yet reported success. It is also possible that the node we are trying to insert was already inserted and then deleted, and then a different node, with the same key, was

inserted into the list. To identify these cases, we check the node that was found in the search. If it is the same node that we are trying to insert, then we know that success should be reported. We also check if the (`next` field of the) node that we are trying to insert is *marked* for deletion. This happens if the node was already inserted into the list and then removed. In this case, we also report success. Otherwise, we attempt to report failure. If there is no node found with the same key, then we can try to insert the node between `pred` and `curr`. But first we check to see if the node was already inserted and deleted (line 35), in which case we can simply report success.

The existence of other threads that help execute the same operation creates various races that should be properly handled. One of them, described in the next paragraph, requires the INSERT method to proceed with executing something that may seem redundant at first glance. The INSERT method creates a state descriptor identical to the existing one and atomically replaces the old one with the new one (Lines 42–45). The replacement foils all pending CAS operations by other threads on this state descriptor, and avoids confusion as to whether the operation succeeds or fails. Next, the method executes the actual insertion of the node into the list (Lines 46–48) and it attempts to report success (Lines 49–52). If any of the atomic operations fail, the insertion starts from scratch. The actual insertion into the list (Lines 46–48) is different from the insertion in the original lock-free linked-list. First, the `next` pointer in the new node is not privately set, as it is now accessible by all threads that help the insert operation. It is set by a CAS which verifies that the pointer has not changed since before the search. Namely, the old value is read in Line 16 and used as the expected value in the CAS of Line 46. This verification avoids another race, which is presented below. Moreover, the atomic modification of the `next` pointer in the previous node to point to the inserted node (Lines 47–48) uses the version of that `next` pointer to avoid the ABA problem. This is also justified below.

Let us first present the race that justifies the (seemingly futile) replacement of the state descriptor in Lines 42–45. Suppose Thread T_1 is executing an INSERT operation of a key k. T_1 finds an existing node with the key k and is about to report failure. T_1 then gets stalled for a while, during which the other node with the key k is deleted and a different thread, T_2, helping the same INSERT operation that T_1 is executing, does find a proper place to insert the key k, and does insert it, but at that point T_1 regains control and changes the descriptor state to erroneously report failure. This sequence of events is bad, because a key has been inserted but failure has been reported. To avoid such a scenario, upon finding a location to insert k, T_2 modifies the operation descriptor to ensure that no stalled thread can wake up and succeed in writing a stale value into the operation descriptor.

Next, we present a race that justifies the setting of the `next` pointer in the new node (Line 46). The INSERT method verifies that this pointer has not been modified since it started the search. This is essential to avoid the following scenario. Suppose Thread T_1 is executing an INSERT of key k and finds a place to insert the new node N

```
1:  public boolean insert(int tid, int key) {
2:    long phase = maxPhase();                                  ▷ getting the phase for the op
3:    Node newNode = new Node(key);                                  ▷ allocating the node
4:    OpDesc op = new OpDesc(phase, OpType.insert, newNode,null);
5:    state.set(tid, op);                                          ▷ publishing the operation
6:    help(phase);            ▷ when finished - no more pending operation with lower or equal phase
7:    return state.get(tid).type == OpType.success;
8:  }
9:
10: private void helpInsert(int tid, long phase) {
11:   while (true) {
12:     OpDesc op = state.get(tid);
13:     if (!(op.type == OpType.insert && op.phase == phase))
14:       return;                                    ▷ the op is no longer relevant, return
15:     Node node = op.node;                            ▷ getting the node to be inserted
16:     Node node_next = node.next.getReference();
17:     Window window = search(node.key,tid,phase);
18:     if (window == null)                             ▷ operation is no longer pending
19:       return;
20:     if (window.curr.key == node.key) {                          ▷ chance of a failure
21:       if ((window.curr==node)||(node.next.isMarked())){                    ▷ success
22:         OpDesc success =
23:           new OpDesc(phase, OpType.success, node, null);
24:         if (state.compareAndSet(tid, op, success))
25:           return;
26:       }
27:       else {                                  ▷ the node was not yet inserted - failure
28:         OpDesc fail=new OpDesc(phase,OpType.failure,node,null);
29:                                  ▷ the following CAS may fail if search results are obsolete:
30:         if (state.compareAndSet(tid, op, fail))
31:           return;
32:       }
33:     }
34:     else {
35:       if (node.next.isMarked()){                  ▷ already inserted and deleted
36:         OpDesc success =
37:           new OpDesc(phase, OpType.success, node, null);
38:         if (state.compareAndSet(tid, op, success))
39:           return;
40:       }
41:       int version = window.pred.next.getVersion();                    ▷ read version.
42:       OpDesc newOp=new OpDesc(phase,OpType.insert,node,null);
43:                                      ▷ preventing another thread from reporting a failure:
44:       if (!state.compareAndSet(tid, op, newOp))
45:         continue;                          ▷ operation might have already reported as failure
46:       node.next.compareAndSet(node_next,window.curr,false,false);
47:       if (window.pred.next.compareAndSet
48:           (version, node.next.getReference(), node, false, false)) {
49:         OpDesc success =
50:           new OpDesc(phase, OpType.success, node, null);
51:         if (state.compareAndSet(tid, newOp, success))
52:           return;
53:       }
54:     }
55:   }
56: }
```

Figure 2.4: The insert operation

in between a node that contains $k-1$ and a node that contains $k+2$. Now T_1 gets stalled for a while and T_2, helping the same INSERT operation, inserts the node N with the key k , after which it also inserts another new node with key $k+1$, while T_1 is stalled. At this point, Thread T_1 resumes without knowing about the insertion of these two nodes. It modifies the next pointer of N to point to the node that contains $k+2$. This modification immediately foils the linked-list because it removes the node that contains $k+1$ from the list. By making T_1 replace the `next` field in N atomically only if this field has not changed since before the search, we know that there could be no node between N and the node that followed it at the time of the search.

Finally, we justify the use of a version for the `next` pointer in Line 47, by showing an ABA problem that could arise when several threads help executing the same insert operation. Suppose Thread T_1 is executing an INSERT of the key k into the list. It searches for a location for the insert, finds one, and gets stalled just before executing Line 47. While T_1 is stalled, T_2 inserts a different k into the list. After succeeding in that insert, T_2 tries to help the same insert of k that T_1 is attempting to perform. T_2 finds that k already exists and reports failure to the state descriptor. This should terminate the insertion that T_1 is executing with a failure report. But suppose further that the other k is then removed from the list, bringing the list back to exactly the same view as T_1 saw before it got stalled. Now T_1 resumes and the CAS of Line 47 actually succeeds. This course of events is bad, because a key is inserted into the list while a failure is reported about this insertion. This is a classical ABA problem, and we solve it using versioning of the next pointer. The version is incremented each time the `next` pointer is modified. Therefore, the insertion and deletion of a different k key while T_1 is stalled cannot go unnoticed. [2]

2.3.5 The Delete Operation

In this section we describe the DELETE operation. Again, a more complicated mechanism is required to safely execute the operation by multiple threads. Most of the problems are solved by a heavy use of the operation record to coordinate the concurrently executing threads. However, an interesting challenge here is the proper report of success or failure of the deletion in a consistent manner. We handle this problem using the `success bit` as described below.

The pseudo-code of the DELETE operation is provided in Figure 2.5. The DELETE operation starts when a thread changes its state descriptor to announce the key that needs to be deleted, and that the current state is *search_delete* (the first stage in the delete operation). The thread that performs this DELETE operation is called the *operation owner*. After setting its state descriptor, other threads may help the delete. The main part of the DELETE operation, which is run in the HELPDELETE method, is

[2] We also implemented a more involved technique for handling this problem, using only a regular Markable Pointer. The full code for this alternative solution is given in Appendix B.

partitioned into two. It starts with the initial state *search_delete* and searches for the requested key. If the requested key is found, then the state is updated to *execute_delete*, while leaving the PRED and CURR pair of pointers in the operation descriptor. From that point and on, there is a specific node whose deletion is attempted. In particular, when the state becomes *execute_delete*, it can never go back to *search_delete*. If the requested key is not found, HELPDELETE will attempt to report failure (Lines 27–28).

As the state becomes *execute_delete* and the node to be deleted is fixed, the second stage is executed in Lines 36–44. The attemptMark method used on the pointer in Line 38 tests that the pointer points to the expected reference, and if so, attempts by an atomic CAS to mark it for deletion. It returns true if the CAS succeeded, or if the node was marked already. In lines 37–39, the thread repeatedly attempts to mark the found node as deleted. After succeeding, it runs a search for the node. Our SEARCH method guarantees that the node of the corresponding DELETE operation is "physically" disconnected from the list. After deleting the node, the state is changed into *determine_delete* (Line 41–43), a special state meaning the operation is to be completed by the owner thread. The deleted node is linked to the operation descriptor, and the method returns.

Helping DELETE is different from helping INSERT in the sense that the help method in this case does not execute the entire DELETE operation to its completion. Instead, it stops before determining the success of the operation, and lets the operation owner decide whether its operation was successful. Note that this does not foil wait-freedom, as the operation owner will never get stuck on deciding whether the operation was successful. When the help method returns, there are two possibilities. The simpler possibility is that the requested key was not found in the list. Here it is clear that the operation failed and in that case the state is changed by the helper to a failure and the operation can terminate. The other possibility is that the requested key was found and deleted. In this case, it is possible that several DELETE operations for the same key were run concurrently by several operation owners and by several helping threads. As the delete succeeded, it has to be determined which operation owner succeeded. In such a case there are several operation owners for the deletion of the key k and only one operation owner can return success, because a single DELETE has been executed. The others operation owners must report failure. This decision is made by the operation owners (and not by the helping threads) in Line 9 of the DELETE method itself. It employs a designated `success bit` in each node. Whoever sets this bit becomes the owner of the deletion for that node in the list and can report success. We believe that this technique for determining the success of a thread in executing an operation in the presence of helping threads can be useful in future constructions of wait-free algorithms.

```
1: public boolean delete(int tid, int key) {
2:   long phase = maxPhase();                                   ▷ getting the phase for the op
3:   state.set(tid, new OpDesc
4:   (phase, OpType.search_delete, new Node(key),null));                              ▷ publishing
5:   help (phase);          ▷ when finished - no more pending operation with lower or equal phase
6:   OpDesc op = state.get(tid);
7:   if (op.type == OpType.determine_delete)
8:                                  ▷ Need to compete on the ownership of deleting this node:
9:     return op.searchResult.curr.success.compareAndSet(false, true);
10:  return false;
11: }
12:
13: private void helpDelete(int tid, long phase) {
14:   while (true) {
15:     OpDesc op = state.get(tid);
16:     if (!((op.type == OpType.search_delete ||
17:       op.type == OpType.execute_delete) &&
18:       op.phase==phase))
19:       return;                                    ▷ the op is no longer relevant, return
20:     Node node = op.node;                          ▷ holds the key we want to delete
21:     if (op.type == OpType.search_delete) {
22:       Window window = search(node.key,tid,phase);
23:       if (window==null)
24:         continue;                          ▷ operation is no longer the same search_delete
25:       if (window.curr.key != node.key) {                    ▷ key doesn't exist - failure
26:         OpDesc failure=new OpDesc(phase,OpType.failure,node,null);
27:         if (state.compareAndSet(tid, op, failure))
28:           return;
29:       }
30:       else {                                  ▷ key exists - continue to executre_delete
31:         OpDesc found = new
32:           OpDesc(phase, OpType.execute_delete, node, window);
33:         state.compareAndSet(tid, op, found);
34:       }
35:     }
36:     else if (op.type == OpType.execute_delete) {
37:       Node next = op.searchResult.curr.next.getReference();
38:       if (!op.searchResult.curr.next.attemptMark(next, true))                          ▷ mark
39:         continue;                          ▷ will continue to try to mark it, until it is marked
40:       search(op.node.key,tid,phase);                  ▷ to physically remove the node
41:       OpDesc determine = new OpDesc
42:         (op.phase,OpType.determine_delete,op.node,op.searchResult);
43:       state.compareAndSet(tid, op, determine);
44:       return;
45:     }
46:   }
47: }
```

Figure 2.5: The delete operation

2.3.6 The Contains Operation

The CONTAINS method does not modify the list structure. Accordingly, some publications claim that it is wait-free even without the use of a help mechanism (see [HS08]). This is not entirely accurate. For example, consider a linked-list of sorted strings. A CONTAINS method traversing it without any help may never reach the letter B, because of infinite concurrent insertions of strings starting with an A. Thus, we provide here an implementation of the CONTAINS method that employs a help mechanism[3].

The CONTAINS operation starts when a thread changes its state descriptor to announce the key it wants to find. It then proceeds to the help method as usual. In the HELPCONTAINS method, a helping thread calls the SEARCH method, and uses a CAS to try to alter the state to a success or failure, depending on whether the wanted key was found. The help mechanism guarantees that the search will not suffer from infinite concurrent insertions of new keys, since other threads will help this operation before entering new keys (perhaps excluding a key they are already in the process of inserting). The pseudo-code for the CONTAINS and the HELPCONTAINS methods is depicted in Figure 2.6. The HELPCONTAINS method differs from the HELPINSERT and HELPDELETE methods in that it doesn't require a loop, as a failure of the CAS updating the state for this operation can only occur if the operation was already completed.

2.3.7 Memory management

The algorithm in this work relies on a garbage collector (GC) for memory management. A wait-free GC does not currently exist. This is a common difficulty for wait-free algorithms. A frequently used solution, which suits this algorithm as well, is Michael's Hazard Pointers technique [Mic04]. Hazard pointers can be used for the reclamation of the operation descriptors as well, and not only for the reclamation of the list nodes themselves.

2.4 A Correctness Proof

In this section we elaborate the proof for correctness and wait-freedom of the algorithm described in Section 2.3, and in particular of its Java implementation in Appendix A. All references to lines of code refer to the implementation of Appendix A. We begin this section with an overview containing only the highlights of the proof. A full proof follows after that.

[3]Technically, for a list of sorted integers, it is possible to easily implement a wait-free contains that does not use the help mechanism since the number of possible keys is bounded. However, this yields a poor bound on the time.

```
1:  public boolean contains(int tid, int key) {
2:      long phase = maxPhase();
3:      Node n = new Node(key);
4:      OpDesc op = new OpDesc(phase, OpType.contains, n, null);
5:      state.set(tid, op);
6:      help(phase);
7:      return state.get(tid).type == OpType.success;
8:    }
9:
10:   private void helpContains(int tid, long phase) {
11:     OpDesc op = state.get(tid);
12:     if (!((op.type == OpType.contains) && op.phase==phase))
13:       return;                                    ▷ the op is no longer relevant, return
14:     Node node = op.node;                         ▷ the node holds the key we need to search
15:     Window window = search(node.key, tid, phase);
16:     if (window == null)
17:       return;                                    ▷ can only happen if operation is already complete.
18:     if (window.curr.key == node.key) {
19:       OpDesc success = new OpDesc(phase, OpType.success, node, null);
20:       state.compareAndSet(tid, op, success);
21:     }
22:     else {
23:       OpDesc failure = new OpDesc(phase, OpType.failure, node, null);
24:       state.compareAndSet(tid, op, failure);
25:     }
26:   }
```

Figure 2.6: The *contains* and *helpContains* methods

2.4.1 Highlights

Basic Concepts and Definitions. The *mark bit*, is the bit on the next field of each node, and it is used to mark the node as logically deleted. A node can be marked or unmarked according to the value of this bit. We define the nodes that are *logically in the list* to be the unmarked nodes that are reachable from the list's head. Thus, a *logical change* to the list, is a change to the set of unmarked nodes reachable from the head. We say that a node is an *infant node* if it has never been reachable from the head. These are nodes that have been prepared for insertions but have not been inserted yet.

In the proof we show that at the linearization point of a successful insert, the inserted value becomes logically in the list and that at a linearization point of a successful delete, a node with the given value is logically deleted from the list. To show this, we look at the actual *physical* modifications that may occur to the list.

Proof Structure. One useful invariant is that a *physical change* to the list can only modify the node's next field, as a node's key is final and never changes after the initialization of a node. A second useful invariant is that a marked node is never unmarked, and that it's next field never changes (meaning, it will keep pointing to the same node). This is ascertained by examining all the code lines that change a node's next field, and noting that all of them do it using a CAS which prevents a change from

taking effect if the node is marked. We next look at all possible physical changes to a node's next field, and show that each of them falls in one of the following four categories:

* Marking: changing the mark bit of a node that is *logically in the list* to true.

* Snipping: physically removing a *marked* node out of the list.

* Redirection: a modification of an infant node's next pointer (in preparation for its insertion).

* Insertion: a modification of a non-infant node to point to an infant node (making the latter non-infant after the modification).

Proving that every *physical change* to a node's next field falls into one of the four categories listed above, is the most complicated part of the formal proof, and is done by induction, with several intermediate invariants. Finally, it is shown that any operation in the *marking* category matches a successful delete operation and any operation in the *insertion* category matches a successful insert operation. Thus, at the proper linearization points the linked list changes according to its specification. Furthermore, it is shown that physical operations in the *Redirection* and *Snipping* categories cause no *logical* changes to the list, which completes the linearizability proof.

To show wait-freedom, we claim that the helping mechanism ensures that a limited number of concurrent operations can be executed while a given insert or delete execution is pending. At the point when this number is exhausted, all threads will help the pending operation, and then it will terminates within a limited number of steps.

2.4.2 General

The linked list interface corresponds to that of a set of keys. The keys considered to be in the set at any given point are the keys found on *unmarked* (see Definition 2.4.17) nodes reachable from the head. An insert(key) method should succeed (return true) and add the key to the set if and only if the key is not in the set; otherwise it should fail (return false). A delete(key) method should succeed (return true) and remove a key from the set if and only if the key is in the set; otherwise it should fail (return false). The contains method is not included in this proof, since it has not changed from previous implementations, and is independent of the rest of the proof.

2.4.3 Definitions

Definition 2.4.1. Head key and Tail key. The Head key is defined to be smaller than all valid keys, and the tail key is greater than all valid keys.

Definition 2.4.2. A threadID (or tid). A threadID is a unique identifier for each thread.

Definition 2.4.3. Operation. An operation is an attempt to insert or delete a key from the list, and is initiated by calling either the insert or the delete method.

Definition 2.4.4. Legal operation. A legal operation is initiated by a thread calling either the insert(tid, key) or delete(tid, key) method with its own tid. (Calling it with a different tid is considered illegal.) Moreover, the key must be strictly greater than the head key, and strictly smaller than the tail key. We assume no illegal operations are attempted.

Definition 2.4.5. Operation phase number. Each operation receives a *phase* number, chosen at the insert or delete method that initiated it. This is the number returned from the maxPhase() method called from the appropriate (insert or delete) method.

Definition 2.4.6. Operation's methods. The insert or delete method that initiated an operation is part of the operation. In addition the search, helpInsert, and helpDelete methods all receive a tid and a phase number as parameters. They are thus considered as a part of the operation that corresponds to this tid & phase pair.

Definition 2.4.7. Operation owner. The operation owner, or the owner thread, is the thread that initiated the operation.

Definition 2.4.8. The operation's node, the operation's key. The (single) node allocated in each insert or delete operation will be called the operation's node. The node can also be said to belong to the operation. Its key will be called the operation's key. At an insert operation, we may also refer to the operation's node as the *inserted node*.

Definition 2.4.9. Successful operation. A successful operation is an operation for which the (insert or delete) method that initiated it returned true.

Definition 2.4.10. Thread's entry. A thread's entry in the state array is the entry in the state array corresponding to state[tid].

Definition 2.4.11. Thread's state. A thread's state is the OpType of its entry in the state array (one of: insert, search_delete, execute_delete, success, failure, determine_delete).

Definition 2.4.12. State's phase number. A state's *phase number* is the phase number present at the phase field of its entry in the state array.

Definition 2.4.13. Pending states. The insert, search_delete and execute_delete states are considered *pending* states. The other states are *non-pending*.

Definition 2.4.14. Pending operation. An operation is considered *pending* if its owner's state is pending with the phase number of the operation.

Definition 2.4.15. Publishing an operation. A thread *publishes* an operation, by (first) changing its state into pending, with the phase number of the operation. (This is done only at the insert and delete methods, and if all operations are legal, can only be done by the operation's owner.)

Definition 2.4.16. List initialization. The list initialization includes all the actions done in the constructor of the list. These operations must all be completed before the initialization of the first operation to the list.

Definition 2.4.17. Mark bit. A node's *mark bit* is the additional bit at the node's next field. A node is considered *marked* if this node is on (set to 1). Otherwise a node is said to be unmarked.

Definition 2.4.18. Reachable node. A *reachable* node is a node reachable from the head. Sometimes we shall specifically mention 'a node reachable from node x', but otherwise reachable means reachable from the head.

Definition 2.4.19. Nodes/Keys logically in the list. The set of nodes logically in the list is the set of unmarked reachable nodes. The set of keys logically in the list is the set of keys that are in the set of nodes logically in the list.

Definition 2.4.20. Logical change. A *logical change* to the list is a change to the set of *unmarked reachable* nodes.

Definition 2.4.21. Physical change. A *physical change* to the list is a change to one of the fields (key, next, or mark bit) of a node.

Definition 2.4.22. Infant node. At any given point, an infant node is a node that was not reachable until that point.

Definition 2.4.23. Node's logical set. A node's logical set is the set of unmarked nodes reachable from it, not including itself.

Definition 2.4.24. Node's inclusive logical set. A node's inclusive logical set is the set of unmarked nodes reachable from it, including itself. (Note that for a marked node, its logical set is identical to its inclusive logical set.)

One final note regarding the definitions: in order to prove correctness, we must also assume the phase number will not overflow its range and thus become negative (or duplicate). When using the long field, this can be done by assuming no more than 2^{63} operations are executed on the list. Although this limit is surely enough for any practical use, we do not want to give any bound to the number of operations, because that will severely limit the value of the wait-freedom definition. Instead, we will assume that if the number of operations is bigger than 2^{63}, the phase field will be replaced with a field of sufficient size.

2.4.4 General List Invariants

Observation 2.4.25. After the list initialization, the head and tail never change, meaning that the head and tail fields of the list always refer to the same nodes.

Observation 2.4.26. A node's key is never changed after its initialization.

Observation 2.4.27. New nodes are always allocated unmarked.

Observation 2.4.28. All nodes, excluding the head, are unreachable at the moment of allocation.

Claim 2.4.29. *A marked node is never unmarked.*

Proof: Changes to a node's next field are made only in lines 18, 39, 75, 138, 140, 174. We shall go over them one by one.
Line 18, node.next is initialized as unmarked.
Line 39, head.next is set to unmarked.
Line 75, a CAS that cannot change the mark is performed.
Line 138, a CAS that cannot change the mark is performed.
Line 140, a CAS that cannot change the mark is performed.
Line 174, attemptMark is made to try and set the mark to true.

Claim 2.4.30. *A marked node's next field never changes.*

Proof: Changes to node.next field are made only in lines 18, 39, 75, 138, 140, 174. We shall go over them one by one.
Line 18, initialization. The node cannot be marked at this point.
Line 39, head.next is set. The head cannot be marked at this point, since this is executed only in the constructor of the list, and marking has not yet taken place.
Line 75, a CAS is performed that checks the node.next to be unmarked.
Line 138, a CAS is performed that checks the node.next to be unmarked.
Line 140, a CAS is performed that checks the node.next to be unmarked.
Line 174, attemptMark is performed. It is a CAS instruction that never changes the reference of node.next, and can only change the node from unmarked to marked.

Claim 2.4.31. *Once a node has become marked, its next field will never change.*

Proof: This follows directly from Claims 2.4.29,2.4.30

Observation 2.4.32. The search method never touches infant nodes. In particular, a window (Pred, Curr) that is returned from the search method never contains infant nodes.

This is correct since the search method only traces objects that are reachable (or were once reachable from the head).

Observation 2.4.33. Throughout the code, window(Pred, Curr) instances are created only in the search method. This means that both Pred and Curr in any window instance are never infant nodes.

We are about to introduce the most important and complicated lemma in the proof. Loosely speaking, this lemma characterizes all the possible physical changes that may be applied to a node. But before introducing the lemma, we need to define those changes.

Definition 2.4.34. Marking, Snipping, Redirection, and Insertion

Marking: the mark bit of the next field of a reachable node is set (from 0 to 1).
Snipping: the execution of an atomic change (CAS) from the state: A->R->B, when R is marked and A is unmarked and reachable, to A->B, when A is still unmarked.
Redirection: loosely speaking, redirection is the operation of preparing a new node A for insertion to the list. It consists of setting its next pointer to point to what should be its next node, B. Formally, a redirection is an atomic change of a node A's next field to point to a node B such that:
(a) B is not an infant (see Definition 2.4.22) at the time the CAS is executed.
(b) B.key > A.key (recall that, by Observation 2.4.26, keys do not change during the execution).
(c) A's logical set (see Definition 2.4.23) at the time the CAS is executed (and before the CAS assignment takes effect) is a sub-set of B's inclusive logical set (see Definition 2.4.24) at the time the CAS is executed.

Insertion: loosely speaking, insertion is the atomic operation that adds a node B into the list by making a reachable node point to it. Formally, insertion is an atomic modification (CAS) of a node A's next field to point to B such that:
(a) A is reachable and unmarked at the time the CAS operation is executed, and also immediately after the CAS assignment takes effect.
(b) B.key > A.key
(c) B is an infant immediately before the CAS (as a result of this CAS, B ceases being an infant).
(d) Immediately before the CAS, A's logical set and B's logical set are identical. (Intuitively speaking, the insertion logically adds B to the list, without making any other logical changes).

Lemma 2.4.35. *After the list is initiated, there are only four possible modifications of a node's next field: Marking, Snipping, Redirection and Insertion, as defined in Definition 2.4.34. (These four possible changes do not include the allocation of nodes.) Furthermore:*
1) Marking can occur only in line 174, and line 174 may result in Marking or have no effect at all.

2) Insertion can occur only in line 140, and line 140 may result in Insertion, Redirection, or have no effect at all.

Proof: The proof is by induction. Before any modifications are made to any node's next field, it is trivially true that all the modifications were one of the allowed modifications. We shall prove that if all the modifications until a particular moment in time were one of the allowed four defined in Definition 2.4.34, then all the modifications made at that moment also fall into that category.
Let T_i be a moment in time, and assume that all modifications of a node's next field before T_i were Marking, Snipping, Redirection or Insertion. We shall prove that all the modifications made at T_i are also one of these four. But before proving it directly, we need several additional claims.

Claim 2.4.36. *Before T_i, an infant node cannot be marked.*

Proof: Before T_i, the only changes possible to a node's next field are the four mentioned above. Of these, only marking can result in a node being marked, and marking can only be done on a reachable (and thus non-infant) node.

Claim 2.4.37. *Before T_i, a reachable node cannot become unreachable while it is unmarked.*

Proof: Before T_i, the only possible changes to the next field of nodes are Marking, Snipping, Redirection and Insertion; none of them will cause an unmarked node to become unreachable:
Marking doesn't change reachability.
Snipping only snips out a single marked node.
Redirection may only add nodes to the set of unmarked reachable nodes of a given node.
Insertion may only add a node to the set of unmarked reachable nodes of a given node.

Claim 2.4.38. *Before T_i, if B is in A's logical set (see Definition 2.4.23) at any given moment, then it will remain in A's logical set as long as it is unmarked.*

The proof is by observing that none of the four possible changes to a next field can invalidate this invariant, and is similar to the proof of the previous claim.

Claim 2.4.39. *Before T_i, a node may only ever point to a node with a key higher than its own.*

Proof by induction: Before the first execution line after the initialization, the only node pointing to another node is head-> tail, and is thus sorted by the definition of the head key and tail key (Definition 2.4.1). By Observation 2.4.26, a node's key is never changed. Before T_i, the only possible changes to a node's next field are Marking, Snipping, Redirection and Insertion. Marking doesn't change the pointed node, and

thus cannot invalidate the invariant. Snipping, by definition, only snips out a node, and thus, by transitivity, if the invariant holds before snipping, it will hold after it. Redirection and Insertion can, by definition, only change a node's next field to point to a node with a higher key.

Corollary 2.1. *Before T_i, the list is sorted in a strictly monotonously increasing order, and there are no two reachable nodes with the same key.*

Proof: This follows directly from Claim 2.4.39.

Claim 2.4.40. *Before T_i, the head is never marked.*

Proof: Changes to a node's next field are only made in lines 18, 39, 75, 138, 140, 174. Looking at these lines, we can see that the only place that a node can become marked is in line 174. In this line, an attempt is made to mark the node that appears as the Curr field in a window. By Observation 2.4.33, this window was originally returned from the search method. In the search method, Curr can only be read from the next field of a node. Before T_i, a node can only ever point to a node with a higher key than its own, by Claim 2.4.39. By Definition 2.4.1, no node can have a key smaller than the head key, so we conclude that before T_i, no node can point to the head, and thus the head cannot be returned as the Curr field in a window by the search method, and thus it cannot be marked.

The following Claim refers to the linearization point of the search method. Loosely speaking, it means that before T_i, the search method works correctly.

Claim 2.4.41. *Before T_i, when calling the search(key) method, if the method returns with a valid (not null) window (Pred,Curr), then during the method's execution there was a point (the search linearization point) in which all the following were true:*
(a) Pred.key < key, and Pred was the last node in the list satisfying this condition.
(b) Curr.key >= key, and Curr was the first node in the list satisfying this condition.
(c) Pred was unmarked.
(d) Curr was unmarked.
(e) Pred.next pointed to Curr.

Proof: We start by proving that Pred.key<key. Pred is initialized in line 69 as the head, and by Definition 2.4.1, head.key < all possible keys. Pred is later modified only in line 84, but the failure of the condition in line 82 guarantees that the new value of Pred.key will remain lower than the key (recall that, by Observation 2.4.26), a node's key never changes so pred.key < key throughout the run. Next, we show that curr.key >= key upon return from the search method. If the search method did not return null, then it must have returned via line 83. The condition in line 82 guarantees that Curr.key >= key. Given that Pred.key<key and Curr.key >= key and since the list is sorted in a strictly monotonously increasing order (by Corollary 2.1), showing (e), i.e., that

Pred.next pointed to Curr, will guarantee the second part of (a), i.e., that Pred was the last node satisfying Pred.key < key, and similarly, the second part of (b), that Curr was the first node satisfying Curr.key >= key. So it remains to show that (e) holds, to conclude that (a) and (b) also hold. We next show that (e), (c), and (d) hold.

The last update of Curr before returning from the search method must happen after the last update of Pred, because whenever Pred is updated, there is always an update to Curr right after. (See Lines 69, 70, and 84 where Pred is modified.) There are three possible cases for when curr was last updated, and for each we will show that (c), (d), and (e) hold:

1. The last update to Curr was in line 70.

Then during the read of Pred's (head) next field, Pred pointed to Curr (e), and Pred was unmarked since, by Claim 2.4.40, the head is never marked (c). Now, if the condition in line 73 were true, then Curr would have been updated again (either in line 79 or again in 70) and thus after the last update to Curr it was false, meaning that Curr wasn't marked at line 72, which happened after line 70. Since a marked node is never unmarked (Claim 2.4.29), then it was also unmarked during the read of Pred's next field in line 70 (d).

2. The last update to Curr was in line 79.

The condition in line 78 guarantees that line 79 can only be reached if the CAS in line 75 (Snipping) succeeds. That CAS changes pred.next field to point to the value that Curr will receive in line 79, and only succeeds if Pred is unmarked. Thus, if we reached line 79, then at the point immediately after that CAS, Pred.next pointed to Curr eventual value (e), and Pred was unmarked (c). Similarly to the previous case, if this is the last update of Curr, then the loop condition in line 73 checked after this update must be false (otherwise there would be another update), and thus Curr was unmarked during the read of line 80, and since a marked node is never unmarked (Claim 2.4.29), then also during the CAS of line 73. (d)

3. The last update to Curr was in line 84.

In line 84 Pred gets the value of Curr, and right after that Curr gets the value of Succ. This Succ value was read either at line 72 or 80, in each case, from Curr.next. So in the execution of line 84, Pred gets the Curr that pointed to the Succ that is now being put into Curr. So during the setting of Succ (line 72 or 80) prior to the last update of Curr in line 84, the eventual Pred pointed to the eventual Curr (e). Also, after the read of Succ (the eventual Curr) either in line 72 or 80, the condition in line 73 is checked, and must be false (otherwise Curr would be updated again), which guarantees that at that point the (eventual) Pred wasn't marked (c). Finally, after the last update of Curr in line 84, curr.next is read again (line 72), and tested again (in line 73) to make sure Curr isn't marked either, and therefore was also not marked at any time before (d).

Claim 2.4.42. *Before T_i, a node's next field never points to an infant node.*

(Note that this isn't entirely trivial, since infant means 'has never been reachable from the head', and not 'has never been reachable from any node.) Proof: Let us see that the four possible changes to a node's next field cannot cause a non-reachable node to point to an infant node. (Making a reachable node point to an infant node will simply cause the node to cease being infant, not invalidate the claim). Marking, Snipping and Insertion specifically define that only the next field of a reachable node can be changed, and Redirection specifically defines that the newly pointed node must be a non-infant.

Corollary 2.2. *Before T_i, an unmarked node that is reachable from any node, is also reachable from the head. (Alternatively, the logical set (Definition 2.4.23) of any node is a subset of the set of nodes logically in the list (Definition 2.4.19).)*

Proof: This follows from Claims 2.4.29,2.4.37,2.4.42.

Definition 2.4.43. A node's maximal set is the set of all the unmarked reachable nodes with a key greater than its own.

Claim 2.4.44. *Before T_i, Redirection (Definition 2.4.34) cannot affect A's logical set (Definition 2.4.23) if A's logical set is already maximal prior to the Redirection.*

Proof: By definition, Redirection can only add nodes to a node's logical set. But it is impossible to add nodes to it, if it is already maximal, since by Corollary 2.2 and Definition 2.4.43, a node's logical set is always a subset of its maximal set.

Corollary 2.3. *An unmarked node that is not an infant is reachable and thus logically in the list.*

Proof: This follows from Claim 2.4.37.

Claim 2.4.45. *Before T_i, a Redirection of the next field of a node A cannot change the logical set of any non-infant node B.*

Proof: If A is an infant itself, then it cannot be reachable from any node (Claim 2.4.42), and thus redirection on its next field can only affect its own logical set, and since A is an infant, this is allowed. If A is marked, then by definition, Redirection cannot be applied on its next field anyway. If A is non-infant and unmarked, then it is reachable (Corollary 2.3). So, since the list is sorted (Corollary 2.1), all the unmarked reachable nodes with a key greater than A.key are reachable from A. By definition, Redirection can only increase A's logical set, but there are no keys larger than A's key in the list. Also, redirection cannot be made to point to an infant (i.e., unreachable) node. Thus, redirecting A's next pointer cannot change the logical set of A.

We are now ready to show that any modifications to a node's next field at time T_i are restricted to Marking, Snipping, Redirection, and Insertion, and thus conclude the

proof of Lemma 2.4.35.
Changes to a node's next field are made only in lines 18, 39, 75, 138, 140, 174. We shall go over them one by one:
Line 18: The allocation of a new node is excluded from the statement of the Lemma.

Line 39: This line is only executed during the list initialization, and is thus also excluded from the Lemma assertion.

Line 75: This instruction line is Snipping, which is inside the search method. For the change to take place, the CAS must succeed. Let us verify that in this case all the Snipping requirements are met:
Pred, Curr, and Succ nodes of the search method are here A, R, and B of the snipping definition. We need to show that the CAS is from state Pred->Curr->Succ when Pred is unmarked and reachable and Curr is marked, to Pred->Succ, when Pred is still unmarked and reachable. The condition in line 73 guarantees that the Curr was marked. Claims 2.4.29 and 2.4.31 guarantee that once marked, it will remain marked, and that its next field will never change. Thus, if the CAS in line 75 succeeds, we know for certain that before its execution the state was Pred->Curr->Succ (that CAS checks Pred->Curr, and Curr->Succ is guaranteed by Claim 2.4.31), that Curr was marked (Claim 2.4.29), and that Pred wasn't marked (the CAS verifies this). Also, since the search method never reaches infant nodes (Observation 2.4.32) and Pred is unmarked, then Pred is reachable (By Corollary 2.3). Thus, after the execution, the state is Pred->Succ, Curr is still marked, and Pred is still not marked, and thus also surely reachable (Claim 2.4.37). Also note that A is surely reachable: this is true because line 75 is inside the search method, which never reaches infant nodes and A is also unmarked. We conclude that it is a legal Snipping, and thus line 75 can either do nothing or a legal Snipping.

Line 138: This instruction line is Redirection, which is done inside the helpInsert method. Let us see that if the CAS succeeds, then all the redirection requirements are met: this CAS attempts to set the inserted node's (see Definition 2.4.8) next field to point to window.curr. Window is the value returned from the search method called in line 110. (This search is for the operations's key.) We need to show:
(a) Window.curr is not an infant. (This is immediate from Observation 2.4.33.)
(b) Window.curr.key > operation's key.
(c) Immediately before the CAS, the inserted node's logical set is a subset of the Window.curr inclusive logical set (Definitions 2.4.23, 2.4.24).

(a) is immediate, as stated above.
(b) The search method linearization claim (Claim 2.4.41) guarantees that Window.curr.key >= the operation's key. The condition in line 113 guarantees that window.curr.key != operation's key, otherwise line 138 wouldn't have been reached, so window.curr.key

must be larger than the operation's key.
(c) Now, line 138 is trying by a CAS to replace the inserted node's next with window.curr, and it compares the inserted node's next to the value node_next read in line 109, before the search method that returned the window is called. If node_next is null and the CAS succeeds, then the set of unmarked reachable nodes from the inserted node immediately before the CAS is the empty set, trivially fulfilling the condition. So we shall assume node_next is not null. By Claim 2.4.42, node_next is also not an infant node. By Claim 2.4.41 (the search method linearization claim), there was a point, the search linearization point, at which Window.curr was the first unmarked reachable node in the list, with a key >= the operation's key.
We will prove (c) by claiming following: Before T_i, suppose there is a point in time in which two nodes A and B satisfy that there exists a key K such that:
1. Neither A nor B are infants.
2. Both A.key and B.key >= K.
3. A is the first unmarked reachable node in the list satisfying A.key >= K.
Then B's logical set will always be a subset of A's logical set (as long as none of them is reclaimed). This is true by induction: at the search linearization point, the set of unmarked nodes reachable from A is the maximum possible set for nodes with a key greater than K, and thus, B's logical set is surely a subset of this set. The four possible changes to a node's next field before T_i:
Marking: Only affects the mark of the marked node, and not the reachability of any node from any node. This clearly can't produce an unmarked reachable node from B which is not reachable from A.
Snipping: Only snips out a marked node, and doesn't affect reachability of any unmarked nodes.
Redirection: Since neither A nor B are infant, by Claim 2.4.45 Redirection cannot change the set of unmarked reachable nodes from them.
Insertion: Two Cases:
1. At the search linearization point, B is marked; thus, its next field cannot be changed (by Claim 2.4.31), and in particular cannot be changed by insertion. So the only way to add a node to B's logical set is by insertion (changing the next field of one of the unmarked nodes reachable from B), but all these nodes are also reachable from A, and thus this will also add this node to the set of unmarked nodes reachable from A.
2. At the search linearization point, B is unmarked. Since B is not an infant, it is reachable from the head, and since A was the first unmarked node with a key greater than K in the list, then B was reachable from A. Since while B is unmarked it will remain reachable from A (by Claim 2.4.38), then also changing B's next field by insertion directly will add the new node to the set of nodes reachable from A. Once B is marked and its logical set is still a subset of A's logical set, we are back to case 1.

Line 140: This line is inside the helpInsert method as well. The instruction in it

is normally an Insertion, and can sometimes be a (private case of) Redirection. Let us see that if the CAS succeeds, either all Insertion or all Redirection requirements are met.
This line contains a CAS that changes window.pred to point to the inserted node. Window was the search result for the operation's key in line 110. Note that this CAS also checks for version. To prove a valid Insertion, we need to show that all the following are true:
(a) window.pred is reachable and unmarked immediately before the CAS, and also immediately after the CAS.
(b) The operation's key $>$ window.pred.key
(c) The inserted node is an infant immediately before the CAS.
(d) Immediately before the CAS, window.pred's logical set is identical to the inserted node's logical set.
Note that if (c) is not fulfilled, then this is a legal Redirection, so we will focus on proving (a), (b), (d).
(a) window.pred is clearly not an infant since it was returned by the search method. The CAS makes sure it is unmarked and, by Corollary 45, reachable.
(b) This is immediate from Claim 2.4.41 since window is the result of the search for the operation's key in line 110.
(d) In the CAS of line 140, we compare window.pred.next pointed node to the one previously read from the inserted node's next field. If the inserted node's next field hasn't changed between its reading and the CAS, then immediately before the CAS both the inserted node's next field and the window.pred.next point to the same node, so obviously the set of unmarked nodes reachable from both is identical. If the node pointed by the inserted node's next field has changed, then the set of unmarked nodes reachable from it could only have grown, since before T_i all changes to a node's next field can only add to the set of unmarked nodes reachable from it. However, window.pred is unmarked and reachable, its logical set is the maximum, and thus the two sets must still be equal.
Now, recall that we don't need (and can't) prove item (c), since line 140 can be either Insertion or Redirection. Note that if it is a Redirection, it is a futile one by Claim 2.4.45 (meaning that it doesn't change the logical set of any node). In general, all redirections changing the next field of a non-infant node are futile (unwanted, but harmless).

Line 174: This line contains the Marking instruction, and it is inside the helpDelete method. In this line we attempt to mark the next field of a node stored in the searchResult.curr field of a state entry. This field (the searchResult of the OpDesc class) can only be written to a value different than null in line 168. In line 168 this field(the searchResult of the OpDesc) receives the result of a search method. Thus, since the search method doesn't return in its window nodes that are infants (Observation 2.4.32), we know for sure that this is an attempt to mark a non-infant node. If it is already

marked, then this line cannot possibly make any difference. If it is not, then by Corollary 2.3 this node is reachable, and thus this is a CAS to mark the next field of a reachable node, and thus a legal marking.

To conclude, we have seen that if all modifications to a node's next field before T_i are due to Marking, Snipping, Redirection or Insertion, then all modifications at T_i also belong to one of these categories, and we have finished proving Lemma 2.4.35.

Corollary 2.4. *All the claims used during the proof of Lemma 2.4.35 hold throughout the run (And not only 'before T_i').*

Proof: A direct result of proving Lemma 2.4.35. For the rest of the proof we shall treat those claims in their general form.

Corollary 2.5. *Insertion and Marking (as defined in Definition 2.4.34) are the only logical changes to the list, when Insertion adds exactly one node (the inserted node) into the list, and Marking removes exactly one node (the marked node) from the list.*

Proof: This is a direct result of Observation 2.4.26 (a node's key is never changed), and of Lemma 2.4.35.
Snipping: Only changes the reachability of a marked node, and thus makes no logical changes to the list.
Redirection: According to Claim 2.4.45, it is clear that Redirection cannot make logical changes to the list.
Marking: Since we mark a reachable node, it is clear this takes this node logically out of the list.
Insertion: By definition Insertion inserts a previously infant node, while making no other changes to the set of unmarked reachable nodes.

Notes about parallel CASes that happen at exactly the same time:
a. A node cannot be marked more than once even at the same moment, since this is done by a CAS on its next field. So it is safe to assume that each marking has the effect of logically removing a distinct node from the list.
b. The same node cannot be inserted more than once even at the same moment, since at any given moment the node has only one possible place in the list, and thus the Insertion CAS is on a specific field. So it is safe to assume that each Insertion has the effect of logically adding a distinct node to the list.
c. It can easily be shown that the same node cannot be marked and inserted at the same moment (it must be inserted before it is marked), but this is not necessary for the point of our discussion.

Claim 2.4.46. *An infant node can only cease being infant via its Insertion.*

Proof: By Claim 2.4.42, no node can ever point to an infant node, and thus, the first time a node is pointed to is when it stops being infant, meaning that the first time it is

pointed to it is from a node reachable from the head. This first time it is pointed to must be when one of the four possible changes occurs:
Marking doesn't change the pointed node, and thus the change that causes the node to cease being infant cannot be Marking.
Snipping by definition changes a node's next field to point to a node that was already pointed to by another node before, and thus it cannot be the first time a node is pointed to.
Redirection by definition changes a node's next field to point to a node that is not an infant, and thus it cannot be the change that causes a node to cease being infant.
So the first time a node is pointed to by the next field of any node can only be when an Insertion occurs. This Insertion makes the node reachable from the head, and thus no longer infant.

Claim 2.4.47. *An infant node is never marked.*

Proof: Of the four possible instructions that modify a node's next field, the only one modifying the mark of a node is Marking, which is done on a reachable (and thus non-infant) node by definition.

Observation 2.4.48. The fields of an Operation Descriptor (opDesc in the code) are final. That is, they are never changed after initialization.

Claim 2.4.49. *After initialization, a non-pending state (success, failure or determine_delete) for a thread cannot be altered by any thread other than itself.*

Proof: The state array changes in lines 49, 56, 117, 123,130,136,142,163,169,179.
Lines 49 and 56: These lines are inside the insert method and the delete method. Both methods are only called by the operation owner thread to initiate the operation. By definition of legal operations (Definition 2.4.4), both are only called with the tid of the thread owner. Thus, both can only alter the state of the running thread.
Lines 117,123,130 and 136 are inside the helpInsert method. They contain a CAS that only succeeds if the operation is the one read in line 105. The condition in line 106 guarantees that it is a pending (insert) operation, and Observation 2.4.48 guarantees that it will remain so.
Line 142: This line contains a CAS and only succeeds if the old operation is the one created in line 133. This operation is a pending (insert) state.
Lines 163,169,179: These lines are inside the helpDelete method, and they use a CAS that compares the given parameter to the value read at line 151. The condition in line 152 guarantees that the state in that case is pending (either search_delete or execute_delete).

Claim 2.4.50. *Each thread can execute at most one operation with a given phase number. This means that a pair consisting of a threadID (Definition 2.4.2) and a phase number (Definition 2.4.5) uniquely identifies an operation.*

Proof: By definition of legal operations (Definition 2.4.4), the insert and delete methods, which initiate operations, can only be executed with a threadID matching the thread that runs them. So for any given threadID, the operations are executed sequentially, one after the other. When a thread calls the maxPhase method twice in succession, then this method is guaranteed to return two different phaseIDs, since each call increases the maxPhase integer by at least one during the run of each maxPhase. Note that if the CAS that increases this integer fails, then it must hold that another thread has incremented this number, as all modifications to this number (except for its initialization) are increments.

Claim 2.4.51. *A non-pending operation (Definition 2.4.14) cannot revert to pending.*

Proof: We have already seen in the proof of the previous claim that a non-pending operation can only be changed inside the insert or delete methods (which are only executed by the owner thread and not by helper threads). But these methods never change an operation's state to non-pending; they only create a new state operation, which will have a different phaseID.

Claim 2.4.52. *A search method might only return a null window if its operation (Definition 2.4.6) is no longer pending.*

Proof: Immediate from the condition in line 76.

2.4.5 The Insert Operation

Recall first the definition of Insertion (Definition 2.4.34). Note the difference between an Insertion, which is a single CAS that inserts a node, and an insert operation (Definitions 2.4.3,2.4.6), which consists several methods that may be called by several different threads, and is initiated when the owner thread of the operation calls the insert method (Definition 2.4.7). Also recall that a successful operation is one for which the method that initiated it returned true (Definition 2.4.9). In this part of the proof, we want to establish a connection between Insertions and insert operations. In particular, we will show a one-to-one correspondence between Insertions and successful insert operations. We will use the (tid, phase) pair as a connector between them. First, we shall define four functions.

Definition 2.4.53. The Insert Functions - A,B,C,D

Function A: Insertion -> (tid, phase) pair.
Matches each Insertion to the (tid, phase) pair that were that parameters for the helpInsert method that the insertion was a part of. (Recall that, by Lemma 2.4.35, Insertions may only occur in line 140).

Function B: insert operation -> (tid, phase) pair.

Matches each insert operation to a (tid, phase) pair, such that the tid is that of the owner thread (which by definition of legal operations is also the tid parameter of the insert method), and the phase is the number returned from the maxPhase() method invoked inside the insert method that initiated this operation.

Function C: Insertion -> insert operation. C(x) = B^{-1}(A(x)).

Function D: insert operation -> Insertion or NULL.
D(y) = C^{-1}(y) if defined
or NULL otherwise

Claim 2.4.54. *A thread whose state is other than insert can only reach the insert state in an insert method called by the same thread.*

Proof:
Line 49: This line is indeed inside the insert method. It changes the state for the given tid and, by definition of legal operations (Definition 2.4.4) the given tid must be that of the running thread.
Line 136: This line contains a CAS that can only succeed if the current state of the operation owner is the one read in line 105, and the condition in line 106 guarantees that the state of the operation owner is already an insert.
The rest of the lines never attempt to write an operation with the insert state.

Claim 2.4.55. *Function A (Definition 2.4.53) is an injective function.*

Proof : Let x be an insertion. By Lemma 2.4.35, we know that Insertion can only take place in a successful CAS in line 140, so we know that x took place in line 140. In this line an attempt is made to insert the node read in line 108 from the variable op into the list. The condition in line 106 guarantees that this op has the same (tid, phase) pair as A(x). Claim 2.4.50 guarantees that there is no other operation with the same (tid, phase) pair. The node read in line 108 is the operation's node (Definition 2.4.8) allocated in the insert method, so if another Insertion x' exists such that A(x) = A(x'), then both Insertions are inserting the same node. But by definition of Insertion, it is inserting an infant node into the list, and immediately after that the node is no longer infant. So two Insertions of the same node cannot happen at two different times. Two Insertions of the same node also cannot happen at the same moment because the list is sorted (Corollary 2.1), and thus at a single moment a node can only be inserted into a specific place in the list. So two simultaneous insertions of the same node must execute a CAS on the same predecessor for this node, which cannot be done at the same time. We conclude that each node can only be inserted (via Insertion) once, and thus two distinct Insertions must insert two distinct nodes, and thus have two distinct (tid, phase) pairs.

Claim 2.4.56. *Function B is an injective function.*

This follows directly, and is a private case of Claim 2.4.50.

Claim 2.4.57. *Function C is defined for every insertion, and is injective.*

Proof: C(x) is defined as B^{-1}(A(x)). We should first note that B^{-1} is well defined. We know this from Claim 2.4.56 that B is injective. B^{-1} is thus also injective, and By Claim 2.4.55 A is also injective. So C is injective as a composition of injective functions. We still need to show that C is defined for every Insertion x. A is defined for every Insertion x, but B^{-1} is most certainly not defined for every (tid, phase) pair. However, B is defined for all insert operations, and thus B^{-1} is defined for all (tid, phase) pairs that match an insert operation. This is all we need, since for every insertion x, A(x) is indeed a (tid, phase) pair that matches an insert operation. This is true since Insertion only happens at the helpInsert method, and helpInsert is only called in line 94, when the condition in line 93 guarantees that the (tid, phase) pair matches a state of a (pending) insert. Claim 2.4.54 guarantees that this can only be the case if the (tid, phase) pair matches an insert operation.

Claim 2.4.58. *Function D is well defined.*

Proof: This is true since C is an injective function (Claim 2.4.57).

Claim 2.4.59. *A helpInsert method will not be finished while its operation is pending.*

Proof : The HelpInsert method is comprised of an infinite loop and can only be ended in one of the following lines:
107: The condition in line 106 guarantees this can only happen if the insert operation is no longer pending.
112: The condition in line 111 guarantees this can only happen if the search method returned null, which can only happen if the operation (given to it) is no longer pending (Claim 2.4.52)
118: The condition in line 117 guarantees this can only happen if the state was successfully changed to non-pending in the same line (117).
124: The condition in line 123 guarantees this can only happen if the state was successfully changed to non-pending in the same line (123).
131: The condition in line 130 guarantees this can only happen if the state was successfully changed to non-pending in the same line (130).
143: The condition in line 142 guarantees this can only happen if the state was successfully changed to non-pending in the same line (142).

Corollary 2.6. *A help(phase) method will not finish while a pending insert operation with the same phase number exists.*

This is an immediate conclusion by the structure of the help method (it calls the helpInsert method) and Claim 2.4.59.

Corollary 2.7. *An insert method will not be finished while the operation initiated by it is still in a pending state.*

This is an immediate conclusion by the structure of the insert method (it calls the help method) method and Corollary 2.7.

Claim 2.4.60. *A thread in a pending inert state can only reach a different state in the helpInsert method.*

Proof: According to Corollary 2.7, the insert and delete methods cannot change this state since while the insert is pending the owner thread hasn't yet finished the insert operation. The helpDelete method cannot change this state since every change of a state in it is by a CAS that ensures it only changes a pending delete (search_delete or execute_delete) state. The rest of the lines that change a state are only inside the helpInsert method.

Claim 2.4.61. *A thread's state can only be changed into success in the helpInsert method, and only if the insert's operation's node is no longer an infant.*

Proof: A success state can only be written (throughout the code) in the helpInsert method. We will go over each of the lines that change a state into success and see that they can only be reached if the operation's node is no longer an infant. A success state can be written in the following lines:
Line 117: The condition in line 114 guarantees that this line can only be reached in one of two cases:
1. The operation's node is marked, and thus, by Claim 2.4.47 is not an infant.
2. The node was returned inside the window that the search method returned, and thus is not an infant (Observation 2.4.32).
Line 130: The condition in line 128 guarantees this line can only be reached if the inserted node is marked and thus non-infant (Claim 2.4.47).
line 142: The condition in line 140 guarantees this line can only be reached if the CAS in line 140 succeeded. By Lemma 2.4.35, if this CAS succeeds it is either an Insertion of the inserted node (which thus ceases being an infant node), or a Redirection, and thus the inserted node is already not an infant, by definition of Redirection (Definition 2.4.34).

Claim 2.4.62. *For each successful insert operation (recall successful means returned true) denoted y, D(y) is not NULL.*

Proof : Another way to formulate this claim is that for each insert method that returned true, an insertion took place during the insert operation. The structure of the insert

method guarantees that it returns true if and only if the state of the owner will be changed into success. By Claim 2.4.61, it means that the operation's node is no longer an infant. By Claim 2.4.46, this implies that a corresponding Insertion took place.

Claim 2.4.63. *The key added to the list as a result of an Insertion (the Insertion is denoted x) is identical to the key given as a parameter to the insert method that initiated C(x).*

Proof: The key added to the list as a result of an Insertion x is the key that is on the inserted node, which is the operation's node read from the state in line 108. The operation's node for an insert operation is created in line 47, with the key given to the insert method.

Claim 2.4.64. *Immediately before insertion, the key to be inserted is not in the list.*

Proof: By Corollary 2.1 a valid insert retains the strict monotonicity of the list, and therefore the key cannot exist in the list during Insertions.

Claim 2.4.65. *For an unsuccessful insert operation (meaning that the insert method that initiated it returned false), denoted y, D(y) = NULL.*

Proof: An unsuccessful insert operation can only happen if the pending insert operation changed to something other than success. By Claim 2.4.60, this can only happen in the helpInsert method. By Corollary 2.7, the insert operation will not be finished while the state is still pending, and if the state changed to success, the operation will not fail. Since for an insert operation y, d(y) can only be an Insertion in which the inserted node is the operation's node, it is enough to show that the operation's node in a failing operation can never be inserted. A word of caution: it is not enough to show that when the insert operation ceases to be in a pending state, the node of that operation is still an infant. We also need to show that it cannot possibly be inserted later by other threads currently inside a helpInsert method for the same operation. Let us go over the changes of the state inside the helpInsert method. For each one, we shall see that one (and only one) of the following holds:
1. It changes the state to success, and thus the operation will result in being successful, not relevant here (lines 117, 130, 142).
2. It changes the state but to a state that is still a pending insert, and thus the insert method must still be in progress and cannot (yet) return false (line 136).
3. It changes the state to failure, but we can show that the node that belongs to the operation is certainly an infant and also cannot be inserted later(line 123).
In line 110 a search is done for the operation's key. The condition in line 113 guarantees that line 123 can only be reached if the search method found a node with the same key. The condition in line 114 guarantees that line 123 can only be reached if that node is not the operation's node, and also that the operation's node is not marked (at least not

at this time). So, during the search of line 110 we know that there was a point, the search linearization point, at which:
1. The operation's node was not in the list.
2. The operation's node was not marked. (Claim 2.4.29)
3. A different node with the same key was in the list (we will call it the hindering node).
1 and 2 together mean that the operation's node was infant at the time (using Claims 2.4.37, and also 2.4.29 again). Now, if an Insertion of the operation's node is to take place, then it must be in a concurrent thread running helpInsert of the same operation after reading op in line 105. (If it did not yet read op, it will find the operation no longer pending, and will return from the helpInsert method.) Now, before this concurrent thread gets to the Insertion (line 140), it must also change the state in a (successful) CAS in line 136. There are two possible cases:
1. The CAS of the failure in line 123 takes place before the concurrent CAS in line 136. But then the CAS in line 136 cannot be successful, because it will compare the current state to the obsolete state read in line 105.
2. The CAS of the failure in line 123 takes place after the concurrent CAS in line 136. Then, in order for the CAS of the failure in line 123 to succeed, it must have read the old state after the CAS of the concurrent thread in line 136. This also means that it reached line 110, and the search point, only after the concurrent thread read the version of its window.pred in line 133. Now, if at the time the concurrent thread read the version in line 133, the hindering node was already in the list, then the window.pred could not point past it (since the keys are sorted), and thus, the CAS of the Insertion in line 142 must have failed. It can only succeed if the pointer in window.pred hasn't changed, and also it points to a node equal to one read from the operation's node next field, which must have a key greater than the (identical) key of the hindering node and the operation's node. If at the time the concurrent thread read the version in line 133 the hindering node was not yet in the list, and window.pred.next pointed to a node with a key greater than the operation's key, then the hindering node must be later inserted into the list, and this must change the next field of the window.pred, advancing the version, and ensuring that the insertion CAS in line 140 cannot succeed (indeed, this is the reason why we needed the version in the first place).

Claim 2.4.66. *For every Insertion x, the CAS operation that caused it occured during the execution time of the insert method that initiated C(x).*

(This claim is necessary to show that this Insertion is a legal linearization point for the insert method.)
Proof: For the insert operation y = C(x), we know that D(y) = x, which means by Claim 2.4.65 that y was a successful insert, and returned true. That can only happen if the state of the owner thread was success, which by Claim 2.4.61, can only happen if the Insertion x has already taken place. The Insertion cannot take place before the insert operation C(x) starts, because the (tid, phase) pair of a state is created only at

the insert method that initiates the operation.

Claim 2.4.67. *An insert operation can only fail if during a search method belonging to that operation, another node with the same key was logically in the list (at the linearization point of that search method).*

Proof: We have seen in the Proof of Claim 2.4.65 that an unsuccessful insert operation can only be the result of a CAS changing the state of the operation to failure in line 123. The combined conditions of lines 113 and 114 guarantee that this line can only be reached if an appropriate hindering node, with the same key, was returned from the search method of that operation that was called in line 110.

Lemma 2.4.68. *An insert method that finished 'works correctly', meaning that one of the following has happened:*

* *It returned true, inserted the key, and at the point of linearization no other node with the same key existed.*

* *It returned false, made no logical changes to the list, and at the point of linearization another node with the same key existed.*

Proof: If the insert method that initiated an operation denoted y returned true (i.e., the operation was successful), then by Claim 2.4.62 D(y) is a corresponding Insertion, which happened during the execution time of the insert (Claim 2.4.66), which inserted a key that was not in the list at that time (by Claim 2.4.64). If the insert method returned false, then by Claim 2.4.66 it corresponds to no insertion, and by Claim 2.4.67, another node with the same key existed at the linearization point of a search method that belonged to this operation, and this point is also defined as the linearization point of the insert operation.

2.4.6 The Delete Operation

Recall first the definition of Marking (Definition 2.4.34), and the definition of a delete operation (Definitions 2.4.3,2.4.6). Also recall that a successful operation is one for which the method that initiated it returned true (Definition 2.4.9).

Claim 2.4.69. *A thread's state can only be changed into search_delete in the delete method called by the same thread.*

Proof: Line 56 is inside the delete method, and indeed changes a thread's state into a search_delete. By the definition of legal operations (Definition 2.4.4) this can only be called by the same thread. The other lines that change a thread's state (49, 117, 123,130,136,142,163,169,179) never attempt to make the state a search_delete.

Claim 2.4.70. *A thread can only reach the execute_delete state directly from the search_delete state.*

Proof: An attempt to set a state to execute_delete is made only in line 169 using a CAS. The condition in line 156 guarantees that this CAS may only succeed if the value it is compared to (the previous state) is a search_delete.

Claim 2.4.71. *A delete method will not finish while the operation it belongs to is pending.*

Proof: The delete method is constructed so that it publishes a (pending) delete (using the search_delete state), and then calls the help method. The help method loops through the state array. If by the time it reads the state of the owner thread of the delete operation it is no longer pending, there is nothing left to prove. If it is still pending, it will call the helpDelete method. If so, we can now refer to the helpDelete method, which was called as part of this delete operation by the operation owner. This helpDelete method is constructed by an infinite loop that may only exit at one of the following lines:
line 154: In which case the condition in lines 152-153 guarantees that the operation is no longer pending. (By Claim 2.4.51 it cannot return to a pending state.)
line 164: In which case the condition in line 163 guarantees the operation is changed to no longer being pending.
line 180: The condition in line 172 guarantees that line 180 can only be reached if the operation was at state execute_delete. There are two possible cases for this.
1. When the helpDelete method called by the operation owner reached line 179, the CAS succeeded, and thus in line 180 the operation is no longer pending.
2. When the helpDelete method called by the operation owner reached line 179, the CAS didn't succeed. This can only happen if some other thread changed the state. But a different thread could not have done it in the insert or delete method, since those can only be called by the operation owner (by definition of legal operations). All other changes to a state are by means of a CAS. The only one that can possibly change an execute_delete state is the one in line 179, which would have made the state determine_delete and no longer pending. Other lines cannot be reached if the value that is compared to is an operation with a state of execute_delete. (In other words, there is no need to check that the CAS in line 179 succeeded, because it can only fail if another thread already executed the same CAS.)

Claim 2.4.72. *An execute_delete can only be changed into a determine_delete state, and only in a CAS in line 179.*

This is an immediate result of the proof of the previous claim. We shall briefly reiterate the relevant parts.
An execute_delete state cannot be changed inside the delete or insert methods, since in these methods a thread only changes its own state (by definition of legal operations), but, according to the previous claim, the owner thread will not finish the delete method that initiated this operation while the operation is pending. The remaining changes to

a state are executed by a CAS, and the only CAS that compares the previous value to a state with execute_delete is the one in line 179, which changes it into a determine_delete.

Definition 2.4.73. Possible routes of a delete operation's state.

Route 1: published search_delete in line 56 -> CAS into failure in line 163.
Route 2: published search_delete in line 56 -> CAS into execute_delete in line 169 -> CAS into determine_delete in line 179.

Claim 2.4.74. *The state of any delete operation from publishing until it is not pending can only follow one of the two routes in Definition 2.4.73.*

Proof:
Fact: A pending delete operation's state cannot be changed outside the helpDelete method.
This is because inside the helpInsert there is a CAS that checks that the operation is a pending insert operation, and the changes in the delete and insert methods cannot be made since the operation is still pending. Using this fact, we can just focus on the changes inside the helpDelete method.
Line 163 is a CAS that leads to failure, and the condition in line 156 guarantees the previous state is search_delete.
Line 169 is a CAS that leads to execute_delete, and the condition in line 156 guarantees the previous state is search_delete.
Line 179 is a CAS that leads to determine_delete, and the condition in line 172 guarantees the previous state is execute_delete.

We shall now define two functions that will correlate between delete operations that followed route 2, and Marking, as defined in Definition 2.4.34. Using a process similar to the one we used in our proof of the insert operation, we wish to prove a one-to-one correspondence between successful delete operations and Markings.

Definition 2.4.75. The Delete Functions A,B

Function A: Delete Operations that followed route 2 -> Marking.
For a delete operation that followed route 2 (as defined in Definition 2.4.73), denoted y, the operation was at some point in a state of execute_delete. At that point, there was a window stored in the searchResult field of that operation descriptor. (The condition in line 158 guarantees that a state of execute_delete always contains a valid (not null) window in the searchResult field.) We say that A(y) is the Marking of the node that was stored in searchResult.Curr. (We shall prove immediately that this defines a single Marking for every delete operation that followed route 2.)

Function B : Marking -> Delete Operation that followed route 2.

We say for a marking, denoted x, that : B(x) = y if and only if both of the following are true:
1. A(y) = x.
2. y was a successful delete operation (returned true).
We shall prove soon that function B is well defined and injective.

Claim 2.4.76. *A is defined for every delete operation that reached the execute_delete state, denoted y, and the Marking A(y) always takes place during the run of the operation y. (It always takes place between the invocation of the delete method that started it, and the end of the same delete method.) Furthermore, A(y) is a Marking of a node that has the same key as was given as a parameter to the delete method that initiated the (y) operation.*

Proof:
Part One: A(y) matches every delete operation that reached the execute_delete state to at least one Marking that executed during its run:
By Claim 2.4.74, execute_delete can only be changed in a CAS in line 179. The condition in line 174 guarantees that this CAS can only be reached if the node found at the op.searchResult.curr is marked. The searchResult window was returned from a search method called in line 157. By Claim 2.4.41, there was a point in time that this node was unmarked, and this certainly happened during this delete operation. The condition in line 160 guarantees that the execute_delete state would only have been reached if the searchResult.curr.key equalled the operation's key.
Part Two: A(y) matches every delete operation to no more than one Marking:
By Claim 2.4.74, execute_delete can only be reached once (with a specific operation descriptor) in a delete operation. The Marking can only be done on the op.searchResult.curr. This is only a single node, and each node cannot be marked more than once (Claim 2.4.29). Thus execute_delete correlates to no more than one Marking.

Claim 2.4.77. *Function B matches each Marking, denoted x, to a single and distinct delete operation. (By the definition of Function B (Definition 2.4.75), it also follows that this delete operation returned true.)*

Proof: Each delete operation that reached the execute_delete state matches the Marking of the node found in the op.searchResult.curr, and its state can only be changed into determine_delete (Claims 2.4.74,2.4.76). In the delete method belonging to this operation, after the help method is done the operation is no longer pending, and thus it must have reached the determine_delete at that point. Then all the delete operations that reached execute_delete can only be exited in line 61. Line 61 contains a CAS on the op.searchResult.curr.d, trying to change it from false to true. Since this field is initiated as false, and is never modified apart from this line, then no more than one operation can succeed on this CAS, but if at least one of them tried, then at least (and exactly) one must succeed.

Claim 2.4.78. *For a successful delete operation denoted y, there exists a Marking x of a node with the same key as the operation's key (see Definition 2.4.8), satisfying B(x) = y.*

Proof: A successful delete operation can only go by route 2, since route 1 always ends with a failure by Definition 2.4.73. By Claim 2.4.76, we conclude that A(y) = x is a Marking of a node with the operation's key. Since the delete operation was successful, and A(y) = x, then by definition of function B, B(x) = y, and by Claim 2.4.77, B is well defined.

Claim 2.4.79. *A delete operation that followed route 1 made no logical changes to the list.*

Insertion can only take place inside the helpInsert method, which cannot be reached in a delete operation. Marking can take place only in line 174 (By Lemma 2.4.35), but the condition in line 172 guarantees that this line can only be reached if the state of the operation was at some point execute_delete, which means this delete operation followed route 2.

Claim 2.4.80. *A delete operation can only follow route 1 if at some point during its execution there is no node in the list with a key equal to the operation's key.*

Proof: Route 1 requires a CAS in line 163. The condition in line 160 guarantees that line 163 can only be reached if the search method called in line 157 returned a window with window.Curr.key != the operation's key. By Claim 2.4.41, there was a point, the search linearization point, when this node was the first node in the list satisfying that its key >= the operation's key, meaning that the operation's key was not in the list at that time. (This search linearization point is also the linearization point for a delete operation that followed route 1.)

Lemma 2.4.81. *A delete method that finished 'works correctly', meaning that one of the following has happened:*

* *It returned true, and during the operation a reachable node with a corresponding key was marked, and this Marking, denoted x, satisfies B(x) = y.*

* *It returned false. During the operation a node with a corresponding key was marked, but this Marking, denoted x, doesn't satisfy B(x) = y (and also no other Marking satisfies that condition).*

* *It returned false, without making any logical changes to the list, and during its run there was a moment in which the operation's key wasn't logically in the list.*

Proof: If the delete method returned from line 61, then by the condition in line 59 we know that it finished in the determine_delete state, meaning it followed route 2. If the CAS in line 61 succeeded, then the method returned true, and by Claim 2.4.78, there exists a Marking x satisfying B(x) = y as required. This is case 1. If the CAS in line 61 failed, then no Marking x can satisfy B(x) = y since by definition of function B (Definition 2.4.75) it can only match Markings to successful delete operations. This is case 2. If the method did not return from line 61, then it returned false, and by the condition in line 60, we know it followed route 1. Then by Claims 2.4.79 and 2.4.80, it made no logical changes to the list, and during its run there was the linearization point in which the operation's key wasn't in the list.

2.4.7 Wait-Freedom

Definition 2.4.82. The pending time of an operation is the time interval in which the operation is pending.

Claim 2.4.83. *The number of logical changes to the list at a given interval of time is bounded by the number of (delete and insert) operations that were pending during that interval.*

Proof: Recall by Corollary 2.5 that Insertion and Marking are the only logical changes to the list. Claim 2.4.57 matches every Insertion to a distinct insert operation, and Claim 2.4.66 guarantees that this Insertion happened during the execution of the insert operation. Claims 2.4.76 and 2.4.77 match every Marking to a distinct delete operation, and guarantee the marking happened during it. We conclude that, in a given time interval, every logical change to the list, be it Marking or Insertion, is matched in a one-to-one correspondence to a distinct operation that happened (at least partially) during this time interval, and thus the number of logical changes is bounded by the operations.

Claim 2.4.84. *The number of Redirections (as defined in Definition 2.4.34) at a given interval of time is bounded by [the number of insert operations that were pending (at least partially) at the time interval] * [Logical changes to the list linearized at that time interval + 1] * 2*

Proof: Redirections can result from a CAS either in line 138 or 140, both in the helpInsert method. For a given insert operation, and a given logical state of the list, the new value in both of these CASes is uniquely defined: In line 138, a pointer to the first node in the list with a key larger than its own, and in line 140, a pointer operation's node. The logical states that the list can be in an interval are: its initial state + another state for each logical change. Hence the total number of logical states the list can be in a given interval is the number of logical changes to the list linearized at that time interval + 1. We multiply by two because the Redirection can happen at either line 138

or 140. The last missing argument is that a different Redirections set exists for every operation's node; hence we also multiply by the number of insert operations.

Claim 2.4.85. *The number of Snippings (as defined in Definition 2.4.34) at a given interval of time is bounded by [Overall marked nodes that existed at some point during that time interval] * [Insertions + Redirections that happened during that interval of time + 1]*

Proof: By Definition 2.4.34, only reachable marked nodes can be snipped. Once a node is marked it is no longer reachable and thus cannot be snipped again, that is, unless it becomes reachable again. A node can become reachable again by Insertion or a Redirection. (Marking doesn't affect reachability and Snipping only makes a single reachable node unreachable). So any marked node can be snipped at most 1 + Number of Insertions + Number of Redirections.

Claim 2.4.86. *The number of successful CASes performed on nodes at any interval of time is bounded by the Insertions + Markings + Redirections + Snippings performed at that interval, and thus bounded.*

Proof: By Lemma 2.4.35, all the changes to a node's next field are either Markings, Snippings, Insertions, or Redirections. We have bounded all of those groups in the previous claims of this subsection, and thus the total number of successful CASes is bounded.

Claim 2.4.87. *Each CAS on the state array belongs to a specific operation.*

CASes on the state array are done only in the helpInsert and helpDelete methods. By Definition 2.4.6, each instance of these methods belongs to a specific operation.

Claim 2.4.88. *The number of successful CASes on the state array belonging to any delete operation is bounded by a constant of 2.*

By Claim 2.4.74 a delete operation may have either one successful CAS (from search_delete to failure) or two successful CASes (from search_delete to execute_delete to determine_delete).

Claim 2.4.89. *The number of operations that have a pending time with an overlap to the pending time of any given operation is bounded by twice the overall number of threads in the system.*

Intuitively, this is the outcome of the help mechanism, which basically guarantees that a thread will not move on to subsequent operations before helping a concurrent operation that began before its last operation.
Proof: The structure of the maxPhase method guarantees that for two non-concurrent executions of it, the later one will receive a larger phase number. For a given operation,

at the moment it becomes pending, any other thread is pending on no more than one operation. It can later begin a new operation, but this new operation will have a larger phase number. Each operation (be it an insert or a delete operation) calls the help method. The help method structure guarantees that it will not exit while there is a pending operation with a smaller phase number. So no thread can start a third concurrent operation while the given operation is still pending.

Claim 2.4.90. *The number of physical changes (i.e., successful CASes) on the list that can occur during the pending time of any given operation is bounded.*

Proof: By Claim 2.4.89, the number of other operations that can be pending while the given operation is pending is bounded. Thus, the number of physical changes that can happen during this time is bounded by Claims 2.4.83,2.4.84,2.4.85,2.4.86.

Claim 2.4.91. *The number of successful CASes on the state array belonging to an insert operation is bounded.*

CASes on the state array that are performed during an insert operation are only performed in the helpInsert method, and all of them check that the previous state is a pending insert state. Thus, once a CAS successfully changes the operation to something other than a pending insert, no more CASes are possible inside the helpInsert method. Thus, the only possible CAS that has the potential of unbounded repetition is the one in line 136. After a thread succeeds in that CAS, it will not attempt it again before it attempts the CAS in line 140. If it fails the CAS in line 140, it must be due to a physical change to a node's next field that was made since the (linearization point of the) search method called by in line 110, but that may only happen a limited number of times, by Claim 2.4.90. Thus, there is only a bounded number of times that the CAS in 136 can succeed in that insert operation, until the CAS in line 140 of that insert operation succeeds (at least once) as well. After the CAS in line 140 succeeds, the operation's node has already been inserted to the list. It cannot become unreachable while it is unmarked (Claim 2.4.37). Thus, after that point, each thread that restarts the loop of lines 104-146 will not reach line 136 again, because either the condition in line 113 or the one in line 128 will be true, and the method will exit.

Claim 2.4.92. *All the methods in the code will exit in a bounded number of steps.*

Proof: We shall go over all the methods one by one.
All constructors are just field initializations that contain no loops or conditions, and thus will be finished in a small number of steps.
The maxPhase method doesn't contain loops or conditions, and will thus finish after a small number of steps. (Note: it doesn't check the condition of the CAS, and will exit even if the CAS fails.)
The search method is bounded since it searches a specific key and only goes forward

in the list, so it must reach it (or beyond it) after a bounded number of times and thus exit in line 83. (This is because the tail always holds a key larger than all other possible keys by definition 2.4.1, so there is at least one key that answers the condition in line 83.) The only possibility for a search method to go backwards in the list is if the condition in line 78 returns true. For this to happen, the CAS in line 75 must have failed, which may only happen a bounded number of times while the operation is pending. If the operation is no longer pending, the condition in line 76 guarantees that the search method will exit.
The helpDelete and helpInsert methods call the search method, which is bounded. Other than that, they might only enter another iteration of a loop because of changes that were made to the list or state, but these changes are bounded by Claims 2.4.90,2.4.87,2.4.88,2.4.91 while the operation is pending, and once it becomes non-pending, it will exit due to the condition in line 106 or 152.
The help method is a finite loop that calls helpInsert and helpDelete a finite number of times.
The insert and delete methods call the maxPhase and the help method, and have no loops. Note that even though the delete method attempts a CAS, it returns even if the CAS fails.

Corollary 2.8. *Wait-Freedom*

Proof: A result of the previous claim.

2.4.8 Final Conclusion

Corollary 2.9. *The described algorithm creates a wait-free linked-list.*

This follows from Lemmas 2.4.68, 2.4.81, and Corollary 2.8.

2.5 Linearization Points

In this section we specify the linearization point for the different operations of the linked-list. The SEARCH method for a key k returns a pair of pointers, denoted *pred* and *curr*. The prev pointer points to the node with the highest key smaller than k, and the curr pointer points to the node with the smallest key larger than or equal to k. The linearization point of the SEARCH method is when the pointer that connects *pred* to *curr* is read. This can be either at Line 36 or 45 of the SEARCH method. Note that *curr*'s `next` field will be subsequently read, to make sure it is not *marked*. Since it is an invariant of the algorithm that a marked node is never unmarked, it is guaranteed that at the linearization point both *pred* and *curr* nodes were unmarked.

The linearization point for a CONTAINS method is the linearization point of the appropriate SEARCH method. The appropriate SEARCH method is the one called from within the HELPCONTAINS method by the thread that subsequently successfully reports

the result of the same CONTAINS operation. The linearization point of a *successful insert* is in Lines 47-48 (together they are a single instruction) of the *helpInsert* method. This is the CAS operation that physically links the node into the list. For a *failing insertion*, the linearization point is inside the linearization point of the SEARCH method executed by the thread that reported the failure.

The linearization point of a *successful delete* is at the point where the node is *logically* deleted, which means successfully marked (Line 38 in the helpDelete method). Note that it is possible that this is executed by a helping thread and not necessarily by the operation owner. Furthermore, the helping thread might be trying to help a different thread than the one that will eventually own the deletion. The linearization point of an *unsuccessful delete* is more complex. A delete operation may fail when the key is properly deleted, but a different thread is selected as the owner of the delete. In this case, the current thread returns failure, because of the failure of the CAS of the DELETE method (at Line 9). In this case, the linearization point is set to the point when the said node is logically deleted, in Line 38 of HELPDELETE. The linearization point of an *unsuccessful delete*, originating from simply not finding the key, is the linearization point of the SEARCH method executed by the thread that reported the failure.

2.6 A Fast-Path-Slow-Path Extension

2.6.1 overview

In this section, we describe the extension of the naive wait-free algorithm using the fast-path-slow-path methodology. The goal of this extension is is to improve performance and obtain a fast wait-free linked-list. We provide a short description of the method here. Full motivation and further details appear in [KP12]. A full Java code for the fast-path-slow-path list is presented in Appendix C.

The idea behind the fast-path-slow-path [KP12] approach is to combine a (fast) lock-free algorithm with a (slower) wait-free one. The lock free algorithm provides a basis for a fast path and we use Harris's lock-free linked-list for this purpose. The execution in the fast path begins by a check whether a help is required for any operation in the slow path. Next, the execution proceeds with running the fast lock-free version of the algorithm while counting the number of contentions that end with a failure (i.e., failed CASes)[4]. Typically, few failures occur and help is not required, and so the execution terminates after running the faster lock-free algorithm. If this fast path fails to make progress, the execution moves to the slow path, which runs the slower wait-free algorithm described in Section 2.3, requesting help (using an operation descriptor in its slot in the state array) and making sure the operation eventually terminates.

[4]Another point to consider is the possibility that a thread can't make progress since other threads keep inserting new nodes to the list, and it can't finish the search method. We address this potential problem in Section 2.6.7.

The number of CAS failures allowed in the fast path is limited by a parameter called MAX_FAILURES. The help is provided by threads running both the fast and slow path, which ensures wait-freedom: if a thread fails to complete its operation, its request for help is noticed both in the fast and in the slow path. Thus, eventually all other threads help it and its operation completes. However, help is not provided as intensively as described in Section 2.3. We use the *delayed help* mechanism, by which each thread only offers help to other threads once every several operations, determined by a parameter called HELPING_DELAY.

Combining the fast-path and the slow-path is not trivial, as care is needed to guarantee that both paths properly run concurrently. On top of other changes, it is useful to note that the DELETE operation must compete on the `success-bit` even in the fast-path, to avoid a situation where two threads running on the two different paths both think they were successful in deleting a node.

2.6.2 The Delayed Help Mechanism

In order to avoid slowing the fast path down, help is not provided to *all* threads in the beginning of *each* operation execution. Instead, help is provided to at most one thread, in a round-robin manner. Furthermore, help is not provided in each run of an operation, but only once every few operation executions. This scheme still guarantees wait-freedom for the threads that require help, but it does not overwhelm the system with contention of many helping threads attempting to run the same operation on the same part of the list.

The above mechanism is called *delayed-help*. In addition to an entry in the state array, each thread maintains a helping record. The first field in a helping record holds the *TID* of the *helped thread*. This thread is the next one in line to receive help, if needed. In addition to the TID of the helped thread, the helping record holds a *nextCheck* counter, initialized to the HELPING_DELAY parameter and decremented with each operation that does not provide help, and a phase number, recording the phase the helped thread had when the help of the previous thread terminated.

Before a thread T performs an operation (in the fast or slow path), T decrements the `nextCheck` counter in its helping record by one. If `nextCheck` reaches zero, than T checks whether the helped thread has a pending operation (i.e., it needs help) and whether this pending operation has the same phase that was previously recorded. This means that the helped thread made no progress for a while. If this is the case, then T helps it. After checking the helped thread's state and providing help if required, T updates its help record. The field holding the TID of the helped thread is incremented to hold the id of the next thread, the phase of this next thread is recorded, and NEXTCHECK is initialized to HELPING_DELAY. Pseudo-code for this is depicted in Figure 2.7

```
 1: class HelpRecord {
 2:       int curTid; long lastPhase; long nextCheck;
 3:       HelpRecord() { curTid = -1; reset(); }
 4:       public void reset() {
 5:         curTid = (curTid + 1) % Test.numThreads;
 6:         lastPhase = state.get(curTid).phase;
 7:         nextCheck = HELPING_DELAY;
 8:       }
 9:    }
10:
11: private void helpIfNeeded(int tid) {
12:    HelpRecord rec = helpRecords[tid*width];
13:    if (rec.nextCheck-- == 0) {                    ▷ delay help HELPING_DELAY times
14:       OpDesc desc = state.get(rec.curTid);
15:       if (desc.phase == rec.lastPhase) {                    ▷ help might be needed
16:          if (desc.type == OpType.insert)
17:             helpInsert(rec.curTid, rec.lastPhase);
18:          else if (desc.type == OpType.search_delete ||
19:             desc.type == OpType.execute_delete)
20:             helpDelete(rec.curTid, rec.lastPhase);
21:       }
22:       rec.reset();
23:    }
24: }
```

Figure 2.7: The delayed help mechanism

2.6.3 The Search Method

The FASTSEARCH method (Figure 2.8) is identical to the original lock-free search, except for counting the number of failed CAS operations. If this number reaches MAX_FAILURES, FASTSEARCH returns null. It is up to the caller (fastInsert or fastDelete) to move to the slow path, if null is returned. The slowSearch method (called SEARCH hereafter) operation is identical to the wait-free search introduced method in Section 2.3.

2.6.4 The Insert Operation

The INSERT operation (Figure 2.9) starts in the fast path and retreats to SLOWINSERT (Figure 2.10) when needed. It starts by checking if help is needed. After that, it operates as the original lock-free insert, except for counting CAS failures. It also checks whether FASTSEARCH has returned a null, in which case it reverts to the slow path.

The SLOWINSERT method (Figure 2.10) is similar to the wait-free insert, except that it performs its own operation only, and does not help other operations. The helpInsert method is identical to the wait-free method presented in Section 2.3.

2.6.5 The Delete Operation

The DELETE method (Figure 2.11) is similar to the delete operation of the original lock-free list with some additions. In addition to checking the number of failures, further

```
1:  public Window fastSearch(int key) {
2:    int tries = 0; Node pred = null, curr = null, succ = null;
3:    boolean[] marked = {false};
4:    boolean snip;
5:    retry : while (tries++ < MAX_FAILURES) {                    ▷ do I need help?
6:      pred = head;
7:      curr = pred.next.getReference();                          ▷ advancing curr
8:      while (true) {
9:        succ = curr.next.get(marked);                           ▷ advancing succ
10:       while (marked[0]) {                                     ▷ curr is logically deleted
11:                                  ▷ The following line is an attempt to physically remove curr:
12:         snip = pred.next.compareAndSet(curr, succ, false, false);
13:         if (!snip) continue retry;                            ▷ list has changed, retry
14:         curr = succ;                                          ▷ advancing curr
15:         succ = curr.next.get(marked);                         ▷ advancing succ
16:       }
17:       if (curr.key >= key)                                    ▷ the window is found
18:         return new Window(pred, curr);
19:       pred = curr; curr = succ;                               ▷ advancing pred & curr
20:     }
21:   }
22:   return null;                                                ▷ asking for help
23: }
```

Figure 2.8: The FPSP fastSearch method

```
1:  public boolean insert(int tid, int key) {
2:    helpIfNeeded(tid);
3:    int tries = 0;
4:    while (tries++ < MAX_FAILURES) {                            ▷ do I need help?
5:      Window window = fastSearch(key);
6:      if (window == null)                                 ▷ search failed MAX_FAILURES times
7:        return slowInsert(tid, key);
8:      Node pred = window.pred, curr = window.curr;
9:        if (curr.key == key)
10:       return false;                                         ▷ key exists - operation failed.
11:     else {
12:       Node node = new Node(key);                            ▷ allocate the node to insert
13:       node.next = new
14:         VersionedAtomicMarkableReference<Node>(curr, false);
15:       if (pred.next.compareAndSet(curr, node, false, false))
16:         return true;                                        ▷ insertion succeeded
17:     }
18:   }
19:   return slowInsert(tid, key);
20: }
```

Figure 2.9: The FPSP insert method

```
 1: private boolean slowInsert(int tid, int key) {
 2:   long phase = maxPhase();                          ▷ getting the phase for the op
 3:   Node n = new Node(key);                                  ▷ allocating the node
 4:   n.next = new
 5:     VersionedAtomicMarkableReference<Node>(null, false);
 6:   OpDesc op = new OpDesc(phase, OpType.insert, n,null);
 7:   state.set(tid, op);                    ▷ publishing the operation - asking for help
 8:   helpInsert(tid, phase);                              ▷ only helping itself here
 9:   return state.get(tid).type == OpType.success;
10: }
```

Figure 2.10: The FPSP slowInsert method

```
 1: public boolean delete(int tid, int key) {
 2:   helpIfNeeded(tid);
 3:   int tries = 0; boolean snip;
 4:   while (tries++ < MAX_FAILURES) {                          ▷ do I need help?
 5:     Window window = fastSearch(key);
 6:     if (window == null)                    ▷ search failed MAX_FAILURES times
 7:       return slowDelete(tid, key);
 8:     Node pred = window.pred, curr = window.curr;
 9:     if (curr.key != key)                     ▷ key doesn't exist - operation failed
10:       return false;
11:     else {
12:       Node succ = curr.next.getReference();
13:                          ▷ The following line is an attempt to logically delete curr:
14:       snip = curr.next.compareAndSet(succ, succ, false, true);
15:       if (!snip)
16:         continue;                                                  ▷ try again
17:                        ▷ The following line is an attempt to physically remove curr:
18:       pred.next.compareAndSet(curr, succ, false, false);
19:                          ▷ the following is needed for cooperation with the slow path:
20:       return curr.d.compareAndSet(false, true);
21:     }
22:   }
23:   return slowDelete(tid, key);
24: }
```

Figure 2.11: The FPSP delete method

cooperation is required between threads. Determining which thread deleted a value is complicated in the wait-free algorithm and requires some cooperation from the fast path as well. In particular, after performing a delete that is considered successful in the fast path, the new DELETE method must also atomically compete (i.e., try to set) the extra `success bit` in the node. This bit is used by the wait-free algorithm to determine which thread owns the deletion of a node. Neglecting to take part in setting this bit may erroneously allow both a fast-path delete and a concurrent slow-path delete to conclude that they both are successful for the same delete. Upon failing to set the `success bit` in the node, DELETE returns failure.

The SLOWDELETE method (Figure 2.12) is similar to the wait-free version of the DELETE method, except that it does not need to help any other threads. The

```
 1: private boolean slowDelete(int tid, int key) {
 2:    long phase = maxPhase();                                  ▷ getting the phase for the op
 3:    state.set(tid, new OpDesc
 4:       (phase, OpType.search_delete, new Node(key), null));
 5:    helpDelete(tid,phase);                                    ▷ only helping itself here
 6:    OpDesc op = state.get(tid);
 7:    if (op.type == OpType.determine_delete)
 8:                                  ▷ the following competes on the ownership of deleting the node:
 9:       return op.searchResult.curr.d.compareAndSet(false, true);
10:    return false;
11: }
```

Figure 2.12: The FPSP slowDelete method

helpDelete method is identical to the one presented in Section 2.3.

2.6.6 Linearization Points

The linearization points are simply the linearization points of the lock-free and wait-free algorithms, according to the path in which the operation takes place. In the fast path, a successful insert operation is the CAS linking the node to the list (line 15 in the insert method), and an unsuccessful one is at the fastSearch method (line 9 or 15, whichever is read last). A successful delete is linearized in a successful CAS in line 14 of the delete method. Note that it is possible for an unsuccessful delete to be linearized at this point too, if a slow-path operation will own this deletion eventually. The usual unsuccessful delete (the key doesn't exist) linearization point is similar to the one described in Section 2.5, at the begining of the fastSearch method if the key didn't exist then, or at the point when it was marked, if it did exist. The other linearization points, those of the slow-path, are unchanged from those elaborated on in Section 2.5. It is worth noting that the linearization point of a successful delete in the slow path, which is always upon marking the node, might actually happen during a run of the fast path of a delete method.

2.6.7 The Contains Operation and Handling Infinite Insertions

In Section 2.3.6, we noted that infinite concurrent insertions into the list create a challenge to the wait-freedom property, since the contains method may never be able to reach the desired key if more and more keys are inserted before it. This problem has a bound when dealing with integers, as there is a bound to the number of possible integer keys, but has no bound when dealing with other types of keys, such as strings. If every operation on the list is always done using the help mechanism, this problem cannot occur, since other threads will help the pending operations before entering new keys. This is how the problem was handled in the contains method in Section 2.3.6.

It is perhaps debatable whether a wait-free algorithm should offer a solution for this problem, as the failure does not happen due to contention, but due to the fact

that the linear complexity of the problem (in the number of keys) increases while the thread is working on it. This debate as beyond the scope of our work, and our goal here is to offer solutions to the problem. For the basic wait-free algorithm, we could solve this problem by making sure that all operations (including CONTAINS) will use the helping mechanism. However, for the fast-path-slow-path extension, it is by definition impossible to force all threads to use the helping mechanism, as this would contradict the entire point of the fast-path-slow-path. Instead, a thread must be able to recognize when its operation is delayed due to many concurrent insertions, and ask for help (aka, switch to the slow path) if this problem occurs. The purpose of this section is to suggest an efficient way to do that.

The idea is that each thread will read the number of total keys in the list prior to starting the search. During the search, it will count how many nodes it traversed, and if the number of traversed nodes is higher than the original total number of keys (plus some constant), it will abort the search and ask for help in its operation. The problem is that maintaining the size of the list in a wait-free manner can be very costly. Instead, we settle for maintaining a field that approximates the number of keys. The error of the approximation is also bounded by a constant (actually, a linear function in the number of threads operating on the list). Thus, before a thread starts traversing the list, it should read the approximation, denoted $Size\_App$, and if it traverses a number of nodes that is greater than $Size\_App + Max\_Error + Const$, switch to the slow path and ask for help.

To maintain the approximation for the number of keys in the list, the list contains a global field with the approximation, and each thread holds a private counter. In its private counter, each thread holds the number of nodes it inserted to the list minus the number of nodes it deleted from the list since the last time it updated the global approximation field. To avoid too much contention in updating the global field, each thread only attempts to update it (by a CAS) once it reached a certain *soft_threshold* (in absolute value). If the CAS failed, the thread continues the operation as usual, and will attempt to update the global approximation field at its next insert or delete operation. If the private counter of a thread reached a certain *hard_threshold*, it asks for help in updating the global counter, similarly to asking help for other operations.

Some care is needed to implement the helping mechanism for updating the approximation field in a wait-free manner. This is not very complicated, but is also not completely trivial. The full Java code that also handles this difficulty is given if Appendix C.

2.7 Performance

Implementation and platform. We compared four Java implementations of the linked-list. The first is the lock-free linked-list of Harris, denoted *LF*, as implemented by Herlihy and Shavit in [HS08]. (This implementation was slightly modified to allow

nodes with user-selected keys rather than the object's hash-code. We also did not use the *item* field.)

The basic algorithm described in Section is denoted *WF-Orig* in the graphs below. A slightly optimized version of it, denoted *WF-Opt*, was changed to employ a delayed help mechanism, similar to the one used in the fast-path- slow-path extension. This means that a thread helps another thread only once every k operations, where k is a parameter of the algorithm set to 3. The idea is to avoid contention by letting help arrive only after the original thread has a reasonable chance of finishing its operation on its own. This optimization is highly effective, as seen in the results. Note that delaying help is not equivalent to a fast-path-slow-path approach, because all threads always ask for help (there is no fast path). All the operations are still done in the *helpInsert* and *helpDelete* methods.

The fast-path-slow-path algorithm, denoted *FPSP*, was run with the HELPING_DELAY parameter set to 3, and MAX_FAILURES set to 5. This algorithm combines the new wait-free algorithm described in this chapter with Harris's lock-free algorithm, to achieve both good performance and the stronger wait-freedom progress guarantee.

We ran the tests in two environments. The first was a SUN's Java SE Runtime, version 1.6.0 on an IBM x3400 system featuring 2 Intel(R) Xeon(R) E5310 1.60GHz quad core processors (overall 8 cores). The second was a SUN FIRE machine with an UltraSPARC T1 8 cores each running four hyper-threads.

Workload and methodology. In the micro-benchmarks tested, we ran each experiment for 2 seconds, and measured the overall number of operations performed by all the threads during that time. Each thread performed 60% CONTAINS, and 20% INSERT and DELETE operations, with keys chosen randomly and uniformly in the range $[1, 1024]$. The number of threads ranges from 1-16 (in the Intel(R) Xeon(R)) or from 1-32 (In the UltraSPARC). We present the results in Figure 2.13. The graphs show the total number of operations done by all threads in thousands for all four implementations, as a function of the number of threads. In all the tests, we executed each evaluation 8 times, and the averages are reported in the figures.

Results. It can be seen that the fast-path-slow-path algorithm is almost as fast as the lock-free algorithm. On the Intel machine, the two algorithms are barely distinguishable; the difference in performance is 2-3%. On the UltraSPARC the fast-path-slow-path suffers a noticeable (yet, reasonable) overhead of 9-14%. The (slightly optimized) basic wait-free algorithm is slower by a factor of 1.3–1.6, depending on the number of threads. Also, these three algorithms provide an excellent speed up of about 7 when working with 8 threads (on both machines), and about 24 when working with 32 multi-threads on the UltraSPARC. The basic non-optimized version of the wait-free algorithm doesn't scale as well. There, threads often work together on the same operation, causing a deterioration in performance and scalability. The simple delayed-help optimization enables concurrency without foiling the worst-case wait-freedom guarantee.

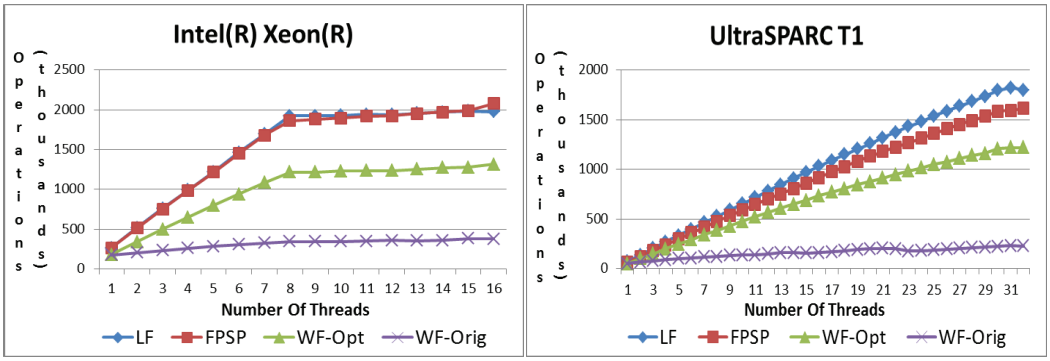

Figure 2.13: The number of operations done in two seconds as a function of the number of threads

2.8 Conclusion

We presented a wait-free linked-list. To the best of our knowledge, this is the first design of a wait-free linked-list in the literature, apart from impractical universal constructions. This design facilitates for the first time the use of linked-lists in environments that require timely responsiveness, such as real-time systems. We have implemented this linked-list in Java and compared it to Harris's lock-free linked-list. The naive wait-free implementation is slower than the original lock-free implementation by a factor of 1.3 to 1.6. We then combined our wait-free design with Harris's lock-free linked-list design using the fast-path-slow-path methodology, and implemented the extended version as well. The extended algorithm obtains performance which is very close to the original lock-free algorithm, while still guaranteeing non-starvation via wait-freedom.

Chapter 3

A Practical Wait-Free Simulation for Lock-Free Data Structures

3.1 Introduction

In the previous chapter we designed the first practical wait-free linked-list. To do that, we started from the lock-free linked list [Har01], added a helping mechanism, and then applied the fast-path-slow-path methodology [KP12] to enhance the performance. Such a design process is complicated and error prone. As discussed in Section 1.2, in this chapter we present a general transformation that converts any lock-free data structure (given in a normalized form, that we define) to a wait-free data structure. That is, our transformation is a generic way to add a helping mechanism and combine the help-less lock-free algorithm with the wait-free one according to the fast-path-slow-path methodology in such a way that preserves linearizability and performance. A (shorter) version of this transformation was published in [TP14].

Using the new method, we have designed and implemented wait-free linked-list, skiplist, and tree and we measured their performance. It turns out that for all these data structures the wait-free implementations are only a few percent slower than their lock-free counterparts.

The contributions of this chapter include the definition of the normalized form for a lock-free data structure; the design of the generic wait-free simulation for a normalized lock-free data structure; A demonstration of the generality of the normalized representation, by showing the normalized representation for lock-free linked-list, skiplist and tree; A formal argument for the correctness of the transformation, and thus also the obtained wait-free data structures; and implementations and measurements validating the efficiency of the proposed scheme.

We limit our discussion to the field of lock-free linearizable data structures. We believe our ideas can be applied to other algorithms as well, such as lock-free implementations of STM, but this is beyond the scope of this work.

The chapter is organized as follows. Section 3.2 discusses additional related work to

that described in Section 1.2. In Section 3.3 we provide an overview of the proposed transformation. In Section 3.4 we briefly discuss the specifics of the shared memory model assumed in this work. In Section 3.5 we examine typical lock-free data structures, and characterize their properties in preparation to defining a normalized representation. The normalized representation is defined in Section 3.6, and the wait-free simulating for a normalized lock-free data structure appears in Section 3.7. We prove the correctness of the transformation in 3.8. We discuss the generality of the normalized form in Section 3.9. Next, in Section 3.10, we show how to easily convert four known lock-free data structures into the normalized form, and thus obtain a wait-free version for them all. Some important optimizations are explained in Section 3.11 , and our measurements are reported in Section 3.12.

3.2 Additional Related Work

The idea of mechanically transforming an algorithm to provide a practical algorithm with a different progress guarantee is not new, and not limited to universal constructions. Taubenfeld introduced contention-sensitive data structures (CSDS) and proposed various mechanical transformation that enhance their performance of progress guarantees [Tau09]. Ellen et al suggested a transformation of obstruction-free algorithms into wait-freedom algorithms under a different computation model known as semisynchronous [FLMS05]. This construction does not extend to the standard asynchronous model.

Recently, we have seen some progress with respect to practical wait-free data structures. A practical design of a wait-free queue relying on *compare and swap* (CAS) operations was presented in [KP11]. Next, an independent construction of a wait-free stack and queue appeared in [FK11]. A wait-free algorithm for the linked-list has been published in [TBKP12] and given here in Chapter 2. Finally, a wait-free implementation of a red-black tree appeared in [NSM13].

One of the techniques employed in this work is the fast-path-slow-path method, which attempts to separate slow handling of difficult cases from the fast handling of the more typical cases. This method is ubiquitous in systems in general and in parallel computing particularly [Lam87, MA95, AK99, AK00], and has been adopted recently [KP12] for creating fast wait-free data structures.

3.3 Transformation overview

The move from the lock-free implementation to the wait-free one is executed by simulating the lock-free algorithm in a wait-free manner. The simulation starts by simply running the original lock-free operation (with minor modifications that will be soon discussed). A normalized lock-free implementation has some mechanism for detecting failure to make progress (due to contention). When an operation fails to make progress it asks for help from the rest of the threads. A thread asks for help by enqueuing a succinct

description of its current computation state on a wait-free queue (we use the queue of [KP11]). One modification to the fast lock-free execution is that each thread checks once in a while whether a help request is enqueued on the help queue. Threads that notice an enqueued request for help move to helping a single operation on the top of the queue. Help includes reading the computation state of the operation to be helped and then continuing the computation from that point, until the operation completes and its result is reported.

The major challenges are in obtaining a succinct description of the computation state, in the proper synchronization between the (potentially multiple) concurrent helping threads, and in the synchronization between helping threads and threads executing other operations on the fast lock-free path. The normalized representation is enforced in order to allow a succinct computation representation, to ensure that the algorithm can detect that it is not making progress, and to minimize the synchronization between the helping threads to a level that enables fast simulation.

The helping threads synchronize during the execution of an operation at critical points, which occur just before and just after a modification of the data structure. Assume that modifications of the shared data structure occur using a CAS primitive. A helping thread runs the operation it attempts to help independently until reaching a CAS instruction that modifies the shared structure. At that point, it coordinates with all helping threads which CAS should be executed. Before executing the CAS, the helping threads jointly agree on what the CAS parameters should be (address, expected value, and new value). After deciding on the parameters, the helping threads attempt to execute the CAS and then they synchronize to ensure they all learn whether the CAS was successful. The simulation ensures that the CAS is executed exactly once. Then each thread continues independently until reaching the next CAS operation and so forth, until the operation completes. Upon completing the operation, the operation's result is written into the computation state, the computation state is removed from the queue, and the *owner* thread (the thread that initiated the operation in the first place) can return.

There are naturally many missing details in the above simplistic description, but for now we will mention two major problems. First, synchronizing the helping threads before each CAS, and even more so synchronizing them again at the end of a CAS execution to enable all of them to learn whether the CAS was successful, is not simple. It requires adding version numbering to some of the fields in the data structure, and also an extra `modified bit`. We address this difficulty in Section 3.7.

The second problem is how to succinctly represent the computation state of an operation. An intuitive observation (which is formalized later) is that for a lock-free algorithm, there is a relatively light-weight representation of its computation state. This is because by definition, if at any point during the run a thread stops responding, the remaining threads must be able to continue to run as usual. This implies that if a thread modifies the data structure, leaving it in an "intermediate state" during the

computation, then other threads must be able to restore it to a "normal state". Since this often happens in an execution of a lock-free algorithm, the information required to do so must be found on the shared data structure, and not (solely) in the thread's inner state. Using this observation, and distilling a typical behavior of lock-free algorithms, we introduce a normalized representation for a lock-free data structure, as defined in Section 3.6. The normalized representation is built in a way that enables us to represent the computation state in a compact manner, without introducing substantial restrictions on the algorithm itself.

There is one additional key observation required. In the above description, we mentioned that the helping threads must synchronize in critical points, immediately before and immediately after each CAS that modifies the data structure. However, it turns out that with many of the CASes, which we informally refer to as *auxiliary* CAS*es*, we do not need to use synchronization at all. As explained in Section 3.5, the nature of lock-free algorithms makes the use of auxiliary CASes common. Most of Section 3.5.2 is dedicated to formally define *parallelizable methods*; these are methods that only execute auxiliary CASes, and can therefore be run by helping threads without any synchronization. These methods will play a key role in defining normalized lock-free representation in Section 3.6.

3.4 Model and General Definitions

We consider a standard shared memory setting. In each computation step, a single thread executes on a target address in the shared memory one of three atomic primitives: READ, WRITE, or CAS. A computation step may also include a local computation, which may use local memory.

A CAS primitive is defined according to a triplet: `target address`, `expected-value` and `new-value`. A CAS primitive atomically compares the value of the target address to the expected-value, and WRITES the new value to the target address if the `expected-value` and old value in the target address are found identical. A CAS in which the `expected-value` and old value are indeed identical returns true, and is said to be *successful*. Otherwise the CAS returns false, and is *unsuccessful*. A CAS in which the `expected-value` and `new-value` are identical is a *futile* CAS.

An abstract data type, ADT, is defined by a state machine, and is accessed via *operations*. An operation receives zero or more input parameters, and returns one result, which may be null. The state machine of a type is a function that maps a state and an operation (including input parameters) to a new state and a result of the operation.

A *method* is a sequence of code-instructions that specify computation steps, including local computation. The next computation step to be executed may depend on the results of previous computation steps. Similarly to an operation, a method receives zero or more input parameters, and returns one result, which may be null. A code instruction inside a method may invoke an additional method.

A special method, ALLOCATE, which receives as an input the amount of memory needed and returns a pointer to the newly allocated memory is assumed to be available. We assume automatic garbage collection is available. This means that threads need not actively invoke a DEALLOCATE method, and an automatic garbage collector reclaims memory once it is no longer reachable by the threads. For further discussion about memory management, see Section 3.12.1.

A data structure implementation is an implementation of an ADT. (e.g., Harris's linked-list is a data structure implementation). Such an implementation is a set of methods that includes a method for each operation, and may include other supporting methods.

A *program* is a set of one or more methods, and an indication which method is the entry point of the program. In an *execution*, each thread is assigned a single program. The thread executes the program by following the program's code-instructions, and execute computation steps accordingly.

An execution is a (finite or infinite) sequence of computation steps, cleaned out of the local computation. A scheduling is a (finite or infinite) sequence of threads. Each execution defines a unique scheduling, which is the order of the threads that execute the computation steps. Given a set of threads, each of which coupled with a program, and a scheduling, a unique corresponding execution exists.

An execution must satisfy MEMORY CONSISTENCY. That is, each READ primitive in the execution must return the value last WRITTEN, or successfully CASed, to the same target address. Also. Each CAS must return true and be successful if and only if the `expected-value` is equal to the last value written (or successfully CASes) into the same target address. Most works do not particularly define MEMORY CONSISTENCY and take it for granted, but the way we manipulate executions in our correctness proof (Section 3.8) makes this definition essential.

3.5 Typical Lock-Free Algorithms

In this section we provide the intuition on how known lock-free algorithms behave and set up some notation and definitions that are then used in Section 3.6 to formally specify the normalized form of lock-free data structures.

3.5.1 Motivating Discussion

Let us examine the techniques frequently used within lock-free algorithms. We target linearizable lock-free data structures that employ CASes as the synchronization mechanism. A major difficulty that lock-free algorithms often need to deal with is that a CAS instruction executes on a single word (or double word) only, whereas the straightforward implementation approach requires simultaneous atomic modification of

multiple (non-consecutive) words[1]. Applying a modification to a single-field sometimes leaves the data structure inconsistent, and thus susceptible to races. A commonly employed solution is to use one CAS that (implicitly) blocks any further changes to certain fields, and let any thread remove the blocking after restoring the data structure to a desirable consistent form and completing the operation at hand.

An elegant example is the delete operation in Harris's linked-list [Har01]. In order to delete a node, a thread first sets a special *mark* bit at the node's next pointer, effectively blocking this pointer from ever changing again. Any thread that identifies this "block" may complete the deletion by physically removing the node (i.e., execute a CAS that makes its predecessor point to its successor). The first CAS, which is executed only by the thread that initiates the operation, can be intuitively thought of as an *owner* CAS.

In lock-free algorithms' implementations, the execution of the owner CAS is often separated from the rest of the operation (restoring the data structure to a "normal" form, and "releasing" any blocking set by the owner CAS) into different methods. Furthermore, the methods that do not execute the owner CAS but only restore the data structure can usually be safely run by many threads concurrently. This allows other threads to unblock the data structure and continue executing themselves. We call such methods *parallelizable methods*.

3.5.2 Notations and Definitions Specific to the Normalized Form.

In this section we formally define concepts that can be helpful to describe lock-free data structures, and are used in this work to define the normalized form.

Definition 3.5.1. (Equivalent Executions.) Two executions E and E' of operations on a data structure D are considered *equivalent* if the following holds.

- (Results:) In both executions all threads execute the same data structure operations and receive identical results.
- (Relative Operation Order:) The order of invocation points and return points of all data structure operations is the same in both executions.
- (Comparable length:) either both executions are finite, or both executions are infinite.

Note that the second requirement does not imply the same timing for the two executions. It only implies the same relative order of operation invocations and exits. For example, if the ith operation of thread T_1 was invoked before the jth operation of T_2 returned in E, then the same must also hold in E'. Clearly, if E and E' are equivalent executions, then E is linearizable if and only if E' is linearizable.

In what follows we consider the invocation of methods. A method is invoked with zero of more input parameters. We would like to discuss situations in which two or more invocations of a method receive the exact same input parameters. If the method

[1] This is one of the reasons why transactional memories are so attractive.

parameters do not include pointers to the shared memory, then comparing the input is straight-forward. However, if a method is invoked with the same input I at two different points in the execution t_1 and t_2, but I includes a pointer to a memory location that was allocated or deallocated between t_1 and t_2, then even though I holds the same bits, in effect, it is different. The reason for this this that in t_1 and t_2 I holds a pointer to a different "logical memory", which happens to be physically allocated in the same place. To circumvent this difficulty, we use the following definition.

Definition 3.5.2. (Memory Identity.) For a method input I and an execution E, we say that I *satisfies memory identity* for two points in the execution t_1 and t_2, if no memory in I, or reachable from I, is allocated or deallocated between t_1 and t_2 in E.

Next, we identify methods that can be easily run with help, i.e., can be executed in parallel by several threads without harming correctness and while yielding adequate output. For those familiar with Harris's linked-list, a good example for such a method is the search method that runs at the beginning of the DELETE or the INSERT operations. The search method finds the location in the list for the insert or the delete and during its list traversal it snips out of the list nodes that were previously marked for deletion (i.e., logically deleted entries). The search method can be run concurrently by several threads without harming the data structure coherence and the outcome of any of these runs (i.e., the location returned by the search method for use of insert or delete) can be used for deleting or inserting the node. Therefore, the search method can be easily helped by parallel threads. In contrast, the actual insertion, or the act of marking a node as deleted, which should happen exactly once, is a crucial and sensitive (owner) CAS, and running it several times in parallel might harm correctness by making an insert (or a delete) occur more than once.

To formalize parallelizable methods we first define a harmless, or *avoidable* parallel run of a method. Intuitively, an avoidable method execution is an execution in which each CAS executed during the method can potentially be avoided in an alternative scheduling. That is, in an avoidable method execution, there is an equivalent execution in which the method does not modify the shared memory at all.

Definition 3.5.3. (Avoidable method execution) A run of a method M by a thread T on input I in an execution E of a program P is avoidable if there exists an equivalent execution E' for E such that in both E and E' each thread follows the same program, both E and E' are identical until right before the invocation of M by T on input I, and in E' each CAS that T executes during M either fails or is futile.

Definition 3.5.4. (Parallelizable method.) A method M is a parallelizable method of a given lock-free algorithm, if for any execution in which M is called by a thread T with an input I the following two conditions holds. First, the execution of a parallelizable method depends only on its input, the shared memory, and the results of the method's CAS operations. In particular, the execution does not depend on the executing thread's local state prior to the invocation of the parallelizable method.

Second, At any point in E that satisfies memory identity to I with the point in E in which M is invoked, If we create and run a finite number of parallel threads, and the program of each of these threads would be to run method M on input I, then in any possible resulting execution E', all executions of M by the additional threads are avoidable.

Loosely speaking, for every invocation of a parallelizable method M by one of the newly created threads, there is an *equivalent execution* in which this method's invocation does not change the data structure at all. In concrete known lock-free algorithms, this is usually because every CAS attempted by the newly created thread might be executed by one of the other (original) threads, thus making it fail (unless it is futile). For example, Harris's linked-list search method is parallelizable. The only CASes that the search method executes are those that physically remove nodes that are already logically deleted. Assume T runs the search method, and that we create an additional thread T_a and run it with the same input.

Consider a CAS in which T_a attempts to physically remove a logically deleted node from the list. Assume T_a successfully executes this CAS and removes the node from the list. Because the node was already logically deleted, this CAS does not affect the results of other operations. Thus, there exists an equivalent execution, in which this CAS is not successful (or not attempted at all.) To see that such an equivalent execution exists, consider the thread T_1 that marked this node as logically deleted in the first place. This thread must currently be attempting to physically remove the node so that it can exit the delete operation. An alternative execution in which T_1 is given the time, right before T_a executes the CAS, to physically remove the node, and only then does T_a attempt the considered CAS and fails, is equivalent.

It is important to realize that many methods, for example, the method that logically deletes a node from the list, are not parallelzable. If an additional thread executes CAS that logically deletes a node from the list, then this can affect the results of other operations. Thus, there exist some executions, that have no equivalent executions in which the additional thread does not successfully execute this CAS.

Parallelizable methods play an important role in our construction, since helping threads can run them unchecked. If a thread cannot complete a parallelizable method, helping threads may simply execute the same method as well.

We now focus on a different issue. In order to run the fast-path-slow-path methodology, there must be some means to identify the case that the fast path is not making progress on time, and then move to the slow path. To this end, we define the *Contention failure counter*. Intuitively, a contention failure counter is a counter associated with an invocation of a method (i.e. many invocations of the method imply separate counters), measuring how often the method is delayed due to contention.

Definition 3.5.5. (Contention failure counter.) A *contention failure counter* for a method M is an integer field C associated with an invocation of M (i.e. many invocations

of M imply many separate contention failure counters). Denote by C(t) the value of the counter at time t. The counter is initialized to zero upon method invocation, and is updated by the method during its run such that the following holds.

- (Monotonically increasing:) Each update to the contention failure counter increments its value by one.
- (Bounded by contention:) Assume M is invoked by Thread T and let $d(t)$ denote the number of data structure modifications by threads other than T between the invocation time and time t. Then it always hold that $C(t) \leq d(t)$. [2]
- (Incremented periodically:) The method M does not run infinitely many steps without incrementing the contention failure counter.

Remark. The contention failure counter can be kept in the local memory of the thread that is running the method.

A lock-free method must complete within a bounded number of steps if no modifications are made to the data structure outside this method. Otherwise, allowing this method to run solo results in an infinite execution, contradicting its lock-freedom. Thus, the requirements that the counter remains zero if no concurrent modifications occur, and the requirement that it does not remain zero indefinitely, do not contradict each other. The contention failure counter will be used by the thread running the method to determine that a method in the fast-path is not making progress and so the thread should switch to the slow path.

For most methods, counting the number of failed CASes can serve as a good *contention failure counter*. However, more complex cases exist. We further discuss such cases in Appendix E.

In order to help other threads, and in particular, execute CAS operations for them, we will need to have CASes published. For this publication act, we formalize the notion of a CAS description.

Definition 3.5.6. (CAS description.) A CAS *description* is a structure that holds the triplet $(addr, expected, new)$ which contains an address (on which a CAS should be executed), the value we expect to find in this address, and the new value that we would like to atomically write to this address if the expected value is currently there. Given a pointer to a CAS *description*, it is possible to *execute it* and the execution can be either successful (if the CAS succeeds) or unsuccessful (if the CAS fails).

3.6 Normalized Lock-Free Data Structures

In this section, we specify what a normalized lock-free data structure is. We later show how to simulate a normalized lock-free algorithm in a wait-free manner automatically.

[2] In particular, this implies that if no modifications were made to the data structure outside the method M since its invocation until time t, then $C(t) = 0$.

3.6.1 The Normalized Representation

A normalized lock-free data structure is one for which each operation can be presented in three stages, such that the middle stage executes the owner CASes, the first is a preparatory stage and the last is a post-execution step.

Using Harris's linked-list example, the DELETE operation runs a first stage that finds the location to mark a node as deleted, while sniping out of the list all nodes that were previously marked as deleted. By the end of the search (the first stage) we can determine the main CAS operation: the one that marks the node as deleted. Now comes the middle stage where this CAS is executed, which logically deletes the node from the list. Finally, in a post-processing stage, we attempt to snip out the marked node from the list and make it unreachable from the list head.

In a normalized lock-free data structure, we require that: any access to the data structure is executed using a read or a CAS; the first and last stages be parallelizable, i.e., can be executed with *parallelizable methods*; and each of the CAS primitives of the second stage be protected by versioning. This means that there is a counter associated with the field that is incremented with each modification of the field. This avoids potential ABA problems, and is further discussed in Section 3.7.

Definition 3.6.1. A lock-free data structure is provided in a normalized representation if:

- Any modification of the shared memory is executed using a CAS operation.
- Every *operation* of the data structure consists of executing three methods one after the other and which have the following formats.

  1) CAS**-generator**, whose input is the operation's input, and its output is a list of CAS-*descriptors*. The CAS-generator method may optionally output additional data to be used in the WRAP-UP method.

  2) CAS**-executor**, which is a fixed method common to all data structures implementations. Its input is the list of CAS-*descriptors* output by the CAS-generator method. The CAS-executor method attempts to execute the CASes in its input one by one until the first one fails, or until all CASes complete. Its output is the index of the CAS that failed (which is -1 if none failed).

  3) Wrap-Up, whose input is the output of the CAS-*executor method* plus the list of CAS-descriptors output by the CAS-generator, plus (optionally) any additional data output by the CAS-generator method to be used by the WRAP-UP method. Its output is either the operation's result, which is returned to the owner thread, or an indication that the operation should be restarted from scratch (from the GENERATOR method).
- The GENERATOR and the WRAP-UP methods are parallelizable and they have an associated *contention failure counter*.
- Finally, we require that the CASes that the GENERATOR method outputs be for fields that employ versioning (i.e., a counter is associated with the field to avoid

an ABA problem). The version number in the `expected-value` field of a CAS that the GENERATOR method outputs cannot be greater than the version number currently stored in the `target address`. This requirement guarantees that if the `target address` is modified after the GENERATOR method is complete, then the CAS will fail.

All lock-free data structures that we are aware of today can be easily converted into this form. Several such normalized representations are presented in Section 3.10. This is probably the best indication that this normalized representation covers natural lock-free data structures. In Section 3.9 we show that all abstract data types can be implemented in a normalized lock-free data structure, but this universal construction is likely to be inefficient.

Intuitively, one can think of this normalized representation as separating owner CASes (those are the CASes that must be executed by the owner thread) from the other (denoted auxiliary) CASes. The auxiliary CASes can be executed by many helping threads and therefore create *parallelizable methods*. Intuitively, the first (generator) method can be thought of as running the algorithm without performing the owner CASes. It just makes a list of those to be performed by the executor method, and it may execute some auxiliary CASes to help previous operations complete.

As an example, consider the DELETE operation of Harris's linked-list. When transforming it to the normalized form, the GENERATOR method should call the search method of the linked-list. The search method might snip out marked (logically deleted) nodes; those are auxiliary CASes, helping previous deletions to complete. Finally, the search method returns the node to be deleted (if a node with the needed key exists in the list). The CAS that marks this node as logically deleted is the owner CAS, and it must be executed exactly once. Thus, the GENERATOR method does not execute this owner CAS but outputs it to be executed by the CAS-EXECUTER method. If no node with the needed key is found in the list, then there are no owner CASes to be executed, and the GENERATOR method simply returns an empty list of CASes.

Next, the CAS-EXECUTOR method attempts to execute all these owner CASes. In Harris's linked list, like in most known algorithms, there is only one owner CAS. The CAS-EXECUTER method attempts the owner CAS (or the multiple owner CASes one by one), until completing them all, or until one of them fails. After the CAS-EXECUTER method is done, the operation might already be over, or it might need to start from scratch (typically if a CAS failed), or some other auxiliary CASes should be executed before exiting. The decision on whether to complete or start again (and possibly further execution of auxiliary CASes) is done in the WRAP-UP method. In Harris' linked-list example, if the GENERATOR method outputted no CASes, then it means that no node with the required key exists in the list, and the wrap-up method should return with failure. If a single CAS was outputted by the GENERATOR but its execution failed in the EXECUTER, then the operation should be restarted from scratch. Finally, if a single CAS

was outputted by the GENERATOR and it was successfully executed by the EXECUTER, then the wrap-up method still needs to physically remove the node from the list (an auxiliary CAS), and then return with success. Removing the node from the list can be done similarly to the original algorithm, by calling the SEARCH method again.

We note that the normalized representation requires all data structure modifications to be executed with a CAS, and allows no simple WRITE primitives. This is in fact the way most lock-free data structures work. But this requirement is not restrictive, since any WRITE primitive can be replaced by a loop of repeatedly reading the old value and then trying to CAS it to the new value until the CAS is successful.

To see that this does not foil the lock-free property, replace the WRITES with such loop CASes one by one. Now, for a single such replacement note that either the CASes always succeed eventually and then the algorithm is still lock-free, or there exists an execution of this loop that never terminates. In the later case, other threads must be executing infinitely many steps that foil the CASes, while the current thread never modifies the data structure. This is similar to a case where this thread is not executing at all, and then the other threads must make progress, since the algorithm (without the looping thread) is lock-free.

3.7 Transformation Details

In this section, we provide the efficient wait-free simulation of any normalized lock-free data structure. To execute an operation, a thread starts by executing the normalized lock-free algorithm with a *contention failure counter* checked occasionally to see if contention has exceeded a predetermined limit. To obtain non-starvation, we make the thread check its *contention failure counter* periodically, e.g., on each function call and each backward jump. If the operation completes, then we are done. Otherwise, the contention failure counter eventually exceeds its threshold and the slow path must be taken.

There is also a possibility that the *contention failure counter* never reaches the predetermined limit for any execution of a single method, but that the WRAP-UP method constantly indicates that the operation should be restarted from fresh. (This must also be the result of contention, because if an operation is executed alone in the lock-free algorithm it must complete.) Thus, the thread also keeps track of the number of times the operation is restarted, and if this number reaches the predetermined limit, the slow path is taken as well. The key point is that an operation cannot execute infinitely many steps in the fast-path. Eventually, it will move to the slow-path.

The slow path begins by the thread creating an `operation record` object that describes the operation it is executing. A pointer to this operation record is then enqueued in a wait-free queue called the `help queue`. Next, the thread helps operations on the `help queue` one by one according to their order in the queue, until its own operation is completed. Threads in the fast path that notice a non-empty `help queue`

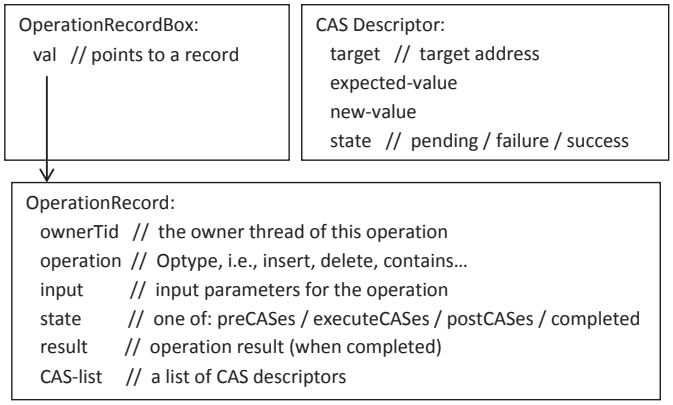

Figure 3.1: Operation Record

provide help as well, before starting their own fast-path execution.

3.7.1 The Help Queue and the Operation Record

The description of operations that require help is kept in a wait-free queue, similar to the one proposed by Kogan and Petrank in [KP11]. The queue in [KP11] supports the standard ENQUEUE and DEQUEUE operations. We slightly modify it to support three operations: ENQUEUE, PEEK, and CONDITIONALLY-REMOVE-HEAD. ENQUEUE operations enqueue a value to the tail of the queue as usual. The new PEEK operation returns the current head of the queue, without removing it. Finally, the `conditionally-remove-head` operation receives a value it expects to find at the head of the queue, and removes it (dequeues it) only if this value is found at the head. In this case it returns *true*. Otherwise, it does nothing and returns *false*. This queue is in fact simpler to design than the original queue, because DEQUEUE is not needed, because PEEK requires a single read, and the `conditionally-remove-head` can be executed using a single CAS. (Therfore, `conditionally-remove-head` can be easily written in a wait-free manner.) Some care is needed because of the interaction between ENQUEUE and CONDITIONALLY-REMOVE-HEAD, but a similar mechanism already appears in [KP11], and we simply used it in our case as well. The Java implementation for our variation of the queue is given in Appendix D.

We use this queue as the `help queue`. If a thread fails to complete an *operation* due to contention, it asks for help by enqueuing a request on the `help queue`. This request is in fact a pointer to a small object (the operation record box) that is unique to the operation and identifies it. It is only reclaimed when the operation is complete. In this operation record box object there is a pointer to the `operation record` itself, and this pointer is modified by a CAS when the operation's status needs to be updated. We specify the content of this object and record in Figure 3.1.

```
1: void help (boolean beingHelped, OperationRecordBox myHelpBox) {
2:     while (true) {
3:       OperationRecordBox head = helpQueue.peekHead();
4:       if (head != null)
5:         helpOp(head);
6:       if (!beingHelped || myHelpBox.get().state == OpState.completed)
7:         return;
8:     }
9:   }
```

Figure 3.2: The help method

3.7.2 Giving Help

When a thread T starts executing a new operation, it first PEEKs at the head of the `help queue`. If it sees a non-null value, then T helps the enqueued operation before executing its own operation. After helping to complete one operation, T proceeds to execute its own operation (even if there are more help requests pending on the queue).

To participate in helping an operation, a thread calls the HELP method, telling it whether it is on the fast path, and so willing to help a single operation, or on the slow path, in which case it also provides a pointer to its own operation record box. In the latter case, the thread is willing to help all operations up to its own operation. The HELP method will PEEK at the head of the `help queue`, and if it sees a non-null operation record box, it will invoke the HELPOP method. A null value means the help queue is empty, and so no further help is needed.

The HELPOP, invoked by the HELP method, helps a specific *operation* O, until it is completed. Its input is O's operation record box. This box may either be the current head in the `help queue` or it is an operation that has been completed and is no longer in the `help queue`. As long as the operation is not complete, HELPOP calls one of the three methods, PRECASES, EXECUTECASES, or POSTCASES, as determined by the operation record. If the operation is completed, HELPOP attempts to remove it from the queue using CONDITIONALLY-REMOVE-HEAD. When the HELPOP method returns, it is guaranteed that the operation record box in its input represents a completed operation and is no longer in the `help queue`.

The PRECASES method invokes the CAS-GENERATOR method of the normalized lock-free algorithm, which generates the list of CAS-descriptions for the CAS-EXECUTOR. As the CAS-GENERATOR method is parallelizable, it can be run by several threads concurrently at no risk[3]. It runs a monitored version of the generator, which occasionally checks the contention failure counter in order to guarantee this method will not run forever. If the contention failure counter reaches the predetermined threshold, the thread simply quits this method with null and reads the operation record box to see if another thread has made progress with this operation (if not, the HELPOP method will call the PRECASES method again).

[3]This is formally proved at Section 3.8.

```
 1: void helpOp(OperationRecordBox box) {
 2:       OperationRecord record = null;
 3:       do {
 4:         record = box.val;
 5:         OpState state = record.state;
 6:         if (state == OpState.preCASes) {
 7:           preCASes(box, record);        ▷ Executes the CAS generator supplied by the normalized
    algorithm plus attempt to make the result visible.
 8:         }
 9:         if (state == OpState.executeCASes) {
10:           int failedIndex = executeCASes(record.list);  ▷ carefully execute the CAS list outputted
    by the CAS generator.
11:           record.failedCasIndex = failedIndex;
12:           record.state = OpState.postCASes;
13:         }
14:         if (state == OpState.postCASes) {
15:           postCASes(box, record); ▷ execute the wrap-up method, plus some administrative work
16:         }
17:       } while (state != OpState.completed);
18:       helpQueue.conditionallyRemoveHead(box);
19:    }
```

Figure 3.3: The helpOp method

The preCASes method allocates a new operation record that holds the result of the run of the cas-generator method. The outcome of the preCASes can either be a null pointer if the method was stopped by the contention failure counter, or a list of cas-descriptors if the method completed successfully. If the result of the cas-generator execution is not a null, the preCASes method creates a new operation record and attempts to make it the official global operation record for this operation by attempting to atomically change the operation record box to reference it. There is no need to check whether this attempt succeeded as the cas-generator method is a parallelizable method and any result by any of its concurrent executions is a proper result that can be used to continue the operation.

If the `OperationRecord` is not replaced by a new one, then soon enough all threads will only run this method, all helping the same operation. In that case, it is guaranteed to be completed because the simulation is equivalent to running this operation solo[4]. After the *OperationRecord* is successfully replaced by a cas, some threads might still be executing the generator method. Since we monitor the execution with a contention failure counter, and since the counter is required to be incremented repeatedly (cannot maintain any value forever), then we know that these threads do will not execute infinitely many steps in these methods.

The cas-executor method is not parallelizable and therefore helping threads cannot simply run it concurrently. Only one execution of each cas is allowed, and it should be clear to everyone whether each cas execution succeeded or failed. So we replace it with a carefully designed concurrent method, named executeCASes (Figure

[4]A formal argument for the wait-freedom is given in Section 3.8.2.

```
1: void preCASes(OperationRecordBox box, OperationRecord record) {
2:       cas-list = MonitoredRun(Of GeneratorMethod on record);
3:       if (cas-list != null) {
4:         newRecord =
             new OperationRecord(record.ownerTid, record.operation, record.input, Op-
   State.executeCASes, null, cas-list);
5:         CAS(box.val, record, newRecord);
6:       }
7:     }
```

Figure 3.4: The preCASes method

3.5) .

The executeCASes method receives as its input a list of CAS-descriptors to be executed. Each CAS description is also associated with a `state` field, which describes the execution state of this CAS: succeeded, failed, or still pending. The controlled execution of these critical CASes requires care to ensure that: each CAS is executed exactly once, the success of the CAS gets published even if one of the threads stops responding, and an ABA problem is not created by letting several threads execute this sensitive CAS instead of the single thread that was supposed to execute it in the original lock-free algorithm. The ABA problem is introduced because a thread may be inactive for a while and then successfully execute a CAS that had been executed before, if after its execution the `target address` was restored back to its old value.

Ideally, we would have liked to execute three instructions atomically: (1) read the `state`, (2) attempt the CAS (if the `state` is pending), and (3) update the CAS `state`. Unfortunately, since these three instructions work on two different locations (the CAS's `target address` and the descriptor's `state` field) we cannot run this atomically without using a heavy mutual exclusion machinery that foils wait-freedom (and is also costly).

To solve this atomicity problem, we introduce both a versioning mechanism to the fields being CASed, and an additional bit, named `modification-bit`, to each CASed field. (In a practical implementation, the `modified-bit` is on the same memory word as the version number.)

The `modified-bit` will signify that a successful CAS has been executed by a helping thread, but (possibly) not yet reported. So when a CAS is executed in the slow path, a successful execution will put the new value together with the `modified-bit` set. As a result, further attempts to modify this field must fail, since the `expected-value` of any CAS never has this bit set. When a field has the `modified-bit` set, it can only be modified by a special CAS primitive designated to clear the `modified-bit`. This CAS, which we refer to as a ClearBit CAS, is the only CAS that is executed *without* incrementing the version number. It only clears the `modified-bit`, and nothing more. However, before any thread attempts the ClearBit CAS, it must first update the `state` of the CAS to reflect success.

Our transformation keeps the invariant that in the entire data structure, only a

single `modified-bit` might be set at any given moment. This is exactly the bit of the CAS that is currently being helped by all helping threads. Before clearing this `modified-bit`, no other CAS execution can be helped.

Let us examine an execution of the executeCASes method. The executing thread goes over the CASes in the list one by one, and helps execute them as follows. First, it reads the CAS `state`. If it is successful, it attempts the ClearBit CAS to clear the `modified-bit`, in case it hasn't been done before. The `expected-value` of the ClearBit CAS exactly matches the `new-value` of the CAS-descriptor except that the `modified-bit` is set. Thus, due to the version number, the ClearBit CAS can only clear a `modified-bit` that was switched on by the same CAS-descriptor. (This is formally proved in Section 3.8.1.)

Otherwise, if the CAS `state` is currently set to failure, then the executeCASes method immediately returns with the index of the failing CAS. Otherwise, the `state` is pending, and executeCASes attempts to execute the listed CAS and set the `modified-bit` atomically with it. Next, it checks whether the `modified bit` is set, and if it is, it sets the (separate) CAS `state` field to success and only then attempts to clear the `modified-bit.`

Setting the `state` field to success is done with an atomic CAS, which only succeeds if the previous `state` is pending. This is required to solve a race condition in which the execution of the CAS-descriptor has failed, yet the `modified-bit` is set to true is the result of a successful execution of a later CAS. Afterwards, and only if the `state` is now indeed success, the ClearBit is attempted. Next, if at that point the CAS `state` field is still not set to success, then it means the CAS has failed, and thus executeCASes sets this `state` to failure and returns. Otherwise, success is achieved and executeCASes proceeds to the next CAS in the list.

The existence of the `modified-bit` requires minor modifications to the fast-path. First, READ primitives should ignore the `modified-bit` (always treat it as if the bit were off.) This should be easy: the `modified-bit` is adjacent to the version number, which does not normally influence the execution (only when calculating the next version number for the `new-value` of a CAS.)

Second, when a thread attempts a CAS and the CAS fails in the fast-path, it should check to see whether the CAS failed because the `modified-bit` in the required field is set, and if so, whether the CAS would have succeeded were the bit turned off.

Thus, after a CAS in the fast-path fails, instead of continuing as usually, the thread that attempted the CAS READS the value from the CAS's `target address`. If this value differs from the CAS's `expected-value` in other bits than the `modified-bit`, then the thread continues the execution as usual, since the CAS has "legitimate" reasons for failure. However, if the value in the CAS's `target address` is identical to the CAS's `expected-value` in all the bits but the `modified-bit`, then the thread pauses its current execution and calls the help method to participate in helping the current operation to complete (clearing this bit in the process.)

```
 1: private void executeCASes(CAS-list cl) {
 2:       for (int i = 0; i < cl.size(); i++) {
 3:         ICasDesc cas = cl.get(i);
 4:         if (cas.GetState() == CasState.success) {
 5:           cas.ClearBit();
 6:           continue;
 7:         }
 8:         if (cas.GetState() == CasState.failure)
 9:           return i;
10:         cas.ExecuteCas();
11:         if (cas.ModifiedBitSet()) { ▷ Checks whether the modified bit in the target address is set.
12:           cas.CASStateField(CasState.pending, CasState.success); ▷ Attempt with a CAS to change
    the descriptor's state from pending to success.
13:           if (cas.GetState == CasState.success) { cas.ClearBit(); }
14:         }
15:         if (cas.GetState() != CasState.success) {
16:           cas.WriteStateField(CasState.failure); ▷ CAS MUST HAVE FAILED, SET THE DESCRIPTOR'S
    STATE TO FAILURE.
17:            RETURN i;
18:         }
19:       }
20:       RETURN -1;                          ▷ THE ENTIRE CAS-LIST WAS EXECUTED SUCCESSFULLY
21:   }
```

Figure 3.5: The executeCASes Method

After the help method returns the `modified-bit` is guaranteed to have been cleared. Thus, the CAS is attempted again, and the execution continues as usual from that point. Even if the re-execution fails, there is no need to READ the `target address` again. It is guaranteed that the value in the `target address` is now different from the CAS's `expected-value`: if the `modified-bit` is turned back on after being cleared, it can only be done together with incrementing the version number.

After the CASes are executed, the HELPOP method calls the POSTCASES method (Figure 3.6), which invokes the WRAP-UP method of the original lock-free algorithm. If the WRAP-UP method fails to complete due to contention, the monitored run will return null and we will read again the `operation record box`. If the WRAP-UP method was completed without interruption, the POSTCASES method attempts to make its private operation record visible to all by atomically attempting to link it to the `operation record box`. Note that its private `operation record` may indicate a need to start the operation from scratch, or may indicate that the operation is completed. When the control is returned to the HELPOP method, the record is read and the execution continues according to it.

3.8 Correctness

Our goal is to prove that given a normalized linearizable lock-free data structure implementation for a particular abstract data type, our transformation generates a wait-free linearizable implementation for the same abstract data type. As a preliminary

```
1: void postCASes(OperationRecordBox box, OperationRecord record) {
2:    shouldRestart, operationResult = MonitoredRun(of Wrapup Method on record);
3:    if (operationResult == Null) Return
4:    if (shouldRestart)
5:       newRecord = new OperationRecord(record.ownerTid, record.operation, record.input, Op-
   State.preCASes, null, null);
6:    else
7:       newRecord = new OperationRecord(record.ownerTid, record.operation, record.input, Op-
   State.completed, operationResult, null);
8:    box.val.compareAndSet(record, newRecord);
9: }
```

Figure 3.6: The postCASes Method

step, we first prove that the implementation of the executeCASes method, as given in Figure 3.5 is correct. The exact definition of a correct behavior of the executeCASes method is given in the following subsection (Definition 3.8.1). Subsection 3.8.1 proves that our implementation is indeed correct. Given this result, Subsection 3.8.2 proves that the generated algorithm of our transformation is linearizable and wait-free.

3.8.1 Correctness of the executeCASes Implementation

In the executeCASes method, potentially many threads are working together on the same input (same cas list). A cas-list is a structure that holds zero or more cas-descriptors, and a field indicating the length of the list. Each cas-descriptor consists of four fields: `target address`, `expected-value`, `new-value`, and `status`. The three first fields are final (never altered) after a cas-descriptor has been initialized.

Loosely speaking, to an "outside observer" that inspects the shared memory, many threads executing the executeCASes method on a certain cas-list should appear similar to a single thread executing the cas-executer method (the second method in the normalized form) on a private (but identical) cas-list input. Recall that in the cas-executer method, the cases are executed according to their order until they are completed or the first one among them fails. The output of the method is the index of the first cas that failed, or minus one if no cas failed.

The main difference between an execution of the cas-executer method by a single thread, and concurrent executions of the executeCASes method by many threads, is that in the latter each cas is executed in two steps. We refer to the first (main) step simply as executing the cas-descriptor, and to the second step as executing the ClearBit of the cas-descriptor. An execution of a cas-descriptor (which occurs in line 10 of the executeCASes method) is an execution of a cas for which the `target address`, `expected-value` and `new-value` are the same as the cas-descriptor's, except that the `new-value` is altered such that the `modified-bit` is set. An execution of a ClearBit of a cas-descriptor (which occurs in lines 5 and 13) is an execution of a cas for which the `target address` and the `new-value` are the same as the cas-descriptor's,

and the `expected-value` is identical to the `new-value` except that the `modified-bit` is set. (Thus, the `expected-value` of the second step is the `new-value` of the first step.)

In what follows, we formally define what is a correct concurrent behavior for the executeCASes method and prove that the implementation of it given in Figure 3.5 is indeed correct. The correctness of the transformation, as detailed in subsection 3.8.2, relies on the correctness of the executeCASes method stated here. However, the two proofs are independent of each other, and the reader may skip the proof in this subsection if he chooses to, without loss of clarity.

Definition 3.8.1. (Correct Behavior of the executeCASes method.) When one or more threads execute concurrently the executeCASes method using the same cas-list input, the following should hold.

- All computation steps inside the executeCASes method are either: a) an execution of a cas-descriptor, b) a ClearBit of a cas-descriptor, or c) applied on the memory of the cas-list (e.g., altering the `state` field of a cas-descriptor).
- For every cas-descriptor c: a) any attempt to execute c except the first attempt (by some thread) must fail, and b) any attempt to execute the ClearBit of c except the first attempt (by some thread) must fail.
- Before a cas-descriptor c in a cas-list cl is executed for the first time: a) all the previous cas-descriptors in cl have been *successfully* executed, and b) ClearBit has already been executed for all the previous cas-descriptors in cl.
- Once some thread has completed executing the executeCASes method on an input cas-list cl the following holds.
  1) Either all the cas-descriptors have been successfully executed, or all the cas-descriptors have been executed until the first one that fails. Further cas-descriptors (after the first one that fails) have not been executed, and will not be executed in the rest of the computation.
  2) A ClearBit was successfully executed for each cas-descriptor that was successfully executed.
- The return value of the executeCASes for *every* thread that completes it is:
  1) The index of the first (and only) cas-descriptor whose execution failed the first time it was attempt, if such exists.
  2) -1 otherwise.

Our goal is to prove that the executeCASes method as implemented in Figure 3.5 is correct by Definition 3.8.1, assuming that its input is legal. More precisely, we consider an execution E in which the executeCASes method is invoked (possibly many times). We use several assumptions on E, (all fulfilled by an execution of an algorithm

that results from applying our transformation on a normalized form algorithm) about how the executeCASes method is used, and prove that E fulfills definition 3.8.1. We assume the following.

Assumption 3.8.2. Only a single CAS-list for which the execution (by some thread) is not yet completed is active at any given moment. More precisely: whenever the executeCASes method is invoked in E with an input CAS-list cl, then for all prior invocations of the executeCASes method with an input CAS-list cl', by any thread, one of the following holds.
1) cl and cl' are equal.
2) An execution of the executeCASes method for which the input was cl' is already completed.

Remark. Note that we do not assume that *all* executions of the executeCASes method with input cl' are already completed.

Assumption 3.8.3. Any address that is used as a `target address` of any CAS-descriptor is only ever modified in E with a CAS (no writes). Outside the executeCASes method, all the CASes that modify this address has the `modified-bit` off both in the `expected-value` and in the `new-value`.

Assumption 3.8.4. A version number is associated with every address that is used as a `target address` of a CAS-descriptor. For every CAS in E that attempts to modify such an address outside the executeCASes method, the version number of the `new-value` is greater by one than the version number of the expected value. (That is, each successful CAS increments the version number by one.)

Assumption 3.8.5. CAS-descriptors are initialized with a pending `state`, and the `state` field is never modified outside the executeCASes method.

Assumption 3.8.6. When a CAS-descriptor is initialized, the version number in the `expected-value` field is no greater than the current version number stored in the `target address` of the CAS. (That is, CAS-descriptors are not created speculatively with "future" version numbers.)

Remark. Usually in a CAS, the `expected-value` is a value that was previously read from the `target address`. If that is the case, this assumption will always hold.

To simplify the proof, we first define a few terms used in the proof. First, we define a total order between all the CAS-lists that are used as an input to the executeCASes method in E, and to all the CAS-descriptors used in these CAS-lists.

Definition 3.8.7. (Total order of CAS-lists.) Given two different CAS-lists cl_1 and cl_2 used as an input to the executeCASes method in E, we say that cl_1 is before cl_2 (or

prior to cl_2) if the first time that an executeCASes method with input cl_1 is invoked in E is prior to the first time that an executeCASes method with input cl_2 is invoked in E.

Remark. Note that by Assumption 3.8.2, if cl_1 is prior to cl_2, then some thread completes executing the executeCASes method on input cl_1 before the first time that the executeCASes method is invoked with cl_2.

Definition 3.8.8. (Total order of CAS-descriptors.) Given a CAS-descriptor c_1 that belongs to a CAS-list cl_1, and a CAS-descriptor c_2 that belongs to a CAS-list cl_2, we say that c_1 is before c_2 (or prior to c_2) if either: 1) cl_1 is before cl_2, or 2) cl_1 and cl_2 are equal, and c_1 appears before c_2 in the CAS-list.

Next, we define the most recent CAS-list, most recent executeCASes iteration, and most recently active CAS-descriptor for a given point in time t. For an execution E, time t is the point in the execution after exactly t computation steps.

Definition 3.8.9. (Most recent CAS-list, most recent executeCASes iteration, most recently active CAS-descriptor) At time t, the *most recent* CAS-*list* cl is the latest CAS-list (Definition 3.8.7) such that an executeCASes method is invoked with cl as an input before time t. The *most recent* executeCASes *iteration* at time t is the latest iteration (with the largest i variable) of the loop in lines 2–17 of the executeCASes method that any thread was executing at or before t on the most recent cl of time t. The *most recently active* CAS-*descriptor* is the CAS-descriptor that is read at the beginning of the most recent executeCASes iteration.

Remark. Note that if the first time the executeCASes method is invoked in E is after time t, then the most recent CAS-list, most recent executeCASes iteration, and most recently active CAS-descriptor are undefined for time t.

Definition 3.8.10. (`modified-bit` belongs to a CAS-descriptor.) We say that a `modified-bit` that is true at time t *belongs* to the CAS-descriptor whose execution switched this bit to true most recently prior to t. (Note that a `modified-bit` can only be set to true in line 10 of the executeCASes method. (Assumption 3.8.3.))

Claim 3.8.11. *At any point in the computation, if a* `modified-bit` *is on, it belongs to some* CAS-*descriptor.*

Proof. By Assumption 3.8.3, a `modified-bit` cannot be switched on outside of the executeCASes method. Inside the executeCASes method, it can only be switched on by executing a CAS-descriptor in line 10. It follows from Definition 3.8.10 that when a `modified-bit` is on, it belongs to some CAS-descriptor. □

In what follows we state several invariants that are true throughout execution E. After stating them all, we will prove them using induction on the computation steps of the execution. The induction hypothesis is that all the following invariants are correct after i computation steps, and we shall prove they all hold after $i+1$ computation steps. When proving that an invariant holds for $i+1$ steps, we will freely use the induction hypothesis for any one of the invariants, and may also rely on the fact that previously proved invariants hold for $i+1$ steps. All the invariants trivially hold for $i=0$ steps: the first invariant holds since by Assumption 3.8.5 all CAS-descriptors are initialized as pending, and the rest of the invariants hold for $i=0$ steps vacuously, since they refer to a condition that is always false before a single execution step is taken.

Invariant 3.1. The `state` of a CAS-descriptor that has not yet been executed is pending.

Invariant 3.2. If the `state` of a CAS-descriptor is failure, then the first attempt to execute the CAS-descriptor has already occurred, and it has failed.

Invariant 3.3. If the `state` of a CAS-descriptor is success, then the first attempt to execute the CAS-descriptor has already occurred, and it has succeeded.

Invariant 3.4. If a CAS-descriptor's `state` is not pending (i.e., either success or failure), then it is final (never changes again).

Invariant 3.5. An attempt to execute a particular CAS-descriptor in a given CAS-list, except the first attempt by the first thread that attempts it, must fail.

Invariant 3.6. If some thread t is currently executing the nth iteration of the loop in some instance of the executeCASes method (formally: if the last computation step taken by t is inside the nth iteration of the loop), then the `states` of the CAS-descriptors read in iterations 0 to $n-1$ of the same executeCASes instance are success.

Invariant 3.7. If a CAS-descriptor c in a CAS-list cl has been executed, then the `states` of all the previous CAS-descriptors in cl are success.

Invariant 3.8. If the `state` of a CAS-descriptor c in a CAS-list cl is not pending, then the `states` of all the previous CAS-descriptors in cl are success.

Invariant 3.9. If some thread t has already completed the execution of an executeCASes method with input CAS-list cl, then either 1) the `states` of all the CAS-descriptors in cl are success, or 2) the `state` field of exactly one CAS-descriptor c in cl is failure, the `states` of all the CAS-descriptors before c in cl (if any) are success, and the `states` of all the CAS-descriptors after c in cl (if any) are pending.

Invariant 3.10. If a CAS-descriptor has already been successfully executed, then one of the following holds.

1) The CAS-descriptor's `state` field indicates success, or
2) The CAS-descriptor's `state` field indicates a pending state, and the `target address` of the CAS still holds the CAS's `new-value`, and in particular, the `modified-bit` is set to true.

Invariant 3.11. If some thread t is currently executing the loop in lines 2–19 (formally: if the last execution step taken by t is inside the loop), in which the CAS-descriptor c is read, but the iteration t is executing is not the most recent EXECUTECASES iteration, (which means that c is not the most recently active CAS-descriptor), then c's state is not pending.

Invariant 3.12. If some thread t has already completed executing the loop in lines 2–19 (either by breaking out of the loop in line 9 or 17, or by continuing to the next loop from line 6, or simply by reaching the end of the iteration), in which the CAS-descriptor c is read, then there is no `modified-bit` that is set to true and that belongs to c.

Invariant 3.13. If a certain `modified-bit` is true, then this `modified-bit` belongs to the most recently active CAS-descriptor.

Proof. (Invariant 3.1.) Each CAS-descriptor is initialized as pending, and its `state` can potentially be changed only in lines 12 and 16 of the EXECUTECASES method. Before a thread t executes one of these lines for a certain CAS-descriptor, it first attempts to execute the same CAS-descriptor in line 10. Thus, if a CAS-descriptor has never been executed, its `state` must be pending. □

Proof. (Invariant 3.2.) Assume by way of contradiction that in step $i+1$ a thread t sets a CAS-descriptor c's `state` to failure, and that the first attempt to execute c has not yet occurred or has been successful. Step $i+1$ must be an execution of line 16, since this is the only line that sets a `state` field to failure (Assumption 3.8.5). Consider the execution right after t executed line 10 of the same iteration of the loop in lines 2–19. t has just executed c, so it is impossible that c has not yet been executed. Thus, the first attempt to execute c must have been successful.

By the induction hypothesis (Invariant 3.10), in each computation step after c was first executed (and in particular, after thread t executed it in line 10), and until step i, c's `state` is either success, or it is pending and the `modified-bit` is set to true. Thus, when t executes line 11, there are two cases.

The first case is that c's `state` is success. Since there is no code line that changes a `state` back to pending, and since until step $i+1$ the `state` cannot be failure by the induction hypothesis (Invariant 3.10), then the `state` must also be success when t executes line 15. Thus, the condition in this line is false, line 16 is not reached, and t cannot set c's `state` to failure at step $i+1$, yielding contradiction for the first case.

The second case is that c's `state` field is pending and that the `modified-bit` is set. In that case, t will attempt by a CAS to switch the `state` from pending to success in

line 12. After executing this line, c's `state` must be success (since it cannot be failure by the induction hypothesis (Invariant 3.10), and if it were pending the CAS would have changed it to success). Similarly to the previous case, the `state` must also be success when t executes line 15, and thus line 16 is not reached, yielding contradiction for the second case. □

Proof. (Invariant 3.3.) Assume by way of contradiction that in step $i+1$ a thread t sets a CAS-descriptor c's `state` to success, and that the first attempt to execute c has not yet occurred or has been unsuccessful. Step $i+1$ must be an execution of line 12, since this is the only line that sets a `state` field to success (Assumption 3.8.5). t has already executed line 10 of the same iteration of the loop, thus the first attempt to execute the CAS-descriptor has already occurred, and thus it must have failed.

Consider the execution when t executes line 11 of the same iteration of the loop. The `modified-bit` must have been on, otherwise line 12 would not have been reached. By Claim 3.8.11, this `modified-bit` must belong to a CAS-descriptor. We consider three cases. The first case is that the `modified-bit` belongs to c. In this case c's first execution attempt must have been successful, yielding contradiction.

The second case is that the `modified-bit` belongs to a CAS-descriptor prior to c. However, when t executes line 11, then by the induction hypothesis (Invariant 3.13), the `modified-bit` must belong to the most recently active CAS-descriptor. Since c is active at that point, then any CAS-descriptor prior to c cannot be the most recently active one by definition, and thus the `modified-bit` cannot belong to it, yielding contradiction for the second case.

The third case is that the `modified-bit` belongs to a CAS-descriptor that comes after c. Thus, by the induction hypothesis (Invariant 3.11), after i computation steps c's `state` cannot be pending. (t is executing the loop in lines 2–19 after i steps, but c cannot be the most recently active CAS-descriptor since a later CAS-descriptor has already been active to set the `modified-bit` to true.) If c's `state` is not pending after i steps, then t cannot set it to success in step $i+1$ via an execution of line 12, yielding contradiction for the third case. □

Proof. (Invariant 3.4.) This follows directly from Invariants 3.2 and 3.3, which are already proven for $i+1$ steps. That is, if c's `state` is failure after i steps, then by Invariant 3.2, the first attempt to execute c must have failed. Thus, by Invariant 3.3, the `state` cannot be success after $i+1$ steps. Similarly, if c's `state` is success after i steps, then by Invariant 3.3, the first attempt to execute c must have succeeded. Thus, by Invariant 3.2, the `state` cannot be failure after $i+1$ steps. Finally, a `state` cannot be changed from success or failure to pending, because no line in the executeCASes method changes a `state` to pending, and by Assumption 3.8.5, no line in the code outside the executeCASes does that either. □

Proof. (Invariant 3.5.) Assume that in step $i+1$ a CAS-descriptor c is attempted, and this is not the first attempt to execute this CAS. We shall prove this attempt must fail. By Assumption 3.8.4, each CAS-descriptor is to a `target address` that is associated with a version number. Furthermore, by combining Assumption 3.8.4 with Assumption 3.8.6, the version number of the `expected-value` is never greater than the current value stored in the `target address`. Thus, we consider two cases. The first case is that the first attempt to execute a c had succeeded. In this case, after this execution, the version number is greater than the expected-value's version number, and thus the attempt to execute it again in step $i+1$ must fail.

The second case is that the first attempt to execute a c had failed. If it failed because at the time of the attempt the version number stored in the `target address` had already been greater than the version number of the `expected-value`, then this must still be true, and the attempt to execute c in step $i+1$ must also fail. If the first attempt to execute c failed because even though the version numbers matched, the value stored in the `target address` differed from that of the `expected-value`, and the difference was not limited to the `modified-bit`, then in order for the execution attempt in step $i+1$ to succeed the value stored in the `target address` must then be changed, but in such a case the version number must be incremented, and thus again c's execution in step $i+1$ is doomed to failure.

The last possibility is that the first attempt to execute c had failed only because the `modified-bit` was set to true at the time. Since the `modified-bit` can be switched off by executing a ClearBit *without* incrementing the version number, this could theoretically allow c to be successfully executed later. However, this is impossible. Consider c's first execution. Since this happens before step $i+1$, then by the induction hypothesis (Invariant 3.13), if the `modified-bit` was set, the `modified-bit` must belonged to the most recently active CAS-descriptor. This cannot be c, since c was not successfully executed at the time. Thus, by the induction hypothesis (Invariant 3.11) c's state at the time was not pending. And thus, by Invariants 3.2 and 3.3, c must have been executed before, and this cannot be c's first execution. □

Proof. (Invariant 3.6.) To reach the nth iteration, t must have first completed iterations 0 to $n-1$. Consider t's execution of line 15 for each of these iterations. In this line, the `state` of the CAS-descriptor that is read in the same iteration is checked. If the `state` is set to success, then by Invariant 3.4 (which is already proved for $i+1$ steps), the `state` is also success after $i+1$ steps, and we are done. If the `state` is not success, then t will break out of the loop in line 15, and the nth iteration would not be reached. □

Proof. (Invariant 3.7.) By the induction hypothesis for the same invariant (Invariant 3.7), the invariant holds after i steps. Assume by way of contradiction that the invariant does not hold after $i+1$ steps. Thus, the $i+1$-st step must be one of the following.
1) A thread t executes a CAS-descriptor c in a CAS-list cl while the `state` of a previous CAS-descriptor in cl is not success.

2) The `state` of a CAS-descriptor c_2 in a CAS-list cl changes from success to a different value, while a later CAS-descriptor in cl has already been executed.

The first case yields contradiction because if t is executing a CAS-descriptor c, then the `states` of all the previous CAS-descriptors in the same list must be success by Invariant 3.6, which is already proved for $i+1$ steps. The second case yields a contradiction because a non-pending state is final by Invariant 3.4, which is also already proved for $i+1$ steps. □

Proof. (Invariant 3.8.) If the `state` of a CAS-descriptor c is not pending, then c has already been executed by Invariant 3.1 (which is already proved for $i+1$ steps). If c has already been executed, then the `states` of all the previous CAS-descriptors in the same cl are success by Invariant 3.8 (which is also already proved for $i+1$ steps). □

Proof. (Invariant 3.9.) By the induction hypothesis for the same invariant (Invariant 3.9), the invariant holds after i steps. Assume by way of contradiction that the invariant does not hold after $i+1$ steps. Thus, the $i+1$-st step must be one of the following.
1) A thread t completes the execution of the executeCASes method on input CAS-list cl, yet cl does not meet the requirements.
2) A thread t changes the `state` field of a CAS-descriptor in a CAS-list cl that met the requirements after i steps. (And this cl was used as an input to an executeCASes invocation that is already completed.)

Consider the first possibility, and in particular, consider which computation step could be the last computation step that t executes when completing the execution of the executeCASes method on input cl. For each of them, we will demonstrate that after it, cl must meet the requirements of Invariant3.9, thus reaching contradiction for the first possibility. The last computation step in an execution of the executeCASes method can be one of the following. a) Reading a failure value out of a CAS-descriptor's `state` field and breaking out of the loop (lines 8-9)[5]. Thus, by Invariant 3.8, which is already proved for $i+1$ steps, the fact that the CAS-descriptor's `state` field is failure (not pending), proves that the `states` of all the previous CAS-descriptors in the list are success, and the fact that the CAS-descriptor's `state` field is failure (not success), proves that the `states` of all the later CAS-descriptor in the list are pending.

b) Writing a failure value to a CAS-descriptor's `state` field and breaking out of the loop (lines 16-17). Again, by Invariant 3.8, the fact that the CAS-descriptor's `state` is failing implies that earlier CAS-descriptors's `states` are success and later CAS-descriptor's `states` are pending.

c) attempting to clear the `modified-bit` and "continuing" after the last iteration of the loop in lines 5-6. In this case, the fact that the condition in line 4 was true implies that the `state` of the last CAS-descriptor in the list was success, and by Invariant 3.4, which is already proved to $i+1$ steps, the `state` of the last CAS-descriptor must still be

[5]note that breaking out of the loop is not a computation step by itself, since it is neither a READ, WRITE or CAS to the shared memory, but just an internal computation.

success after $i+1$ steps. Thus, using Invariant 3.8, which is also proved for $i+1$ steps, the `states` of all the previous CAS-descriptors must be success as well.

d) Reading a success value out of the last CAS-descriptor in a CAS-list and finishing the last loop iteration (line 15). In this case, again, the `state` of the last CAS-descriptor is success, and thus, using Invariant 3.8, the `states` of all the previous CAS-descriptors are also success. In all of the cases (a)-(d), the CAS-list meets the requirements of Invariant 3.9, and thus the invariant is not violated, yielding contradiction for the first possibility.

Now consider the second possibility. By Invariant 3.4, which is already proved for $i+1$ steps, if the `state` of a CAS-descriptor is not pending then it never changes again. Thus, in step $i+1$ thread t must be changing the `state` of c from pending to a different value. However, since cl met the requirements of Invariant 3.9 for a CAS-list used as input for a completed executeCASes method after i steps, and yet c, which belongs to cl, has it `state` set to pending, it means that after i steps there must be a CAS-descriptor in cl before c, whose `state` is failure. By Invariant 3.8, which is already proved for $i+1$ steps, after $i+1$ steps, if a CAS-descriptor's `state` is not pending, then the `states` of all previous CAS-descriptors in the same CAS-list are success. Thus, changing c's state to anything other than pending in step $i+1$ yields contradiction. □

Proof. (Invariant 3.10.) Assume by way of contradiction that in step $i+1$ thread t executes a step that violates Invariant 3.10 for a CAS-descriptor c. By using the induction hypothesis for the same Invariant 3.10, such a step must be one of the following.
1) A successful execution of c (after which neither of the post conditions holds).
2) Changing c's `state` field either from pending to failure, of from success to a different value.
3) Changing the value stored in c's `target address` from the `new-value` with a set `modified-bit` to a different value (while the `state` is pending).

We will go over each of these possibilities. In the first case, step $i+1$ must be the first execution of c (by Invariant 3.5, which is already proved for $i+1$ steps). Thus, by the induction hypothesis (Invariant 3.1) c's `state` must be pending after i steps. Thus, after $i+1$ steps, c's state is still pending (since executing c does not change its `state` field), and since the execution in step $i+1$ is successful, then after $i+1$ steps the value stored in c's `target address` is c's `new-value`, with the `modified-bit` set. It follows that after step $i+1$ Invariant 3.10 still holds, yielding contradiction for the first case.

Consider the second case. Recall we assumed that step $i+1$ violates Invariant 3.10. For the second case (i.e., a change of c's state field) to violate the invariant, c must have been successfully executed at some step before step $i+1$. By Invariant 3.5, any attempt but the first attempt to execute c cannot be successful. Thus, the first attempt to execute c must have been successful. It follows that step $i+1$ cannot change the `state` of c to failure, by using Invariant 3.2, which is already proved for $i+1$ steps. Furthermore, step $i+1$ also cannot change c's state to pending, simply because no line

in the code does that, yielding contradiction for the second case.

Finally, consider the third case. By Assumption 3.8.3, changing the value stored in a `target address` of any CAS-descriptor, while the `modified-bit` is set, cannot be done outside the executeCASes method. The only places in the code where the contents of an address with a set `modified-bit` can be switched are the ClearBit instructions in lines 5 and 13. However, note that in order to reach a contradiction, we need to refer both to the possibility that step $i+1$ changes the value stored in c's `target address` because it is an execution of the ClearBit of c, and that step $i+1$ changes the value stored in c's `target address` because it is an execution of a ClearBit of a different CAS-descriptor c', that shares the same `target address`.

If step $i+1$ is a ClearBit of c, then in order to execute it either in line 5 or 13, c's `state` must be previously checked and found to be success. By using the induction hypothesis (Invariant 3.4) the state of c must still be success after i steps, and since changing the value stored in the target address does not change the `state`, then also after $i+1$ steps. Thus, the invariant holds after step $i+1$, yielding contradiction for this particular sub-case of the third case.

Now consider the possibility that step $i+1$ is a ClearBit of a CAS-descriptor c' different than c that shares the same `target address`. By the assumption of the third case, the value stored in the `target address` after i computation steps is the `new-value` of c with the `modified-bit` set. Thus, in order for the ClearBit of c' to successfully change this value, c and c' must both have the exact same `new-value`, including the version number. Thus, it is impossible for both c and c' to be executed successfully, since the first one of them that is executed successfully increments the version number. We assumed (contradictively) that c was executed successfully, and thus c' cannot be successful. Thus, by the induction hypothesis (Invariant 3.3) the `state` of c' cannot be success in the first i computation steps, and thus a ClearBit instruction of c' cannot be reached for the $i+1$-st step, completing the contradiction.□

Proof. (Invariant 3.11.) Assume by way of contradiction that after $i+1$ steps 1) thread t_1 is executing the loop in lines 2–17 in which the CAS-descriptor c is read, 2) c's `state` is pending, and 3) c is not the most recently active CAS-descriptor. By the induction hypothesis for the same invariant (Invariant 3.11), one of these three is not true after i steps. Thus, one of the following holds.

1) In step $i+1$ t_1 starts executing a new iteration of the loop in lines 2–17. (This could also be the first iteration in a new executeCASes invocation.) c is the CAS-descriptor for this new iteration, c's `state` is pending, and c is not the most recently active CAS-descriptor.

2) In step $i+1$ c's state is changed back to pending.

3) In step $i+1$ a thread t_2 starts executing a new iteration of the loop in lines 2–17 (possibly the first iteration in a new executeCASes invocation), thus making c no longer the most recently active CAS-descriptor.

We consider each of these cases. In the first case, let t_2 be the thread that executed (or is executing) an iteration that is after the iteration t_1 is currently executing. (If no such thread exists, then c is the most recently active CAS-descriptor and we are done. Also, note that we do not assume $t_1 \neq t_2$.) If t_2 is executing (or was executing) a later iteration than t_1 is currently executing, then we consider two possibilities. The first possibility is that t_2 is executing (or was executing) a later iteration on the same CAS-list that t_1 is iterating on. This case leads to a contradiction because c's `state` cannot be pending by Invariant 3.6, which is already proved for $i+1$ iterations. The second possibility is that t_2 is iterating (or was iterating) on a different cl than t_1 is currently iterating on. Thus, by Assumption 3.8.2, some thread already completed the execution of an executeCASes method with cl as the input. This leads to a contradiction because by Invariant 3.9, which is already proved for $i+1$ steps, either the `states` of all the CAS-descriptor are success (and then c's `state` cannot be pending), or that there is a CAS-descriptor with a `state` failure before c (and then, by using Invariant 3.6, t_1 cannot be executing the iteration in which c is read).

We now turn to consider the second case. This case yields a contradiction immediately, because no line of code inside the executeCASes changes a state back to pending, and by Assumption 3.8.5, no line of code outside the executeCASes method does that either.

Finally, we consider the third case. The proof here is very similar to the first case. We consider two possibilities. The first possibility is that t_2 is executing a later iteration on the same CAS-list that t_1 is iterating on. This case leads to a contradiction because c's `state` cannot be pending by Invariant 3.6, which is already proved for $i+1$ iterations. The second possibility is that t_2 is iterating on a different cl than t_1 is iterating on. Thus, by Assumption 3.8.2, some thread already completed the execution of an executeCASes method with cl as the input. This leads to a contradiction because by Invariant 3.9, which is already proved for $i+1$ steps, either the `states` of all the CAS-descriptor are success (and then c's `state` cannot be pending), or that there is a CAS-descriptor with a `state` failure before c (and then, by using Invariant 3.6, t_1 cannot be executing the iteration in which c is read). □

Proof. (Invariant 3.12.) By the induction hypothesis for the same invariant (Invariant 3.12), the invariant holds after i steps. Assume by way of contradiction that the invariant does not hold after $i+1$ steps. Thus, the $i+1$-st step must be one of the following.
1) A thread t_2 successfully executes a CAS-descriptor c (line 10), while a different thread t has already completed a loop iteration in which c was read.
2) A thread t completes the execution of an iteration in which c is read, while there is still a `modified-bit` that is set to true and that belongs to c.

If the first case is true, then by Invariant 3.5, which is already proved for $i+1$ steps, step $i+1$ must be the first step in which c is executed. Consider t's execution of the iteration in which c is read. If t reached line 6, then c's `state` much have been

success, which by the induction hypothesis (Invariant 3.3) means c had been executed before. If t reached line 9, then c's `state` must have been failure, which by the induction hypothesis (Invariant 3.2) also means c had been executed before. If t did not complete the loop in either line 6 or 9, then t must have reached and executed line 10, which again means that c was executed before step $i+1$. Whichever way t completed the iteration, CAS-descriptor c must have been executed before step $i+1$, thus it cannot be executed successfully in step $i+1$, yielding contradiction for the first case.

If the second case is true, then consider the different possibilities for t to complete the loop. If t breaks out of the loop in line 9 or in line 17, then c's `state` is failure. By Invariant 3.2, which is already proved for $i+1$ steps, this means the first attempt to execute c was not successful. By Invariant 3.5, it follows that no execution of c is successful until step $i+1$. It follows that there is no `modified-bit` that belongs to c, yielding contradiction for this sub-case of the second case.

If t completes the loop via the continue in line 6 then in t's last execution step inside the loop (which is assumed to be step $i+1$ of the execution) t attempts by a CAS to clear the `modified-bit`. If the `modified-bit` is previously set to true and belongs to c, then the value stored in c's `target address` is the same as the `expected-value` for the ClearBit CAS, and the `modified-bit` will be successfully cleared, yielding contradiction for this sub-case of the second case.

If t completes the loop by reading a success value out of c's `state` field and then reaching the end in line 15, then consider the execution when t executes line 11 of the same iteration. If the `modified-bit` is off at that time, then a `modified-bit` cannot belong to c at step $i+1$, since c has already been executed at least once, and thus further attempts of it until step $i+1$ must fail (Invariant 3.5). If the `modified-bit` is on, then t will reach line 12. When t executes the CAS in this line, then either the `state` is changed from pending to success, either the `state` is already success (the `state` cannot be failure, otherwise t would not have read a success value from it in line 15, because a non-pending state is final (by the induction hypothesis (Invariant 3.4). It follows that when t reached line 13, it attempted a ClearBit CAS to clear the `modified-bit`. If the `modified-bit` is previously set to true and belongs to c, then the value stored in c's `target address` is the same as the `expected-value` for the ClearBit CAS, and the `modified-bit` will be successfully cleared, yielding contradiction. □

Proof. (Invariant 3.13.) By the induction hypothesis for the same invariant (Invariant 3.13), the invariant holds after i steps. Assume by way of contradiction that the invariant does not hold after $i+1$ steps. Thus, the $i+1$-st step must be one of the following.
1) A thread t successfully executes a CAS-descriptor c (line 10), while c is not the most recently active CAS-descriptor.
2) A thread t starts a new iteration of the loop in lines 2–17, thus making c no longer the most recently active CAS-descriptor, while a `modified-bit` that belongs to c is on.

Consider the first case. Since c is successfully executed at step $i+1$, then by Invariant

3.5, which is already proved for $i+1$ steps, this must be the first attempt to execute c. Thus, by using the induction hypothesis (Invariant 3.1), c's `state` must be pending. Thus, by the fact that t is currently executing the loop iteration in which c is read, and by using the induction hypothesis (Invariant 3.11), c is the most recently active CAS-descriptor, yielding contradiction for the first case.

Now consider the second case. We claim that since t starts an iteration that is after the iteration in which c is read, then *some* thread t' (which may be t) has previously completed an iteration of the executeCASes method in which c is read. To see this, consider the iteration that t starts. If it is a later iteration on the same CAS-list to which c belong, then t itself must have completed the iteration in which c is read (thus, $t' = t$). If it is a later iteration on a different CAS-list, then by Assumption 3.8.2, some thread (which is t') has already completed an execution of the executeCASes method on the CAS-list to which c belong. To complete the executeCASes method, t' must either complete the iteration in which c is read, or break out of the loop earlier. However, t' cannot break out of the loop earlier, because that requires a CAS-descriptor with a failure `state` to be in the CAS-list before c, and if that were the case, then by the induction hypothesis (Invariant 3.7) c could not have been executed, and thus there could not have been a `modified-bit` belonging to c. To conclude, some thread t' has completed an iteration of the executeCASes method in which c is read. It follows by Invariant 3.12, which is already proved for $i+1$ steps, that there is no `modified-bit` belonging to c, yielding contradiction. □

At this point, Invariants 3.1–3.13 are all proved to be correct throughout the execution. Relying on these invariants, we now complete the proof for the correctness of the executeCASes method.

Observation 3.8.12. All execution steps inside the executeCASes method are either: a) an execution of a CAS-descriptor, b) a ClearBit of a CAS-descriptor, or c) applied on the memory of the CAS-list.

Proof. True by observing the code. Line 10 (execution of a CAS-descriptor) and lines 5,13 (ClearBit of a CAS-descriptors) are the only lines that execute on shared memory that is not inside the CAS-list. The other computation steps either read a `state` field of a CAS-descriptor, write to a `state` field, execute a CAS on a `state` field, or read the number of CASes in the CAS-list. □

Claim 3.8.13. *Before a* ClearBit *of a* CAS*-descriptor c is executed for the first time, c has been successfully executed.*

Proof. A ClearBit for a CAS-descriptor c can only be attempted (either in line 5,13) if the `state` of the c was previously read and turned out to be success. By Invariant 3.3, this means that c had been successfully executed before. □

Claim 3.8.14. *For every* CAS*-descriptor* c*:*
1) Any attempt to execute c *except the first attempt (by some thread) must fail.*
2) Any attempt to execute the ClearBit *of* c *except the first attempt (by some thread) must fail.*

Proof. (1) is simply restating the already proved Invariant 3.5. It remains to prove (2). Recall that an execution of a ClearBit is an execution of a CAS in which the `target address` is c's `target address`, the `expected-value` is c's `new-value` (including the version number) except that the `modified-bit` is on, and the `new-value` is the exact `new-value` of c. By Claim 3.8.13, when c's ClearBit is executed, c has already been successfully executed, and it follows that the version number stored in the `target address` is already at least equals to the version number of the `expected-value` of the ClearBit CAS. By Assumption 3.8.4, the version number is incremented in every successful CAS outside the executeCASes method. It follows that the version is incremented in any successful CAS excluding the ClearBit CAS, in which it remains the same. Thus, If the first execution of the ClearBit CAS fails, every further execution of it must fail as well, since the value stored in the `target address` can never hold the `expected-value` of the CAS. Similarly, if the first execution c's ClearBit is successful, then after it the `modified-bit` is off, and cannot be set on again without the version number being incremented. And thus, additional executions of c's ClearBit CAS must fail. □

Claim 3.8.15. *A* `modified-bit` *that belongs to a* CAS*-descriptor* c *can only be turned off by executing the* ClearBit *of* c*.*

By Assumption 3.8.3, a `modified-bit` cannot be turned off outside the executeCASes method since CASes outside the executeCASes method always expect the `modified-bit` to be off. Inside the executeCASes method, a `modified-bit` can only potentially be turned off when executing a ClearBit CAS. It remains to show that a `modified-bit` that belongs to a CAS-descriptor c cannot be turned off by executing a ClearBit of a different CAS-descriptor c'.

If any `modified-bit` belongs to c, it follows that c has been successfully executed. By Claim 3.8.13, to execute the ClearBit of c', c' must first also be successfully executed. In order for the ClearBit of c' to turn off a `modified-bit` that belongs to c, both c and c' must have the same `target address`, and, moreover, the same `new-value`, otherwise executing the ClearBit of c' would fail. However, if both c and c' have the same `new-value`, both must share the same version number in the `expected-value`, which implies that only one of them can possibly succeed. Thus, c' couldn't have been successfully executed, and thus it cannot clear the `modified-bit` of c.

Claim 3.8.16. *Before a* CAS*-descriptor* c *in a* CAS*-list* cl *is executed for the first time:*
1) All the previous CAS*-descriptors in* cl *have been* successfully *executed.*

2) ClearBit *has already been executed for all the previous* cas*-descriptors in cl. (Note: the claim vacuously holds for* cas*-descriptors that are never executed.)*

Proof. By Invariant 3.6, when c is executed, all the previous cas-descriptors in cl has their `state` set to success, which by Invariant 3.3 means they have all been successfully executed, proving (1). By Invariant 3.12, all `modified-bit`s of all the previous cas-descriptors have already been switched off, which by Claim 3.8.15 implies that the ClearBit of all the previous cas-descriptors in cl has already been executed, proving (2). □

Claim 3.8.17. *For any* cas*-descriptor c, the first attempt to execute the* ClearBit *of c (by some thread) is successful. (Note: the claim vacuously holds for* cas*-descriptors for which a* ClearBit *is never executed.)*

Proof. Immediately after executing c, the value stored in the `target address` is exactly the `expected-value` of the ClearBit cas. This value cannot be changed before a ClearBit cas is executed, since no cas except the ClearBit expects to find the `modified-bit` on, and there are no writes (without a cas) to the `target address` (Assumption 3.8.3). Thus, until a ClearBit is executed on this address, the value remains unchanged. By Claim 3.8.15, a ClearBit of a cas-descriptor other than c cannot be successful. Thus, the value in the `target address` remains the expected value of the ClearBit cas until the ClearBit is executed, and thus, the first attempt to execute the ClearBit of c is successful. □

Claim 3.8.18. *Once some thread has completed executing the* executeCASes *method on an input* cas*-list cl the following holds.*
1) Either all the cas*-descriptors have been successfully executed, or all the* cas*-descriptors have been executed until one that fails. Further* cas*-descriptors (after the first one that fails) have not been executed, and will also not be executed in the rest of the computation.*
2) A ClearBit *was successfully executed for each* cas*-descriptor that was successfully executed.*

Proof. By Claim 3.9, once some thread has completed the executeCASes method on the input cl, either the `state` field of all the cas-descriptors cl is set to success, or that one of them is set to failure, the ones previous to it to success, and the ones after it to pending. By Invariants 3.2 and 3.3, the cas-descriptors whose `state` is success were executed successfully, and the cas descriptor whose `state` is failure failed. By Invariant 3.7, cas-descriptors after the cas-descriptor that failed are not executed. Thus, (1) holds.

The thread that completed executing the executeCASes method on cl, has completed executing an iteration for each successful cas-descriptor in cl, and thus by Invariant 3.12, all the `modified-bit`s have already been switched off. By Claim

3.8.15, a `modified-bit` can only be turned off by a ClearBit of the CAS-descriptor that previously set the bit on, and thus, it follows that a ClearBit was successfully executed for each successful CAS-descriptor, and (2) holds. □

Claim 3.8.19. *The return value of the* executeCASes *for* every *thread that completes it is:*
1) The index of the first (and only) CAS*-descriptor whose execution failed the first time it was attempted, if such exists.*
2) -1 otherwise.

Proof. Each thread that executes the executeCASes method may exit it via one of three possible code-lines: 9, 17 or 20. If the thread exited via line 9, or via line 17, and returned i (the loop variable), then the `state` of the ith CAS-descriptor is failure, and thus its execution has failed by Invariant 3.2. By Claim 3.8.18 (1), this must be the only CAS that failed. Thus, in the case that a thread exits via line 9 or via line 17, the returned value is that of the first and only CAS-descriptor whose execution failed the first time it was attempted.

If a thread reaches line 20 and returns -1, then immediately before that it must be executing the last iteration of the loop in lines 2–19. Thus, by Invariant 3.6, the `states` of all the previous CAS-descriptors are success, and thus, by Invariant 3.3, all the CAS-descriptors before the last one were executed successfully. As to the last one, its `state` must be success as well (and thus, it must also have succeeded), otherwise when the thread reads the CAS-descriptors `state` and compares it to success in line 15, it would enter the if clause and leave through line 17. Thus, in the case that a thread exits reaches 20, all the CAS-descriptors were executed successfully, and -1 is returned.□

Lemma 3.8.20. *The implementation of the* executeCASes *method as given in Figure 3.5, is correct, meaning that it satisfies Definition 3.8.1.*

Proof. Follows from Observation 3.8.12, and Claims 3.8.14, 3.8.16, 3.8.18, and 3.8.19.□

3.8.2 Linearizability and WaitFreedom

Assume that LF is a linearizable lock-free algorithm given in the normalized form for a certain abstract data type, ADT. Let WF be the output algorithm of our transformation as described in Section 3.7 with LF being the simulated lock-free algorithm. Our goal is to prove that WF is a linearizable wait-free algorithm for the same abstract data type, ADT.

We claim that for every execution of WF, there is an *equivalent execution* (Definition 3.5.1) of LF. Since we know that LF is correct and linearizable, it immediately follows that WF is correct and linearizable as well. We start from a given execution of WF, denoted E_0, and we reach an equivalent execution of LF in several steps.

For each intermediate step, we are required to prove two key points. First, that the newly created execution preserves memory consistency. That is, each READ returns the last value written (or put via CAS) to the memory, and each CAS succeeds if and only if the value previously stored in the `target address` equals the `expected-value`. Proving memory consistency is required in order to prove that the newly created execution is indeed an execution.

Second, for each intermediate step, we are required to prove equivalency. That is, that each thread executes the same data structure operations in both executions, that the results are the same, and that the relative order of invocation and return points is unchanged. For the last execution in the series of equivalent executions, we will also prove that it is an execution of LF.

Step I: Removing Steps that Belong to the Additional Memory used by WF

WF uses additional memory than what is required by LF. Specifically, WF uses a `help queue`, in which it stores operation record boxes, which point to operation records. Operation records hold CAS-lists, which are in fact also used by LF, only that the CAS lists used by WF holds an extra `state` field for each CAS, not used in the original LF algorithm. In this step we erase all the computation steps (READS, WRITES, and CASES) on the additional memory used by WF.

Let E_1 be the execution resulting from removing from E_0 all the execution steps on the additional memory (the memory of the `help queue`, the operation record boxes, and the operation records excluding the CAS-lists - yet including the `state` field of each CAS in the CAS-lists).

Claim 3.8.21. *E_0 and E_1 are equivalent, and E_1 preserves memory consistency.*

Proof. E_1 has the same invocations and results of operations as E_0, and their relative order remain unchanged, thus E_0 and E_1 are equivalent by definition. E_1 preserves memory consistency since E_0 is memory consistent, and each memory register used in E_1 is used in E_1 in exactly the same way (same primitives with same operands, results, and order) as in E_0. □

Step II: Tweaking CASes of the executeCASes Method

Most of the steps of E_0 that belong to neither the GENERATOR, WRAPUP or CAS-EXECUTER method were dropped in E_1. However, in E_1 there are still two sources for steps that should be dropped. The main source is the executeCASes method (the other source will be reminded shortly). Recall that E_0 is an execution of WF, which employs both the CAS EXECUTER method (in the fast path) and the concurrent executeCASes method (in the slow path), while the original algorithm LF only employs the CAS EXECUTER method. By Lemma 3.8.20, all the computation steps of the

executeCASes method are either executing a cas-descriptor, executing a ClearBit of a cas-descriptor, or steps on the `state` field of a cas-descriptor in the cas list.

Steps on the `state` field were already dropped in the move from E_0 to E_1. Next, according to Lemma 3.8.20, each execution of a cas-descriptor that is not the first attempt to execute a given cas-descriptor, and each execution of a ClearBit that is not the first attempt to execute the ClearBit for the same cas-descriptor, must fail. It follows that these cases do not modify the memory and can be dropped without violating memory consistency. Afterwards, according to Lemma 3.8.20, what remains of the executeCASes are pairs of successful cases: each successful execution of a cas-descriptor is followed by a successful execution of a ClearBit cas of the same descriptor. Possibly, at the end of these successful pairs remains a single unsuccessful execution of a cas-descriptor.

We now tweak these pairs cases to be identical to an execution of the (fast path) cas-executer method. To do that, each pair is merged into a single cas. More precisely, the `new-value` of each execution of a cas-descriptor is changed such that the `modified-bit` is off (this alternative `new-value` is the same as the original `new-value` of the following ClearBit cas), and each ClearBit cas is dropped. After this change what remains of the executeCASes method is identical to the cas-executer method (except that the cases are executed by several thread instead of by a single thread, but this will be handled when moving from E_2 to E_3). However, the last change can potentially violate memory consistency.

Memory consistency is potentially violated for read primitives that were originally (that is, in E_0 and E_1) executed between an execution of a cas-descriptor to the following ClearBit cas. Memory consistency is violated because the value stored in the `target address` now has the `modified-bit` switched off immediately after the first execution of the cas, instead of being switched off only after the ClearBit cas. More importantly than read primitives, the memory consistency of cas primitives executed (in E_0 and E_1) between a cas-descriptor and the following ClearBit cas is also potentially violated.

To regain memory consistency, read primitives in between a pair are changed such that their returned value indicates that the `modified-bit` is unset. Recall that when we described the changes induced to the fast-path in our transformation, we mentioned that all read operations always disregard the `modified-bit` (the fast-path acts as if the bit were off). Thus, changing the execution such that now the bit is really off only takes us "closer" into an execution of LF.

cas primitives that occurred in between a pair of cases are handled as follows. Recall that in order to be compatible with the `modified-bit`, the fast path in WF is slightly altered. This is the second source of computation steps (the first being the ClearBit cases) that belong to WF and that do not originate from the three methods of the normalized structure. Whenever a cas is attempted and failed in the fast-path of WF, the same memory address is subsequently read. If the value is such that implies

that the CAS could have succeeded were the `modified-bit` switched off, then HELP is called, and then the CAS is retried. In what follows we simultaneously remove the extra READs and CASes originating from this modification of the fast-path and restore memory consistency.

For each CAS that failed in the fast-path, examine the corresponding READ following it. If the result of this READ indicates that the CAS should fail regardless of the `modified-bit`, then move the CAS forward in the execution to be at the place where the READ is, and drop the READ. If the results of the READ indicates that the CAS should succeed (or can succeed if the `modified-bit` would be switched off), then drop both the CAS and the READ. (The re-attempt of the CAS is guaranteed to be after the `modified-bit` is switched off.) We are now ready to formally define E_2.

Let E_2 be the execution resulted from applying the following changes to E_1.

- Each execution of a CAS-descriptor in the executeCASes method, excluding the first attempt for each CAS-descriptor, is dropped.
- Each execution of a ClearBit CAS is dropped.
- The remaining execution of CAS-descriptors in the executeCASes method are changed such that their `new-value` has the `modified-bit` off.
- For each unsuccessful CAS executed in the fast path:
  - If the CAS was re-attempted as a result of the subsequent corresponding READ, drop both the CAS and the READ, and keep only the re-attempt of the CAS (regardless whether this re-attempt succeeds or fails.)
  - Otherwise, move the CAS later in the execution to the place where the subsequent READ is, and drop the READ.
- (Remaining) READ primitives that were originally between a pair of a CAS-descriptor execution and the corresponding ClearBit execution, and that target the same memory address such as these CASes, are modified such that their returned value has the `modified-bit` switched off.

Claim 3.8.22. *E_2 and E_1 are equivalent, and E_2 preserves memory consistency.*

Proof. E_2 has the same invocations and results of operations as E_1, and their relative order remain unchanged, thus E_1 and E_2 are equivalent by definition. Dropping executions of CAS-descriptors that are not the first attempt of a given CAS-descriptor cannot violate memory consistency, because these CASes are unsuccessful by Lemma 3.8.20, and thus do not change the memory. Dropping the ClearBit CASes together with modifying the execution of the CAS-descriptors such that they set the `modified-bit` to off changes the state of the memory only for the time between each such pair of CASes, and thus can only violate memory consistency at these times. Consider the primitives that occur at these time frames.

By the definition of the normalized form, WRITE primitives are not used on these addresses. Furthermore, there could be no successful CASes between such a pair of CASes, because the `modified-bit` is on at these times, and the CLEARBIT CAS is the only CAS that ever has the `modified-bit` set in its `expected-value`. An Unsuccessful CAS receives special treatment. It is followed by a designated READ. If this READ determines the CAS can fail regardless of the `modified-bit`, then at the time of the READ, the CAS can fail without violating memory consistency in E_2 as well. Since in E_2 this CAS is moved in place of the READ (and the READ is dropped), then memory consistency is preserved for these CASes as well.

If the designated READ determines that the CAS may succeed, then the CAS is re-attempted. In such a case the CAS (together with the READ is dropped, and thus it does not violate memory consistency anymore. As for the re-attempt CAS, because it is only attempted after HELP is called, it is guaranteed to be executed after the CLEARBIT CAS. There are thus two options. Either the re-attempt CAS succeeds (both in E_1 and in E_2), and thus it is certainly not between a a pair of CASes, or the re-attempt CAS can fail. If it fails, then this cannot violate memory consistency. This is true even if the re-attempt CAS occurs between a (different) pair of CASes, because the fact that the CAS is re-attempted implies that its version number suits the previous pair of CASes, and cannot suit the new pair that is surrounding the re-attempt CAS.

As for other READ primitives between a pair of CASes(other than the designated READ that are specially inserted after a failure in a CAS), they are modified to return the value with the `modified-bit` off. Thus, memory consistency is restored for these READ primitives as well. □

Step III: Changing the Threads that Executed the Steps

In E_2 all the execution steps belong, or could legitimately belong, to one of the GENERATOR, WRAPUP, and cas executer methods. However, the threads that executes the steps are still mixed up differently than in LF. In this step the execution steps or their order are not altered, but the threads that execute them are switched. In E_3, the original threads of E_2 (which are the same as the threads of E_1 and of E_0) act accordingly to LF, and other additional threads (not present in E_2) are created to execute redundant runs of the GENERATOR and WRAPUP methods.

While a thread executes an operation in the fast path, without helping other operations, he follows the original LF algorithm. However, this changes when a thread moves to the slow path. First, a thread can move to the slow path because the contention failure counter of either the GENERATOR or WRAPUP methods causes it to stop. In such a case, the method has not been completed and will be executed again in the slow path. The execution steps originating from this uncompleted method are thus moved to an additional thread created for this purpose.

In the slow path, we examine all the executions of the GENERATOR and WRAPUP

methods. For each execution of such a method, we go back and examine what happens afterwards in E_0. If the thread that executes this method in E_0 later successfully CAS the operation record with the method's result to the operation record box (either in line 5 in the preCASes method (Figure 3.4) or in lines 6 or 8 in the postCASes method (Figure 3.6)), then the computation steps of this method are moved to the owner of the operation being helped (the thread that asked for help). Note that it is also possible that these steps belong to this owner thread in the first place, and are not moved at all.

If the thread that executes the method (either GENERATOR or WRAPUP) does not successfully CAS the result of the method into the operation record box, then the results of the method are simply discarded and never used. In this case, the computation steps of this method are moved to an additional thread created for this method only.

It remains to switch the owner of the CASes originating from the executeCASes method of the slow path. Some of them were dropped in the move from E_1 to E_2, and the rest were modified. We set the owner of the operation being helped (the thread that asked for help) to be the thread that executes these remaining CASes.

Let E_3 be the execution resulted from applying the following changes to E_2.

- For each GENERATOR method or WRAPUP method that is not completed due to contention (either in the fast path or in the slow path), create an additional thread, and let it execute the computation steps originating from this method.

- For each GENERATOR method or WRAPUP method executed in the slow path, whose results are not later successfully CASed into the operation record box, create an additional thread, and let it execute the computation steps originating from this method.

- For each GENERATOR method or WRAPUP method executed in the slow path, whose results *are* later successfully CASed into the operation record box, let the owner thread of the operation being helped execute the computation steps originating from this method.

- For each execution of the executeCASes method, let the owner of the operation being helped execute the CASes that originated from this method (if any remained in E_2).

Since E_3 includes additional threads that are not a part of E_2, we can only claim that E_3 and E_2 are equivalent when considering only the threads that participate in E_2. We formalize this limited equivalency as follows.

Definition 3.8.23. (Limited Equivalency of Executions.) For two executions E and E' we say that E limited to the threads of E' and E' are equivalent if the following holds.

- (Results:) The threads of E' execute the same data structure operations and receive identical results in both E' and E.

- (Relative Operation Order:) The order of invocation points and return points of all data structure operations is the same in both executions. In particular, this means that threads of E that do not participate in E' execute no data structure operations.
- (Comparable length:) either both executions are finite, or both executions are infinite.

Claim 3.8.24. *E_3 limited to the threads of E_2 and E_2 are equivalent, and E_3 preserves memory consistency.*

Proof. All the threads of E_2 have the same invocations and results of operations in E_3 that they have in E_2, and their relative order remains unchanged, thus E_3 and E_2 are equivalent by definition. By Claim 3.8.22, E_2 preserves memory consistency. E_3 only differs from E_2 in the threads that execute the primitive steps, but the steps themselves and their order remain unchanged, thus E_3 preserves memory consistency as well. □

Claim 3.8.25. *E_3 is an execution of LF, possibly with additional threads executing the* GENERATOR *and* WRAPUP *methods.*

Proof. By Claim 3.8.24, E_3 preserves memory consistency. It remains to show that each thread in E_3 either 1) follows the LF program structure of GENERATOR, CAS EXECUTER and WRAPUP methods, or 2) executes a single parallelizable method (either the GENERATOR or WRAPUP). To do this, we need to simultaneously consider executions E_3 and E_0. Note that each computation step in E_3 originates from a single computation step in E_0. (Some computation steps from E_0 were dropped and have no corresponding computation steps in E_3. Some computation steps in E_0 were slightly altered by changing the value of the `modified-bit`, and some were transferred to a different thread. Still, each computation step in E_3 originates from a single specific computation step in E_0.)

Fix an operation executed in E_3 and follow the thread that executes it. Originally, in E_0, the thread starts by offering help. However, all the computation steps that involve reading the help queue and operation records were already dropped in the move from E_0 to E_1; the remaining computation steps that involve helping the operation of a different thread are transferred either to the thread being helped or to an additional thread in the move from E_2 to E_3. Thus, in E_3 the thread starts executing the GENERATOR directly.

Originally, in E_0, while the execution is in the fast-path it is similar to LF with three small modifications. The first modification is that after executing a CAS that fails, the thread executes a READ on the `target address`, and then possibly re-executes the CAS. These extra steps were dropped in the transition from E_1 to E_2. The second modification is that the execution of the GENERATOR and WRAPUP methods is monitored, in the sense that a contention failure counter is updated and read periodically. However, there is no need for the contention failure counter to be in the shared memory. It is in a thread's local memory, and thus such monitoring occurs in the local steps and is not

reflected in the execution. It only affects the execution if the contention threshold is reached and help is asked. The third modification is that the number of times that the WRAPUP method indicates that the operation should be restarted from scratch is also monitored, in order to move to the slow-path if this number reaches a predetermined limit. Similarly to the contention failures counter, this monitoring is done within a threads's local computation.

Thus, as long as the execution of an operation in E_0 is in the fast-path (which could very well be until its completion), the corresponding execution in E_3 of the operation's owner thread is according to LF. Next, we examine what happens in E_0 when the thread asks for help and move to the slow-path. The method that was interrupted by the contention failure counter (if any) is transferred to an additional thread.

Once an operation in E_0 is in the slow path, the owner thread, and possibly helping threads, start executing one of three methods: the GENERATOR, EXECUTECASES, or WRAPUP, depending on the `state` of the operation record pointed by the operation record box. We examine how this execution is reflected in E_3.

For the GENERATOR and WRAPUP methods, the owner thread (the thread that asked for the help) executes in E_3 the steps of the thread that in E_0 successfully replaced the operation record with a CAS. These steps were transferred to the owner thread in the transition from E_2 to E_3. Other executions of the GENERATOR and WRAPUP methods, by threads that did not successfully replaced the operation record, are transferred to additional threads. Since only one thread may successfully CAS the operation record box from pointing to a given operation record to point to a new one, then in E_3 the owner thread executes the required parallelizable method (either GENERATOR or WRAPUP) once, as is done in an execution of LF. Afterwards, in E_0, helping threads will start executing the next required method (if any) according to the new `state` of the operation record.

The case is different for the EXECUTECASES method. Executions of the EXECUTECASES method are not transferred to additional threads, and the steps that are transferred to the owner in the transition from E_2 to E_3 were possibly executed by several different threads in E_0. To see that the steps that are executed in E_3 by the owner are indeed an execution of the CAS-EXECUTER method, we rely on Lemma 3.8.20. By this lemma, the first attempts of all the CAS-descriptors in the CAS-list are done according to their order, and once the first CAS-descriptor fails, the following CAS-descriptors in the list will not be attempted. In the transition from E_1 to E_2, only these first attempts of each CAS-descriptor in the list are kept, and further attempts are dropped. Also, the attempted CASes are changed and have the `modified-bit` of the `new-value` switched off. These modified CASes are transferred to the owner thread in the transition from E_2 to E_3.

Thus, in E_3, the owner thread executes the CASes of the list one by one according to their order, until one of them fails. This is simply an execution of the CAS-EXECUTER method. By Lemma 3.8.20, before the first thread exits the EXECUTECASES method,

all these CASes (all first attempts of CAS-descriptors) have already occurred. Thus, when in E_0 the operation's `state` is changed to post-CASes, and helping threads might start executing the WRAPUP method, all the computation steps of the EXECUTECASES (possibly apart from steps that are dropped in the transition from E_0 to E_1 or from E_1 to E_2) are already completed.

Regarding the output of the EXECUTECASES method, according to Lemma 3.8.20, the returned value of the EXECUTECASES method is the index of the first CAS that fails, or -1 if all CASes are executed successfully. In E_0, this value is stored inside the operation record and is used as the for the threads that read the operation record and execute the WRAPUP method. Thus, in E_0, and also in E_3, the WRAPUP method execution have the correct input.

We conclude that the execution of each operation in E_3 is according to LF. If in E_0 the operation is completed in the fast-path, then the operation owner executes the operation similarly in E_3, minus extra steps that were dropped, and steps that give help that are transferred either to additional threads or to the owner of the helped operation.

If an operation in E_0 starts in the fast-path and then moves to the slow-path, then the parallelizable methods (GENERATOR and WRAPUP) are transferred to the operation owner if their output was used, or to additional threads if the output was discarded. The execution of the EXECUTECASES is modified to an execution of CAS-EXECUTER and is transferred to the owner thread. Thus, E_3 is an execution of LF, possibly with extra threads, each of them executes once either the GENERATOR method, or the WRAPUP method. □

Step IV: Dropping Additional Threads

The purpose of this step is to drop all of the additional threads along with the parallelizable methods they are executing. Each additional thread executes a single parallelizable method. Each additional thread executes only a finite number of steps (because the method it executes is monitored in E_0 by a contention failure counter), and thus only a finite number of successful CASes. Thus, to drop an additional thread along with the parallelizable method it executes, we use the characteristic property of parallelizble methods, as given in Definition 3.5.4.

For each additional t executing a parallelizable method, we replace the execution with an equivalent execution in which all the threads follow the same program, but t's execution is avoidable. That is, t executes only futile and non-successful CASes. Such an execution, which is also an execution of LF plus additional threads executing parallelizable methods, exists by Definition 3.5.4. Then, t is simply dropped from the execution entirely. This does not violate memory consistency, because t's execution steps do not alter the data structure at all. This process is repeated for every additional thread.

Let E_4 be the execution resulted from the process describe above. Specifically, for

each additional thread t, we replace the execution with an equivalent execution in which t's executed method is avoidable, as is guaranteed by Definition 3.5.4, and then each additional thread is dropped.

Claim 3.8.26. *E_3 limited to the threads of E_4 and E_3 are equivalent, and E_4 preserves memory consistency.*

Proof. For each additional thread, the transition to an equivalent execution as guaranteed by Definition 3.5.4 preserves equivalence and memory consistency. An additional thread that only executes READs, failed CASes, and futile CASes can be dropped without harming memory consistency (as it does not alter the shared memory). □

Claim 3.8.27. *E_2 and E_4 are equivalent.*

Proof. E_2 and E_4 has the same set of threads: threads that are added in the transition from E_2 to E_3 are dropped in the transition from E_3 to E_4. Both E_2 and E_4 are equivalent to E_3 limited to their threads (Claims 3.8.24 and 3.8.26). It follows that E_2 and E_4 are equivalent. □

Claim 3.8.28. *E_4 is an execution of LF.*

Proof. By Claim 3.8.25, E_3 is an execution of LF with possibly additional threads executing parallelizable methods. The equivalent execution guaranteed in Definition 3.5.4 is such in which each thread follows the same program. Thus, each (non-additional) thread follows the same program in E_3 and in E_4, which means that each thread in E_4 follows an execution of LF. All the additional threads of E_3 are dropped, and thus E_4 is an execution of LF. □

Linearizability of WF

Corollary 3.14. *For each execution of WF, there exists an equivalent execution of LF.*

Proof. Follows directly from Claims 3.8.21, 3.8.22, 3.8.27, and 3.8.28. □

Theorem 3.15. *WF is a linearizable.*

Proof. It is given that LF is linearizable. For each execution of WF there exists an equivalent execution of LF (Corollary 3.14). Thus, each execution of WF is linearizable, and WF itself is linearizable. □

Wait Freedom of WF

To show that WF is wait-free, we first claim that it is lock-free. Then, we show that due to the helping mechanism, WF cannot be lock-free without being wait-free as well.

Claim 3.8.29. *WF is lock-free.*

Proof. Assume by way of contradiction that WF is not lock-free. Thus, there exists an infinite execution of WF in which only a finite number of operations are completed. By Corollary 3.14, for each execution of WF exists an equivalent execution of LF. By definition of equivalent executions, the equivalent execution of LF must also be infinite, and only a finite number of operations may be completed in it. This contradicts the fact that LF is lock-free. □

Theorem 3.16. *WF is wait-free.*

Proof. Assume by way of contradiction that WF is not wait-free. Thus, there exists an infinite execution of WF, in which some thread executes infinitely many steps yet completes only a finite number of operations. Let E be such an execution, and T the thread that completes only a finite number of operations. Consider the last operation that T starts (which it never completes).

T cannot execute infinitely many steps in the fast-path: executions of the GENERATOR and WRAPUP methods are monitored by a contention failures counter, and at some point in an infinite execution of them the threshold must be reach, and help will be asked. Thus, it is impossible to execute infinitely many steps in a single method of the fast-path. However, it is also impossible to execute infinitely many loops of the GENERATOR, CAS-EXECUTER and WRAPUP methods, since when a certain threshold is reached, help is asked. Thus, at some point, T must ask for help.

When asking for help, T enqueues a help request into the wait-free `help queue`. Since this queue is wait-free, then after a finite number of steps the help request must be successfully enqueued into the queue, with only a finite number of help requests enqueued before it.

While the `help queue` is not empty, each thread, when starting a new operation, will first help the operation at the head of the `help queue` until it is completed and removed from the help queue. Only then, the thread will go and execute its own operation. It follows that once a help request for an operation *op* is enqueued to the `help queue`, each thread can only complete a finite number of operations before *op* is completed. To be accurate, if at a given moment *op* is the n'th operation in the queue, then each thread can complete a maximum of n operations before *op* is completed.

Thus, once T successfully enqueues the help request into the `help queue`, only a finite number of operations can be completed before T completes its operation. Since T never completes its operation, then only a finite number of operations can be completed at all. Thus, in the infinite execution E, only a finite number of operations is completed. This contradicts the fact that WF is lock-free (Claim 3.8.29). □

3.9 On the Generality of the Normalized Form

Our simulation can automatically transform any lock-free linearizable data structure given in a normalized form into a wait-free one. A natural question that arises is how

general the normalized form is. Do all abstract data types (ADT) have a normalized lock-free implementation? We answer this question in the affirmative. However, the value of this general result is theoretical only as we do not obtain *efficient* normalized lock-free implementations. The main interest in the transformation described in this chapter is that it attempts to preserve the efficiency of the given lock-free data structure. Thus, it is not very interesting to invoke it on an inefficient lock-free implementation.

We claim that any ADT can be implemented by a normalized lock-free algorithm (given that it can be implemented sequentially). This claim is shown by using (a simplified version of) the universal construction of Herlihy [Her90], which transforms any sequential data structure to a linearizable lock-free one. Recall that in this universal construction, there is a global pointer to the shared data structure. To execute an operation, a thread reads this pointer, creates a local copy of the data structure, executes the operation on the local copy, and attempts by a CAS to make the global pointer point to its local copy. If the CAS succeeds the operation is completed, and if it fails, the operation is restarted from scratch. We observe that this construction is in effect already in the normalized form, it just needs to be partitioned correctly into the three methods.

Specifically, the CAS-GENERATOR method creates the local copy of the data structure, executes the operation on it, and outputs a list with a single CAS descriptor. The CAS defined in the CAS-descriptor is the attempt to make the global pointer point to the local copy that was prepared in the CAS-generator. The CAS-executer method is the fixed method of the normalized representation, which simply attempts this CAS and (since it is the only one) reports the result. The WRAP-UP method then indicates a restart from scratch if the CAS failed, or returns with the appropriate results if it succeeded.

Of course, this construction is not practical. A lock-free data structure built in this manner is likely to be (very) inefficient. But this construction shows that each ADT can be implemented using the normalized form.

3.10 Examples: the Transformation of Four Known Algorithms

In this section we will present how we converted four known lock-free data structures into wait-free ones, using the described technique. The four data structures are: Harris's linked-list, Fomitchev & Ruppert's linked-list, a skiplist, and a binary-search-tree. During this section we will also explain how to wisely construct the *parallelizable* GENERATOR and WRAP-UP methods, in a manner which is easy to implement, efficient, and strait-forward.

3.10.1 Harris's linked-list

Harris designed a practical lock-free linked-list. His list is a sorted list of nodes in which each node holds an integer key, and only one node with a given key may be in the list at any given moment. He employed a special `mark bit` in the `next` pointer of every node, used to mark the node as logically deleted. Thus, a node is deleted by first marking its next pointer using a CAS (in effect, locking this pointer from ever changing again) and then physically removing it from the list by a CAS of its predecessor's `next` field. Inserting a new node can be done using a single CAS, making the new node's designated predecessor point to the new node. In this section we assume familiarity with Harris's linked-list. A reader not familiar with it may skip this section and read on.

We start by noting that Harris's SEARCH method, which is used by both the INSERT and DELETE operations, is a *parallelizable method*. The SEARCH method's input is an integer key, and its output is a pair of adjacent nodes in the list, the first with a key smaller than the input value, and the second with a key greater than or equal to the input value. The SEARCH method might make changes to the list: it might physically remove marked nodes, those nodes that are logically deleted. The search method is restarted in practice anytime an attempted CAS fails. (Such an attempted CAS is always an auxiliary CAS, attempting to physically remove a logically deleted node.) A simple enough *contention failure counter* for this method can be implemented by counting number of failed CASes.

We now specify a normalized version of Harris's linked-list:

- A *contention failure counter* for all of the methods in Harris's linked-list can be implemented by counting the number of failed CASes.
- The (parallelizable) GENERATOR method is implemented as follows: For an insert(key) operation:
  - Call the original SEARCH(KEY) method.
  - If a node is found with the wanted key, return an empty list of CAS-descriptors. (The insert fails.)
  - If a pair (pred, succ) is returned by the search method, create a new node n with the key, set n.next = succ, and return a list with a single CAS descriptor, describing a change of pred.next to point to n.

  The GENERATOR method for a delete(key) operation is:
  - Call the original SEARCH(KEY) method.
  - If no node is found with the given key, return an empty list of CAS-descriptors.
  - If a node n was found appropriate for deletion, return a list with a single CAS-descriptor, describing a change of n.next to set its `mark-bit`.

  The GENERATOR method for a contains(key) operation is:

- return an empty list of of CAS-descriptors.

- The (parallelizable) WRAP-UP method is implemented as follows: For an insert(key) or a delete(key) operation:

  - If the list of CAS-descriptors is empty, exit with result false (operation failed).
  - If the CAS-descriptor was executed successfully, exit with result true (operation succeeded).
  - If the CAS-descriptor was not successful, indicate that a restart of the operation is required.

  For a contains(key) operation:

  - Call the original contains(key) method (which is already a parallelizable method) and exit with the same result.

We would like to make a remark concerning the contention failure counter. Implementing a counter that simply counts the number of CAS failures is good enough for a linked-list of integers (like the one Harris and others have implemented), but is insufficient for a linked-list of strings, and other data types as well. This is because infinitely many insertions before the key searched for by a CONTAINS method or a SEARCH method, can delay a thread forever without it ever failing a CAS operation. In such cases a more evolved contention failure counter is needed. Its implementation requires holding an approximation counter on the number of keys in the list. Holding the exact count is possible, but inefficient, whereas maintaining an approximation with a bounded error can be achieved with a negligible time overhead and is enough. The more evolved contention failure counter reads the approximation at the beginning of each method and its value is #failed CASes + Max(0, traversed keys - (approximation + max error)). The full details for implementing this contention failure counter along with the needed approximation appear in Appendix E.

3.10.2 Binary Search Tree

The first practical lock-free binary search tree was presented in [EFRvB10]. The algorithm implements a leaf-oriented tree, meaning that all the keys are stored in the leaves of the tree, and each internal node points to exactly two children. When a thread attempts to insert or delete a node, it begins its operation by a CAS on an internal node's *state* field. It stores a pointer to an *Info* object, describing the desired change. This (owner) CAS effectively locks this node, but it can be unblocked by any other thread making the desired (*auxiliary*) CASes. In [EFRvB10], storing the initial pointer to the *Info* object is also referred to as *Flagging*, and we shall use this notation as well. In a DELETE operation, they also use *Marking*, that permanently locks the internal node that is about to be removed from the tree. Familiarity with [EFRvB10] is required to

fully understand this part. In a nutshell, an INSERT is separated into three CASes:

- I-1. Flagging the internal node that its child sub-tree is needed to be replaced.
- I-2. Replacing the child pointer to point to the new sub-tree
- I-3. Unflagging the parent.

A DELETE operation is separated into four CASes:

- D-1. Flagging the grandfather of the leaf node we wish to delete.
- D-2. Marking the parent of the node we wish to delete(this parent will be removed from the tree as well, but the child is the only leaf node that is to be removed).
- D-3. Changing the grandfather's child pointer to point to a new sub-tree.
- D-4. Unflagging the grandparent.

The neat design of this algorithm makes it very easy to convert it into the normalized structure and thus into a wait-free algorithm, since the methods in it are separated by their functionality. It contains a SEARCH method, designed to find a key or its designated location. This method does not change the data structure, and is thus trivially a *parallelizable method*.
It contains additional parallelizable methods designed to help intended operations already indicated by Info fields: The HELP, HELP-DELETE, HELP-MARKED and HELP-INSERT methods.

In this algorithm, the linearization points of the operations happens **after** the blocking (owner) CASes, inside the *parallelizable methods*, thus the normalized version would have to do some work after the CAS-EXECUTOR method is completed. This is naturally done in the WRAP-UP method.

- A contention failure counter implementation consists of the following.
  - Count the number of times CASes failed.
  - Count the number of times parallelizable methods were called (except the first time for each method).
- The GENERATOR, For an insert(key) operation:
  - Call the original SEARCH(KEY) method.
  - If a node with the requested key is found, return an empty list of CASes.
  - If the parent is *Flagged*: call the (original) HELP method, and afterwards restart the Generator.

  - Return a list with a single CAS-descriptor containing a CAS to change the state of the designated parent to point to an *Info* object describing the insertion (CAS-I-1).

- The WRAP-UP method for an insert(key) operation:
  - If the list of CASes is empty, exit with result false (operation failed).
  - If CAS-I-1 failed, return *restart operation from scratch.*
  - Else, call (the original parallelizable method) HELPINSERT (which will perform CAS-I-2 and CAS-I-3) and exit with true (operation succeeded).

- The GENERATOR method, for a delete(key) operation:
  - Call the original SEARCH(KEY) method.
  - If a node with the requested key was not found, return an empty list of CASes.
  - If the grandparent is *Flagged*: call the (original) HELP method, and afterwards restart the GENERATOR method.
  - If the parent is *Flagged*: call the (original) HELP method, and afterwards restart the GENERATOR method.
  - Return a list with a single CAS-descriptor, containing a CAS to change the state of the grandparent to point to an *Info* object describing the deletion (CAS-D-1).

- The WRAP-UP method, for a delete(key) operation:
  - If the list of CASes is empty, exit with result false (operation failed).
  - If CAS-D-1 failed, return *restart operation from scratch.*
  - Else, call the (original) HELPDELETE method (which potentially executes CAS-D-2, CAS-D-3, and CAS-D-4, but may fail).
    * if HELPDELETE returned true, return *operation succeeded.*
    * else, return *restart operation from scratch.*

- The GENERATOR method, for a contains(key) operation:
  - Return an empty list of CASes.

- The WRAP-UP method, for a contains(key) operation:
  - call the original SEARCH(KEY) method.
  - If a node with the requested key was found, exit with result true.
  - Else, exit with result false.

Note that the binary-search-tree algorithm is designed in a way that during a single operation, each *parallelizable* method can only be called more than once as a result of contention (since other thread had to make a change to the tree that affects the same node). Additionally, the remark about Harris's linked-list (the additional effort needed in some cases in order to implement a contention failure counter) applies here as well.

3.10.3 Skiplist

Let us refer to the lock-free skiplist that appears on [HS08]. It is composed of several layers of the lock-free linked-list of Harris. Each node has an array of `next` fields, each point to the next node of a different level in the skiplist. Each `next` field can be *marked*, signifying the node is logically deleted from the corresponding level of the skiplist. The keys logically in the list are defined to be those found on *unmarked* nodes of the lowest list's level. To delete a key, first the FIND(KEY) method is called. If a corresponding node is found, its `next` fields are marked by a CAS from its top level down to level zero. To insert a key, again, the FIND(KEY) method is called first, returning the designated predecessor and successor for each level. The node is inserted to the lowest (zero) level first, and then to the rest of the levels from bottom up. Familiarity with chapter 14.4 of [HS08] is required to fully understand the process.

When designing this algorithm, a subtle design decision was made that carries interesting implications for our purposes. As the algorithm appears in [HS08], the only auxiliary CASes are snipping out marked nodes in the FIND method, similar to Harris's linked-list. Fully linking a node up after it has been inserted to the lowest level is done only by the thread that inserted the node. Thus, in order to achieve lock-freedom, operations by other threads must be allowed to complete while some nodes are incomplete (not fully linked). These operations might include inserting a node immediately after an incomplete node, or even deleting an incomplete node. Allowing such operations to complete causes some difficulties. One result is that when two nodes are being inserted concurrently, and they are intended to be adjacent nodes at some level of the skiplist, it is possible that the node that should come first will bypass the link to its designated successor, skipping over it, and even past other nodes entered concurrently to the same level. This cannot happen at the bottom level, and so it does not hamper the algorithm's correctness, but it can cause higher levels to hold less nodes than they were supposed to, arguably foiling the log(n) complexity of the skiplist.

It is a small and relatively simple change to make the linking up of an inserted node to be done by auxiliary CASes, which are attempted by each thread that traverse that node in the FIND method, instead of doing it by owner CASes only attempted by the thread that inserts the node. If we would make this change, these CASes could be done by other threads in their GENERATOR method. As it is, however, they can only be done in the WRAP-UP method, and only by the owner thread. Since our purpose here is to show how our technique should be used to convert a **given** lock-free algorithm into a

wait-free one, and not to suggest variants to the lock-free algorithm, we shall focus on showing how to normalize the algorithm of [HS08] this way.

- A contention failure counter for each method can be implemented by counting the number of failed CASes.
- The GENERATOR for an insert(key) operation:
  - Call the original FIND(KEY) method.
  - If a node is found with the desired key, return an empty list of CASes.
  - Else, create a new node n with the key, set its `next` field in each level to point to the designated successor, and return a list with a single CAS-desciptor, to change the prev.next at the bottom level to point to n.
- The WRAP-UP method for an insert(key) operation:
  - If the CAS-list is empty, return false (operation failed).
  - If the CAS in the CAS-list failed, return *restart operation from scratch.*
  - Else, follow the original algorithm's linking up scheme. That is, until the new node is fully linked:
    - Call FIND(KEY).
    - Try by a CAS to set the predecessor's next field to point to the newly inserted node for each unlinked level. Use the successor returned from the FIND method as the expected value for the CAS. Restart the loop if the CAS fails.
- The GENERATOR method for a delete(key) operation:
  - Call the original FIND(KEY) method.
  - If no node is found with the given key, return an empty CAS-list.
  - If a node n was found appropriate for deletion, return a list with a CAS-descriptor for each level in which the node is linked, from the highest down to level zero, to mark its `next` field.
- The WRAP-UP method for a delete(key) operation is as follows.
  - If the CAS-list is empty, return false (operation failed).
  - Else, if all CASes were successful, return true (operation succeeded).
  - Else, return *restart operation from scratch.*
- The GENERATOR method for a contains(key) operation:
  - Return an empty list of CASes.

- The WRAP-UP method for a contains(key) operation is as follows.
  - Call the original FIND(KEY) method.
  - If a node with the requested key was found, exit with result true.
  - Else, exit with result false.

The remark about Harris's linked-list (the additional effort needed in some cases in order to implement a contention failure counter) applies here as well.

3.10.4 The Linked-List of Fomitchev and Ruppert

In the list of Fomitchev and Ruppert, before deleting a node, a backlink is written into it, pointing to its (last) predecessor. This backlink is later used to avoid searching the entire list from the beginning the way Harris did when a node he used was deleted. Fomitchev and Ruppert employ two special bits in each node's `next` field. The `mark` bit, similarly to Harris's algorithm, to mark a node as logically deleted, and the `flag` bit, that is used to signal that a thread wants to delete the node pointed by the flagged pointer. Deletion is done in four phases:

- Flagging the predecessor
- Writing the backlink on the victim node to point to the predecessor
- Marking the victim node
- physically disconnecting the node and unflagging the predecessor (both done in a single CAS).

The main (owner) CAS in this case, which must be done in the CAS-EXECUTER method, is the first (flagging the predecessor). This flagging blocks any further changes to the predecessor until the flag is removed. Removing the flag can be done by any thread in the parallelizable HELPFLAGGED method. The second phase, of writing the backlink, is actually not done by a CAS, but by a direct WRITE. This is safe, since the algorithm is designed in a way that guarantees that for a specific node, there is only a single value that will be written to it (even if many threads will write it). Keeping this non-CAS modification of the data structure will not harm our transformation and it will still provide a correct wait-free algorithm, yet it does not technically match our definition of the normalized representation. To solve this, we can replace this WRITE action with a CAS that uses $NULL$ as the expected-value. This change have no algorithmic applications. The insert operation is done similarly to the insert operation in Harris's linked-list, except that it uses the backlinks to avoid searching the list from the beginning, and that it calls the HELPFLAGGED method to remove the "lock" on a flagged node, if needed.

- A contention failure counter implementation consists of the following.

  - Count the number of times CASes failed.
  - Count the number of times the HELPFLAGGED method is called (except the first time).

- The GENERATOR, for an insert(key) operation:
  - Call the original search(key) method.
  - If a node is found with the wanted key, return an empty list of CAS-descriptors.
  - Else, if a window(pred, succ) is returned, and pred is flagged, call the (original) HELPFLAGGED method.
  - If a window (pred, succ) that is fit for inserting the key is found, create a new node n with the key, set n.next = succ, and return a list with a single CAS-descriptor, describing a change of pred.next to point to n.

- The WRAP-UP method for an insert(key) operation:
  - If the list of CAS-descriptors is empty, exit with result false (operation failed).
  - If the CAS-descriptor was executed successfully, exit with result true (operation succeeded).
  - If the CAS-descriptor was not successful, indicate *restart operation from scratch.*

- The GENERATOR, for a delete(key) operation:
  - Call the original search(key) method.
  - If no node is found with the given key, return an empty list of CAS-descriptors.
  - If a victim node and its predecessor were found, return a list with a single CAS-descriptor, describing a change of the predecessor.next so that its `flag-bit` will be set.

- The WRAP-UP method for a delete(key) operation:
  - If the list of CAS-descriptors is empty, exit with result false (operation failed).
  - If the CAS-descriptor was executed successfully, call the (original) HELPFLAGGED method, and afterwards exit with result true (operation succeeded).
  - If the CAS-descriptor was not successful, indicate *restart operation from scratch.*

- The GENERATOR method for a contains(key) operation:
  - Return an empty list of CASes.

- The WRAP-UP method for a contains(key) operation:

– Call the original SEARCH(KEY) method.

– If a node with the requested key was found, exit with result true.

– Else, exit with result false.

As with all the examples, the remark appearing after Harris's linked list applies here as well. In the following section, we describe an important optimization that is especially important in the case of the transformation of the list of Fomitchev & Ruppert; the normalized representation of the algorithm does not fully utilize the strength of the backlinks, which is a key feature of this algorithm when comparing it to Harris's. Using the optimization in 3.11.1 guarantees that most operations will still fully utilize the backlinks, while the few operations that will complete in the slow path may extract only part of its benefits.

3.11 Optimizations

3.11.1 Using the Original Algorithm for the Fast Path

In order to use our simulation technique and obtain a wait-free practical algorithm, the first thing we need to do is to express the lock-free data structure in the normalized form. As mentioned above, in our work we expressed four data structures this way. Our intuition is that the data structure in the normalized form is in some way "the same" as the original algorithm, only expressed differently. In what follows, we provide some formalization for this intuition and then use it for an optimization.

Definition 3.11.1. (Interoperable Data Structures.) We say that two lock-free data structure algorithms are *interoperable* if they can be run on the same memory concurrently and maintain linearizability and correctness.

The above definition means that for each data-structure operation that we would like to perform, we can arbitrarily choose which of the two algorithms to use for running it, and the entire execution remains linearizable for the same ADT. All of the four normalized algorithms we created are *interoperable* with their original versions[6]. We would like to exploit this fact in order to use the original lock-free algorithm, and not the normalized version of it, as the fast-path for the simulation. The slow path, in which help is given, still works in the normalized manner. This optimization is possible, but requires some care. To safely allow the original algorithm to work with the help mechanism of the normalized algorithm, we require that a slightly stronger parallelism property will be kept by the parallelizable methods. Recall that a *parallelizable method* is a one whose executions are avoidable. In what follows we strengthen the definition of avoidable method execution.

[6]Excluding the fact that version numbers must be added to the original algorithms as well.

Definition 3.11.2. Strongly avoidable method execution: A run of a method M by a thread T on input I in an execution E of a program P is strongly avoidable if there exists an equivalent execution E' for E such that in both E and E' each thread follows the same program, both E and E' are identical until right before the invocation of M by T on input I, in E' each CAS that T executes in M either fails or is futile, and (the new requirement): In E and E' the shared memory reaches the same states in the same order.

A state of the shared memory is simply the contents of all memory. Failed CASes, futile CASes, and READ primitives, do not alter the state of the shared memory. The new requirement does not mean that after n computation steps the state of the shared memory is the same in E and in E', since each one of them can have a different set of computation steps that do not alter the memory. The meaning of the extra requirement is that the alternative execution E' is not only equivalent to E, but is also indistinguishable from it, in the sense that an observer who examines the shared memory cannot tell whether E or E' has taken place.

This stronger definition is not needed for our technique to work, only to ensure a safe use of this specific optimization. All of the four algorithms we expressed in the normalized form naturally fulfill this stronger requirement. Thus, since the original algorithm can work interoparably with the normalized one, it can also work interoparably with the normalized one in the presence of "extra" avoidable executions of parallelizable methods, and we can safely use it as the fast-path, given that we adjust it to have contention failure counters for its methods.

3.11.2 Avoiding versions

As explained in Section 3.7.2, while executing the CASes, a helping thread may create an ABA problem if it is delayed and then returns to execute when the CAS it is attempting to simulate has already been completed and the algorithm has moved on. To ensure that this helping thread does not foil the execution, we introduced versioning to make sure its CAS fails and it can continue executing properly. For some data structures, ABA problems of this type cannot occur because the original data structure is designed to avoid them. For example, the tree algorithm of Ellen et al. [EFRvB10] allows helping threads to operate within the original lock-free algorithm and it supports such help with a special mechanism that eliminates such ABA problems. Therefore, for the tree there is no need to add the versioning mechanism to each CAS, and indeed we did not use versioning when making the tree wait-free. This does not eliminate the need to use the `modified-bit` for a structured execution of the public CASes.

3.12 Performance

3.12.1 Memory Management

In this work we do not specifically address the standard problem of memory management for lock-free (and wait-free) algorithms. In the Java implementation we just use Java's garbage collector, which is probably not wait-free. If the original lock-free algorithm has a solution for memory management, then the obtained simulation works well with it, except that we need to reclaim objects used by the generated algorithm: the operation records and the operation record boxes. This can be done using hazard pointers [Mic04]. The implementation is tedious, but does not introduce a significant difficulty and we do not deal with it in the currently.

3.12.2 Our Wait-Free Versions vs. the Original Lock-Free Structures

We chose four well-known lock-free algorithms, and used the transformation described in this chapter to derive a wait-free algorithm for each. We implemented these algorithms and, when possible, used the optimizations described in Section 3.11. The performance of each wait-free algorithm was compared against the original lock-free algorithm. We stress that we compared against the original lock-free version of the algorithm without adding versioning to the CAS operations and without modifying it to fit a normalized representation.

The four lock-free algorithms we chose were Harris's linked-list [Har01], the binary-search-tree of Ellen et al. [EFRvB10], the skiplist of Herlihy and Shavit [HS08], and the linked-list of Fomitchev and Ruppert [FR04]. All implementations were coded in Java. The Java implementations for the lock-free algorithms of Harris's linked-list and the skiplist were taken from [HS08]. We implemented the binary search tree and the list of Fomitchev and Ruppert ourselves, in the most straightforward manner, following the papers.

All the tests were run on SUN's Java SE Runtime, version 1.6.0. We ran the measurements on 2 systems. The first is an IBM x3400 system featuring 2 Intel(R) Xeon(R) E5310 1.60GHz quad core processors (overall 8 cores) with a memory of 16GB and an L2 cache of 4MB per processor. The second system features 4 AMD Opteron(TM) 6272 2.1GHz processors, each with 8 cores (overall 32 cores), each running 2 hyper-threads (overall 64 concurrent threads), with a memory of 128GB and an L2 cache of 2MB per processor.

We used a micro-benchmark in which 50% of the operations are *contains*, 25% are *insert*, and 25% are *delete*. Each test was run with the number of threads ranging from 1 to 16 in the IBM, and 1 to 32 in the AMD. In one set of tests the keys were randomly and uniformly chosen in the range $[1, 1024]$, and in a different set of tests the keys were chosen in the range $[1, 64]$. In each test, each thread executed 100,000 operations overall. We repeated each test 15 times, and performance averages are reported in the figures.

The maximum standard deviation is less than 5%. The contention threshold was set to $k = 2$. In practice, this means that if one of the three simulation stages encounters k failed CASes, it gives up the fast path and moves to the slow path.

Figure 3.7 compares the four algorithms when running on the AMD (the left graph of each couple) and on the IBM (right) for 1024 possible keys. The figure show the execution times (seconds) as a function of the number of threads.

For 1024 keys, the performance of the wait-free algorithms is comparable to the lock-free algorithms, the difference being 2% on average. The close similarity of the performance between the original lock-free algorithms and the wait-free versions produced using our simulation suggests that the slow-path is rarely invoked.

Figure 3.8 indicates how many times the slow path was actually invoked in each of the wait-free data structures as a function of the number of threads. Keep in mind that the overall number of operations in each run is 100,000 multiplied by the number of threads. The results reported are again the averages of the 15 runs (rounded to whole numbers). As expected, the fraction of operations that require the slow path is very small (maximum fraction of about 1/3,000 of the operations). The vast majority of the operations complete in the fast-path, allowing the algorithm to retain performance similar to the lock-free algorithm. Yet, a minority of the operations require the help mechanism to guarantee completion in a bounded number of steps, thus achieving wait-freedom.

The results for 64 keys are depicted in figures 3.9 and 3.10. The behavior for 64 keys is different than for 1024 keys. The smaller range causes a lot more contention, which in turn causes a lot more operations to ask for help and move to the slow-path. Asking for help in the slow path too frequently can dramatically harm the performance. This is most vividly displayed in the tree data structure on the AMD. When running 32 parallel threads, about 1 in 64 operations asks for help and completes in the slow-path. This means that roughly during half of the execution time there is an operation running in the slow-path. As a result, all threads help this operation, sacrificing scalability for this time. Thus, it is not surprising that the performance are down by about 50%.

In such circumstances, it is advisable to set the contention threshold to a higher level. Setting it to 3 (instead of 2) causes a significant improvement in the performance. This comes with the cost of allowing some operations to take longer, as some operations will first fail 3 times, and only then ask for help.

3.12.3 Our Wait-Free Transformation vs. a Universal Construction

Universal constructions achieve a difficult task, as they go all the way from a sequential data structure to a concurrent wait-free implementation of it. It may therefore be difficult to also make the resulting wait-free algorithm efficient enough to become practicable. Our technique builds on a tailored made lock-free data structure and achieve the smaller step from lock-freedom to wait-freedom. This may be the reason why we are able to

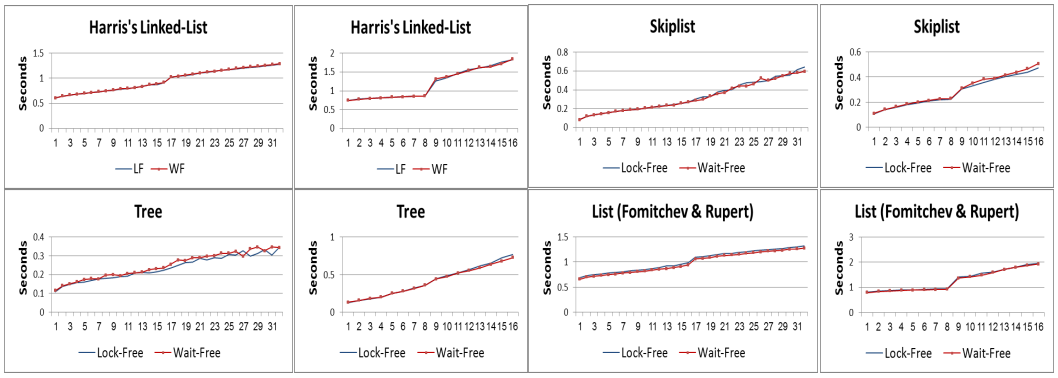

Figure 3.7: Lock-Free versus Wait-Free algorithms, 1024 keys. Left: AMD. Right: IBM

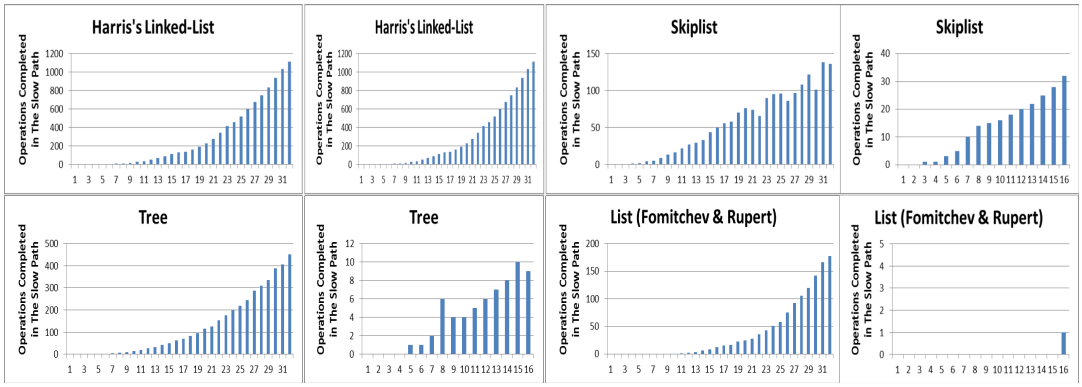

Figure 3.8: Number of Operation Completed in the Slow Path., 1024 keys. Left: AMD. Right: IBM

Figure 3.9: Lock-Free versus Wait-Free algorithms, 64 keys. Left: AMD. Right: IBM

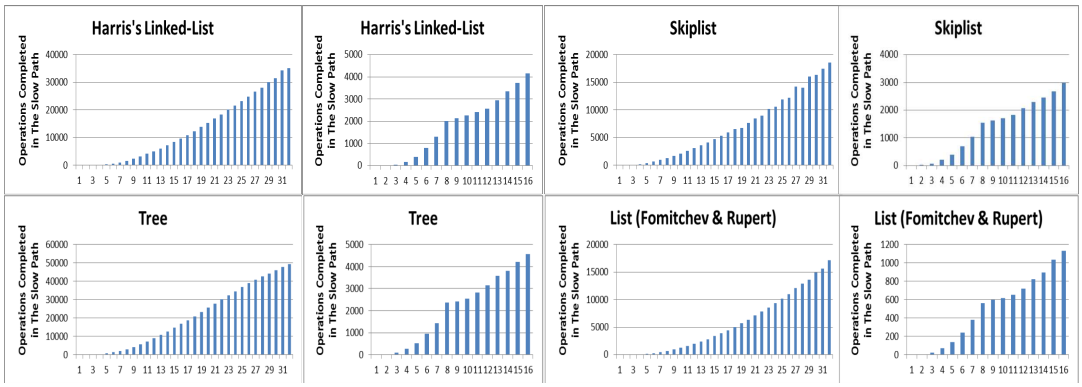

Figure 3.10: Number of Operation Completed in the Slow Path, 64 keys. Left: AMD. Right: IBM

retain practicable performance.

To demonstrate the performance difference, we implemented the state of the art universal construction of Chuong, Ellen, and Ramachandran [CER10] for a standard sequential algorithm of a linked-list. The obtained wait-free linked-list was compared against the wait-free linked-list generated by applying our technique to Harris's lock-free linked-list. [7]

We ran the two implementations on our AMD Opetron system featureing 4 AMD Opteron(TM) 6272 2.1GHz processors, each with 8 cores (overall 32 cores), each running 2 hyper-threads (overall 64 concurrent threads), with a memory of 128GB and an L2 cache of 2MB per processor. In the micro-benchmark tested, each thread executed 50% contains, 25% insert, and 25% delete operations. The keys were randomly and uniformly chosen from the range $[1, 1024]$. The number of threads was ranging from 1 to 32. In each measurement, all the participating threads were run concurrently for 2 seconds, and we measured the overall number of operations executed. Each test was run 10 times, and the average scores are reported in the figures.

In Figure 3.11 the total number of operations (in millions) done by all the threads is reported as a function of the number of the threads. It can be seen that the wait-free list obtained in this chapter (and so also the lock-free linked-list) drastically outperforms the universal construction for any number of threads. Also, while our list scales well all the way up to 32 threads, the list of the universal construction does not scale at all. Figure 3.12 is based on the same data, but demonstrates the ratio between our construction of the wait-free linked-list and the universal construction of wait-free linked list. For a single thread, our list is 6.8 times faster and this ratio grows with any additional thread,

[7] Note that implementing the universal construction of [CER10] on Harris's lock-free linked-list, instead of using the universal construction on a standard sequential list, is possible, but ill-advised. Although both implementations would result in a wait-free list, the one based on a lock-free algorithm would undoubtedly be slower. The universal construction already handles the inter-thread race conditions, and implementing it on Harris's linked-list would force it to also use the (unneeded) synchronization mechanisms of Harris.

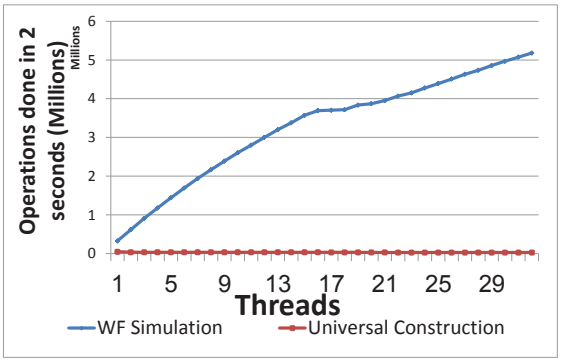

Figure 3.11: Our Wait-Free List against a Universal Construction List

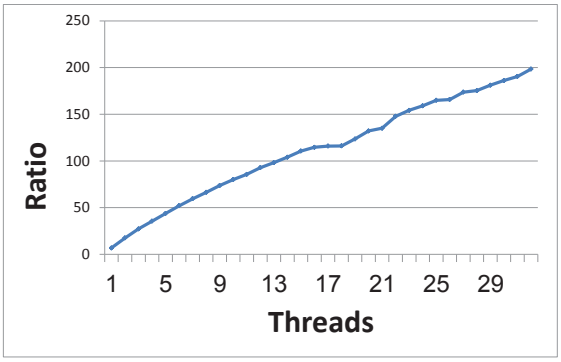

Figure 3.12: Ratio between Our List and a Universal Construction List

up to a factor of 198 times faster than the universal construction for 32 threads.

Chapter 4

Help!

4.1 Introduction

As discussed in Section 1.2, a helping mechanism, such as the one employed in Chapters 2 and 3, is a common technique used to obtain wait-freedom. Curiously, despite its abundant use, to date, helping has not been formally defined nor was its necessity rigorously studied.

In this chapter we offer a rigorous study of the interaction between wait-freedom and helping. We start with presenting a formal definition of help, capturing the intuition of one thread helping another to make progress. Next, we present families of object types for which help is necessary in order to obtain wait-freedom. In other words, we prove that for some types there are no linearizable wait-free help-free implementations. In contrast, we show that other, simple types, can be implemented in a linearizable wait-free manner without employing help. Finally, we provide a universal strong primitive for obtaining wait-free with no help. Specifically, given a wait-free help-free fetch&cons object, one can implement any type in a wait-free help-free manner.

Naturally, the characterization of types which require help depends on the primitives being used, and while our results are generally stated for READ, WRITE, and CAS, we discuss additional primitives as well. In particular, we show that exact order types (Definition 4.5.1) cannot be both help-free and wait-free even if the FETCH&ADD primitive is available, but the same statement is not true for global view types (Definition 4.6.18). Finally, we show that a fetch&cons primitive is universal for wait-free help-free objects. This means that given a wait-free help-free fetch&cons object, one can implement any type in a wait-free help-free manner.

This chapter is organized as follows. Section 4.2 discusses additional related work to the work discussed in Section 1.2. Model and definitions for this chapter are given in Section 4.3. The concept of help is formally defined in Section 4.4. In Section 4.5 we define exact order types (examples are the queue and the stack), and prove that they cannot be implemented in a wait-free help-free manner. Section 4.6 defines global view types (such as a snapshot object) and proves a similar claim for them. In Section

4.7, we prove that if the CAS primitive is not available, max-registers also cannot be implemented in a help-free wait-free manner. Section 4.8 discusses some types that can be both help-free and wait-free. Section 4.9 proves that a fetch&cons primitive is universal, in the sense that given such a primitive, every type can be implemented in a help-free wait-free manner. We end this chapter with a short discussion in Section 4.10.

4.2 Additional Related Work

Helping mechanisms come in different forms. Many wait-free implementations use a designated `announcement` array, with a slot for each process. Each process uses its slot to describe the operation it is currently seeking to execute, and other processes read this announcement and help complete the operation. This is perhaps the most widely used helping mechanism, appearing in specific designs, as well as in universal constructions [Her88], and also in the general technique presented in Chapter 3.

But other forms of help exist. Consider, for example, the form of help that is used for the double-collect snapshot algorithm of [AAD$^+$93]. In this wait-free snapshot object, each UPDATE operation starts by performing an embedded SCAN and adding it to the updated location. A SCAN operation op_1 that checks the object twice and sees no change can safely return this view. If a change has been observed, then the UPDATE operation op_2 that caused it also writes the view of its embedded SCAN, allowing op_1 to adopt this view and return it, despite the object being, perhaps constantly, changed. Thus, intuitively, the UPDATES help the SCANS.

4.3 Model and Definitions

We consider a standard shared memory setting with a fixed set of processes P. In each computation step, a process executes a single atomic primitive on a shared memory register, possibly preceded with some local computation. The set of atomic primitives contains READ, WRITE primitives, and usually also CAS. Where specifically mentioned, it is extended with the FETCH&ADD primitive.

A CAS primitive is defined by a triplet, consisting of a target register, an expected-value, and a new-value. When a CAS step is executed, the value stored in the target register is compared to the expected-value. If they are equal, the value in the target register is replaced with the new-value, and the Boolean value true is returned. In such a case we say that the CAS is *successful*. Otherwise, the shared memory remains unchanged, and false is returned. We stress that a CAS is executed atomically.

A FETCH&ADD primitive is defined by a target register and an integer value. An execution of the FETCH&ADD primitive atomically returns the value previously stored in the target register and replaces it with the sum of the previous value and the FETCH&ADD's integer value.

A *type* (e.g., a FIFO queue) is defined by a state machine, and is accessed via *operations*. An operation receives zero or more input parameters, and returns one result, which may be null. The state machine of a type is a function that maps a state and an operation (including input parameters) to a new state and a result of the operation.

An *object*, is an implementation of a type using atomic primitives. An implementation specifies the primitives and local computation to be executed for each operation. The local computation can influence the next chosen primitive step. When the last primitive step of an operation is finished, the operation's result is computed locally and the operation is completed.

In the current work, we consider only executions of objects. Thus, a *program* of a process consists of operations on an object that the process should execute. The program may include local computations, and results of previous operations may affect the chosen future operations and their input parameters. A program can be finite (consisting of a finite number of operations) or infinite. This may also depend on the results of operations.

A *history* is a log of an execution (or a part of an execution) of a program. It consists of a finite or infinite sequence of computation steps. Each computation step is coupled with the specific operation that is being executed by the process that executed the step. The first step of an operation is also coupled with the input parameters of the operation, and the last step of an operation is also associated with the operation's result. A single computation step is also considered a history (of length one).

A *schedule* is a finite or infinite sequence of process ids. Given a schedule, an object, and a program for each process in P, a unique matching history corresponds. For a given history, a unique schedule corresponds. Given two histories, h_1, h_2, we denote by $h_1 \circ h_2$ the history derived from the concatenation of history h_2 after h_1. Given a program *prog* for each process in P, and a history h, for each $p \in P$ we denote by $h \circ p$ the history derived from scheduling process p to take another single step following its program immediately after h.

The *set of histories created by an object* O is the set that consists of every history h created by an execution of any fixed set of processes P and any corresponding programs on object O, in any schedule S.

A history defines a partial order on the operations it includes. An operation op_1 is before an operation op_2 if op_1 is completed before op_2 begins. A sequential history is a history in which this order is a total order. A linearization [HW90] L of a history h is a sequence of operations (including their input parameters and results) such that 1) L includes all the operations that are completed in h, and may include operations that are started but are not completed in h, 2) the operations in L have the same input parameters as the operations in h, and also the same output results for operations that are completed in h, 3) for every two operations op_1 and op_2, if op_1 is completed before op_2 has begun in h, and op_2 is included in L, then op_1 is before op_2 in L, and 4) L is consistent with the type definition of the object creating history h. An object O is a

linearizable implementation of type T if each history in the set of histories created by O has a linearization.

Lock-freedom and wait-freedom are forms of progress guarantees. In the context of our work, they apply to objects (which are, as mentioned above, specific implementations of types). An object O is lock-free if there is no history h in the set of histories created by O such that 1) h is infinite and 2) only a finite number of operations is completed in h. That is, an object is lock-free if at least one of the executing processes must make progress and complete its operation in a finite number of steps. Wait-freedom is a strictly stronger progress guarantee. An object O is wait-free if there is no history h in the set of histories created by O such that 1) h includes an infinite number of steps by some process p and 2) the same process p completes only a finite number of operations in h. That is, O is wait-free if every process that is scheduled to run infinite computation steps must eventually complete its operation, regardless of the scheduling.

4.4 What is Help?

The conceptual contribution of this work is in establishing that many types cannot be implemented in a linearizable wait-free manner without employing a helping mechanism. To establish such a conclusion, it is necessary to accurately define help. In this section we discuss help intuitively, define it formally, and consider examples showing that the formal definition expresses the intuitive concept of help. Additionally, we will establish two general facts about help-free wait-free implementations.

4.4.1 Intuitive Discussion

Many wait-free algorithms employ an array with a designated entry for each process. A process announces in this array what operation it wishes to execute, and other processes that see this announcement might help and execute this operation for it. Such mechanisms are used in most wait-free universal constructions, dating back to [Her88] and many other constructions since. These mechanisms are probably the most explicit way to offer help, but not the only one possible. Considering help in a more general form, we find it helpful[1] to think of the following scenario.

Consider a system of three processes, p_1, p_2, p_3, and an object that implements a FIFO queue. The program of p_1 is ENQUEUE(1), the program of p_2 is ENQUEUE(2), and the program of p_3 is DEQUEUE(). First consider a schedule in which p_3 starts running solo until completing its operation. The result of the dequeue, regardless of the implementation of the FIFO queue, is null. If before scheduling p_3, we schedule p_1 and let it complete its operation, and only then let p_3 run and complete its own operation, p_3 will return 1. If we schedule p_1 to start executing its operation, and stop it at some point (possibly before its completion) and then run p_3 solo until completing its operation,

[1] Pun intended.

it may return either null or 1. Hence, if we consider the execution of p_1 running solo, there is (at least) one computation step S in it, such that if we stop p_1 immediately before S and run p_3 solo, then p_3 returns null, and if we stop p_1 immediately after S and run p_3 solo, p_3 returns 1.

Similarly, if we consider the execution of p_2 running solo, there is (at least) one computation step that "flips" the value returned by p_3 when running solo from null to 2. We now consider a schedule that interleaves p_1 and p_2 until one of them completes. In any such execution, there is (at least) one computation step that "flips" the result of p_3 from null to either 1 or 2. If a step taken by p_2 "flips" the result of p_3 and causes it to return 1 (which is the value enqueued by p_1) we say that p_2 helped p_1. Similarly, if a step taken by p_1 "flips" the result of p_3 and causes it to return 2, then p_1 helped p_2.

This is the intuition behind the help notion that is defined below. Some known lock-free queue algorithms do not employ help, such as the lock-free queue of Michael and Scott [MS96]. However, we prove in Section 4.5 that any *wait-free* queue algorithm must employ help.

4.4.2 Help Definition

We say that an operation *op* belongs to history h if h contains at least one computation step of *op*. Note that *op* is a specific instance of an operation on an object, which has exactly one invocation, and one result. We say that the *owner* of *op* is the process that executes *op*.

Definition 4.4.1. (Linearization Function.) We say that f is a linearization function over a set of histories H, if for every $h \in H$, $f(h)$ is a linearization of h.

Definition 4.4.2. (Decided Operations Order.) For a history h in a set of histories H, a linearization function f over H, and two operations op_1 and op_2, we say that op_1 is *decided before* op_2 in h with respect to f and the set of histories H, if there exists no $s \in H$ such that h is a prefix of s and $op_2 \prec op_1$ in $f(s)$.

Definition 4.4.3. (Help-Free Implementation.) A set of histories H is without help, or help-free, if there exists a linearization function f over H such that for every $h \in H$, every two operations op_1, op_2, and a single computation step γ such that $h \circ \gamma \in H$ it holds that if op_1 is decided before op_2 in $h \circ \gamma$ and op_1 is not decided before op_2 in h, then computation step γ is a step in the execution of op_1 by the owner of op_1.

An object is a help-free implementation, if the set of histories created by it is help-free.

To better understand this definition, consider an execution of an object. When considering two concurrent operations, the linearization of these operations dictates which operation comes first. The definition considers the specific step, γ, in which it is

decided which operation comes first. In a help-free implementation, γ is always taken by the process whose operation is decided to be the one that comes first.

Consider the wait-free universal construction of Herlihy [Her88]. One of the phases in this construction is a wait-free reduction from a *fetch-and-cons* list to consensus. A fetch-and-cons (or a fetch-and-cons list) is a type that supports a single operation, *fetch-and-cons*, which receives a single input parameter, and outputs an ordered list of the parameters of all the previous invocations of *fetch-and-cons*. That is, conceptually, the state of a fetch-and-cons type is a list. A *fetch-and-cons* operation returns the current list, and adds (hereafter, cons) its input operation to the head of the list.

The reduction from fetch-and-cons to consensus is as follows. A special announce array, with a slot for each process, is used to store the input parameter of each ongoing *fetch-and-cons* operation. Thus, when a process desires to execute a *fetch-and-cons* operation, it first writes its input value to its slot in the announce array.

Next, the process reads the entire announce array. Using this information, it calculates a *goal* that consists of all the operations recently announced in the array. The process will attempt to cons *all* of these operations into the fetch-and-cons list. It reads the current state of the fetch-and-cons list, and appends this list to the end of its own goal (removing duplications.) Afterwards, the process starts executing (at most) n instances of consensus (n is the number of processes). In each instance of consensus, a process proposes its own process id.

The goal of the process that wins the consensus represents the updated state of the fetch-and-cons list. Thus, if the process wins a consensus instance, it returns immediately (as its own operation has definitely been applied). If it loses a consensus, it updates its goal again to be his original goal (minus duplications that already appear it the updated state) followed by the new list, which is the goal of the last winner. After participating in n instances of consensus, the process can safely return, since at least one of the winners in these instances already sees the process's operation in the announces array, and includes it in its goal.

This is a classic example of help. Wait-freedom is obtained due to the fact that the effect of process p winning an instance is adding to the list all the items it saw in the announce array, not merely its own item. To see that this algorithm is not help-free according to Definition 4.4.3 consider a system of four processes[2]. Each process first announces its wanted item in the ANNOUNCE array, and then reads all of the array. Assume p_1's place in the array is before p_2's, but that p_2 writes to the announce array first. p_3 then reads the announce array and sees p_2's item. Then p_1 writes to the announce array, and then p_4 reads the entire announce array.

At this point p_1 and p_2 are stalled, while p_3 and p_4 start competing in consensus. If the winner is p_3, then the item of p_2 is added to the list, but the item of p_1 not as yet. If p_4 wins the consensus, then it adds p_1's item before p_2's item. Thus, there exists an

[2]A tighter analysis considering only three processes is possible.

execution, in which the question of which of the fetch-and-cons operations of p_1 and p_2 comes first is decided while p_1 and p_2 are stalled. This contradicts help-freedom.

4.4.3 General Observations

In this subsection we point out two facts regarding the decided operations order (Definition 4.4.2) that are useful to prove that some types cannot be both wait-free and help-free. The first fact is true for non help-free implementations as well, as it is derived directly from the linearizability criteria. It states that for completed operations, the decided order must comply with the partial order a history defines, and for future operations, the decided order cannot contradict partial orders that may apply later on.

Observation 4.4.4. In any history h:
(1) Once an operation is completed it must be decided before all operations that have not yet started.
(2) While an operation has not yet started it cannot be decided before any operation of a different process.
(3) In particular, the order between two operations of two different processes cannot be decided as long as none of these operations have started.

The second fact is an application of the first observation for help-free implementations.

Claim 4.4.5. *In a help-free implementation in a system that includes at least three processes, for a given history h and a linearizarion function f, if an operation op_1 of a process p_1 is decided before an operation op_2 of a process p_2, then op_1 must be decided before any future (not yet started) operation of any process.*

Proof. Immediately following h, allow p_2 to run solo long enough to complete the execution of op_2. By Observation 4.4.4, op_2 must now be decided before any future operation. Thus, by transitivity, op_1 must be decided before any future operation as well. In a help-free implementation, op_1 cannot be decided before a different operation as a result of a step of p_2. Thus, op_1 must be decided before future operations already at h. □

4.5 Exact Order Types

In this section we prove that some types cannot be implemented in a linearizable, wait-free, and help-free manner. Simply put: for some types, wait-freedom requires help. We first prove this result for systems that support only READ, WRITE, and CAS primitives. We later extend the proof to hold for systems that support the FETCH&ADD primitive as well. This section focuses on exact order types. Roughly speaking, these are types in which switching the order between two operations changes the results of

future operations. An intuitive example for such a type is the FIFO queue. The exact location in which an item is enqueued is important, and will change the results of future dequeues operations.

In what follows we formally define exact order types. This definition uses the concept of a sequence of operations. If S is a sequence of operations, we denote by $S(n)$ the first n operations in S, and by S_n the n-th operation in S. We denote by $(S + op?)$ a sequence that contains S and possibly also the operation op. That is, $(S + op?)$ is in fact a set of sequences that contains S, and also sequences that are similar to S, except that a single operation op is inserted in somewhere between (or before or after) the operations of S.

Definition 4.5.1. (Exact Order Types.) An exact order type t is a type for which there exists an operation op, an infinite sequence of operations W, and a (finite or an infinite) sequence of operations R, such that for every integer $n \geq 0$ there exists an integer $m \geq 1$, such that for at least one operation in $R(m)$, the result it returns in any execution in $W(n+1) \circ (R(m) + op?)$ differs from the result it returns in any execution in $W(n) \circ op \circ (R(m) + W_{n+1}?)$.

Examples of such types are a queue, a stack, and the fetch-and-cons used in [Her88]. To gain some intuition about the definition, consider the queue. Let op be an ENQUEUE(1) operation, W be an infinite sequence of ENQUEUE(2) operations, and R be an infinite sequence of DEQUEUE operations. The queue is an exact order type, because the $(n+1)$-st dequeue returns a different result in any execution that starts with $n+1$ ENQUEUE(2) operations compared to in any execution that starts with n ENQUEUE(2) operations and then an ENQUEUE(1).

More formally, let n be an integer, and set m to be $n+1$. Executions in $W(n+1) \circ (R(m) + op?)$ start with $n+1$ ENQUEUE(2) operations, followed by $n+1$ DEQUEUE operations. (There is possibly an ENQUEUE(1) somewhere between the dequeues, but not before any of the ENQUEUE(2).) Executions in $W(n) \circ op \circ (R(m) + W_{n+1}?)$ start with n ENQUEUE(2) operations, then an ENQUEUE(1) operation, and then $n+1$ DEQUEUE operations. (Again, there is possibly an ENQUEUE(2) somewhere between the dequeues.) From the specification of the FIFO queue, the last DEQUEUE must return a different result in the first case (in which it must return 2) than in the second case (in which it must return 1).

We now turn to prove that any exact order type cannot be both help-free and wait-free. Let Q be a linearizable, help-free implementation of an exact order type. The reader may find it helpful to consider a FIFO queue as a concrete example throughout the proof. We will prove that Q is not wait-free. For convenience, we assume Q is lock-free, as otherwise, it is not wait-free and we are done. Let op_1, W, and R be the operation and sequences of operations, respectively, guaranteed in the definition of exact order types. Consider a system of three processes, p_1, p_2, and p_3. The program of process p_1 is the operation op_1. The program of process p_2 is the infinite sequence W.

The program of process p_3 is the (finite or infinite) sequence R. The operation of p_1 is op_1, an operation of p_2 is denoted op_2, and the first operation of p_3 is denoted op_3.

We start by proving two claims that are true to any execution of Q in which p_1, p_2, and p_3 follow their respective programs. These claims are the only ones that directly involve the definition of exact order types. The rest of the proof considers a specific execution, and builds on these two claims.

Claim 4.5.2. *Let h be a history such that the first n operations are already decided to be the first n operations of p_2 (which are $W(n)$), and p_3 has not yet taken any step. (Denote the $(n+1)$-st operation of p_2 by op_2.)*
(1.) If in h op_1 is decided before op_3, then the order between op_1 and op_2 is already decided.
(2.) Similarly, if in h op_2 is decided before op_3, then the order between op_1 and op_2 is already decided.

Proof. For convenience, we prove (1). The proof of (2) is symmetrical. Assume that in h op_1 is decided before op_3, and let m be the integer corresponding to n by the definition of exact order types. Immediately after h, let p_3 run in a solo execution until it completes exactly m operations. Denote the history after this solo execution of p_3 by h', and consider the linearization of h'.

The first n operations in the linearization must be $W(n)$. The linearization must also include exactly m operations of p_3 (which are $R(m)$), and somewhere before them, it must also include op_1. The linearization may or may not include op_2. There are two cases. If the $(n+1)$-st operation in the linearization is op_1, then the linearization is in $W(n) \circ op_1 \circ (R(m) + W_{n+1}?)$, while if the $n+1$-st operation in the linearization is op_2, then the linearization must be exactly $W(n+1) \circ op_1 \circ R(m)$ which is in $W(n+1) \circ (R(m) + op_1?)$. We claim that which ever is the case, the order between op_1 and op_2 is already decided in h'.

To see this, consider any continuation $h' \circ x$ of h'. Consider the linearization of $h' \circ x$. This linearization must also start with $W(n)$, must also include $R(m)$, and somewhere before $R(m)$ it must include op_1. It may or may not include op_2 somewhere before R_m. All the rest of the operations must be linearized after R_m, because they where not yet started when R_m was completed. Thus, the prefix of the linearization of $h' \circ x$ (and of any other continuation of h' as well) must belong to either $W(n) \circ op_1 \circ (R(m) + W_{n+1}?)$ or to $W(n+1) \circ (R(m) + op_1?)$.

In h', the operations $R(m)$ are already completed, and their results are set. By definition of exact order types, these results cannot be consistent with both $W(n) \circ op_1 \circ (R(m) + W_{n+1}?)$ and $W(n+1) \circ (R(m) + op_1?)$. Thus, if the linearization of h' is in $W(n) \circ op_1 \circ (R(m) + W_{n+1}?)$, then the results of $R(m)$ mean that the prefix of the linearization of any continuation of h' cannot be in $W(n+1) \circ (R(m) + op_1?)$, and thus must also belong to $W(n) \circ op_1 \circ (R(m) + W_{n+1}?)$. Similarly, if the linearization of h' is

in $W(n+1) \circ (R(m) + op_1?)$, then the prefix of the linearization of any continuation of h' must be in $W(n+1) \circ (R(m) + op_1?)$ as well.

Thus, in h', the $(n+1)$-st operation is already decided, meaning that the order between op_1 and op_2 is already decided. Since Q is a help-free implementation, then the order between op_1 and op_2 cannot be decided during the solo execution of p_3 which is the delta between h and h'. It follows that the order between op_1 and op_2 is already decided in h. □

Claim 4.5.3. *Let h, h', and h'' be three histories, such that in all of them the first n operations are already decided to be the first n operations of p_2 (which are $W(n)$), and p_3 has not yet taken any step. (Denote the $(n+1)$-st operation of p_2 by op_2.) Further more, in h the order between op_1 and the op_2 is not yet decided, in h' op_1 is decided before op_2, and in h'' op_2 is decided before op_1.*
(1.) h' and h'' are distinguishable by p_3.
(2.) h and h' are distinguishable by at least one of p_2 and p_3.
(3.) h and h'' are distinguishable by at least one of p_1 and p_3.

Remark. (3.) is not needed in the proof, but is stated for completeness.

Proof. Let m be the integer corresponding to n by the definition of exact order types. We start by proving (1). Assume that immediately after h' p_3 is run in a solo execution until it completes exactly m operations. The linearization of this execution must start with $W(n)$, followed by op_1. This linearization must also include the first m operations of p_3 (which are $R(m)$, and it may or may not include op_2. Thus, the linearization must be in $W(n) \circ op_1 \circ (R(m) + W_{n+1}?)$.

Now assume that immediately after h'' p_3 is run in a solo execution until it completes exactly m operations. This time, the linearization must be in $W(n+1) \circ (R(m) + op_1?)$. By the definition of exact order types, there is at least one operation in $R(m)$, that is, at least one operation of p_3, which returns a different result in these two executions. Thus, h' and h'' are *distinguishable* by process p_3.

We turn to prove (2). Assume that immediately after h' p_2 is run until it completes op_2, and then p_3 is run in a solo execution until it completes exactly m operations. The linearization of this execution must be exactly $W(n) \circ op_1 \circ W_{n+1} \circ R(m)$ which is in $W(n) \circ op_1 \circ (R(m) + W_{n+1}?)$.

Now assume that immediately after h p_2 is run until it completes op_2 and then p_3 is run in a solo execution until it completes exactly m operations. At the point in time exactly after op_2 is completed, and exactly before p_3 starts executing op_3, op_2 is decided before op_3 (Observation 4.4.4). Thus, by Claim 4.5.2, the order between op_1 and op_2 is already decided. Since the order is not decided in h, the implementation is wait-free, and p_1 has not taken another step since h, it follows that op_2 must be decided before op_1.

```
1:  h = ϵ;
2:  op1 = the single operation of p1;
3:  while (true)                                                    ▷ main loop
4:    op2 = the first uncompleted operation of p2;
5:    while (true)                                                  ▷ inner loop
6:      if op1 is not decided before op2 in h ∘ p1
7:        h = h ∘ p1;
8:        continue;                                                 ▷ goto line 5
9:      if op2 is not decided before op1 in h ∘ p2
10:       h = h ∘ p2;
11:       continue;                                                 ▷ goto line 5
12:     break;                                                      ▷ goto line 13
13:   h = h ∘ p2;                                                   ▷ this step will be proved to be a CAS
14:   h = h ∘ p1;                                                   ▷ this step will be proved to be a failed CAS
15:   while (op2 is not completed in h)                             ▷ run p2 until op2 is completed
16:     h = h ∘ p2;
```

Figure 4.1: The algorithm for constructing the history in the proof of Theorem 4.3.

In other words, in the execution in which after h p_2 completes op_2 and then p_3 completes exactly m operations, op_2, which is W_{n+1}, is decided before both op_3 and op_1. Thus, the linearization of this execution must be in $W(n+1) \circ (R(m) + op_1?)$.

By the definition of exact order types, there is at least one operation in $R(m)$, that is, at least one operation of p_3, which returns a different result in these two executions. Thus, h and h' are *distinguishable* by at least one of the processes p_2 and p_3. The proof of (3) is similar. □

In the rest of the proof of the main theorem we build an infinite history h, such that the processes p_1, p_2, and p_3 follow their respective programs, and p_1 executes infinitely many (failed) CAS steps, yet never completes its operation, contradicting wait-freedom. The algorithm for constructing this history is depicted in Figure 4.1. In lines 5–12, p_1 and p_2 are scheduled to run their programs as long as it is not yet decided which of their operations comes first. Afterwards, the execution of Q is in a critical point. If p_1 were to take a step, then op_1 will be decided before op_2, and if p_2 were to take a step, then op_2 will be decided before op_1. We prove using indistinguishability arguments, that the next step by both p_1 and p_2 is a CAS. Next (line 13), p_2 executes its CAS, and then (line 14) p_1 attempts a CAS as well, which is going to fail. Afterwards, p_2 is scheduled to complete its operation, and then the above is repeated with p_2's next operation.

It is shown that in iteration $n+1$ of the algorithm for constructing h, the n first operations are already decided to be the first n operations of p_2 (that is, $W(n)$), and iteration $n+1$ is a "competition" between op_1 and W_{n+1}. The key feature of exact order types, is that once the $(n+1)$-st operation is decided, it must be distinguishable by p_3, because a long enough solo execution of p_3 returns different results if the $(n+1)$-st operation is op_1 or if it is W_{n+1}. Let us formalize this claim.

We prove a series of claims on the execution of history h, which is a history of

object Q. Most claims refer to the state of the execution of Q in specific points in the execution, described by a corresponding line in the algorithm given in Figure 4.1. These claims are proved by induction, where the induction variable is the iteration number of the main loop (lines 3–16). The induction hypothesis is that claims (4.5.4–4.5.14) are correct. Claim 4.5.4 is the only one to use the induction hypothesis directly. The other claims follow from Claim 4.5.4.

Claim 4.5.4. *Immediately after line 4, it holds that 1) the order between op_1 and op_2 is not yet decided, and 2) all the operations of p_2 prior to op_2 are decided before op_1.*

Proof. For the first iteration of the main loop, this is trivial because h is empty (Observation 4.4.4). For iteration $i \geq 2$, it follows from the induction hypothesis, Observation 4.5.11, and Claim 4.5.14. □

Observation 4.5.5. The order between op_1 and op_2 cannot be decided during the inner loop (lines 5–12).

This follows from the fact that Q is help-free, and from inspecting the conditions in lines 6 and 9.

Observation 4.5.6. Process p_3 never takes a step in h.

Claim 4.5.7. *The order between op_1 and op_2 must be decided before any one of op_1 and op_2 is completed.*

Proof. If op_1 is completed, then op_1 must be decided before all future operations of p_3 (Observation 4.4.4). All the operations of p_2 prior to op_2 are already decided before op_1 (Claim 4.5.4), and by Observation 4.5.6, p_3 hasn't taken any steps. Thus, by Claim 4.5.2, the order between op_1 and op_2 is already decided.

Similarly, if op_2 is completed, then op_2 must be decided before all future operations of p_3 (Observation 4.4.4). Again, all the operations of p_2 prior to op_2 are already decided before op_1 (Claim 4.5.4), and by Observation 4.5.6, p_3 hasn't taken any steps. Thus, by Claim 4.5.2, the order between op_1 and op_2 is already decided. □

Claim 4.5.8. *The execution of the inner loop (lines 5–12) is finite.*

Proof. By combining Observation 4.5.5 and Claim 4.5.7, no operation in Q is completed in h during the execution of the inner loop. Since Q is lock-free, and each loop iteration adds a single step to h, this cannot last infinitely. □

Observation 4.5.9. Immediately before line 13 op_1 is decided before op_2 in $h \circ p_1$, op_2 is decided before op_1 in $h \circ p_2$, and, hence, the order of op_1 and op_2 is not decided in h.

From observing the code, the inner loop exits and line 13 is reached only if the next step of either p_1 or p_2 will decide the order. Since the queue algorithm is help-free, in $h \circ p_1$, op_1 is decided before op_2, and in $h \circ p_2$, op_2 is decided before op_1.

Claim 4.5.10. *Immediately before line 13 the following holds.*
(1.) The next primitive step in the programs of both p_1 and p_2 is to the same memory location.
(2.) The next primitive step in the programs of both p_1 and p_2 is a CAS.
(3.) The expected-value of both the CAS *operations of p_1 and p_2 is the value that appears in the designated address.*
(4.) The new-value of both the CAS *operations is different than the expected-value.*

Proof. By Observation 4.5.9, in $h \circ p_1$, op_1 is decided before op_2. It follows that op_1 is decided before op_2 in $h \circ p_1 \circ p_2$ as well. Similarly, op_2 is decided before op_1 in $h \circ p_2 \circ p_1$. By Claim 4.5.3 (1), it follows that $h \circ p_1 \circ p_2$ must be *distinguishable* from $h \circ p_2 \circ p_1$ by process p_3. It immediately follows that the next primitive step of both p_1 and p_2 is to the same memory location. Furthermore, the next step of both p_1 and p_2 cannot be a READ primitive. Also, it cannot be a CAS that does not change the shared memory, i.e., a CAS in which the expected-value is different than the value in the target address, or a CAS in which the expected-value and new-value are the same.

Thus, the next step by both p_1 and p_2 is either a WRITE primitive or a CAS which satisfies conditions (3) and (4) of the claim. It remains to show the next step is not a WRITE. Assume by way of contradiction the next step by p_1 is a WRITE. Then, $h \circ p_1$ is indistinguishable from $h \circ p_2 \circ p_1$ to all process excluding p_2, again contradicting Claim 4.5.3 (1). A similar argument also shows that the next step of p_2 cannot be a WRITE.□

Claim 4.5.10 immediately implies:

Corollary 4.1. *The primitive step p_2 takes in line 13 is a successful* CAS*, and the primitive step p_1 takes in line 14 is a failed* CAS.

Observation 4.5.11. Immediately after line 13, op_2 is decided before op_1.

This follows immediately from Observation 4.5.9, and from line 13 of the algorithm for constructing h. Next, for convenience, we denote the first operation of p_3 as op_3.

Claim 4.5.12. *Immediately before line 13, the order between op_1 and op_3 is not yet decided.*

Proof. Process p_3 has not yet taken any steps (Observation 4.5.6), and thus its operation cannot be decided before op_1 (Observation 4.4.4). Assume by way of contradiction that op_1 is decided before op_3. All the operations of p_2 prior to op_2 are already decided before op_1 (Claim 4.5.4) and thus by Claim 4.5.2, the order between op_1 and op_2 is already decided. But the order between op_1 and op_2 is not yet decided before line 13 (Claim 4.5.4 and Observation 4.5.5), yielding contradiction. □

Claim 4.5.13. *Immediately after completing line 16, the order between op_1 and op_3 is not yet decided.*

Proof. By Claim 4.5.12, the order between op_1 and op_3 is not yet decided before line 13. Steps by p_2 cannot decide the order between op_1 and op_3 in a help-free algorithm, and thus the only step which could potentially decide the order until after line 16 is the step p_1 takes in line 14. Assume by way of contradiction this step decides the order between op_1 and op_3.

If this step decides the order between op_1 and op_3 then after this step op_1 must be decided before op_3. By Corollary 4.1, this step is a failed CAS. Thus, the state immediately before this step and the state immediately after this step are indistinguishable to all processes other than p_1. This contradicts Claim 4.5.3 (2). □

Claim 4.5.14. *Immediately after line 16, the order between op_1 and the operation of p_2 following op_2 is not yet decided.*

Proof. The operation of p_2 following op_2 has not yet begun, and thus it cannot be decided before op_1 (Observation 4.4.4). Assume by contradiction that op_1 is decided before the next operation of p_2. Thus, by Claim 4.4.5, op_1 must be decided before all future operations of p_3, including op_3. But by Claim 4.5.13, op_1 is not yet decided before op_3, yielding a contradiction. □

Corollary 4.2. *Q is not wait-free.*

Proof. By Claim 4.5.8, each execution of the inner loop is finite. Thus, there are infinitely many executions of the main loop. In each such execution, p_1 takes at least a single step in line 14. Thus p_1 takes infinitely many steps. Yet, by combining Claims 4.5.4, and 4.5.7, op_1 is not completed in any iteration of the main loop, which implies it is never completed. Thus, Q is not wait-free. □

Since the assumptions on Q were that it is linearizable, help-free, and lock-free, we can rephrase Corollary 4.2 as follows.

Theorem 4.3. *A wait-free linearizable implementation of an exact order type cannot be help-free.*

It is interesting to note that in history h built in this proof, process p_3 never takes a step. Nevertheless, its existence is necessary for the proof. History h demonstrates that in a lock-free help-free linearizable implementation of an exact order type, a process may fail a CAS infinitely many times, while competing processes complete infinitely many operations. This is indeed a possible scenario in the lock-free help-free linearizable queue of Michael and Scott [MS96], where a process may never successfully ENQUEUE due to infinitely many other ENQUEUE operations.

4.5.1 Generalizing the Proof To Cover the Fetch&Add Primitive

In the proof of Theorem 4.3, we assumed the allowed primitives were READ, WRITE, and CAS. Another primitive, not as widely supported in real machines, is the FETCH&ADD

primitive. As we shall see in Section 4.6, when it comes to the question of wait-free help-free types, the FETCH&ADD primitive adds strength to the computation, in the sense that some types that cannot be implemented in a wait-free help-free manner using only the READ, WRITE, and CAS primitives, can be implemented in a wait-free help-free manner if the FETCH&ADD primitive is allowed (An example for such a type is the fetch&add type itself). However, in this subsection we claim that types such as the queue and stack cannot be implemented in a linearizable, help-free, wait-free manner, even if FETCH&ADD is available. In what follows we give this proof.

If we allow the FETCH&ADD primitive, yet leave the proof of Theorem 4.3 unchanged, the proof fails since Claim 4.5.10 fails. Originally, Claim 4.5.10 shows that immediately before line 13, the next steps in the programs of both p_1 and p_2 are CAS primitives to the same location. Furthermore, the claim shows that each of these CAS operations, if executed immediately, will modify the data structure. That is, the expected-value is the same as the value in the target address, and the new-value is different than the expected-value. Claim 4.5.10 proves this by elimination: it proves that the next steps of both p_1 and p_2 cannot be a READ, a WRITE, or a CAS that doesn't modify the data structure. This remains true when FETCH&ADD is allowed. However, a CAS that changes the data structure ceases to be the only remaining alternative.

We claim that immediately before line 13, it is impossible that the next steps of both p_1 and p_2 are FETCH&ADD, because then $h \circ p_1 \circ p_2$ is indistinguishable from $h \circ p_2 \circ p_1$ by p_3. After any of these two sequences, the order between op_1 and op_2 must be decided, and thus the first one of them must also be decided before all the future operations of p_3 (Claim 4.4.5). Thus, a long enough solo execution of p_3 will reveal which of one of op_1 and op_2 is linearized first, and the indistinguishability yields a contradiction.

Thus, immediately before line 13 it is impossible that the next steps of both p_1 and p_2 are FETCH&ADD. However, it is possible indeed that one of them is FETCH&ADD, and the other is a CAS. This foils the rest of the proof. To circumvent this problem, we add an extra process, denoted p_0. The program of p_0 consists of a single operation, denoted op_0.

A solo execution of p_3 should return different results if op_0 is executed, op_1 is executed, or op_2 is executed. For instance, in the case of the FIFO queue, op_0 can be ENQUEUE(0), op_1 ENQUEUE(1), and op_2 ENQUEUE(2). As before, the program of p_2 is an infinite sequence of ENQUEUE(2) operations. The program of p_3 is an infinite sequence of DEQUEUE operations. In these settings, three process (p_0, p_1, p_2) "compete" to linearize their operation first in each iteration of the main-loop.

The inner loop (originally lines 5–12) is modified to advance the three processes. The conditions in lines 6 and 9 need not be changed; it is enough to check for each operation that it is not decided before one of the other two: at the first time an operation of op_0, op_1 and op_2 is decided before another one of these three operations, it is also decided before the last one. To see this, assume without loss of generality that at the first time such a decision is made, op_1 is decided before op_2. By Claim 4.4.5, it must also be

decided before future operations of p_3. Run p_3 long enough, and see which operation comes first. Since op_0 is not yet decided before op_1, and cannot be decided to be before it during a solo execution of p_3, then p_3 must witness that op_1 is the first linearized operation, which implies that op_1 is decided before op_0 as well.

Thus, after the inner loop, the order between op_0, op_1, and op_2 is not yet decided, but if any of the processes p_0, p_1 or p_2 takes a step, its operation will be decided before the other two. As before, the next step of all of them must be to the same memory location. As before, their next steps cannot be a READ, a WRITE, or a CAS that does not change the memory. It is possible that the next step of one of them is FETCH&ADD, but as shown above, it is impossible that the next step of *two* of them is FETCH&ADD. Thus, the next step of at least one out of p_0 and p_1 must be a CAS. Next, we schedule p_2 to take a step, and afterwards we schedule p_0 or p_1 (we choose the one whose next step is a CAS) to take is step. This step must be a failed CAS.

The proof continues similarly as before. The failed CAS cannot decide an operation before op_3 because of indistinguishability. Process p_2 runs to complete op_2, and the above is repeated with the next operation of p_2. In each iteration of the main loop, at least one of p_0 and p_1 takes a single step, but neither op_0 or op_1 is ever completed, and thus the data structure is not wait-free. The conclusion is that a queue (or a stack) cannot be linearizable, help-free, and wait-free, even if the FETCH&ADD primitive as available.

To generalize this result to a family of types, we need to slightly strengthen the requirements of exact order types. The current definition of exact order types implicitly implies a repeated "competition" between two threads, the result of which can be witnessed by a third thread. Extending this definition to imply a repeated competition of three threads yields the following definition.

Definition 4.5.15. (Extended Exact Order Types.) An extended exact order type T is a type for which there exist two operations op_0 and op_1, an infinite sequence of operations W, and a (finite or an infinite) sequence of operations R, such that for every integer $n \geq 0$ there exists an integer $m \geq 1$, such that for at least one operation in $R(m)$, the value it returns in any execution in $W(n+1) \circ ((R(m) + op_0?) + op_1?)$ differs from the value it returns in any execution in $W(n) \circ op_0 \circ ((R(m) + W_{n+1}?) + op_1?)$, and both differ from the value it returns in any execution in $W(n) \circ op_1 \circ ((R(m) + W_{n+1}?) + op_0?)$.

4.6 Global View Types

In this section we investigate a different set of types, that can also not be obtained in a wait-free manner without using help. These are types that support an operation that returns some kind of a global view. We start by addressing a specific example: a single-scanner snapshot. We later identify accurately what other types belong to this group. The technique of the proof used here is similar to that of Section 4.5, but the

details are different and more complicated.

The single scanner snapshot type supports two operations: UPDATE and SCAN. Each process is associated with a single register entry, which is initially set to $\perp$. An UPDATE operation modifies the value of the register associated with the updater, and a SCAN operation returns an atomic view (snapshot) of all the registers. This variant is referred to as a single-writer snapshot, unlike a mulit-writer snapshot object that allows any process to write to any of the shared registers. In a single scanner snapshot, only a single SCAN operation is allowed at any given moment[3].

Let S be a linearizable, help-free implementation of a single scanner snapshot. We prove that S is not wait-free. For convenience, we assume S is lock-free, as otherwise, it is not wait-free and we are done. Consider a system of three processes, p_1, p_2, and p_3. The program of p_1 is a single UPDATE(0) operation, the program of p_2 is an infinite sequence alternating between UPDATE(0) and UPDATE(1) operations, and the program of process p_3 is an infinite sequence of SCAN operations.

Again, we build an infinite history h, such that the process p_1, p_2, and p_3 follow their respective programs. This time, we show that in h either p_1 executes infinitely many (failed) CAS steps, yet never completes its operation (as before), or alternatively, that starting at some point, neither p_1 nor p_2 complete any more operations, but at least one of them executes infinitely many steps.

The algorithm for constructing this history is depicted in Figure 4.2. In every iteration, the operations of p_1, p_2, p_3 are denoted op_1, op_2, op_3 respectively. In lines 6–13, processes p_1 and p_2 are scheduled to run their programs as long as neither op_1 nor op_2 is decided before op_3. After the loop is ended, if p_1 takes another step op_1 will be decided before op_3, and if p_2 takes another step then op_2 will be decided before op_3.

Then, in lines 14–15, p_3 is run as much as possible without changing the property achieved at the end of the previous loop. That is, when the loop of lines 14–15 is stopped, it is still true that 1) if p_1 takes another step then op_1 will be decided before op_3, and 2) if p_2 takes another step then op_2 will be decided before op_3. However, if p_3 will take another step, then at least one of (1) and (2) will no longer hold.

Now, the execution is divided into two cases. The first possibility is that if p_3 takes another step, both (1) and (2) will cease to hold simultaneously. In this case, similarly to the proof of Theorem 4.3, we show that both the CAS operations of p_1 and p_2 are to the same address, we allow p_2 to successfully executes its CAS, and let p_1 attempt its CAS and fail. Afterwards both op_2 and op_3 are completed, and we repeat the process with the next operations of p_2 and p_3.

The other possibility is that the next step of p_3 only causes one of the conditions (1) and (2) to cease to hold. Then, we allow p_3 to take the next step, and afterwards schedule the process (either p_1 or p_2) that can take a step without causing its operation

[3]Formally, the type is a snapshot, and a single-scanner implementation is a constrained implementation of it, in the sense that its correctness is only guaranteed as long as no two SCAN operations are executed concurrently.

```
1:  h = ε;
2:  while (true)                                                          ▷ main loop
3:    op1 = the first uncompleted operation of p1;
4:    op2 = the first uncompleted operation of p2;
5:    op3 = the first uncompleted operation of p3;                        ▷ scan operation
6:    while (true)                                                        ▷ first inner loop
7:      if op1 is not decided before op3 in h ∘ p1
8:        h = h ∘ p1;
9:        continue;                                                       ▷ goto line 6
10:     if op2 is not decided before op3 in h ∘ p2
11:       h = h ∘ p2;
12:       continue;                                                       ▷ goto line 6
13:     break;                                                            ▷ goto line 14
14:   while (op1 is decided before op3 in h ∘ p3 ∘ p1 and op2 is decided before op3 in h ∘ p3 ∘ p2)   ▷
    second inner loop
15:     h = h ∘ p3
16:   if (op1 is not decided before op3 in h ∘ p3 ∘ p1 and op2 is not decided before op3 in h ∘ p3 ∘ p2)
17:     h = h ∘ p2;                                          ▷ this step will be proved to be a CAS
18:     h = h ∘ p1;                                  ▷ this step will be proved to be a failed CAS
19:     while (op2 is not completed in h)                    ▷ run p2 until op2 is completed
20:       h = h ∘ p2;
21:   else
22:     Let k ∈ {1, 2} satisfy opk is not decided before op3 in h ∘ p3 ∘ pk
23:     Let j ∈ {1, 2} satisfy opj is decided before op3 in h ∘ p3 ∘ pj
24:     h = h ∘ p3;
25:     h = h ∘ pk;
26:   while (op3 is not completed in h)                      ▷ run p3 until op3 is completed
27:     h = h ∘ p3;
```

Figure 4.2: The algorithm for constructing the history in the proof of Theorem 4.7.

to be decided before op_3. We prove this step is not a "real" progress, and cannot be the last step in the operation. Afterwards we allow op_3 to be completed, and repeat the process with the next operation of p_3.

Throughout the proof we avoid using the fact that a SCAN (op_3) returns a different result when it is linearized after op_1 and before op_2 compared to when it is linearized before op_2 and after op_1. We rely only on the fact that op_3 returns three different results if it is linearized before both UPDATE operations, before one of them, or after both. This more general approach slightly complicates the proof in a few places, but it makes the proof hold for additional types. In particular, this way the proof also holds for an increment object.

We use a similar inductive process as we did when proving Theorem 4.3: we prove a series of claims on the execution of history h, which is a history of object S. These claims are proved by induction, where the induction variable is the iteration number of the main loop (lines 2–27). The induction hypothesis is that claims (4.6.1–4.6.13) are correct. Claim 4.6.1 is the only one to use the induction hypothesis directly, while the other claims follow from it.

Claim 4.6.1. *Immediately after line 5, it holds that 1) the operation op_3 has not yet started, 2) the order between op_1 and op_3 is not yet decided, 3) the order between op_2*

and op_3 is not yet decided.

Proof. For the first iteration, none of the operations has started, thus the claim holds by Observation 4.4.4. For iteration $i \geq 2$, the claim follows from the induction hypothesis and Claim 4.6.13. □

Claim 4.6.2. *Immediately before line 14, it holds that 1) the operation op_1 is not decided before any operation by either p_2 or p_3, 2) the operation op_2 is not decided before any operation by either p_1 or p_3, and 3) the operation op_3 is not decided before any operation by either p_1 or p_2.*

Loosely speaking, this claim states that no new ordering is decided during the execution of the first inner loop (lines 6–13).

Proof. By Claim 4.6.1(1), op_3 has not yet started after line 5. Since p_3 never advances in lines 6–13 then op_3 has not yet started immediately before line 14. Thus, op_3 cannot be decided before any operation of a different process (Observation 4.4.4), and we obtain (3). □

We now turn to prove (1). First, we observe that before line 14, op_1 is not decided before op_3: by Claim 4.6.1(2), the operation op_1 is not decided before op_3 after line 5; the condition in line 7 guarantees that op_1 is not decided before op_3 as result of line 8, and the fact that the algorithm is help-free guarantees op_1 is not decided before op_3 as result of line 11.

Second, we claim op_1 is not decided before any operation of p_2. Assume by way of contradiction that op_1 is decided before an operation op of p_2. Thus, op_1 must be decided before all future operations of p_3 (Claim 4.4.5), including op_3. We just proved that op_1 is not decided before op_3, yielding a contradiction. Therefore, op_1 cannot be decided before any operation of p_2.

Finally, we claim that op_1 is not decided before any future operation of p_3. Assume by way of contradiction that op_1 is decided before an operation op of p_3. Using again Claim 4.4.5, op_1 must be decided before all future operations of p_2, yielding contradiction.

Thus, we have shown that immediately before line 14, the operation op_1 is not decided before op_3, not decided before any operation of p_2, and not decided before any (future) operation of p_3 as well, and (1) is proved. Condition (2) is proven the same way as (1).

Claim 4.6.3. *No operation in h is completed during the execution of the first inner loop (lines 6–13).*

Proof. By Claim 4.6.2, neither op_1 nor op_2 are decided before op_3 immediately before line 14. However, at the same point, op_3 has not yet begun (by Claim 4.6.1 and observing the code). If op_1 (or op_2) were completed immediately before line 14, then by

Observation 4.4.4, it must have been decided before the future operation op_3. Thus, neither op_1 nor op_2 are completed immediately before line 14. Since only p_1 and p_2 take steps during the first inner loop and they do not complete their operations, it follows that no operation is completed. □

Claim 4.6.4. *No operation in h is completed during the execution of the second inner loop (lines 14–15).*

Proof. By Claim 4.6.2, neither op_1 nor op_2 are decided before op_3 immediately before line 14. The operation op_3 itself has not yet begun before line 14. The condition in line 14 guarantees that op_3 will not be decided before op_1 or op_2 during the execution of the second inner loop, since after the second inner loop is over, a single step by p_1 (p_2) will decide op_1 (op_2) before op_3. Thus, after the second inner loop, the order between op_3 and op_1 and the order between op_3 and op_2 are not yet decided. In what follows we show that op_3 cannot be completed before these orders are decided, and thus reach the conclusion that op_3 cannot complete during the execution of the second inner loop.

The result of op_3 depends on the orders between op_3 and op_2, and between op_3 and op_1: the operation op_3 returns a certain result if op_3 is linearized before both op_1 and op_2, a different result if op_3 is linearized after both the other operations, and yet a different result than both previous results if op_3 is linearized before only one of op_1 and op_2.

It follows that if the result of op_3 is consistent with none of op_1 and op_2 linearized before it, then op_3 must already be decided before both op_1 and op_2. If the result is consistent with both op_1 and op_2 being linearized after op_3, then both must already be decided before op_3. Next, we claim that if the result is consistent with op_3 being linearized before exactly one of op_1 and op_2, then it must already be decided before each one. Assume by way of contradiction op_3 is not yet decided before either op_1 or op_2, but returns a result consistent with being linearized before exactly one of them.

According to the condition of line 14, after the second loop is completed, at least one of op_1 and op_2 will be decided before op_3 if its owner process will take one step. Let the owner take this step, and its operation (either op_1 or op_2) is now decided before op_3. Thus, op_3 must now be decided before the other operation (one of op_1 and op_2). However, in a help-free implementation op_3 cannot be decided before another operation as a result of a step taken by a process other than p_3, and thus op_3 must have been decided before either op_1 or op_2 before the second inner loop was completed, which yields a contradiction.

To conclude, op_3 cannot be completed before the order is decided, which means op_3 cannot be completed during the second inner loop. No other operation can be completed during the second inner loop as p_3 is the only process that advances in that loop. □

Claim 4.6.5. *The executions of the first inner loop (lines 6–13) and second inner loop (lines 14–15) are finite.*

Proof. In each iteration of the first and second inner loops, a process advances a step in h. The history h is a history of the lock-free object S, and thus an infinite execution without completing an operation is impossible. By Claims 4.6.3 and 4.6.4, no operation is completed in these two loops, and thus their execution must be finite. □

Claim 4.6.6. *Immediately before line 16, it holds that 1) the operation op_1 is not decided before any operation by either p_2 or p_3, and 2) the operation op_2 is not decided before any operation by either p_1 or p_3.*

Proof. These conditions have already been shown to hold before line 14 (Claim 4.6.2). The only process that takes steps in the second inner loop (lines 14–15) is p_3. In a help-free algorithm, steps by p_3 can only decide an operation of p_3 before any other operation. □

Observation 4.6.7. If the condition in line 16 is true, then immediately before line 17, the operation op_1 is decided before op_3 in $h \circ p_1$, the operation op_1 is not decided before op_3 in $h \circ p_3 \circ p_1$, the operation op_2 is decided before op_3 in $h \circ p_2$, and the operation op_2 is not decided before op_3 in $h \circ p_3 \circ p_2$.

Claim 4.6.8. *If the condition in line 16 is true, then immediately before line 17 the following holds.*
(1.) The next primitive step in the programs of p_1, p_2, and p_3 is to the same memory location.
(2.) The next primitive step in the programs of both p_1 and p_2 is a CAS.
(3.) The expected-value of both the CAS *operations of p_1 and p_2 is the value that appears in the designated address.*
(4.) The new-value of both the CAS *operations is different than the expected-value.*

Proof. By Observation 4.6.7, in $h \circ p_1 \circ p_3$, the operation op_1 is decided before op_3, while in $h \circ p_3 \circ p_1$, the operation op_1 is not decided before op_3. Thus, in an execution in which p_3 runs solo and completes op_3 immediately after $h \circ p_1 \circ p_3$ it must return a different result than in an execution in which p_3 runs solo and completes op_3 immediately after $h \circ p_3 \circ p_1$ (because each operation by p_1 changes the return value of op_3). Thus, $h \circ p_3 \circ p_1$ and $h \circ p_1 \circ p_3$ must be distinguishable, and thus the next primitive step in the programs of both p_1 and p_3 must be to the same memory location. Similarly, the next primitive step in the programs of both p_2 and p_3 must be to the same memory location, and (1) is proved.

As mentioned, op_3's result is different if p_3 completes op_3 solo immediately after $h \circ p_1$ than op_3's result if p_3 completes op_3 solo immediately after $h \circ p_3 \circ p_1$. Thus, the next primitive step by the program of p_1 cannot be a READ, otherwise the two executions will be indistinguishable by p_3. Similarly, the next primitive step of p_1 cannot be a CAS that does not change the shared memory (i.e., a CAS in which the expected-value is different from the value in the target address, or a CAS in which the expected-value and

the new-value are the same.) The symmetric argument for p_2 demonstrates that the next step in the program of p_2 cannot be a READ or a CAS that does not change the shared memory as well.

Thus, the next steps of both p_1 and p_2 are either a WRITE, or a CAS that satisfies (3) and (4). It remains to show the next steps are not a WRITE. Assume by way of contradiction that the next step by p_1 is a WRITE. Thus, $h \circ p_2 \circ p_1$ is indistinguishable from $h \circ p_1$ to all processes excluding p_2. Assume that after either one of these two histories, p_1 runs solo and completes the execution of op_1, and immediately afterwards, p_3 runs solo and completes the execution of op_3. If p_2 executes the next step following h, then op_3 should return a result consistent with an execution in which both op_1 and op_2 are already completed. In the other case, in which the step following h is taken by p_1, op_3 should return a result consistent with an execution in which op_1 is completed and op_2 is not. Since both of these results are different, but the histories are indistinguishable to p_3, we reach a contradiction. Thus, in h, the next step by p_1 is not a WRITE, and similarly, the next step by p_2 is also not a WRITE. □

Claim 4.6.8 immediately implies:

Corollary 4.4. *If the condition in line 16 is true, then the primitive step p_2 takes in line 17 is a successful* CAS*, and the primitive step p_1 takes in line 18 is a failed* CAS*.*

Claim 4.6.9. *If the condition in line 16 is true, then immediately after line 18, the operation op_1 is not decided before any operation of p_3.*

Proof. Immediately before line 16, the operation op_1 is not decided before any operation of p_3 by claim 4.6.6. In a help-free implementation such as S, an operation can only be decided before another operation following a step of its owner process. Following this rule, the only step which could potentially decide op_1 before any operation of p_3 is the step p_1 takes at line 18. By Corollary 4.4, this step is a failed CAS. Thus the state before this step and after this step are indistinguishable to all processes excluding p_1. Assume by way of contradiction that this failed CAS decides op_1 to be before an operation op of p_3. If p_3 is run solo right before the failed CAS of p_1, and this run is continued until op is completed, the result of op should be consistent with op_1 not yet executed (since op_1 cannot be decided before op in a help-free implementation during a solo execution of p_3); If p_3 is run solo right after the failed CAS of p_1, and this run is continued until op is completed, the result of op should be consistent with op_1 already executed. These two scenarios are indistinguishable by p_3, yet the results are different according to the semantics of the specification and the respective programs, yielding a contradiction. □

Corollary 4.5. *If the condition in line 16 is true, then immediately after line 18, the operation op_1 is not yet completed.*

Proof. Immediately after line 18, the operation op_1 is not decided before any operation of p_3 (Claim 4.6.9). Were op_1 completed, then by Observation 4.4.4, it must have also been decided before all future operations of p_3 that have not started yet. □

Claim 4.6.10. *If the condition in line 16 is false, then immediately after line 25, the order between op_j and op_3 is not yet decided. Furthermore, the order between op_j and any of the future operation by p_3 is not decided as well.*

Proof. Immediately before line 14 the order between op_j and op_3, or between op_j and any future operation of p_3 is not yet decided (Claim 4.6.2). In lines 14–15, p_3 is the only process to advance. By the condition in line 14, op_3 is not decided before neither op_1 or op_2 during the execution of the second inner loop. Furthermore, by the condition in line 16, and by the definition of op_j, op_3 is not decided before op_j after line 24. p_j did not make any step since line 14, and thus op_j cannot be decided before op_3 or any future operation of p_3. □

Observation 4.6.11. If the condition in line 16 is false, then immediately after line 25, op_j is not yet completed.

The above is true because op_j did not execute a step since line 14, and was not completed at the time (Claim 4.6.3).

Claim 4.6.12. *If the condition in line 16 is false, then immediately after line 25, op_k is not decided before any operation of p_3.*

Proof. According to the condition of line 16 and the definition of k, after line 25 op_k is not decided before op_3. Immediately after line 25, the order between op_3 and op_j is not yet decided (Claim 4.6.10). Thus, op_k is not decided before op_j (because op_j may still be before op_3, which in turn may still be before op_k). Assume by contradiction that after line 25, op_k is decided before some operation op of p_3. Note that op_j is not decided before op at this point (Claim 4.6.10). Let p_3 run solo until op is completed.

We claim that after such a run, op is decided before op_j. It is already assumed (contradictively) that op_k is decided before op; no future operations of p_k (operations not yet started) can be decided before the already completed op (Observation 4.4.4); all operations of p_j before op were completed before op has begun, and are thus before it.

Thus, for every operation $O \neq op_j$ the order between O and op is already decided. According to the semantics of the specification, op returns a different result if op_j is before op than if op_j is after op, given that the relative order between op and all other operations is fixed. Consequently, once op is completed, the order between op_j and op must also be decided. However, op_j cannot be decided before op: it was not decided before op immediately after line 25 and p_j has not taken a step since. The only remaining possibility is that op is decided before op_j.

If op is indeed decided before op_j then by transitivity op_k is decided before op_j. However, since p_3 is not the owner of op_k, then a solo execution of p_3 cannot decide op_k to be before op_j in the help-free S, yielding a contradiction. □

Corollary 4.6. *If the condition in line 16 is false, then immediately after line 25, op_k is not yet completed.*

Proof. Immediately after line 25, the operation op_k is not decided before any operation of p_3 (Claim 4.6.12). Were op_k completed, then by Observation 4.4.4 it must have been decided before all future operations of p_3 that have not started yet. □

Claim 4.6.13. *Immediately after exiting the loop of lines 26–27, it holds that 1) the operation p_3 has completed operation op_3 and has not yet started the next operation, 2) the operation op_1 has not yet completed, 3) the order between op_1 and any future operation of p_3 is not yet decided, and 4) the order between the first uncompleted operation of p_2 and any future operation of p_3 is not yet decided.*

Proof. By Claim 4.6.4, op_3 is not completed in lines 14–15. In lines 16–25, p_3 takes at most one step (line 24), because op_3 is not completed in lines 14–15, the step must be a step of op_3, and not of the next operation of p_3. The code of lines 26–27 ensures p_3 will complete op_3 if it is not yet completed, but will not start the next operation, guaranteeing (1).

Now, divide into two cases. If the condition in line 16 is true, then op_2 is completed in lines 19–20. Thus, the first uncompleted operation of p_2 has not yet begun. The order between two operations that have not yet begun cannot be decided (Observation 4.4.4), and we get (4). The operation op_1 has not yet completed by Corollary 4.5, giving (2). The operation op_1 was not decided before any operation of p_3 after line 18 (Claim 4.6.9). Process p_1 did not take another step since line 18 and S is a help-free implementation, thus op_1 is not decided before any operation of p_3. Any future operation of p_3 cannot be decided before op_1 by Observation 4.4.4, and thus we get (3).

If the condition in line 16 is false, then op_1 and op_2 are op_j and op_k (not necessarily in that order). Immediately after line 25, op_k is not decided before any operation of p_3 (Claim 4.6.12), op_k is not completed (Corollary 4.6), op_j is not decided before any operation of p_3 (Claim 4.6.10), and op_j is not completed (Observation 4.6.11)). In lines 26–27 only p_3 may progress, thus both op_1 and op_2 cannot be completed (guaranteeing (2)), and cannot be decided before any other operation since S is help-free (guaranteeing (3) and (4)). □

Claim 4.6.14. *Every iteration of the main loop (lines 2–27) is finite.*

Proof. In every iteration of the main loop, the executions of the first inner loop (lines 6–13) and second inner loop (lines 14–15) are finite (Claim 4.6.5). The other two inner loops (lines 19–20 and 26–27) run a single process exclusively until it completes its

operation, which always takes a finite number of execution steps in a lock-free algorithm. Thus, each execution of an inner loop is finite, as every iteration of the main loop is finite. □

Claim 4.6.15. *S is not wait-free.*

Proof. By Claim 4.6.14, each iteration of the main loop is finite. It follows that when history h is constructed following the algorithm in Figure 4.2 the main loop is run infinitely many times. Thus, we consider two cases. The first case is that the condition in line 16 is true only a finite number of times (in a finite number of iterations of the main loop). In this case, We consider the part of history h created since after the last iteration in which the condition in line 16 is true. If the condition is never true, we consider the entire history h. In this part of the history, neither p_1 nor p_2 complete any operation: in each iteration these operations are not completed until after line 25 (Corollary 4.6, Observation 4.6.11), and only p_3 makes progress in lines 26–27. On the other hand, in each iteration at least one of p_1 and p_2 takes at least one step - in line 25. This contradicts wait-freedom.

The second case is that the condition in line 16 is true infinitely many times. In this case, op_1 is never completed (Claim 4.6.13 (2)), yet p_1 takes infinitely many steps: each time the condition in line 16 is true, p_1 takes a step in line 18, also contradicting wait-freedom. □

Since the assumptions on S were that it is linearizable, help-free, and lock-free, we can rephrase Claim 4.6.15 as follows.

Theorem 4.7. *A wait-free linearizable single-scanner snapshot implementation cannot be help-free.*

4.6.1 From Single Scanner Snapshot to Global View Types

The first natural observation is that if a wait-free linearizable single-scanner snapshot cannot be implemented without help, then this conclusion holds for more general snapshot variants as well, such as the multiple-scanner snapshot object, or simply the snapshot object. However, we can generalize the result further. The proof relies on the fact that for every SCAN, its result changes if it is linearized before op_1 and op_2, compared to when it is linearized after the first of op_1 and op_2, and compared to when it is linearized after both.

In what follows, we generalize this result to global view types. Similarly to the proof above, we think of a single operation op (similar to op_1 of p_1), an infinite sequence of operations Modifiers (similar to the infinite UPDATE sequence of op_2, and an infinite sequence of operations Views (similar to the infinite SCAN sequence of p_3).

Next, a certain property that holds for every *modifier* and every *view* operation is needed. Specifically, this property states that the view returns a different result if

another (either modifier or the *op* operation) is added before it, and yet a different result if both the modifier and *op* are added before it. For this purpose, we define three sets of sequential histories for each pair of modifier and view. Set_0 is histories in which the view is after the specified modifier, but not after any other modifier, and not after *op* either. Set_1 is histories in which either *op* or one more modifier is before the view, and Set_2 is the histories in which both the one more modifier and *op* are before the view. The definition follows.

Definition 4.6.16. (Modifiers-Viewers Sets.) Given a single operation denoted *op*, an infinite sequence of operations denoted Modifiers, and an infinite sequence of operations denoted Views, for every pair of integers $i \geq 0$ and $j \geq 1$ we define the following three *modifier-i-view-j* sets.

Set_0 is the set of all sequential histories h that include the first i Modifiers operations in their relative order, include the first j Views operations in their relative order, include no other operation and the last operation in h is in Views.

Set_1 is the set of all sequential histories h that include the first i Modifiers operations in their relative order, include the first j Views operations in their relative order, include *op* or include the $(i+1)$-st operation of Modifiers somewhere after the first i operations of Modifiers but not both, include no other operations, and the last operation in h is in Views.

Set_2 is the set of all sequential histories h that include the first $i+1$ operations of Modifiers in their relative order, include the first j operations of Views in their relative order, include *op*, include no other operations, and the last operation in h is in Views.

The interests we have in these sets relies in the result of the last (view) operation in each history. Specifically, for our proof to hold, if two histories h and h' belong to two different modifier-i-view-j set, then the results of their last operation should be different. We use the following definition to help formalize this.

Definition 4.6.17. (Modifiers-Viewers Result Sets.) Given a single operation denoted *op*, an infinite sequence of operations denoted Modifiers, and an infinite sequence of operations denoted Views, for every pair of integers $i \geq 0$ and $j \geq 1$ we define the following three *modifier-i-view-j-results* sets as follows:
$RS_i = \{r | r$ is the returned value of the last (view) operation in a history $h \in Set_i \}$

Definition 4.6.18. (Global View Types.) A type t, for which there exists an operation *op*, an infinite sequence of operations Modifiers and an infinite sequence of operations Views, such that for every pair of integers $i \geq 0$ and $j \geq 1$ the three modifier-i-view-j-results sets are disjoint sets, is called a *Global View Type.*

Using this definition, Theorem 4.7 is generalized as follows:

Theorem 4.8. *A global view type has no linearizable, wait-free, help-free implementation.*

Both snapshot objects and increment objects are such types.[4] Another interesting type is the fetch&increment type, which is sometimes used as a primitive. This type supports only a single operation, which returns the previous integer value and increments it by one. In the type of the fetch&increment, *op*, the Modifiers sequence, and the Views sequence, all consists only fetch&increment operations. Its easy to see that for every pair i and j, the three results sets are disjoint, because each set contains histories with a different number of operations. Finally, the fetch-and-cons object, used in [Her88], is another example of a type that satisfies the condition in theorem 4.8.

4.7 Max Registers

In this section we turn our attention to systems that support only the READ and WRITE primitives, (without the CAS primitive). We prove that for such systems, help is often required even to enable lock-freedom. We prove this for the *max-register* type [AACH12]. A max-register type supports two operations, WRITEMAX and READMAX. A WRITEMAX operation receives as an input a non-negative integer, and has no output result. A READMAX operation returns the largest value written so far, or 0, if no WRITEMAX operations were executed prior to the READMAX. If the CAS primitive is allowed, then there is a help-free wait-free max-register implementation. (See Subsection 4.8.2.)

Assume by way of contradiction that M is a linearizable, help-free, lock-free max-register implementation. Consider a system of five processes, p_1, p_2, p_3, p_4, and p_5. The programs of processes p_1, p_2, and p_3 all consists of a single operation, which is WRITEMAX(1), WRITEMAX(2), and WRITEMAX(3) respectively. The programs of processes p_4 and p_5 are both a single READMAX operation. We denote the operations of p_1, p_2, and p_3 by W_1, W_2, and W_3 respectively. We denote the operations of p_4 and p_5 by R_1 and R_2 respectively.

In the proof we build a history h, such that the processes p_1, p_2, p_3, p_4, and p_5 follow their respective programs. We show this yields a contradiction to help-freedom. The algorithm for constructing this history is depicted in Figure 4.3. Processes p_1, p_2 are scheduled to run their programs as long as their operations are not decided before R_1. Process p_4 is scheduled to run its program as long is it can make a step without deciding its operation before W_1, or alternatively, without deciding its operation before W_2. We prove that during the execution of the main loop, no operation is ever decided before any other operation. We also prove that the execution of the main loop must be finite.

Afterwards, the execution of M is in a critical point. If p_1 were to take a step, then W_1 will be decided before R_1; if p_2 were to take a step, then W_2 will be decided before R_1; and if p_4 were to take a step, then R_1 will be decided before both W_1 and W_2. We will prove using indistinguishability arguments that this yields a contradiction.

[4] An increment object supports two operations, INCREMENT and GET.

```
1:  h = ϵ;
2:  while (true)                                              ▷ main loop
3:    if W1 is not decided before R1 in h ∘ p1
4:      h = h ∘ p1;
5:      continue;                                             ▷ goto line 2
6:    if W2 is not decided before R1 in h ∘ p2
7:      h = h ∘ p2;
8:      continue;                                             ▷ goto line 2
9:    if R1 is not decided before W1 in h ∘ p4
10:     h = h ∘ p4;
11:     continue;                                             ▷ goto line 2
12:   if R1 is not decided before W2 in h ∘ p4
13:     h = h ∘ p4;
14:     continue;                                             ▷ goto line 2
15:   break;                                                  ▷ goto line 16
16:   Contradiction          ▷ reaching this line immediately yields contradiction.
```

Figure 4.3: The algorithm for constructing the history in the proof of Theorem 4.11.

Claim 4.7.1. *During the execution of the main loop (lines 2–15), no operation is decided before any other operation.*

Proof. The proof is by induction on the iteration number of the main loop. The induction hypothesis for iteration i is that no operation is decided before another operation during the execution of the first $i-1$ iterations. For the first iteration this is trivial. We assume that the hypothesis is correct for iteration i, and prove that it holds for iteration $i+1$ as well. That is, we prove that given that no operation is decided in the first $i-1$ iterations, no operation is decided in iteration i.

Before the execution of the main loop, no operation is decided before any other operation because h is empty (Observation 4.4.4). By the induction hypothesis, no operation was decided during the first $i-1$ iterations of the main loop. It follows that if an operation is decided before another operation during iteration i of the main loop, then it must be the first time any operation is decided before a different operation in h. If this indeed happens, then it must be in one of the lines: 4,7,10, or 13. We go over them one by one, and prove that an execution of none of them can be the first time an operation is decided before a different operation.

Assume by way of contradiction that the execution of line 4 in iteration i is the first time in which an operation is decided before a different operation. Because M is a help-free algorithm, the operation that is decided before a different operation must be W_1. Since W_1 is now decided before a different operation, and since W_3 is not yet decided, then in particular, W_1 must be decided before W_3 (Claim 4.4.5).

At this point, let p_3 run solo until completing W_3. We consider two cases. The first case is that after this run of p_3, W_3 is decided before R_1. If this is the case, then by transitivity, W_1 must also be decided before R_1. However, in a help-free algorithm W_1 cannot be decided before R_1 during a solo execution of p_3. It follows that W_1 was already decided before R_1 prior to this solo execution. Thus, W_1 must be decided before

R_1 in the execution of line 4. But this contradicts the condition in line 3, and thus the first case is impossible.

The second case is that after the solo run of p_3 which completes W_3, W_3 is not decided before R_1. If W_3 is not decided before R_1 after the completion of W_3, then it follows that W_3 can never be decided before R_1 in a help-free algorithm (because any such future decision cannot be inside the execution of W_3, and will thus be help.) Thus, R_1 cannot possibly return any value ≥ 3: returning such a value would indicate it is after W_3 (because W_3 is decided to be the first operation that writes a value ≥ 3, as no other operation that writes a value ≥ 3 has even started, and W_3 is already completed). But if R_1 cannot possibly return a value ≥ 3, then R_1 is decided before W_3. However, R_1 cannot be decided before another operation during the execution of line 4 or during the solo execution of p_3 in a help-free algorithm. Since we assumed that line 4 is the first time any operation is decided before any other operation, this yields a contradiction, making the second case impossible as well.

Thus, we have established that line 4 in iteration i of the main-loop cannot decide any operation before any other operation. The argument for line 7 is similar. We move on to consider line 10.

Assume by way of contradiction that the execution of line 10 in iteration i is the first time in which an operation is decided before a different operation. Because M is a help-free algorithm, the operation that is decided before a different operation must be R_1. Since R_1 is now decided before a different operation, and since R_2 is not yet decided, then in particular, R_1 must be decided before R_2 (Claim 4.4.5).

At this point, let p_5 run solo until completing R_2. We consider two cases. The first case is that R_2 returns 0. If this is the case, then R_2 is decided before both W_1 and W_2, and by transitivity, R_1 is decided before W_1 and W_2 as well. However, R_1 cannot be decided before W_1 in a help-free algorithm during a solo execution of p_5. It follows that R_1 was already decided before W_1 before this solo execution. Thus, R_1 must be decided before W_1 in the execution of line 10. But this contradicts the condition in line 9, and thus the first case is impossible.

The second case is that R_2 returns a value greater than 0. In such a case, either W_1 or W_2 must be decided before R_2 (depending on the value returned). But both W_1 and W_2 cannot be decided before R_2 during the solo execution of p_5, or during the execution of line 10, in a help-free algorithm. Since we assumed line 10 is the first time any operation is decided before any other operation, then this yields a contradiction, making the second case impossible as well.

Thus, we have established that line 10 in iteration i of the main-loop cannot decide any operation before any other operation. The argument for line 13 is similar, and the proof is complete. □

Corollary 4.9. *No operation is completed during the execution of the main loop (lines 2–15).*

Proof. By Claim 4.7.1, no operation is decided before any other operation during the execution of the main loop. By Observation 4.4.4, an operation that is already completed must be decided before all operations that have not yet started. Since there are operations that are never started (W_3, R_2), but no operation is decided before any operation, then no operation can be completed during the execution of the main loop.□

Corollary 4.10. *The execution of the main loop (lines 2–15) is finite.*

Proof. In each iteration of the main loop excluding the last one, a process takes a step in M. However, during the execution the main loop no operation is completed. (Corollary 4.9.) Since M is a lock-free implementation then this cannot continue infinitely, and thus the execution of the main loop is finite. □

Observation 4.7.2. Immediately before line 16 the order of any two operations is not yet decided. Furthermore, immediately before line 16, it holds that 1) in $h \circ p_1$, the operation W_1 is decided before R_1, 2) in $h \circ p_2$, the operation W_2 is decided before R_1, and 3) in $h \circ p_4$, the operation R_1 is decided before both W_1 and W_2.

From Claim 4.7.1 the order between any two operations is not decided immediately before line 16.
From observing the code, the main loop exits and line 16 is reached only if (1), (2), and (3) hold.

Claim 4.7.3. *Reaching line 16 yields contradiction.*

Proof. By Observation 4.7.2, in $h \circ p_1 \circ p_4$, the operation W_1 is decided before R_1, and in $h \circ p_4 \circ p_1$, the operation R_1 is decided before both W_1 and W_2. It follows that $h \circ p_1 \circ p_4$ and $h \circ p_4 \circ p_1$ must be distinguishable to process p_4, since if p_4 continues to run solo after $h \circ p_4 \circ p_1$ then R_1 must return 0, and if p_1 runs solo after $h \circ p_1 \circ p_4$ then R_1 must return at least 1. It follows that the next steps of both p_1 and p_4 must be to the same memory address. Furthermore, to enable distinguishability by p_4, the next step by p_1 must be a WRITE, and the next step by p_4 must be a READ.

For similar reasons, the next step of p_2 must also be a WRITE to the same memory address. Thus, the next step by both p_1 and p_2 is a WRITE to the same location. Thus, $h \circ p_2 \circ p_1$ and $h \circ p_1$ are indistinguishable to p_4. Since in $h \circ p_2 \circ p_1$ the operation W_2 is decided before R_1, a solo execution by p_4 starting from that point until R_1 is completed must cause R_1 to return 2. Since this is indistinguishable to p_4 from $h \circ p_1$, then a solo execution of p_4 immediately after $h \circ p_1$ must also return 2. However, this would imply W_2 is decided before R_1. But W_2 is not decided before R_1 in h (Observation 4.7.2), and cannot be decided before it during a step of p_1 or during the solo execution of p_4 in a help-free algorithm, yielding a contradiction. □

Theorem 4.11. *A lock-free implementation of a max-register using only* READ *and* WRITE *primitives cannot be help-free.*

Proof. We assumed a lock-free help-free implementation of a max-register using only READ and WRITE primitives. However, while examining the algorithm for constructing history h depicted in Figure 4.3, we reached the conclusion that the main-loop execution must be finite (Corollary 4.10), but also the conclusion that line 16 can never be reached (Claim 4.7.3). This yields contradiction, and proves the Theorem. □

4.8 Types that Do Not Require Help

In this section, we establish that some types can be implemented in a wait-free manner without using help. Loosely speaking, if the type operations dependency is weak enough then no help is required. As a trivial example, consider the *vacuous type*. A vacuous object supports only one operation, NO-OP, which receives no input parameters and returns no output parameters (void). Thus, the result of a NO-OP does not depend on the execution of any previous operations. Consequently, there is no operations dependency at all in the vacuous type. It can trivially be implemented by simply returning void without executing any computation steps, and without employing help.

4.8.1 A Help-Free Wait-Free Set

As a more interesting example, consider the *set* type of a finite domain. The set type supports three operations, INSERT, DELETE, and CONTAINS. Each of the operations receives a single input parameter which is a key in the set domain, and returns a boolean value. An INSERT operation adds the given key to the set and returns true if the key is not already in the set, otherwise it does nothing and returns false. A DELETE operation removes a key from the set and returns true if the key is present in the set, otherwise it does nothing and returns false. A CONTAINS operation returns true if and only if the input key exists in the set.

Consider the following wait-free help-free set implementation given in Figure 4.4. The implementation uses an array with a bit for every key in the set domain. Initially, all bits are set to zero, and the set is empty. To insert a key to the set, a process performs a CAS operation that changes the bit from zero to one. If the CAS succeeds, the process returns true. If the CAS fails, that means that the key is already in the set, and the process returns false. Deletion is executed symmetrically by CASing from one to zero, and contains reads the appropriate bit and returns true if and only if it is set to one.

In this set algorithm, it is easy to specify the linearization point of each operation. In fact, every operation consists of only a single computation step, which is the linearization point of that operation. For any type, an obstruction-free implementation in which the linearization point of every operation can be specified as a step in the execution of *the same* operation is help-free.

The function f that proves such an implementation is help-free is derived naturally

```
1: bool insert(int key) {
2:    bool result = CAS(A[key],0,1);                    ▷ linearization point
3:    return result; }
4: bool delete (int key) {
5:    bool result = CAS(A[key],1,0);                    ▷ linearization point
6:    return result; }
7: bool contains (int key) {
8:    bool result = (A[key] == 1);                      ▷ linearization point
9:    return result; }
```

Figure 4.4: A help-free wait-free set implementation

from the linearization points. For each given history, the operations are ordered according to the order of the execution of their linearization points. Consider a type T, an obstruction-free implementation of it O, and the corresponding set of histories H. Assume the code of O specifies the linearization point of each operation at the execution of a specific computation step of the same operation. Let f be the linearization function derived from this specification.

Claim 4.8.1. *For every $h \in H$, every two operations op_1, op_2, and a single computation step γ such that $h \circ \gamma \in H$, it holds that if op_1 is decided before op_2 in $h \circ \gamma$ and op_1 is not decided before op_2 in h, then γ is the linearization point of op_1.*

As a direct result, γ is executed by the owner of op_1, and thus O is help-free.

Proof. First, we observe that op_1 is not yet linearized in h. If it were, then the order between op_1 and op_2 would have already been decided: were op_2 linearized before op_1 then op_2 would have been decided before op_1, and were op_1 linearized before op_2 or op_1 is linearized and op_2 not, then op_1 would have been decided before op_2. Thus, op_1 cannot be linearized in h.

Second, we observe that op_1 is linearized in $h \circ \gamma$. Were it not, then a solo execution of the owner of op_2 until the linearization of op_2 would have linearized op_2 before op_1, contradicting the assumption that op_1 is decided before op_2 in $h \circ \gamma$. □

4.8.2 A Help-Free Wait-Free Max Register

In Section 4.7 we proved that a lock-free max register cannot be help-free if only READS and WRITES are available. In this subsection we show that a help-free wait-free max register is possible when using the CAS primitive. The implementation uses a shared integer, denoted `value`, initialized to zero. This integer holds the current max value. The implementation is given in Figure 4.5.

A WRITEMAX operation first reads the shared integer `value`. If it is greater than or equal to the input key, then the operation simply returns. Otherwise it tries by a CAS to replace the old (smaller) value with the operation's input key. If the CAS succeeds, the operation returns. Otherwise the operation starts again from the beginning. This

```
 1: void WriteMax(int key) {
 2:    while(true) {
 3:       int local = value;                          ▷ linearization point if value ≥ key
 4:       if (local ≥ key)
 5:          return;
 6:       if (CAS(value, local, key));                ▷ linearization point if the CAS succeeds
 7:          return;
 8: } }
 9: int ReadMax() {
10:    int result = value;                            ▷ linearization point
11:    return result;
12: }
```

Figure 4.5: A help-free wait-free max register implementation

implementation is wait-free because each time the CAS fails, the shared `value` grows by at least one. Thus, a WriteMax(x) operation is guaranteed to return after a maximum of x iterations. A ReadMax operation simply reads the `value` and returns it.

Help-Freedom is proved similarly to the wait-free help-free set, using Claim 4.8.1. In the given max register implementation, the linearization point of every operation can be specified as a step in the execution of *the same* operation, and thus it is help-free. The linearization point of a WriteMax operation is always its last computation step. This is either reading the `value` variable (if the read value is greater than the input key), or the CAS (if the CAS succeeds). The linearization point of a ReadMax is reading the `value`.

4.9 A Universality of Fetch-And-Cons

A fetch-and-cons object allows a process to atomically add (con) an item to the beginning of a list and return the items following it. In this section, we show that fetch-and-cons is universal with respect to help-free wait-free linearizable objects. That is, given a help-free wait-free atomic fetch-and-cons primitive, one can implement any type in a linearizable wait-free help-free manner. Not surprisingly for a universal object, both Theorems 4.3 and 4.8 hold for fetch-and-cons and show it cannot be implemented in a help-free wait-free manner. Before demonstrating the universality of fetch-and-cons, we shortly discuss the strength of different primitives when it comes to overcoming indistinguishability problems.

Consider two processes, p_1 and p_2, at a certain point in an execution. Consider only their immediate next computation step. With this regard, there are five possible states: 1) neither have yet taken its next step, 2) p_1 has taken its next step and p_2 has not, 3) p_2 has taken its next step and p_1 has not, 4) p_1 has taken its next step, and afterwards p_2 has taken its next step, and 5) p_2 has taken its next step, and afterwards p_1 has taken its next step. Different primitives can be measured by their ability to support distinguishability between each of these five possibilities. Perfect

distinguishability allows each process in the system to know exactly which one of the five scenarios occurred.

Using such a metric, we can state that a system supporting only READ and WRITE is weaker than a system that also supports CAS. When both p_1 and p_2 are attempting a CAS at the same memory location, it is possible for every process in the system to distinguish between (4) and (5), while also distinguishing between (3) and (4). This is impossible when using only READ and WRITE. Still, a CAS is not perfect: for example, it is still impossible to distinguish between (2), (3) and (4) at the same time.

FETCH&ADD adds more strength to the system. When both p_1 and p_2 execute FECTH&ADD, in which they add different values to the same location, it is possible for every process in the system to distinguish between (1), (2), (3), and (4). In fact, FETCH&ADD is almost perfect: its only weakness is that it does not allow processes other than p_1 and p_2 to distinguish between (4) and (5). By contrast, fetch-and-cons is perfect: it allows every process in the system to distinguish between all five possibilities. Intuitively, this is the source of its strength.

To show that fetch-and-cons is indeed universal, we use a known wait-free reduction from any sequential object to fetch-and-cons, described in detail in [Her88]. We claim that the reduction is help-free. In essence, each process executes every operation in two parts. First, the process calls fetch-and-cons to add the description of the operation (such as ENQUEUE(2)) to the head of the list, and gets all the operations that preceded it. This fetch-and-cons is the linearization point of the operation.

Second, the process computes the results of its operation by examining all the operations from the beginning of the execution, and thus determining the "state" prior to its own operation and the appropriate result. Note that since every operation is linearized in its own fetch-and-cons step, then this reduction is help-free by Claim 4.8.1.

4.10 Discussion

This chapter studies the fundamental notion of help for wait-free concurrent algorithms. It formalizes the notion, and presents conditions under which concurrent data structures must use help to obtain wait-freedom.

We view our contribution as a lower-bound type of result, which sheds light on a key element that implementations of certain object types must contain. As such, we hope it will have a significant impact on both research and design of concurrent data structures. First, we believe it can lead to modularity in designs of implementations that are shown to require a helping mechanism in order to be wait-free, by allowing to pinpoint the place where help occurs.

Second, we ask whether our definition of help can be improved in any sense, and expect this to be an important line of further research. We think that our proposed definition is a good one, but there exist other possible definitions as well. An open question is how various formalizations of this notion relate to each other. Another

important open problem is to find a definition for the other notion of help, as we distinguish in the introduction. Such a definition should capture the mechanisms that allow a process to set the ground for its own operation by possibly assisting another operation, for the sole purpose of completing its own operation. In this chapter we do not refer to the latter as help, as captured by our definition.

An additional open problem is the further characterizations of families of data structures that require help to obtain wait-freedom. For example, we conjecture that *perturbable objects* [JTT00] cannot have wait-free help-free implementations when using only READ and WRITE primitives, but the proof would need to substantially extend our arguments for the max register type.

Chapter 5

Lock-Free Data-Structure Iterators

5.1 Introduction

Concurrent data structures are often used with large concurrent software. An *iterator* that traverses the data structure items is a highly desirable interface that often exists for sequential data structures but is missing from (almost all) concurrent data-structure implementations. In this chapter we introduce a technique for adding a linearizable wait-free iterator to a wait-free or a lock-free data structure that implements a set, given that their implementations fulfill some necessary conditions. We use this technique to implement an iterator for the lock-free and wait-free linked-lists presented in Chapter 2, and for the lock-free skip-list.

As discussed in Section 1.4, in this chapter we start from the snapshot algorithm of Jayanti [Jay05], extend it to support READ operations, and convert it to be used for data structures that implement the set abstract data type. Once a snapshot is available, we easily obtain an iterator.

Our iterator is quite efficient, and imposes an overhead of roughly 15% on the INSERT, DELETE, and CONTAINS operations when iterators are active concurrently, and roughly 5% otherwise. When compared to the CTrie iterator of [PBBO12], which is the only other available lock-free data structure that offers a linearizable iterator, our iterator demonstrates lower overhead on modifications and read operations, whereas the iteration of the data structure is faster with the CTrie iterator.

This chapter is organized as follows. Section 5.2 discusses the exact conditions a data structure must meet in order for our iteration technique to be applicable. Section 5.3 recaptures the single scanner snapshot of Jayanti. Section 5.4 discusses the differences and difficulties between a single scanner snapshot and an iterator for multiple iterating threads. Together, Sections 5.3 and 5.4 offer a good overview for our algorithm. Section 5.5 gives the details of our wait-free iterator. Section 5.6 discusses the implementation of a snap-collector, which is a major building block used in our iterator implementation and

discussed as a black-box in Section 5.5. Section 5.7 gives a detailed proof of correctness for the iterator presented in this chapter. We give the performance measurement results in Section 5.8, and conclude this chapter in Section 5.9.

5.2 Goals and Limitations

Our technique aims at extending data structures that implement the set ADT to support taking a snapshot as well. Given an atomic snapshot, iterating the data structure becomes trivial, thus throughout this work we will focus on the problem of obtaining a snapshot. The set ADT consists of three operations:

- INSERT. An INSERT operation receives an input key. If the input key is not already in the set, the operation adds the key to the set and returns true. Otherwise, the set remains unchanged and false is returned.
- DELETE. A DELETE operation receives an input key. If the input key is in the set, the operation removes the key from the set and returns true. Otherwise, the set remains unchanged and false is returned.
- CONTAINS. A CONTAINS operation receives an input key. If the input key is in the set, the operation returns true. Otherwise, it returns false.

Our technique is applicable for set data structures that uphold certain conditions. These conditions are met by many data structures that implement sets, but not by all. First, we require that each key is stored in a different node. Second, a node's key should never change until the node is reclaimed. Third, we require it is possible to traverse the data structure's nodes without missing any node that has not been inserted or deleted during the traversal. This last condition may be foiled by some tree implementations for which rotations that rebalance the tree do not support this requirement. To keep our technique wait-free, we also require that traversing the data structure's nodes can be done in a wait-free manner. If traversing the nodes is lock-free but not wait-free, than our technique will yield a lock-free snapshot.

Last, we require a particular *two-steps deletion* process. Two steps deletion is a technique commonly used in lock-free data structures to ensure that a node's outgoing pointers will not be edited during or after the node's removal. It was first introduced by Harris [Har01]. In a two steps deletion, a node is removed from the data structure in two steps. In the first step the node is marked as logically deleted, and this is the linearization point of the deletion. Starting from here, the node's key is no longer in the set. In the second step, the node is physically disconnected from the data structure.

Given the limitations, our technique particularly suits linked-list and skiplists, and we focus on these data structures. For these data structures, there is currently no alternative option for implementing a lock-free snapshot (or an iteration). Considering performance, our first priority is to inflict minimal loss on the INSERT, DELETE and

CONTAINS operations. Naturally, the performance of a SNAPSHOT operation is also important, but we are ready to slightly compromise that in order to help reduce the overhead of the main three operations.

Another data structure for which a concurrent iterator exists is the CTrie [PBBO12]. The approach taken in the CTrie design is different both in performance considerations and in the limitations it induces on the data structure. The CTrie introduces a special *i-node* connector between each two nodes of the tree. That is, a parent points to an i-node, and the i-node points to the parent's child. Such a technique is suitable for data structures with short paths, such as trees, but is ill-suited for a data structure such as a linked-list, where duplicating the time a traversal takes is problematic in practice. Furthermore, the snapshot mechanism of the CTrie relies on the fact that each node has a single predecessor. It is unclear whether this technique can be modified to support data structures such as skiplists, that do not uphold that limitation. Finally, the CTrie allows excellent performance for taking a snapshot, but taking many snapshots severely hinders the performance of the three other operations.

The *dictionary* ADT is a natural extension of the set ADT, which associates a value with each key. The transition from the set ADT to the dictionary ADT is very simple. In fact, the linked-list and skiplist we extended with our snapshot mechanism, as well as the CTrie, support the wider dictionary ADT. For simplicity, we keep the discussion in this chapter limited to the set ADT. However, note that the actual implementations and measurements are done on data structures that support the dictionary ADT.

5.3 Jayanti's Single Scanner Snapshot

Let us now review Jayanti's snapshot algorithm [Jay05] whose basic idea serves the (more complicated) construction in this chapter. This basic algorithm is limited in the sense that each thread has an atomic read/write register associated with it. (this variant is sometimes referred to as a single-writer snapshot, in contrast to a snapshot object that allows any thread to write to any of the shared registers.) Also, it is a single scanner algorithm, meaning that it assumes only one single scanner acting at any point in time, possibly concurrently with many updaters. In [Jay05], Jayanti extends this basic algorithm into more evolved versions of snapshot objects that support multiple writers and scanners. But it does not deal with the issue of a READ operation, which imposes the greatest difficulty for us. In this section we review the basic algorithm, and later present a data structure snapshot algorithm that implements a READ operation (as well as eliminating the single-writer and single-scanner limitations), and combines it with the INSERT, DELETE, and CONTAINS operations.

Jayanti's snapshot object supports two operations: UPDATE and SCAN. An UPDATE operation modifies the value of the specific register associated with the updater, and a SCAN operation returns an atomic view (snapshot) of all the registers. Jayanti uses three arrays of read/write registers, A[n], B[n], C[n], initialized to null, and an additional

```
A[n], B[n], C[n]:  arrays of read/write
registers initiated to Null
ongoingScan: a bit initiated to 0.

Update(tid, newValue)
    1.  A[tid] = newValue
    2.  If (ongoingScan==1)
    3.      B[tid]=newValue
```

```
Scan()
    1.  ongoingScan = 1
    2.  For i in 1..n
    3.      B[i] = NULL
    4.  For i in 1..n
    5.      C[i] = A[i]
    6.  ongoingScan = 0
    7.  For i in 1..n
    8.      If (B[i] != NULL)
    9.          C[i] = B[i]
    10. Array C now holds the Snapshot
```

Figure 5.1: Jayanti's single scanner snapshot algorithm

bit, which we denote *ongoingScan*. This field is initialized to false. Array A may be intuitively considered the main array with all the registers. Array B is used by threads that write during a scan to report the new values they wrote. Array C is never read in the algorithm; it is used to store the snapshot the scanner collects. The algorithm is depicted in figure 5.1. When thread number k executes an UPDATE, it acts as follows. First, it writes the new value to A[k]. Second, it reads the ongoingScan boolean. If it is set to false, then the thread simply exits. If it is set to true, then the thread *reports* the new value by also writing it to B[k], and then it exits.

When the scanner wants to collect a snapshot, it first sets the ongoingScan bit to true. Then, in the second step, it sets the value of each register in the array B to null (in order to avoid leftovers from previous snapshots). Third, it reads the A registers one by one and copies them into the C array. Fourth, it sets the ongoingScan to false. This (fourth) step is the linearization point for the SCAN. At this point array C might not hold an atomic snapshot yet, since the scanner might have missed some updates that happened concurrently with the reading of the A registers. To rectify this, the scanner uses the reports in array B; thus in the final step, it reads the B registers one by one, and copies any non-null value into C. After that, C holds a proper snapshot.

The linearizability correctness argument is relatively simple [Jay05]. The main point is that any UPDATE which completes before the linearization point of the SCAN (line 6) is reflected in the snapshot (either it was read in lines 4-5 or will be read in lines 7-9), while any UPDATE that begins after the linearization point of the SCAN is not reflected in the snapshot. The remaining updates are concurrent with each other and with the scan since they were all active during the linearization point of the SCAN (line 6). This gives full flexibility to reorder them to comply with the semantics of the snapshot object ADT. Note that there is no specific code line that can mark the linearization point of an UPDATE operation.

5.4 From Single Scanner Snapshot to Multiple Data Structure Snapshots

Our goal is to add a SNAPSHOT operation to existing lock-free or wait-free data structures. We are interested in data structures that support the set ADT. Similarly to the scanner object, to take a snapshot, the entire data structure is scanned first, and reports are used to adjust the snapshot afterwards. Here too, a snapshot is linearized after the first scan and before going over the reports. Threads executing the INSERT, the DELETE, or the CONTAINS operations cooperate with a scanner in the following way.

- Execute the operation as usual.
- Check whether there exists a parallel ongoing scan that has not yet been linearized.
- If the check is answered positively, report the operation.

Two major complications that do not arise with a single scanner snapshot algorithm arise here: the need to report operations of other threads, and the need to support multiple concurrent snapshots.

5.4.1 Reporting the Operations of Other Threads

The need to report operations of other threads stems from dependency of operations. Suppose, for example, that two INSERT operations of the same key (not currently exist in the data structure) are executed concurrently, and are not concurrent with any DELETE. One of these operations should succeed and the other should fail. This creates an implicit order between the two INSERTs. The successful INSERT must be linearized before the unsuccessful INSERT. In particular, we cannot let the second operation return before the linearization of the snapshot and still allow the first operation not to be visible in the snapshot. Therefore, we do not have the complete flexibility of linearizing operations according to the time they were reported, as in Section 5.3.

To solve this problem, we add a mechanism that allows threads, when necessary, to report operations executed by other threads. Specifically, in this case, the failing INSERT operation will first report the previous successful INSERT, and only then exit. This will ensure that if the second (failing) INSERT operation returns before the linearization of the snapshot, then the first INSERT operation will be visible in the snapshot. In general, threads need to report operations of other threads if: (1) the semantics of the ADT requires that the operation of the other thread be linearized before their own operation, and (2) there is a danger that the snapshot will not reflect the operation of the other thread.

5.4.2 Supporting Multiple Snapshots

In the basic snapshot algorithm described in Section 5.3, only a single simultaneous scanning is allowed. To construct a useful itertor, we need to support multiple simul-

taneous SNAPSHOT operations. A similar extension was also presented in [Jay05], but our extension is more complicated because the construction in [Jay05] does not need to even support a READ, whereas we support INSERT, DELETE, and CONTAINS.

In order to support multiple snapshots, we cannot use the same memory for all of them. Instead, the data structure will hold a pointer to a special object denoted the *snap-collector*. The snap-collector object holds the analogue of both arrays B and C in the single scanner snapshot, meaning it will hold the "copied" data structure, and the reports required to "fix" it. The snap-collector will also hold a bit equivalent to `ongoingScan`, indicating whether the SNAPSHOT has already been linearized.

5.5 The Data Structure Snapshot Algorithm

The pseudo-code for adding a SNAPSHOT to an applicable underlying data structure is depicted in Figure 5.2. This algorithm applies as is to the wait-free linked-list (Chapter 2, the lock-free linked-list [Mic02], and the lock-free skiplist [HS08].

To optimize performance, we allow several concurrent threads that want to iterate to cooperate in constructing the same snapshot. For this purpose, these threads need to communicate with each other. Other threads, which might execute other concurrent operations, also need to communicate with the snapshot threads and forward to them reports regarding operations which the snapshot threads might have missed. This communication will be coordinated using a snap-collector object.

The snap-collector object is thus a crucial building block of the snapshot algorithm. During the presentation of the snapshot algorithm, we will gradually present the interface the snap-collector should support. The implementation of the snap-collector object that supports the required interface is deferred to Section 5.6. All snap-collector operations are implemented in a wait-free manner so that it can work with wait-free and lock-free snapshot algorithms.

To integrate a snapshot support, the data structure holds a pointer, denoted *PSC*, to a snap-collector object. The PSC is initialized during the initialization of the structure to point to a dummy snap-collector object. When a thread begins to take a (new) snapshot of the data structure, it allocates and initializes a new snap-collector object. Then, it attempts to change the PSC to point to this object using a compare-and-swap (CAS) operation. Concurrent scanners may use the same snap-collector object, if they arrive early enough to be certain they have not missed the linearization point of the SNAPSHOT.

5.5.1 The Reporting Mechanism

A thread executing INSERT, DELETE or CONTAINS operation might need to report its operation to maintain linearizability, if a snapshot is being concurrently taken. It firsts executes the operation as usual. Then it checks the snap-collector object, using the

later's `IsActive` method, to see whether a concurrent snapshot is in progress. If so, and in case forwarding a report is needed, it will use the snap-collector `Report` method. The initial dummy snap-collector object should always return false when the IsActive method is invoked.

There are two types of report. An *insert-report* is used to report a node has been inserted into the data structure, and a *delete-report* used to report a removal. A report consists of a pointer to a node, and an indication which type of report it is. Using a pointer to a node, instead of a copy of it, is essential for correctness (and is also space efficient). It allows a scanning thread to tell the difference between a relevant delete-report to a node it observed, and a belated delete-report to a node with the same key which was removed long ago.

Reporting a Delete Operation

It would have been both simple and elegant to allow a thread to completely execute its operation, and only then make a report if necessary. Such is the case in all of Jayanti's snapshot algorithms presented in [Jay05]. Unfortunately, in the case of a DELETE operation, such a complete separation between the "normal" operation and the submission of the report is impossible because of operation dependence. The following example illustrates this point.

Suppose a thread S starts taking a snapshot while a certain key x is in the data structure. Now, another thread T_1 starts the operation DELETE(x) and a third thread T_2 concurrently starts the operation CONTAINS(x). Suppose T_1 completes the operation and removes x, but the scanner missed this development because it already traversed x, and suppose that now T_1 is stalled and does not get to reporting the deletion. Now T_2 sees that there is no x in the data structure, and is about to return false and complete the CONTAINS(x) operation. Note that the CONTAINS operation must linearize before it completes, whereas the snapshot has not yet linearized, so the snapshot must reflect the fact that x is not in the data structure anymore. Therefore, to make the algorithm linearizable, we must let T_2 first report the deletion of x (this is similarly to the scenario discussed in Section 5.4.1.). However, it cannot do so: to report that a node has been deleted, a pointer to that node is required, but such a pointer is no longer available, since x has been removed.

We solve this problem by exploiting the delete mechanism of the linked-list and skiplist (and other lock-free data structures as well). As first suggested by Harris in [Har01], a node is deleted in two steps. First, the node is marked. A marked node is physically in the data structure, and still enables traversing threads to use it in order to traverse the list, but it is considered *logically deleted*. Second, the node is physically removed from the list. The linearization of the DELETE operation is in the first step. We will exploit this mechanism by reporting the deletion between these two steps (lines 11-13 in Figure 5.2).

Any thread that is about to physically remove a marked node will first report a deletion of that node (given a snapshot is concurrently being taken). This way, the report is appropriately executed *after* the linearization of the DELETE operation. Yet, if a node is no longer physically in the data structure, it is guaranteed to have been reported as deleted (if necessary). Turning back to the previous scenario, if T_2 sees the marked node of x, it will be able to report it. If it doesn't, then it can safely return. The deletion of x has already been reported.

Note that reports of DELETE operations are thus created not only inside a DELETE operation, but also in any occasion where the original algorithm physically removes a node. For example, Harris's algorithm may physically remove a node inside the search method called by the INSERT operation as well. Such removals are also preceded with a report (e.g., line 17 in Figure 5.2).

Reporting an Insert Operation

After inserting a node, the thread that inserted it will report it. To deal with operation dependence, a CONTAINS method that finds a node will report it as inserted before returning, to make sure it did not return prior to the linearization of the corresponding insertion. Furthermore, an INSERT operation that fails because there is already a node N with the same key in the data structure will also report the insertion of node N before returning, for similar reasons.

However, there is one additional potential problem: an unnecessary report might cause the SNAPSHOT to see a node that has already been deleted. Consider the following scenario. Thread T_1 starts INSERT(3). It successfully inserts the node, but gets stalled before checking whether it should report it (between lines 22 and 23). Now thread T_2 starts a DELETE(3) operation. It marks the node, checks to see whether there is an ongoing SNAPSHOT, and since there isn't, continues without reporting and physically removes the node. Now thread S starts SNAPSHOT, announces it is scanning the structure, and starts scanning it. T_1 regains control, checks to see whether a report is necessary, and reports the insertion of the 3. The report is of course unnecessary, since the node was inserted before S started scanning the structure, but T_1 does not know that. T_2 did see in time that no report is necessary, and that is why it did not report the deletion. The trouble is that since the deletion is not reported, reporting the insertion is not only unnecessary, but also harmful: it causes S to see the reported node even though it was removed by T_2 before the SNAPSHOT has begun, contradicting linearizability.

We solve this problem by exploiting again the fact that a node is marked prior to its deletion. An insertion will be reported in the following manner (lines 31-35).

- Read PSC, and record a private pointer to the snap-collector object, SC.
- Check whether there is an ongoing snapshot, by calling SC.IsActive().

- If not, return. If there is, check whether the node you are about to report is marked.

- If it is, return without reporting. If it is not marked, then report it.

The above scheme solves the problem of harmfully reporting an insertion. If the node was unmarked after the relevant SNAPSHOT has already started, then a later delete operation that still takes place before the linearization of the SNAPSHOT will see that it must report the node as deleted. There is, however, no danger of omitting a necessary report; if a node has been deleted, there is no need to report its insertion. If the delete occurred before the linearization of the SNAPSHOT, then the snapshot does not include the node. If the delete occurred after the linearization of the SNAPSHOT, then the insert execution must be still ongoing after the linearization of the SNAPSHOT as well (since it had a chance to see the node is marked), and therefore it is possible to set the linearization of the insertion after the SNAPSHOT as well.

5.5.2 Performing a Data Structure Snapshot

A thread that desires to perform a SNAPSHOT first reads the PSC pointer and checks whether the previous SNAPSHOT has already been linearized by calling the IsActive method (line 53). If the previous SNAPSHOT has already been linearized, then it cannot use the same snapshot, and it will allocate a new snap-collector. After allocating it, it will attempt to make the global PSC pointer point to it using a CAS (line 56). Even if the CAS fails, the thread can continue by taking the new value pointed by the PSC pointer, because the linearization point of the new snap-collector is known not to have occurred before the thread started its SNAPSHOT operation. Therefore, this CAS doesn't interfere with wait-freedom, because the thread can continue even if the CAS fails.

A snapshot of the data structure is essentially the set of nodes present in it. The scanning thread scans the data structure, and uses the snap-collector to add a pointer to each node it sees along the way (lines 62-68), as long as this node is not marked as logically deleted. The scanning thread calls the `AddNode` method of the snap-collector for this purpose.

When the scanning thread finishes going over all the nodes, it is time to linearize the snapshot. It calls the `Deactivate` method in the snap-collector for this purpose (this is similar to setting ongoingScan to zero in Jayanti's algorithm). Afterwards, further calls to the IsActive method will return false. An INSERT, DELETE, or CONTAINS operation that will start after the deactivation will not report to this snap-collector object. If a new SNAPSHOT starts, it is no longer able to use this snap-collector, and so it allocates a new one.

To ensure proper linearization in the presence of multiple scanning threads, some further synchronization is required between them. A subtle implied constraint is that all threads that scan concurrently and use the same snap collector object must decide

on the same snapshot view. This is needed because they all share the same linearization point, which is the (first) time the Deactivate method of the snap-collector has been called.

To ensure the snapshot is consistent for all threads we enforce the following. First, before a thread calls the Deactivate method, it calls the `BlockFurtherNodes` (line 66). The snap-collector ensures that after a call of BlockFurtherNodes returns, further invocations of AddNode cannot install a new pointer, or have any other effect. Second, before the first scanning thread starts putting together the snapshot according to the collected nodes and reports, it blocks any further reports from being added to the snap-collector. This is achieved by invoking the `BlockFurtherReports` method (line 69). From this point on, the snap-collector is in a read-only mode.

Next, the scanning thread assembles the snapshot from the nodes and reports stored in the snap-collector. It reads them using the `ReadPointers` and `ReadReports` methods. A node is in the snapshot iff: 1) it is among the nodes added to the snap-collector OR there is a report indicating its insertion AND 2) there is no report indicating its deletion.

Calculating the snapshot according to these rules can be done efficiently if the nodes and reports in the snap-collector are sorted first. As explained in Section 5.6.1, the snap-collector is optimized so that it holds the nodes sorted throughout the execution, and thus sorting them requires no additional cost. The reports, however, still need to be sorted. Another part of the cost of taking a snapshot is that `BlockFurtherReports` is called for every thread in the system. Thus, the overall complexity of a snapshot is O(n + r*log(r) + t), where n is the number of different nodes viewed during the node-traversal phase, r is the number of reports, and t is the number of threads. This complexity analysis assumes that while traversing the nodes, finding the successor of each node (line 68) is done in O(1) steps. If this takes longer, then the complexity could potentially be worse.

5.5.3 Memory Reclamation

Throughout the algorithm description, the existence of an automatic garbage collection is assumed. In this subsection, the adaptation of the technique into environments without GC is briefly discussed. General techniques for reclaiming deleted nodes in lock-free data structures are discussed in literature, most notably pass the buck [HLMM05], hazard pointers, [Mic04] and the anchor technique [BKP13].

The approach used by all of these techniques is as follows. When a thread physically deletes a node, it becomes the owner of that node, and is responsible for freeing its memory. However, for safety, the memory must not be reclaimed while another thread might still have a pointer to the deleted node. For this purpose, while threads are traversing the data structure, they dynamically announce (in a designated location) which nodes are potentially being accessed by them. In the hazard pointers and pass the

```
Shared data: pointer PSC to snap-collector.
1.  Initialize()
2.    initialize the underlying data structure
        as usual.
3.    PSC = (address of) NewSnapCollector()
4.    PSC->Deactivate()
5.
6.  Delete(int key)
7.    search for a node with required key,
      but before removing  a marked node,
      first call ReportDelete()
8.    if no node  with the key  found
9.      return false
10.   else   // found a victim node with the key
11.     mark the victim node
12.     ReportDelete(pointer to victim)
13.     physically remove the victim node
14.     return true
15.
16. Insert(Node n)
17.   search for the place to insert the node
        n as usual, but before removing
        a marked node, first call ReportDelete()
18.   if n.key is already present in the data
          structure on a different node h
19.     ReportInsert(pointer to h)
20.     return false
21.   else
22.     Insert n into the data structure
23.     ReportInsert(pointer to n)
24.     return true
25.
26. ReportDelete(Node *victim)
27.   SC = (dereference) PSC
28.   if (SC.IsActive())
29.     SC.Report(victim, DELETED, tid)
30.
31. ReportInsert(Node* newNode)
32.   SC = (dereference) PSC
33.   if (SC.IsActive())
34.     if (newNode is not marked)
35.       Report(newNode, INSERTED, tid)
36.
```

```
37. Contains(int key)
38.   search for a node n with the key
39.   if not found  then return false
41.   else if n is marked
42.     ReportDelete(pointer to n)
43.     return false
44.   else
45.       ReportInsert(pointer to n)
          return true

46. Snapshot()
47.   SC = AcquireSnapCollector()
48.   CollectSnapshot(SC)
49.   ReconstructUsingReports(SC)
50.
51. AcquireSnapCollector()
52.   SC = (dereference) PSC
53.   if (SC.IsActive())
54.     return SC
55.   newSC = NewSnapCollector()
56.   CAS(PSC, (reference of)SC, (ref) newSC)
57.   newSC = (dereference) PSC
58.   return newSC
59.
60. CollectSnapshot(SC)
61.   Node curr = head of structure
62.   while (SC.IsActive())
63.     if (curr is not marked)
64.       SC.AddNode(pointer to curr)
65.     if (curr.next is null) // curr is the last
66.       SC.BlockFurtherNodes()
67.       SC.Deactivate()
68.     curr = curr.next
69.   for i = 1… max_tid
70.     SC.BlockFurtherReports(i)
71.
72. ReconstructUsingReports(SC)
73.   nodes = SC.ReadPointers()
74.   reports = SC.ReadReports()
75.   a node N belong to the snapshot iff:
76.    ((N has a reference in nodes
          OR  N has an INSERTED report)
         AND
         (N does not have a DELETED report)
```

Figure 5.2: Adding a Snapshot Support to an Underlying Set Data Structure

buck techniques, the specific nodes that might be accessed are reported explicitly. In the anchor technique, a more general information about the region of the data structure currently being accessed is written.

Whichever technique is used, the owner of the deletion operation uses the announcements to determine when a node can be safely freed. All of these techniques, with slight modifications, are applicable when applying our snapshot algorithm as well. When an owner of a node wants to physically free its memory, it should first establish that no other thread might currently access the node (as is already described in [HLMM05], [Mic04] and [BKP13]), but then, it should also check that the node cannot be present in any snapshot that is still accessible by a thread.

To achieve the latter, each snap-collector object is associated with a counter. When a thread allocates a new snap-collector object, and before attempting a CAS instruction to atomically make this snap-collector object accessible via the PSC, it sets the snap-collector's counter to a value higher by one than the snap-collector currently pointed by the PSC. In addition, there will be another array with a slot for each thread, where each thread declares the number of the oldest snap-collector object the thread has access to.

When a node is deleted, the owner of that node checks the number of the snap-collector currently pointed by the PSC. Once all threads advance past that number, the node cannot be a part of any active snapshot, and it can safely be freed. Some care is needed regarding the reclamation of the memory of the snap-collector object itself. Each thread should use a hazard pointer before it can safely access the snap-collector via the PSC. The thread that originally allocated a snap-collector can safely free it once 1) all threads declare in the designated array that they are no longer using this snap-collector, and 2) no thread has a hazard pointer pointing to that snap-collector.

The additional hazard pointer for the PSC is required to prevent races in which the thread that deallocates the snap-collector missed the fact that it is still needed. While the implementation is tedious, it does not bring up challenging difficulties beyond those already discussed in [HLMM05], [Mic04] and [BKP13].

5.6 The Snap-Collector Object

One can think of the snap-collector object as holding a list of node pointers and a list of reports. The term *install* refers to the act of adding something to these lists. Thus, the snap-collector enables the scanning threads to install pointers, and the modifying threads to install reports. It supports concurrent operations, it is linearizable, and it must be wait-free since it is designed as a building block for wait-free and lock-free algorithms.

The definition of the snap-collector object follows. To relate the new algorithm to the basic one, we also mention for each method (*in italics*), its analogue in the single scanner snapshot.

- AddNode(Node* node). *Analogue to copying a register into array C.* If the BlockFurtherNodes() method (see below) has previously been executed this method does nothing. Otherwise it installs a pointer to the given node. Has no return value.

- Report(Report* report, int tid). *Analogue to reporting a new value in array B.* If the BlockFurtherReports(int tid) (see below) has previously been executed with the same tid, this method does nothing. Otherwise it installs the given report. Has no return value.

- IsActive(). *Analogue to reading the ongoingScan bit.* Returns true if the Deactivate() method has not yet been executed, and false otherwise.

- BlockFurtherNodes(). *No analogue. Required to synchronize between multiple scanners.* After this method is executed at least once, any further calls to AddNode will do nothing. Has no return value.

- Deactivate(). *Analogue to setting ongoingScan to false.* After this method is executed at least once, any call to IsActive returns false, whereas before this method is executed for the first time, IsActive returns true. Has no return value.

- BlockFurtherReports(int tid). *No analogue. Required to synchronize between multiple scanners.* After this method is executed, any further calls to Report with the same tid will do nothing. Has no return value.

- ReadPointers(). *No analogue.* Returns a list of all the pointers installed in the snap-collector object (via the AddNode method) before BlockFurtherNodes was executed.

- ReadReports(). *No analogue.* Returns a list of all the reports installed in the snap-collector object (via the Report method).

Some methods of the snap-collector receive an id as a parameter, which we refer to as tid (Thread Identifier). IDs are supposed to be known in advance, and thus this implicitly assumes prior knowledge of the number of active threads in the system. This requirement is not really a mandatory assumption in our algorithm. It is possible to run the algorithm and allow several threads (or even all of them) to share the same id. The only change required is to ensure the snap-collector implementation would work correctly when the REPORT method is called concurrently from several threads which use the same id. This could easily be done by using a wait-free data structure to hold the reports (for example, the wait-free queue of [KP11] would suit the task well), but it would cause a slight degradation in performance, comparing to the implementation suggested in Subsection 5.6.1.

```
Shared data:
ReportItem[]  ReportHeads
ReportItem[]  ReportTails
NodeWrapperPointer  NodesHead
Bit   Active

1.  Initialize()
2.    Active = true
3.    NodesHead = new sentinel
4.    for i = 1... max_tid
5.       ReportHeads[i] = new sentinel
6.       ReportTails[i] = ReportHeads[i]

7.  AddNode (Node n)
8.      Last =  NodesHead
9.      If (Last.Node.key >= key)
10.       Return Last.Node
11.     NewWrapper = NewNodeWrapper()
12.     NewWrapper.Node = n
13.     NewWrapper.Next = Last
13.     if (CAS(NodeHead,  Last, NewWrapper)
14.       return n
15.     else
16.       return NodesHead.Node

17. Report (Report r, int tid)
18.   ReportItem tail = ReportTails[tid]
19.   ReportItem nItem = new ReportItem()
20.   nItem.report = r
21.   If (CAS(tail.next,null, nItem))
22.     ReportTails[tid] = nItem
```

```
23.  IsActive()
24.    return Active

25.  BlockFurtherNodes()
26.    NodeWrapper blk=NewNodeWrapper()
27.    blk.Node = new sentinel (MAX_VALUE)
28.    blk.Next = Last
29.    NodesHead= blk

30.  Deactivate()
31.    Active = false

32.  BlockFurtherReports(int tid)
33.    ReportItem tail = ReportTails[tid]
34.    CAS(tail.next, null, new sentinel)

35  ReadPointers()
36.    Return a linked-list starting from
       NodesHead

37.  ReadReports()
38.    Return a concatenation of
       all the lists of ReportItems
```

Figure 5.3: An Implementation of the Snap-Collector

5.6.1 The Snap-Collector Implementation

The implementation of the snap-collector object is orthogonal to the SNAPSHOT algorithm, but different implementations can affect its performance dramatically. This section briefly explains the particulars of the implementation used in this work. The pseudo code for the snap-collector is given in Figure 5.3.

Our proposed implementation of the snap-collector slightly changes the ADT semantics of the AddNode method. This is an optimization, and the motivation for this is given in the paragraph about the AddNode method.

The implementation of the snap-collector object maintains a separate linked-list of reports for each thread. Every such linked list holds *ReportItems*, each of which is a report and a pointer to the next ReportItem. The snap-collector also maintains a single linked-list of pointers to the nodes of the data structure. This is a linked list of *NodeWrapperPointers*, each of which is a pointer to a node and a pointer to the next NodeWrapperPointer. Finally, the snap-collector holds one bit field indicating whether it is currently active (not yet deactivated).

IsActive, Deactivate. The IsActive method is implemented simply by reading a bit. The Deactivate method simply writes false to this bit.

AddReport. When a thread needs to add a report using the AddReport method, it adds it to the end of its local linked-list dedicated to this thread's reports. Due to the locality of this list its implementation is fast, which is important since it is used also by threads that are not attempting to take a snapshot of the data structure. Thus, it facilitates low overhead for threads that only update the data structure.

Although no other thread may add a report to the thread local linked- list, a report is still added via a CAS, and not a simple write. This is to allow the scanning threads to block further reports in the BlockFurtherReports method. However, when a thread adds a report, it does not need to check whether the CAS succeeded. Each thread might only fail once in adding a report for every new SNAPSHOT. After failing such a CAS, it will hold that the IsActive method will already return false for this snapshot and therefore the thread will not even try to add another report.

BlockFurtherReports. This method goes to the local linked-list of the thread whose future reports are to be blocked, and attempts by a CAS to add a special dummy report at the end of it to block further addition of reports. This method should only be invoked after the execution of the Deactivate method is completed. The success of this CAS need not be checked. If the CAS succeeds, no further reports can be added to this list, because a thread will never add a report after a dummy. If the CAS fails, then either another scanning thread has added a dummy, or a report has just been added. The first case guarantees blocking further repots, but even in the latter case, no further reports can now be added to this list, because the thread that just added this report will see that the snap-collector is inactive and will not attempt to add another report.

AddNode. One possible approach to implement AddNode is to use a lock-free stack. To install a pointer to a node, a thread reads the head pointer. It attempts by a CAS to add its node after the last node. If it fails, it retries.

Used naively, this approach is not wait-free as the thread may repeatedly fail to add its node and make progress. We use a simple optimization that slightly alters the semantics of the AddNode method. To this end, we modify AddNode to expect nodes to be added to the snapshot view in an ascending order of keys. The AddNode method will (intentionally) fail to add any node whose key is smaller than or equal to the key of the last node added to the snap-collector. When such a failure happens, AddNode returns a pointer to the data structure node that was last added to the snap-collector view of the snapshot. This way, a scanning thread that joins in after a lot of pointers have already been installed, simply jumps to the current location. This also reduces the number of pointers in the snap-collector object to reflect only the view of a single

sequential traverse, avoiding unnecessary duplications. But most importantly, it allows wait-freedom.

Similarly to the naive approach of using a lock-free stack, the snap-collector object holds the head pointer to the stack. To push a pointer to a node that holds the key k, a thread reads the head pointer. If the head node holds a key greater than or equal to k, it doesn't add the node and simply returns the head node. If the CAS to change the head pointer fails, then this means that there is another thread that has just inserted a new node to the snapshot view. In this case, this new node is either the same node we are trying to add or a larger one. Thus, the thread can safely returns the new head, again, without adding the new node.

This optimization serves three purposes: it allows new scanning threads to jumps to the current location; it makes the AddNode method fast and wait-free; and it keeps the list of pointers to nodes sorted by their keys, which then allows a simple iteration over the keys in the snapshot. Note that the CollectSnapshot method in Figure 5.2 also needs to be modified in order to use this optimization: it must use the returned value of AddNode, and assign it to curr.

BlockFurtherNodes. This method sets the head pointer of the nodes list to point to a special dummy with a key set to the maximum value. Combined with our special implementation of AddNode, further calls to AddNode will then read the head's special maximum value and will not be able to add additional nodes.

ReadPointers, ReadReports. These methods simply return a list with the pointers / reports stored in the snap-collector. They are called only after the BlockFurtherNodes, Deactivate, and BlockFurtherReports methods have all been completed, thus the lists of pointers and reports in the snap-collector are immutable at this point.

5.6.2 Some Simple Optimizations

The implementation used for the performance measurements also includes the following two simple optimizations.

Elimination of many of the reports. An additional bit was added to each node, initialized to zero. When a thread successfully inserts a node, and after reporting it if necessary, this bit is set to 1. Future INSERT operations that fail due to this node, and future CONTAINS operations that successfully find this node, first check to see if this bit is set. If so, then they know that this node has been reported, and therefore, there is no need to report the node's insertion.

If a large portion of the operations are CONTAINS operations, as is the case in typical data structure usage, this optimization avoids a significant portion of the reports. This is because in such cases most of the reports are the result of successful CONTAINS

operations. However, note that this optimization is not always recommended, as it adds overhead to the INSERT operations even if SNAPSHOT is never actually called.

Avoidance of repeated sorting. After a single thread has finished sorting the reports, it posts a pointer to a sorted list of the reports, and saves the time it would take other threads to sort them as well, if they haven't yet started to do so.

5.7 Proof

In this section we prove that the construction presented in this chapter is linearizable. Specifically, we show that adding the iterator to Harris's linked-list is linearizable, and to make everything concrete, we assume the specific implementation of a lock free linked-list given in [HS08]. This implementation is a variation of the linked-list of Maged Michael [Mic02], who based his implementation on Harris's algorithm [Har01]. In what follows, we denote the algorithm that extends Harris's linked-list with support for the SNAPSHOT operations as presented in this chapter as the *iterable list algorithm.* Similar claims can be made for an iterable skiplist.

For simplicity, we provide the proof assuming a single scanner, that is, assuming that no two SNAPSHOT operations are executed concurrently. Again, all the claims hold for the case of multiple scanners as well, but some of the arguments and definitions need to be adjusted for this case. The necessary adjustments to the proof for the case of multiple scanners are sketched in Subsection 5.7.6.

We assume that a linearizable snap-collector is given as a basic block. We discuss the linearizability of the snap-collector separately in Subsection 5.7.7. We prove the correctness of the snapshot algorithm in Figure 5.2, which uses such a snap-collector as a building block. We use the linearizability of Harris's linked-list. We rely on characterizations of this linearizability, that are established in literature. In particular, we use the fact that the linearization point of a successful DELETE operation is the CAS that marks the node as logically deleted, and that the linearization point of a successful INSERT operation is the CAS that physically inserts the (new) node into the list.

Our task is to prove that the iterable list algorithm is linearizable. To do that, we will show that for each execution E of the iterable list algorithm, there is a total order of the operations of the iterable list, such that sequential consistency (operations' results are consistent with the total order) and real-time consistency (operations that do not overlap retain their original order) hold.

Computation Model. We use the standard shared-memory model. Each computation step is either a primitive step(an atomic read, write, or compare-and-swap of a register), possibly preceded with some internal computation, or an execution of a compound step of the (linearizable) snap-collector. An execution is a sequence of

computation steps. Each step is characterized by the thread that executed it, and the atomic primitive step or snap-collector operation.

Wait-Freedom. In our snapshot algorithm (Figure 5.2), the loop in lines 62-68 traverses the nodes of the data structure. As mentioned in Section 5.2, to maintain wait-freedom, this is needed to be possible in a wait-free manner. Otherwise, our technique is lock-free, and not wait-free. Other than that, wait-freedom (assuming wait-freedom of the underlying data structure) is trivial. The loop in lines 69-70 runs a constant number of iterations. The method in lines 72-76 runs after the snap-collector is in a read-only mode. Other methods do not include loops.

5.7.1 Overview

As a first step of the proof, we strip an execution of the iterable list algorithm and retain an execution of the underlying (Harris's) linked-list algorithm. This execution is linearizable, thus there is a total order of its operations that satisfies sequential consistency and real-time consistency. We identify this execution, and its matching total order (which we denote *base-order*) in Subsection 5.7.2.

In addition, the same subsection defines several important terms used in the proof. Most importantly, *visible* and *non-visible* operations are defined. Intuitively, an operation is visible by a specific SNAPSHOT if the operation changed the list (e.g., entered a new node into the list) and occurred early enough to influence the snapshot. For example, an DELETE operation that deleted a node from the list before the snapshot has begun is visible by that snapshot.

Other important concepts include *quiet* operations, and the *deactivation point* of a SNAPSHOT operation. Quiet operations are operations that do not change the underlying set, such as CONTAINS operations, and unsuccessful INSERTS and DELETES. The deactivation point of a SNAPSHOT operation, is when the DEACTIVATE method of the snap-collector is executed.

Using these concepts, we can present an algorithm for constructing the *whole-order* for a given execution of the iterable list. Whole-order is the total order that satisfies both sequential and real-time consistency. Whole-order is built from the base-order, inserting the SNAPSHOT operations one by one. However, during the construction of whole-order, the order of (non-snapshot) operations is also slightly adjusted. The construction of whole-order is described Subsection 5.7.3

Before proving that whole-order indeed satisfies sequential and real-time consistency, we need to establish several claims about visibility. This is done in Subsection 5.7.4. For example, we claim that visibility satisfies monotonicity, in the sense that if an operation is visible by a snapshot, then prior operations with the same key are also visible. Most importantly, we show that a snapshot indeed returns what is intuitively visible to it. That is, the result returned by a snapshot is consistent with a sequential execution

in which all the visible operations occur before the snapshot, and all the non-visible operations occur after it.

The final step is to show that whole-order satisfies sequential and real-time consistency. This is done in Subsection 5.7.5, relying on the visibility properties established in Subsection 5.7.4.

5.7.2 Definitions

This subsection identifies and defines the key concepts of our proof. Mainly, for each execution E of the iterable list, we identify an underlying execution of Harris's linked-list E_L, and the linearization of E_L, denoted $Base_E$. We classify operations to quiet (such as CONTAINS) and non-quiet (such as a DELETE that successfully removed a node from the list). We give a specific name for each operation. ($Operation_{k,j}$ is the jth operation executed with key k). Finally, we identify operations that are visible for a specific snapshot, which intuitively, are the operations that affect the snapshot result.

Let E be an execution of the iterable list algorithm. We examine E as consisting of execution steps that originate from Harris's linked-list algorithm, plus some extra steps that originate from our snapshot algorithm. The extra steps include all the steps executed by a thread inside the SNAPSHOT operations, all the steps that are operations of the snap-collector, reading of the `PSC` field (the pointer to the snap-collector), and reading the mark bit of a node inside the *ReportInsert* method (line 34 Figure 5.2). Let E_L be the subset of the computation steps of E that includes all the steps originating from Harris's linked-list algorithm, but none of the extra steps mentioned above. The following observation states that this sub-execution is simply an execution of Harris's linked-list.

Observation 5.7.1. E_L is an execution of Harris's linked-list.

Since E_L is an execution of Harris's linked-list, and Harris's linked-list is a linearizable algorithm, then there is a total order on the linked-list operations that satisfies the linearizability requirements (sequential consistency and real-time consistency). In what follows we formally define this order.

Definition 5.7.2. (Base-Order, $Base_E$.) Let E be an execution of the iterable-list algorithm. We define the base-order of E, denoted $Base_E$, to be a total order on the operations of E_L that satisfies the linearizability requirements, according to the following linearization points.

- A successful INSERT is linearized at the step that physically inserts the (new) node into the list (that is, the step that makes the node reachable from the list's head).
- A successful DELETE is linearized at the step that marks the next pointer of the node that is deleted. (that is, the step of the logical delete).

- An unsuccessful INSERT is linearized at the (last) step that reads the `next` pointer of a node with the same key as the input key of the unsuccessful INSERT
- An unsuccessful DELETE is linearized at the (last) step that reads the `next` pointer of the node with the greatest key that is still smaller than the key of the unsuccessful DELETE.
- A successful CONTAINS is linearized at the (last) step that reads the `next` pointer of the node with the same key as the input key of the CONTAINS.
- For an unsuccessful CONTAINS there are two possibilities for its linearization point. If a node with the same key as the key of the contains was read, and it was found to be logically deleted, then the linearization point of the CONTAINS is the step the logically deleted that node (which was executed by a different thread). If no node with the same key as the key of the contains was read, then the linearization point of the contains is at the (last) step that reads the `next` pointer of the node with the greatest key that is still smaller than the key of the unsuccessful CONTAINS.

Remark. Note that unsuccessful INSERT and successful CONTAINS are linearized upon reading the `next` pointer of a node with the same key, since that is where the mark is located, indicating the node is not logically deleted.

Next, we identify the operations that affect the state of the linked-list. These are operations that successfully insert or remove a node from the list. Specifically, these are INSERT and DELETE operations that return true. Other operations are *quiet* operations, as they do not affect the state of the linked-list, and the results of other operations.

Definition 5.7.3. (Quiet, Non-Quiet Operations). We call an INSERT or DELETE operations that return true *non-quiet* operations. Other operations are called *quiet* operations.

For a non-quiet operation, we also define the *operation node*.

Definition 5.7.4. (Operation Node.) For a successful INSERT operation, the operation node is the new node inserted into the list in the operation's execution. For a successful DELETE operation, the operation node is the node removed from the list. (For other operations, the operation node is undefined.)

Definition 5.7.5. (Operation Key.) For an INSERT, DELETE or CONTAINS operation, the operation key is the key given as an input parameter to the operation.

Definition 5.7.6. (Operation Owner, Scanning Thread.) The operation owner of an operation is the thread that invoked and executed the operation. For a SNAPSHOT operation, we refer to the owner as the scanning thread.

Definition 5.7.7. (Matching Operations.) An INSERT and a DELETE operations that have the same operation node are called *matching* operations.

It is useful to note that each successful DELETE operation has exactly one matching INSERT operation. This is the operation that inserted the node that is removed by the delete. In the base-order, the matching of a DELETE operation will always be the non-quiet operation of the same key that comes before it. Fix any key k, and consider the non-quiet operations with key k according to their base-order, they will be matching, such that each odd position contains an INSERT, and each successive even position has its matching DELETE. The last non-quiet operation may be an insert without a matching delete.

A key technique we use in the proof is to divide the linked-list operations into disjoint sets according to their operation key. In particular, it is important to realize that an operation's result depends only on the (non-quiet) operations with the same input key. Thus, reordering the base-order of the operations, but without changing the relative order of operations with the same key, will not violate sequential consistency. That is, each operation will still return a result consistent with the sequential execution. In Subsection 5.7.3 we indeed alter the base-order in such a way, but for now we need the following definition.

Definition 5.7.8. ($Operation_{k,j}$, $Node(Operation)_{k,j}$.) $Operation_{k,j}$ is the jth operation executed with key k in the base-order. If $Operation_{k,j}$ is non-quiet, then $Node(Operation)_{k,j}$ is its operation node.

Definition 5.7.9. (The critical point of a (non-quiet) operation.) The critical point of a non-quiet operation is the linearization point of the non-quiet operation. Namely, The critical point of a (successful) delete is the CAS that marks the operation node as logically deleted. The critical point of a (successful) insert is the CAS that physically inserts the operation node into the list. (That is, the CAS that causes the operation node to be reachable from the list head.)

Remark. A quiet operation does not have a critical point.

We now move on to discuss snapshots. To simplify the discussion, we start by assuming that snapshots occur one at a time. Namely, there are no simultaneous concurrent snapshots. In this case, we can safely refer to snapshot number i in the execution E, where i is a natural number that is smaller than or equal to the total number of SNAPSHOT operations in E.

For each snapshot, we consider four different phases: *activation*, *node-traversal*, *deactivation*, and *wrap-up*. The activation consists of acquiring a new snap-collector object, and making the `PSC` field point to it. After the activation is complete and until deactivation, any thread that reads the `PSC` and then tests to see if the snapshot is active will see that it is.

Immediately after the activation, the node-traversal starts. In this phase, the scanning thread follows the pointers of the nodes in the list starting from the head until reaching the tail. When reading a new node, the scanning thread first checks to see if it is marked as logically deleted. If it isn't, the scanning thread installs a pointer to this node in the snap-collector, before reading the `next` field and moving to the next node. We know from the properties of Harris's linked-list, that during the node-traversal, the scanning thread must see each node that belongs to the list during the entire traversal phase, and can not see a node that does not belong to the list throughout the entire traversal phase.

The deactivation consists of calling the DEACTIVATE method of the snap-collector. We assume the deactivation is atomic, and refer to the *deactivation point* as an atomic point in time. This assumption is legitimate because linearizability is compositional [HW90, HS08]. Given that the snap-collector object is linearizable, we can prove the linearization of an algorithm that uses the snap-collector, while assuming that its operations are atomic.

The wrap-up of the SNAPSHOT consists of blocking further reports, collecting the reports and constructing a snapshot. We consider all these operations as a single phase, because our reasoning does not require partitioning this phase further to discuss each part separately.

Definition 5.7.10. (The Deactivation Point of Snapshot i.) The deactivation point of Snapshot i, is the (first) point in time that the DEACTIVATE method of the corresponding snap-collector is executed. We assume this point to be an atomic step.

In what follows, we define *visible* and *non-visible* operations by Snapshot i. Both visible and non-visible operations are non-quiet operations. Intuitively, an operation is visible by snapshot i, if it influenced the snapshot, in the sense that the snapshot's result reflects that operation.

Loosely speaking, there are two scenarios that make an operation visible by a snapshot. One is that during the node-traversal phase, the scanning thread observes the list as already reflecting the result of the operation. The other is that the operation is reported (successfully, that is, before reports are blocked) into the snap-collector.

Definition 5.7.11. (Visible Operations by Snapshot i.) We say that a successful DELETE(k), which removes a node N from the list, is visible by Snapshot i, if at least one of the following holds.

1. N is marked as logically deleted before the beginning of the node-traversal phase of Snapshot i.

2. The deletion of N is (successfully) reported in the snap-collector associated with Snapshot i.

3. During the node-traversal step of Snapshot i, the scanning thread reads node N, and finds it logically deleted, and an insertion of N is not (successfully) reported in the snap-collector associated with Snapshot i.

We say that a successful INSERT(k), which inserted the node N into the list, is visible by Snapshot i, if at least one of the following holds.

1. N is (physically) inserted to the list before the beginning of the node-traversal phase of Snapshot i
2. The insertion of N is (successfully) reported to the snap-collector associated with Snapshot i.
3. During the node-traversal step of Snapshot i, the scanning thread reads node N.
4. The deletion of N is (successfully) reported in the snap-collector associated with Snapshot i.

Let us provide some intuition for why Definition 5.7.11 makes sense. If a node is logically deleted or inserted prior to the beginning of the node-traversal phase, then the scanning thread notices it during the node-traversal. Next, if an operation is reported, then the scanning thread knows about it by examining the reports. Item number 4 may seem a bit odd for defining a visible insert, but note that upon seeing a deletion report, the scanning thread can deduce not only that the node N was deleted, but also that it was inserted (beforehand). When an operation is executed concurrently with the snapshot, the scanning thread may notice it during its node-traversal phase. The scanning thread can notice an insertion by seeing the node as logically not deleted, and it can see a deletion by seeing the node as logically deleted.

One counterintuitive issue is that it is possible for the scanning thread to see a node as logically deleted during the node-traversal phase, but the deletion may still be non-visible, because insertion is reported. Consider the following chain of events. Thread T_1 starts a SNAPSHOT, and completes the activation step. Thread T_2 inserts a node N into the list, then check to see if there is an ongoing (active) snapshot. Since there is, it checks to see that the node N is not marked (i.e., the node is not logically deleted), and then it reports the insertion of N. Next, T_3 logically deletes N. It checks to see if there is an active snapshot, and is about to report the deletion, but is stalled. Then T_1 executes the node-traversal phase (seeing N as logically deleted, thus not installing a pointer to it), blocks further reports, then sees a report of N's insertion, and returns N as part of the SNAPSHOT result.

Although the scanning thread (T_1) observes N as logically deleted during the node-traversal, T_1 does not record this fact anywhere, so the report of N as being inserted causes the scanner it to 'forget' what it saw. This is not harmful to linearizability. A similar problem cannot happen if N is marked as logically deleted before the beginning of the node-traversal phase: before a thread reports the insertion of N it checks that a

SNAPSHOT is active (i.e., that a node-traversal phase has began) and *then* that N is unmarked. In this case, either the inserting thread will not yet see an active traversal (and thus will not report insertion), or it will already see the node as marked for deletion, and then it will not report the insertion as well.

Definition 5.7.12. (Non-Visible Operations by Snapshot i.) A non-visible operation is a successful INSERT or DELETE operation that is not visible by Snapshot i.

5.7.3 Constructing the Whole-Order

In what follows we present an algorithm that constructs a *whole-order* for a given execution E of the iterable linked-list algorithm. This whole-order includes all the operations in E (including SNAPSHOTS), and we will claim that it satisfies the two linearization requirements (sequential consistency and real-time consistency). We build it by starting from the base-order ($Base_E$), defined in Definition 5.7.2, and then inserting the SNAPSHOT operations one by one. For each snapshot that we insert, we will also slightly adjust the order of other operations. The purpose of these adjustments is to ensure the following.

- Each operation visible by a snapshot should come before it, and each operation non-visible by a snapshot should come after it.
- Each operation that is completed before the deactivation point of a snapshot should come before it, and each operation that is invoked after the deactivation point of a snapshot should come after it.
- for every key k, the relative order of the operations with that key should be preserved, excluding, perhaps, "internal" reordering of quiet operations that have no non-quiet operations between them.

The algorithm that generates the whole-order, hence, the *whole-order algorithm* is presented in Figure 5.4.

5.7.4 Visibility Properties

To complete the proof, we would like to show that whole-order satisfies both sequential and real-time consistency. However, before we can do that, we need to lay the ground by proving several claims regarding the visibility property. These claims form the heart of the proof.

We claim that a non-quiet operation that is completed before the deactivation point of a snapshot is visible by it (Claim 5.7.14), and that a non-quiet operation that is invoked after the deactivation point of a snapshot is non-visible by it (Claim 5.7.15). The more interesting operations (for the proof) are those that happen concurrently with the deactivation of the snapshot. We claim that visibility to Snapshot i satisfies

1: Init: whole-order = base-order
2: for i in 1... total number of snapshots do:
3: insert Snapshot i immediately before the first operation in whole-order that is non-visible by Snapshot i. (if non-such exists, place Snapshot i at the end of whole-order)
4: initialize three sets of operations, `Premature-Quiets`, `Belated-Quiets`, `Belated-Visibles` to be $\emptyset$
5: for every Op in base-order
6: if (Op is a quiet operation) and (Op is placed before Snapshot i in whole-order) and (Op is invoked after the deactivation point of Snapshot i in E) then:
7: `Premature-Quiets` = `Premature-Quiets` $\cup\{Op\}$
8: if (Op is a quiet operation) and (Op is placed after Snapshot i in whole-order) and (Op is completed before the deactivation point of Snapshot i in E) then:
9: `Belated-Quiets` = `Belated-Quiets` $\cup\{Op\}$
10: if (Op is a visible by Snapshot i) and (Op is placed after Snapshot i in whole-order) then:
11: `Belated-Visibles` = `Belated-Visibles` $\cup\{Op\}$
12: move the operations in `Premature-Quiets` to be immediately after Snapshot i in whole-order (without changing the relative order of the operations in `Premature-Quiets`).
13: move the operations in `Belated-Quiets` $\cup$ `Belated-Visibles` to be immediately before Snapshot i in whole-order (without changing the relative order of the operations in `Belated-Quiets` $\cup$ `Belated-Visibles`.

Figure 5.4: Generating The Whole-Order for an Execution

monotonicity, both in the sense that if an operation is visible by Snapshot i, then all prior non-quiet operations with the same key are also visible (Claim 5.7.16), and in the sense that if an operation is visible by Snapshot i, then it is also visible by later snapshots (Claim 5.1).

Then, we make claims that also consider the timing of quiet operations. We prove that if an operation Op is visible by Snapshot i, then all operations with the same key that are linearized before Op, including quiet operations, are invoked before the deactivation point of Snapshot i (Claim 5.2), and that if an operation Op is non-visible by Snapshot i, then all operations with the same key that are linearized after Op are completed after the deactivation point (Claim 5.7.23).

Finally, we show that Snapshot i indeed returns what is intuitively visible to it. That is, the result returned by Snapshot i is consistent with a sequential execution in which all the visible operations occur before Snapshot i, and all the non-visible operations occur after it (Lemma 5.7.24).

We sometimes say that an operation is visible (non-visible), without specifying by which snapshot, where the snapshot number is clear from the context. We sometimes omit a reference to execution E (like in the following claim), when it is clear from the context.

Claim 5.7.13. *If Op is a non-quiet operation, and its critical point occurs before the beginning of the node-traversal phase of Snapshot i then Op is visible by Snapshot i.*

Proof. This claim follows from Definition 5.7.11. If Op is an INSERT, and the physical insertion of its node occurs before the beginning of the node-traversal phase of Snapshot i, then the Op is visible by Definition 5.7.11. Similarly, if Op is a DELETE, and the logical deletion of its node occurs before the beginning of the node-traversal phase of Snapshot i, then the operation is visible by Definition 5.7.11. □

Claim 5.7.14. *If Op is a non-quiet operation that is completed before the deactivation point of Snapshot i, then Op is visible by Snapshot i.*

Proof. Let Op be a non-quiet operation, and i be an integer such that $1 \leq i \leq$ total number of snapshots. After executing the critical point of Op, the owner thread of Op reads `PSC` and checks if the snapshot is active. We consider three possibilities. Snapshot i might not yet be active at the point of the check, or it might already been deactivated, or it might be active.

If Snapshot i is not yet active, then Op is visible by Snapshot i because its critical point occurs before the beginning of the node-traversal phase of Snapshot i (Claim 5.7.13). If Snapshot i has already been deactivated, then Op is not completed before the deactivation point of it.

If Snapshot i is active at the point of the check, then the owner-thread of Op will attempt to report the operation into the snap-collector associated with Snapshot

i. If the report is successful, then Op is visible by Snapshot i (by Definition 5.7.11). Otherwise, BlockFurtherReports method must have been invoked prior to the completion of the report. The BlockFurtherReports method is only invoked after the deactivation point, and thus in this case Op is not completed before the deactivation point of Snapshot i. □

Claim 5.7.15. *If Op is a non-quiet operation that is invoked after the deactivation point of Snapshot i, then Op is non-visible by Snapshot i.*

Proof. Let Op be a non-quiet operation, and i be an integer such that $1 \leq i \leq$ total number of snapshots. Let us consider (and eliminate) all the different ways for Op to become visible. If Op is an INSERT operation, then the physical insertion does not occur before the beginning of the node-traversal phase; the scanning thread cannot see the operation-node during the node-traversal phase (since this phase ends before the physical insertion); a report of the insertion cannot be made: a thread only reports an insertion if after the physical insertion of the node the thread sees the snapshot is still active; and finally, a report of the deletion of the same node cannot be made: the node is logically deleted only after it is physically inserted, which occurs in this case after the deactivation point, and a thread only reports a deletion if after the logical deletion of the node the thread sees the snapshot as still active.

If Op is a DELETE operation, then the logical deletion does not occur before the beginning of the node-traversal phase; the scanning thread cannot see the operation-node as logically deleted during the node-traversal phase; finally, a report cannot be made: a thread only reports a deletion if after the logical deletion of the node, the thread sees the snapshot as still active. □

Claim 5.7.16. *If* $\text{Operation}_{k,j}$ *is visible by Snapshot i, then all earlier operations of the same key:* $\text{Operation}_{k,q}$*, such that* $q \leq j$*, are either visible by the same snapshot, or quiet.*

Proof. Let q be an integer such that $q < j$. (If $q = j$ the claim is trivial.) If $Operation_{k,q}$ is a *contains* or a non-successful operation, then it is quiet. If $Operation_{k,q}$ is a non-quiet operation, then we examine each of the possible causes for the jth operation on Key k $Operation_{k,j}$ to become visible to Snapshot i, and show that the qth operation must be visible to Snapshot i as well.

If $Operation_{k,j}$ is visible because its critical point (physically inserting the node, or marking it as logically deleted) occurs prior to the beginning of the node-traversal phase of Snapshot i, then the same is true for $Operation_{k,q}$. This is because physically inserting (or logically deleting) a node is the linearization point in Harris's linked-list, and the operations are ordered in the base-order according to these linearization points.

An insert $Operation_{k,j}$ can also be visible if during the node-traversal phase the scanning thread reads $Node(Operation)_{k,j}$, and a delete $Operation_{k,j}$ can be visible if

during the node-traversal phase the scanning thread reads *Node(Operation)*$_{k,j}$, and finds it marked as logically deleted. Both of these events imply that the scanning thread reads *Node(Operation)*$_{k,j}$. Now, if *Node(Operation)*$_{k,j}$ and *Node(Operation)*$_{k,q}$ are the same (i.e., *Operation*$_{k,j}$ and *Operation*$_{k,q}$ are matching operations) then the earlier *Operation*$_{k,q}$ must be a visible insert. If *Node(Operation)*$_{k,j}$ and *Node(Operation)*$_{k,q}$ are not the same, then *Node(Operation)*$_{k,q}$ must have been in the list and then removed, since *Node(Operation)*$_{k,j}$ was inserted after it.

Thus, the physical delete of *Node(Operation)*$_{k,q}$ must happen early enough for the scanning thread to view *Node(Operation)*$_{k,j}$, that is, no later than during the node-traversal phase. Either *Node(Operation)*$_{k,q}$ is marked as logically deleted prior to the node-traversal phase (making *Operation*$_{k,q}$ become visible) or that it is marked as logically deleted, and is physically removed, both during the node-traversal case.

Before physically removing a node, the thread that removes it report it as logically deleted. That is, unless there is no active snapshot. Since the node is both logically and physically deleted during the node-traversal phase, then there must be a report of its deletion. Such a report will guarantee that *Operation*$_{k,q}$ is visible by Snapshot i.

If *Operation*$_{k,j}$ is visible due to a report, similar logic holds. If *Node(Operation)*$_{k,j}$ and *Node(Operation)*$_{k,q}$ are the same one, then the same report also makes *Operation*$_{k,q}$ (which must be an INSERT) visible. Otherwise, the report of *Operation*$_{k,j}$ means that the operation occurs no later than the node-traversal phase (Because afterwards comes the deactivation point, and threads cease to report their operations to the corresponding snap-collector.) Thus, again, either *Node(Operation)*$_{k,q}$ is marked as logically deleted prior to the node-traversal phase or that it is marked as logically deleted, and is physically removed, both during the node-traversal case. □

Claim 5.7.17. *If Operation Op is visible by Snapshot i, then the critical point of Op occurs before the deactivation point of the same snapshot.*

Proof. Consider each of the different causes for the visibility of *Op* according to Definition 5.7.11.

If *Op* is an INSERT visible by Snapshot i because the critical point of *Op* occurs before the beginning of the node-traversal phase of Snapshot i, then it also happens before the deactivation point of Snapshot i, establishing the claim.

If *Op* is an INSERT visible by Snapshot i because during the node-traversal phase of Snapshot i, the scanning thread reads *Op*'s operation node then the critical point of *Op* must occur before the end of the node-traversal phase of Snapshot i, and hence also before the deactivation point, establishing the claim.

If *Op* is an INSERT visible by Snapshot i because of a report of *Op*'s operation node's insertion then consider the order of steps a thread executes when it reports. First, the reporting thread must complete, or witness the completion of the physical insertion of the operation node. Then, the reporting thread reads the `PSC` pointer, and then checks whether the snapshot is still active. The thread only reports if the snapshot is

still active at that point. Thus, the critical point of the *Op* must occur prior to the deactivation point of Snapshot i, establishing the claim.

If *Op* is an INSERT visible by Snapshot i because of a report of *Op*'s operation node's deletion the logic is similar. Consider the order of steps a thread executes when it reports. First, the reporting thread must complete, or witness the completion of the logical deletion of the operation node. Then, the reporting thread reads the `PSC` pointer, and then checks whether the snapshot is still active. The thread only reports if the snapshot is still active at that point. Thus, the logical deletion of the operation node, and hence also the physical insertion of the same node, must occur prior to the deactivation point of Snapshot i, establishing the claim.

If *Op* is a DELETE visible by Snapshot i because the critical point of *Op* occurs before the beginning of the node-traversal phase of Snapshot i, then it also happens before the deactivation point of Snapshot i, establishing the claim.

If *Op* is a DELETE visible by Snapshot i because during the node-traversal phase of Snapshot i, the scanning thread reads *Op*'s operation node and finds it logically deleted, then the critical point of *Op* must occur before the end of the node-traversal phase of Snapshot i, and hence also before the deactivation point, establishing the claim.

If *Op* is a DELETE visible by Snapshot i because of a report of *Op*'s operation node's deletion, consider the order of steps a thread executes when it reports. First, the reporting thread must complete, or witness the completion of the logical deletion of the operation node. Then, the reporting thread reads the `PSC` pointer, and then checks whether the snapshot is still active. The thread only reports if the snapshot is still active at that point. Thus, the logical deletion of the operation node must occur prior to the deactivation point of Snapshot i, establishing the claim. □

Corollary 5.1. *If Operation Op is visible by Snapshot i, then Op is visible by all later snapshots as well.*

Proof. Let i, ℓ be integers such that $0 < i < \ell \leq$ total number of snapshots, and assume *Op* is visible by Snapshot i. Then, by Claim 5.7.17, the critical point of *Op* occurs before the deactivation point of Snapshot i, which is before the node-traversal phase of Snapshot ℓ. Thus, the critical point of *Op* occurs before the beginning of the node-traversal phase of Snapshot ℓ, and thus, by Claim 5.7.13, *Op* is visible by Snapshot ℓ. □

Claim 5.7.18. *If* $\text{Operation}_{k,j}$ *is non-quiet, then* $\forall q,\ q \leq j$*, it holds that* $\text{Operation}_{k,q}$ *is invoked before the critical point of* $\text{Operation}_{k,j}$.

Proof: By Definition 5.7.2, the order of operations in the base-order is according to their linearization points in Harris's linked-list. For a non-quiet operation, this linearization point and the critical point are the same one. Thus, for a non-quiet operation $Operation_{k,j}$, every $Operation_{k,q}$ that comes before $Operation_{k,j}$ in $Base_E$ must be invoked in E prior to the critical point of $Operation_{k,j}$.

Corollary 5.2. *If* $\text{Operation}_{k,j}$ *is visible by Snapshot* i*, then* $\forall q \leq j$ $\text{Operation}_{k,q}$ *is invoked before the deactivation point of Snapshot* i*.*

Proof. This follows from Claims 5.7.17 and 5.7.18. □

Claim 5.7.19. *If during the execution of an operation Op, the owner thread of Op calls the* REPORT *method of the snap-collector associated with Snapshot* i*, then either the report is successful, or Operation Op completes its execution after the deactivation point of Snapshot* i*.*

Proof. The report must be successful, unless the BLOCKFURTHERREPORTS method is executed (by the scanning thread) first. However, BLOCKFURTHERREPORTS is only invoked after the deactivation point. □

Remark. Note that the report attempted during the execution of *Op* does not necessarily report *Op*. Recall that threads sometimes report the operations of other threads. In particular, *Op* could be a quiet operation.

Claim 5.7.20. *Let Op be a* DELETE *operation. If Op is non-visible by Snapshot* i*, then Op's operation node is physically deleted after the deactivation point of Snapshot* i*.*

Proof. The logical deletion of *Op*'s operation node must occur after the beginning of the node-traversal phase of Snapshot i, otherwise the operation would have been visible. Consider the thread T that executed the physical deletion. After seeing the node as logically deleted (alternatively, after marking it as logically deleted itself, if T is the same thread that made the logical deletion), T reads the `PSC` pointer and then checks whether the snapshot is active. The check whether the snapshot is active must happen after the logical deletion, which must occur after the beginning of the node-traversal phase of Snapshot i (otherwise, the operation would have been visible). Thus, either the Snapshot i is still active at the point of the check, or the check is after the deactivation point. If the check is after the deactivation point then so is the physical deletion and the claim is proven. If the snapshot is still active, then T will attempt to report the deletion of *Op*'s operation node. T can't be successful in this report, because that would make *Op* visible. Thus, the report must be completed after the BLOCKFURTHERREPORTS method is executed, which is also after the deactivation point. □

Corollary 5.3. *If (*$\text{Operation}_{k,j}$ *is non-visible by Snapshot* i*) and (*$\text{Node(Operation)}_{k,j}$ *is deleted in E) then* $\text{Node(Operation)}_{k,j}$ *is physically deleted after the deactivation point of Snapshot* i*.*

Proof. If $Operation_{k,j}$ is a DELETE operation, then the corollary is equivalent to Claim 5.7.20. If $Operation_{k,j}$ is an INSERT operation, and its operation-node is deleted in the execution E, then there must be a $q > j$ such that $Operation_{k,q}$ is the matching

operation of $Operation_{k,j}$. By the monotonicity of visibility (Claim 5.7.16), $Operation_{k,q}$ must be non-visible as well, and then by Claim 5.7.20, $Node(Operation)_{k,q}$ (which is the same of $Node(Operation)_{k,j}$ is physically deleted after the deactivation point of Snapshot i. □

In the next claim we show that if an operation is non-visible by Snapshot i, then it is completed after the deactivation point of the snapshot. The goal is to let the analysis set the linearization point of this operation to *after* the linearization point of the snapshot. The setting of the linearization points of the operations and the snapshots will be discussed later. To prove this claim, we must first revisit the linearization points in Harris's linked-list, as discussed in Definition 5.7.2.

In every operation in Harris's linked-list, the list is first searched for the operation key. Searching for a key is done by traversing the nodes of the list starting from the head. During the search, the searching thread attempts to physically remove logically deleted nodes. Occasionally, the traversal needs to restart from the head of the list. This is due to contention, i.e., a failure in the execution of a desired CAS. This CAS may either be an attempt to physically remove a logically deleted node, or it may be an attempt to execute the critical point of the operation. For example, a thread executing an INSERT operation may fail in the CAS that attempts to physically insert the operation-node into the list. In Harris's algorithm, this failure also causes a new traversal of the list, since the result of the old traversal may now be obsolete.

The linearization points of non-quiet operations, as defined in Definition 5.7.2, are according to the execution step of a critical CAS (physically inserting, or logically deleting a node.) As to quiet operations, with the exception of a failed CONTAINS, the linearization points are at the end of the last time the list is traversed during the operation. A failed CONTAINS could be linearized either at the end of the traversal of the list (if the required key is not found), or immediately after the logical deletion of a node with the same key (if the key is found in this logically deleted node). In the following facts 5.7.21 and 5.7.22 we formalize the intuition that if a non-quiet $Operation_{k,j}$ is linearized before another operation on the same key k, then the later operation "sees" the list after the list is modified by $Operation_{k,j}$.

In the first assertion, we state the simple fact that if a later operation does not "see" a node of an earlier operation, than that node must be removed before the later operation terminates.

Fact 5.7.21. If $Operation_{k,j}$ is non-quiet, then $\forall q > j$ either $Operation_{k,q}$ is completed after the physical deletion of $Node(Operation)_{k,j}$, or the owner thread of $Operation_{k,q}$ reads $Node(Operation)_{k,j}$ (during the execution of $Operation_{k,q}$).

Note that $Node(Operation_{k,j})$ is well defined and this node can be read by $Operation_{k,q}$ no matter if $Operation_{k,j}$ is a delete operation (and then $Node(Operation_{k,j})$ is about to be deleted) or an insert operation that is installing $Node(Operation_{k,j})$ into the list. If $Operation_{k,j}$ is a DELETE operation, then we can state a slightly stronger fact.

Fact 5.7.22. If $Operation_{k,j}$ is a non-quiet DELETE operation, then $\forall q > j$ either $Operation_{k,q}$ is after the physical deletion of $Node(Operation)_{k,j}$, or the owner thread of $Operation_{k,q}$ reads $Node(Operation)_{k,j}$ and finds it logically deleted (during the execution of $Operation_{k,q}$).

Claim 5.7.23. *If* Operation$_{k,j}$ *is non-visible by Snapshot* i *then* $\forall q \geq j$ Operation$_{k,q}$ *is completed after the deactivation point of Snapshot* i*.*

Proof. Let q, j be integers such that $q \geq j$. Since $Operation_{k,j}$ is non-visible by Snapshot i, then its critical point must occur after the beginning of the node-traversal phase of Snapshot i (Claim 5.7.13). If $q = j$, then $Operaion_{k,q}$ is completed after the deactivation point of Snapshot i by Claim 5.7.14. Assume $q > j$. Thus, using Fact 5.7.21, either 1) $Operation_{k,q}$ is completed after the physical deletion of $Node(Operation)_{k,j}$, or 2) during $Operation_{k,q}$ the owner thread of it reads $Node(Operation)_{k,j}$. If (1), then because $Operation_{k,j}$ is non-visible by Snapshot i, the deletion of $Node(Operation)_{k,j}$ must occur after the deactivation point of Snapshot i (Corollary 5.3), and thus $Operation_{k,q}$ is completed after the deactivation point as well.

Otherwise (2), $Operation_{k,q}$ must read $Node(Operation)_{k,j}$. We divide into four cases: $Operation_{k,j}$ is an INSERT and the (later) quiet $Operation_{k,q}$ reads $Node(Operation)_{k,j}$ as not logically deleted; that $Operation_{k,j}$ is an INSERT and that $Operation_{k,q}$ reads $Node(Operation)_{k,j}$ as logically deleted; $Operation_{k,j}$ is an INSERT and $Operation_{k,q}$ is its matching DELETE; and finally, $Operation_{k,j}$ is a DELETE.

In all of these cases, the logic is the same: the owner thread of the later $Operation_{k,q}$ must read the `PSC` pointer to see if there is an active snapshot. If Snapshot i is completed (no longer active), then the claim is established. If the snapshot is still active, then the owner of $Operation_{k,q}$ will attempt a report that must fail (otherwise $Operation_{k,j}$ would have been visible), and then by Claim 5.7.19, $Operation_{k,q}$ must complete after the deactivation point of Snapshot i. In all the cases, assume the critical point of $Operation_{k,j}$ occurs after the beginning of the node-traversal phase of Snapshot i. This is certain because $Operation_{k,j}$ is non-visible by Snapshot i, and then by Claim 5.7.13, this assumption is correct.

If $Operation_{k,j}$ is an insert and $Operation_{k,q}$ is a quiet operation that sees $Node(Operation)_{k,j}$ as unmarked, it will read the `PSC` pointer to test if there is an active snapshot. Since the critical point of $Operation_{k,j}$ (which is the physical insertion of $Node(Operation)_{k,j}$) occurs after the beginning of the node-traversal of Snapshot i, then if Snapshot i is not currently active we are done. If Snapshot i is still active, the owner thread of $Operation_{k,q}$ will attempt to report the insertion of $Node(Operation)_{k,j}$. This report must fail (otherwise $Operation_{k,j}$ would have been visible by Snapshot i), and thus by Claim 5.7.19, $Operation_{k,q}$ must complete after the deactivation point of Snapshot i.

If $Operation_{k,j}$ is an insert and $Operation_{k,q}$ sees $Node(Operation)_{k,j}$ as logically deleted, it will read the `PSC` pointer to test if there is an active snapshot. The logical

deletion of *Node(Operation)*$_{k,j}$ can only happen after the physical insertion of *Node(Operation)*$_{k,j}$, which occurs after the beginning of the node-traversal of Snapshot i. Thus, if Snapshot i is not currently active we are done. If Snapshot i is still active, the owner thread of *Operation*$_{k,q}$ will attempt to report the deletion of *Node(Operation)*$_{k,j}$. This report must fail (otherwise *Operation*$_{k,j}$, which is the insertion of the same node, would have been visible by Snapshot i), and thus by Claim 5.7.19, *Operation*$_{k,q}$ must complete after the deactivation point of Snapshot i.

If *Operation*$_{k,j}$ is an insert and *Operation*$_{k,q}$ is its matching (delete) operation, then after the logical deletion, the owner thread of *Operation*$_{k,q}$ will read the `PSC` pointer to see if there is an active snapshot. Again, if not, then we are done. If Snapshot i is still active, then the thread will attempt to report the deletion of the operation-node. This report must fail, otherwise *Operation*$_{k,j}$ would have been visible, and again, by Claim 5.7.19, this means *Operation*$_{k,q}$ is completed after the deactivation point of Snapshot i.

If *Operation*$_{k,j}$ is a DELETE operation, we use the strong Fact 5.7.22. Using this stronger fact, the owner of the (later) *Operation*$_{k,q}$ not only reads *Node(Operation)*$_{k,j}$, but finds it logically deleted. The owner thread of *Operation*$_{k,q}$ will then read the `PSC` pointer. Again, If Snapshot i is no longer active, we are done. Otherwise, the owner of *Operation*$_{k,q}$ will attempt to report a deletion. This report must fail, otherwise *Operation*$_{k,j}$ would have been visible, and again, by Claim 5.7.19, this means *Operation*$_{k,q}$ is completed after the deactivation point of Snapshot i. □

Lemma 5.7.24. $\forall i$ *snapshot i returns a result consistent with a sequential execution in which all the operations that are visible by Snapshot i are executed before it, none of the non-visible operations by Snapshot i are executed before it, and $\forall k$, the relative order of the operations with key k is the same as in the base-order.*

Remark. Note that because of the monotonicity of visibility (Claim 5.7.16), such an order exists.

Proof. Let i be an an integer such that $1 \leq i \leq$ total number of snapshots. We denote by R_i the set of nodes read by the scanning thread of Snapshot i during the node-traversal step, excluding those read as logically deleted; by I_i the set of nodes reported as inserted to the snap-collector corresponding with Snapshot i; and by D_i the set of nodes reported as deleted to the snap-collector. According to the snapshot algorithm, Snapshot i returns $(R_i \cup I_i) \setminus D_i$.

According to the set ADT specifications, a node with associated key k is in the set, iff it is the last node to have been successfully inserted into the set with key k, and no delete(k) followed its insertion. Let k be a possible key. We will examine three cases: 1) the last operation with operation key k visible by Snapshot i is an insert operation, 2) the last operation with operation key k visible by Snapshot i is a delete operation, and 3) no operation with operation key k is visible by Snapshot i. In each case, we will show that for every node with key k, it belongs to $(R_i \cup I_i) \setminus D_i$ iff it is in the set according

to the ADT specifications. Before referring to each of the three cases, we will prove two intermediate claims that will help.

Claim 5.7.25. *If Operation Op is non-visible by Snapshot i then Op's operation node* $\notin (R_i \cup I_i) \setminus D_i$

Proof. If *Op* is non-visible by Snapshot i, then the scanning thread does not see *Op*'s operation node during the node-traversal phase of Snapshot i (as that would make *Op* visible), and thus, *Op*'s operation node $\notin R_i$. Likewise, if *Op* is non-visible by Snapshot i, the insertion of *Op*'s operation node is not (successfully) reported to the snap-collector associated with Snapshot i (as that too would make *Op* visible), and thus, *Op*'s operation node $\notin R_i$. □

Claim 5.7.26. *If* $\text{Operation}_{k,j}$ *is a* DELETE *operation visible by Snapshot i then* $\forall q \leq j$ *such that* $\text{Operation}_{k,q}$ *is non-quiet,* $\text{Node(Operation)}_{k,j} \notin (R_i \cup I_i) \setminus D_i$.

Remark. Recall that for a quiet operation, its operation node is undefined.

Proof. Let q be an integer such that $q \leq j$ and $Operation_{k,q}$ is non-quiet. Due to the monotonicity of visibility (Claim 5.7.16), $Operation_{k,q}$ is visible. Whether $Operation_{k,q}$ is an insert or a delete, the deletion of $Node(Operation)_{k,q}$ is visible. If $Operation_{k,q}$ is a delete, then this is immediate. If $Operation_{k,q}$ is an insert, then $j \neq q$ and the matching operation of $Operation_{k,q}$ must also be visible due to the monotonicity of visibility (Claim 5.7.16). Consider the three possible causes for a delete operation to be visible (Definition 5.7.11).

The deletion of $Node(Operation)_{k,q}$ can be visible because the node is marked as logically deleted prior to the beginning of the node traversal of Snapshot i. In such a case, $Node(Operation)_{k,q} \notin R_i$, because the scanning thread cannot read it to be logically unmarked. Also $Node(Operation)_{k,q} \notin I_i$, because in order to report an insertion a thread reads the `PSC`, then checks if the snapshot is active, then check if the node is marked, and only then report it. However, $Node(Operation)_{k,q}$ is already marked when Snapshot i becomes active, so its insertion cannot be reported. Thus, in that case $Node(Operation)_{k,q} \notin (R_i \cup I_i) \setminus D_i$.

The deletion of $Node(Operation)_{k,q}$ can be visible because the node is successfully reported as deleted into the snap-collector. In that case $Node(Operation)_{k,q} \in D_i$, and thus $Node(Operation)_{k,q} \notin (R_i \cup I_i) \setminus D_i$.

The deletion of $Node(Operation)_{k,q}$ can be visible by Snapshot i because during the node-traversal the scanning thread reads $Node(Operation)_{k,q}$ and finds it logically deleted, and the insertion of $Node(Operation)_{k,q}$ is not successfully reported into the snap-collector associated with Snapshot i. In such a case, $Node(Operation)_{k,q} \notin I_i$ (because the insertion of $Node(Operation)_{k,q}$ is not reported), and $Node(Operation)_{k,q} \notin R_i$ (because the scanning thread read $Node(Operation)_{k,q}$ as logically deleted). We conclude that in this case as well, $Node(Operation)_{k,q} \notin (R_i \cup I_i) \setminus D_i$. □

We now go back to refer to each of the three cases needed to complete the proof of Lemma 5.7.24. *Case1: The last operation with operation key k visible by Snapshot i is an insert operation.* Let Op be this operation, and N the inserted node. In this case, according to the ADT specifications, N is in the set, and no other node with key k is in the set. Thus, we need to show that: i) $N \in R_i \cup I_i$, ii) $N \notin D_i$, and iii) any other node $N_2 \neq N$ with key k satisfies that $N_2 \notin (R_i \cup I_i) \setminus D_i$. The first two follow almost immediately from the definition of visible operations by Snapshot i: if Op is visible, then either the scanning thread reads its associated node during the node traversal ($N \in R_i$) or the node is reported as inserted ($N \in I_i$). An operation that deletes N, if exists, must come after Op. However, Op is the last visible operation with key k, and due to the monotonicity of visibility, such an operation must be non-visible. Thus, $N \notin D_i$, otherwise the operation that deletes N would have been visible.

To complete Case1, it remains to show that any other node $N_2 \neq N$ with key k satisfies that $N_2 \notin (R_i \cup I_i) \setminus D_i$. Let Op_2 be the operation that inserted N_2 into the list. If Op_2 comes after Op in the base-order, then by the monotonicity of visibility (Claim 5.7.16), Op_2 is non-visible, and thus, by Claim 5.7.25 $N_2 \notin R_i \cup I_i \setminus D_i$.

If Op_2 comes before Op in the base-order, then Op is not the first operation with key k in the base-order. Consider the operation previous to Op in the base-order, Op_p, which must be a delete(k). Let j be an integer such that $Operation_{k,j}$ is Op_p. By the monotonicity of visibility (Claim 5.7.16), $Operation_{k,j}$ is visible by Snapshot i. Thus, $Operation_{k,j}$ is a DELETE operation visible by Snapshot i, and thus by Claim 5.7.26, $\forall q \leq j$ such that $Operation_{k,q}$ is non-quiet, $Node(Operation)_{k,q} \notin (R_i \cup I_i) \setminus D_i$. Since Op_2 comes before Op in the base-order, then there exists $q < j$ such that $Op_2 = Operation_{k,q}$, and $N_2 = Node(Operation)_{k,q} \notin (R_i \cup I_i) \setminus D_i$.

Case2: The last operation with key k visible by Snapshot i is a delete operation. In such a case, by the specifications of the ADT, no node with key k should be returned by the snapshot. Thus, we need to show that any node N with key k satisfies: $N \notin (R_i \cup I_i) \setminus D_i$. Let j be an integer such that the last operation with key k visible by Snapshot i is $Operation_{k,j}$ (which is a delete operation). For all $q > j$, by the monotonicity of visibility (Claim 5.7.16), $Operation_{k,q}$ is either quiet (and has no operation-node) or non-visible by Snapshot i. If $Operation_{k,q}$ is non-visible, then by Claim 5.7.25, $Node(Operation)_{k,q} \notin (R_i \cup I_i) \setminus D_i$. If $q \leq j$, then by Claim 5.7.26, $\forall q \leq j$ such that $Operation_{k,q}$ is non-quiet, $Node(Operation)_{k,q} \notin (R_i \cup I_i) \setminus D_i$.

Case3: no operation with operation key k is visible by Snapshot i. Thus any operation with key k is either quiet (and have no operation node) or non-visible by Snapshot i, and then by Claim 5.7.25, its operation node $\notin (R_i \cup I_i) \setminus D_i$. □

Corollary 5.4. *Every snapshot i returns a result consistent with a sequential execution in which all the operations that are visible by Snapshot i are executed before it, none of the non-visible operations by Snapshot i are executed before it, and for every (non-snapshot) operation Op with key k, the set of non-quiet operations with the same key k*

that comes prior to Op is the same as in the base-order.

Proof. Follows immediately from Lemma 5.7.24. The total order we use for in Corollary 5.4 is the same as the one defined in 5.7.24, apart from possible reordering of quiet operations without moving them across any visible operations. Such reordering does not change the result of a sequential execution. □

5.7.5 Sequential and Real-Time Consistency of the Whole-Order.

In what follows we turn our attention back to the whole-order. To prove that the iterable list algorithm is linearizable, we will prove that whole-order satisfies both sequential consistency and real-time consistency. We will use a few intermediate claims in the process.

Claim 5.7.27. *For every operation Op, the set of non-quiet operations with the same key k that comes prior to Op is the same in whole-order and in base-order.*

Remark. Although whole-order contains SNAPSHOT operations, this claim refers only to non-snapshot operations.

Proof. By induction of the steps of the whole-order algorithm. The invariant that for every operation Op, the set of non-quiet operations with the same key k that comes prior to Op is the same as in base-order, is maintained throughout the whole-order algorithm.

Initially, whole-order is set to be identical to base-order. Only lines 12 and 13 move (non-snapshot) operations. We will show that none of these lines can violate the invariant. Let i be an integer such that $1 \leq i \leq$ total number of snapshots. Consider the execution of lines 12 and 13 in loop-iteration number i of the whole-order algorithm. (The term "loop-iteration number i of the whole-order algorithm" should not be confused with Snapshot i of the execution E. Using this terminology, loop-iteration number i in the whole-order algorithm is the i'th execution of Lines 3-13 in the whole-algorithm depicted in Figure 5.4. During the execution of loop-iteration number i of the whole-order algorithm, Snapshot i of the execution E is placed inside the whole-order.)

Line 12. In this line we move quiet operations that are invoked after the deactivation point of Snapshot i to be placed immediately after Snapshot i (given that before the execution of this line these quiet operations are placed before the snapshot). Let Op_A be an operation such that $Op_A \in$ `Premature-Quiets`. Op_A is invoked after the deactivation point of Snapshot i, and thus, By Corollary 5.2, there are no visible operations by Snapshot i with the same key placed after Op_A. Prior to the execution of line 12, Snapshot i is placed before the first non-visible operation (by Snapshot i). Thus, there are no non-visible operations placed between Op_A and Snapshot i in whole-order. Thus, moving Op_A to be immediately after Snapshot i does not move it across any non-quiet operations with the same key.

Line 13. In this line we move quiet operations that are completed before the deactivation point of Snapshot i to be placed immediately before Snapshot i. This line also moves all visible operations by Snapshot i to be placed immediately before the Snapshot i. Again, operations are moved only if before the execution of line 13 they were placed at the "wrong" side of the Snapshot i (in this case, after it). By the monotonicity of visibility (Claim 5.7.16), this replacement cannot cause any non-quiet operation to move across another non-quiet operation with the same key.

As for quiet operations, let Op_B be an operation such that $Op_B \in$ `Belated-Quiets`. Op_B is completed before the deactivation point of Snapshot i, and thus, by Claim 5.7.23, there are no non-visible operations by Snapshot i with the same key placed before Op_B. Thus, moving Op_B to be immediately before Snapshot i cannot move it across any non-visible operation with the same key. It also cannot move it across any visible operation, because any visible operation by Snapshot i placed after Snapshot i belongs to `Belated-Visibles` and is moved along with Op_B, in a way that retains their relative order. □

Claim 5.7.28. *$\forall i$, $1 \leq i \leq$ total number of snapshots, every visible operation by Snapshot i is placed before Snapshot i in whole-order, and every non-visible operation by Snapshot i is placed after Snapshot i in whole-order.*

Proof. Let i be an integer such that $1 \leq i \leq$ total number of snapshots. Snapshot i is first placed in whole-order before all the non-visible operations by Snapshot i (line 3). Later, in line 13, all the visible operations by Snapshot i that were previously placed after Snapshot i, are placed immediately before it. Thus, immediately after the execution of line 13 in loop-iteration number i of the whole-order algorithm, Snapshot i is placed after all the operations visible by it, and before all the operations that are non-visible by it.

It remains to show that no subsequent executions of line 13 will move non-quiet operations across Snapshot i (line 13 is the only line that moves non-quiet operations.) Due to the monotonicity of visibility (Claim 5.1), for every $j > i$ Snapshot j is placed in whole-order after Snapshot i. Subsequent executions of line 13 will move operations that are placed after a snapshot j for which $j > i$ to be immediately before Snapshot j. This does not move them across Snapshot i. □

Claim 5.7.29. *Every* SNAPSHOT *operation in E returns a result consistent with the result it returns in the sequential whole-order execution.*

Proof. The claim follows immediately from Corollary 5.4, Claim 5.7.27 and Claim 5.7.28. Corollary 5.4 claims an SNAPSHOT operation in E returns a result consistent with a sequential execution in which the operations before the snapshot are the visible operations, and for each operation Op, the set of non-quiet operations that come priors to Op is the same as in the base-order. Claim 5.7.28 claims that in whole-order the

operations that comes before a snapshot are the visible ones (and perhaps quiet ones as well). Claim 5.7.27 claims that for each operation Op, the set of non-quiet operations that come priors to Op is the same as in the base-order. □

Corollary 5.5. *whole-order satisfies sequential consistency.*

This follows from claims 5.7.27 and 5.7.29.

Claim 5.7.30. *If Op_1 and Op_2 are two non-concurrent operations in E, none of which is an* SNAPSHOT *operation, then they retain their sequential order in whole-order.*

Proof. By induction of the steps of the whole-order algorithm. Let Op_1 and Op_2 be two non-concurrent operations in E, non of which a SNAPSHOT operation. The invariant that Op_1 and Op_2 retain their sequential order is maintained throughout the whole-order algorithm. Initially, whole-order is set to be identical to base-order. Only lines 12 and 13 move (non-snapshot) operations. We will show that none of these lines can violate the invariant. Let i be an integer such that $1 \leq i \leq$ total number of snapshots. Consider the execution of lines 12 and 13 in loop-iteration number i of the whole-order algorithm.

Line 12. In this line we move quiet operations that are invoked after the deactivation point of Snapshot i to be placed immediately after Snapshot i (given that before the execution of this line they are placed before the snapshot). If neither of Op_1 and Op_2 are moved in the execution of line 12, then their order cannot be disturbed in the execution of this line. If both of them are moved, then again their order cannot be disturbed, because the relative order of moved operations is retained. It remains to consider the case when one operation is moved and the other is not. Assume, without loss of generality, that Op_1 is moved.

If prior to the execution of line 12, Op_2 is not placed in whole-order between Op_1 and Snapshot i, then moving Op_1 across Snapshot i will not disturb the relative order of operations Op_1 and Op_2. If Op_2 is placed between Op_1 and Snapshot i, then we claim that Op_1 and Op_2 must be concurrent in E (yielding contradiction).

Op_2 is invoked before the deactivation point of Snapshot i. Before the execution of line 12, Op_2 is placed in whole-order between Op_1 and Snapshot i, which means Op_2 is placed before Snapshot i. Thus, if Op_2 is non-quiet it must be visible by Snapshot i (as the snapshot is placed before the first non-visible operation), and thus, by Claim 5.7.15, Op_2 cannot be invoked after the deactivation point. If Op_2 is quiet, then again it must be invoked before the deactivation point of Snapshot i, otherwise it would have also been moved in the execution of line 12.

Now, Op_2 is invoked before the deactivation point of Snapshot i, Op_1 is invoked after the deactivation point of Snapshot i, but Op_2 is placed after Op_1 before the execution of 12. Since we assume the invariant holds before the execution line 12, then Op_1 and Op_2 must be concurrent in E.

Line 13. In this line we move quiet operations that are completed before the deactivation point of Snapshot i to be placed immediately before the snapshot. This

line also moves all visible operations by Snapshot i to be placed immediately before the snapshot. Again, operations are moved only if before the execution of this line they were placed at the "wrong" side of the snapshot (in this case, after it). The logic is very similar to the one used for line 12.

If neither of Op_1 and Op_2 are moved in the execution of line 13, then their order cannot be disturbed in the execution of this line. If both of them are moved, then again their order cannot be disturbed, because the relative order of moved operations is retained. It remains to consider the case when one operation is moved and the other is not. Assume, without loss of generality, that Op_1 is moved.

If prior to the execution of line 13, Op_2 is not placed in whole-order between Op_1 and Snapshot i, then moving Op_1 across Snapshot i will not disturb the relative order of operations Op_1 and Op_2. If Op_2 is placed between Op_1 and Snapshot i, then we claim that Op_1 and Op_2 must be concurrent in E (yielding contradiction).

Op_2 is completed after the deactivation point of Snapshot i. Before the execution of line 13, Op_2 is placed in whole-order between Op_1 and Snapshot i, which means Op_2 is placed after Snapshot i. If Op_2 is non-quiet, then it must be non-visible by Snapshot i otherwise it would have also been moved in the execution of line 13, and thus, by Claim 5.7.14, it cannot be completed before the deactivation point. If Op_2 is quiet, then it must be completed after the deactivation point of Snapshot i, otherwise it would have also been moved in the execution of line 13.

Now, Op_2 is completed after the deactivation point of Snapshot i, Op_1 is completed before the deactivation point of Snapshot i, but Op_2 is placed prior to Op_1 before the execution of 13. Since we assume the invariant holds before the execution line 13, then Op_1 and Op_2 must be concurrent in E. □

Claim 5.7.31. *$\forall i$, $1 \leq i \leq$ total number of snapshots, any operation that is completed before the deactivation point of Snapshot i in E is placed before Snapshot i in whole-order, and every operation that is invoked after the deactivation point of Snapshot i in E is placed after Snapshot i in whole-order*

Proof. Let i be an integer such that $1 \leq i \leq$ total number of snapshots, and let Op be a (non-snapshot) operation. If Op is visible by Snapshot i, then by Claim 5.7.15 it cannot be invoked after the deactivation point of Snapshot i. Also, since Op is visible, then by Claim 5.7.28 it must be placed before Snapshot i in whole-order. Thus, Op is placed before Snapshot i in whole-order and invoked before the deactivation point of Snapshot i in E.

If Op is non-visible by Snapshot i, then by Claim 5.7.14 it cannot be completed before the deactivation point of Snapshot i. Also, since Op is non-visible, then by Claim 5.7.28 it must be placed after Snapshot i in whole-order. Thus, Op is placed after Snapshot i in whole-order and completed after the deactivation point of Snapshot i in E.

If Op is a quiet operation that is invoked after the deactivation point of Snapshot i, then line 12 in loop-iteration number i of the whole-order algorithm will move it to be immediately after Snapshot i in whole-order, if it is previously before it. If Op is a quiet operation that is completed before the deactivation point of Snapshot i, then line 13 in loop-iteration number i of the whole-order algorithm will move it to be immediately before Snapshot i in whole-order, if it is previously after it. □

Corollary 5.6. *whole-order satisfies real-time consistency.*

This follows from claims 5.7.30 and 5.7.31.

Theorem 5.7. *The iterable list algorithm is linearizble.*

This follows from Corollary 5.5 and Corollary 5.6.

5.7.6 Adjusting the Proof for Multiple Scanners

The structure of the claims above holds for the case of multiple scanners as well, but some adjustments have to be made. First, instead of referring to an "Snapshot i", the proof should refer to a set of SNAPSHOT operations that share the same snap-collector. The division of a snapshot into the four phases: *activation*, *node-traversal*, *deactivation*, and *wrap-up* remains. The *activation* is completed in the step that assigns the snap-collector to PSC. The deactivation point is at the time the DEACTIVATE method is linearized for the associated snap-collector for the first time. The node-traversal begins immediately after the activation and ends at the deactivation point.

All the SNAPSHOT operations that share the same snap-collector are during their execution at the deactivation point. All of them also return the same result. Thus, they can all be linearized at the deactivation point, similar to the case of the single SNAPSHOT. Also, there is a sequential order between the sets of SNAPSHOT operations. They can be ordered according to their deactivation points (or according to the order in which they are pointed by the PSC, which is the same). Thus, it is still meaningful to refer to Snapshot-set i.

The definition of visibility requires some care: a scanning thread can now see and node but fail to install a report to it, because another scanning thread might invoke BLOCKFURTHERNODES. The definition of visibility for the case of concurrent snapshots follows.

Definition 5.7.32. (Visible Operations by Snapshot-Set i.) We say that a successful DELETE(k), which removed the node N from the list, is visible by Snapshot-set i, if at least one of the following holds.

- N is marked as logically deleted before the beginning of the node-traversal phase of Snapshot-set i.

- The deletion of N is (successfully) reported in the snap-collector associated with Snapshot-set i.
- During the node-traversal phase of Snapshot-set i, a scanning thread reads the node N, and finds it logically deleted, and the insertion of N is not (successfully) reported in the snap-collector associated with Snapshot i, and N is not added into the snap-collector by a different scanning thread that reads N and finds it not logically deleted.

We say that a successful INSERT(k), which inserted the node N into the list, is visible by Snapshot-set i, if at least one of the following holds.

- N is (physically) inserted before the beginning of the node-traversal phase of Snapshot-set i
- The insertion of N is (successfully) reported to the snap-collector associated with Snapshot-set i.
- During the node-traversal step of Snapshot i, a scanning thread reads the node N, and either finds it logically deleted or successfully adds it to the snap-collector associated with Snapshot-set i.
- The deletion of N is (successfully) reported in the snap-collector associated with Snapshot-set i.

In addition to these changes, some of the arguments in the proof specifically refer to the "scanning thread". Such arguments should generally be replaced by similar arguments that refer to "any of the scanning threads". However, naturally, the particulars of these adjustments slightly varies in each specific case.

5.7.7 Linearizability of the Snap-Collector

The snap-collector is a simple object to design, and there are many ways to design it. Due to optimizations, the implementation used in our measurements and described in Section 5.6 does not strictly follow the semantics of the snap-collector ADT. Thus, for the sake of the proof, we describe here a variant which is slightly less efficient, but follows the snap-collector ADT. This snap-collector also has the property of being almost trivially linearizable. We rely on the (linearizable) wait-free queue [KP11]. This queue is based on a linked-list of nodes. Though we refer to the snap-collector as a single object, its ADT semantics practically divide into three separate groups.

AddNode, BlockFurtherNodes, ReadPoiners . We maintain a wait-free queue for these three operations. The ADT semantics require that ReadPointers will return all the nodes added prior to BlockFurtherNodes. To add a node, a thread simply enqueues a pointer to it. To block further nodes, a thread enqueues a special value that can

be distinguished from other values (such as a NULL pointer). To read the pointers of nodes installed in the snap-collector, a thread reads the nodes in the queue one by one, until reaching the special "blocking" node. Nodes that are enqueued after the "blocking" node are not returned. The linearizability of the queue ensures the linearizability of these three operations.

AddReport, BlockFurtherReports, ReadReports . We maintain a separate wait-free queue for the reports of each different thread. The ADT semantics require that ReadReports will return all the reports that were added by a thread before the thread was blocked by the BlockFurtherReports method. To add a report, a thread enqueues it into its own designated queue. To block another thread from adding additional reports, a thread enqueues a special "blocking" value into the queue of the thread whose further reports are to be blocked. To read the reports, a thread reads the reports from all of these queues, in each queue stopping to read once reaching the "blocking" node. The linearizability of the queue ensures the linearizability of these three operations.

Deactivate, IsActive . We maintain a bit field, initiated to true. The ADT semantics require that IsActive will return true as long as Deactivate has not been called. To deactivate, a thread writes false into this field. To check if the snap-collector is active, the bit is read. The linearizability of these two operations is trivial.

5.8 Performance

In this section we report the performance of the proposed iterator, integrated with the lock-free linked-list and skiplist in Java. We used the linked-list implementation as included in the book "The Art of Multiprocessor Programming" by Herlihy and Shavit [HS08], and added to it the iterator mechanism described in this chapter. For the skiplist, we used the Java code of ConcurrentSkipListMap by Doug Lea, and added our mechanism. We also measured the performance of the CTrie [PBBO12]. The CTrie is included in the Scala 2.10.0 distribution, and we used this implementation to measure its performance. The CTrie implementation and our implementations support the dictionary ADT, in which each key is associated with a value. In this chapter we are only interested in the set ADT, so we used the boolean true value to serve as the associated value of all the keys.

All the tests were run on OpenJDK 6, on a system that features 4 AMD Opteron(TM) 6272 2.1GHz processors. Each processor has 8 cores (32 cores overall), and each core runs 2 hyper-threads (i.e., 64 concurrent threads overall). The system employs a memory of 128GB and an L2 cache of 2MB per processor.

The algorithms were tested on a micro-benchmark in which one thread repeatedly executes ITERATION operations, going over the nodes one by one continually. For the other threads, 50% of the operations are CONTAINS, 25% are INSERT, and 25% are

DELETE, with the number of threads varying between 1-31. In each test the keys for each operation were randomly and uniformly chosen in the ranges $[1, 32]$, $[1, 128]$, or $[1, 1024]$. In each test, all the threads were run concurrently for 2 seconds.

We run each specific test-case (i.e., the number of threads and the key range) 8 times for each algorithm (linked-list, CTrie, and skiplist). Each algorithm was run on a separate JVM, after first running this JVM for several seconds on the data structure to allow it to warm up. Each of the 8 measurements was run a 2 seconds interval. We repeated the complete set of experiments 3 times. Thus, overall, each test was run 24 times. The averages of these 24 measurements are reported in the figures.

For each key range, we present three different graphs. In the first graph, we measure the number of operations executed as a fraction of the number of operations executed without the additional iterating thread. For example, for a range of keys $[1, 32]$, for 20 threads, the number of operations executed while an additional thread is continually iterating the nodes is 89% of the number of operations executed by 20 threads in the skiplist data structure that *does not support iteration* at all. Thus, this graph presents the cost of adding the support for an iterator, and having a single thread continually iterate over the structure. For the CTrie, there is no available lock-free implementation that does not support iteration at all, so we simply report the number of operations as a fraction of the number of operations executed when there is no additional concurrent thread iterating over the structure. In the second graph, we report the absolute number of INSERT, DELETE, and CONTAINS operations executed in the different data structures while a single thread was iterating, and in the third graph we report the number of ITERATION operations that the single thread completed. This last measure stands for the efficiency of the iterator itself.

The results appear in Figure 5.5. In general, the results show that the iterator proposed in this chapter has a small overhead on the other threads (which execute INSERT, DELETE and CONTAINS), and in particular, much smaller than the overhead imposed by the CTrie iterator. The overhead of the proposed iterator for other threads is usually lower than 20%, except when the overall number of threads is very small. This means that the proposed iterator does relatively little damage to the scalability of the data structure. As for overall performance, we believe it is less indicative of the contribution of our work, as it reflects mainly the performance of the original data structures regardless of the iterator. Having said that, the linked-list performs best for 32 keys, the skiplist for 128 keys, and the CTrie and skiplist performs roughly the same for 1024 keys.

Standard Deviation and Error Bounds. The standard deviation in the measurements of the linked-list is quite small in all test-cases, up to 4%. This makes the error for 24 measurements bounded by less than 2.5% with a 99% confidence interval. The standard deviation for the skiplist measurements is only slightly higher than that for ranges of $[1, 32]$ and $[1, 128]$ keys, but it is significantly higher for a range of 1024 possible

keys, reaching 16%. This suggests an error bound of up to 10% for these measurements. In particular, this suggests that the fluctuations of the skiplist results with 1024 keys for high number of threads could be the result of measurements error, and that the difference in the absolute number of INSERT, DELETE, and CONTAINS operations for 1024 keys between the CTrie and the skiplist is inside the error margin.

The CTrie measurements have standard deviation of about 10% for ranges of $[1, 32]$ and $[1, 128]$ keys (error bounded at 6%), excluding the measurement of the number of operations done by a single thread while a different thread is constantly performing iterations. This single test-case yielded a high standard deviation of 18% for 32 keys and 84%(!) for 128 keys. The results also show that this test case is where the CTrie performance particularly bad (an overhead of 95% and 98%, and total operations 6 times and 10 times slower than the linked-list, for 32 and 128 keys, respectively). Thus, its bad performance in this case are coupled with high instability. For a 1024 possible keys the CTrie is more stable, with standard deviation of up to 8% in the measurements, suggesting an error bound smaller than 5%.

5.9 Conclusion

In this chapter we added support of lock-free and wait-free iterators for data structures that support set operations and that adhere to certain constraints. Our technique is especially adequate for linked-lists and skiplists. The proposed algorithm takes a snapshot of the data structure, while imposing low overhead on concurrent readers and writers of the data structure. Once a snapshot is obtained, iterations are run on it.

Our construction supports efficient snapshot taking and iterations for data structures that previously lacked it, such as linked-list and skiplist. Compared to previous work by Prokopec et al. [PBBO12] that presented a concurrent trie (CTrie) with support for a snapshot and iterations, the CTrie provides very fast (constant time) SNAPSHOT operations, while our construction enables lower overhead for the regular (INSERT, DELETE, and CONTAINS) set operations.

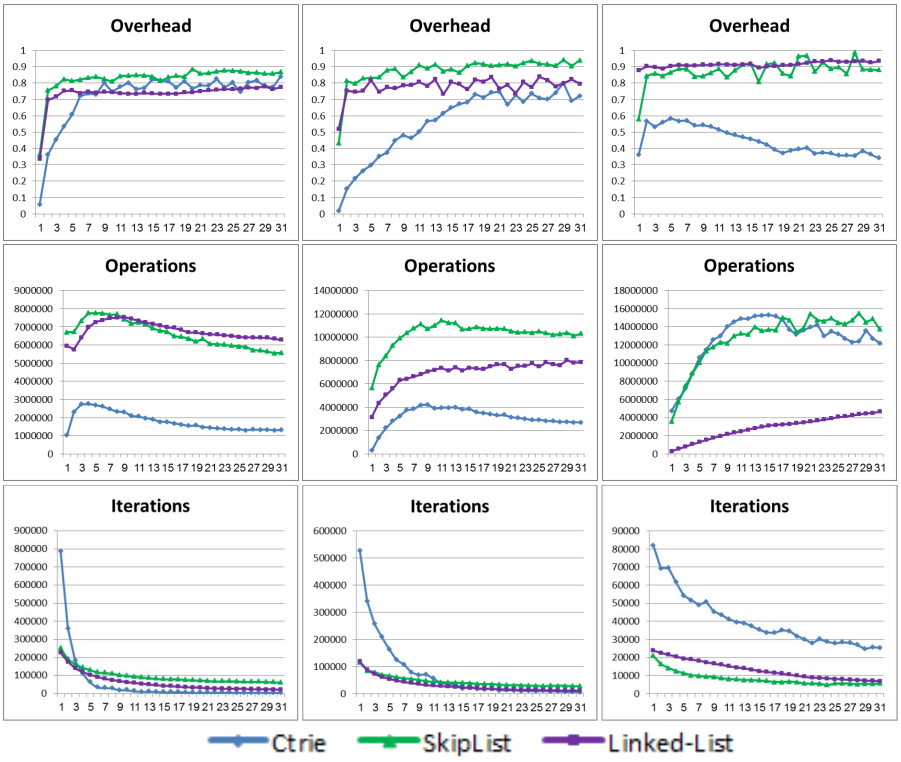

Figure 5.5: Results for 32 possible keys (left) 128 possible keys (middle) 1024 possible keys (right)

Chapter 6

A Practical Transactional Memory Interface

6.1 Introduction

As discussed in Section 1.5, hardware transactional memory (HTM) is becoming widely available on modern platforms. However, software using HTM requires at least two carefully-coordinated code paths: one for transactions, and at least one for when transactions either fail, or are not supported at all. Such a fall-back path is particularly important to enable implementations with a progress guarantee.

We present the MCMS interface that allows a simple design of fast concurrent data structures. MCMS-based code can execute fast when HTM support is provided, but it also executes well on platforms that do not support HTM, and it handles transaction failures as well. To demonstrate the advantage of such an abstraction, we designed MCMS-based linked-list and tree algorithms. The list algorithm outperforms all known lock-free linked-lists by a factor of up to X2.15. The tree algorithm builds on Ellen et al. [EFRvB10] and outperforms it by a factor of up to X1.37. Both algorithms are considerably simpler than their lock-free counterparts.

This chapter is organized is follows. Section 6.2 discusses additional related work to the work covered in Section 1.5. Section 6.3 formally define MCMS and discusses its implementation. Section 6.4 presents our MCMS based linked-list algorithm. Section 6.5 gives our MCMS based binary search tree algorithm. In Section 6.6 we discuss alternatives for a generic fall back execution. Performance measurements are given in Section 6.7, and we conclude this chapter in Section 6.8.

6.2 Additional Related Work

The search of means for simplifying the design of highly concurrent data structures, and in particular lock-free ones, has been long and it led to several important techniques and concepts. Transactional memory [HM93, ST97] is arguably the most general of these; a

transaction can pack any arbitrary operation to be executed atomically. But the high efficacy comes with a cost. State of the art software implementations of transactional memory incur a high performance cost, while hardware support only spans across few platforms, and usually only provides "best-effort" progress guarantee (e.g., the widely available Haswell RTM).

MCAS [IR94] is another tool for simplifying the design of concurrent data structures. It may be viewed as a special case of a transaction. Several CAS-based software implementations of MCAS exist [HFP02, Sun11] with reasonable performance. A similar, yet more restrictive primitive is the recent LLX/SCX [BER13]. These primitives enable to atomically read several words, but write only a single word. Atomically with the single write, it also allows to *finalize* other words, which has the effect of blocking their value from ever changing again. A CAS-based software implementation of these primitives is more efficient than any available implementation of MCAS, and these primitives have been shown to be particularly useful for designing trees [BER14]. Yet, allowing only a single word to be written atomically can be too restrictive: our MCMS linked-list algorithm, which atomically modifies two different pointers, cannot be easily implemented this way.

Dragojevic and Harris explored another form of restricted transactions in [DH12]. They showed that by moving much of the "book keeping" responsibility to the user, and keeping transactions very small, almost all of the overhead of software transactional memory can be avoided. Using their restricted transactions is more complicated than using MCAS, and they did not explore hardware transactional memory.

Speculative lock elision [RG01] is a technique to replace a mutual exclusion lock with speculative execution (i.e., transaction). This way several threads may execute the critical section concurrently. If a read/write or a write/write collision occurs, the speculative execution is aborted and a lock is taken. [BMV+07] studies the interaction between transactions and locks and identifies several pitfalls. Locks that are well suited to work with transactions are proposed in [RHP+07]. Intel's TSX extension also includes support of Hardware Lock Elision (HLE). Our MCMS interface lends itself to lock-elision, and also has the potential to use other fall-back paths, which could be lock-free.

6.3 The MCMS Operation

In this section we specify the MCMS interface, its semantics and implementation. The semantics of the MCMS interface are depicted in Figure 6.1(left). The MCMS operation receives three parameters as input. The first parameter is an array of CAS descriptors to be executed atomically, where each CAS descriptor has an `address`, an `expected value`, and a `new value`. The second parameter, N, is the length of the array, and the last parameter C signifies the number of entries at the beginning of the array that should only be compared (but not swapped). We use a convention that the addresses that should only be compared and not swapped are placed at the beginning of the array.

```
The MCMS Semantics
Atomically execute:
1: bool MCMS (CASDesc* descriptors, int N, int C) {
2:   for i in 1 to N: {
3:     if (*(descriptors[i].address)  !=
             descriptors[i].expected_val)  {
4:          return false;
5:     }
6:   }
7:   for i in  C+1 to N: {
8:     *(descriptors[i].address) =
           descriptors[i].new_val;
9:   }
10: return true;
11:}
```

```
HTM Implementation of the MCMS Operation
1: bool MCMS(CASDesc* descriptors, int N ,int C) {
2: while (true) {
3:   XBEGIN(retry);   // an aborted transaction
                      // jumps to the retry label
4:   for i in 1 to N: {
5:     if(*(descriptors[i].address) !=
             descriptors[i].expected_val) {
6:       XEND();
7:       return false; } }
8:   for i in C+1 to N: {
9:      *(descriptors[i].address) =
             descriptors[i].new_val;  }
10:  XEND();
11:  return true;
12:  retry:   // aborted transactions jump here
13:  for I in 1 to N: {
14:    if(*(descriptors[i].address) !=
             descriptors[i].expected_val) {
15:      return false; } } } }
```

Figure 6.1: The MCMS Semantics (left) and its HTM Implementation (right)

Their associated `new value` field is ignored.

6.3.1 Implementing MCMS with Hardware Transactional Memory

Intel Haswell Restricted Transactional Memory (RTM) introduces three new instructions: XBEGIN, XEND, XABORT. XBEGIN starts a transaction and receives a code location to which execution should branch in case of a transaction abort. XEND announces the end of a transaction, and XABORT forces an abort.

The implementation of MCMS, given in Figure 6.1(right), is mostly straightforward. First, begin a transaction. Then check to see that all the addresses contain their expected value. If not, complete the transaction and return false. If all addresses hold the expected value, then write the new values, complete the transaction and return true. If the transaction aborts, restart from the beginning. However, before restarting, read all the addresses outside a transaction, and compare them to the expected value. If one of them has a value different than the expected value, return false.

This last phase of comparing after an abort is not mandatory, but has two advantages. The first is that in case the transaction failed because another thread wrote to one of the MCMS addresses, then it is possible for the MCMS to simply fail without requiring an additional transaction. The second advantage is that it handles a problem with page faults under RTM. A page fault causes a transaction to abort (without bringing the page). In such a case, simply retrying the transaction repeatedly can be futile, as the transaction will repeatedly fail without loading the page from the disk. Loading the addresses between transactions renders the possibility of repeated failures due to page faults virtually impossible.

6.3.2 Implementing MCMS without TM support

We also implemented the MCMS operation using the method of Harris et al. [HFP02], including some optimizations suggested in that paper. As Harris's algorithm refers to MCAS, and not MCMS, we used identical expected value and new value for addresses that are only meant for comparison. The basic idea in Harris's algorithm is to create an object describing the desired MCAS, and then use a CAS to try and change each target address to point to this object if the address holds the expected value. If all addresses are modified to point to the object this way, then they all can be updated to hold the new values, otherwise the old values are restored. The full details of [HFP02] are considerably more complicated, and are not described here.

When the MCMS algorithm reads from an address that might be the target of an MCAS, it must be able to tell whether that memory holds regular data, or a special pointer to an MCAS descriptor. In our applications, we were able to steal the two least significant bits from target fields. For the list algorithm, each target field holds a pointer to another node, and regular pointer values have zero in those two bits. For the tree algorithm, each target field holds either a pointer or a binary flag, and we shift the flag value to the left by two bits.

6.4 The Linked-List Algorithm

We consider a sorted-list-based set of integers, similar to [Har01, FR04, Val95], supporting the INSERT, DELETE, and CONTAINS operations. Without locks, the main challenge when designing a linked-list is to prevent a node's next pointer from changing concurrently with (or after) the node's deletion. A node is typically deleted by changing its predecessor to point to its successor. This can be done by an atomic CAS, but such a CAS cannot by itself prevent an update to the deleted node's next pointer. For details, see [Har01].

Harris [Har01] solved this problem by partitioning the deletion of a node into two phases. In the first phase, the node's next pointer is *marked*, by setting a reserved bit on this pointer. This locks this pointer from ever changing again, but still allows it to be used to traverse the list. In the second phase, the node is physically removed by setting its predecessor to point to its successor. Harris uses the pointer least significant bit as the *mark bit*. This bit is typically unused, because the next pointer points to an aligned address.

Harris's mark bit is an elegant solution to the deletion problem, but Harris's algorithm still has some drawbacks. First, when a mark bit is used, traversing the list requires an additional masking operation to be done whenever reading a pointer. This operation poses an overhead on list traversals. Second, a thread that fails a CAS (due to contention) often restarts the list traversal from the list head. Fomitchev and Ruppert [FR04] suggested a remedy for the second drawback by introducing back-links into the

linked-list. The back-link is an additional field in each node and it is written during the node's deletion.

Fomitchev and Ruppert used three additional fields in each node in excess of the obligatory `key` and `next` pointer fields. Those fields are: the mark bit (similar to Harris), another *flag bit* (also adjoined to the next pointer), and a back-link pointer. To delete a node, a thread first flags its predecessor, then marks the node to be deleted, then writes the back-link from the node to the predecessor, and finally physically removes the node (the same CAS that removes the node also clears the flag of the predecessor.) Due to the overhead of additional CASes, this list typically performs slower in practice compared to the list of Harris.

To illustrate the simplicity of the MCMS operation we present a new linked-list algorithm. The MCMS list is simpler, faster (if HTM is available), and does not use any additional fields on top of the `key` and `next` pointer fields. Similarly to Fomitchev and Ruppert, the MCMS list never needs to start searching from the head on a contention failure.

The crux of our algorithm is that it uses the atomic MCMS to atomically modify the node's next pointer to be a back-link simultaneously with deleting it from the list (see Figure 6.2(b)). Thus the `next` pointer points to the next node while the node is in the list, and acts as a back-link once the node is deleted. Similar to [Har01, FR04, Val95] and others, we use a sentinel head node with a key of minus infinity, and a tail node with a key of infinity.

The algorithm is given in Figure 6.2(a)(left), and is surprisingly simple. The SEARCH method receives three parameters, a key to search for, and pointers to pointers to the left and right nodes. When the search returns, the pointer fields serves as outputs. The left node is set to the last node with a key smaller than the given search key. The right node is set to the first node with a key equal to or greater than the search key. The left node parameter also serves as in input for the method, and indicates where to start the search from.

An invariant of the algorithm is that if a node A (which was already inserted to the list) points to node B, and B's key is greater than A's key, then both nodes are currently in the list. When node B is deleted, modifying its next pointer to point to A serves two purposes. First, it serves the purpose of the mark bit that ensures any concurrent operation that might try to modify B's next pointer will fail, which is vital to the correctness of the algorithm. Yet, without necessitating a masking operation before using the next pointer. Second, it establishes a back-link, which other threads might use to avoid the necessity of redoing the search from scratch. Yet, this back-link does not necessitate additional fields in the object, nor specific checks before following this back-link.

Note that our algorithm, unlike the one of Fomitchev and Ruppert, doesn't require a separate field for the back link because of the atomicity provided by the MCMS operation. Without MCMS, it is hard to see how to avoid using two fields. Setting the

```
MCMS List:
1. void search(int key, Node** left,Node** right) {
2.   *right = (**left).next;
3.   While ((**right).key < key) {
4.     *left = *right;
5.     *right = (**left).next;  } }
6.
7. bool insert(int key) {
8.   Node *left = head; // head is first node in list
9.   Node *right;
10. Node *newNode = new Node(key);
11. While (true) {
12.   search(key, &left, &right);
13.   if ((*right).key == key)
14.     return false;  // key already exists
15.   (*newNode).next = right;
16.   if (CAS(&((*left).next), right, newNode))
17.     return true;  // successfully inserted
18.   } }
19.
20. bool delete(int key) {
21.  Node* left = head;
22.  Node* right;
23.  While (true) {
24.    search(key, &left, &right)
25.    if ((*right).key != key)
26.      return false; // key doesn't exist
27.    Node* succ = (*right).next;
28.    if (MCMS(<&((*left).next), right, succ>,
29.        <&((*right).next), succ, left>))
30.      return true;  // successfully deleted
31.    } }
32.
33. bool contains(int key) {
34.   Node *left = head, *right;
35.   search(key, &left, &right);
36.   return (*right).key == key;
37. }
```

(a) The MCMS List Code

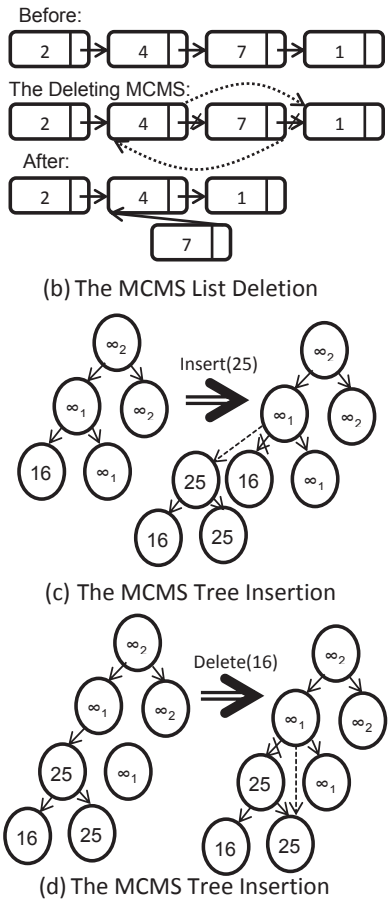

Figure 6.2: The List and Tree Algorithms

back-link first will cause the list to be unsearchable until the other CAS is completed, and postponing CAS-ing the back-link to the end of the DELETE operation is not enough to guarantee a thread never needs to search from the beginning.

6.5 The Binary Search Tree Algorithm

We base our tree algorithm on the binary search tree of Ellen et al. [EFRvB10] (this tree was shown in [BH11] to outperform both the lock-free skiplist Java implementation and the lock-based AVL tree of Bronson et al. [BCCO10]). Our tree is also a leaf oriented tree, meaning all the keys are stored in the leaves of the tree, and each internal node has exactly two children. However, in their original algorithm, each internal node stores a pointer to a designated *Info* object that stores all the information required to complete an operation. When a thread initiates an operation, it first searches the tree for appropriate location to apply it. Then it tests the internal node Info pointer to see whether there is already an ongoing operation, and helps such an operation if

needed. Then it allocates an Info object describing the desired change, and attempts to atomically make the appropriate internal node points to this info object using a CAS. Then, it can proceed with the operation, being aware that it might get help from other threads in the process.

MCMS allows all changes to take place simultaneously. This saves the algorithm designer the need to maintain an Info object, and also boosts performance in the common case, in which an HTM successfully commits. Similarly to a list, a central challenge in a lock-free binary search tree is to ensure that pointers of an internal node will not be modified while (or after) the node is deleted (see [EFRvB10] for details). For this purpose, in the MCMS tree algorithm, each internal node contains a mark bit (in addition to its key, and pointers to two children). The mark bit is in a separate field, not associated with any pointer. Leaf nodes contain only a key. Upon deleting an internal node, its mark bit is set. Each MCMS operation that changes pointers of a node also reads the mark bit and compares it to zero. If the bit is set, the MCMS will return false without changing the shared memory, guaranteeing that a deleted node's pointers are never mistakenly altered.

In order to avoid corner cases, we initialize the tree with two infinity keys, ∞_1 and ∞_2, such that $\infty_2 > \infty_1 >$ any other value. The root always has the value ∞_2 its right child is always ∞_2 and its left child is always ∞_1. This idea is borrowed from the original algorithm [EFRvB10]. Both the INSERT and DELETE operations begin by calling the search method. The search method traverses the tree looking for the desired key, and returns a leaf (which will holds the desired key if the desired key is in the tree), its parent, and its grandparent.

To insert a key, replace the leaf returned by the search method with a subtree containing an internal node with two leaf children, one with the new desired key, and one with the key of the leaf being replaced (See Figure 6.2 (c)). An MCMS operation atomically executes this exchange while guaranteeing the parent is unmarked (hence, not deleted).

To delete a key, the grandparent pointer to the parent is replaced by a pointer to the deleted node's brother (See Figure 6.2 (d)), atomically with setting the parent mark bit on, marking it as deleted, and guarding against concurrent (or later) changes to its child pointers. An MCMS instruction also ensures that the grandparent is unmarked, and that the parent's child pointers retain their expected value during the deletion.

6.6 Fall-back Execution for Failed Transactions

Formally, transactions are never guaranteed to commit successfully, and spurious failures may occur infinitely without any concrete reason. Our experimental results show that such repeated failures are not observed in practice. Nevertheless, we implemented several fall-back avenues that general algorithms using MCMS may benefit from, and we briefly overview them here. Each transaction is attempted several times before switching

```
1. <Node*,Node*,Node*>  Search (int key) {
2.   InternalNode *grandParent, *parent;
3.   Node* leaf = root;
4.   While (leaf points to an InternalNode) {
5.      grandParent = parent;
6.      parent = leaf;
7.      if (key <   (*leaf).key)
8.         leaf = (*parent).left;
9.      else
10.        leaf = (*parent).right;
11.  retrun  (grandParnet, parent, leaf); } }
12.
13. bool Insert (int key) {
14.   InternalNode *parent, *newInternal;
15.   LeafNode *leaf, *newSibling;
16.   LeafNode* newLeaf = new LeafNode(key);
17.   While (true) {
18.      <_ , parent, leaf> = Search(key);
19.      if ((*leaf).key == key)
20.         return false;  // key already exists
21.      newSibling = new LeafNode((*leaf)->key);
22.      if ((*newSibling).key > (*newLeaf).key)
23.        newInternal =
             new InternalNode(newLeaf,
             newSibling, (*newSibling).key);
24.    else
25.       newInternal =
            new InternalNode(newSibling,
            newLeaf, (*newLeaf).key);
   // find address of pointer from parent to leaf:
26.    Node **childPointer;
27.    if ((*newInternal).key < (*parent).key)
28.       childPointer = &((*parent).left);
29.    else
30.       childPointer = &((*parent).right);
   // compare parent mark to 0 and
   // CAS parent pointer to point to newIntren:
31.    if (MCMS (<&((*parent).mark) , 0>,
          <childPointer, leaf, newInternal>))
32.       return true;    // successfully inserted.
33.    } }
```

```
34. bool Delete(int key) {
35.   InternalNode *grandParent, *parent;
36.   LeafNode*  leaf;
37.   While (true) {
38.      <grandParent, parent, leaf> =
           search(key);
39.      if ((*leaf).key != key)
40.         return false; // key doesn't exist
41.      Node** leafPointer;  // the pointer from
                                 //parent to leaf.
42.      Node** sibling;    // the other child pointer
                             //of parent.
43.      if ((*parent).key > (*leaf).key) {
44.         leafPointer = &((*parent).left);
45.         sibling = &((*parent).right); }
46.      else {
47.         leafPointer = &((*parent).right);
48.         sibling = &((*parent).left); }
49.      Node* siblingVal = *sibling;
50.      Node** pPointer;  // the pointer from
                           // grandParent to parent.
51.      if ((*grandParent).key > (*parent).key)
52.         pPointer = &((*grandParent).left);
53.      else
54.         pPointer = &((*grandParent).right);
      // compare gp mark = 0,
      // compare leafPointer points to leaf
      // compare sibling points to siblingVal
      // CAS gp to point to sibling
      // CAS parent to be marked
55.      if (MCMS(<&((*grandParent).mark), 0>,
             <leafPointer, leaf>,
             <sibling, siblingVal>,
             <pPointer, parent, siblingVal>,
             <&((*parent).mark), 0, 1>))
56.            return true;   // successfully deleted.
57.      } }
58.
59. bool Contains(int key) {
60.   LeafNode* leaf;
61.   <_, _, leaf> = search(key);
62.   return (*leaf).key == key; }
```

Figure 6.3: The Tree Algorithm

to a fall-back execution path. The number of retries is a parameter that can be tuned, denoted *MAX_FAILURES*.

6.6.1 Using Locking for the Fall-back Path

The idea of trying to execute a code snippet using a transaction, and take a lock if the transaction fails to commit, is known as *lock elision*. We add a single integer field, denoted *lock* to the data structure. In the HTM implementation of MCMS, before calling XEND the `lock` field is read, and compared to zero. If the lock is not zero, XABORT is called. This way, if any thread acquires the lock (by CASing it to one) all concurrent transactions will fail. If an MCMS operation fails to commit a transaction MAX_FAILURES times, the thread tries to obtain the `lock` by repeatedly trying to CAS it from 0 to 1 until successful. The MCMS is then executed safely. When complete, the thread sets the `lock` back to 0.

Our implementation of lock-elision is slightly different than that of traditional lock-elision. As described in Section 6.3.1, after each transaction abort we compare each address to its expected value, and thus in many cases we can return false after a failure without using any locking or transactions at all.

6.6.2 Non-Transactional MCMS Implementation as a Fall-back Path

Another natural fall-back path alternative is to use the non-transactional MCMS implementation of Harris et al., described in Section 6.3.2. While this implementation was proposed for implementing the MCMS on a platform that does not support HTM, it may also be used as a fall-back when hardware transactions repeatedly fail. Several threads can execute this implementation of the MCMS operation concurrently. However, as mentioned in Section 6.3.2, during the execution of the MCMS operations, the target addresses temporarily store a pointer to a special operation descriptors instead of their "real" data. This requires a careful test for any read of the data structure, which unfortunately comes with a significant overhead.

We experimented with several different mechanisms to guarantee that each read of the data structure is safe. The first mechanism is to always execute the same read procedure that is applied when MCMS is implemented without TM, as described in [HFP02]. The second alternative is to use transactions for the reads as well. Instead of doing a simple read, we can put the read in a transaction, and before executing the transaction XEND, read a `lock` field and abort if it does not equal zero. Each thread that executes a non-transactional MCMS increments the `lock` before starting it, and decrements the `lock` once the MCMS is completed. The reads can be packed into a transaction in different granularity. One may place each read in a different transaction and add a read of the `lock` field; but one may also pack all the reads up to an MCMS into a single transaction and add a single read of the `lock`. We tried a few granularities and found out that packing five reads into a transaction was experimentally optimal.

6.6.3 A Copying-Based Fall-back path

A third avenue for implementing a fall-back for failing transactions is copying-based. Again, a `lock` field is added. Additionally, a single global pointer which points to the data structure is added. When accessing the data structure an indirection is added: the external pointer is read, and the operation is applied to the data structure pointed by it. As usual, each HTM based MCMS operation compares the `lock` to zero before committing, and aborts if the lock is not zero.

Unlike previous solutions, in the copying fall-back implementation the lock is permanent, and the *current copy* of the data structure becomes immutable. After setting the lock to one, the thread creates a complete copy of the data structure, and applies the desired operation on that copy. Other threads that observes the `lock` is set act similarly. The new copy is associated with a new lock that is initiated to zero. Then, a CAS attempts an atomic change of the global pointer to point to the newly created copy instead of the original copy of the data structure (from which it copied the data). Afterwards, execution will continue as usual on the new copy, until the next time a thread will fail to commit a transaction MAX_FAILURES times.

6.7 Performance

In this section we present the performance of the different algorithms and variants discussed in this work. In Figures 6.4 and 6.5 we present the throughput of the list and tree algorithms compared against their lock-free counterparts. Each line in each chart represent a different variant of an algorithm. In the micro-benchmarks tested each thread executes either 50% INSERT and 50% DELETE operations, 20% INSERT, 10% DELETE, and 70% CONTAINS operations, or a 100% CONTAINS operations. The operation keys are integers that are chosen randomly and uniformly in a range of either 1–32, 1–1024, or 1–1048576. Before starting each test, a data structure is pre-filled to 50% occupancy with randomly chosen keys from the appropriate range. Deleted nodes were not reclaimed.

In all our experiments, we set the number of MAX_FAILURES to be 7. With this setting, we see MCMS operations that need to complete execution in the fallback path. Reducing this parameter to 6 causes a (slight) performance degradation in a few scenarios. We also tested the number of total MCMS transaction aborts, and the number of MCMS operations that were completed in the fall-back path, when valid. Higher MAX_FAILURES values yield similar performance, but with almost no executions in the fall-back path. This makes the measurements less informative, so 7 was chosen.

The measurements were taken on an Intel Haswell i7-4770, with 4 dual cores (overall 8 hardware threads) and 6MB cache size, running Linux Suse. Haswell processors with more cores that support HTM are currently unavailable. The algorithms were written in C++ and compiled with GNU C++ compiler version 4.5. In each chart we present

nine algorithms, as specified in the figures.

The fastest performing algorithm is always the HTM-based MCMS without any fall-back path. On a range of 1048576 available keys, this list algorithm outperforms Harris's by 30-60%; on a range of 1024 available keys, it outperforms by 40-115%, and on a range of 32 keys, it outperforms by 20-55%. The tree algorithm outperforms the tree of Ellen et al. by 6-37%. For both data structures the lock-based fall-back path adds very little overhead, and the corresponding algorithms trail behind the algorithms without the fall-back path by 1-5%.

The copying fall-back path algorithm also performs excellently for the linked-list. On average, it performs the same as the lock-based algorithm, with a difference smaller than half a per cent. This makes the HTM MCMS algorithm with the copying fall-back path the fastest lock-free linked-list by a wide margin. The copying tree algorithm is not as good, trailing behind the pure HTM algorithm by about 10%. Yet this algorithm still beats the lock-free algorithm of Ellen et al. in all number of threads for all benchmarks, excluding, surprisingly, the benchmark of 100% contains for 32 and 1024 available keys. This is surprising, because in this benchmark MCMS is not executed at all. We suspect that the reason is the fact that the search method of the copying based tree receives the root of the tree as an input parameter. In the pure HTM algorithm, the root is known at compile time to be final (never changed once it is allocated), which could allow the compiler to optimize its reading.

Using a CAS-based MCMS fall-back path does not work as well as the copying or the lock-based fall-back alternatives. For the list, packing five reads into a transaction yields reasonable performance, usually beating Harris's linked list for a lower number of threads and a larger range of keys (up 20% faster), but trailing up to 40% behind it for 8 threads in 32 or 1024 keys when the micro-benchmark is 50% INSERTS and 50% DELETES. Packing all the reads into a single transaction works quite badly for the longer lists, were the large number of reads causes the vast majority of reading transactions to abort. It also works badly for a 32 keys range when the benchmark is 50% INSERTS and 50% DELETE. The high number of MCMS transactions combined with read transactions results in poor performance. For the tree, is at times better and at times worse than the tree of Ellen et al., and the difference is up to 10%. This holds for the option of packing all the reads into a single transaction as well.

Aborts and fall-back executions. As expected from the performance results, the number of MCMS executions that are completed in the fall-back path is low. For instance, a copying of a list or a tree of 1048576 keys, which one would expect to be costly, never takes place. On the other end, In a list of 32 keys, for 8 threads, in the micro-benchmark of 50% INSERTS and 50% DELETES, copying is executed once every 5000 list operations. In a list of 1024, it is never executed. In a tree of 32 keys when executing with 8 threads, on the 50% INSERTS and 50% DELETES micro-benchmark, a copying occurs once every 1730 tree operations, and once every 54000 operations for a

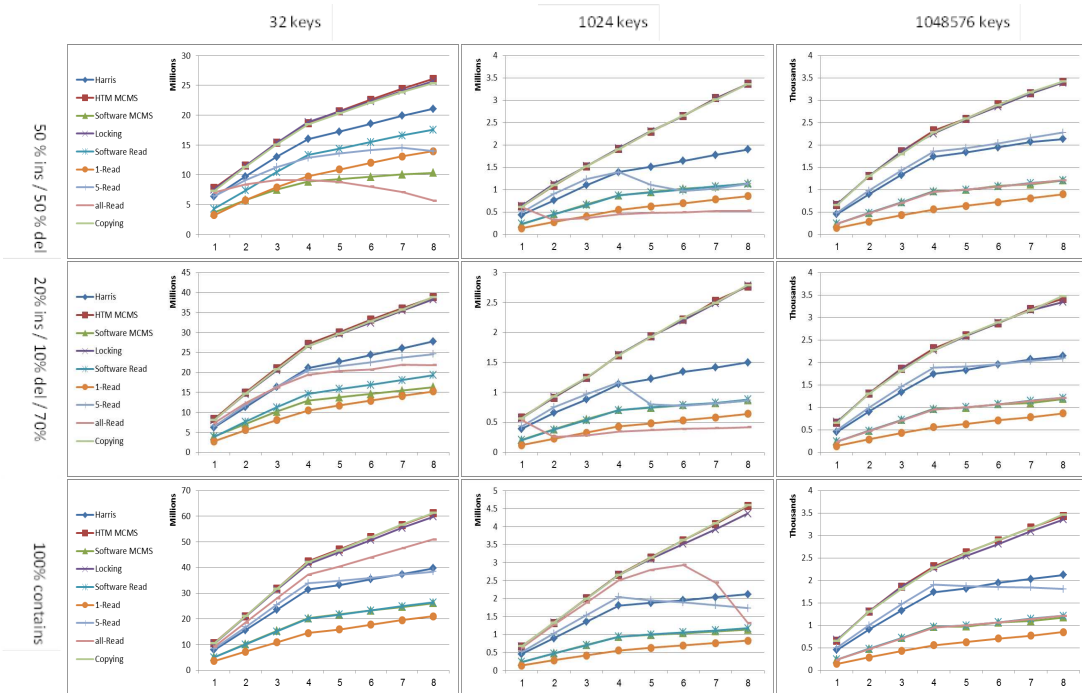

Figure 6.4: MCMS-based lists vs. Harris's linked-list. The x-axis represents the number of threads. The y-axis represents the total number of operations executed in a second (in millions key ranges 32 and 1024, in thousands for key range 1048576.

tree of 1024 keys running 8 threads. In general, note that once an MCMS is executed in the fall-back path, other MCMS's may abort as a result of the *lock* field being set.

6.8 Conclusions

In this work we proposed to use MCMS, a variation of MCAS operation, as an intermediate interface that encapsulates HTM on platforms where HTM is available, and can also be executed in a non-transactional manner when HTM is not available. We established the effectiveness of the MCMS abstraction by presenting two MCMS-based algorithms, for a list and for a tree. When HTM is available, these algorithms outperform their lock-free counterparts. We have also briefly discussed possible "fall-back" avenues for when transactions repeatedly fail. We have implemented these alternatives, and explored their performance overhead.

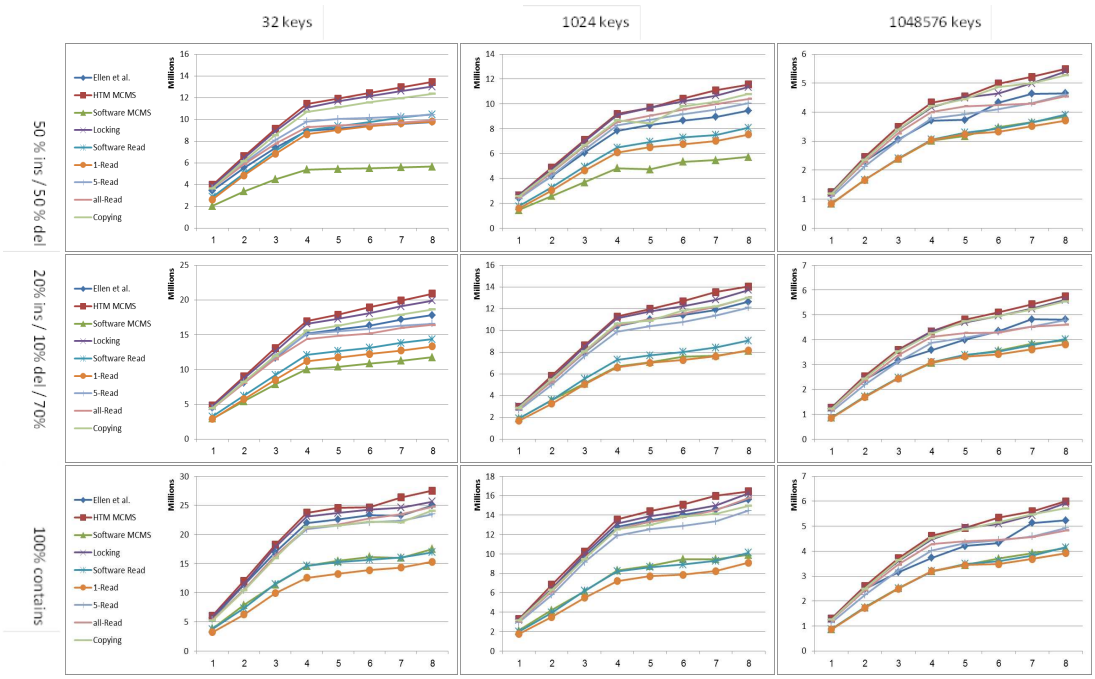

Figure 6.5: MCMS-based trees vs. the BST of Ellen et al. The x-axis represents the number of threads. The y-axis represents millions of operations executed per second.

Chapter 7

Conclusions

In this thesis we presented a significant contribution to the understanding and design of concurrent data structures, and particularly of wait-free data structures. Prior to our work, wait-free data structures were considered notoriously hard to design. While wait-free universal constructions have been known for decades, they only provide theoretical designs, which are too slow for use in practice.

In fact, prior to our work, the only available practical wait-free data structures were for the queue and stack abstract data types. Wait-free stack and queue structures are not easy to design, but they only provide limited parallelism, i.e., a limited number of contention points (the head of the stack, and the head and the tail of the queue).

This dissertation started with a novel design of the first practical wait-free linked-list. This list is the first wait-free data structure that can scale to support a large number of concurrent operations, thanks to the fact that it does not have a limited number of contention points. Using the fast-path-slow-path methodology, we successfully created a wait-free linked-list that is just a few percents slower than the best lock-free linked-list.

Our study continued with a generalization of the technique, which offers an easy wait-free simulation of lock-free data structures. Using our proposed simulation, it becomes easy to design many wait-free data structures, while paying only a small price in the overall throughput. As concrete examples, we used our general simulation to derive fast wait-free skiplist and binary search tree.

Both the wait-free linked-list design and the general wait-free simulation employed a help mechanism in which some threads help other threads to complete their work. The help mechanism was the key feature that allowed our constructions to be wait-free. The next study in this dissertation explored the interaction between wait-freedom and help. We started by formalizing the notion of help. Next, we presented conditions under which concurrent data structures must use help to obtain wait-freedom. Natural examples that satisfy these conditions are a wait-free queue or a wait-free stack.

This contribution is a lower-bound type of result, which sheds light on a key element that implementations of certain object types must contain. As such, we hope it will have a significant impact on both research and design of concurrent data structures.

We believe it can lead to modularity in designs of implementations that are shown to require a helping mechanism in order to be wait-free, by allowing to pinpoint the place where help occurs.

With regard to help, a remaining open problem is to find a full characterizations of the abstract data types that require help to obtain wait-freedom. As a possible interim step towards the goal of full characterization, we conjecture that perturbable objects cannot have wait-free help-free implementations when using only READ and WRITE primitives.

In addition to providing better understanding and implementations of wait-free data structures, our research also focused on extending the interface that such data structures often implement. We added support for lock-free and wait-free iterators for data structures that implement set operations and that adhere to certain constraints. We used our techniques specifically to obtain wait-free iterators for linked-lists and skiplists. The proposed technique excels in imposing a very low overhead on concurrent operations that are applied to the data structure while some threads are iterating over it.

Finally, we were interested in the question whether lock-free and wait-free data structures could benefit from the use of hardware transactional memory. Our study answered this question in the affirmative. We demonstrated that by using an intermediate interface that encapsulates HTM, but can also be executed in a non-transactional manner, lock-free data structures can be made not only faster, but also simpler to design. As examples, we designed a faster lock-free linked-list and a tree.

Appendix A

A Full Java Implementation of the Wait-Free Linked List

In this appendix, we give a full Java implementation for the basic wait-free linked-list. This basic implementation also uses a VersionedAtomicMarkableReference, in which the reference is associated with a version number for avoiding an ABA problem. A solution and Java code with no versioning requirement is specified in Appendix B. The solution there only employs the standard AtomicMarkableReference. The source for the class VersionedAtomicMarkableReference which implements such versioned pointers is also given right after the WFList. It is obtained by slightly modifying the code of the AtomicMarkableReference of Doug Lea.

```
2
import java.util.concurrent.atomic.AtomicReferenceArray;
import java.util.concurrent.atomic.AtomicBoolean;
import java.util.concurrent.atomic.AtomicLong;

public class WFList {
  enum OpType {insert, search_delete, execute_delete, success, failure,
determine_delete, contains};

  private class Window {
    public final Node pred, curr;
    public Window(Node p, Node c) { pred = p; curr = c; }
  }

  private class Node {
    public final int key;
    public final VersionedAtomicMarkableReference<Node> next;
    public final AtomicBoolean d;
    public Node (int key) {
      next = new VersionedAtomicMarkableReference<Node>(null, false);
      this.key = key; d = new AtomicBoolean(false);
    }
```

```
}

private class OpDesc {
  public final long phase; public final OpType type;
  public final Node node; public final Window searchResult;
  public OpDesc (long ph, OpType ty, Node n, Window sResult) {
    phase = ph; type = ty; node = n; searchResult = sResult;
  }
}
private final Node head, tail;
private final AtomicReferenceArray<OpDesc> state;
private final AtomicLong currentMaxPhase; // used in maxPhase method

public WFList () {
  currentMaxPhase = new AtomicLong(); // used in maxPhase method
  currentMaxPhase.set(0);
  head = new Node(Integer.MIN_VALUE); // head's key is smaller than all
      the rests'
  tail = new Node(Integer.MAX_VALUE); // tail's key is larger than all
      the rests'
  head.next.set(tail, false);          // init list to be empty
  state = new AtomicReferenceArray<OpDesc>(Test.numThreads);
  for (int i = 0; i < state.length(); i++)  // state entry for each
      thread
    state.set(i, new OpDesc(0, OpType.success, null,null));
}

public boolean insert(int tid, int key) {
  long phase = maxPhase(); // getting the phase for the op
  Node newNode = new Node(key); // allocating the node
  OpDesc op = new OpDesc(phase, OpType.insert, newNode,null);
  state.set(tid, op); // publishing the operation.
  help(phase);  // when finished - no more pending operation with lower
       or equal phase
  return state.get(tid).type == OpType.success;
}

public boolean delete(int tid, int key) {
  long phase = maxPhase(); // getting the phase for the op.
  state.set(tid, new OpDesc(phase, OpType.search_delete, new Node(key),
      null));// publishing.
  help (phase); // when finished - no more pending operation with lower
       or equal phase
  OpDesc op = state.get(tid);
  if (op.type == OpType.determine_delete)
    // compete on the ownership of deleting this node
    return op.searchResult.curr.d.compareAndSet(false, true);
  return false;
}
```

```
  private Window search(int key, int tid, long phase) {
    Node pred = null, curr = null, succ = null;
    boolean[] marked = {false};  boolean snip;
    retry : while (true) {
      pred = head;
      curr = pred.next.getReference(); // advancing curr
      while (true) {
        succ = curr.next.get(marked); // advancing succ and reading curr.
            next's mark
        while (marked[0]) { // curr is logically deleted a should be
            removed
          // remove a physically deleted node :
          snip = pred.next.compareAndSet(curr, succ, false, false);
          if (!isSearchStillPending(tid,phase))
            return null; // to ensure wait-freedom.
          if (!snip) continue retry; // list has changed, retry
          curr = succ; // advancing curr
          succ = curr.next.get(marked); // advancing succ and reading curr
              .next's mark
        }
        if (curr.key >= key)  // the curr.key is large enough - found the
            window
          return new Window(pred, curr);
        pred = curr; curr = succ; // advancing pred  & curr
      }
    }
  }

  private void help(long phase) {
    for (int i = 0; i < state.length(); i++) {
      OpDesc desc = state.get(i);
      if (desc.phase <= phase) { // help all pending operations with a
          desc.phase <= phase
        if (desc.type == OpType.insert) { // a pending insert operation.
          helpInsert(i, desc.phase);
        } else if (desc.type == OpType.search_delete
            || desc.type == OpType.execute_delete) { // a pending delete
                operation
          helpDelete(i, desc.phase);
        } else if (desc.type == OpType.contains) { helpContains(i, desc.
phase); }
      }
    }
  }

  private void helpInsert(int tid, long phase) {
    while (true) {
      OpDesc op = state.get(tid);
      if (!(op.type == OpType.insert && op.phase == phase))
        return; // the op is no longer relevant, return
```

```
      Node node = op.node; // getting the node to be inserted
      Node node_next = node.next.getReference(); //must read node_next
          before search
      Window window = search(node.key,tid,phase); //search a window to
          insert the node into
      if (window == null) // can only happen if operation is no longer
          pending
        return;
      if (window.curr.key == node.key) { // key exists - chance of a
          failure
        if ((window.curr == node) || (node.next.isMarked())) {
          // the node was already inserted - success
          OpDesc success = new OpDesc(phase, OpType.success, node, null);
          if (state.compareAndSet(tid, op, success))
            return;
        }
        else { // the node was not yet inserted - failure
          OpDesc fail = new OpDesc(phase, OpType.failure, node, null);
          // CAS may fail if search results are obsolete
          if (state.compareAndSet(tid, op, fail))
            return;
        }
      }
      else {
        if (node.next.isMarked()) { // node was already inserted and
            marked (=deleted)
          OpDesc success = new OpDesc(phase, OpType.success, node, null);
          if (state.compareAndSet(tid, op, success))
            return;
        }
        int version = window.pred.next.getVersion(); // read version for
            CAS later.
        OpDesc newOp = new OpDesc(phase, OpType.insert, node, null);
    // the following prevents another thread with obsolete search results
        to report failure:
        if (!state.compareAndSet(tid, op, newOp))
          continue; // operation might have already reported as failure
        node.next.compareAndSet(node_next, window.curr, false, false);
        // if successful - than the insert is linearized here :
        if (window.pred.next.compareAndSet(version, node.next.getReference
() , node, false, false)) {
          OpDesc success = new OpDesc(phase, OpType.success, node, null);
          if (state.compareAndSet(tid, newOp, success))
            return;
        }
      }
    }

  private void helpDelete(int tid, long phase) {
```

```
    while (true) {
      OpDesc op = state.get(tid);
      if (!((op.type == OpType.search_delete || op.type == OpType.
execute_delete)
          && op.phase==phase))
        return; // the op is no longer relevant, return
      Node node = op.node;  // the node holds the key we want to delete
      if (op.type == OpType.search_delete) {  // need to search for the
          key
        Window window = search(node.key,tid,phase);
        if (window==null)
          continue; // can only happen if operation is no longer the same
              search_delete
        if (window.curr.key != node.key) {
          // key doesn't exist - failure
          OpDesc failure = new OpDesc(phase, OpType.failure, node, null);
          if (state.compareAndSet(tid, op, failure))
            return;
        }
        else {
          // key exists - continue to executre_delete
          OpDesc found = new OpDesc(phase, OpType.execute_delete, node,
window);
          state.compareAndSet(tid, op, found);
        }
      }
      else if (op.type == OpType.execute_delete) {
        Node next = op.searchResult.curr.next.getReference();
        if (!op.searchResult.curr.next.attemptMark(next, true)) // mark
            the node
          continue; // will continue to try to mark it, until it is marked
        search(op.node.key,tid,phase); // will physically remove the node
        OpDesc determine =
          new OpDesc(op.phase, OpType.determine_delete, op.node, op.
searchResult);
        state.compareAndSet(tid, op, determine);
        return;
      }
    }
  }

  public boolean contains(int tid, int key) {
    long phase = maxPhase();
    Node n = new Node(key);
    OpDesc op = new OpDesc(phase, OpType.contains, n, null);
    state.set(tid, op);
    help(phase);
    return state.get(tid).type == OpType.success;
  }
```

```
  private void helpContains(int tid, long phase) {
    OpDesc op = state.get(tid);
    if (!((op.type == OpType.contains) && op.phase==phase))
      return; // the op is no longer relevant, return
    Node node = op.node;  // the node holds the key we want to find
    Window window = search(node.key, tid, phase);
    if (window == null)
      return; // can only happen if operation is already complete.
    if (window.curr.key == node.key) {
      OpDesc success = new OpDesc(phase, OpType.success, node, null);
      state.compareAndSet(tid, op, success);
    }
    else {
      OpDesc failure = new OpDesc(phase, OpType.failure, node, null);
      state.compareAndSet(tid, op, failure);
    }
  }

  private long maxPhase() {
    long result = currentMaxPhase.get();
    // ensuring maxPhase will increment before this thread next operation
        :
    currentMaxPhase.compareAndSet(result, result+1);
    return result;
  }

  private boolean isSearchStillPending(int tid, long ph) {
    OpDesc curr = state.get(tid);
    return (curr.type == OpType.insert || curr.type == OpType.
search_delete
        || curr.type == OpType.execute_delete || curr.type == OpType.
contains) &&
      curr.phase == ph; // the operation is pending with a phase lower
          than ph.
  }
}

public class VersionedAtomicMarkableReference<V>  {

    private static class ReferenceBooleanTriplet<T> {
        private final T reference;
        private final boolean bit;
        private final int version;
        ReferenceBooleanTriplet(T r, boolean i, int v) {
            reference = r; bit = i; version = v;
        }
    }

    private final AtomicReference<ReferenceBooleanTriplet<V>>  atomicRef;
```

```
    public VersionedAtomicMarkableReference(V initialRef, boolean
initialMark) {
        atomicRef = new AtomicReference<ReferenceBooleanTriplet<V>> (new
ReferenceBooleanTriplet<V>(initialRef, initialMark,0));
    }

    public V getReference() {
        return atomicRef.get().reference;
    }

    public boolean isMarked() {
        return atomicRef.get().bit;
    }

    public V get(boolean[] markHolder) {
        ReferenceBooleanTriplet<V> p = atomicRef.get();
        markHolder[0] = p.bit;
        return p.reference;
    }

    public boolean weakCompareAndSet(V       expectedReference,
                                     V       newReference,
                                     boolean expectedMark,
                                     boolean newMark) {
        ReferenceBooleanTriplet<V> current = atomicRef.get();
        return  expectedReference == current.reference &&
            expectedMark == current.bit &&
            ((newReference == current.reference && newMark == current.bit)
 ||
             atomicRef.weakCompareAndSet(current,
                                     new ReferenceBooleanTriplet<V>(
newReference,
                                                                    newMark,
current.version+1)));
    }

    public boolean compareAndSet(V       expectedReference,
                                 V       newReference,
                                 boolean expectedMark,
                                 boolean newMark) {
        ReferenceBooleanTriplet<V> current = atomicRef.get();
        return  expectedReference == current.reference &&
            expectedMark == current.bit &&
            ((newReference == current.reference && newMark == current.bit)
 ||
             atomicRef.compareAndSet(current,
                                     new ReferenceBooleanTriplet<V>(
newReference,
                                                                    newMark,
current.version+1)));
```

```
    }

    public void set(V newReference, boolean newMark) {
        ReferenceBooleanTriplet<V> current = atomicRef.get();
        if (newReference != current.reference || newMark != current.bit)
            atomicRef.set(new ReferenceBooleanTriplet<V>(newReference,
newMark, current.version+1));
    }

    public boolean attemptMark(V expectedReference, boolean newMark) {
        ReferenceBooleanTriplet<V> current = atomicRef.get();
        return  expectedReference == current.reference &&
            (newMark == current.bit ||
             atomicRef.compareAndSet
             (current, new ReferenceBooleanTriplet<V>(expectedReference,
                                                      newMark, current.version+1)
));
    }

    public int getVersion()
    {
      return atomicRef.get().version;
    }

    public boolean compareAndSet(int version, V expectedReference, V
newReference, boolean expectedMark, boolean newMark) {
       ReferenceBooleanTriplet<V> current = atomicRef.get();
         return  expectedReference == current.reference &&
            expectedMark == current.bit && version == current.version &&
            ((newReference == current.reference && newMark == current.bit)
 ||
             atomicRef.compareAndSet(current,
                                     new ReferenceBooleanTriplet<V>(
newReference,
                                                                    newMark,
current.version+1)));
    }
}
```

Appendix B

Avoiding Versioned Pointers for the Wait-Free Linked List

In the implementation of the basic wait-free linked-list, we used a versioned pointer at the next field of each node. While such a solution is the simplest, it requires the use of a wide CAS. In this appendix we provide a way to avoid the use of versioned pointers. This solution only uses regular pointers with a single mark bit, similarly to the original lock-free algorithm by Harris (in Java, this mark bit is implemented via the AtomicMarkeableReference class). In the basic implementation, we used the CAS of Line 140 (the line notations correspond to the code in Appendix A), when inserting a new node into the list. As described in Section 2.3, we need it to avoid the following ABA problem. Suppose Thread T_1 is executing an insert of the key k into the list. It searches for a location for the insert, it finds one, and gets stalled just before executing Line 140. While T_1 is stalled, T_2 inserts a different k into the list. After succeeding in that insert, T_2 tries to help the same insert of k that T_1 is attempting to perform. T_2 finds that k already exists and reports failure to the state descriptor. This should terminate the insertion that T_1 is executing with a failure report. But suppose further that the other k is then removed from the list, bringing the list back to exactly the same view as T_1 saw before it got stalled. Now T_1 resumes and the CAS of Line 140 actually succeeds. This course of events is bad, because a key is inserted into the list while a failure is reported about this insertion. Instead of using a versioned pointer to solve this problem, we can use a different path. We will mark the node that is about to be inserted as logically deleted. This way, even if the ABA problem occurs, the node will never appear in the list. Namely, when failure is detected, we can mark the next pointer of the node we failed inserting. While this won't prevent the node from physically being inserted into the list because of the described ABA problem, it will only be inserted as a logically deleted node, and will be removed next time it is traversed, without ever influencing the logical state of the list. However, marking the next field of the node requires care. Most importantly, before we mark the node, we must be certain that it was not already inserted to the list (by another thread), and when we mark it, we ought

to be sure that the operation will be correctly reported as failure (even if the marked node was later physically inserted). To ensure this, we use a gadget denoted *block*. The block is a node with two extra fields - the threadID and the phase of the operation it is meant to fail. The procedure for a failing insertion is thus as follows. Say an operation for inserting a node with key 4 is in progress. This node would be called the *failing node*. Upon searching the list, a node that contains key 4 was found. This node is the *hindering node*.

* Using a CAS, a block will be inserted right after the hindering node.
* The failing node's next field will be marked.
* The state of the operation will be changed to failure.
* The block will be removed.

By ensuring that a node right before a block (this is the hindering node) cannot be logically deleted, and that a new node cannot be inserted between the hindering node and the block, it is guaranteed that when marking the failing node as deleted, a failing node was not yet inserted into the list (since the block is still there, and thus also the hindering node). The block's next field will never be marked, and will enable traversing the list. The block key will be set to a value that is lower than all possible keys in the list (can be the same as the head key). This serves two purposes: first, it allows to differentiate between a regular node and a block (in a strongly typed language such as Java, this is done differently), and second, it allows the contains method to work unchanged, without being aware of the existence of blocks, since it will always traverse past a (node/block) with a smaller key than the one searched for. In Java, the block looks like this :

```
2
private class Block extends Node {
    int tid; long phase;
    public Block (int tid, long phase) {
      super(Integer.MIN_VALUE); this.tid = tid; this.phase = phase;
    }
  }
```

Upon reaching a block, we need to make sure that the failing node's next field is marked, report the operation as failed, and then remove the block. This is done in the removeBlock method :

```
2
private void removeBlock(Node pred, Block curr) {
    OpDesc op = state.get(curr.tid);
    // both loops are certain to finish after test.numofThreads iterations
        (likely sooner)
```

```
    while (op.type == OpType.insert && op.phase == curr.phase) {
      // mark the node that its insertion is about to be set to failure
      while (!op.node.next.attemptMark(op.node.next.getReference(), true))
;
      OpDesc failure = new OpDesc(op.phase, OpType.failure, op.node, null)
;
      state.compareAndSet(curr.tid, op, failure); // report failure
      op = state.get(curr.tid);
    }
    // physically remove the block (if CAS fails, then the block was
        already removed)
    pred.next.compareAndSet(curr, curr.next.getReference(), false, false);
  }
```

Note that since the presence of a block doesn't allow certain modifications to the list until it is removed (such as deleting the hindering node), we must allow all threads to help remove a block in order to obtain wait-freedom (or even lock-freedom). Accordingly, the search method plays a role in removing blocks when it traverses them, similarly to the role it plays in physically removing marked nodes. Thus, the loop in the search method to remove marked nodes (lines 73-81) should be modified to :

```
2
while (marked[0] || curr instanceof Block) {
          if (curr instanceof Block) {
            removeBlock(pred, (Block)curr);
          }
          else {
            // remove a physically deleted node :
            snip = pred.next.compareAndSet(curr, succ, false, false);
            if (!isSearchStillPending(tid,phase))
              return null; // to ensure wait-freedom
            if (!snip) continue retry; // list has changed, retry
          }
          curr = succ; // advancing curr
          succ = curr.next.get(marked); // advancing succ and reading curr
              .next's mark
        }
```

As mentioned above, we should also make sure that the hindering node will not be marked while the block is still after it. To ensure that, we modify the part in the helpDelete method, that handles the execute_delete OpType (lines 172-181) to be :

```
2
else if (op.type == OpType.execute_delete) {
        Node next = op.searchResult.curr.next.getReference();
        if (next instanceof Block) { // cannot delete a node while it is
            before a block
          removeBlock(op.searchResult.curr, (Block)next);
          continue;
```

```
        }
        if (!op.searchResult.curr.next.attemptMark(next, true)) // mark
            the node
          continue; // will continue to try to mark it, until it is marked
        search(op.node.key,tid,phase); // will physically remove the node
        OpDesc determine =
          new OpDesc(op.phase, OpType.determine_delete, op.node, op.
searchResult);
        state.compareAndSet(tid, op, determine);
        return;
      }
```

The only thing left is to modify the helpInsert method, so that it will insert a block upon failure. Some additional care is needed since, in the basic implementation, observing that the node to be inserted is marked was an indication that the operation succeeded. Now, it can only be used as such an indication if there is not a hindering node with block after it that is trying to fail that same operation. Once the block is removed, the fact that the node's next field is marked can indeed be used for an indication of success, since if it was marked because of a block, the fact that the block was already removed tells us that the operation was already reported as failing in the state array, and there is no danger it will be mistakenly considered a success. The modified helpInsert method is as follows :

2

```
private void helpInsert(int tid, long phase) {
    while (true) {
      OpDesc op = state.get(tid);
      if (!(op.type == OpType.insert && op.phase == phase))
        return; // the op is no longer relevant, return
      Node node = op.node; // getting the node to be inserted
      Node node_next = node.next.getReference(); //must read node_next
          before search
      if (node_next instanceof Block)
      {
        removeBlock(node, (Block)node_next);
        continue;
      }
      Window window = search(node.key,tid,phase);//search a window to
          insert the node into
      if (window == null) // can only happen if operation is no longer
          pending
        return;
      if (window.curr.key == node.key) {// key exists - chance of a
          failure
        if ((window.curr == node) || (node.next.isMarked())) {
          Node window_succ = window.curr.next.getReference();
          if (window_succ instanceof Block) {
            removeBlock(window.curr, (Block)window_succ);
```

```
            continue;
          }
          // the node was already inserted - success
          OpDesc success = new OpDesc(phase, OpType.success, node, null);
          if (state.compareAndSet(tid, op, success))
            return;
        }
        else { // the node was not yet inserted - failure
          Node window_succ = window.curr.next.getReference();
          Block block = new Block(tid, op.phase);
          block.next.set(window_succ, false);
          // linearization point for failure :
          if (window.curr.next.compareAndSet(window_succ, block, false,
false))
            removeBlock(window.curr, block); // will complete the
                operation
        }
      }
      else {
        if (node.next.isMarked()) { // node was already inserted and
            marked (=deleted)
          OpDesc success = new OpDesc(phase, OpType.success, node, null);
          if (state.compareAndSet(tid, op, success))
            return;
        }
        OpDesc newOp = new OpDesc(phase, OpType.insert, node, null);
    // the following prevents another thread with obsolete search results
        to report failure:
        if (!state.compareAndSet(tid, op, newOp))
          continue; // operation might have already reported as failure
        node.next.compareAndSet(node_next, window.curr, false, false);
        // if successful - than the insert is linearized here :
        if (window.pred.next.compareAndSet(node.next.getReference(), node,
 false, false)) {
          OpDesc success = new OpDesc(phase, OpType.success, node, null);
          if (state.compareAndSet(tid, newOp, success))
            return;
        }
      }
    }
  }
```

The linearization point of a failing insert operation is now moved to the CAS that inserts the block. The list is still wait-free, since each thread that comes upon a block can always remove it in a bounded number of steps.

Appendix C

The Full Code of the Fast-Path-Slow-Path Extension for the Wait-Free Linked-List

```
2
  import java.util.concurrent.atomic.AtomicLong;
  import java.util.concurrent.atomic.AtomicReference;
  import java.util.concurrent.atomic.AtomicReferenceArray;
  import java.util.concurrent.atomic.AtomicBoolean;

  public class FPSPList implements {
    enum OpType {insert, search_delete, execute_delete, success, failure,
  determine_delete, contains, update_approximation};

    private class Window {
      public final Node pred, curr;
      public Window(Node p, Node c) { pred = p; curr = c; }
    }

    private class Node {
      public final int key;
      public VersionedAtomicMarkableReference<Node> next;
      public final AtomicBoolean successBit;
      public Node (int key) {
        this.key = key;
        successBit = new AtomicBoolean(false);
        // the next field will be initialized later.
      }
    }

    private class OpDesc {
      public final long phase; public final OpType type;
      public final Node node; public final Window searchResult;
```

```
  public OpDesc (long ph, OpType ty, Node n, Window sResult) {
    phase = ph; type = ty; node = n; searchResult = sResult;
  }
}

class HelpRecord {
  int curTid; long lastPhase; long nextCheck;
  public HelpRecord() { curTid = -1; reset(); }
  public void reset() {
    curTid = (curTid + 1) % Test.numThreads;
    lastPhase = state.get(curTid).phase;
    nextCheck = HELPING_DELAY;
  }
}

private class Approximation {

  public Approximation (int size, int tid, long phase) {
    this.app_size = size; this.tid = tid; this.phase = phase;
  }
  final int app_size;
  final int tid;     // used to allow safe help
  final long phase;  // used to allow safe help
}


private final Node head, tail;
private final AtomicReferenceArray<OpDesc> state;
private final AtomicLong currentMaxPhase;
private final HelpRecord helpRecords[];
private final long HELPING_DELAY = 20;
private final int MAX_FAILURES = 20;
private final int width = 128; // an optimization, to avoid false
    sharing.
private AtomicReference<Approximation> app; // holds the size
    approximation
private final int[] difCouners; // a private size counter for each
    thread
private final int soft_threshold = 35; // a thread will try to update
    size approximation
private final int hard_threshold = 50; // a thread will ask help to
    update size approximation

public FPSPList () {
  currentMaxPhase = new AtomicLong(); // used in maxPhase method
  currentMaxPhase.set(1);
  head = new Node(Integer.MIN_VALUE); // head's key is smaller than all
      the rests'
  tail = new Node(Integer.MAX_VALUE); // tail's key is larger than all
      the rests'
```

```
    head.next = new VersionedAtomicMarkableReference<Node>(tail, false);//
        init an empty list
    tail.next = new VersionedAtomicMarkableReference<Node>(tail, false);

    state = new AtomicReferenceArray<OpDesc>(Test.numThreads);
    helpRecords = new HelpRecord[Test.numThreads*width];

    for (int i = 0; i < state.length(); i++) { // state & helpRecord
        entries for each thread
      state.set(i, new OpDesc(0, OpType.success, null,null));
      helpRecords[i*width] = new HelpRecord();
    }

    difCouners = new int[Test.numThreads*width];
    app = new AtomicReference<Approximation>(new Approximation(0, -1, -1))
;
  }

  private void helpIfNeeded(int tid) {
    HelpRecord rec = helpRecords[tid*width];
    if (rec.nextCheck-- == 0) { // only check if help is needed after
        HELPING_DELAY times
      OpDesc desc = state.get(rec.curTid);
      if (desc.phase == rec.lastPhase) { // if the helped thread is on the
          same operation
        if (desc.type == OpType.insert)
          helpInsert(rec.curTid, rec.lastPhase);
        else if (desc.type == OpType.search_delete || desc.type == OpType.
execute_delete)
          helpDelete(rec.curTid, rec.lastPhase);
        else if (desc.type == OpType.contains)
          helpContains(rec.curTid, rec.lastPhase);
        else if (desc.type == OpType.update_approximation)
          helpUpdateGlobalCounter(rec.curTid, rec.lastPhase);
      }
      rec.reset();
    }
  }

  public boolean insert(int tid, int key) {
    if (updateGlobalCounterIfNeeded(tid, difCouners[tid*width]))
      difCouners[tid*width] = 0;
    helpIfNeeded(tid);
    int tries = 0;
    while (tries++ < MAX_FAILURES) { // when tries reaches MAX_FAILURES -
        switch to slowPath
      Window window = fastSearch(key, tid);
      if (window == null) {// happens if search failed MAX_FAILURES times
        boolean result = slowInsert(tid, key);
        if (result)
```

```
          difCouners[tid*width]++;
        return result;
      }
      Node pred = window.pred, curr = window.curr;
      if (curr.key == key)
        return false; // key exists - operation failed.
      else {
        Node node = new Node(key); // allocate the node to insert
        node.next = new VersionedAtomicMarkableReference<Node>(curr, false
);
        if (pred.next.compareAndSet(curr, node, false, false))
          return true; // insertion succeeded
      }
    }
    boolean result = slowInsert(tid, key);
    if (result)
      difCouners[tid*width]++;
    return result;
  }

  public boolean delete(int tid, int key) {
    if (updateGlobalCounterIfNeeded(tid, difCouners[tid*width]))
      difCouners[tid*width] = 0;

    helpIfNeeded(tid);
    int tries = 0; boolean snip;
    while (tries++ < MAX_FAILURES) { // when tries reaches MAX_FAILURES -
        switch to slowPath
      Window window = fastSearch(key, tid);
      if (window == null) {// happens if search failed MAX_FAILURES times
        boolean result = slowDelete(tid, key);
        if (result)
          difCouners[tid*width]--;
        return result;
      }
      Node pred = window.pred, curr = window.curr;
      if (curr.key != key) // key doesn't exist - operation failed
        return false;
      else {
        Node succ = curr.next.getReference();
        snip = curr.next.compareAndSet(succ, succ, false, true); //
            logical delete
        if (!snip)
          continue; // try again
        pred.next.compareAndSet(curr, succ, false, false); // physical
            delete (may fail)
        boolean result = curr.successBit.compareAndSet(false, true); //
            needed for cooperation with slow path
        if (result)
          difCouners[tid*width]--;
```

```
      return result;
    }
  }
  boolean result = slowDelete(tid, key);
  if (result)
    difCouners[tid*width]--;
  return result;
}


final long MaxError = Test.numThreads*hard_threshold;

public Window fastSearch(int key, int tid) {
  long maxSteps = sizeApproximation()+MaxError;
  int tries = 0;
  Node pred = null, curr = null, succ = null;
  boolean[] marked = {false};
  boolean snip;
  retry : while (tries++ < MAX_FAILURES) { // when tries reaches
     MAX_FAILURES - return null
    long steps = 0;
    pred = head;
    curr = pred.next.getReference(); // advancing curr
    while (true) {
      steps++;
      if (steps >= maxSteps)
      {
        return null;
      }
      succ = curr.next.get(marked); // advancing succ and reading curr.
          next's mark
      while (marked[0]) { // curr is logically deleted a should be
          removed
        if (steps >= maxSteps)
        {
          return null;
        }
        // remove a physically deleted node :
        snip = pred.next.compareAndSet(curr, succ, false, false);
        if (!snip) continue retry; // list has changed, retry
        curr = succ; // advancing curr
        succ = curr.next.get(marked); // advancing succ and reading curr
            .next's mark
        steps++;
      }
      if (curr.key >= key) // the curr.key is large enough - found the
          window
        return new Window(pred, curr);
      pred = curr; curr = succ;  // advancing pred  & curr
    }
```

```
  }
  return null;
}

private boolean slowInsert(int tid, int key) {
  long phase = maxPhase(); // getting the phase for the op
  Node n = new Node(key); // allocating the node
  n.next = new VersionedAtomicMarkableReference<Node>(null, false); //
      allocate node.next
  OpDesc op = new OpDesc(phase, OpType.insert, n,null);
  state.set(tid, op); // publishing the operation - asking for help
  helpInsert(tid, phase); // only helping itself here
  return state.get(tid).type == OpType.success;
}

private boolean slowDelete(int tid, int key) {
  long phase = maxPhase(); // getting the phase for the op
  state.set(tid, new OpDesc(phase, OpType.search_delete, new Node(key),
      null));//publishing
  helpDelete(tid,phase);   // only helping itself here
  OpDesc op = state.get(tid);
  if (op.type == OpType.determine_delete)
    // compete on the ownership of deleting this node
    return op.searchResult.curr.successBit.compareAndSet(false, true);
  return false;
}

private Window search(int key, int tid, long phase) {
  Node pred = null, curr = null, succ = null;
  boolean[] marked = {false}; boolean snip;
  retry : while (true) {
    pred = head;
    curr = pred.next.getReference(); // advancing curr
    while (true) {
      succ = curr.next.get(marked); // advancing succ and reading curr.
          next's mark
      while (marked[0]) { // curr is logically deleted a should be
          removed
        // remove a physically deleted node :
        snip = pred.next.compareAndSet(curr, succ, false, false);
        if (!isSearchStillPending(tid,phase))
          return null; // to ensure wait-freedom.
        if (!snip) continue retry; // list has changed, retry
        curr = succ; // advancing curr
        succ = curr.next.get(marked); // advancing succ and reading curr
            .next's mark
      }
      if (curr.key >= key)  // the curr.key is large enough - found the
          window
        return new Window(pred, curr);
```

```
      pred = curr; curr = succ; // advancing pred & curr
    }
  }
}

private void helpInsert(int tid, long phase) {
  while (true) {
    OpDesc op = state.get(tid);
    if (!(op.type == OpType.insert && op.phase == phase))
      return; // the op is no longer relevant, return
    Node node = op.node; // getting the node to be inserted
    Node node_next = node.next.getReference(); //must read node_next
        before search
    Window window = search(node.key,tid,phase); //search a window to
        insert the node into
    if (window == null) // can only happen if operation is no longer
        pending
      return;
    if (window.curr.key == node.key) { // key exists - chance of a
        failure
      if ((window.curr == node) || (node.next.isMarked())) {
        // the node was already inserted - success
        OpDesc success = new OpDesc(phase, OpType.success, node, null);
        if (state.compareAndSet(tid, op, success))
          return;
      }
      else { // the node was not yet inserted - failure
        OpDesc fail = new OpDesc(phase, OpType.failure, node, null);
        // CAS may fail if search results are obsolete
        if (state.compareAndSet(tid, op, fail))
          return;
      }
    }
    else {
      if (node.next.isMarked()) { // node was already inserted and
          marked (=deleted)
        OpDesc success = new OpDesc(phase, OpType.success, node, null);
        if (state.compareAndSet(tid, op, success))
          return;
      }
      int version = window.pred.next.getVersion(); // read version for
          CAS later.
      OpDesc newOp = new OpDesc(phase, OpType.insert, node, null);
      // the following prevents another thread with obsolete search
          results to report failure:
      if (!state.compareAndSet(tid, op, newOp))
        continue; // operation might have already reported as failure
      node.next.compareAndSet(node_next, window.curr, false, false);
      // if successful - than the insert is linearized here :
```

```
        if (window.pred.next.compareAndSet(version, node_next, node, false
, false)) {
          OpDesc success = new OpDesc(phase, OpType.success, node, null);
          if (state.compareAndSet(tid, newOp, success))
            return;
        }
      }
    }
  }

  private void helpDelete(int tid, long phase) {
    while (true) {
      OpDesc op = state.get(tid);
      if (!((op.type == OpType.search_delete || op.type == OpType.
execute_delete)
          && op.phase==phase))
        return; // the op is no longer relevant, return
      Node node = op.node;  // the node holds the key we want to delete
      if (op.type == OpType.search_delete) {  // need to search for the
          key
        Window window = search(node.key,tid,phase);
        if (window==null)
          continue; // can only happen if operation is no longer the same
              search_delete
        if (window.curr.key != node.key) {
          // key doesn't exist - failure
          OpDesc failure = new OpDesc(phase, OpType.failure, node, null);
          if (state.compareAndSet(tid, op, failure))
            return;
        }
        else {
          // key exists - continue to executre_delete
          OpDesc found = new OpDesc(phase, OpType.execute_delete, node,
window);
          state.compareAndSet(tid, op, found);
        }
      }
      else if (op.type == OpType.execute_delete) {
        Node next = op.searchResult.curr.next.getReference();
        if (!op.searchResult.curr.next.attemptMark(next, true)) // mark
            the node
          continue; // will continue to try to mark it, until it is marked
        search(op.node.key,tid,phase); // will physically remove the node
        OpDesc determine =
          new OpDesc(op.phase, OpType.determine_delete, op.node, op.
searchResult);
        state.compareAndSet(tid, op, determine);
        return;
      }
    }
```

```
}

public boolean contains(int tid, int key) {
  long maxSteps = sizeApproximation()+MaxError;
  long steps = 0;
  boolean[] marked = {false};
  Node curr = head;
  while (curr.key < key) { // search for the key
    curr = curr.next.getReference();
    curr.next.get(marked);
    if (steps++ >= maxSteps)
      return slowContains(tid, key);
  }
  return (curr.key == key && !marked[0]); // the key is found and is
      logically in the list
}

private boolean slowContains(int tid, int key) {;
  long phase = maxPhase();
  Node n = new Node(key);
  OpDesc op = new OpDesc(phase, OpType.contains, n, null);
  state.set(tid, op);
  helpContains(tid, phase);
  return state.get(tid).type == OpType.success;
}

private void helpContains(int tid, long phase) {
  OpDesc op = state.get(tid);
  if (!((op.type == OpType.contains) && op.phase==phase))
    return; // the op is no longer relevant, return
  Node node = op.node;  // the node holds the key we want to find
  Window window = search(node.key, tid, phase);
  if (window == null)
    return; // can only happen if operation is already complete.
  if (window.curr.key == node.key) {
    OpDesc success = new OpDesc(phase, OpType.success, node, null);
    state.compareAndSet(tid, op, success);
  }
  else {
    OpDesc failure = new OpDesc(phase, OpType.failure, node, null);
    state.compareAndSet(tid, op, failure);
  }
}

private long maxPhase() {
  long result = currentMaxPhase.get();
  // ensuring maxPhase will increment before this thread next operation
      :
  currentMaxPhase.compareAndSet(result, result+1);
  return result;
```

```
  }

  private boolean isSearchStillPending(int tid, long ph) {
    OpDesc curr = state.get(tid);
    return (curr.type == OpType.insert || curr.type == OpType.
search_delete
        || curr.type == OpType.execute_delete || curr.type == OpType.
contains) &&
        curr.phase == ph; // the operation is pending with a phase lower
            than ph.
  }




  private boolean updateGlobalCounterIfNeeded(int tid, int updateSize) {
    if (Math.abs(updateSize) < soft_threshold)
      return false; // no update was done.
    Approximation old = app.get();
    // old.tid != -1 means you cannot update since a help for updating is
        currently in action
    if (old.tid == -1)
    {
      Approximation newApp = new Approximation(old.app_size + updateSize,
-1, -1);
      if (app.compareAndSet(old, newApp))
        return true; // update happened successfully.
    }
    if (Math.abs(updateSize) < hard_threshold)
      return false; // update failed once, we will try again next
          operation.
    // need to ask for help in updating the counter, since it reached
        hard_threshold
    long phase = maxPhase();
    Node n = new Node(updateSize); // we will use the node key field to
        hold the update size needed.
    OpDesc desc = new OpDesc(phase, OpType.update_approximation, n, null);
    state.set(tid, desc);
    helpUpdateGlobalCounter(tid, phase);
    // after the help returned, the counter is surely updated.
    return true;
  }

  private void helpUpdateGlobalCounter(int tid, long phase) {
    while (true) {
      OpDesc op = state.get(tid);
      if (!((op.type == OpType.update_approximation) && op.phase==phase))
        return; // the op is no longer relevant, return
      Approximation oldApp = app.get();
      if (op != state.get(tid))
```

```
        return; // validating op.
      if (oldApp.tid != -1) { // some help (maybe this one) is in process
        OpDesc helpedTid = state.get(oldApp.tid);
        if (helpedTid.phase == oldApp.phase && helpedTid.type == OpType.
update_approximation) {
          // need to report to the oldApp.tid that its update is completed
              .
          OpDesc success = new OpDesc(helpedTid.phase, OpType.success,
helpedTid.node, null);
          state.compareAndSet(oldApp.tid, helpedTid, success);
        }
        // now we are certain the success has been reported, clean the
            approximation field.
        Approximation clean = new Approximation(oldApp.app_size, -1, -1);
        app.compareAndSet(oldApp, clean);
        continue;
      }
      int updateSize = op.node.key; // here we hold the updateSize
      Approximation newApp = new Approximation(oldApp.app_size+updateSize,
 tid, phase);
      app.compareAndSet(oldApp, newApp);
    }
  }

  private long sizeApproximation() {
    return app.get().app_size;
  }

}
```

Appendix D

The Wait-Free Queue Used in the Wait-Free Simulation

In the simulation given in Chapter 3, we rely on a wait-free queue supporting the operations *enqueue, peek* and *conditionally-remove-head*, rather then *enqueue* and *dequeue* as given in [KP11]. Adjusting the queue from [KP11] to our needs was a very easy task. The java implementation of the adjusted queue that we used is provided here.

```
import java.util.concurrent.atomic.AtomicInteger;
import java.util.concurrent.atomic.AtomicReference;
import java.util.concurrent.atomic.AtomicReferenceArray;


public class WFQueueAd<V>  {

  class Node {
    public V value;
    public AtomicReference<Node> next;
    public int enqTid;
    public AtomicInteger deqTid;
    public Node (V val, int etid) {
      value = val;
      next = new AtomicReference<Node>(null);
      enqTid = etid;
      deqTid = new AtomicInteger(-1);
    }
  }

  protected class OpDesc {
    public long phase;
    public boolean pending;
    public boolean enqueue;
    public Node node;
    public OpDesc (long ph, boolean pend, boolean enq, Node n) {
```

```
    phase = ph;
    pending = pend;
    enqueue = enq;
    node = n;
  }
}

protected AtomicReference<Node> head, tail;
protected AtomicReferenceArray<OpDesc> state;

public AtomicInteger enqed = new AtomicInteger(0);
public AtomicInteger deqed = new AtomicInteger(0);

public WFQueueAd () {
  Node sentinel = new Node(null, -1);
  head = new AtomicReference<Node>(sentinel);
  tail = new AtomicReference<Node>(sentinel);

  state = new AtomicReferenceArray<OpDesc>(Test.numThreads);

  for (int i = 0; i < state.length(); i++) {
    state.set(i, new OpDesc(-1, false, true, null));
  }
}

public void enq(int tid, V value) {
  long phase = maxPhase() + 1;
  state.set(tid,
            new OpDesc(phase, true, true, new Node(value, tid)));
  help(phase);
  help_finish_enq();
}

public V peekHead() {
  Node next = head.get().next.get();
  if (next == null)
    return null;
  return next.value;
}

public boolean conditionallyRemoveHead(V expectedValue) {
  Node currHead = head.get();
  Node next = currHead.next.get();
  if (next == null || !next.value.equals(expectedValue))
    return false;
  if (head.compareAndSet(currHead, next)) {
    help_finish_enq();
    currHead.next.set(null);
    return true;
  }
```

```
    else
      return false;
  }

  protected void help(long phase) {
    for (int i = 0; i < state.length(); i++) {
      OpDesc desc = state.get(i);
      if (desc.pending && desc.phase <= phase) {
        if (desc.enqueue) {
          help_enq(i, phase);
        }
      }
    }
  }

  protected void help_enq(int tid, long phase) {
    while (isStillPending(tid, phase)) {
      Node last = tail.get();
      Node next = last.next.get();
      if (last == tail.get()) {
        if (next == null) {
          if (isStillPending(tid, phase)) {
            if (last.next.compareAndSet
                                  (next, state.get(tid).node)) {
              help_finish_enq();
              return;
            }
          }
        } else {
          help_finish_enq();
        }
      }
    }
  }

  protected void help_finish_enq() {
    Node last = tail.get();
    Node next = last.next.get();
    if (next != null) {
      int tid = next.enqTid;
      OpDesc curDesc = state.get(tid);
      if (last == tail.get() && state.get(tid).node == next) {
        OpDesc newDesc = new OpDesc
                          (state.get(tid).phase, false, true, next);
        state.compareAndSet(tid, curDesc, newDesc);
        tail.compareAndSet(last, next);
      }
    }
  }
```

```
  protected long maxPhase() {
    long maxPhase = -1;
    for (int i = 0; i < state.length(); i++) {
      long phase = state.get(i).phase;
      if (phase > maxPhase) {
        maxPhase = phase;
      }
    }
    return maxPhase;
  }

  protected boolean isStillPending(int tid, long ph) {
    return state.get(tid).pending &&
    state.get(tid).phase <= ph;
  }
}
```

Appendix E

Implementing a Contention Failure Counter in the Presence of Infinite Insertions

A somewhat hidden assumption in the fast-path-slow-path technique (and consequently, in the simulation presented in Chapter 3 as well), is the ability to be able to identify effectively when a thread fails to complete an operation due to contention. Failing to recognize contention will foil wait-freedom, as the relevant thread will not ask for help. Counting the number of failed CASes is generally a very effective way of identifying contention. However, it is not always enough. For example, in the binary search tree, a thread may never fail a CAS, and yet be held forever executing auxiliary CASes for other threads' operations. Identifying such a case is generally easy. For the binary tree algorithm, we did so by counting invocations of the parallelizable help methods.

However, there is one problem that often presents a greater difficulty. We refer to this problem as *the infinite insertions problem*. This is a special case in which a thread in a lock-free algorithm may never complete an operation and yet never face contention.

Consider what happens when a data structure keeps growing while a thread is trying to traverse it. For example, consider what happens in a linked-list, if while a thread tries to traverse it to reach a certain key, other threads keep inserting infinitely many new nodes before the wanted key. The thread might never reach the needed key. The complexity of searching the key in this case is linear at the size of the list, but this size keeps growing. If the list size is some how limited (for example, if all the keys in the list must be integers), then this cannot go on forever, and eventually the traversing thread must reach the key it seeks (or discover it is not there). Such a bound on the size of the data structure can be used to assert for the wait-freedom of some of the algorithms we have discussed in Chapter 3, but it provides a rather poor bound for the wait-freedom property, and it cannot at all be used at some cases. (Such as in a list that employs strings, instead of integers, as keys.)

To implement a contention failure counter that is robust to this problem, we offer the following mechanism to enable a thread to identify if the data structure is getting larger while it is working on it. The idea is that each thread will read a field stating the size of the data structure prior to traversing. For example, in a list, a skiplist or a tree, it can read the number of nodes of the data structure. During the operation, it will count how many nodes it traverses, and if the number of traversed nodes is higher than the original total number of nodes (plus some constant), it will abort the fast-path and will ask for help.

However, a naive implementation of this basic idea performs poorly in practice, since maintaining the exact number of nodes in a wait-free manner can be very costly. Instead, we settle for maintaining a field that approximates the number of keys. The error of the approximation is bounded by a linear function of the number of threads operating on the data structure. Thus, before a thread starts traversing the data structure, it should read the approximation, denoted Size-App, and if it traverses a number of nodes that is greater than Size-App + Max-Error + Const, switch to the slow path and ask for help.

To maintain the approximation for the number of nodes, the data structure contains a global field with the approximation, and each thread holds a private counter. In its private counter, each thread holds the number of nodes it inserted to the data structure minus the number of nodes it deleted from it since the last time the thread updated the global approximation field. To avoid too much contention in updating the global field, each thread only attempts to update it (by a cas) once it reaches a certain *soft_threshold* (in absolute value). If the cas failed, the thread continues the operation as usual, and will attempt to update the global approximation field at its next insert or delete operation. If the private counter of a thread reaches a certain *hard_threshold*, it asks for help in updating the global counter. This is done similarly to asking help for other operations: it should enqueue a request into the help-queue. The input for the operation of updateGlobalCounter is an integer stating the required adjustment. The *Generator* method here is reading the global counter, and then output a single cas description, describing a cas that alters the old counter value with the wanted new one. The Wrap-up method exits the operation if the cas succeeded, or indicates that the operation should be restarted if the cas failed[1]. Such an approximation of the size of the data structure can be maintained very cheaply, and is enough to solve the problem of the infinite insertions.

[1]In essence, we have just described the normalized lock-free algorithm for a shared counter.

Bibliography

[AACH12] James Aspnes, Hagit Attiya, and Keren Censor-Hillel. Polylogarithmic concurrent data structures from monotone circuits. *J. ACM*, 59(1):2:1–2:24, March 2012.

[AAD+93] Yehuda Afek, Hagit Attiya, Danny Dolev, Eli Gafni, Michael Merritt, and Nir Shavit. Atomic snapshots of shared memory. *J. ACM*, 40(4):873–890, September 1993.

[ADT95] Yehuda Afek, Dalia Dauber, and Dan Touitou. Wait-free made fast (extended abstract). In *Proceedings of the Twenty-Seventh Annual ACM Symposium on Theory of Computing, 29 May-1 June 1995, Las Vegas, Nevada, USA*, pages 538–547, 1995.

[AK99] James H. Anderson and Yong-Jik Kim. Fast and scalable mutual exclusion. In *Distributed Computing, 13th International Symposium, Bratislava, Slavak Republic, September 27-29, 1999, Proceedings*, pages 180–194, 1999.

[AK00] James H. Anderson and Yong-Jik Kim. Adaptive mutual exclusion with local spinning. In *Distributed Computing, 14th International Conference, DISC 2000, Toledo, Spain, October 4-6, 2000, Proceedings*, pages 29–43, 2000.

[AM99] James H. Anderson and Mark Moir. Universal constructions for large objects. *IEEE Trans. Parallel Distrib. Syst.*, 10(12):1317–1332, 1999.

[And94] James H. Anderson. Multi-writer composite registers. *Distributed Computing*, 7(4):175–195, 1994.

[AST09] Yehuda Afek, Nir Shavit, and Moran Tzafrir. Interrupting snapshots and the java$^{\text{tm}}$ size() method. In *Distributed Computing, 23rd International Symposium, DISC 2009, Elche, Spain, September 23-25, 2009. Proceedings*, pages 78–92, 2009.

[Bar93] Greg Barnes. A method for implementing lock-free shared-data structures. In *SPAA*, pages 261–270, 1993.

[BCCO10] Nathan Grasso Bronson, Jared Casper, Hassan Chafi, and Kunle Olukotun. A practical concurrent binary search tree. In *Proceedings of the 15th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPOPP 2010, Bangalore, India, January 9-14, 2010*, pages 257–268, 2010.

[BER13] Trevor Brown, Faith Ellen, and Eric Ruppert. Pragmatic primitives for non-blocking data structures. In *ACM Symposium on Principles of Distributed Computing, PODC '13, Montreal, QC, Canada, July 22-24, 2013*, pages 13–22, 2013.

[BER14] Trevor Brown, Faith Ellen, and Eric Ruppert. A general technique for non-blocking trees. In *ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPoPP '14, Orlando, FL, USA, February 15-19, 2014*, pages 329–342, 2014.

[BH11] Trevor Brown and Joanna Helga. Non-blocking k-ary search trees. In *Principles of Distributed Systems - 15th International Conference, OPODIS 2011, Toulouse, France, December 13-16, 2011. Proceedings*, pages 207–221, 2011.

[BKP13] Anastasia Braginsky, Alex Kogan, and Erez Petrank. Drop the anchor: lightweight memory management for non-blocking data structures. In *25th ACM Symposium on Parallelism in Algorithms and Architectures, SPAA '13, Montreal, QC, Canada - July 23 - 25, 2013*, pages 33–42, 2013.

[BMV+07] Jayaram Bobba, Kevin E. Moore, Haris Volos, Luke Yen, Mark D. Hill, Michael M. Swift, and David A. Wood. Performance pathologies in hardware transactional memory. In *34th International Symposium on Computer Architecture (ISCA 2007), June 9-13, 2007, San Diego, California, USA*, pages 81–91, 2007.

[CER10] Phong Chuong, Faith Ellen, and Vijaya Ramachandran. A universal construction for wait-free transaction friendly data structures. In *SPAA 2010: Proceedings of the 22nd Annual ACM Symposium on Parallelism in Algorithms and Architectures, Thira, Santorini, Greece, June 13-15, 2010*, pages 335–344, 2010.

[CIR12] Tyler Crain, Damien Imbs, and Michel Raynal. Towards a universal construction for transaction-based multiprocess programs. In *Distributed Computing and Networking - 13th International Conference, ICDCN 2012, Hong Kong, China, January 3-6, 2012. Proceedings*, pages 61–75, 2012.

[DH12] Aleksandar Dragojevic and Tim Harris. STM in the small: trading generality for performance in software transactional memory. In *European Conference on Computer Systems, Proceedings of the Seventh EuroSys Conference 2012, EuroSys '12, Bern, Switzerland, April 10-13, 2012*, pages 1–14, 2012.

[EFRvB10] Faith Ellen, Panagiota Fatourou, Eric Ruppert, and Franck van Breugel. Non-blocking binary search trees. In *Proceedings of the 29th Annual ACM Symposium on Principles of Distributed Computing, PODC 2010, Zurich, Switzerland, July 25-28, 2010*, pages 131–140, 2010.

[EHS12] Faith Ellen, Danny Hendler, and Nir Shavit. On the inherent sequentiality of concurrent objects. *SIAM J. Comput.*, 41(3):519–536, 2012.

[FK07] Panagiota Fatourou and Nikolaos D. Kallimanis. Time-optimal, space-efficient single-scanner snapshots & multi-scanner snapshots using CAS. In *Proceedings of the Twenty-Sixth Annual ACM Symposium on Principles of Distributed Computing, PODC 2007, Portland, Oregon, USA, August 12-15, 2007*, pages 33–42, 2007.

[FK09] Panagiota Fatourou and Nikolaos D. Kallimanis. The redblue adaptive universal constructions. In *Distributed Computing, 23rd International Symposium, DISC 2009, Elche, Spain, September 23-25, 2009. Proceedings*, pages 127–141, 2009.

[FK11] Panagiota Fatourou and Nikolaos D. Kallimanis. A highly-efficient wait-free universal construction. In *SPAA 2011: Proceedings of the 23rd Annual ACM Symposium on Parallelism in Algorithms and Architectures, San Jose, CA, USA, June 4-6, 2011 (Co-located with FCRC 2011)*, pages 325–334, 2011.

[FLMS05] Faith Ellen Fich, Victor Luchangco, Mark Moir, and Nir Shavit. Obstruction-free algorithms can be practically wait-free. In *Distributed Computing, 19th International Conference, DISC 2005, Cracow, Poland, September 26-29, 2005, Proceedings*, pages 78–92, 2005.

[FLP85] Michael J. Fischer, Nancy A. Lynch, and Michael S. Paterson. Impossibility of distributed consensus with one faulty process. *J. ACM*, 32(2):374–382, April 1985.

[FR04] Mikhail Fomitchev and Eric Ruppert. Lock-free linked lists and skip lists. In *Proceedings of the Twenty-Third Annual ACM Symposium*

on Principles of Distributed Computing, PODC 2004, St. John's, Newfoundland, Canada, July 25-28, 2004, pages 50–59, 2004.

[Gre02] Michael Greenwald. Two-handed emulation: how to build non-blocking implementation of complex data-structures using DCAS. In *Proceedings of the Twenty-First Annual ACM Symposium on Principles of Distributed Computing, PODC 2002, Monterey, California, USA, July 21-24, 2002*, pages 260–269, 2002.

[Har01] Timothy L. Harris. A pragmatic implementation of non-blocking linked-lists. In *Distributed Computing, 15th International Conference, DISC 2001, Lisbon, Portugal, October 3-5, 2001, Proceedings*, pages 300–314, 2001.

[Her88] Maurice P. Herlihy. Impossibility and universality results for wait-free synchronization. In *Proceedings of the Seventh Annual ACM Symposium on Principles of Distributed Computing (PODC)*, pages 276–290, 1988.

[Her90] Maurice Herlihy. A methodology for implementing highly concurrent data structures. In *Proceedings of the Second ACM SIGPLAN Symposium on Princiles & Practice of Parallel Programming (PPOPP), Seattle, Washington, USA, March 14-16, 1990*, pages 197–206, 1990.

[Her91] Maurice Herlihy. Wait-free synchronization. *ACM Trans. Program. Lang. Syst.*, 13(1):124–149, January 1991.

[Her93] Maurice Herlihy. A methodology for implementing highly concurrent objects. *ACM Trans. Program. Lang. Syst.*, 15(5):745–770, 1993.

[HFP02] Timothy L. Harris, Keir Fraser, and Ian A. Pratt. A practical multi-word compare-and-swap operation. In *Distributed Computing, 16th International Conference, DISC 2002, Toulouse, France, October 28-30, 2002 Proceedings*, pages 265–279, 2002.

[HHL+05] Steve Heller, Maurice Herlihy, Victor Luchangco, Mark Moir, William N. Scherer III, and Nir Shavit. A lazy concurrent list-based set algorithm. In *Principles of Distributed Systems, 9th International Conference, OPODIS 2005, Pisa, Italy, December 12-14, 2005, Revised Selected Papers*, pages 3–16, 2005.

[HLMM05] Maurice Herlihy, Victor Luchangco, Paul A. Martin, and Mark Moir. Nonblocking memory management support for dynamic-sized data structures. *ACM Trans. Comput. Syst.*, 23(2):146–196, 2005.

[HM93] Maurice Herlihy and J. Eliot B. Moss. Transactional memory: Architectural support for lock-free data structures. In *Proceedings of the 20th Annual International Symposium on Computer Architecture. San Diego, CA, May 1993*, pages 289–300, 1993.

[HS08] Maurice Herlihy and Nir Shavit. *The art of multiprocessor programming.* Morgan Kaufmann, 2008.

[HW90] Maurice Herlihy and Jeannette M. Wing. Linearizability: A correctness condition for concurrent objects. *ACM Trans. Program. Lang. Syst.*, 12(3):463–492, 1990.

[IR94] Amos Israeli and Lihu Rappoport. Disjoint-access-parallel implementations of strong shared memory primitives. In *Proceedings of the Thirteenth Annual ACM Symposium on Principles of Distributed Computing, Los Angeles, California, USA, August 14-17, 1994*, pages 151–160, 1994.

[Jay05] Prasad Jayanti. An optimal multi-writer snapshot algorithm. In *Proceedings of the 37th Annual ACM Symposium on Theory of Computing, Baltimore, MD, USA, May 22-24, 2005*, pages 723–732, 2005.

[JTT00] Prasad Jayanti, King Tan, and Sam Toueg. Time and space lower bounds for nonblocking implementations. *SIAM J. Comput.*, 30(2):438–456, 2000.

[KP11] Alex Kogan and Erez Petrank. Wait-free queues with multiple enqueuers and dequeuers. In *Proceedings of the 16th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPOPP 2011, San Antonio, TX, USA, February 12-16, 2011*, pages 223–234, 2011.

[KP12] Alex Kogan and Erez Petrank. A methodology for creating fast wait-free data structures. In *Proceedings of the 17th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPOPP 2012, New Orleans, LA, USA, February 25-29, 2012*, pages 141–150, 2012.

[Lam74] Leslie Lamport. A new solution of dijkstra's concurrent programming problem. *Commun. ACM*, 17(8):453–455, August 1974.

[Lam87] Leslie Lamport. A fast mutual exclusion algorithm. *ACM Trans. Comput. Syst.*, 5(1):1–11, 1987.

[MA95] Mark Moir and James H. Anderson. Wait-free algorithms for fast, long-lived renaming. *Sci. Comput. Program.*, 25(1):1–39, 1995.

[Mic02] Maged M. Michael. High performance dynamic lock-free hash tables and list-based sets. In *Proceedings of the fourteenth annual ACM symposium on Parallel algorithms and architectures*, SPAA '02, pages 73–82, New York, NY, USA, 2002. ACM.

[Mic04] Maged M. Michael. Hazard pointers: Safe memory reclamation for lock-free objects. *IEEE Trans. Parallel Distrib. Syst.*, 15(6):491–504, 2004.

[MS96] Maged M. Michael and Michael L. Scott. Simple, fast, and practical non-blocking and blocking concurrent queue algorithms. In *Proceedings of the Fifteenth Annual ACM Symposium on Principles of Distributed Computing, Philadelphia, Pennsylvania, USA, May 23-26, 1996*, pages 267–275, 1996.

[NSM13] Aravind Natarajan, Lee Savoie, and Neeraj Mittal. Concurrent wait-free red black trees. In *Stabilization, Safety, and Security of Distributed Systems - 15th International Symposium, SSS 2013, Osaka, Japan, November 13-16, 2013. Proceedings*, pages 45–60, 2013.

[PBBO12] Aleksandar Prokopec, Nathan Grasso Bronson, Phil Bagwell, and Martin Odersky. Concurrent tries with efficient non-blocking snapshots. In *Proceedings of the 17th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPOPP 2012, New Orleans, LA, USA, February 25-29, 2012*, pages 151–160, 2012.

[Plo89] S. A. Plotkin. Sticky bits and universality of consensus. In *Proceedings of the Eighth Annual ACM Symposium on Principles of Distributed Computing (PODC)*, pages 159–175, 1989.

[RG01] Ravi Rajwar and James R. Goodman. Speculative lock elision: enabling highly concurrent multithreaded execution. In *Proceedings of the 34th Annual International Symposium on Microarchitecture, Austin, Texas, USA, December 1-5, 2001*, pages 294–305, 2001.

[RHP+07] Christopher J. Rossbach, Owen S. Hofmann, Donald E. Porter, Hany E. Ramadan, Bhandari Aditya, and Emmett Witchel. Txlinux: using and managing hardware transactional memory in an operating system. In *Proceedings of the 21st ACM Symposium on Operating Systems Principles 2007, SOSP 2007, Stevenson, Washington, USA, October 14-17, 2007*, pages 87–102, 2007.

[RST95] Yaron Riany, Nir Shavit, and Dan Touitou. Towards A practical snapshot algorithm. In *ISTCS*, pages 121–129, 1995.

[Rup00] Eric Ruppert. Determining consensus numbers. *SIAM J. Comput.*, 30(4):1156–1168, 2000.

[ST97] Nir Shavit and Dan Touitou. Software transactional memory. *Distributed Computing*, 10(2):99–116, 1997.

[Sun11] Håkan Sundell. Wait-free multi-word compare-and-swap using greedy helping and grabbing. *International Journal of Parallel Programming*, 39(6):694–716, 2011.

[Tau09] Gadi Taubenfeld. Contention-sensitive data structures and algorithms. In *Distributed Computing, 23rd International Symposium, DISC 2009, Elche, Spain, September 23-25, 2009. Proceedings*, pages 157–171, 2009.

[TBKP12] Shahar Timnat, Anastasia Braginsky, Alex Kogan, and Erez Petrank. Wait-free linked-lists. In *Principles of Distributed Systems, 16th International Conference, OPODIS 2012, Rome, Italy, December 18-20, 2012. Proceedings*, pages 330–344, 2012.

[TP14] Shahar Timnat and Erez Petrank. A practical wait-free simulation for lock-free data structures. In *ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPOPP)*, pages 357–368, 2014.

[Val95] John D. Valois. Lock-free linked lists using compare-and-swap. In *Proceedings of the Fourteenth Annual ACM Symposium on Principles of Distributed Computing, Ottawa, Ontario, Canada, August 20-23, 1995*, pages 214–222, 1995.

נוספת על השלמת הפעולה שלו, אשר תכליתה היא לאפשר את התקדמותם של תהליכים אחרים. באופן מפתיע, על אף שעזרה היא מרכיב קריטי במימושים רבים של מבני נתונים מקביליים, עזרה לא נחקרה באופן מעמיק כמושג עצמאי.

אנו עורכים מחקר פורמאלי ומדויק על הקשר שבין עזרה לבין חופש מהמתנה. במיוחד אנו מעוניינים בשאלה: האם עזרה היא הכרחית להבטחת חופש מהמתנה ? כדי לאפשר מחקר של הנושא, אנו מתחילים עם הצעה של הגדרה פורמאלית למושג העזרה. אנו טוענים בעזרת דוגמאות שההגדרה הפורמאלית תואמת את התפישה האינטואיטיבית המקובלת של המונח. לאחר מכן, אנו מציגים ומנתחים תכונות של ממשקים אשר בעבורם כל מימוש של מבני נתונים חופשי מהמתנה מחייב שימוש בעזרה. ממשקים אלו כוללים טיפוסים נפוצים מאד כמו מחסנית ותור. מן הצד השני, אנו מראים שיש גם ממשקים אשר ניתנים למימוש באופן שהוא חופשי מהמתנה גם ללא עזרה.

איטרטור

כמעט שאין בנמצא מבני נתונים חופשיים מהמתנה, או אפילו רק חופשיים מנעילה, שתומכים בפעולות אשר דורשות מידע גלובאלי על מצבו של מבנה הנתונים, כמו ספירה של כמות האלמנטים בתוך מבנה הנתונים או מעבר על מרכיביו הבסיסיים של המבנה (איטרציה). באופן כללי, פעולות כגון אלו יהיו פשוטות מאד למימוש אם ניתן יהיה לקבל תמונה אטומית של מבנה הנתונים.

התרומה המשמעותית הבאה התור של עבודת המחקר הזו היא תכנון איטרטור חופשי מהמתנה ויעיל עבור מבני נתונים אשר תומכים בממשק הקבוצה. אנו משתמשים בתכנון המוצע על־מנת לממש איטרטור לרשימה מקושרת ולרשימת דילוגים. כדי להשיג את האיטרטור, אנחנו קודם כל מאפשרים לקיחת תמונת מצב אטומית של מבנה הנתונים. בהינתן התמונה האטומית, קל לספק איטרטור, או לספור את מספר האלמנטים במבנה

רתימת זיכרון טרנזקציוני למבני נתונים עם הבטחות התקדמות

זיכרון טרנזקציוני הולך ונהיה מושג מרכזי בתכנות מקבילי. לאחרונה, אינטל הציגה הרחבות למעבדים שלה אשר כוללות חומרה של זיכרון טרנזקציוני. עם זאת, ישנן כמה סיבות שיגרמו למתכנת שלא לעשות שימוש בזיכרון טרנזקציוני. ראשית, חומרה מתאימה קיימת רק בחלק מן המעבדים בשוק. שנית, החומרה הקיימת מבוססת על שיטת: מאמץ מיטבי, אך לא מבטיחה הצלחה של הטרנזקציה. לכן, כדי לעבוד עם זיכרון טרנזקציוני, יש לספק גם מסלול חלופי למקרה שטרנזקציות נכשלות.

התרומה המרכזית האחרונה של עבודת מחקר זו היא פרדיגמת תכנות אשר מאפשרת שימוש בחומרת זיכרון טרנזקציוני תוך ניצול יתרונותיה כאשר היא קיימת, ומן הצד השני מאפשרת יצירת תוכנות שעובדות נכון ובביצועים סבירים גם על מעבדים שאינם כוללים חומרה זו. למטרה זו, אנו מציעים כימוס של הזיכרון הטרנזקציוני בתוך פעולת ביניים. פעולת הביניים מתקמפלת למימוש שעושה שימוש בזיכרון טרנזקציוני כאשר החומרה המתאימה קיימת, ולמימוש שלא עושה שימוש בזיכרון כזה כאשר החומרה אינה בנמצא. במקרים מסוימים, פעולת הביניים שלנו יכולה אף להיות ממומשת באופן שתומך גם במסלול חלופי במקרה שטרנזקציות נכשלות באופן תדיר. תכונה זו הופכת את השימוש בפרדיגמה שלנו למתאים גם עבור מבני נתונים אשר תומכים בהבטחת התקדמות.

רשימה מקושרת מקבילית עם הבטחת ההתקדמות החזקה ביותר: חופש מהמתנה, לא הייתה קיימת עד לאחרונה.

התרומה הראשונה של מחקר זה היא רשימה מקבילית מהירה, מעשית, וחופשית מהמתנה. האלגוריתם שלנו מתבסס על אלגוריתם של רשימה מקבילית מאת האריס, ומרחיב אותו תוך שימוש בעזרה על מנת להפוך אותו לחופשי מהמתנה. הקשיים הטכניים המרכזיים בבניה הם להבטיח שהתהליכים שמעניקים עזרה יבצעו כל פעולה פעם באופן נכון, פעם אחת בלבד, ויחזירו תוצאה מתאימה (הצלחה או לא) בהתאם לשאלה האם פעולתו של התהליך הנעזר צלחה. התמודדות עם קשיים אלו היא משימה מורכבת.

לאחר מכן, אנו משפרים את הרשימה המקושרת באמצעות שיטת: מסלול־מהיר־מסלול־איטי, על מנת להפוך אותה למהירה אפילו יותר, ולהשיג ביצועים שכמעט זהים לרשימה של האריס, אשר אינה נותנת הבטחת התקדמות כה חזקה. העיקרון בשיטת מסלול־מהיר־מסלול־איטי הוא עבודה במסלול מהיר שנותן הבטחת התקדמות חלשה יותר (חופש מנעילה), ויחד עם זאת לתמוך במעבר למסלול איטי יותר שתומך בחופש מהמתנה, כאשר ההתקדמות אינה עונה על הנדרש.

טכניקת סימולציה חופשית המתנה למבני נתונים חופשיים מנעילה

השלב הבא שלנו הוא לבחון את תהליך העיצוב של שעשינו עבור הרשימה המקושרת, ולנסות להכליל אותו עבור טווח רחב של מבני נתונים מקביליים. התהליך שעשינו עבור הרשימה המקושרת הוא להתחיל ממבנה נתונים שנותן הבטחת התקדמות חלשה יותר (חופש מנעילה), להשקיע מאמץ בעיצוב מנגנון עזרה נכון על מנת להשיג חופש מהמתנה, ואז להשקיע מאמץ נוסף על מנת ליצור שילוב נכון ויעיל של שני האלגוריתמים בשיטת מסלול־מהיר־מסלול־איטי. בניית מבנה נתונים בשיטת מסלול־מהיר־מסלול־איטי איננה תהליך קל: יש לתכנן את המסלול המהיר והאיטי כך שיוכלו לעבוד בסנכרון וביעילות על אותו מבנה הנתונים על־מנת שהתוצאה הסופית תהיה מבנה נתונים שהוא נכון, יעיל, וחופשי מהמתנה.

אנו שואלים האם כל התהליך הזה יכול להיעשות באופן מכאני, ואם כך גם על ידי מי שאיננו מומחה בתחום. כלומר, אנו שואלים האם בהינתן מבנה נתונים חופשי מנעילה, ניתן להכיל עליו מנגנון עזרה כללי על מנת להשיג מבנה נתונים חופשי מהמתנה, ואז באופן אוטומטי לשלב היטב בין מבנה הנתונים החופשי מנעילה עם מבנה הנתונים החופשי מהמתנה בשיטת מסלול־מהיר־מסלול־איטי.

אנו מראים שהתשובה לשאלה זו היא חיובית. התרומה החשובה השנייה של עבודת המחקר הזו היא תהליך המרה אוטומטי אשר ממיר מבנה נתונים חופשי מנעילה למבנה נתונים שהוא גם יעיל וגם חופשי מהמתנה.

על הקשר שבין עזרה לחופש מהמתנה

כאשר עיצבנו את הרשימה המקושרת החופשית מהמתנה, וגם כאשר פיתחנו את טכניקת הסימולציה הכללית, השתמשנו במנגנון של עזרה על מנת להשיג חופש מהמתנה. זוהי גישה שנפוצה גם בעבודות רבות נוספות בתחום. באופן בלתי פורמאלי, כאשר תהליך מעניק עזרה, הוא מבצע עבודה שהיא

תקציר

כיום, כאשר כמעט כל מחשב אוגד בתוכו מספר מעבדים, חישוב מקבילי הפך להיות הסטנדרט המקובל. מבני נתונים מקביליים מתוכננים במטרה לנצל את כל המעבדים הזמינים על מנת להשיג ביצועים מהירים ככל הניתן. בעבודה זו אנו מעצבים מבני נתונים מקביליים חדשים, מציעים טכניקות לשיפור התכונות של מבני נתונים מקביליים, מתכננים איטרטורים יעילים עבור מבני נתונים מקביליים, מפתחים טכניקות פיתוח תוכנה חדשות, ומוכיחים באופן פורמאלי כמה מגבלות על התכונות שמבני נתונים מקביליים כלשהם יכולים לספק.

בייחוד אנו מתמקדים במבני נתונים אשר תומכים בהבטחות התקדמות. מבין הבטחות ההתקדמות הנפוצות בספרות, חופש מהמתנה (wait-freedom)היא ההבטחה החזקה ביותר, והבטחה זו היא מושג מרכזי בעבודת המחקר הזו. אנחנו מתחילים מתכנון הרשימה המקושרת חופשית ההמתנה הראשונה בעלת ביצועים מעשיים. לאחר מכן אנחנו מכלילים את הטכניקה, ומציעים שיטה אוטומטית שמאפשרת אף למי שאינו מומחה בתחום לעצב מבני נתונים יעילים וחופשיים מהמתנה. אנחנו משתמשים בטכניקה הזו על מנת ליצור עץ חיפוש בינארי ורשימת דילוגים (skiplist)מהירים וחופשיים מהמתנה.

לאחר מכן אנחנו עוברים לחקירה של מושג העזרה באלגוריתמים חופשיים מהמתנה. חופש מהמתנה מושג במקרים רבים על ידי כך שמאפשרים לתהליכים לעזור לתהליכים אחרים להשלים את עבודתם. אנו מציעים הגדרה פורמאלית מדויקת למושג העזרה, ומוכיחים שעבור ממשקים רבים קיומו של מבנה נתונים חופשי מהמתנה ללא עזרה הוא בלתי אפשרי.

הצעד הבא שלנו הוא בעיצובו של איטרטור שיכול לשמש מבני נתונים מקביליים חופשיים מהמתנה. איטרטור הוא ממשק שמאפשר מעבר על כל מרכיביו הבסיסיים של מבנה נתונים מסוים. עד לאחרונה, לא היו כלל מבני נתונים מקביליים וחופשיים מהמתנה שתמכו בממשק האיטרטור, ואנו מגשרים על פער זה. לקראת סיום, אנו מציעים פרדיגמת תכנות חדשה אשר מאפשרת לעשות שימוש בחומרת זיכרון טרנזקציונית (Hardware Transactional Memory)במבני נתונים מקביליים, ובפרט במבני נתונים עם הבטחת התקדמות.

רשימות מקושרות חסרות המתנה

רשימה מקושרת היא אחד ממבני הנתונים הנפוצים ביותר. על פניו מבנה נתונים זה הוא מועמד מצוין למיקבול, משום שעדכונים לחלקים שונים של הרשימה יכולים להתבצע באופן מקבילי בלתי תלוי. ואכן, רשימות מקושרות מקביליות עם הבטחות התקדמות הן נפוצות בספרות. יחד עם זאת,

המחקר בוצע בהנחייתו של פרופסור ארז פטרנק, בפקולטה למדעי המחשב.

חלק מן התוצאות בחיבור זה פורסמו כמאמרים מאת המחבר ושותפיו למחקר בכנסים במהלך תקופת מחקר הדוקטורט של המחבר, אשר גרסאותיהם העדכניות ביותר הינן:

Keren Censor-Hillel, Erez Petrank, and Shahar Timnat. Help! In *Proceedings of the 34th Annual ACM Symposium on Principles of Distributed Computing, PODC 2015, Donostia-San Sebastian, Spain, July 21-23, 2015*.

Erez Petrank and Shahar Timnat. Lock-free data-structure iterators. In *Distributed Computing - 27th International Symposium, DISC 2013, Jerusalem, Israel, October 14-18, 2013. Proceedings*, pages 224–238, 2013.

Shahar Timnat, Anastasia Braginsky, Alex Kogan, and Erez Petrank. Wait-free linked-lists. In *Principles of Distributed Systems, 16th International Conference, OPODIS 2012, Rome, Italy, December 18-20, 2012. Proceedings*, pages 330–344, 2012.

Shahar Timnat, Maurice Herlihy, and Erez Petrank. A practical transactional memory interface. In *Euro-Par 2015 Parallel Processing - 21st International Conference, Vienna, Austria, August 24-28, 2015. Proceedings*.

Shahar Timnat and Erez Petrank. A practical wait-free simulation for lock-free data structures. In *ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPOPP)*, pages 357–368, 2014.

תודות

אני רוצה להודות למנחה שלי, פרופסור ארז פטרנק, על ההדרכה והתמיכה שהעניק לי. מתחילת הלימודים ועד לסופם, לא היה לי ספק לרגע שהוא המנחה הטוב ביותר עבורי. ארז תמיד קיבל אותי עם חיוך וברוחב לב, ותמיד ידע להעניק עצה טובה הן לדוקטורט והן לחיים בכלל. הוא עבורי הרבה יותר ממנחה לדוקטורט.

תודה מיוחדת גם לפרופסור קרן צנזור־הלל, שהעבודה במחיצתה הייתה הן פורייה והן מהנה במיוחד. לסיום, אני רוצה להודות לפרופסור מוריס הרליהי על רעיונותיו החכמים שעזרו למקד את עבודתי, ועל הדרך הנעימה בה הציע אותם.

הכרת תודה מסורה לטכניון על מימון מחקר זה. אני רוצה להודות גם למר וגברת ג'ייקובס בעבור מלגת ג'ייקובס שקיבלתי בשנת 2012. עבודתי נתמכה גם על ידי הקרן הלאומית למדע, מלגה מספר 283/10, ועל ידי קרן המחקר המשותפת של ארצות הברית וישראל, מגלה מספר 2012171.

מבני נתונים מקביליים יעילים

חיבור על מחקר

לשם מילוי חלקי של הדרישות לקבלת התואר
דוקטור לפילוסופיה

שחר תמנת

הוגש לסנט הטכניון – מכון טכנולוגי לישראל
סיוון התשע"ה חיפה יוני 2015

מבני נתונים מקביליים יעילים

שחר תמנת