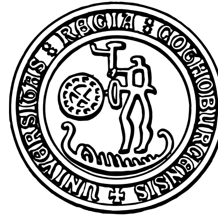
Thesis for the degree of Doctor of Philosophy

Efficient and Practical Non-Blocking Data Structures

HÅKAN SUNDELL

CHALMERS | GÖTEBORG UNIVERSITY



Department of Computing Science
Chalmers University of Technology and Göteborg University
SE-412 96 Göteborg, Sweden

Göteborg, 2004

Efficient and Practical Non-Blocking Data Structures
HÅKAN SUNDELL
ISBN 91-7291-514-5

© HÅKAN SUNDELL, 2004.

Doktorsavhandlingar vid Chalmers tekniska högskola
Ny serie nr 2196
ISSN 0346-718X

Technical report 30D
ISSN 1651-4971
School of Computer Science and Engineering

Department of Computing Science
Chalmers University of Technology and Göteborg University
SE-412 96 Göteborg
Sweden
Telephone + 46 (0)31-772 1000

Cover: A skip list data structure with concurrent inserts and deletes.

Chalmers Reproservice
Göteborg, Sweden, 2004

Abstract

This thesis deals with how to design and implement efficient, practical and reliable concurrent data structures. The design method using mutual exclusion incurs serious drawbacks, whereas the alternative non-blocking techniques avoid those problems and also admit improved parallelism. However, designing non-blocking algorithms is a very complex task, and a majority of the algorithms in the literature are either inefficient, impractical or both.

We have studied how information available in real-time systems can improve and simplify non-blocking algorithms. We have designed new methods for recycling of buffers as well as time-stamps, and have applied them on known non-blocking algorithms for registers, snapshots and priority queues.

We have designed, to the best of our knowledge, the first practical lock-free algorithm of a skip list data structure. Using our skip list construction we have designed a lock-free algorithm of the priority queue abstract data type, as well as a lock-free algorithm of the dictionary abstract data type.

We have designed, to the best of our knowledge, the first practical lock-free algorithm of a doubly linked list data structure. The algorithm supports well-defined traversals in both directions including deleted nodes. Using our doubly linked list construction we have designed a lock-free algorithm of the deque abstract data type. For the lock-free algorithms presented in this thesis, we have given correctness proofs of the strong consistency property called linearizability and the non-blocking properties.

We have made implementations for actual systems of the algorithms presented in this thesis and a representative set of related non-blocking as well as lock based algorithms in the literature. We have built a framework that combines the implementations in the form of a software library that offers a unified and efficient interface in combination with a portable design.

We have performed empirical performance studies of the data structures presented in this thesis in comparison with related alternative solutions. The experiments performed on an extensive set of multi-processor systems show significant improvements for non-blocking alternatives in general, and for the implementations presented in this thesis in particular.

Keywords: *synchronization, non-blocking, shared data structure, skip list, doubly linked list, priority queue, dictionary, deque, snapshot, shared register, real-time, shared memory, lock-free, wait-free, abstract data type.*

List of Included Papers and Reports

This thesis is based on the following publications:

1. B. Allvin, A. Ermedahl, H. Hansson, M. Papatriantafilou, H. Sundell and P. Tsigas, "Evaluating the Performance of Wait-Free Snapshots in Real-Time Systems," in *Proceedings of SNART'99 Real Time Systems Conference*, pages 196–207, Aug. 1999.

2. H. Sundell, P. Tsigas and Y. Zhang, "Simple and Fast Wait-Free Snapshots for Real-Time Systems," in *Proceedings of the 4th International Conference On Principles Of Distributed Systems*, pages 91–106, Studia Informatica Universalis, Dec. 2000.

3. H. Sundell and P. Tsigas, "Space Efficient Wait-Free Buffer Sharing in Multiprocessor Real-Time Systems Based on Timing Information," in *Proceedings of the 7th International Conference on Real-Time Computing Systems and Applications*, pages 433–440, IEEE press, Dec. 2000.

4. H. Sundell and P. Tsigas, "NOBLE: A non-blocking inter-process communication library," in *Proceedings of the 6th Workshop on Languages, Compilers and Run-time Systems for Scalable Computers*, Lecture Notes in Computer Science, Springer Verlag, Mar. 2002.

5. H. Sundell and P. Tsigas, "Fast and Lock-Free Concurrent Priority Queues for Multi-Thread Systems," in *Proceedings of the 17th International Parallel and Distributed Processing Symposium*, 11 pp., IEEE press, Apr. 2003.

6. H. Sundell and P. Tsigas, "Scalable and Lock-Free Concurrent Dictionaries," in *Proceedings of the 19th ACM Symposium on Applied Computing*, pages 1438–1445, Mar. 2004.

7. H. Sundell and P. Tsigas, "Simple Wait-Free Snapshots for Real-Time Systems with Sporadic Tasks" in *Proceedings of the 10th International Conference on Real-Time Computing Systems and Applications*, Lecture Notes in Computer Science, Springer Verlag, Aug. 2004.

8. H. Sundell and P. Tsigas, "Lock-Free and Practical Deques using Single-Word Compare-And-Swap," Computing Science, Chalmers University of Technology, Tech. Rep. 2004-02, Mar. 2004.

ACKNOWLEDGEMENTS

First of all and very respectfully, I want to thank my supervisor Philippas Tsigas, without his great mind, effort and enthusiasm this wouldn't have been possible. An encouraging, cooperative and truly interested advisor is something that every Ph.D. student wants, whereof I am one of the privileged.

I am very honored to have Prof. Dr. Friedhelm Meyer auf der Heide as my opponent. I am also honored to my grading committee, constituted of Doc. Lenka Carr-Motycková, Doc. Peter Damaschke and Prof. Bertil Svensson. Thank you all, for your helpful comments during the writing of this thesis.

I am also grateful to my examinor Peter Dybjer and my local graduate committee, constituted of Marina Papatriantafilou and Aarne Ranta, for keeping track of my work during the years. I wish to especially thank Marina for her countless efforts with giving feedback and suggestions to our papers. I also wish to thank the rest of the Distributed Computing and Systems research group for their continuous support and feedback during the years; Niklas Elmqvist, Anders Gidenstam, Phuong Hoai Ha, Boris Koldehofe, Yi Zhang and former group members. I especially thank Boris Koldehofe for being my office mate and reliable friend.

This work was enabled by the great organization called ARTES, without which I would never have got a graduate student position. ARTES have supported the major part of this work, using the national SSF foundation. I wish to thank Hans Hansson and Roland Grönroos for their endless work, and also thank the other academic and industrial members of ARTES for their support.

The inspiring and constructive environment at Chalmers has also contributed to the success of this work. I wish to thank all colleagues at the department and section, academic as well as administrative staff, faculty and other graduate students for contributing to a very fruitful and nice environment. This mostly adheres to work, but also to free time activities as skiing trips, dragonboat races, baseball matches, movies and dinners. Thank you all and keep up the good work!

Last, but not least, I wish to thank my family and all friends. I am forever grateful to my beloved wife Ann-Sofie, my son Tim, my parents Kjell and Jane, my brothers Peter and Morgan, my aunt and uncle, my parents-in-law, my other relatives and friends. I would for certain not be here without your support.

Håkan Sundell

Göteborg, November 2004.

Contents

1 Introduction **1**
1.1 Concurrent Programming Systems 1
1.2 Shared Memory 2
1.2.1 Consistency 3
1.2.2 Atomic Primitives 4
1.3 Real-Time Systems 6
1.4 Synchronization 8
1.4.1 Mutual Exclusion 9
1.4.2 Non-blocking Synchronization 10
1.4.3 Consensus 14
1.4.4 Synchronization for Real-Time Systems 14
1.5 Shared Data Structures 18
1.5.1 Advanced Atomic Primitives 18
1.5.2 Memory Management 19
1.5.3 Register 21
1.5.4 Snapshot 22
1.5.5 Linked List 23
1.5.6 Stack 24
1.5.7 Queue 24
1.5.8 Deque 25
1.5.9 Priority Queue 26
1.5.10 Dictionary 26
1.6 Contributions 27
1.6.1 Methods for Real-Time Systems 27
1.6.2 Algorithms of Shared Data Structures 30
1.6.3 Implementation Work 33
1.6.4 Design of Framework 36
1.6.5 Experiments 37

2 Wait-Free Snapshot **39**
2.1 Introduction . . . 40
2.2 System and Problem Description . . . 42
2.2.1 Real-time Multiprocessor System Configuration . . . 42
2.2.2 The Model . . . 43
2.3 The Protocol . . . 44
2.3.1 The unbounded version . . . 44
2.3.2 Bounding the Construction . . . 46
2.4 Experiments . . . 52
2.5 Conclusions and Future Work . . . 54

3 Wait-Free Shared Buffer **57**
3.1 Introduction . . . 58
3.2 System and Problem Description . . . 60
3.2.1 Real-Time Multiprocessor System Configuration . . . 60
3.2.2 The Model . . . 61
3.3 The Protocol . . . 62
3.3.1 The Unbounded Algorithm . . . 62
3.3.2 Bounding the Time-Stamps . . . 65
3.4 Implementation . . . 69
3.5 Examples . . . 71
3.6 Conclusions . . . 72

4 NOBLE **73**
4.1 Introduction . . . 74
4.2 Design and Features of NOBLE . . . 75
4.2.1 Usability-Scope . . . 76
4.2.2 Easy to use . . . 76
4.2.3 Easy to Adapt . . . 79
4.2.4 Efficiency . . . 80
4.2.5 Portability . . . 80
4.2.6 Adaptable for different programming languages . . . 81
4.3 Examples . . . 82
4.4 Experiments . . . 83
4.5 Conclusions . . . 89

5 Lock-Free Priority Queue **91**
5.1 Introduction . . . 92
5.2 System Description . . . 95
5.3 Algorithm . . . 96

5.4 Correctness . . . 104
5.5 Experiments . . . 110
5.5.1 Low or Medium Concurrency . . . 110
5.5.2 Full concurrency . . . 120
5.5.3 Results . . . 120
5.6 Extended Algorithm . . . 122
5.7 Related Work with Skip Lists . . . 127
5.8 Conclusions . . . 128

6 Lock-Free Dictionaries 131
6.1 Introduction . . . 132
6.2 System Description . . . 135
6.3 Algorithm . . . 135
6.3.1 Memory Management . . . 136
6.3.2 Traversing . . . 139
6.3.3 Inserting and Deleting Nodes . . . 142
6.3.4 Helping Scheme . . . 146
6.3.5 Value Oriented Operations . . . 147
6.4 Correctness . . . 149
6.5 Experiments . . . 159
6.6 Related Work with Skip Lists . . . 165
6.7 Conclusions . . . 166

7 Lock-Free Deque and Doubly Linked List 167
7.1 Introduction . . . 168
7.2 System Description . . . 171
7.3 The Algorithm . . . 171
7.3.1 The Basic Steps of the Algorithm . . . 174
7.3.2 Memory Management . . . 175
7.3.3 Pushing and Popping Nodes . . . 176
7.3.4 Helping and Back-Off . . . 182
7.3.5 Avoiding Cyclic Garbage . . . 185
7.4 Correctness Proof . . . 186
7.5 Experimental Evaluation . . . 194
7.6 Operations for a Lock-Free Doubly Linked List . . . 198
7.7 Conclusions . . . 204

8 Conclusions 205

9 Future Work 209

List of Figures

1.1 Three processes communicating using message passing (left) versus shared memory (right) . . . 2
1.2 Uniform versus non-uniform memory access (UMA vs NUMA) architecture for multiprocessor systems . . . 3
1.3 Example of possible scenario with concurrent read and write operations to shared memory with a relaxed consistency model. 4
1.4 The Test-And-Set (TAS), Fetch-And-Add (FAA) and Compare-And-Swap (CAS) atomic primitives. . . . 5
1.5 The "Test and Test-And-Set" spinning lock. . . . 6
1.6 Example of real-time terminology used for periodic tasks. . . 8
1.7 In a lock-free solution each operation may have to retry an arbitrary number of steps depending on the behavior of the concurrent operations. . . . 11
1.8 In a wait-free solution each operation is guaranteed to finish in a limited number of its own steps. The worst case execution time of each operation is therefore deterministic and not dependent of the behavior of concurrent operations. . . . 13
1.9 Example of timing information available in real-time system . 17
1.10 A snapshot data structure with five components . . . 22
1.11 Three processes accessing different parts of the shared linked list . . . 24

2.1 Shared Memory Multiprocessor System Structure . . . 43
2.2 Pseudocode for the Unbounded Snapshot Algorithm . . . 44
2.3 Unbounded Snapshot Protocol . . . 45
2.4 Intuitive presentation of the atomicity/linearizability criterion satisfied by our wait-free solution . . . 47
2.5 A cyclic buffer with several updater tasks and one snapshot task . . . 48

2.6 The bounded Snapshot protocol 49
2.7 Estimating the buffer length - worst case scenario 49
2.8 Pseudo-code for the Bounded Snapshot Algorithm 50
2.9 Descriptions of Scenarios for experiment 52
2.10 Experiment with 1 Scan and 10 Update processes - Scan task comparison 53
2.11 Experiment with 1 Scan and 10 Update processes - Update task comparison 54

3.1 Shared Memory Multiprocessor System Structure 61
3.2 Architecture of the Algorithm 62
3.3 The unbounded algorithm 63
3.4 The Register Structure 63
3.5 Rapidly Increasing Tags 65
3.6 Tag Range 66
3.7 Tag Value Recycling 68
3.8 Tag Reuse 69
3.9 Algorithm changes for bounded tag size 70
3.10 The registers located on each processor as a column of the matrix 71
3.11 Code to be implemented on each involved processor. 72

4.1 Experiments on SUN Enterprise 10000 - Solaris 85
4.2 Experiments on SGI Origin 2000 - Irix 86
4.3 Experiments on Dual Pentium II - Win32 87
4.4 Experiments on Dual Pentium II - Linux 88

5.1 Shared Memory Multiprocessor System Structure 95
5.2 The skip list data structure with 5 nodes inserted. 95
5.3 The Node structure. 96
5.4 Concurrent insert and delete operation can delete both nodes. 96
5.5 The Fetch-And-Add (FAA) and Compare-And-Swap (CAS) atomic primitives. 97
5.6 Functions for traversing the nodes in the skip list data structure. 99
5.7 The algorithm for the Insert operation. 100
5.8 The algorithm for the DeleteMin operation. 101
5.9 The algorithm for the RemoveNode function. 102
5.10 The algorithm for the HelpDelete function. 103
5.11 Experiment with priority queues and high contention, with initial 100 nodes, using spinlocks for mutual exclusion. 111

5.12 Experiment with priority queues and high contention, with initial 1000 nodes, using spinlocks for mutual exclusion. . . . 112
5.13 Experiment with priority queues and high contention, with initial 100 nodes, using semaphores for mutual exclusion. . . 113
5.14 Experiment with priority queues and high contention, with initial 1000 nodes, using semaphores for mutual exclusion. . . 114
5.15 Experiment with priority queues and high contention, with initial 100 or 1000 nodes 115
5.16 Experiment with priority queues and high contention, running with average 100-10000 nodes 116
5.17 Experiment with priority queues and high contention, varying percentage of insert operations, with initial 1000 (for 10-40 %) or 0 (for 60-90 %) nodes. Part 1(3). 117
5.18 Experiment with priority queues and high contention, varying percentage of insert operations, with initial 1000 (for 10-40 %) or 0 (for 60-90 %) nodes. Part 2(3). 118
5.19 Experiment with priority queues and high contention, varying percentage of insert operations, with initial 1000 (for 10-40 %) or 0 (for 60-90 %) nodes. Part 3(3). 119
5.20 Maximum timestamp increasement estimation - worst case scenario 123
5.21 Timestamp value recycling 125
5.22 Deciding the relative order between reused timestamps 125
5.23 Creation, comparison, traversing and updating of bounded timestamps. 126

6.1 Example of a Hash Table with Dictionaries as branches. . . . 134
6.2 Shared memory multiprocessor system structure 135
6.3 The skip list data structure with 5 nodes inserted. 136
6.4 Concurrent insert and delete operation can delete both nodes. 136
6.5 The Fetch-And-Add (FAA) and Compare-And-Swap (CAS) atomic primitives. 137
6.6 The basic algorithm details. 138
6.7 The algorithm for the traversing functions. 139
6.8 The algorithm for the SearchLevel function. 141
6.9 The algorithm for the Insert function. 143
6.10 The algorithm for the FindKey function. 144
6.11 The algorithm for the DeleteKey function. 145
6.12 The algorithm for the HelpDelete function. 146
6.13 The algorithm for the FindValue function. 148

6.14 Experiment with dictionaries and high contention on SGI Origin 2000, initialized with 50,100,...,10000 nodes 160
6.15 Experiment with full dictionaries and high contention on SGI Origin 2000, initialized with 50,100,...,10000 nodes 161
6.16 Experiment with dictionaries and high contention on Linux Pentium II, initialized with 50,100,...,10000 nodes 162
6.17 Experiment with full dictionaries and high contention on Linux Pentium II, initialized with 50,100,...,10000 nodes 163

7.1 Shared Memory Multiprocessor System Structure 170
7.2 The doubly linked list data structure. 171
7.3 The Fetch-And-Add (FAA) and Compare-And-Swap (CAS) atomic primitives. 172
7.4 Concurrent insert and delete operation can delete both nodes. 173
7.5 Illustration of the basic steps of the algorithms for insertion and deletion of nodes at arbitrary positions in the doubly linked list. 174
7.6 The basic algorithm details. 177
7.7 The algorithm for the PushLeft operation. 178
7.8 The algorithm for the PushRight operation. 179
7.9 The algorithm for the PopLeft function. 180
7.10 The algorithm for the PopRight function. 181
7.11 The algorithm for the HelpDelete sub-operation. 183
7.12 The algorithm for the HelpInsert sub-function. 184
7.13 The algorithm for the RemoveCrossReference sub-operation. . 186
7.14 Experiment with deques and high contention. 195
7.15 Experiment with deques and high contention, logarithmic scales. 196
7.16 The algorithm for the Next operation. 199
7.17 The algorithm for the Prev operation. 200
7.18 The algorithm for the Read function. 200
7.19 The algorithm for the InsertBefore operation. 201
7.20 The algorithm for the InsertAfter operation. 202
7.21 The algorithm for the Delete function. 203

List of Tables

1.1 An example of consensus numbers for common atomic primitives. 14

3.1 Example task scenario on eight processors 69

4.1 The shared data objects supported by NOBLE 77
4.2 The shared data objects supported by NOBLE (continued) . 78
4.3 The distribution characteristics of the random operations . . 83

Chapter 1

Introduction

This thesis deals with how to design efficient algorithms of data structures that can be shared among several execution entities (i.e. computer programs), where each execution entity can be either a *process* or a *thread*. Thus, the data structures can be accessed *concurrently* by the execution entities; either in an interleaved manner where the execution entities gets continuously *pre-empted*, or fully in *parallel*.

1.1 Concurrent Programming Systems

The definition of a modern computer suitable for concurrent programming is quite involved. Depending on the number of processors that are closely connected to each other, a computer can be either a *uni-* or *multi-processor* system. The computer system can also be constructed out of several separate computation nodes, where each node (which can be a computer system of its own, either a uni- or multi-processor) is physically separated from the others, thus forming a *distributed* system.

These computation nodes and whole computer systems can be tied together using either a *shared memory* system or a *message passing* system. Depending on how physically separated each computation node is from the others, the system is interpreted as a single computer or as a *cluster* system.

In a message passing system, the inter-communication between each computation node is done by exchanging information packages over the supported inter-connection network. A shared memory system gives a higher abstraction level, and gives the impression to the connected computation nodes of the existence of a global memory with shared access possibilities. See Figure 1.1 for an example scenario where three processes are communi-

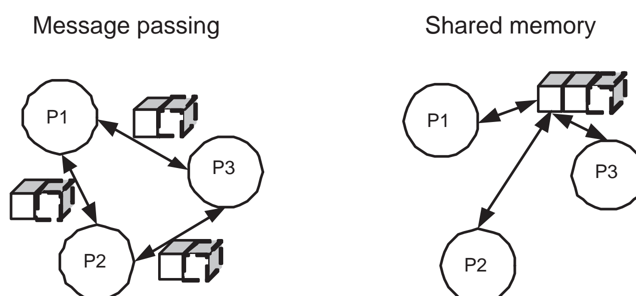


Figure 1.1: Three processes communicating using message passing (left) versus shared memory (right)

cating using message passing versus shared memory techniques.

1.2 Shared Memory

The real memory that constitutes the shared memory can be either centrally located or distributed in parts over the individual computation nodes. The inter-node communication network needed in order to establish the shared memory, can be implemented in either hardware or software on top of message passing hardware.

The shared memory system can be either *uniformly distributed* as in the uniform memory access (UMA) architecture [5] or *non-uniformly distributed* as in the non-uniform memory access (NUMA) architecture [68]. In a NUMA system, the response time of a memory access depends on the actual distance between the processor and the real memory, although it is the same for each processor on the same node. For the UMA system, the response time for memory accesses is the same all over the system, see Figure 1.2. An example of a UMA system is the Sun Starfire [24] architecture, and the SGI Origin [70] and the Sun Wildfire [42] architectures are examples of modern NUMA systems.

The shared memory implementations on computer systems with clearly physically separated computation nodes are usually called *distributed shared memory*.

As the bandwidth of the memory bus or inter-node network is limited, it is important to avoid *contention* on the shared memory if possible. Heavy contention can lead to significantly lower overall performance of the shared memory system (especially for NUMA systems), and therefore memory ac-

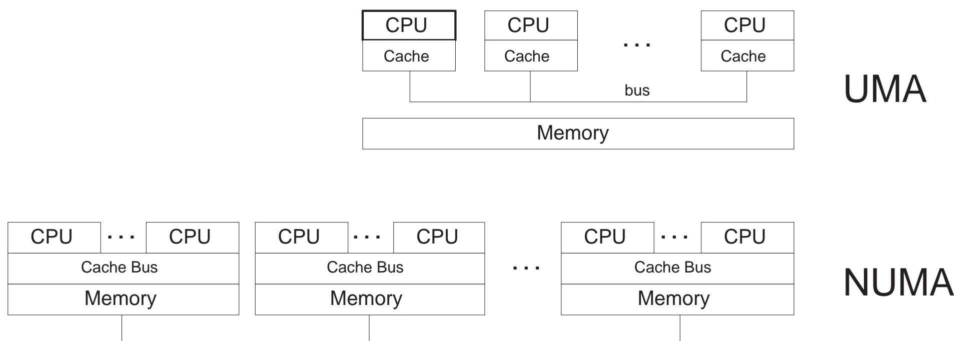


Figure 1.2: Uniform versus non-uniform memory access (UMA vs NUMA) architecture for multiprocessor systems

cesses from each processor should if possible be coordinated to work on different parts of the memory and at different points in time.

1.2.1 Consistency

In order for concurrent programs to coordinate the possibly parallel accesses to the shared memory and achieve a consistent view of the shared memory's state, it is necessary that the implementation of the shared memory guarantees the fulfillment of some formally defined memory *consistency model*.

The most intuitive and strongest model is where all concurrent memory accesses are viewed as each taking effect at a unique and the very same time by all processors, i.e. a single *total order*. However, for shared memory implementations without central hardware memory this model might be very hard to achieve. Therefore more realistic models have been proposed in the literature. In 1979, Lamport introduced the *sequential consistency* [66] model, which is one of the widest accepted models. In this model the results of any execution is the same as if the operations of all the processors were executed in some sequential order, and the operations of each individual processor appear in this sequence in the order specified by its program.

Several weaker consistency models have been proposed in the literature, for example the *casual consistency* and the *relaxed consistency* models. Many modern processor architectures implements weak memory consistency models where the view of the orders of read and writes may differ significantly from the normal intuition. This approach is taken mainly because of efficiency reasons as described by Adve and Gharachorloo [1]. For example,

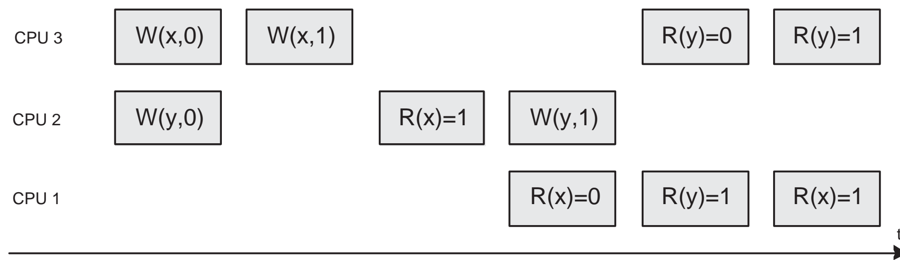


Figure 1.3: Example of possible scenario with concurrent read and write operations to shared memory with a relaxed consistency model.

see Figure 1.3 that shows a possible scenario using the default model on the Sparc v.9 processor architecture, called the *relaxed memory order*. Systems with such weak consistency models, though usually provides special machine instructions that force specified write or read operations to be viewed consistently in some decided order by other processors. However, these instructions must be used with care, as excessive use of them will significantly reduce the system's overall performance.

1.2.2 Atomic Primitives

In order to enable synchronization between processes accessing the shared memory, the system has to provide some kind of *atomic primitives* for this purpose. In this context, atomic means that no other operation can interfere with or interrupt the operation from when the operation starts until it has finished execution of all sub steps. Basic read and write operation of normal-sized memory words are usually designed to be atomic with respect to each other and other operations.

In order to make consistent updates of words in the shared memory, stronger atomic primitives than read and writes are needed in practice, although Lamport has showed how to achieve *mutual exclusion* [67] using only reads and writes. Using mutual exclusion the process is guaranteed to be alone accessing and modifying some part of the shared memory. However, using hardware atomic primitives for updates have several benefits, like that they give better and more predicted performance, as they either take full or no effect. As they are not based on explicit mutual exclusion techniques they also have better fault-tolerance, because of reasons that will be elaborated more in Section 1.4.

There are different kinds of atomic primitives available on different plat-

forms, some less powerful than the others. All platforms do not directly support all known atomic primitives; some only support a limited subset or even none. The latter is especially common on older 8-bit platforms often used for embedded systems. The functionality of all atomic primitives can though usually be implemented using other means.

The most common atomic primitives for synchronization are Test-And-Set (TAS), Fetch-And-Add (FAA) and Compare-And-Swap (CAS), which are described in Figure 1.4.

```
function TAS(value:pointer to word):boolean
    atomic do
        if *value = 0 then
            *value := 1;
            return true;
        else return false;

procedure FAA(address:pointer to word, number:integer)
    atomic do
        *address := *address + number;

function CAS(address:pointer to word, oldvalue:word
 , newvalue:word):boolean
    atomic do
        if *address = oldvalue then
            *address := newvalue;
            return true;
        else return false;
```

Figure 1.4: The Test-And-Set (TAS), Fetch-And-Add (FAA) and Compare-And-Swap (CAS) atomic primitives.

There are also other atomic primitives defined that are not as commonly available as these three. Some platforms provide an alternative to the CAS operation with a pair of operations called Load-Link / Store-Conditional (LL/SC). According to the ideal definition, the SC operation only updates the variable if it is the first SC operation after the LL operation was performed (by the same processor) on that variable. All problems that can be solved efficiently with the LL/SC operations can also be solved with the CAS operation. However, the CAS operation does not solve the so-called *ABA problem* in a simple manner, a problem which does not occur with LL/SC. The reason is that the CAS operation can not detect if a variable was read to be A and then later changed to B and then back to A by some concurrent

```
procedure Lock(lock:pointer to word)
    while true do
      if *lock = 0 then
        if TAS(lock) then break

procedure UnLock(lock:pointer to word)
    *lock := 0;
```

Figure 1.5: The "Test and Test-And-Set" spinning lock.

processes - the CAS will perform the update even though this might not be intended by the algorithm's designer. The LL/SC operations can instead detect any concurrent update on the variable between the time interval of a LL/SC pair, independent of the value of the update. Unfortunately, the real hardware implementations of LL/SC available are rather weak, where SC operations might fail spuriously because of prior shared memory accesses or even undefined reasons, and LL/SC pairs may not be nested by the same processor.

All the previous primitives are used to make consistent updates of one variable. There are also atomic primitives defined for two and more variables, like Double-Word Compare-And-Swap (CAS2) [49], not to be confused with CAS for double word-size (i.e. 64-bit integers on 32-bit systems). CAS2 (often called DCAS in the literature) do not exist as an implementation in real hardware on any modern platform, the only architecture that supported it was the Motorola 68020 family of processors [62].

Although the hardware atomic primitives might be very practical, they must be used with care as they can cause heavy contention on the shared memory if massively used in parallel on the same parts of the memory. As mentioned in Section 1.2, heavy contention can lead to significantly reduced performance of the shared memory system. See Figure 1.5 for an example of an implementation of a spinning lock, that avoids unnecessary calls of the expensive TAS atomic primitive when the lock is occupied and instead spins on a local read operation.

1.3 Real-Time Systems

In the field of computer science scientists have always worked towards making computing more reliable and correct. However, the interpretation of correctness is not always the same. A real-time system is basically a system

where the timing of a result is as important as the result itself. A real-time system has to fulfill some timing guarantees, i.e. the computation time of some specific computation may never exceed some certain limit.

The importance of these timing guarantees can vary. In a *hard* real-time system the fulfillment of all these guarantees is critical for the existence and purpose of the whole system. In a *soft* real-time system it is acceptable if some of the guarantees are not met all the time.

The computational work in a real-time system is usually divided into parts called *tasks*. A task can be the equivalent to either a process or a thread. The execution order of these tasks is controlled by a special part of the operating system kernel, the *scheduler*. Each task has a set of timing and urgency parameters. These parameters and requirements are examined by the scheduler either before starting (off-line) or while running (on-line); depending on the result the scheduler then pre-empts and allows the different tasks to execute. Whether the scheduling is done off-line or on-line depends on the actual scheduler used and also if dynamic adding of new tasks to the system is allowed or not.

Tasks can be classified in different categories, *periodic*, *sporadic* or even *aperiodic*. Periodic tasks are tasks that should be performed over and over again with a certain time interval between each execution, for example we should perhaps perform 50 computations per second of image frames to produce smooth video output. Sporadic tasks arrive not more often than a certain time interval and aperiodic tasks can arrive at any time and at any rate.

Because of the need of some tasks to be ready earlier than others, tasks are given priorities that will be used by the scheduler. These can be either fixed for each task or be dynamically changed during run-time.

The terminology we use for specifying the properties of periodic real-time tasks are as follows:

T - Period. The period describes how often the task arrives, ready to be executed.

p - Priority. This is usually a unique value given to each task, with lower value describing higher priority, starting with 0 for the highest priority.

C - Worst case execution time. The task has a bound on how long it can possibly take to be fully executed in case of no interruptions.

R - Worst case response time. This is the longest time it can possibly take for a task to be finally executed after it has arrived ready for execution.

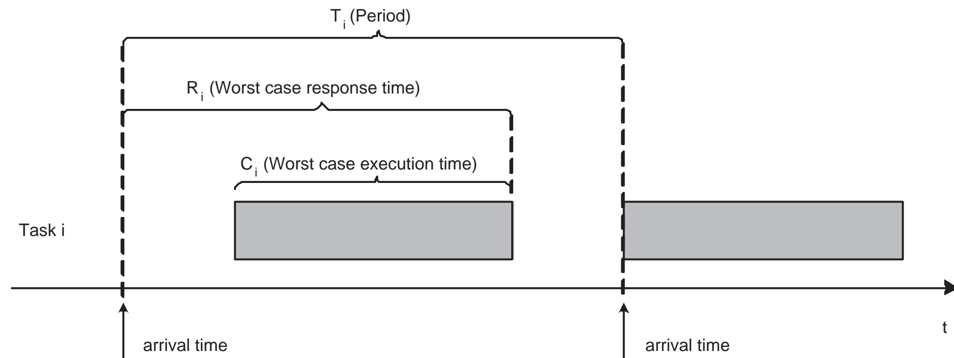


Figure 1.6: Example of real-time terminology used for periodic tasks.

See Figure 1.6 for an example task with the corresponding scheduling information marked in a timing diagram.

If using fixed priority scheduling with periodic tasks, the worst case response times [17] for each task can be computed using fixed point iteration as follows:

$$R_i = C_i + \sum_{j \in hp(i)} \left\lceil \frac{R_i}{T_j} \right\rceil C_j \tag{1.1}$$

The intuitive description of this formula is basically that the response time of a task is the execution time of the task plus the maximum time it can possibly be delayed by other tasks that have higher priority ($hp(i)$ is the set of tasks having higher priority than task i).

1.4 Synchronization

In a concurrent programming system we usually have several tasks, executed more or less in parallel. These tasks together form sub-tasks of a greater task, the purpose of the whole system. As these tasks are related to each other in some way with shared resources, they have to *synchronize* in some manner. Without synchronization it is highly possible and probable that the concurrent access of some shared resource will lead to inconsistency, as most shared resources don't allow more than one participant to access them concurrently in a predictive manner.

1.4.1 Mutual Exclusion

The general way of synchronizing the accesses of resources is to use *mutual exclusion*. Mutual exclusion can be implemented in several ways, one method is using message passing, perhaps having a token system that distributes the accesses in a safe and predictive manner. Another way of many available to implement mutual exclusion is to disable interrupts or similar.

Mutual exclusion is often provided by the operating system as a primitive and mostly incorporates the support of shared memory. Using shared memory the tasks can synchronize and share information in shared data structures. Using mutual exclusion the tasks can make sure that only one task can have access to a shared resource or data structure at one time.

However, mutual exclusion (also called locking) has some significant drawbacks:

- They cause *blocking*. This means that tasks that are eligible to run have to wait for some other task to finish the access of the shared resource. Blocking also makes the computation of worst-case response times more complicated, and the currently used computation methods are quite pessimistic. Blocking also can cause unwanted delays of other tasks as the effect propagates through the task schedule, also called the *convoy effect*.
- If used improperly they can cause *deadlocks*, i.e. no task involved in the deadlock can proceed, possibly because they are waiting for a lock to be released that is owned by a task that has died or is otherwise incapable of proceeding.
- They can cause *priority inversion*. The exclusion of other tasks while one low priority task is holding the lock can cause a high priority task to actually have to wait for middle priority tasks to finish.

These problems have been recognized since long, especially for real-time systems, and are thoroughly researched. Several solutions exist, such as software packages that fit as a layer between the mutual exclusion control of the shared resource and the scheduler of the real-time system. The most common solutions are called the *priority ceiling protocol* (PCP) and *immediate priority ceiling protocol* (IPCP) [18, 71, 93, 95]. They solve the priority inversion problem and limits the amount of blocking a task can be exposed to in a uni-processor system. There are also solutions, though significantly less efficient, for multi-processor systems like the *multi-processor priority ceiling protocol* (MPCP) [93].

1.4.2 Non-blocking Synchronization

Researchers have also looked for other solutions to the problems of synchronizing the access of shared resources. The alternative to using locks is called *non-blocking*, and circumvents the usage of locks using other means than mutual exclusion.

As non-blocking algorithms do not involve mutual exclusion, all steps of the defined operations can be executed concurrently. This means that the criteria for consistency correctness are a bit more complex than for mutual exclusion. The correctness condition used for concurrent operations (in general) is called *linearizability*. Linearizability was formally defined by Herlihy and Wing [50] and basically means that for each real concurrent execution there exists an equivalent imaginary sequential execution that preserves the partial order of the real execution.

The fulfillment of the linearizability property enables the shared resources to still be accessed in a predictive manner, called *atomic*. In this context, atomic means that the operation can be viewed by the processes as it occurred at a unique instant in time, i.e. the *effect* of two operations can not be viewed as taking place at the same time.

Traditionally, there are two basic levels of non-blocking algorithms, called *lock-free* and *wait-free*. Common with most non-blocking algorithms are that they take advantage of atomic primitives. However, there are solutions available that do not rely on atomic primitives other than read or write as will be shown further on in the thesis.

The definitions of lock-free and wait-free algorithms guarantee the progress of always at least one (all for wait-free) operation, independent of the actions performed by the concurrent operations. As this allows other processes to even completely halt, lock-free and wait-free operations are in this sense strongly fault tolerant. Recently, some researchers also proposed *obstruction-free* [48] algorithms to be non-blocking, although this kind of algorithms do not give any progress guarantees.

Several studies exist that indicate a possibly superior performance for non-blocking implementations compared to traditional ones using mutual exclusion, as for example the study by Michael and Scott [85] of common data structures and simple applications, and the study by Lumetta and Culler [73] of a simple queue structure. Non-blocking techniques can significantly improve the systems' overall performance as showed by Tsigas and Zhang [115][118], on both application as well as operating system level. Several successful attempts to incorporate non-blocking techniques into operating system kernels has been made, for example the Synthesis OS by Massalin

and Pu[75] and the Asterix real-time OS by Thane et al. [113].

Lock-Free

Roughly speaking, lock-free is the weakest form of non-blocking synchronization. Lock-free algorithms are designed with having the idea in mind that synchronization conflicts are quite rare and should be handled as exceptions rather than a rule. However, it must then always be possible to detect when those exceptions appear. When a synchronization conflict is noticed during a lock-free operation then that operation is simply restarted from the beginning, see Figure 1.7.

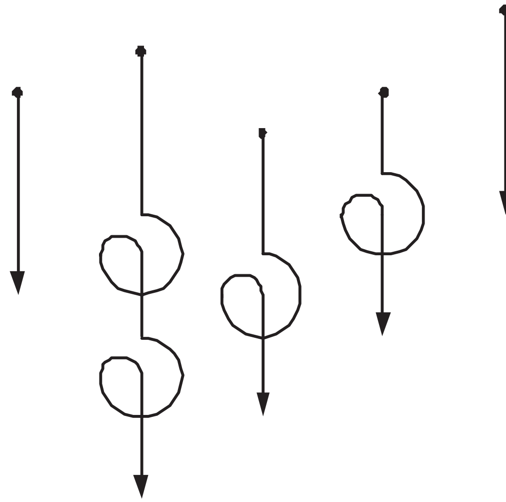


Figure 1.7: In a lock-free solution each operation may have to retry an arbitrary number of steps depending on the behavior of the concurrent operations.

The basic ideas of most lock-free algorithms for updating shared data objects are as follows:

1. Prepare a (partly) copy of the object you want to update.
2. Update the copy
3. Make the copy valid using some strong atomic primitive like Compare-And-Swap (CAS) or similar. The CAS operation makes sure that the update from the old to the new value is done in one atomic step, unless

a conflict with some other operation occurred. If a conflict is noticed, then retry from step 1 again.

This of course assumes that the shared object is constructed using a dynamic structure with linked pointers for each separate node. There exist other lock-free algorithms that rely on other means than swinging pointers with strong atomic primitives. One example is a lock-free algorithm for a shared buffer [65] that only relies on the orders of read and write operations.

Although lock-free algorithms solve the priority inversion, deadlock and blocking problem, other problems might appear instead. For instance, it might be that a race condition arises between the competing tasks. In the worst case, the possible retries might continue forever and the operation will never terminate, causing the operation to suffer from what is called *starvation*. It is important to note that starvation also can occur with many implementations of locks as the definition of mutual exclusion does not guarantee the *fairness* of concurrent operations, i.e. it is not guaranteed that each process will get the lock in a fair (with respect to the response times for other processes) or even limited time.

One way to lower the risk of race conditions as well as the contention on the shared memory is called *back-off* [46]. This means that before retrying again after a failed attempt to update, the operation will sleep for a time period selected according to some method. The length of this time period could for example be set initially to some value dependent on the level of concurrency, and then increase either linearly or exponentially with each subsequent retry. Another method that could be combined with the previous or just by it self is randomization.

Because of the possibility for starvation, lock-free techniques stand alone are not directly suitable for hard real-time systems. However, if you have enough information about your environment, you might be able to efficiently bound the maximum number of retries.

Wait-Free

The idea of Wait-free constructions goes one step further than lock-free and also avoids the unbounded number of retries. This means that any wait-free operation will terminate in a finite number of execution steps regardless of the actual level of concurrency, see Figure 1.8.

Many wait-free algorithms are quite complex. There are several main ideas behind designing a wait-free algorithm for a shared data object. Many of the methods are based on copying the object into different buffers and then

redirecting the concurrent operations into separate buffers to avoid conflicts. The buffer with the most recent value is then usually activated using pointers and some synchronization among the operations. This synchronization can be e.g. some handshaking protocol or invocation of some strong atomic primitive like CAS.

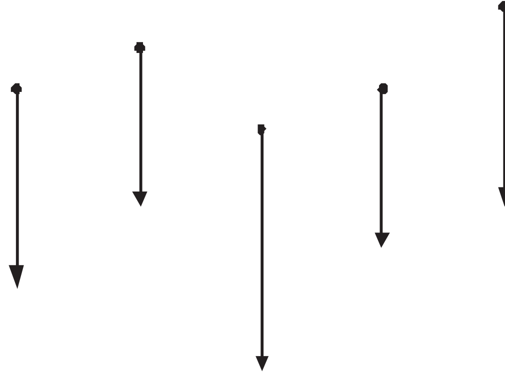


Figure 1.8: In a wait-free solution each operation is guaranteed to finish in a limited number of its own steps. The worst case execution time of each operation is therefore deterministic and not dependent of the behavior of concurrent operations.

Another method for implementing wait-free solutions is called *helping*. In the context of real-time task sets with priorities, helping is quite similar to the PCP protocol in the actual execution pattern, and works like follows:

1. Each operation, name it e.g. $operation_i$, first announces some information about what it is going to do with the shared data object in some global data structure.

2. Then this global data structure is checked for other operations that are announced and have not completed e.g. due to pre-emption. The announced operations are then performed by $operation_i$. This procedure is performed in several steps, of which each is verified using an atomic primitive.

3. When all announced operations are helped, $operation_i$ is performed in the same way as the helping of the other operations. As all the other concurrent operations also follow the helping scheme, it can be that

Table 1.1: An example of consensus numbers for common atomic primitives.

Atomic Primitive	Consensus Number
Read & Write	1
Test-And-Set (TAS)	2
Fetch-And-Add (FAA)	2
Compare-And-Swap (CAS)	∞
Load-Linked/Store-Conditionally (LL/SC)	∞

$operation_i$ has been helped itself, thus the helping scheme works in a recursive manner.

This method requires that the operation first must be split into several steps that fulfill some special properties that are needed to guarantee consistency between the concurrent operations that try to help with this step. It has been proved that using the helping scheme it is possible to implement a wait-free algorithm for every possible shared data structure. This universal construction [45] is unfortunately not practically usable for all cases, which emphasizes the need for specially designed wait-free algorithms for each specific shared data object.

1.4.3 Consensus

The relative power of each atomic primitive has been researched by Herlihy [45] and others. Atomic primitives with lower *consensus number* can always be implemented using atomic primitives of higher consensus number. Roughly speaking, the consensus number describes how many concurrent operations of the same kind that can wait-free agree on the same value of the result using this operation. See Table 1.4.3 for an example of consensus numbers for common atomic primitives.

Herlihy [45] has shown that atomic primitives with infinite consensus numbers are universal, i.e. they can be used to construct wait-free implementations of any shared data structure.

1.4.4 Synchronization for Real-Time Systems

When considering the characteristics and limits of non-blocking algorithms, one has to keep in mind that they are usually designed for the universal case. If we take advantage of information about the behavior of the specific systems that we are going to use, the possibilities for designing efficient non-blocking algorithms can drastically improve.

There are two basic kinds of information to take advantage of, either by looking at certain aspects of the scheduler, or by using timing information about the involved tasks. Looking at two kinds of schedulers, we can categorize the use of information to three methods:

- Priority scheduling. The priority scheduling policy means that tasks can only be pre-empted by tasks with higher priority. The task with highest priority will therefore not experience any pre-emptions, thus ensuring atomicity of this task.

- Quantum scheduling. The quantum scheduling policy means that there is a granularity in the possible times when tasks can be pre-empted. This can be of advantage in the design of the algorithms as the number of possible conflicts caused by concurrent operations can be bounded.

- Timing information. If all involved tasks in the real-time system have well-specified timing characteristics of the execution by the scheduler, this can be used for bounding the amount of possible interactions between the concurrent tasks, as will be shown further on in this thesis. Usually the scheduler has this information when it decides if a task is schedulable or not.

The Effect of Priority Scheduling

If we have a system where each task has a different priority assigned to it, we can make use of this when designing non-blocking data structures. The actual problem that we have when concurrently accessing shared data structures between tasks occurs when two or more tasks access the data structure at the same time. In a uni-processor system this can only occur when a task that is in a critical section is pre-empted by a higher priority task. Seen from the perspective from the pre-empted task, the higher priority task is atomic. As all tasks have different priorities, one task must have the highest priority of all the tasks sharing the data structure. This highest priority task will not be pre-empted by other tasks accessing the data structure, and can therefore be seen as atomic.

In this way one can design a non-blocking agreement protocol, using the idea that at least one task will always proceed and finish its critical section without interference. This has been done by Anderson et al. [10] who have been researching thoroughly in this area.

One can also apply the ideas of helping schemes that were introduced by Herlihy [45] and others. If tasks that get pre-empted by higher priority tasks get helped by the pre-empting task to finish the operation, we can make a helping scheme for uni-processor systems without using strong atomic primitives like CAS. As we always have a highest priority task in the system, we can assure that all tasks will be fully helped. This idea is for example used by Tsigas and Zhang [117] in their research.

The Effect of Quantum Scheduling

One problem when designing non-blocking shared data structures is that you have to consider the possibility the algorithm being pre-empted anywhere and also infinitely often. If we have some information about how often the task can be pre-empted, we could make a simpler design.

If the scheduler is quantum-based that means that we have some limits on how often the scheduler can pre-empt the running task. After the first pre-emption and when the task starts executing again, the task is guaranteed not to be pre-empted until a certain time has passed, the quantum.

This means that we could design our algorithm in a way so that we need to anticipate at most one pre-emption. If we could recognize that our algorithm got pre-empted we can safely run a certain number of program statements without the risk of conflicts. This kind of constructions has been proposed by Anderson et al. [10].

Anderson et al. [9] and others have also looked at how to extend the results from using priority and quantum scheduling to also apply for multi-processor systems. It seems that the consensus hierarchy defined by Herlihy [45] now collapses. A lower consensus object can be used for making consensus with more processes than this consensus object is defined for according to Herlihy.

The available consensus objects available in real-time systems are in reality either low consensus objects (like read or write which has a consensus number of 1 or TAS which have consensus number of 2) or infinite consensus objects (like CAS or LL/SC). The lack of middle-between consensus objects makes the priority and quantum scheduling design methods quite unpractical on multi-processor systems, as strong atomic primitives like CAS have to be used although a lower consensus object is actually needed.

The Effect of Using Timing Information

In most real-time systems, tasks are defined to be periodic and have deterministic behavior, running under a scheduler with static priority. This applies to both uni- and multi-processor systems. This means that we have information about worst-case execution times, worst-case response times, periods and priorities, see Figure 1.9.

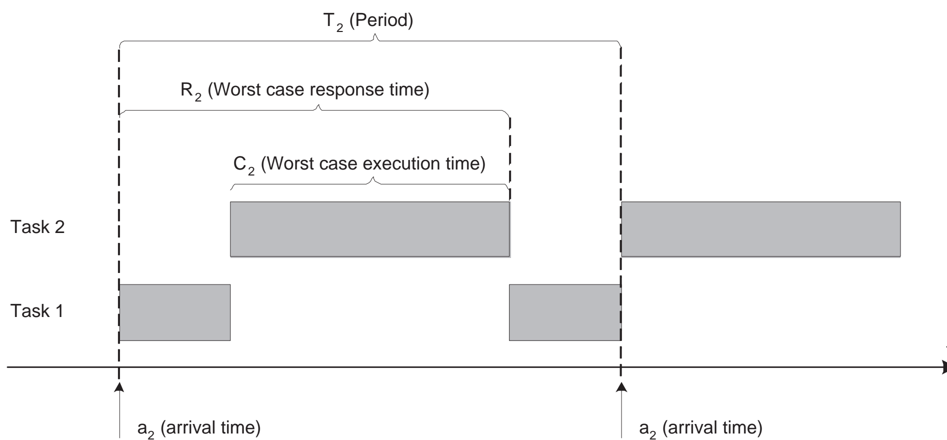


Figure 1.9: Example of timing information available in real-time system

When designing a non-blocking algorithm, it could be useful to know about the number of possible conflicts during a certain time interval. As we have the timing information about each task available we could use this information to simplify the non-blocking algorithms. Chen and Burns [25] used timing information to bound the buffer sizes needed to implement a wait-free shared buffer using no atomic primitives besides reads and writes.

Using the available timing information, the risk of possible starvation for lock-free algorithms could be eliminated, thus allowing lock-free algorithms for use in hard real-time systems. Anderson et al. [11] [12] has designed methods for bounding the number of retries of lock-free algorithms for uni-processor systems, and Tsigas and Zhang [119] have designed an adoption scheme for multiprocessor systems.

1.5 Shared Data Structures

Tasks that cooperate in a concurrent system need some kind of communication to synchronize and share information necessary for the execution. One method used is message passing and another is shared memory. Using shared memory, tasks can share variables that can be grouped together into bigger structures of information and thus achieve a higher abstraction level.

The data structures commonly used in sequential programs need to be modified in order to be used in concurrent systems. Modifications of shared data structures are done in several steps. If these steps are interleaved with modifications from other tasks, this can result in inconsistency of the data structure. Therefore the data structure needs to be protected from other tasks modifying it while the operation is executing. This can either be done using mutual exclusion or non-blocking methods.

We have studied how common data structures and abstract data types can be adopted to multiprocessor systems. The focus has been on efficient and practical non-blocking implementations and we have compared them to corresponding solutions using mutual exclusion.

There exists several general schemes for how to implement any data structure to be non-blocking, for example the universal schemes by Herlihy [46] and by Anderson and Moir [14][13]. Prakash et al. [89] and Turek et al. [120] developed schemes to substitute the locks in a concurrent algorithm with non-blocking replacements that use helping techniques and unbounded tags. Other researchers have developed methods for adopting existing sequential implementations to be non-blocking. Barnes presented a lock-free adoption method [20] and Shavit and Touitou introduced the concept of *software transactional memory* [98]. However, as the universal non-blocking design and adoption schemes add very significant or even huge overhead, they are not suitable for practical applications, and perform significantly worse than corresponding non-blocking implementations that are designed and optimized for a specific purpose. This has been verified by empirical studies, for example Fraser [32] has done experiments comparing transactional memory based solutions with optimized and specifically designed implementations.

1.5.1 Advanced Atomic Primitives

Many algorithms in the literature rely on more powerful atomic primitives than what is actually available in hardware in common and modern computer architectures. One probable major reason for this is that designing

and reasoning about non-blocking algorithms can often be very complex, and raising the abstraction level using powerful building blocks (seen as black boxes) could significantly lower the observable complexity.

One set of atomic primitives that is often used in the literature of non-blocking algorithms, is the Load-Linked/Validate-Linked/Store-Conditional (LL/VL/SC) set of operations. As explained earlier in Section 1.2.2, these operations only exist as weak implementations and only on some platforms. Therefore attempts have been made to implement those operations in software using available atomic primitives, like for example CAS. Wait-free implementations of the LL/VL/SC operations have been proposed by Israeli and Rappoport [56], Anderson and Moir [14], Moir [87] and by Jayanti [58] [59]. Unfortunately all of the proposed implementations have significant drawbacks that reduce their usefulness in practice, as they are either using unbounded tags that besides from being unreliable also reduces the value domain, or use excessive amount of memory. Michael [81] recently presented non-blocking implementations of LL/VL/SC that require significantly less amount of memory by using the hazard pointer (see Section 1.5.2) type of garbage collection. In order to be wait-free, the presented algorithm though requires unbounded tags and extended CAS operations of double-word size.

One other well researched topic is software implementations of the Multi-Word Compare-And-Swap (CASN) operation. The CASN operation can atomically update an arbitrary number of memory words of arbitrary addresses. Implementations of the CASN operation are often given in combination with replacement implementations for the read and write operations of the affected memory words. A wait-free implementation has been proposed by Anderson and Moir [14], and several lock-free implementations based on the LL/VL/SC primitives have been proposed by Israeli and Rappoport [56], Moir [88], and Ha and Tsigas [41][40]. All these implementations have the drawback that they limit the value domain which size is inversely proportional to the number of processes. A lock-free implementation based directly on CAS exist by Harris et al. [43] and limits the value domain with 2 bits of the memory word size.

1.5.2 Memory Management

In order for an implementation to be non-blocking, the underlying operations have to be non-blocking as well. Implementations of dynamic sized shared data structures have to rely on the ability to allocate and free memory dynamically in a way that is consistent with concurrent accesses to the memory (i.e. memory can not be freed if some process is accessing, or is

about to access, that part). Reliable memory management methods could also be very effective in solving the ABA problem (see Section 1.2.2) that sometimes occurs with the use of the CAS atomic primitive in the design of non-blocking algorithms. Unfortunately, current systems' default dynamic memory allocation support is based on mutual exclusion, as well as the majority of the available garbage collection schemes.

Hesselink and Groote [52, 53] presented a wait-free implementation for memory management of a set of shared tokens. However, the solution is very limited and can not be used for implementing shared data structures in general.

Valois [122] have presented, with later correction by Michael and Scott [83], practical methods for lock-free dynamic allocation and lock-free reference counting. The methods add significant overhead as they are based on the FAA and CAS atomic primitives, and have the drawbacks that the memory has to be pre-allocated (before the lock-free application starts) and can not be used for arbitrary purposes as the shared counters used for reference counting have to be present indefinitely. The size of each memory block that can be dynamically allocated is also fixed in the presented implementation. Detlefs et al. [27] presented a lock-free reference counting method that can allow the freed memory to be re-used for arbitrary usage, but is unfortunately based on the non-available (in any modern platform) CAS2 atomic primitive.

Michael [78] [80] introduced another approach using *hazard pointers*. The method is lock-free, efficient as it is only based on atomic read and writes, and can guarantee that memory that are currently referenced (i.e. the process has a pointer to it) and possibly accessed by a process will not be concurrently freed. The time complexity of freeing memory for possible garbage collection is amortized to be low, as the relatively large job of scanning all the shared hazard pointers are only done periodically. Thus, the response times of a delete operation on a dynamic data structure that uses garbage collection, might vary much more drastically when using hazard pointers than when using reference counting. Consequently, this behavior might be of significant concern to real-time systems. The method also has the drawback that it can not guarantee that pointers from structures in the memory itself are pointing to memory that will not be concurrently freed, as guaranteed by for example reference counting techniques. Herlihy et al. [47] have presented a similar method to hazard pointers that instead are based on the CAS2 atomic primitive.

Michael [82] recently presented a lock-free dynamic memory allocation scheme that can handle arbitrary sizes of allocated blocks. Gidenstam et

al. [34] have presented a lock-free dynamic memory allocation scheme with similar properties using a different approach. However, these schemes are only truly lock-free up to a certain block size (as they then call the system's default memory allocation scheme) and can possibly incur a large overhead in the total amount of memory reserved compared to what is actually allocated. Additionally, in order to be used in practice for supporting dynamic data structures, the allocation schemes have to be combined with a suitable garbage collection scheme.

1.5.3 Register

One of the simplest things that is often necessary to share between different nodes in a system is a single value, via a *register* or memory word. The shared memory system often supports multiple readers and writers to access each register in an atomic and consistent way. However, there may be cases where shared memory is not provided or is limited in some way. The constraint may be the number of direct communication paths between each node in a multi-processor system, or it may even be that some communication paths are significantly slower than the others and should therefore be avoided.

One efficient wait-free algorithm for a shared register was constructed by Vitanyi et al. [124], where a n-reader n-writer shared register could be constructed by a set of simple 1-reader 1-writer registers (in practice implemented by using message passing and local memory). The initial algorithm though had a practical drawback by the use of unbounded time-stamps, which also severely limits the size of the register's value domain. Israeli and Shaham [57] presented an algorithm of a n-reader n-writer shared register that is optimal in the respect of space and time complexity. However, the implementation constructed of 1-reader 1-writer registers is only sketched and is likely to impose a large overhead which limits its suitability for practical applications.

In many practical applications, the size of an individual register might not suffice and the application therefore needs to atomically update a multiple of memory words, i.e. a shared *buffer*. Several implementations of non-blocking buffers have been proposed in the literature. For example, Lamport [65] presented a non-blocking buffer using only atomic read and writes, and with constraints on the number of concurrent writers. Larsson et al. [69] have designed an algorithm of a wait-free buffer for a single writer and multiple readers using stronger atomic primitives. Tsigas and Zhang [114] constructed a general n-reader and m-writer non-blocking shared buffer with optimal space requirements, using the FAA and CAS atomic primitives

and with constraints of assumed available real-time scheduling information. One simple and straightforward implementation of a general shared buffer would be to use a single shared pointer that are updated atomically using CAS to point to the currently active buffer version, where each version could be allocated dynamically and freed consistently (together with the pointer de-referencing operation) using some garbage collection mechanism.

1.5.4 Snapshot

Taking snapshots is a very important operation in real-time systems. Snapshots are used in many embedded and distributed real-time systems e.g. the brake control system of a car, where the brake on each wheel has to know the state of the other wheels at the same instant in time. A snapshot is a consistent instant view of several (often distributed) shared variables that are logically connected. Using snapshots, all involved processors receive consistent views of the system.

A snapshot object supports two operations, the first is to write to a single register (called update) and the other is to read multiple registers (scan). Each of these operations must be performed in one atomic step, see Figure 1.10. Each shared value is represented as a part of the snapshot structure, called the component.

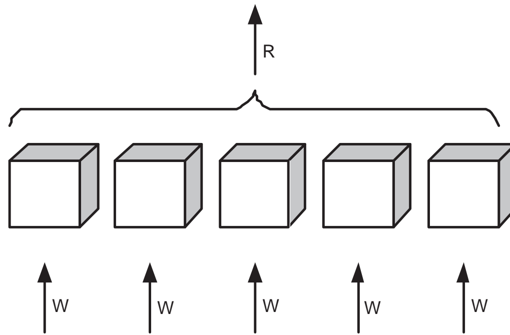


Figure 1.10: A snapshot data structure with five components

There exist several implementations of non-blocking snapshot objects in the literature. For example, a general wait-free n-reader m-writer snapshot algorithm based on atomic read and writes was constructed by Anderson [6] [7] [8] and a similar construction was also presented by Afek et al. [2]. However, these implementations have quite high space requirements and are

computationally expensive. Kirousis et al. [63] presented more efficient solutions with the constraints of allowing a single reader. They also presented efficient solutions with the additional constraint of allowing a single writer per component. Ermedahl et al. [29] have presented an even more efficient solution for the same constraints using the TAS atomic primitive.

1.5.5 Linked List

A linked list is a general data structure that can be used to implement several abstract data types. It is a dynamic data structure, where each individual node is ordered uniquely relatively to each other (i.e. each node has two neighbor nodes, one previous and one next). The linked list can be either singly linked or doubly linked. In a *singly linked list* each node contains information in the form of a reference to the next node, and in a *doubly linked list* each node also has a reference to the previous node. New nodes can be inserted anywhere, arbitrary nodes can be deleted and the whole list can be traversed from one end to the other.

The adoption of the linked list concept to a concurrent environment, adds quite some complexity compared to the sequential case. For example, each process can traverse the list independently of each other and modify arbitrary parts of the list concurrently, see Figure 1.11. In the sequential case, the current position in the list while traversing could be represented by an index that is based on counting the number of preceding nodes from the start of the list. In the concurrent case, this kind of index can not be used, and the only possible reference is the current node of traversal itself. The situation gets even more complicated if the node that is referenced gets deleted, and thus does not have a clearly defined predecessor or successor node. The commonly accepted solution is to treat a reference to a deleted node as being invalid, i.e. re-start the traversal from the start or end of the list.

Valois [123][122] has constructed a lock-free implementation of a singly linked list using the CAS atomic primitive, and also introduced the *cursor* concept in order to keep track of each process' relative position while traversing the list. The presented linked list implementation is based on auxiliary nodes that exists between each real node and relies on strong garbage collection. Harris [44] presented a more efficient solution, that eliminates the need for auxiliary nodes as it can update the next pointers atomically together with a deletion mark using the standard CAS operation. This is accomplished by using the often otherwise unused least significant bits of the pointer representation (e.g. in most 32-bit systems memory references

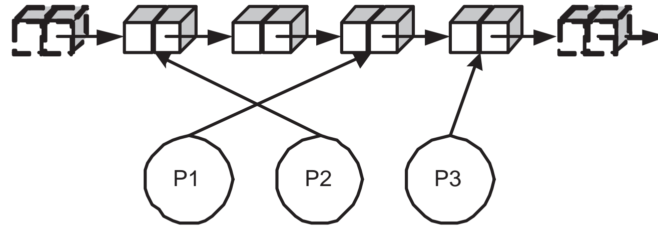


Figure 1.11: Three processes accessing different parts of the shared linked list

are forced to be aligned by 4 bytes). The presented solution though has the drawback that traversing operations has to restart from the start of the list in case of concurrent deletions at the point of traversal.

Valois [122] also presented a lock-free implementation of a doubly linked list using multiples of auxiliary nodes and the CAS atomic primitive. However, the presented implementation is not general and complete as it lacks the possibility to delete nodes. Greenwald [38] presented a general lock-free doubly linked list implementation based on the non-available CAS2 atomic primitive.

1.5.6 Stack

The *stack* abstract data type is a last-in first-out (LIFO) type of buffer. Even though it intuitively has most applications in the sequential case, it has also applications in the concurrent case (e.g. keeping shared lists of free items) and several non-blocking implementations have been proposed in the literature. For example Valois [122] presented a lock-free implementation of a stack that is based on linked lists and uses the CAS atomic primitive. Michael [78] has presented a more efficient solution that is compatible with the hazard pointers type of garbage collection.

1.5.7 Queue

The *queue* abstract data type is a first-in first-out (FIFO) type of buffer. Shared queues are fundamental and have a vast number of applications, as for example when streaming information in producer and consumers scenarios or for implementing the *ready-queue* for multi-processor scheduling.

Herman and Damian-Iordache [51] outlined a wait-free implementation of a shared queue. However, the proposed implementation is not practical

because of its very high time complexity and limitations on the number of nodes.

Valois [121][122] makes use of linked lists in his lock-free implementation which is based on the CAS primitive. Prakash et al. [90] also presented an implementation using linked lists and the CAS primitive, though with the drawback of using unbounded tags that besides from being unreliable also limits the maximum number of nodes. Michael and Scott [84] presented a more efficient solution using similar techniques. Michael [78] later presented a modified solution eliminating the need of tags by using the hazard pointer type of garbage collection.

Gong and Wing [35] and later Shann et al. [97] presented a lock-free shared queue based on a cyclic array and the CAS primitive, though with the drawback of using unbounded tags that besides from being unreliable also limits the size of the value domain. Tsigas and Zhang [116] presented a more efficient solution without the previous drawbacks, and uses a different approach for solving the ABA problem.

As JAVA has approached the real-time community and introduced real-time tasks, the need for wait-free solutions with shared data structures like queues has been identified. Tsigas and Zhang [117] have recently constructed a wait-free (for either the consumer or producer side, not both) queue that meets the demands from the real-time specification for JAVA (RTJ) [21], which makes use of the fact of present priority based scheduling and therefore can do without any strong atomic primitives.

1.5.8 Deque

The *deque* (i.e. double-ended queue) abstract data type is interesting as it unifies the concepts of stacks and queues. It is a fundamental data structure and can be used for example to implement the ready queue used for scheduling of tasks.

Several lock-free implementations have been proposed in the literature. Arora et al. [15] presented a lock-free deque implementation especially suited for scheduling purposes. It is based on the CAS atomic primitive, and has the significant drawback that it does not support all operations and only allows a limited level of concurrency. Greenwald [37] presented a general implementation of a lock-free deque and there is also a publication series of another deque implementation [3],[28] with the latest version by Martin et al. [74]. However, both of the previous implementations are based on the non-available CAS2 atomic primitive.

Michael [79] has developed a general lock-free deque implementation

based on the CAS primitive. However, it provides a limited level of parallelism as all operations have to synchronize, even when they operate on different ends of the deque. Moreover, for some practical purposes it requires a special extended version of the CAS operation that can atomically operate on two adjacent words, which is not available on all modern platforms.

1.5.9 Priority Queue

The *priority queue* abstract data type extends the queue concept with priorities on each individual node. Shared priority queues are fundamental data structures and have many applications. For example, the ready-queue that is used in the priority based scheduling of tasks in many real-time systems can often be implemented using a concurrent priority queue.

Israeli and Rappoport [55] have presented a wait-free algorithm for a shared priority queue. This algorithm makes use of the Multi-Word Store-Conditionally atomic primitive which does not exist except for inefficient software implementations. Greenwald [37] has presented an outline for a lock-free priority queue based on the non-available CAS2 atomic primitive. However, there exists an attempt for a wait-free algorithm by Barnes [19] that uses existing atomic primitives, though this algorithm does not comply with the generally accepted definition of the wait-free property. The algorithm is not yet implemented and the theoretical analysis predicts worse behavior than the corresponding sequential algorithm, which makes it of little or no practical interest.

1.5.10 Dictionary

The *dictionary* abstract data type allows associations of values with keys to be stored and searched for. The data structure is often designed in combination with *hash tables* where each branch (from each entry in the hash table) is also a dictionary data structure. The standard implementation of the standard spreading component of a hash-table is usually wait-free as it is based on an array structure by definition.

Valois [122] presented a general lock-free dictionary data structure that are based on an ordered linked list and using the CAS primitive. The path using concurrent ordered lists has been improved by Harris [44], and later Michael [77] presented a significant improvement by using the hazard pointer type of garbage collection. This combined solution of hash-tables and ordered linked lists was lately improved by Shalev and Shavit [96] that allows the hash-table to dynamically increase in size (though not possible to shrink)

in a lock-free manner.

However, the use of ordered linked lists allow only linear time complexity for searching, and in the sequential case the use of tree structures can achieve a logarithmic search time complexity. Valois [122] presented an incomplete idea of how to design a concurrent *skip list* data structure, which would allow search operations to achieve a probabilistic logarithmic time complexity.

Gao et al. [33] recently presented a lock-free algorithm of a hash-table data structure with fully dynamic size (if the algorithm is provided with a suitable lock-free memory management), that store all items inside of the hash-table and thus does not explicitly have branches. However, for keys that will generate the same hash-index, those items are stored consecutively at other (possibly unoccupied) hash-indices and thus force a linear searching procedure. The algorithm is quite involved and promises an amortized time complexity of constant, and has quite high memory requirements that are proportional to the number of processes (i.e. every process may temporary need two copies each of the whole hash-table). Moreover, the algorithm does not support updates of already inserted associations.

In contrast to a sequential dictionary, where the update operation might be redundant, a specially defined update operation is necessary for doing updates on a concurrent dictionary abstract data type, as consecutive calls of delete and insert operations for the purpose of updating will not be consistent with concurrent operations.

1.6 Contributions

The contributions of this thesis can roughly be divided into the following categories, that are inter-connected and dependent on each other:

- **Methods for Real-Time Systems.**
- **Algorithms of Shared Data Structures.**
- **Implementation Work.**
- **Design of Framework.**
- **Experiments.**

1.6.1 Methods for Real-Time Systems

We have studied how timing information available in real-time systems can be used to improve and simplify non-blocking algorithms for shared data

structures. The focus has been on designing methods for bounding the needed buffer sizes and also recycling and bounding of shared time-stamp values, in hard real-time systems where the timing information is reliable.

Some non-blocking algorithms, which have been proposed in the literature, use buffers that grow arbitrarily large with time and thus require infinitely large buffers. However, in order for the algorithms to be implemented in practice, the buffers have to be of fixed or at least of bounded sizes. Therefore, the number of possible simultaneously used memory cells in the buffer has to be bounded and unused memory cells have to be recycled. The work on the bounding of buffer sizes and the recycling of buffer cells has resulted in one new method:

- **A method for bounding the size of a shared buffer.** The shared buffer of interest is concurrently accessed by periodic or sporadic tasks. The accesses to the individual buffer cells are controlled and limited by a shared index, which is possibly incremented by the tasks, thus there is always a deterministic span of indexes of cells that can be accessed at any time. We have presented an analysis method that is specialized for an algorithm of a wait-free snapshot data structure. The resulting new algorithm, after applying the analysis and algorithmic changes for cyclic buffer accesses, requires only atomic read and write operations on the shared memory, and is described in Chapter 2.

Several non-blocking algorithms, which have been proposed in the literature, use time-stamps that can take arbitrary high values. However, in order for the algorithms to be implemented in practice, those time-stamps must be able to be represented in memory words of fixed size. Therefore, the possible span of the time-stamp values has to be bounded and unused time-stamp values have to be recycled. The work with recycling and bounding of time-stamps has resulted in two new methods:

- **A method for bounding of time-stamps used in a fixed number of shared variables.** The time-stamps are used as values for comparison in a fixed number of variables shared by periodic tasks. Continuously newer time-stamps are introduced by the tasks and are compared to other time-stamps present in the shared variables, and as tasks finish their execution, older time-stamps will cease to exist in the system. Thus, there is always a deterministic span of values of the present time-stamps. We have designed an algorithm for recycling and consistent comparison of possibly recycled time-stamps, such that

the needed domain size of the new representation of time-stamps is twice the maximum span of possible time-stamp values in the system at any instance in time. We have presented an analysis of a wait-free algorithm by Vitanyi et al. [124] of a shared register, that can determine the maximum span of possible values of present time-stamps at any time. The resulting new algorithm, after applying the algorithmic changes for the new comparison, is described together with the analysis in Chapter 3.

- **A method for bounding of time-stamps used in a dynamic number of shared variables.** The time-stamps are used as values for comparison in a dynamic number as well as a fixed number of variables shared by periodic tasks. The purpose of the comparisons is to decide if the time-stamps of the dynamic variables are newer or older than the time-stamps in the fixed variables. Continuously, newer time-stamps are introduced by the tasks and are compared to other time-stamps present in the shared variables, and as tasks finish their execution, older time-stamps will cease to exist in the fixed variables - but are not guaranteed to cease to exist in the dynamic variables. Thus, the span of values of the present time-stamps is non-deterministic. We have designed an algorithm to deterministically modify the time-stamps in a set of dynamic variables (that are traversable and limited), such that the comparison will give the same results, but the span of values of present time-stamps will be deterministic. We have designed an algorithm for recycling and consistent comparison of possibly recycled time-stamps (including a special value that represents an infinitely high time-stamp), such that the needed domain size of the new representation of time-stamps is about twice the maximum span of possible time-stamp values in the system at any instance in time. We have presented an analysis of a lock-free algorithm of a priority queue, that given the dynamic time-stamp variables are periodically updated using the new algorithm, can determine the maximum span of possible values of present time-stamps at any time. The resulting new algorithm, after applying the algorithmic changes for the new comparison and periodic updating of dynamic time-stamps, are described together with the analysis in Chapter 5.

The above two results of analyzing the maximum span of present time-stamps at any time in the system, can be generalized in the following equation (assuming that each task invocation can increase the newest time-stamp with a value of at most 1):

$$MaxSpan = \sum_{i=0}^{n} \left(\left\lceil \frac{\max_{v \in \{0..\infty\}} LT_v}{T_i} \right\rceil + 1 \right) \tag{1.2}$$

Where LT_v denotes the maximum life-time that the time-stamp with value v exists in the system, and T_i denotes the period of task i.

1.6.2 Algorithms of Shared Data Structures

We have designed new algorithms and implementations for the following shared data structures and abstract data types:

- Wait-Free Shared Register.
- Wait-Free Snapshot.
- Lock-Free Skip List.
- Lock-Free Priority Queue.
- Lock-Free Priority Queue with Real-Time Properties.
- Lock-Free Dictionary.
- Lock-Free Doubly Linked List.
- Lock-Free Deque.

Using the new methods for utilizing the timing information available in real-time systems, see Section 1.6.1, we have designed the following new algorithms of shared data structures:

- **Wait-Free Shared Register.** We have designed a new version of the wait-free algorithm by Vitanyi et al. [124] of a shared register for multiple readers and writers. The new version uses bounded time-stamps and is suitable for real-time systems with periodic tasks. The algorithm is given together with implementation details using message passing methods, and is described in Chapter 3.
- **Wait-Free Snapshot.** We have designed a new version of the wait-free algorithm by Kirousis et al. [63] of a snapshot data structure for one reader and multiple writers per component. The new version uses a buffer of bounded size for each of the components, and is suitable for real-time systems with periodic as well as sporadic tasks. The algorithm is described in Chapter 2.

- **Lock-Free Priority Queue with Real-Time Properties.** We have designed a new version of the lock-free algorithm by Sundell and Tsigas of a priority queue data structure. The new version uses time-stamps to enable specific semantic properties of the operations with respect to each others timing, and is suitable for real-time systems with periodic tasks. The algorithm is given together with implementation details, and is described in Section 5.6.

We have designed several new lock-free algorithms of advanced shared data structures. The structures are dynamic and therefore rely on dynamic memory allocation as well as reliable garbage collection facilities. The algorithms are in addition to being efficient, also practical as the algorithms are described including important and non-trivial implementation level details. In order to obtain the highest possible parallelism, the overall strategy of the algorithms is to only synchronize the concurrent operations when they fundamentally affect overlapping parts of the data structure. The given implementation details specially address the memory management scheme by Valois [122] and corrected by Michael and Scott [83], but can be easily adapted to any sufficient scheme with similar properties and interface. The algorithms have not only been tested on real hardware, but are also given with rigorous and mathematical style proofs, that give evidence for the linearizability and lock-free properties. The new lock-free algorithms for general purpose are as follows:

- **Lock-Free Skip List.** We have designed the first[1] lock-free algorithm of a concurrent skip list data structure that is based on the CAS atomic primitive. The main observation used, is that although the skip list can have arbitrarily many links from each node in the data structure, only one link on each node determines globally the nodes state as present or not in the data structure. These links form a singly linked list, which could be updated atomically but separately from the other links, whose purpose is of increasing the performance. We have used the basic ideas of Harris [44] singly linked construction that uses CAS, together with additional links that point backwards in order to give hints for the concurrent operations that possibly are helping deletions in progress.

[1] Our results were submitted for reviewing in October 2002 and published as a technical report [105] in January 2003. It was officially published in April 2003 [106], receiving a best paper award, and an extended version was also published in March 2004 [110]. Very similar constructions have appeared in the literature afterwards, by Fraser in February 2004 [32], Fomitchev in November 2003 [30] and Fomitchev and Ruppert in July 2004 [31]

Moreover, the actual values of each node, represented with a pointer to an arbitrary object, are updated using CAS and also effects the global interpretation of each node's state as being present or not in the structure. The algorithm is given together with implementation as well as performance improving details, and is described in the context of its derived realizations of abstract data types in Chapters 5 and 6.

- **Lock-Free Priority Queue.** Using our lock-free skip list construction, we have designed a new algorithm of the priority queue abstract data type. In order for our skip list construction to be used as a priority queue, the first node on the lowest level in the skip list is treated as the node of minimum priority (as the nodes at the lowest level in the skip list are always sorted according to their keys by definition, and keys can be used for representing the priority). As in our basic skip list construction, the state of presence in the structure is represented by the value, which enables each node to be atomically updated or deleted. The operations use CAS for the atomic updates, even though our skip list construction can not verify that a node is the first at the very same moment as it gets conceptually deleted. This intuitive need to verify and change two separate parts of the global state in one atomic step, would imply the need for stronger atomic primitives that could operate on a multiple of arbitrary located memory words. The proper use of our skip list construction for priority queue operations is enabled by the observation that linearizability does not imply that it must be possible to observe and verify all involved properties of the global state in one atomic step. In fact, the linearizability property depends on what actually is observed by the concurrent operations, and not on what possibly could have been observed. The algorithm, together with implementation details, is described in Chapter 5.

- **Lock-Free Dictionary.** Using our lock-free skip list construction, we have designed a new algorithm of the dictionary abstract data type. To use our skip list data structure as a dictionary is straight-forward from its definition. In addition we have also designed value-oriented search and delete operations, in order to achieve the full dictionary capability. The algorithm takes advantage of certain properties of the supporting memory management scheme in order to improve the performance of traversing operations. The algorithm is given together with implementation details, and is described in Chapter 6.

- **Lock-Free Doubly Linked List.** We have designed the first[2] lock-free algorithm of a concurrent doubly linked list data structure that is based on the CAS atomic primitive. The algorithm is based on an asymmetric approach, where the data structure is treated as a singly linked list with auxiliary links backwards. Both the forward and backward links are updated consistently using CAS, combined with helping schemes that make sure that the data structure always converges to a perfect doubly linked list. The algorithm supports reliable traversals through both directions, including traversals through deleted nodes. In order to avoid possible cyclic garbage when deleting, which can not be handled properly by garbage collector schemes based on reference counting, we have designed a scheme that makes sure that cyclic garbage can never occur. The algorithm is given together with implementation details, and is described in Chapter 7 with the general operations specially described in Section 7.6.

- **Lock-Free Deque.** Using our lock-free doubly linked list construction, we have designed a new algorithm of the deque abstract data type. The deque operations are implemented by inserting and deleting nodes at the ends of the doubly linked list. The operations use CAS for the atomic updates, even though our doubly linked construction can not verify that a node is the first at the very same moment as it gets conceptually deleted. This intuitive need to verify and change two separate parts of the global state in one atomic step, would imply the need for stronger atomic primitives that could operate on a multiple of arbitrary located memory words. The proper use of our doubly linked list construction for deque operations is enabled by the observation that linearizability does not imply that it must be possible to observe and verify all involved properties of the global state in one atomic step. In fact, the linearizability property depends on what actually is observed by the concurrent operations, and not on what possibly could have been observed. The algorithm is given together with implementation details, and is described in Chapter 7.

1.6.3 Implementation Work

All algorithms in this thesis have been implemented on actual hardware. In order to give fair comparisons and also to provide with a effective platform

[2] Our results were submitted to PODC for reviewing in February 2004 and published as a technical report [109] in March 2004.

for further developments, we have also implemented a variety of other algorithms in the literature as well as basic procedures. The implementation work has also been a substantial resource for the correct development of our algorithms, as it has given extensive feedback as well as worthful insights into the complexity of the algorithm design. The implementation work can roughly be divided into the following parts:

- **Atomic primitives.** As some modern architectures of shared memory only provide relaxed memory consistency, we have implemented atomic read and write primitives for the Sun Sparc v8/v9 architectures. These procedures are explicitly called from higher level implementations when atomic read/write operations are absolutely necessary. We have also implemented a variety of atomic primitives of higher consensus level in assembly language, using known and self-developed designs on the Sparc v8/v9 for Sun Solaris, Mips III for SGI Irix V6.5 and the Intel IA-32 for Linux and Win-32 architectures. The atomic primitives include the Test-And-Set (TAS), Fetch-And-Add (FAA) with or without return of previous value, Compare-And-Swap (CAS) with a boolean result or the previous value returned, unconditional Swap and the Load-Linked/Validate-Linked/Store-Conditional (LL/VL/SC) primitives. These primitives have been implemented using the hardware primitives provided by the specific architectures as well as by using substitution schemes [87] when necessary.

- **Multi-word atomic primitives.** Some algorithms require more powerful atomic primitives like the Double-Word-Compare-And-Swap (CAS2), which are not supported in hardware by any modern architecture. Therefore we have implemented the algorithm of a Multi-Word-Compare-And-Swap (CASN) primitive by Harris et al. [43] that is based on CAS, and also the CASN algorithm by Ha and Tsigas [40] [41] that is based on LL/VL/SC.

- **Memory management.** All non-blocking algorithms of dynamic data structures are dependent on underlying non-blocking schemes for dynamic memory allocation and garbage collection. This is normally not supported by the programming environment, as for example in Java which employs blocking stop-the-world techniques for garbage collection. We have implemented the lock-free CAS-based memory management and garbage collection scheme by Valois [122] and corrected by Michael and Scott [83]. The garbage collection scheme is

based on reference counting and is tightly connected to the memory allocation scheme, which is based on a fixed-size free-list of fixed node size. We have also implemented the lock-free reference counting scheme by Detlefs et al. [27] that is compatible with an arbitrary memory allocation scheme. As this garbage collector scheme is based on CAS2, it was implemented using our CASN implementation. Additionally, we have implemented the garbage collection scheme that uses CAS and a fixed number of hazard pointers by Michael [78] [80], together with an implementation of a free-list of fixed node size.

- **Shared data structures.**
  - *Register.* We have implemented the wait-free algorithm of a shared register for real-time systems, that is presented in Chapter 3.
  - *Snapshot.* We have implemented the following algorithms of a snapshot data structure: 1) the wait-free algorithm for a single reader and a single writer per component by Ermedahl et al. [29]; 2) the wait-free algorithm of a single reader and multiple writers per component by Kirousis et al. [63]; 3) the wait-free algorithm for real-time systems and a single reader and multiple writers per component, that is presented in Chapter 2; 4) a single-lock based algorithm for multiple readers and multiple writers per component.
  - *Linked List.* We have implemented the following algorithms of a singly linked list data structure: 1) the lock-free algorithm of a singly linked list data structure, by Valois [123] [122]; 2) a lock-free algorithm of a singly linked list data structure, derived from the construction by Harris [44]; 3) a single-lock based algorithm. We have implemented the following algorithms of a doubly linked list data structure: 1) the lock-free algorithm without deletion support, by Valois [122]; 2) the lock-free algorithm that is presented in Chapter 7; 3) a single-lock based algorithm.
  - *Stack.* We have implemented the following algorithms of the stack abstract data type: 1) the lock-free algorithm by Valois [122]; 2) a single-lock based algorithm.
  - *Queue.* We have implemented the following algorithms of the queue abstract data type: 1) the lock-free algorithm by Valois [121] [122]; 2) the lock-free algorithm by Tsigas and Zhang [116]; 3) a single-lock based algorithm.

- *Deque.* We have implemented the following algorithms of the deque abstract data type: 1) the lock-free algorithm by Martin et al. [74] [28] [3]; 2) the lock-free algorithm by Michael [79]; 3) the lock-free algorithm that is presented in Chapter 7; 4) a single-lock based algorithm.
- *Priority Queue.* We have implemented the following algorithms of the priority queue abstract data type: 1) the lock-free algorithm that is presented in Chapter 5; 2) the lock-free algorithm for real-time systems, that is presented in Section 5.6; 3) a single-lock based algorithm; 4) the multiple-lock based algorithm by Hunt et al. [54]; 5) the multiple-lock based algorithm by Lotan and Shavit [72]; 6) the multiple-lock based algorithm by Jones [60] that requires semaphores.
- *Dictionary.* We have implemented the following algorithms of the dictionary abstract data type: 1) the lock-free algorithm by Michael [77]; 2) the lock-free algorithm that is presented in Chapter 6; 3) a single-lock based algorithm.

1.6.4 Design of Framework

Many of the non-blocking algorithms for shared data structure that we have implemented in this thesis, have different interface and requirements on the environment. We have designed a framework that tries to unify all implementations such that it has all the following features: easy to use, efficient and portable. The framework is provided in the form of a software library.

For each data structure in the framework, we have designed a common interface that covers the functionality of all involved implementations, and as well tries to hide unnecessary complexity by abstraction. In order to fit the common interface, some implementations have been extended with more functionality than what was originally described in the literature.

The abstraction of the user interface is also designed in order to make it easy to change the actual implementation used for a particular data structure, in a way that is transparent to the application. In order to be efficient, the software library is implemented in standard C and assembly languages, with the aim of achieving a minimum overhead. In order to be portable, it is designed in a layered structure where system dependent and independent features are separated. The software library has been successfully implemented on the Sun Sparc Solaris, Mips SGI Irix and Intel IA-32 for Linux and Win-32 platforms.

The software library called NOBLE, which also has been released to the academic community via a public web-site, is described in Chapter 4.

1.6.5 Experiments

A majority of the implementations presented in this thesis have been evaluated empirically using experiments on actual hardware, in separate studies together with other related implementations. The experiments have been performed for benchmarking as well as for testing purposes. The major mean for measuring the relative performance that have been used in this thesis, is micro-benchmarks specialized for each data structure.

Different algorithms have, because of their different designs, different performance characteristics, that might also depend on the setting and target architecture. Some algorithms might benefit from a special distribution of the operations, while others might do the opposite. In order to be as fair as possible for a general review, we have used randomized scenarios with certain distributions of the involved operations. After the creation of each scenario, each of the implementations is executing concurrent operations according to the sequence given by the very same scenario. In order to reduce the possibility for influences due to cache contents between the implementations, the caches of the involved CPU's are normalized before each test run. The number of threads involved is varied between 1 and a maximum, and thus covers the un-contented as well as contented cases, with or without preemptions. The experiments are repeated a chosen number of times, and an average of the execution time (i.e. real-world clock time) of the scenarios with each setting are calculated.

In order to be as accurate as possible, and to avoid possible interference from nondeterministic factors, the experiments have been performed with very low or no other load on the systems. The multiprocessor systems that we have had full or partly access to during the experimental work in this thesis, are the following:

- **Sun Enterprise 10000.** This multiprocessor machine is based on the uniform memory architecture (UMA) called Starfire, and was equipped with 64 Ultrasparc processors. The machine was running the Sun Solaris operating system.

- **SGI Origin 2000.** This multiprocessor machine is based on the non-uniform memory archtecture (NUMA) with a cache-coherency protocol to provide consistency. We had access to two different machines,

one with 64 and the other with 40 (later reduced to 29) processors. The machines were running the SGI Irix v6 operating system.

- **Sun Ultra 80.** This multiprocessor machine is UMA based, and was equipped with 4 Ultrasparc processors. The machine was running the Sun Solaris operating system.

- **Compaq Server PC.** This multiprocessor machine is UMA based, and was equipped with 2 Pentium II processors. The machine was running the Linux or Windows NT operating systems.

In our experiments with the lock based algorithms, we mainly used the "Test and Test-And-Set" implementation for the locks. In some experiment settings with a higher degree of preemptions, and thus involving settings where semaphores could be relevant, we also used the default system semaphore implementations.

The results from our experiments show significant performance improvements relative to the lock based as well as other non-blocking implementations that have appeared in the literature for the specific data structures. The results of the experiments performed on the data structures in this thesis are presented in Chapters 2, 4, 5, 6 and 7.

Chapter 2

Simple Wait-Free Snapshots for Real-Time Systems with Sporadic Tasks[1]

Håkan Sundell, Philippas Tsigas
Department of Computing Science
Chalmers Univ. of Technol. and Göteborg Univ.
412 96 Göteborg, Sweden
E-mail: {phs, tsigas}@cs.chalmers.se

Abstract

A wait-free algorithm for implementing a snapshot mechanism for real-time systems is presented in this paper. Snapshot mechanisms give the means to a real-time task to read a globally consistent set of variable values while other concurrent tasks are updating them. Such a mechanism can be used to solve a variety of communication and synchronization problems, including system monitoring and control of real-time applications. Typically, implementations of such mechanisms are based on interlocking. Interlocking protects the consistency of the shared data by allowing only one process at a time to access the data. In a real-time environment locking typically

[1]This is a revised version of the paper presented at RTCSA 2004 [111] and published as a technical report [108], partly based on prior work presented at OPODIS 2000 [112].

leads to difficulties in guaranteeing deadlines of high priority tasks because of the blocking. Researchers have introduced non-blocking algorithms and data structures that address the above problems. In this paper we present a simple and efficient wait-free (non-blocking) snapshot algorithm by making use of timing information that is available and necessary to the scheduler that schedules the tasks of real-time systems. Experiments on a SUN Enterprise 10000 multiprocessor system show that the algorithm that we propose here, because of its simplicity, outperforms considerably the respective wait-free snapshot algorithm that is not using the timing information.

2.1 Introduction

In any multiprocessing system co-operating processes share data via shared data structures. To ensure consistency of the shared data structures programs typically rely on some form of software synchronization. In this paper we are interested in designing a shared data structure for co-operative tasks in real-time multi-processor systems allowing processes to read a globally consistent set of variable values while other concurrent tasks are updating them.

The challenges that have to be faced in the design of inter-task communication protocols for multi-process systems become more delicate when these systems have to support real-time computing. In real-time multi-process systems inter-task communication protocols i) have to support sharing of data between different tasks; ii) must meet strict time constraints, the HRT deadlines; and iii) have to be efficient in time and in space since they must perform under tight time and space constraints.

The classical, well-known and most simple solution when designing shared data structures enforces mutual exclusion. Mutual exclusion protects the consistency of the shared data by allowing only one process at a time to access the data. Mutual exclusion i) causes large performance degradation especially in multiprocessor systems [100]; ii) leads to complex scheduling analysis since tasks can be delayed because they were either preempted by other more urgent tasks, or because they are blocked before a critical section by another process that can in turn be preempted by another more urgent task and so on, (this is also called as the convoy effect) [64]; and iii) leads to priority inversion in which a high priority task can be blocked for an unbounded time by a lower priority task [95]. Several synchronization protocols have been introduced to solve the priority inversion problem for uniprocessor [95] and multiprocessor [92] systems. The solution presented

in [95] solves the problem for the uniprocessor case with the cost of limiting the schedulability of task sets and also making the scheduling analysis of real-time systems hard. The situation is much worse in a multiprocessor real-time system, where a task may be blocked by another task running on a different processor [92].

To address the problems that arise from blocking, researchers have proposed non-blocking implementations of shared data structures. Two basic non-blocking methods have been proposed in the literature, *lock-free* and *wait-free*. Lock-free implementations of shared data structures guarantee that at any point in time in any possible execution some operation will complete in a finite number of steps. In cases with overlapping accesses, some of them might have to repeat the operation in order to correctly complete it. This implies that there might be cases in which the timing may cause some process(es) to have to retry a potentially unbounded number of times, leading to an unacceptable worst-case behavior for hard real-time systems. However, they usually perform well in practice. In wait-free implementations each task is guaranteed to *correctly* complete any operation in a *bounded* number of its own steps, regardless of overlaps and the execution speed of other processes; i.e. while the lock-free approach might allow (under very bad timing) individual processes to starve, wait-freedom strengthens the lock-free condition to ensure individual progress for every task in the system.

Non-blocking implementation of shared data objects is a new alternative approach for the problem of inter-task communication. Non-blocking mechanisms allow multiple tasks to access a shared object at the same time, but without enforcing mutual exclusion to accomplish this. Non-blocking inter-task communication does not allow one task to block another task, and gives significant advantages over lock-based schemes because:

1. it can not cause priority inversion, avoids lock convoys that make scheduling analysis hard and delays longer.

2. it provides high fault tolerance (processor failures will never corrupt shared data objects) and eliminates deadlock scenarios from two or more tasks both waiting for locks held by the other.

3. and more significantly it completely eliminates the interference between process scheduling and synchronization.

Non-blocking protocols on the other hand have to use more delicate strategies to guarantee data consistency than the simple enforcement of

mutual exclusion between the different operations on the data object. These new strategies on the other hand, in order to be useful for real-time systems, should be efficient in time and space in order to perform under the tight space and time constraints that real-time systems demand.

In this paper we show how to exploit information that is part of the special nature of the real-time systems in order to design a simple snapshot algorithm with one scanner that is efficient in time and space as needed. The algorithm that we propose here outperforms significantly — due to its simplicity — the respective one not using this information [4, 29]. Experiments on a SUN Enterprise 10000 has shown that the new construction gives 4 times better response time for the update operations for all scenarios and with 20 % better response time for all practical settings. Please notice that we have one scan task at a time and multiple concurrent update tasks per component in multiple components.

Previously Chen and Burns in [25], exploited the use of the same information for the construction of a non-blocking shared buffer. Research at the University of North Carolina [9, 10] and [94] by Anderson et al. has shown that wait-free algorithms can be simplified considerably in real-time systems by exploiting the way that processes are scheduled for execution in such systems. In [4, 29] it has also been shown that wait-free methods actually can be very efficient and relatively low demanding in memory consumption. In our experimental evaluation of the protocol we compare with this solution.

The rest of this paper is organized as follows. In Section 2 we describe the computer systems that we consider and give a description of the problem. Section 3 presents our protocol and later we show how to bound the size of the buffers used in the algorithm. Section 4 shows some experiments. The paper concludes with Section 5.

2.2 System and Problem Description

2.2.1 Real-time Multiprocessor System Configuration

A typical abstraction of a shared memory multi-processor real-time system configuration is depicted in Figure 2.1. Each node of the system contains a processor together with its local memory. All nodes are connected to the shared memory via an interconnection network. A set of co-operating tasks[2] (processes) with timing constraints are running on the system performing their respective operations. Each task is sequentially executed on one of the

[2]throughout the paper the terms *process* and *tasks* are used interchangeably

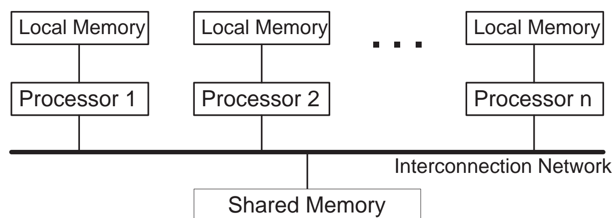


Figure 2.1: Shared Memory Multiprocessor System Structure

processors, while each processor can serve (run) many tasks at a time. The co-operating tasks now, possibly running on different processes, use shared data objects built in the shared memory to co-ordinate and communicate. Every task has a maximum computing time and has to be completed by a time specified by a deadline. Tasks synchronize their operations through read/write operations to shared memory.

2.2.2 The Model

In this paper we are interested in the snapshot problem or snapshot object, which involves taking an "instantaneous" picture of a set of variables, all in one atomic operation. The snapshot is taken by one task, the *scanner*, while each of the snapshot variables may concurrently and independently be *updated* by other processes (called *updaters*). A snapshot object is also called a *composite register*, consisting of a number of *components* (indexed 1 through c), which constitute the entities which can be updated and snapshot. We will use the two terms (snapshot object and composite register) interchangeably.

The accessing of the shared object is modeled by a history h. A history h is a finite (or not) sequence of operation invocation and response events. Any response event is preceded by the corresponding invocation event. For our case there are two different operations that can be invoked, a snapshot operation or an update operation. An operation is called complete if there is a response event in the same history h; otherwise, it is said to be pending. A history is called complete if all its operations are complete. In a global time model each operation q "occupies" a time interval $[s_q, f_q]$ on one linear time axis ($s_q < f_q$); we can think of s_q and f_q as the starting and finishing time instants of q. During this time interval the operation is said to be *pending*. There exists a precedence relation on operations in history denoted

```
// Global variables
snapshotindex: integer
value[NR_COMPONENTS][∞]: valtype
// Local variables
tempindex,k,index: integer

procedure Update(cid:integer, data:valtype)
U1    value[cid][snapshotindex]:=data;

procedure Scan(snapshotdata[NR_COMPONENTS]: valtype)
S1    tempindex:=snapshotindex;
S2    snapshotindex:=tempindex+1;
S3    for k:=0 to NR_COMPONENTS-1 do
S4        for index:=tempindex to 0 step -1 do
S5            if value[k][index] ≠ NIL then
S6                snapshotdata[k]:=value[k][index];
S7                break;
```

Figure 2.2: Pseudocode for the Unbounded Snapshot Algorithm

by $<_h$, which is a strict partial order: $q_1 <_h q_2$ means that q_1 ends before q_2 starts; operations incomparable under $<_h$ are called *overlapping*. A complete history h is linearizable if the partial order $<_h$ on its operations can be extended to a total order $\rightarrow_{\mathrm{h}}$that respects the specification of the object [45].

2.3 The Protocol

2.3.1 The unbounded version

We first start with a simple unbounded snapshot protocol that first appeared in Kirousis et.al. [63]. The protocol uses buffers of infinite length, the architecture of this protocol is shown in figure 2.3. The pseudo-code for the algorithm is presented in figure 2.2. The architecture of our unbounded construction is as follows: For each component $k = 0, \ldots, c-1$, we introduce an unbounded number of subregisters $value[k][l]$, $l = 0, \ldots, \infty$ which are written to by the updater of the corresponding component and are read by the scan function. We call these subregisters **memory locations**. The second index of each memory location $value[k][l]$ is its address (the first indicates the corresponding component). A memory location holds a value

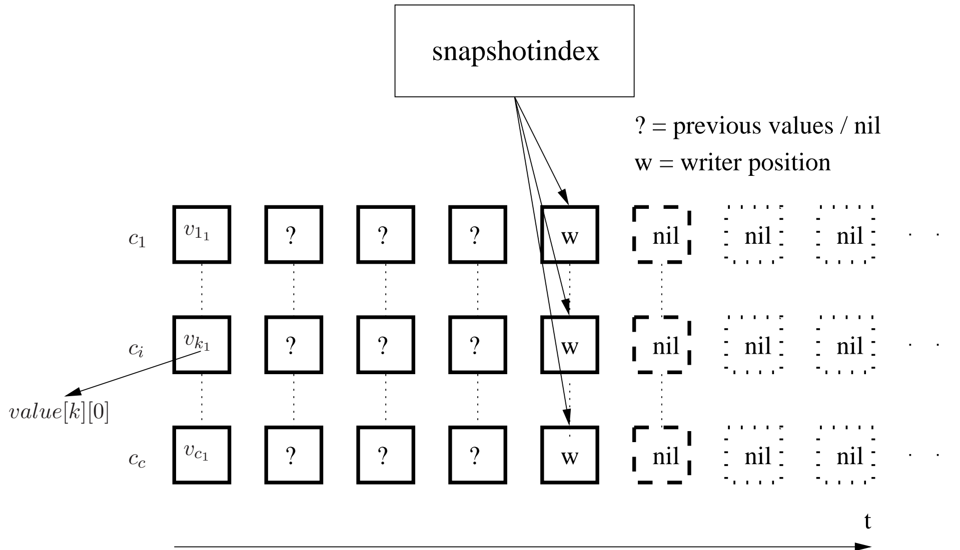


Figure 2.3: Unbounded Snapshot Protocol

that belongs either to the set of values of the corresponding component or is a special new value denoted by **NIL**. The type of all these values is denoted by *valtype*. We call them **component values**. Initially, the subregisters $value[k][l]$ for $k = 0, \ldots, c-1$ and $l = 1, \ldots, \infty$ hold the value **NIL**, while the subregisters $value[k][0]$, $k = 0, \ldots, c-1$ hold a value from the set of values of the corresponding component. Moreover, we introduce a subregister *snapshotindex* which holds as value an integer (a pointer to a memory location). This subregister can be written to by the scanner and can be read by all updaters. It is initialized with the value 0.

In the protocol the scanner is the controller: it is the one who determines where the updaters must write. All that an updater has to do is to write its value to the memory location forwarded by the scanner through a pointer. More specifically, the protocol works as follows: An updater first reads *snapshotindex* and then writes its value to the memory location of the corresponding component that is pointed to by *snapshotindex*. The scanner, on the other hand, first increments *snapshotindex* by one; stores its old value into a local variable *tempindex* and then for each component $k = 0, \ldots, c-1$ gets the value to be returned by reading $value[k][tempindex], \ldots, value[k][0]$ in this order until it gets a value which is not **NIL**. The scanner, by forwarding to the updater, with its very first sub-operation, a new subregister,

which it does not use again during the current snapshot, it succeeds to avoid reading values written by update operations that started after its own starting point. Moreover, the scanner, by scanning the subregisters in the reverse order from the one that they were forwarded in previous operations and by returning the first "non-empty" value, it achieves to return non overwritten values.

The snapshot protocol presented here is based on the following idea: if each scan returns for each component a value which is not overwritten (cf. figure 2.4(a)) and which is written by an update which started before the start of the scan (cf. figure 2.4(b,c)), then the solution satisfies an atomicity criterion [7, 8, 63] that enables us to argue for each component separately and hence leads to a more modular proof. For the following paragraphs and the intuitive understanding of the solution, the reader should keep in mind that the intuitive presentation of the criterion is summarized in figure 2.4.

2.3.2 Bounding the Construction

The systems that we are a looking at are real-time uniprocessor or multiprocessor systems. In these systems tasks come to the respective scheduler with a number of parameters that allow these schedulers to decide whether these tasks are schedulable.

We assume that we have n tasks in the system, indexed $t_1 ... t_n$. The tasks can be either periodic or sporadic. For each task t_i we will use the standard notations T_i, C_i, R_i and D_i to denote the period (i.e. min period for sporadic tasks), worst case execution time, worst case response time and deadline, respectively. The deadline of a task is less or equal to its period.

For a system to be safe, no task should miss its deadlines, i.e. $\forall i \mid R_i \leq D_i$.

For a system scheduled with fixed priority, the response time for a task in the initial system can be calculated using the standard response time analysis techniques [17]. If we with B_i denote the blocking time (the time the task can be delayed by lower priority tasks) and with $hp(i)$ denote the set of tasks with higher priority than task t_i, the response time R_i for task t_i can be formulated as:

$$R_i = C_i + B_i + \sum_{j \in hp(i)} \left\lceil \frac{R_i}{T_j} \right\rceil C_j \tag{2.1}$$

The summand in the above formula gives the time that task t_i may be delayed by higher priority tasks. For systems scheduled with dynamic priorities, there are other ways to calculate the response times [17].

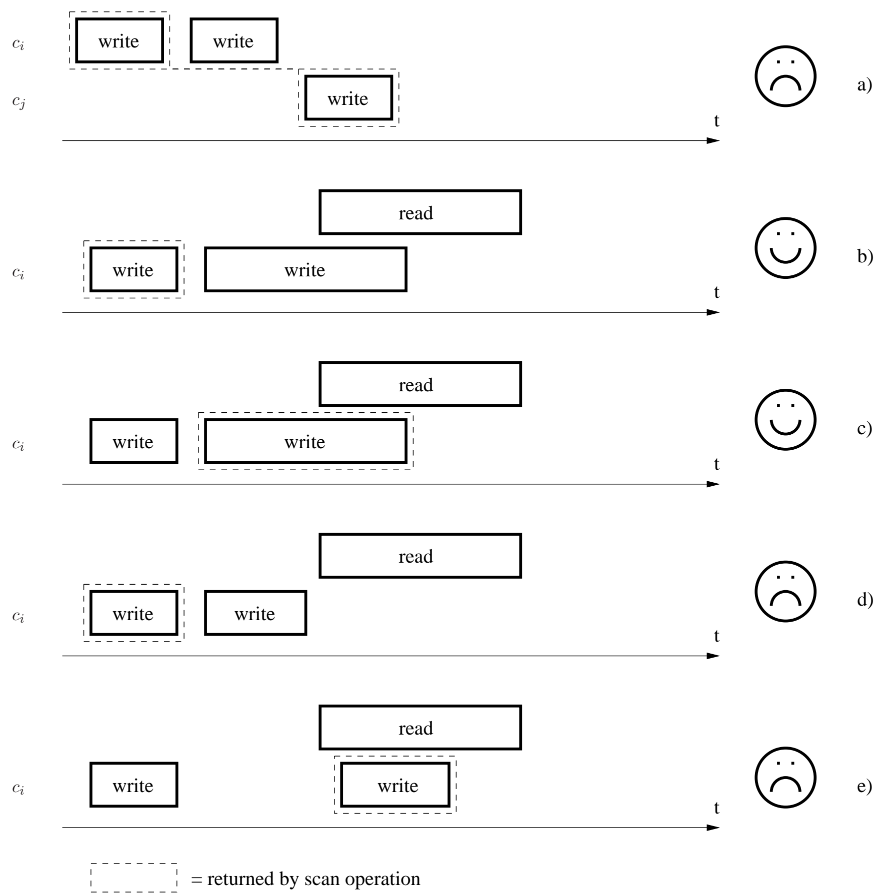


Figure 2.4: Intuitive presentation of the atomicity/linearizability criterion satisfied by our wait-free solution

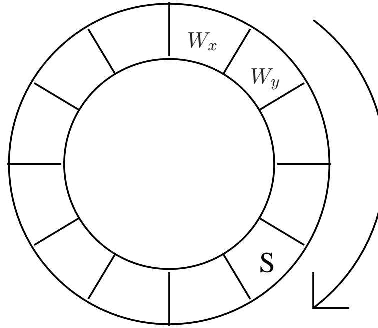


Figure 2.5: A cyclic buffer with several updater tasks and one snapshot task

We will use T_S to denote the snapshot task period and T_{W_i} to denote the updater tasks period. To simplify the formulas we assume that tasks can be preempted at arbitrary points during their execution and that there are no overheads for context switching or interrupt handling. We also assume that one of the tasks in the system acts as a scanner task, say t_{scan}, but in the original system it doesn't have any mechanism to get a consistent snapshot.

In this subsection, we will show how to transform the unbounded space protocol of the previous subsection into one that uses bounded space only.

In the bounded space protocol as well, we are going to keep the role of the scanner as the controller of the game. It still is the one who determines the subregister where the updater is going to write. However, because the number of the subregisters must be bounded, instead of forwarding a new subregister each time, the scanner has to find an obsolete subregister which will be forwarded to the updater after erasing its contents. We call this procedure of erasing the contents of a subregister and its forwarding to the updater, as *recycling* of the subregister.

We keep the techniques used in the previous algorithm, that is: (i) The updaters write to the memory location forwarded by the snapshot operation. (ii) The snapshot operation, by forwarding with its very first sub-operation a recycled subregister, which it is not going to use again during the current snapshot, it succeeds to avoid reading component values written by update operations which start after its own starting point. (iii) The scanner in each snapshot operation reads the remaining memory locations in the reverse order from the one that they had been previously forwarded.

Thus, the problem of designing a correct algorithm that uses a bounded number of subregisters is reduced to the problem of having the scanner

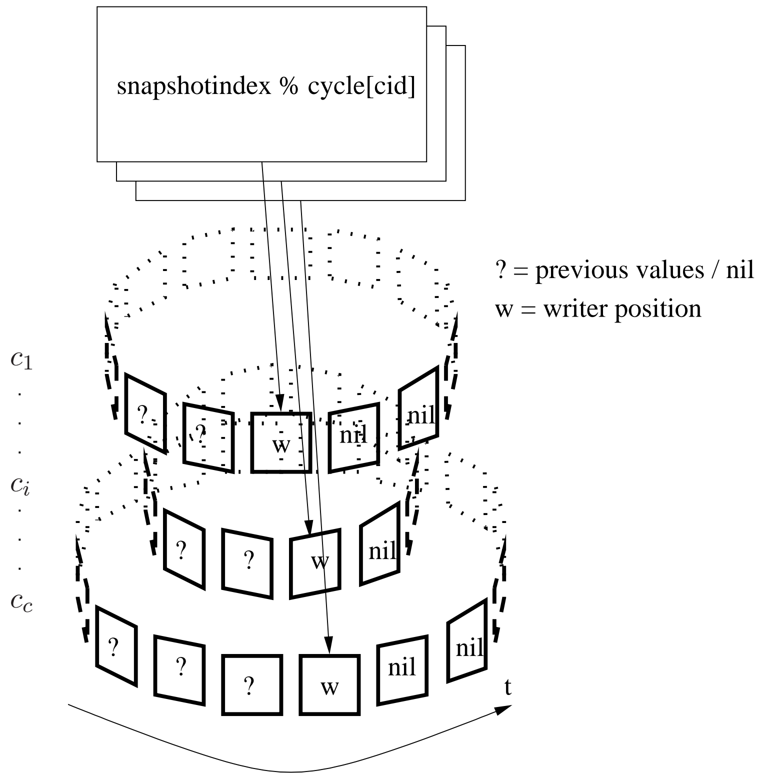


Figure 2.6: The bounded Snapshot protocol

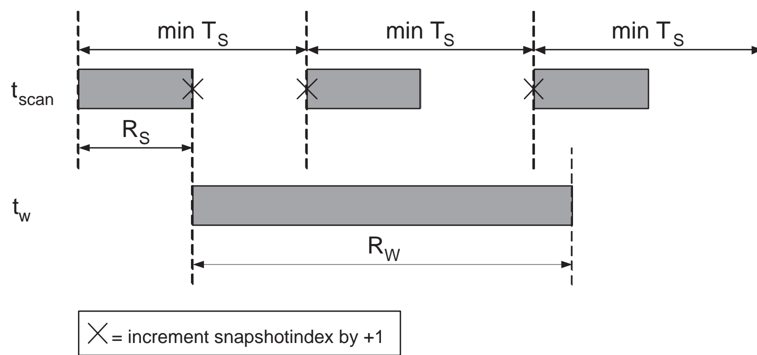


Figure 2.7: Estimating the buffer length - worst case scenario

```
// Global variables
snapshotindex:integer
value[NR_COMPONENTS]:pointer to valtype
cycle[NR_COMPONENTS]:integer
// Local variables
index,k,tempindex:integer

procedure Initialize
I1    for index:=0 to NR_COMPONENTS-1 do
I2        value[index]:= new valtype[cycle[index]];

procedure Update(cid:integer, data:valtype)
U1    tempindex:=snapshotindex modulo cycle[cid];
U2    value[cid][tempindex]:=data;

procedure Scan(snapshotdata[NR_COMPONENTS]:valtype)
S1    tempindex:=snapshotindex+1;
      /* clean phase */
S2    for k:=0 to NR_COMPONENTS-1 do
S3        value[k][tempindex modulo cycle[k]] := NIL;
S4    snapshotindex:=tempindex;
S5    tempindex:=tempindex-1;
      /* read phase */
S6    for k:=0 to NR_COMPONENTS-1 do
S7        index:=tempindex modulo cycle[k];
S8        while index ≠ (tempindex+1) modulo cycle[k] do
S9          if value[k][index] ≠ NIL then
S10           snapshotdata[k] := value[k][index];
S11           break;
S12         index:=(index-1) modulo cycle[k];
```

Figure 2.8: Pseudo-code for the Bounded Snapshot Algorithm

choose each time a *provably* obsolete subregister for recycling. By doing this we can use the timing information that comes together with the task set in real-time systems. For the beginning please note that the unbounded construction that was presented in the previous section has the nice property that the scanner task is always at least one position ahead of the updaters when accessing the buffers, see figure 2.5. This leads us to consider replacing the unbounded buffer with a cyclical buffer mechanism where the buffer slots are now going to be "forwarded" cyclically by the scanner. Each circular data buffer now is implemented by an array of l entries. Each entry is capable of holding one copy of the data that an updater wants to write. The next step then is to analyze the conditions that the cyclical buffers have to satisfy in order to maintain the safety properties that were described above. Note that the buffer length can be of different length for each individual component, and that the buffer length is dependent on the timing characteristics of the updaters that write to this component, and also dependent on the timing characteristics of the scanner task which advances the buffer index. See figure 2.8 for the algorithm pseudo-code and figure 2.6 for an explanation how the algorithm interacts with the cyclic buffers.

As the updater always has to have a valid buffer slot to write to, we know that we need a buffer of at least length one. So to calculate how many more indexes we need for each buffer we compare the maximum time it takes for an updater to finish, to the minimum time it takes between two subsequent increments of the index done by the scanner function. First we assume that the tasks always execute within their response times R with arbitrary many interruptions, and that the execution time C is comparably small. This means that the increment respective the write to the buffer slot can occur anytime within the interval for the response time. The maximum time for an updater function to finish is the same as the response time R_W for its task t_W. The minimum time between two index increments is when the first increment is executed at the end of the first interval and the next increment is executed at the very beginning of the second interval, i.e. $T_S - R_S$. The minimum time between the subsequent increments will then be the period (min for sporadic tasks) T_S. The worst case scenario between an updater and the scanner is when the snapshotindex is incremented directly after the updater has read it, as it is shown in Figure 2.7. Regardless of the timing characteristics of the involved tasks, it will always be possible to have at least one increment, and adding the one always needed by the updater this adds up to an absolute minimum of two buffer slots.

If $R_W \leq (\min T_S - R_S)$ then:

Scenario	Scan Period (us)	Update Period (us)	Buffer Length
1	500	50	3
2	200	50	3
3	100	50	3
4	50	50	4
5	50	100	6
6	50	200	10
7	50	500	22

Figure 2.9: Descriptions of Scenarios for experiment

$$l = 2 \tag{2.2}$$

If $R_W > (\min T_S - R_S)$ then:

$$l = \left\lfloor \frac{R_W - (\min T_S - R_S)}{\min T_S} \right\rfloor + 3 \tag{2.3}$$

We are now combining those two expressions into one single expression. The floor function can safely turned into a ceiling function by subtracting one from the constant in equation 2.3. We also have to consider the longest buffer length we need considering all the updater tasks that are updating the same component. If we denote the length of the buffer for component k with l_k, and the group of updaters to component k with $wr(k)$, our calculations lead us to the following formula:

$$l_k = \left\lceil \frac{\max_{i \in wr(k)} R_{W_i} - \min T_S + R_S}{\min T_S} \right\rceil + 2 \tag{2.4}$$

The last formula that we have calculated describes the buffer length that our construction needs in order to guarantee the safety property of our circular-buffer. It can be clearly seen that the buffer lengths keep very low when the snapshot task period is bigger than the updater task period, actually very similar buffer lengths as what can be achieved with more sophisticated snapshot algorithms, like the wait-free [4, 29] that does not use the timing information but instead uses more advanced synchronization primitives that the tasks can use in order to synchronize.

2.4 Experiments

A number of experiments have been performed in order to measure experimentally the performance of the new construction. To give an interesting

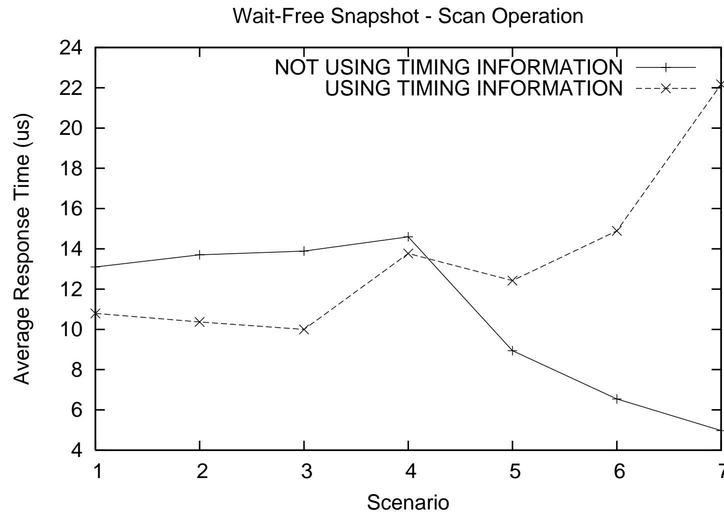


Figure 2.10: Experiment with 1 Scan and 10 Update processes - Scan task comparison

and appropriate comparison we have done experiments with the wait-free snapshot algorithm [4, 29], and then done the similar experiments with the bounded-time algorithm presented in this paper. The experiments have been executed on a Sun Enterprise 10000 parallel machine with 64 processors. The machine is based on the Sun Starfire [24] uniform memory access (UMA) architecture. The system considered is consisting of 1 scan process and 10 updater processes. The tasks have been generated as periodic tasks, with one task per CPU. The periods of the scan and update tasks have been changed according to some selected scenarios, see figure 2.9. Several long executions of the scenarios have been executed and the average response times for the scan and update operations have been measured. The wait-free respective the bounded-time algorithms have been executed with exactly the same environment and parameters. The buffer lengths have been computed according to equation 2.4 presented in the analysis. To give an interesting and appropriate comparison we have compared the algorithm presented here with the wait-free snapshot algorithm presented in [4, 29]. Both these algorithms use the same unbounded memory construction, [4, 29] bounds it efficiently without using the timing information.

The result of the experiments can be viewed in figures 2.10 and 2.11. According to the experiments, the new construction gives 400 % better re-

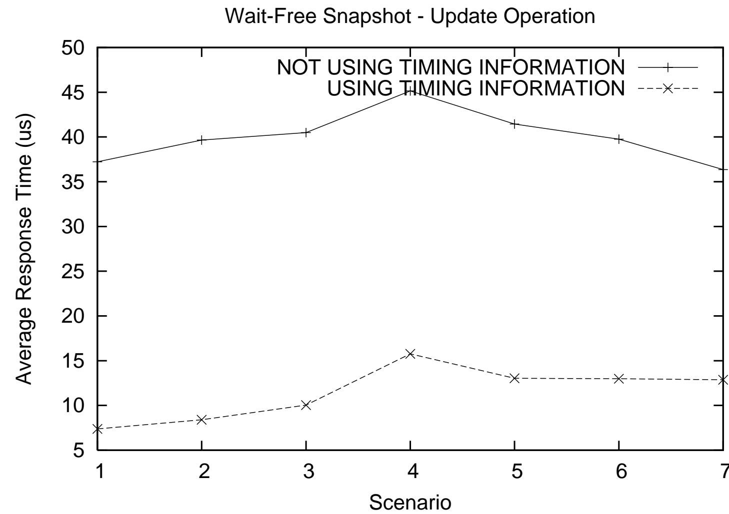


Figure 2.11: Experiment with 1 Scan and 10 Update processes - Update task comparison

sponse time for the update operations for all scenarios and with 20 % better response time for all scenarios that are common in practical settings compared to [4, 29]. The protocol presented in [4, 29] can perform better than the algorithm presented here, but only with respect to the scan operations, and only when the scan period is lower than the update period. The reason for this is that for the construction presented here, the buffer lengths increase as the period of a scan operation increases. But as we mentioned above the new construction gives 400 % better response times for the update operations for all scenarios. These are very significant results as we usually do a lot more update operations than scan operations. Although the scan operation can be slower for the bounded-time for some scenarios, we can assume that we will get a trade-off because of the benefits with the faster update operations.

2.5 Conclusions and Future Work

We have looked at the problem of taking a snapshot of several shared data components in a concurrent system by using timing information about the system that is available on real-time systems. By exploiting this information we design a simple snapshot algorithm with one scanner that is efficient in

time and space. The efficiency of the algorithm was experimentally evaluated on a SUN Enterprise 10000 multiprocessor.

Chapter 3

Space Efficient Wait-Free Buffer Sharing in Multiprocessor Real-Time Systems Based on Timing Information[1]

Håkan Sundell, Philippas Tsigas
Department of Computing Science
Chalmers Univ. of Technol. and Göteborg Univ.
412 96 Göteborg, Sweden
E-mail: {phs, tsigas}@cs.chalmers.se

Abstract

A space efficient wait-free algorithm for implementing a shared buffer for real-time multi-processor systems is presented in this paper. The commonly used method to implement shared buffers in real-time systems is based on mutual exclusion. Mutual exclusion is penalized by blocking that typically leads to difficulties in guaranteeing deadlines in real-time systems. Researchers have introduced non-blocking algorithms and data structures that address

[1]This is an revised and extended version of the paper presented at RTCSA 2000 [103].

the above problems. Many of the non-blocking algorithms that appeared in the literature have very high space demands though, some even unbounded, which makes them not suitable for real-time systems. In this paper we look at a simple, elegant and easy to implement algorithm that implements a shared buffer but uses unbounded time-stamps and we show how to bound the time-stamps by using the timing information that is available in many real-time systems. Our analysis and calculations show that the algorithm resulting from our approach is space efficient. The protocol presented here can support an arbitrary number of concurrent read and write operations.

3.1 Introduction

In real-time systems and in distributed systems in general we have several concurrent tasks that need to communicate and synchronize in order to be able to fulfill the responsibilities of the system. There are several different means to accomplish this in a multi-processor system. In general the tasks communicate using shared data objects. The shared data objects can be centralized or distributed, and can be accessed uniform or non-uniform. The major requirement from manufacturers when designing these shared data objects is to keep space constraints on random access memory as low as possible; in 1998, 1.5 billion 8-bit micro-controllers were sold, compared to only 63.7 million 32-bit micro-controllers [26].

In order to ensure consistency on shared data objects one commonly used method is mutual exclusion. Mutual exclusion protects the consistency of the shared data by allowing only one process at time to access the data. Mutual exclusion i) causes large performance degradation especially in multi-processor systems [100]; ii) leads to complex scheduling analysis since tasks can be delayed because they were either preempted by other more urgent tasks, or because they are blocked before a critical section by another process that can in turn be preempted by another more urgent task and so on (this is also called as the convoy effect) [64]; and iii) leads to priority inversion in which a high priority task can be blocked for an unbounded time by a lower priority task [95]. Several synchronization protocols have been introduced to solve the priority inversion problem for uni-processor [95] and multi-processor [93] systems. The solution presented in [95] solves the problem for the uni-processor case with the cost of limiting the schedulability of task sets and also making the scheduling analysis system, where a task may be blocked by another task running on a different processor [93].

To address the problems that arise from blocking, researchers have pro-

posed non-blocking implementations of shared data structures. Two basic non-blocking methods have been proposed in the literature, *lock-free* and *wait-free*. *Lock-free* implementations of shared data structures guarantee that at any point in time in any possible execution some operation will complete in a finite number of steps. In cases with overlapping accesses, some of them might have to repeat the operation in order to correctly complete it. This implies that there might be cases in which the timing may cause some process(es) to have to retry a potentially unbounded number of times, leading to a for hard real-time systems unacceptable worst-case behavior, although usually they perform well in practice. In *wait-free* implementations each task is guaranteed to *correctly* complete any operation in a *bounded* number of its own steps, regardless of overlaps and the execution speed of other processes; i.e. while the lock-free approach might allow (under very bad timing) individual processes to starve, wait-freedom strengthens the lock-free condition to ensure individual progress for every task in the system.

Non-blocking implementation of shared data objects is a new alternative approach for the problem of inter-task communication. Non-blocking mechanisms allow multiple tasks to access a shared object at the same time, but without enforcing mutual exclusion to accomplish this. Non-blocking inter-task communication does not allow one task to block another task, and gives significant advantages over lock-based schemes because:

1. it cannot cause priority inversion and avoids lock convoys that make scheduling analysis hard and delays longer.
2. it provides high fault tolerance (processor failures will never corrupt shared data objects) and eliminates deadlock scenarios from two or more tasks both waiting for locks held by the other.
3. and more significantly it completely eliminates the interference between process scheduling and synchronization.

Non-blocking protocols on the other hand have to use more delicate strategies to guarantee data consistency than the simple enforcement of mutual exclusion between the different operations on the data object. These new strategies on the other hand, in order to be useful for real-time systems, should be efficient in time and space in order to perform under the tight space and time constraints that real-time systems demand.

Some of the wait-free protocols presented in the literature have very high space demands, some even require unbounded space, which makes them of

no practical interest for real-time systems. In real-time systems usually tasks come together with their timing characteristics, such as their worst-case execution time, their period and etceteras. In this paper we look at a simple, elegant and easy to implement algorithm that implements a shared buffer but uses unbounded time-stamps and we show how to bound the time-stamps by using the timing information that is available in a real-time system. The solution allows any arbitrary number of readers and writers to concurrently access the buffer. The resulting algorithm as we show has low memory demands.

Previously Chen and Burns in [25], exploited the use of the timing information for the construction of a non-blocking shared buffer; they were the first to show how to use timing-based information to implement a fully asynchronous reader/writer mechanism; in their work they considered the case where there is only one writer. The algorithm presented in this paper allows arbitrary number of readers and writers to perform their respective operations. Research at the University of North Carolina [9, 10] and [94] by Anderson et al. has shown that wait-free algorithms can be simplified considerably in real-time systems by exploiting the way that processes are scheduled for execution in such systems. Research work investigating the relation between non-blocking synchronization and real-time systems dates back to 1974 [101, 102]. Massalin and Pu [75] and Greenwald and Cheriton [39] were the first to develop lock-free real-time kernels. Last but not least the real-time specifications of JAVA [21] include wait-free synchronization.

The rest of this paper is organized as follows. In Section 2 we describe basic characteristics of the computer systems that we are considering together with the formal requirements that any solution to the synchronization problem that we are considering must guarantee. In Section 3 we show how to use the timing information in order to bound the time-stamps of the unbounded protocol that is also presented in this section. Section 4 presents some examples showing the effectiveness of our results. The paper concludes with Section 5.

3.2 System and Problem Description

3.2.1 Real-Time Multiprocessor System Configuration

A typical abstraction of a shared memory multi-processor real-time system configuration is depicted in figure 3.1. Each node of the system contains a processor together with its local memory. All nodes are connected to the

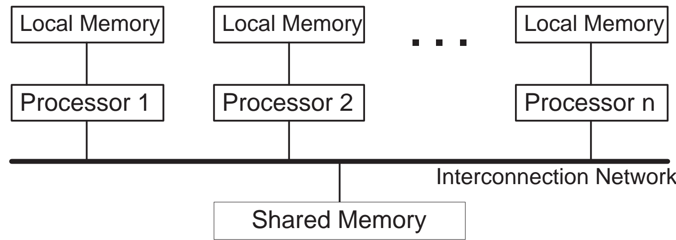


Figure 3.1: Shared Memory Multiprocessor System Structure

shared memory via an interconnection network. A set of co-operating tasks[2] (processes) with timing constraints are running on the system performing their respective operations. Each task is sequentially executed on one of the processors, while each processor can serve (run) many tasks at a time. The co-operating tasks, possibly running on different processes, use shared data objects built in the shared memory to co-ordinate and communicate. Every task has a maximum computing time and has to be completed by a time specified by a deadline. Tasks synchronize their operations through read/write operations to shared memory. The shared memory may not though be uniformly accessible for all nodes in the system. Some processors can have slower access or other restrictions like no access at all to some part of the shared memory.

3.2.2 The Model

In this paper we are interested in the problem of constructing an atomic shared buffer.

The accessing of the shared object is modeled by a history h. A history h is a finite (or not) sequence of operation invocation and response events. Any response event is preceded by the corresponding invocation event. For our case there are two different operations that can be invoked, a read operation or a write operation. An operation is called complete if there is a response event in the same history h; otherwise, it is said to be pending. A history is called complete if all its operations are complete. In a global time model each operation q "occupies" a time interval $[s_q, f_q]$ on one linear time axis ($s_q < f_q$); we can think of s_q and f_q as the starting and finishing time instants of q. During this time interval the operation is said to be *pending*. There exists a precedence relation on operations in history denoted

[2]throughout the paper the terms *process* and *tasks* are used interchangeably

by $<_h$, which is a strict partial order: $q_1 <_h q_2$ means that q_1 ends before q_2 starts; Operations incomparable under $<_h$ are called *overlapping*. A complete history h is linearizable if the partial order $<_h$ on its operations can be extended to a total order $\rightarrow_{\mathrm{h}}$that respects the specification of the object [45]. For our object this means that each read operation should return the value written by the write operation that directly precedes the read operation by this total order ($\rightarrow_{\mathrm{h}}$).

To sum it up, as we are looking for a non-blocking solution to the general reader/writer problem for real-time systems we are looking for a solution that satisfies:

- Every read operation guarantees the integrity and coherence of the data it returns.
- The behavior of each read and write operation is predictable and can be calculated for use in the scheduling analysis.
- Every possible history of our protocol should be linearizable [45].

3.3 The Protocol

3.3.1 The Unbounded Algorithm

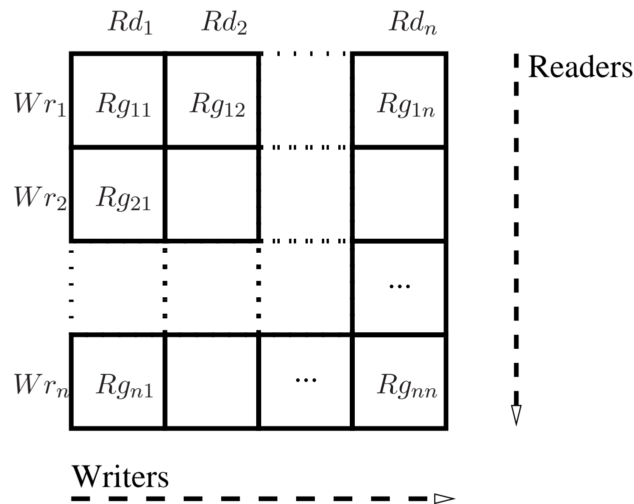


Figure 3.2: Architecture of the Algorithm

```
THE UNBOUNDED ALGORITHM FOR N-READER
N-WRITER SHARED REGISTER

Reader (on processor i):
    tag_max := 0
    for j := 1 to n
        if tag(Rg_ji) > tag_max then
            tag_max := tag(Rg_ji)
            value := value(Rg_ji)
    for j := 1 to n
        Rg_ij := (value,tag_max)
    return value

Writer (on processor i):
    tag_max := 0
    for j := 1 to n
        if tag(Rg_ji) > tag_max then
            tag_max := tag(Rg_ji)
    for j := 1 to n
        Rg_ij := (value,tag_max + 1)
```

Figure 3.3: The unbounded algorithm

We first start with a simple, elegant and easy to implement unbounded protocol that appeared in [124]. The algorithm uses a matrix of 1-reader 1-writer registers, see figure 3.2. We denote the reader task and the writer task running on processor i with Rd_i and Wr_i respectively. The matrix is formed in a way such that each register Rg_{ij} can be read by processor j and written by processor i.

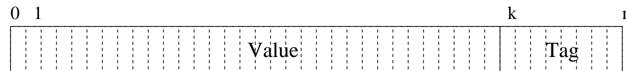


Figure 3.4: The Register Structure

The algorithm originally uses unbounded time-stamps and consequently

the data read from or written into each of the registers contains a data pair of a value and a tag, see figure 3.4. Each of the registers are read from or written to in one atomic operation. In the algorithm the tag value is unbounded. The pseudo-code for the algorithm can be viewed in figure 3.3. The tag value indicates the freshness of the value; a higher tag means a newer value. In this algorithm the reader reads in columns and the writers writes in rows as seen in figure 3.2. When the reader wants to get the latest value from the shared register it reads all the registers in its column and takes the value with the highest tag. Then it writes this value together with the tag to all registers in its row. When the writer wants to write a new value it first looks for the highest possible tag in the matrix by reading a column. Then it writes the new value in its row together with that tag value incremented by one.

The value for the tags during the execution increases rapidly and thus many bits have to be allocated for the tag value on the register to ensure there is no overflow. As the register contains a limited number of bits, this means that we have fairly few bits to allocate for the actual value. In real-time systems random access memory is also a limited resource and registers usually contain few bits. Further the system is usually required be able to run continuously for a very long period, which means that we have to allocate a lot of bits for the tag field, to ensure there is no overflow.

Let us now consider a system with eight processors and eight writer tasks, one task on each CPU. Assume that the period of each task is 10 ms, and that the tasks are interleaved in time as it is shown in figure 3.5. Each task starts its execution after the previous task has started to write the incremented tag to one of the registers, but not necessarily all. In this way this register will be scanned by the next writer tasks when they are scanning for the highest tag value in the system. The last writer finishes its execution before the first writer restarts its execution and the procedure repeats itself over and over. In this scenario, each invocation of a writer task will increase the tag by one, and in each period we will have increased the tag by eight. This means that in just a second of the systems execution we will have a tag value of 800, that requires 10 bits. For an hour of execution, we will have a tag value of 2 880 000, occupying 22 bits. This clearly indicates that we cannot use the algorithm as is in a real-time system with limited memory capabilities, and therefore it is of great importance to be able to bound the size of the tag field. In the next subsection we show how to use the timing information that is available in real-time systems to efficiently bound the size of the time-stamps.

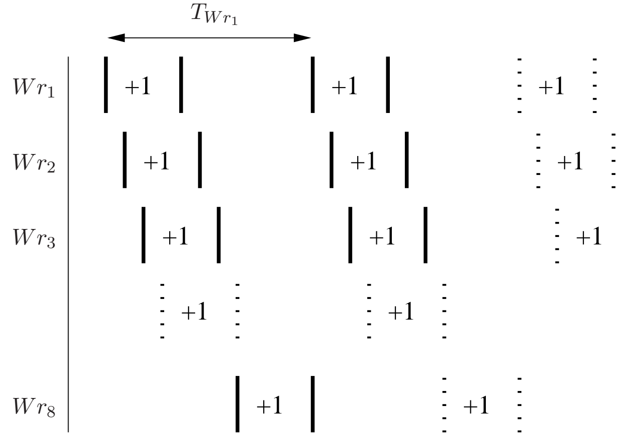


Figure 3.5: Rapidly Increasing Tags

3.3.2 Bounding the Time-Stamps

In this part of the paper we will see how to recycle the tags. The way that the algorithm is going to work is similar to the way the algorithm that uses unbounded time-stamps works. Namely, the writer produces a new time-stamp every time it writes, and the reader returns the value of the most recent time-stamp. The idea is to maintain a bounded number of time-stamps, and keep track of the ordering in which they were issued.

We are assuming that all tasks are periodic and that all tasks will meet their deadlines which are shorter than their respective periods. In real-time systems it is very often the case that we have very good information about the tasks.

We assume that we have n tasks in the system, indexed $t_1 \dots t_n$. For each task t_i we will use the standard notations T_i, C_i, R_i, D_i and B_i to denote the period, worst case execution time, worst case response time, deadline and blocking time (the time the task can be delayed by lower priority tasks), respectively. Also $hp(i)$ denotes the set of tasks with higher priority than task t_i. The deadline of a task is less or equal to its period.

For a system to be safe, no task should miss its deadlines, i.e. $\forall i \mid R_i \leq D_i$. The response time R_i for a task in the initial system can be calculated using the standard response time analysis techniques [17] as:

$$R_i = C_i + B_i + \sum_{j \in hp(i)} \left\lceil \frac{R_i}{T_j} \right\rceil C_j \qquad (3.1)$$

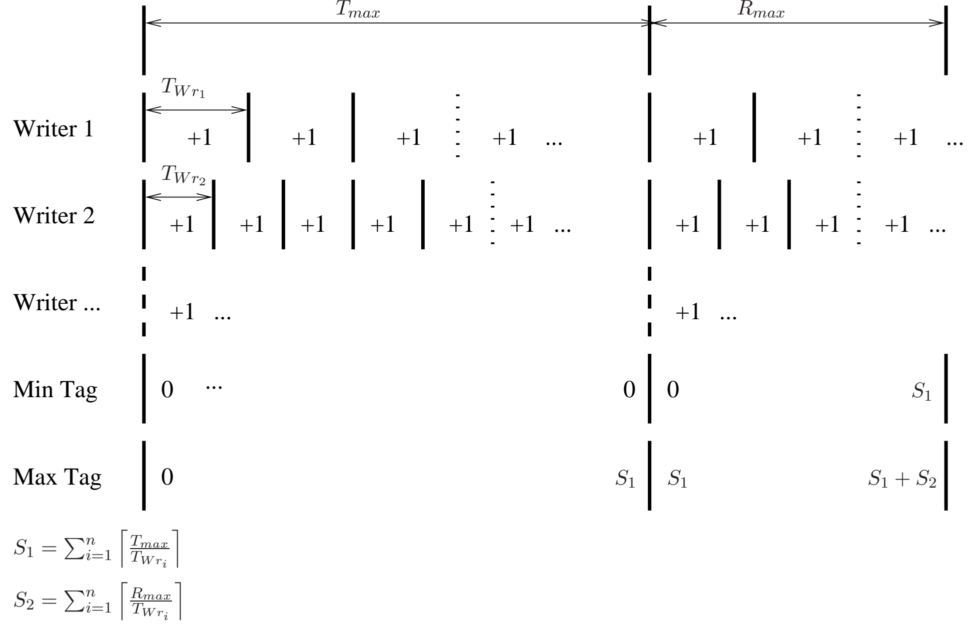


Figure 3.6: Tag Range

In the next part of this section we are going to use the following notation, where Rd_i and Wr_i denotes the reader respective the writer tasks:

$$T_{Wr_{max}} = \max_{i \in \{1..n\}} T_{Wr_i}$$
$$T_{Rd_{max}} = \max_{i \in \{1..n\}} T_{Rd_i}$$
$$T_{max} = \max\{T_{Wr_{max}}, T_{Rd_{max}}\}$$

$$R_{Wr_{max}} = \max_{i \in \{1..n\}} R_{Wr_i}$$
$$R_{Rd_{max}} = \max_{i \in \{1..n\}} R_{Rd_i}$$
$$R_{max} = \max\{R_{Wr_{max}}, R_{Rd_{max}}\}$$

In order to bound the time-stamps we will first try to find an upper bound for $t_2 - t_1$, where t_1 is and t_2 are respectively the time stamps that a task can observe in two consecutive invocations in any possible execution of the unbounded algorithm.

Let us now take an arbitrary execution ϵ, for simplicity we assume that the tags have been initialized to 0. We are considering the worst possible

scenario. Like in the previous section we assume that each writer task will increase the highest tag value by one. We are assuming for the worst case that the task with the longest period T_{max} is scanning the matrix for the highest tag in the beginning of its execution, before any other task has written any other value. This task is then suspended for almost all of its period and writes the highest tag value in its row at the very end of the task period. Thus the lowest tag will still be zero and the upper boundary for the highest tag will be:

$$S_1 = \sum_{i=1}^{n} \left\lceil \frac{T_{max}}{T_{Wr_i}} \right\rceil$$

At the start of the execution of the task with the longest response time R_{max}, this task starts to scan for the highest value and will therefore get S_1 as the result. We are now assuming that this task is writing this tag in its row at the very end of its response time. And thus the lowest tag in the system will now be S_1. During the execution of this task, the highest tag may have been increased at most by:

$$S_2 = \sum_{i=1}^{n} \left\lceil \frac{R_{max}}{T_{Wr_i}} \right\rceil$$

Therefore the highest possible tag in the system at the end of this execution of this task is:

$$MaxTag = \sum_{i=1}^{n} \left\lceil \frac{T_{max}}{T_{Wr_i}} \right\rceil + \sum_{i=1}^{n} \left\lceil \frac{R_{max}}{T_{Wr_i}} \right\rceil$$

This means that during the execution that spanned from time 0 to time $T_{max} + R_{max}$ the tag values will span between zero and $S_1 + S_2$, which leads to the following lemma:

Lemma 1 *In any possible execution the time-stamps that two consecutive tasks can observe are going to be* $MaxTag = \sum_{i=1}^{n} \left\lceil \frac{T_{max}}{T_{Wr_i}} \right\rceil + \sum_{i=1}^{n} \left\lceil \frac{R_{max}}{T_{Wr_i}} \right\rceil$ *far apart.*

The above lemma gives us a bound on the length of the "window" of active time-stamps for any task in any possible execution. In the unbounded construction the writers, by producing larger time-stamps every time they slide this window on the $[0, \ldots, \infty]$ axis, always to the right. The approach now is instead of sliding this window on the set $[0, \ldots, \infty]$ from left to right,

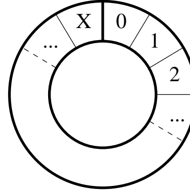


Figure 3.7: Tag Value Recycling

to cyclically slide it on a $[0, \ldots, X]$ set of consecutive natural numbers, see figure 3.7. Now at the same time we have to give a way to the tasks to identify the order of the different time stamps because the order of the physical numbers is not enough since we are re-using time-stamps. The idea is to use the bound that we have calculated for the span of different active time-stamps. Let us then take a task that has observed t_1 as the lowest time-stamp at some invocation τ. When this task runs again as τ', it can conclude that the active time-stamps are going to be between t_1 and $(t_1 + MaxTag) \mod X$. On the other hand we should make sure that in this interval $[t_1, \ldots, (t_1 + MaxTag) \mod X]$ there are no old time-stamps. By looking closer to the previous lemma we can conclude that all the other tasks have written values to their registers with time stamps that are at most $MaxTag$ less than t_1 at the time that τ wrote the value t_1. Consequently if we use an interval that has double the size of $MaxTag$, τ' can conclude that old time-stamps are all on the interval $[(t_1 - MaxTag) \mod X, \ldots, t_1]$.

Therefore we can use a tag field with double the size of the maximum possible value of the tag.

$$TagFieldSize = MaxTag * 2$$
$$TagFieldBits = \lceil \log_2 TagFieldSize \rceil$$

In this way τ' will be able to identify that v_1, v_2, v_3, v_4 (see figure 3.8) are all new values if $d_2 + d_3 < MaxTag$ and can also conclude that:

$$v_3 < v_4 < v_1 < v_2$$

The new mechanism that will generate new tags in a cyclical order and also compare tags in order to guarantee the linearizability is presented in figure 3.9.

The proof for the linearizability of this construction is the same as the

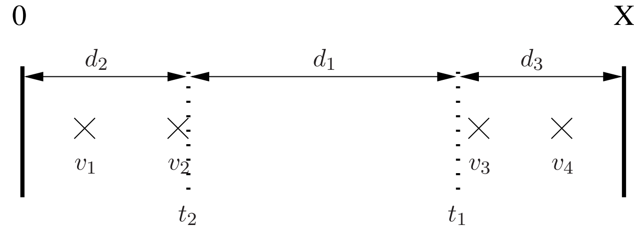


Figure 3.8: Tag Reuse

Task	Period	Task	Period
Wr_1	1000	Rd_1	500
Wr_2	900	Rd_2	450
Wr_3	800	Rd_3	400
Wr_4	700	Rd_4	350
Wr_5	600	Rd_5	300
Wr_6	500	Rd_6	250
Wr_7	400	Rd_7	200
Wr_8	300	Rd_8	150

Table 3.1: Example task scenario on eight processors

linearizability proof of the unbounded one.

Theorem 1 *The algorithm presented in this section implements a bounded multi-reader, multi-writer buffer that uses bounded memory space.*

3.4 Implementation

The whole idea of this algorithm is to provide some kind of shared memory when there is none directly available by the hardware. In this chapter we are describing how to actually implement a shared register on a distributed real-time system using the protocol described in this paper.

We assume that we have a set of nodes, each containing a microcontroller consisting of a CPU, local memory and I/O capabilities. Each node has some way of communicating with the other, i.e. message passing. A message can be sent from any of the nodes to any other of the nodes.

We run the original algorithm on each of the nodes, independently. What

Comparison algorithm for bounded tag size

tag_{max} := tag(Rg_{j1})
for j := 1 to n
    tag := tag(Rg_{ji})
    if (tag > tag$_{max}$ and (tag - tag$_{max}$) $\leq MaxTag$)
        or (tag < tag$_{max}$
        and (tag + $TagFieldSize$ - tag$_{max}$) $\leq MaxTag$)
    then
        tag$_{max}$:= tag

New tag generation for bounded tag size

    for j := 1 to n
        Rg_{ij} := (value,
            ($tag_{max} + 1$) modulo $TagFieldSize$)

Figure 3.9: Algorithm changes for bounded tag size

remains for the complete implementation is how to actually read from or write to the individual single registers.

The original algorithm makes use of a matrix of 1-reader 1-writer registers, see figure 3.2. These single registers have to be distributed among the different nodes and reside on real local memory. As each register R_{ij} can be written to by processor i and read from by processor j, we can put all the registers R_{ij}, $1 \leq i \leq n$ on the local memory of processor j. These registers actually form a column of the original matrix, see figure 3.10.

The implementation of a read from the matrix R_{ij} is then straightforward as seen in figure 3.11 ; the local read function just returns the local memory contents of R_{ij}. When the contents of one of the registers are to be changed, a message is prepared from the processor that can write to the specific register. The message is then sent to the processor that can read from it. The receiving node j that got a message from node i then updates the corresponding register R_{ij}.

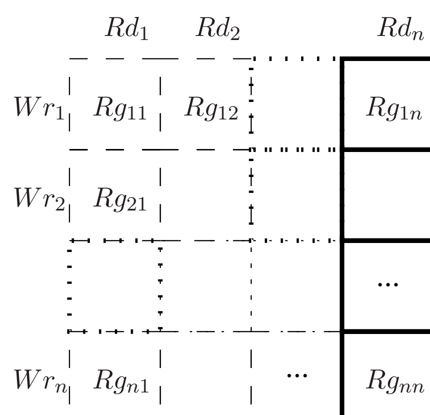


Figure 3.10: The registers located on each processor as a column of the matrix

3.5 Examples

In order to show the effectiveness of our analysis, consider the scenario described in table 3.1. The tasks are running on 8 processors, where writer Wr_i and reader Rd_i are executing on the same processor. We are also assuming that the reader and writer on the same processor are executing atomically with respect to each other. All deadlines are considered to be met.

If we assume that the maximum response time is equal to the maximal task period, and then apply the formulas from the analysis we get:

$$T_{max} = R_{max} = 1000$$

$$MaxTag = \sum_{i=1}^{n} \left\lceil \frac{T_{max}}{T_{Wr_i}} \right\rceil + \sum_{i=1}^{n} \left\lceil \frac{R_{max}}{T_{Wr_i}} \right\rceil = \sum_{i=1}^{8} \left(\left\lceil \frac{1000}{T_{Wr_i}} \right\rceil + \left\lceil \frac{1000}{T_{Wr_i}} \right\rceil \right) = 1 + 1 + 2 + 2 + 2 + 2 + 2 + 2 + 2 + 2 + 3 + 3 + 3 + 3 + 4 + 4 = 38$$

$$TagFieldSize = 38 * 2 = 76$$

$$TagFieldBits = \lceil \log_2 76 \rceil = 7$$

A tag field size of 7 bits is relatively low considering that we are looking at a scenario with different task periods. Using 16-bit registers for imple-

```
CODE TO BE IMPLEMENTED ON EACH PROCESSOR i:

// Local variables
localR[NUMBER_OF_PROCESSORS]: integer

structure RWMessage
    from: integer
    value: integer

function ReadR(i:integer, j:integer): integer
    return localR[i];

procedure WriteR(i:integer, j:integer, vt: integer)
    m.from:=i;
    m.value:=vt;
    Send Message m to Processor j

procedure On Received Message(m)
    localR[m.from]:=m.value;
```

Figure 3.11: Code to be implemented on each involved processor.

menting the shared register we can safely use 9 bits for the actual value contents. Without bounding the tag size, we would have reached a maximum tag value of 68400 in only one hour of execution, thus needing more than 16 bits.

If we consider the scenario discussed in subsection 3.1 of this paper, where we had 8 writer tasks, the bounded version that we propose needs only 4 bits.

3.6 Conclusions

We have studied an algorithm for wait-free implementation of an atomic n-reader n-writer shared register. The algorithm uses unbounded time-stamps.

We have shown how to use timing information available on real-time systems to bound the time-stamps. According to our examples the modified algorithm has small space requirements.

Chapter 4

NOBLE: A Non-Blocking Inter-Process Communication Library[1]

Håkan Sundell, Philippas Tsigas
Department of Computing Science
Chalmers Univ. of Technol. and Göteborg Univ.
412 96 Göteborg, Sweden
E-mail: {phs, tsigas}@cs.chalmers.se

Abstract

Many applications on shared memory multi-processor machines can benefit from the exploitation of parallelism offered by non-blocking synchronization. In this paper, we introduce a library called NOBLE, which supports multi-process non-blocking synchronization. NOBLE provides an inter-process communication interface that allows the user to transparently select the synchronization method that is best suited to the current application. The library provides a collection of the most commonly used data types and protocols in a form that allows them to be used by non-experts. We describe the functionality and the implementation of the library functions, and illustrate the library programming style with example programs. We also provide

[1]This is a revised and extended version of the paper presented at LCR 02 [104].

experiments showing that using the library can considerably reduce the run-time of parallel code on shared memory machines.

4.1 Introduction

Software implementations of synchronization constructs are usually included in system libraries. The design of a good synchronization library can be challenging. Many efficient implementations for the basic synchronization constructs (locks, barriers and semaphores) have been proposed in the literature. Many such implementations have been designed with the aim to lower the contention when the system is in a high congestion situation. These implementations give different execution times under different contention instances. But still the time spent by the processes on synchronization can form a substantial part of the program execution time [61, 76, 86, 125]. The reason for this is that typical synchronization is based on blocking that unfortunately results in poor performance. More precisely, blocking produces high levels of contention on the memory and the interconnection network, and more significantly, because it causes convoy effects: if one process holding a lock is preempted, other processes on different processors waiting for the lock will not be able to proceed. Researchers have introduced non-blocking synchronization to address the above problems.

Two basic non-blocking methods have been proposed in the literature, *lock-free* and *wait-free* [45]. *Lock-free* implementations of shared data structures guarantee that at any point in time in any possible execution some operation will complete in a finite number of steps. In cases with overlapping accesses, some of them might have to repeat the operation in order to correctly complete it. This implies that these implementations, though usually performing well in practice, might lead to a worst-case behavior unacceptable for hard real-time systems, where some processes might be forced to retry a potentially unbounded number of times. In *wait-free* implementations each task is guaranteed to *correctly* complete any operation in a *bounded* number of its own steps, regardless of overlaps of the individual steps and the execution speed of other processes; i.e. while the lock-free approach might allow (under very bad timing) individual processes to starve, wait-freedom strengthens the lock-free condition to ensure individual progress for every task in the system.

In the design of inter-process communication mechanisms for parallel and high performance computing, there has been much advocacy arising from the theory community for the use of non-blocking synchronization primitives

rather than the use of blocking ones. This advocacy is intuitive and many researchers have for more than two decades developed efficient non-blocking implementations for several shared data objects. In [115, 118] Tsigas and Zhang show by manually replacing lock-based synchronization code with non-blocking ditto in parallel benchmark applications, that non-blocking performs as well as, and often even better than lock-based synchronization. Despite this advocacy, the scientific discoveries on non-blocking synchronization have not migrated much into practice, even though synchronization is still the major bottleneck in many applications. The lack of a standard library for non-blocking synchronization of shared data objects commonly used in parallel applications has played a significant role for this slow migration. The experience from implementing our previous work on non-blocking algorithms [29, 103, 112, 114, 116] has been a natural start for developing a software library, named NOBLE.

NOBLE offers a library support for non-blocking multi-process synchronization in shared memory systems. NOBLE has been designed in order to: i) provide a collection of shared data objects in a form which allows them to be used by non-experts, ii) offer an orthogonal support for synchronization where the developer can change synchronization implementations with minimal changes, iii) be easy to port to different multi-processor systems, iv) be adaptable for different programming languages and v) contain efficient known implementations of its shared data objects. We will throughout the paper illustrate the features of NOBLE using the C language, although other languages are supported.

The rest of the paper is organized as follows. Section 2 outlines the basic features of NOBLE and the implementation and design steps that have been taken to support these features. Section 3 illustrates by a way of an example the use of the features of NOBLE in a practical setting. Section 4 presents run-time experiments that show the performance benefits that can be achieved by using NOBLE. Finally, Section 5 concludes this paper.

4.2 Design and Features of NOBLE

When designing NOBLE we identified a number of characteristics that had to be covered by our design in order to make NOBLE easy to use for a wide range of practitioners. We designed NOBLE to have the following features:

- Usability-Scope - NOBLE provides a collection of fundamental shared data objects that are widely used in parallel and real-time applications.

- Easy to use - NOBLE provides a simple interface that allows the user to use the non-blocking implementations of the shared data objects in the same way the user would have used lock-based ones.
- Easy to Adapt - No need for changes at the application level are required by NOBLE and different implementations of the same shared data objects are supported via a uniform interface.
- Efficient - Users will have to experience major improvements in order to decide to replace the existing trusted synchronization code with new methods. NOBLE has been designed to be as efficient as possible.
- Portable - NOBLE has been designed to support general shared memory multi-processor architectures and offers the same user interface on different platforms. The library has been designed in layers, so that only changes in a limited part of the code are needed in order to port the library to a different platform.
- Adaptable for different programming languages.

4.2.1 Usability-Scope

NOBLE provides a collection of non-blocking implementations of fundamental shared data objects in a form that allows them to be used by non-experts. This collection includes most of the shared data objects that can be found in a wide range of parallel applications, i.e. stacks, queues, linked lists, snapshots and buffers. NOBLE contains the most efficient realizations known for its shared data objects. See Tables 4.1 and 4.2 for a brief description.

4.2.2 Easy to use

NOBLE provides a precise and readable specification for each shared data object implemented. The user interface of the NOBLE library is the defined library functions that are described in the specification. At the very beginning the user has only to include the NOBLE header at the top of the code (`#include <Noble.h>`).

In order to make sure that none of the functions or structures that we define causes any name-conflict during the compilation or the linking stage, only the functions and the structures that have to be visible to the user of NOBLE are exported. All other names are invisible outside of the library. The names that are exported start with the three-letter combination NBL.

Object	Operations	Implementations	Description
Queue	Enqueue, Dequeue	NBLQueueCreateLF(). Algorithms by Valois [122], Michael and Scott [83]. Back-off strategies added.	Lock-Free. Uses the Compare-And-Swap (CAS) atomic primitive and has its own memory management scheme that uses the CAS and Fetch-And-Add (FAA) atomic primitives.
		NBLQueueCreateLF2(). Algorithms by Tsigas and Zhang [116]	Lock-Free. Based on cyclical arrays and uses the CAS atomic primitive in a sparse manner.
		NBLQueueCreateLB().	Lock-Based. Uses the in NOBLE configured mutual exclusion method (spin-locks) and memory management scheme (malloc).
Stack	Push, Pop	NBLStackCreateLF(). Algorithms by Valois [122], Michael and Scott [83]. Back-off strategies added.	Lock-Free. Uses the CAS atomic primitive and has its own memory management scheme that uses the CAS and FAA atomic primitives.
		NBLStackCreateLB().	Lock-Based. Uses the in NOBLE configured mutual exclusion method (spin-locks) and memory management scheme (malloc).
Singly Linked List	First, Next, Insert, Delete, Read	NBLSLListCreateLF(). Algorithms by Valois [122], Michael and Scott [83]. Back-off strategies added.	Lock-Free. Uses auxiliary nodes and the CAS atomic primitive. Has its own memory management scheme that uses the CAS and FAA atomic primitives.
		NBLSLListCreateLF2(). Algorithms by Harris [44], Valois [122], Michael and Scott [83]. Back-off strategies added.	Lock-Free. Based on using unused bits of pointer variables and the CAS atomic primitive. Uses the same memory management as NBLSLListCreateLF.
		NBLSLListCreateLB().	Lock-Based. Uses the in NOBLE configured mutual exclusion method (spin-locks) and memory management scheme (malloc).

Table 4.1: The shared data objects supported by NOBLE

Object	Operations	Implementations	Description
Register	Read, Write	NBLRegisterCreateWF(). Algorithms by Sundell and Tsigas [103].	Wait-Free. Uses time-stamps with a size that are calculated using timing information available in real-time systems.
Snapshot	Scan, Update	NBLSUSnapshotCreateWF(). Algorithms by Ermedahl et al [29].	Wait-Free. Single updater allowed per component. Uses the Test-And-Set (TAS) atomic primitive.
		NBLMUSnapshotCreateWF(). Algorithms by Kirousis et al [63]	Wait-Free. Multiple updaters allowed per component. Uses matrixes of shared registers.
		NBLMUSnapshotCreateWF2(). Algorithms by Sundell et al [112]	Wait-Free. Multiple updaters allowed per component. Uses cyclical arrays with lengths calculated using timing information available in real-time systems.
		NBLSUSnapshotCreateLB(). NBLMUSnapshotCreateLB().	Lock-Based. Uses the in NOBLE configured mutual exclusion method (spin-locks) and memory management scheme (malloc).

Table 4.2: The shared data objects supported by NOBLE (continued)

For example the implementation of the shared data object *Queue* is `struct NBLQueue` in NOBLE.

4.2.3 Easy to Adapt

For many of the shared data objects that are realized in NOBLE, a set of different implementations is offered. The user can select the implementation that better suits the application needs. The selection is done at the creation of the shared data object. For example, there are several different creation functions for the different implementations of the Queue shared data object in NOBLE, their usage is described below.

```
NBLQueue *NBLQueueCreateLF(int nrOfBlocks, int backOff);
/* Create a Queue using the implementation LF */
NBLQueue *NBLQueueCreateLF2(int nrOfBlocks); /* Create a Queue
using the implementation LF2 */
```

NOBLE also offers a simple interface to standard lock-based implementations of all shared data objects provided. The mechanisms for handling mutual exclusion and dynamic memory allocation can be configured in NOBLE, the default built-in mechanisms are simple spin-locks respective the systems standard memory manager - *malloc*.

In all other steps of use of the shared data object, the programmer does not have to remember or supply any information about the implementation (synchronization method) used. This means that all other functions regarding operations on the shared data objects have only to be informed about the actual instance of the shared data object. The latter gives a unified interface to the user: all operations take the same number of arguments and have the same return value(s) independently of the implementation:

```
NBLQueueFree(handle);
NBLQueueEnqueue(handle,item);
NBLQueueDequeue(handle);
```

All names for the operations are the same regardless of the actual implementation type of the shared data object, and more significantly the semantics of the operations are also the same.

4.2.4 Efficiency

The only information that is passed to the library during the invocation of an operation is a *handle* to a private data structure. Henceforth, any information concerning the implementation method, which is used for this particular shared data object instance, has to be inside this private data structure. For fast redirection of the program flow to the correct implementation, function pointers are used. Each instance of the data structure itself contains a set of function pointers, one for each operation that can be applied on it:

```
typedef struct NBLQueue {
     void *data;
     void (*free)(void *data);
     void (*enqueue)(void *data,void *item);
     void *(*dequeue)(void *data);
} NBLQueue;
```

The use of function pointers inside each instance, allows us to produce inline redirection of the program flow to the correct implementation. Instead of having one central function that redirects, we define several macros that redirect directly from the user-level. From the user's perspective this usually makes no difference, these macros can be used in the same way as pure functions:

```
   #define NBLQueueFree(handle) (handle->free(handle->data))
   #define NBLQueueEnqueue(handle,item) (handle->enqueue(handle->
data,item))
   #define NBLQueueDequeue(handle) (handle->dequeue(handle->data))
```

4.2.5 Portability

The interface to NOBLE has been designed to be the same independently of the chosen platform. Internal library dependencies are not visible outside of the library. Application calls to the library look exactly the same with respect to the NOBLE library when moving to another platform. The implementations that are contained in NOBLE, use hardware synchronization primitives (Compare-And-Swap, Test-And-Set, Fetch-And-Add, Load-Link/Store-Conditional) [45, 118], that are widely available in many com-

monly used architectures. Still, in order to achieve the same platform independent interface, NOBLE has been designed to provide an internal level abstraction of the hardware synchronization primitives. All hardware dependent operations that are used in the implementation of NOBLE, are reached using the same interface:

```
#include "Platform/Primitives.h"
```

This file depends on the actual platform and is only visible to the developers of the library. However, a set of implementations (with the same syntax and semantics) of the different synchronization primitives needed by the implementations have to be provided for the different platforms. These implementations are only visible inside NOBLE and there are no restrictions on the way they are implemented.

NOBLE at this point has been successfully implemented on the SUN Sparc - Solaris platform, the Silicon Graphics Mips - Irix platform, the Intel x86 - Win32 platform as well as the Intel x86 - Linux platform.

4.2.6 Adaptable for different programming languages

NOBLE is realized in C and therefore easily adaptable to other popular programming languages that support importing functions from C libraries.

C++ is directly usable with NOBLE. The basic structures and operation calls of the NOBLE shared data objects have been defined in such a way that real C++ class functionality can also be easily achieved by using wrap-around classes, with no loss of performance.

```
class NOBLEQueue {
private:
     NBLQueue* queue;
public:
     NOBLEQueue(int type) {if(type==NBL_LOCKFREE) queue=
           NBLQueueCreateLF(); else ...  }
     ~ NOBLEQueue() {NBLQueueFree(queue);}
     inline void Enqueue(void *item) {NBLQueueEnqueue(queue,item);}
     inline void *Dequeue() {return NBLQueueDequeue(queue);}
};
```

Because of the inline statements and the fact that the function calls in NOBLE are defined as macros, the function calls of the class members will

be resolved in nearly the same way as in C, with virtually no performance loss.

4.3 Examples

In this section we give an overview of NOBLE by way of an example: *a shared stack* in a multi-threaded program. First we have to create the stack using the appropriate create functions for the implementation that we want to use for this data object. We decide to use the implementation *LF* that requires to supply the maximum size of the stack and the type of back-off strategy as an input to the create function. We select 10000 stack-elements for the maximum size of the stack and the exponential back-off strategy:

```
stack=NBLStackCreateLF(10000, BOT_EXPONENTIAL);
```
where stack is a globally defined pointer variable:
```
NBLStack *stack;
```

Whenever we have a thread that wants to invoke a stack operation the appropriate function has to be called:

```
NBLStackPush(stack, item);
```
or
```
item=NBLStackPop(stack);
```

When our program does not need the stack any more we can do some cleaning and give back the memory allocated for the stack:

```
NBLStackFree(stack);
```

If we decide later on to change the implementation of the *stack* that our program uses, we only have to change one single line in our program. For example if we want to change from the *LF* implementation to the *LB* implementation we only have to change the line:

```
stack=NBLStackCreateLF(10000, BOT_EXPONENTIAL);
```
to
```
stack=NBLStackCreateLB();
```

Experiment	Operation 1	Operation 2	Operation 3	Operation 4	Operation 5
Queue	Enqueue 50%	Dequeue 50%			
Stack	Push 50%	Pop 50%			
Snapshot	Update or Scan 100%				
Singly Linked List	First 10%	Next 20%	Insert 60%	Delete 10%	
Queue - Low	Enqueue 25%	Dequeue 25%	Sleep 50%		
Stack - Low	Push 25%	Pop 25%	Sleep 50%		
Snapshot - Low	Update or Scan 50%	Sleep 50%			
Singly Linked List - Low	First 5%	Next 10%	Insert 30%	Delete 5%	Sleep 50%

Table 4.3: The distribution characteristics of the random operations

4.4 Experiments

We have performed a significant number of experiments in order to measure the performance benefits that can be achieved from the use of NOBLE. Since lock-based synchronization is known to usually perform better than non-blocking synchronization when contention is very low, we used the following micro-benchmarks in order to be as fair as possible:

- High contention - The concurrent threads are continuously invoking operations, one after the other to the shared data object, thus maximizing the contention.
- Low contention - Each concurrent thread performs other tasks between two consecutive operations to the shared data object. The contention in this case is lower, and quite often only one thread is using the shared data object at one time.

In our experiments each concurrent thread performs 50 000 randomly chosen sequential operations. Each experiment is repeated 50 times, and an

average execution time for each experiment is estimated. Exactly the same sequential operations are performed for all different implementations compared. Where possible, the lock-free implementations have been configured to use the exponential back-off strategy. All lock-based implementations are based on simple spin-locks using the TAS atomic primitive. For the low contention experiments each thread randomly selects to perform a set of 1000 to 2000 sequential writes to a shared memory register with a new computed value. A clean-cache operation is performed just before each subexperiment. The distributions characteristics of the random operations for each experiment are shown in table 4.3.

The experiments were performed using different number of threads, varying from 1 to 30. We performed our experiments on a Sun Enterprise 10000 StarFire [23] system. At that point we could have access to 30 processors and each thread could run on its own processor, utilizing full concurrency. We have also run a similar set of experiments on a Silicon Graphics Origin 2000 [70] system, where we had access to 63 processors. Besides from those two highly parallel super-computers, a set of experiments was also performed on a Compaq dual-processor Pentium II PC running Win32 as well as Linux. The concurrent tasks in the experiments where executed using the pthread package on the Solaris and the Linux platforms. On the Win32 platform the default system threads were used instead. The pthread package is implemented a bit differently on the Irix platform, so in order to run our experiments accurately, we had to execute the tasks using standard Unix processes and also implement our own shared memory allocation package.

The results from these experiments are shown in Figures 4.1,4.2,4.3 and 4.4, the average execution time is drawn as a function of the number of processes. From all the results that we collected we could definitely conclude that NOBLE, largely because of its non-blocking characteristics and partly because of its efficient implementation, outperforms the respective lock-based implementations significantly. For the singly linked list, NOBLE was up to 64 times faster than the lock-based implementation. In general, the performance benefits from using NOBLE and lock-free synchronization methods increase with the number of processors. For the experiments with low contention, NOBLE still performs better than the respective lock-based implementations, for a high number of processors NOBLE performs up to 3 times faster than the respective lock-based implementations.

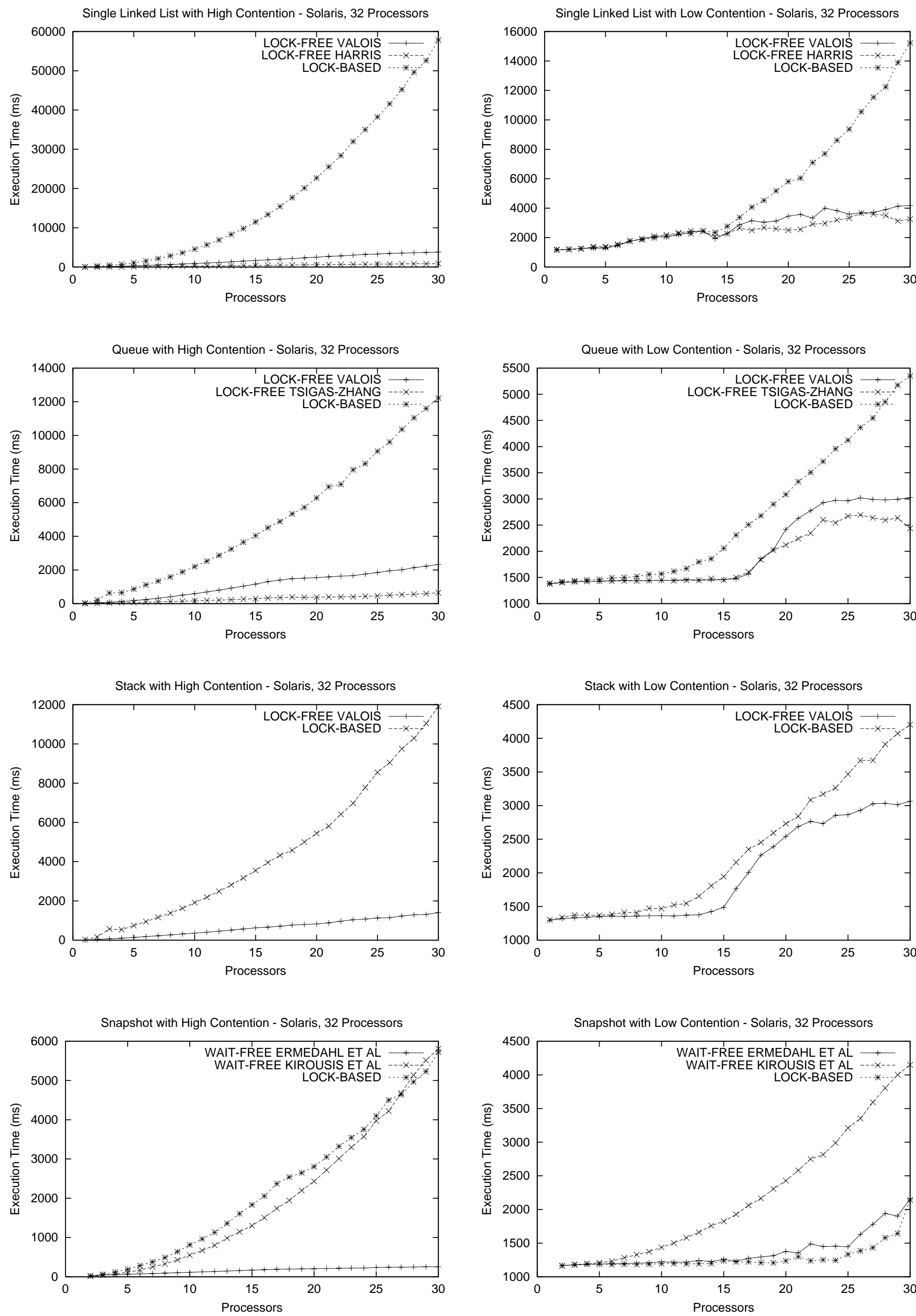


Figure 4.1: Experiments on SUN Enterprise 10000 - Solaris

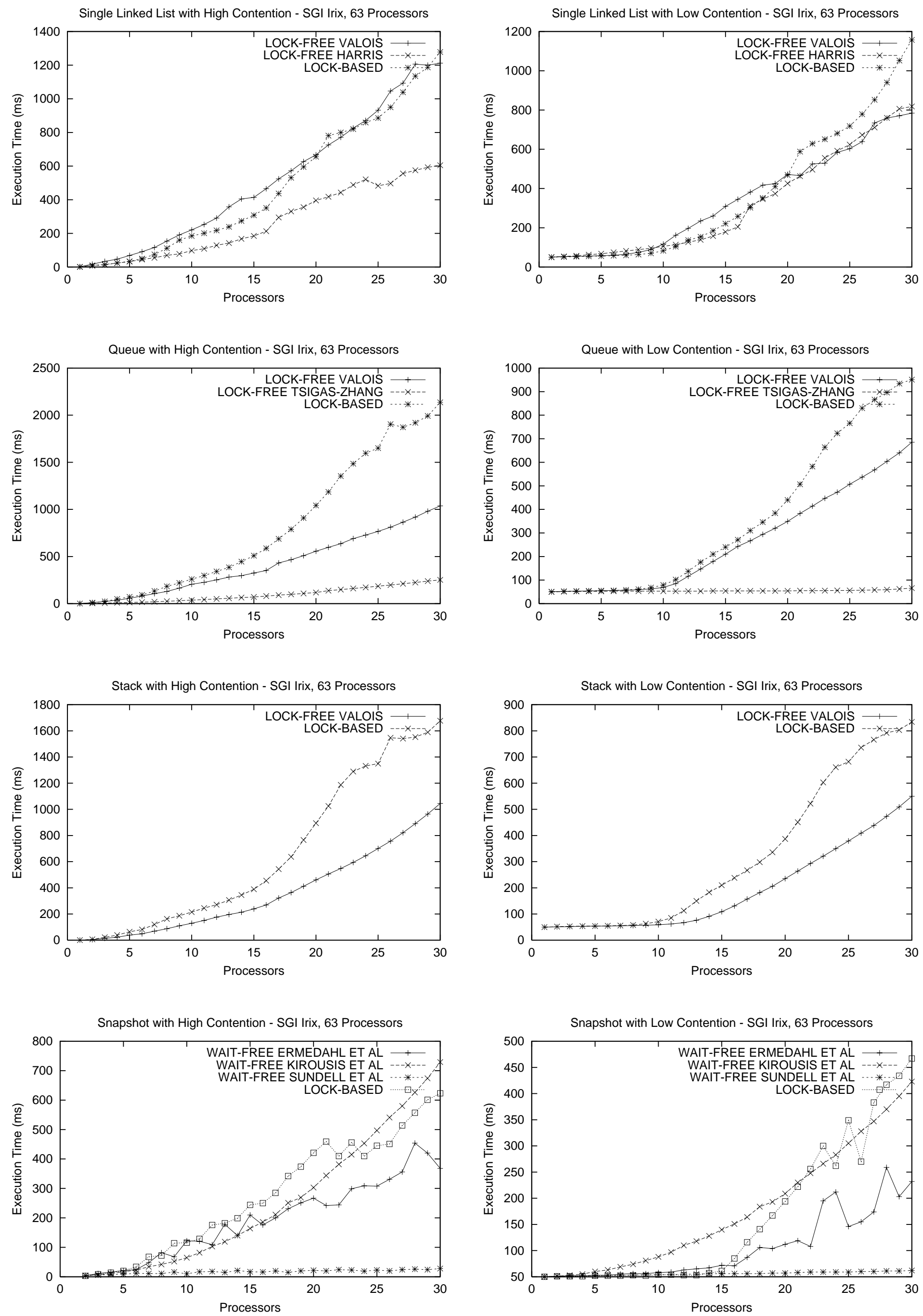


Figure 4.2: Experiments on SGI Origin 2000 - Irix

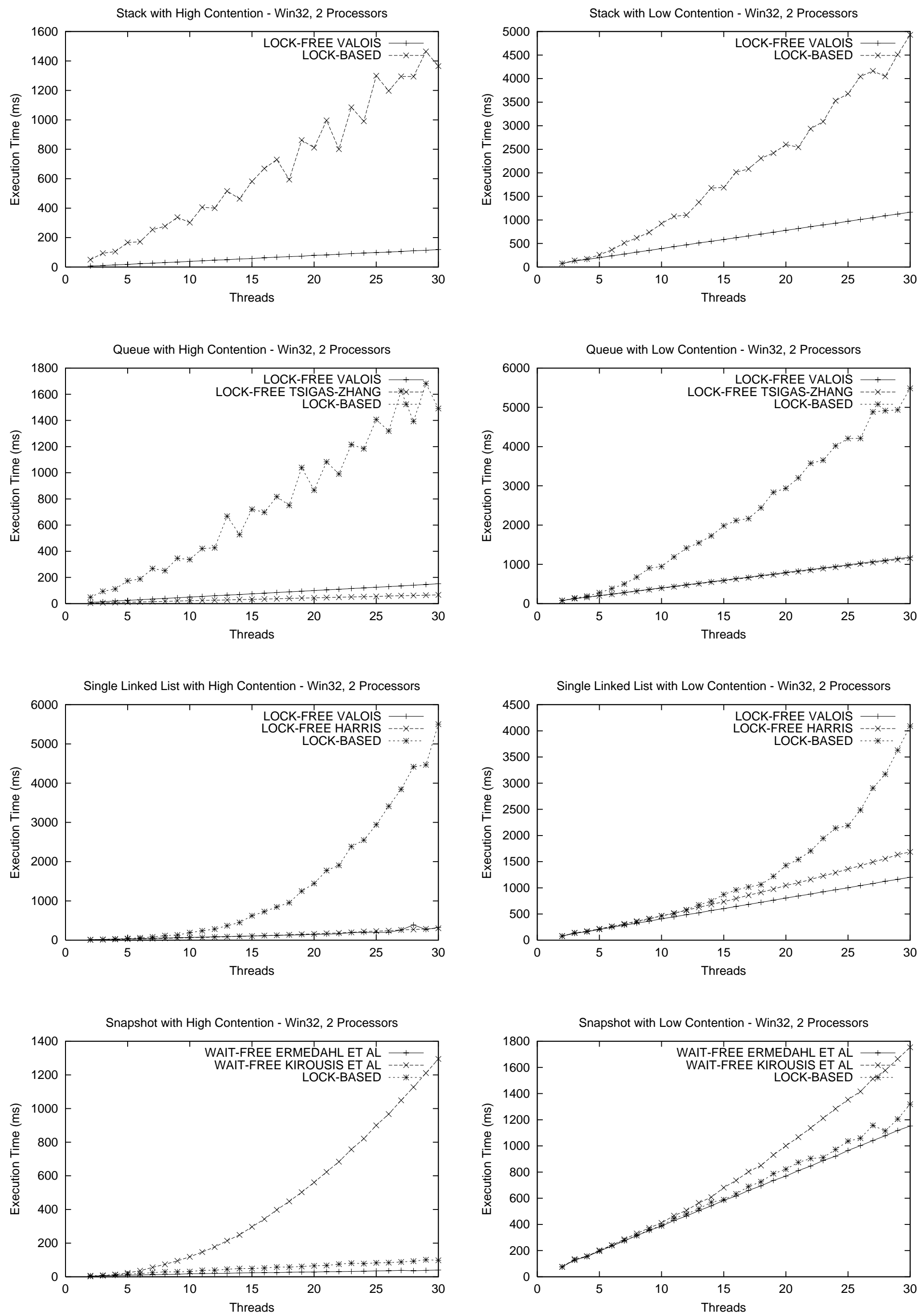


Figure 4.3: Experiments on Dual Pentium II - Win32

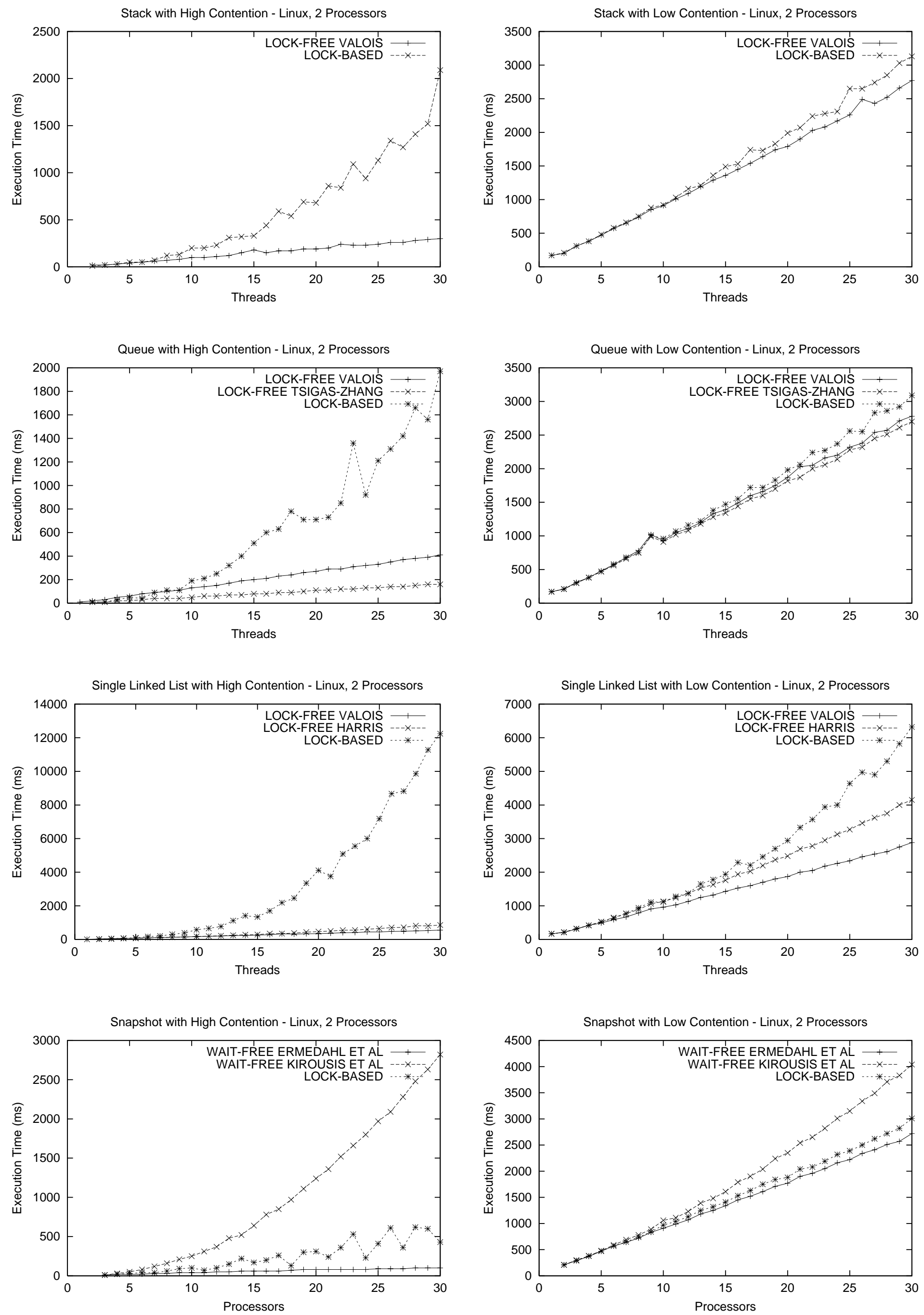


Figure 4.4: Experiments on Dual Pentium II - Linux

4.5 Conclusions

NOBLE is a library for non-blocking synchronization that includes implementations of several fundamental and commonly used shared data objects. The library is easy to use and existing programs can be easily adapted to use it. The programs using the library, and the library itself, can be easily tuned to include different synchronization mechanisms for each of the supported shared data objects. Experiments show that the non-blocking implementations in NOBLE offer significant improvements in performance, especially on multi-processor platforms. NOBLE currently supports four platforms, the architectures of SUN Sparc with Solaris, Intel x86 with Win32, Intel x86 with Linux and SGI Mips with Irix.

The first versions of NOBLE have just been made available for outside use and can be used freely for the purpose of research and teaching. It can be downloaded from http://www.noble-library.org. We hope that NOBLE will narrow the gap between theoretical research and practical application.

Future work in the NOBLE project includes the extension of the library with more implementations and new shared data objects, as well as porting it to platforms more specific for real-time systems.

.

Chapter 5

Fast and Lock-Free Concurrent Priority Queues for Multi-Thread Systems[1]

Håkan Sundell, Philippas Tsigas
Department of Computing Science
Chalmers Univ. of Technol. and Göteborg Univ.
412 96 Göteborg, Sweden
E-mail: {phs, tsigas}@cs.chalmers.se

Abstract

We present an efficient and practical lock-free implementation of a concurrent priority queue that is suitable for both fully concurrent (large multi-processor) systems as well as pre-emptive (multi-process) systems. Many algorithms for concurrent priority queues are based on mutual exclusion. However, mutual exclusion causes blocking which has several drawbacks and degrades the system's overall performance. Non-blocking algorithms avoid blocking, and several implementations have been proposed. Previously known non-blocking algorithms of priority queues did not perform well in practice

[1] This is an extended and revised version of the paper with the same title that was presented at IPDPS 2003 [106] that was awarded with the best paper award in the algorithms category, and published as a technical report [105].

because of their complexity, and they are often based on non-available atomic synchronization primitives. Our algorithm is based on the randomized sequential list structure called skip list, and a real-time extension of our algorithm is also described. In our performance evaluation we compare our algorithm with a well representable set of earlier implementations of priority queues known. The experimental results clearly show that our lock-free implementation outperforms the other lock-based implementations in practical scenarios for 3 threads and more, both on fully concurrent as well as on pre-emptive systems.

5.1 Introduction

Priority queues are fundamental data structures. From the operating system level to the user application level, they are frequently used as basic components. For example, the ready-queue that is used in the scheduling of tasks in many real-time systems, can usually be implemented using a concurrent priority queue. Consequently, the design of efficient implementations of priority queues is a research area that has been extensively researched. A priority queue supports two operations, the *Insert* and the *DeleteMin* operation. The abstract definition of a priority queue is a set of key-value pairs, where the key represents a priority. The *Insert* operation inserts a new key-value pair into the set, and the *DeleteMin* operation removes and returns the value of the key-value pair with the lowest key (i.e. highest priority) that was in the set.

To ensure consistency of a shared data object in a concurrent environment, the most common method is to use mutual exclusion, i.e. some form of locking. Mutual exclusion degrades the system's overall performance [100] as it causes blocking, i.e. other concurrent operations can not make any progress while the access to the shared resource is blocked by the lock. Using mutual exclusion can also cause deadlocks, priority inversion (which can be solved efficiently on uni-processors [95] with the cost of more difficult analysis, although not as efficient on multiprocessor systems [92]) and even starvation.

To address these problems, researchers have proposed non-blocking algorithms for shared data objects. Non-blocking methods do not involve mutual exclusion, and therefore do not suffer from the problems that blocking can cause. Lock-free implementations are non-blocking and guarantee that regardless of the contention caused by concurrent operations and the interleaving of their sub-operations, always at least one operation will progress.

However, there is a risk for starvation as the progress of other operations could cause one specific operation to never finish. This is although different from the type of starvation that could be caused by blocking, where a single operation could block every other operation forever, and cause starvation of the whole system. Wait-free [45] algorithms are lock-free and moreover they avoid starvation as well, in a wait-free algorithm every operation is guaranteed to finish in a limited number of steps, regardless of the actions of the concurrent operations. Recently, researchers also include obstruction-free [48] implementations to be non-blocking, although this kind of implementation is weaker than lock-free and thus does not guarantee progress of any concurrent operation. Non-blocking algorithms have been shown to be of big practical importance in practical applications [115, 118], and recently NOBLE, which is a non-blocking inter-process communication library, has been introduced [104].

There exist several algorithms and implementations of concurrent priority queues. The majority of the algorithms are lock-based, either with a single lock on top of a sequential algorithm, or specially constructed algorithms using multiple locks, where each lock protects a small part of the shared data structure. Several different representations of the shared data structure are used, for example: Hunt *et al.* [54] presents an implementation which is based on heap structures, Grammatikakis *et al.* [36] compares different structures including cyclic arrays and heaps, and most recently Lotan and Shavit [72] presented an implementation based on the skip list structure [91]. The algorithm by Hunt *et al.* locks each node separately and uses a technique to scatter the accesses to the heap, thus reducing the contention. Its implementation is publicly available and its performance has been documented on multi-processor systems. Jones [60] also makes use of multiple locks, but implements a fully dynamic tree structure, and tries to only lock the part of the tree neccessary at each moment in time. Lotan and Shavit extend the functionality of the concurrent priority queue and assume the availability of a global high-accuracy clock. They apply a lock on each pointer, and as the multi-pointer based skip list structure is used, the number of locks is significantly more than the number of nodes. Its performance has previously only been documented by simulation, with very promising results. The algorithm by Shavit and Zemach [99] is not adressed in this paper, as they implement a bounded[2] priority queue, whereas we adress the general priority queues.

Israeli and Rappoport have presented a wait-free algorithm for a con-

[2] The set of possible priorities is restricted.

current priority queue [55]. This algorithm makes use of strong atomic synchronization primitives that have not been implemented in any currently existing platform. Greenwald has also presented an outline for a lock-free priority queue [37] based on atomic primitives that are not available in modern multiprocessor systems. However, there exists an attempt for a wait-free algorithm by Barnes [19] that uses existing atomic primitives, though this algorithm does not comply with the generally accepted definition of the wait-free property. The algorithm is not yet implemented and the theoretical analysis predicts worse behavior than the corresponding sequential algorithm, which makes it not of practical interest.

One common problem with many algorithms for concurrent priority queues is the lack of precise defined semantics of the operations. It is also seldom that the correctness with respect to concurrency is proved, using a strong property like linearizability [50].

In this paper we present a lock-free algorithm of a concurrent priority queue that is designed for efficient use in both pre-emptive as well as in fully concurrent environments. Inspired by Lotan and Shavit [72], the algorithm is based on the randomized skip list [91] data structure, but in contrast to [72] it is lock-free. It is also implemented using common synchronization primitives that are available in modern systems. The algorithm is described in detail later in this paper, and the aspects concerning the underlying lock-free memory management are also presented. The precise semantics of the operations are defined and a proof is given that our implementation is lock-free and linearizable.

We have performed experiments that compare the performance of our algorithm with a well representable set of earlier implementations of concurrent priority queues known, i.e. the implementation by Lotan and Shavit [72], the implementation by Hunt *et al.* [54], and the implementation by Jones [60]. Experiments were performed on three different platforms, consisting of a multiprocessor system using different operating systems and equipped with either 2, 6 or 29 processors. Our results show that our algorithm outperforms the other lock-based implementations in practical scenarios for 3 threads and more, in both highly pre-emptive as well as in fully concurrent environments. We also present an extended version of our algorithm that also addresses certain real-time aspects of the priority queue as introduced by Lotan and Shavit [72].

The rest of the paper is organized as follows. In Section 5.2 we define the properties of the systems that our implementation is aimed for. The actual algorithm is described in Section 5.3. In Section 5.4 we define the precise semantics for the operations on our implementations, as well show-

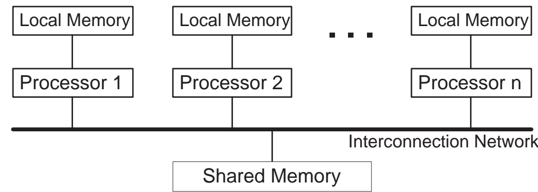


Figure 5.1: Shared Memory Multiprocessor System Structure

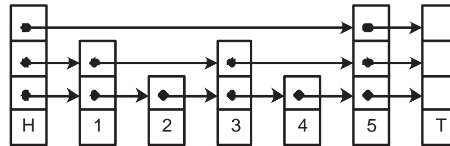


Figure 5.2: The skip list data structure with 5 nodes inserted.

ing correctness by proving the lock-free and linearizability property. The experimental evaluation that shows the performance of our implementation is presented in Section 5.5. In Section 5.6 we extend our algorithm with functionality that can be needed for specific real-time applications. In Section 5.7 we discuss related work with skip lists that have appeared in the literature after our first publications. We conclude the paper with Section 5.8.

5.2 System Description

A typical abstraction of a shared memory multi-processor system configuration is depicted in Figure 5.1. Each node of the system contains a processor together with its local memory. All nodes are connected to the shared memory via an interconnection network. A set of co-operating tasks is running on the system performing their respective operations. Each task is sequentially executed on one of the processors, while each processor can serve (run) many tasks at a time. The co-operating tasks, possibly running on different processors, use shared data objects built in the shared memory to co-ordinate and communicate. Tasks synchronize their operations on the shared data objects through sub-operations on top of a cache-coherent shared memory. The shared memory may not though be uniformly accessible for all nodes in the system; processors can have different access times on different parts

structure Node
  key,level,validLevel ⟨,*timeInsert*⟩ : **integer**
  value : **pointer to word**
  next[level],prev : **pointer to** Node

Figure 5.3: The Node structure.

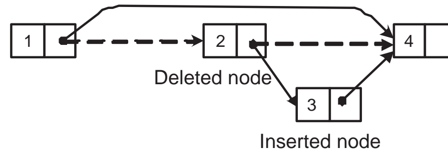


Figure 5.4: Concurrent insert and delete operation can delete both nodes.

of the memory.

5.3 Algorithm

The algorithm is based on the sequential skip list data structure invented by Pugh [91]. This structure uses randomization and has a probabilistic time complexity of $O(logN)$ where N is the maximum number of elements in the list. The data structure is basically an ordered list with randomly distributed short-cuts in order to improve search times, see Figure 5.2. The maximum height (i.e. the maximum number of next pointers) of the data structure is $logN$. The height of each inserted node is randomized geometrically in the way that 50% of the nodes should have height 1, 25% of the nodes should have height 2 and so on. To use the data structure as a priority queue, the nodes are ordered in respect of priority (which has to be unique for each node[3]), the nodes with highest priority are located first in the list. The fields of each node item are described in Figure 5.3 as it is used in this implementation. For all code examples in this paper, code that is between the "⟨" and "⟩" symbols are only used for the special real-time[4] version of our implementation that involves timestamps (see Section 5.6), and are thus

[3]In order to assign several objects the same priority, this limitation can be overcome by building the priority (key) so that only some bits represent the real priority and remaing bits are choosen in order to achive uniqueness.

[4]In the sense that DeleteMin operations can only return items that were fully inserted before the start of the DeleteMin operation.

procedure FAA(address:**pointer to word**, number:**integer**)
  atomic do
    *address := *address + number;

function CAS(address:**pointer to word**, oldvalue:**word**,
newvalue:**word**):**boolean**
  atomic do
    if *address = oldvalue **then**
      *address := newvalue;
      return true;
    else return false;

Figure 5.5: The Fetch-And-Add (FAA) and Compare-And-Swap (CAS) atomic primitives.

not included in the standard version of the implementation.

In order to make the skip list construction concurrent and non-blocking, we are using two of the standard atomic synchronization primitives, Fetch-And-Add (FAA) and Compare-And-Swap (CAS). Figure 5.5 describes the specification of these primitives which are available in most modern platforms.

As we are concurrently (with possible preemptions) traversing nodes that will be continuously allocated and reclaimed, we have to consider several aspects of memory management. No node should be reclaimed and then later re-allocated while some other process is traversing this node. This can be solved for example by careful reference counting. We have selected to use the lock-free memory management scheme invented by Valois [122] and corrected by Michael and Scott [83]. This was selected because of its simplicity, completeness with lock-free memory allocation, full reference counting that guarantees full stability (with no needs for retries from the head node) while traversing even when exposed to concurrent changes, and that it only makes use of commonly available atomic synchronization primitives like FAA and CAS.

To insert or delete a node from the list we have to change the respective set of next pointers. These have to be changed consistently, but not necessary all at once. Our solution is to have additional information on each node about its deletion (or insertion) status. This additional information will guide the concurrent processes that might traverse into one partial deleted or inserted node. When we have changed all necessary next pointers, the node is fully deleted or inserted.

One problem, that is general for non-blocking implementations that are based on the linked-list structure, arises when inserting a new node into the list. Because of the linked-list structure one has to make sure that the previous node is not about to be deleted. If we are changing the next pointer of this previous node atomically with CAS, to point to the new node, and then immediately afterwards the previous node is deleted - then the new node will be deleted as well, as illustrated in Figure 5.4. There are several solutions to this problem. One solution is to use the CAS2 operation as it can change two pointers atomically, but this operation is not available in any modern multi-processor system. A second solution is to insert auxiliary nodes [122] between each two normal nodes, and the latest method introduced by Harris [44] is to use one bit of the pointer values as a deletion mark. On most modern 32-bit systems, 32-bit values can only be located at addresses that are evenly dividable by 4, therefore bits 0 and 1 of the address are always set to zero. The method is then to use the previously unused bit 0 of the next pointer to mark that this node is about to be deleted, using CAS. Any concurrent *Insert* operation will then be notified about the deletion, when its CAS operation will fail.

One memory management issue is how to de-reference pointers safely. If we simply de-reference the pointer, it might be that the corresponding node has been reclaimed before we could access it. It can also be that bit 0 of the pointer was set, thus marking that the node is deleted, and therefore the pointer is not valid. The following functions are defined for safe handling of the memory management:

function READ_NODE(address:**pointer to pointer to** Node):**pointer to** Node /* De-reference the pointer and increase the reference counter for the corresponding node. In case the pointer is marked, NULL is returned */

function COPY_NODE(node:**pointer to** Node):**pointer to** Node /* Increase the reference counter for the corresponding given node */

procedure RELEASE_NODE(node:**pointer to** Node) /* Decrement the reference counter on the corresponding given node. If the reference count reaches zero, then call RELEASE_NODE on the nodes that this node has owned pointers to, then reclaim the node */

While traversing the nodes, processes will eventually reach nodes that are marked to be deleted. As the process that invoked the corresponding *DeleteMin* operation might be pre-empted, this *DeleteMin* operation has to be helped to finish before the traversing process can continue. However, it

```
// Global variables
head,tail : pointer to Node
// Local variables
node2 : pointer to Node

function ReadNext(node1:pointer to pointer to Node,
 level:integer):pointer to Node
R1    if IS_MARKED((*node1).value) then
R2        *node1:=HelpDelete(*node1,level);
R3    node2:=READ_NODE((*node1).next[level]);
R4    while node2=NULL do
R5        *node1:=HelpDelete(*node1,level);
R6        node2:=READ_NODE((*node1).next[level]);
R7    return node2;

function ScanKey(node1:pointer to pointer to Node,
 level:integer, key:integer):pointer to Node
S1    node2:=ReadNext(node1,level);
S2    while node2.key < key do
S3        RELEASE_NODE(*node1);
S4        *node1:=node2;
S5        node2:=ReadNext(node1,level);
S6    return node2;
```

Figure 5.6: Functions for traversing the nodes in the skip list data structure.

is only necessary to help the part of the *DeleteMin* operation on the current level in order to be able to traverse to the next node. The function *ReadNext*, see Figure 5.6, traverses to the next node of *node1* on the given level while helping (and then sets *node1* to the previous node of the helped one) any marked nodes in between to finish the deletion. The function *ScanKey*, see Figure 5.6, traverses in several steps through the next pointers (starting from *node1*) at the current level until it finds a node that has the same or higher key (priority) value than the given key. It also sets *node1* to be the previous node of the returned node.

The implementation of the *Insert* operation, see Figure 5.7, starts in lines I2-I4 with creating the new node (*newNode*) and choosing its height (*level*) by calling the *randomLevel* function. This function roughly simulates a repeated coin tossing, counting the number of times (up to the maximum level) the upper (or lower if that was choosen) side of the coin turns up from the start of the function, thus giving the distribution associated with

```
// Local variables
node1,node2,newNode,savedNodes[maxlevel]: pointer to Node

function Insert(key:integer, value:pointer to word):boolean
I1    ⟨TraverseTimeStamps();⟩
I2    level:=randomLevel();
I3    newNode:=CreateNode(level,key,value);
I4    COPY_NODE(newNode);
I5    node1:=COPY_NODE(head);
I6    for i:=maxLevel-1 to 1 step -1 do
I7        node2:=ScanKey(&node1,i,key);
I8        RELEASE_NODE(node2);
I9        if i<level then savedNodes[i]:=COPY_NODE(node1);
I10   while true do
I11       node2:=ScanKey(&node1,0,key);
I12       value2:=node2.value;
I13       if not IS_MARKED(value2) and node2.key=key then
I14           if CAS(&node2.value,value2,value) then
I15               RELEASE_NODE(node1);
I16               RELEASE_NODE(node2);
I17               for i:=1 to level-1 do
I18                   RELEASE_NODE(savedNodes[i]);
I19               RELEASE_NODE(newNode);
I20               RELEASE_NODE(newNode);
I21               return true_2;
I22           else
I23               RELEASE_NODE(node2);
I24               continue;
I25       newNode.next[0]:=node2;
I26       RELEASE_NODE(node2);
I27       if CAS(&node1.next[0],node2,newNode) then
I28           RELEASE_NODE(node1);
I29           break;
I30       Back-Off
I31   for i:=1 to level-1 do
I32       newNode.validLevel:=i;
I33       node1:=savedNodes[i];
I34       while true do
I35           node2:=ScanKey(&node1,i,key);
I36           newNode.next[i]:=node2;
I37           RELEASE_NODE(node2);
I38           if IS_MARKED(newNode.value) or CAS(&node1.next[i],node2,newNode) then
I39               RELEASE_NODE(node1);
I40               break;
I41           Back-Off
I42   newNode.validLevel:=level;
I43   ⟨newNode.timeInsert:=getNextTimeStamp();⟩
I44   if IS_MARKED(newNode.value) then newNode:=HelpDelete(newNode,0);
I45   RELEASE_NODE(newNode);
I46   return true;
```

Figure 5.7: The algorithm for the Insert operation.

```
// Local variables
prev,last,node1,node2 : pointer to Node

function DeleteMin():pointer to Node
D1      ⟨TraverseTimeStamps();⟩
D2      ⟨time:=getNextTimeStamp();⟩
D3      prev:=COPY_NODE(head);
D4      while true do
D5          node1:=ReadNext(&prev,0);
D6          if node1=tail then
D7              RELEASE_NODE(prev);
D8              RELEASE_NODE(node1);
D9              return NULL;
        retry:
D10         value:=node1.value;
D11         if node1 ≠ prev.next[0] then
D12             RELEASE_NODE(node1);
D13             continue;
D14         if not IS_MARKED(value) ⟨and compareTimeStamp(time,
              node1.timeInsert)>0⟩ then
D15             if CAS(&node1.value,value,GET_MARKED(value)) then
D16                 node1.prev:=prev;
D17                 break;
D18             else goto retry;
D19         else if IS_MARKED(value) then
D20             node1:=HelpDelete(node1,0);
D21         RELEASE_NODE(prev);
D22         prev:=node1;
D23     for i:=0 to node1.level-1 do
D24         repeat
D25             node2:=node1.next[i];
D26         until IS_MARKED(node2) or CAS(&node1.next[i],node2,
              GET_MARKED(node2));
D27     prev:=COPY_NODE(head);
D28     for i:=node1.level-1 to 0 step -1 do
D29         RemoveNode(node1,&prev,i);
D30     RELEASE_NODE(prev);
D31     RELEASE_NODE(node1);
D32     RELEASE_NODE(node1); /* Delete the node */
D33     return value;
```

Figure 5.8: The algorithm for the DeleteMin operation.

```
// Local variables
last: pointer to Node

procedure RemoveNode(node: pointer to Node,
  prev: pointer to pointer to Node, level:integer)
RN1    while true do
RN2        if node.next[level]=1 then break;
RN3        last:=ScanKey(prev,level,node.key);
RN4        RELEASE_NODE(last);
RN5        if last≠node or node.next[level]=1 then break;
RN6        if CAS(&(*prev).next[level],node,
             GET_UNMARKED(node.next[level])) then
RN7            node.next[level]:=1;
RN8            break;
RN9        if node.next[level]=1 then break;
RN10       Back-Off
```

Figure 5.9: The algorithm for the RemoveNode function.

the skip list [91]. In lines I5-I11 the implementation continues with a search phase to find the node after which *newNode* should be inserted. This search phase starts from the head node at the highest level and traverses down to the lowest level until the correct node is found (*node1*). When going down one level, the last node traversed on that level is remembered (*savedNodes*) for later use (this is where we should insert the new node at that level). Now it is possible that there already exists a node with the same priority as of the new node, this is checked in lines I12-I24, the value of the old node (*node2*) is changed atomically with a CAS. Otherwise, in lines I25-I42 it starts trying to insert the new node starting with the lowest level and increasing up to the level of the new node. The next pointers of the nodes (to become previous) are changed atomically with a CAS. After the new node has been inserted at the lowest level, it is possible that it is deleted by a concurrent *DeleteMin* operation before it has been inserted at all levels, and this is checked in lines I38 and I44.

The *RemoveNode* procedure, see Figure 5.9, removes the given *node* from the linked list structure at the given *level*, using a given hint *prev* for the previous node. It first searches for the right position of the previous node according to the key of *node* in line RN3. It verifies in line RN5 that *node* is still part of the linked list structure at the present level. In line RN6 it tries to remove *node* by changing the next pointer of the previous node using

```
// Local variables
prev,last,node2 : pointer to Node

function HelpDelete(node:pointer to Node, level:integer):pointer to Node
H1    for i:=level to node.level-1 do
H2        repeat
H3            node2:=node.next[i];
H4        until IS_MARKED(node2) or CAS(&node.next[i],node2,
              GET_MARKED(node2));
H5    prev:=node.prev;
H6    if not prev or level≥prev.validLevel then
H7        prev:=COPY_NODE(head);
H8        for i:=maxLevel-1 to level step -1 do
H9            node2:=ScanKey(&prev,i,node.key);
H10           RELEASE_NODE(node2);
H11   else COPY_NODE(prev);
H12   RemoveNode(node,&prev,level);
H13   RELEASE_NODE(node);
H14   return prev;
```

Figure 5.10: The algorithm for the HelpDelete function.

the CAS sub-operation. If the CAS failed, possibly because of concurrent changes to the *prev* node, the whole procedure retries. As this procedure can be invoced concurrently on the same node argument, it synchronizes with the possibly other invocations in lines RN2, RN5, RN7 and RN9 in order to avoid executing sub-operations that have already been performed.

The *DeleteMin* operation, see Figure 5.8, starts from the head node and finds the first node (*node*1) in the list that does not have its deletion mark on the value set, see lines D3-D15. It tries to set this deletion mark in line D15 using the CAS primitive, and if it succeeds it also writes a valid pointer to the prev field of the node. This prev field is necessary in order to increase the performance of concurrent *HelpDelete* functions, these operations otherwise would have to search for the previous node in order to complete the deletion. The next step is to mark the deletion bits of the next pointers in the node, starting with the lowest level and going upwards, using the CAS primitive in each step, see lines D23-D26. Afterwards in lines D27-D29 it starts the actual deletion by calling the *RemoveNode* procedure, starting at the highest level and continuing downwards. The reason for doing the deletion in decreasing order of levels, is that concurrent search operations also start at the highest level and proceed downwards, in this way the concurrent search operations

will sooner avoid traversing this node.

The algorithm has been designed for pre-emptive as well as fully concurrent systems. In order to achieve the lock-free property (that at least one thread is doing progress) on pre-emptive systems, whenever a search operation finds a node that is about to be deleted, it calls the *HelpDelete* function and then proceeds searching from the previous node of the deleted. The *HelpDelete* function, see Figure 5.10, tries to fulfill the deletion on the current level and returns a reference to the previous node when the deletion is completed. It starts in lines H1-H4 with setting the deletion mark on all next pointers in case they have not been set. In lines H5-H6 it checks if the node given in the prev field is valid for deletion on the current level, otherwise it searches for the correct previous node (*prev*) in lines H7-H10. The actual deletion of this node on the current level takes place in line H12 by calling the *RemoveNode* procedure.

In fully concurrent systems though, the helping strategy can downgrade the performance significantly. Therefore the algorithm, after a number of consecutive failed attempts to help concurrent *DeleteMin* operations that hinders the progress of the current operation, puts the operation into back-off mode. When in back-off mode, the thread does nothing for a while, and in this way avoids disturbing the concurrent operations that might otherwise progress slower. The duration of the back-off is proportional to the number of threads, and for each consecutive entering of back-off mode during one operation invocation, the duration is increased exponentially.

5.4 Correctness

In this section we present the proofs of correctness for our algorithm. We first prove that our algorithm is a linearizable one [50] and then we prove that it is lock-free. A set of definitions that will help us to structure and shorten the proof is first explained in this section. We start by defining the sequential semantics of our operations and then introduce two definitions concerning concurrency aspects in general.

Definition 1 *We denote with L_t the abstract internal state of a priority queue at the time t. L_t is viewed as a set of pairs $\langle p, v \rangle$ consisting of a unique priority p and a corresponding value v. The operations that can be performed on the priority queue are Insert (I) and DeleteMin (DM). The time t_1 is defined as the time just before the atomic execution of the operation that we are looking at, and the time t_2 is defined as the time just after the atomic execution of the same operation. The return value of $true_2$ is returned by*

an Insert operation that has succeeded to update an existing node, the return value of true is returned by an Insert operation that succeeds to insert a new node. In the following expressions that defines the sequential semantics of our operations, the syntax is $S_1 : O_1, S_2$, where S_1 is the conditional state before the operation O_1, and S_2 is the resulting state after performing the corresponding operation:

$$\langle p_1, _\rangle \notin L_{t_1} : \mathbf{I_1}(\langle \mathbf{p_1}, \mathbf{v_1}\rangle) = \mathbf{true},\\ \mathbf{L_{t_2}} = \mathbf{L_{t_1}} \cup \{\langle \mathbf{p_1}, \mathbf{v_1}\rangle\} \tag{5.1}$$

$$\langle p_1, v_{1_1}\rangle \in L_{t_1} : \mathbf{I_1}(\langle \mathbf{p_1}, \mathbf{v_{1_2}}\rangle) = \mathbf{true_2},\\ \mathbf{L_{t_2}} = \mathbf{L_{t_1}} \setminus \{\langle \mathbf{p_1}, \mathbf{v_{1_1}}\rangle\} \cup \{\langle \mathbf{p_1}, \mathbf{v_{1_2}}\rangle\} \tag{5.2}$$

$$\langle p_1, v_1\rangle = \{\langle \min p, v\rangle | \langle p, v\rangle \in L_{t_1}\}\\ : \mathbf{DM_1}() = \langle \mathbf{p_1}, \mathbf{v_1}\rangle,\ \mathbf{L_{t_2}} = \mathbf{L_{t_1}} \setminus \{\langle \mathbf{p_1}, \mathbf{v_1}\rangle\} \tag{5.3}$$

$$L_{t_1} = \emptyset : \mathbf{DM_1}() = \perp \tag{5.4}$$

Definition 2 *In a global time model each concurrent operation Op "occupies" a time interval $[b_{Op}, f_{Op}]$ on the linear time axis ($b_{Op} < f_{Op}$). The precedence relation (denoted by '$\rightarrow$') is a relation that relates operations of a possible execution, $Op_1 \rightarrow Op_2$ means that Op_1 ends before Op_2 starts. The precedence relation is a strict partial order. Operations incomparable under $\rightarrow$ are called* overlapping. *The overlapping relation is denoted by $\|$ and is commutative, i.e. $Op_1 \parallel Op_2$ and $Op_2 \parallel Op_1$. The precedence relation is extended to relate sub-operations of operations. Consequently, if $Op_1 \rightarrow Op_2$, then for any sub-operations op_1 and op_2 of Op_1 and Op_2, respectively, it holds that $op_1 \rightarrow op_2$. We also define the direct precedence relation $\rightarrow_d$, such that if $Op_1 \rightarrow_d Op_2$, then $Op_1 \rightarrow Op_2$ and moreover there exists no operation Op_3 such that $Op_1 \rightarrow Op_3 \rightarrow Op_2$.*

Definition 3 *In order for an implementation of a shared concurrent data object to be linearizable [50], for every concurrent execution there should exist an equivalent (in the sense of the effect) and valid (i.e. it should respect the semantics of the shared data object) sequential execution that respects the partial order of the operations in the concurrent execution.*

Next we are going to study the possible concurrent executions of our implementation. First we need to define the interpretation of the abstract internal state of our implementation.

Definition 4 *The pair $\langle p, v \rangle$ is present ($\langle p, v \rangle \in L$) in the abstract internal state L of our implementation, when there is a next pointer from a present node on the lowest level of the skip list that points to a node that contains the pair $\langle p, v \rangle$, and this node is not marked as deleted with the mark on the value.*

Lemma 2 *The definition of the abstract internal state for our implementation is consistent with all concurrent operations examining the state of the priority queue.*

Proof: As the next and value pointers are changed using the CAS operation, we are sure that all threads see the same state of the skip list, and therefore all changes of the abstract internal state seems to be atomic. □

Definition 5 *The decision point of an operation is defined as the atomic statement where the result of the operation is finitely decided, i.e. independent of the result of any sub-operations after the decision point, the operation will have the same result. We define the verification point of an operation to be the atomic statement where a sub-state of the priority queue is read, and this sub-state is verified to have certain properties before the passing of the decision point. We also define the state-change point as the atomic statement where the operation changes the abstract internal state of the priority queue after it has passed the corresponding decision point.*

We will now use these points in order to show the existence and location in execution history of a point where the concurrent operation can be viewed as it occured atomically, i.e. the *linearizability point*.

Lemma 3 *An Insert operation which succeeds ($I(\langle p, v \rangle) = true$), takes effect atomically at one statement.*

Proof: The decision point for an *Insert* operation which succeeds ($I(\langle p, v \rangle) = true$), is when the CAS sub-operation in line I27 (see Figure 5.7) succeeds, all following CAS sub-operations will eventually succeed, and the *Insert* operation will finally return *true*. The state of the list (L_{t_1}) directly before the passing of the decision point must have been $\langle p, _ \rangle \notin L_{t_1}$, otherwise the CAS would have failed. The state of the list directly after passing the decision point will be $\langle p, v \rangle \in L_{t_2}$. Consequently, the linearizability point will be the CAS sub-operation in line I27. □

Lemma 4 *An Insert operation which updates ($I(\langle p, v\rangle) = true_2$), takes effect atomically at one statement.*

Proof: The decision point for an *Insert* operation which updates ($I(\langle p, v\rangle) = true_2$), is when the CAS will succeed in line I14. The state of the list (L_{t_1}) directly before passing the decision point must have been $\langle p, _\rangle \in L_{t_1}$, otherwise the CAS would have failed. The state of the list directly after passing the decision point will be $\langle p, v\rangle \in L_{t_2}$. Consequently, the linearizability point will be the CAS sub-operation in line I14. □

Lemma 5 *A DeleteMin operations which fails ($DM() = \perp$), takes effect atomically at one statement.*

Proof: The decision point for an *DeleteMin* operations which fails ($DM() = \perp$), is when the hidden read sub-operation of the $ReadNext$ sub-operation in line D5 successfully reads the next pointer on lowest level that equals the tail node. The state of the list (L_t) directly before the passing of the decision point must have been $L_t = \emptyset$. Consequently, the linearizability point will be the hidden read sub-operation of the next pointer in line D5. □

Lemma 6 *A DeleteMin operation which succeeds ($DM() = \langle p_1, v_1\rangle$ where $\langle p_1, v_1\rangle = \{\langle \min p, v\rangle | \langle p, v\rangle \in L_{t_1}\}$), takes effect atomically at one statement.*

Proof: The decision point for an *DeleteMin* operation which succeeds is when the CAS sub-operation in line D15 (see Figure 5.8) succeeds. The state of the list (L_t) directly before passing of the decision point must have been $\langle p_1, v_1\rangle \in L_{t_3}$, otherwise the CAS would have failed. The state of the list directly after passing the CAS sub-opertion in line D15 (i.e. the state-change point) will be $\langle p, _\rangle \notin L_{t_4}$. The state of the list at the time of the read sub-operation of the next pointer in D11 (i.e. the verification point) must have been $\langle p_1, v_1\rangle = \{\langle \min p, v\rangle | \langle p, v\rangle \in L_{t_1}\}$. Unfortunately this does not completely match the semantic definition of the operation.

However, none of the other concurrent operations linearizability points is dependent on the to-be-deleted node's state as marked or not marked during the time interval from the verification point to the state-change point. Clearly, the linearizability points of Lemma 3 is independent during this time interval, as the to-be-deleted node must be different from the corresponding $\langle p, v\rangle$ term, as Lemma 3 views the to-be-deleted node as present during

the time interval. The linearizability point of Lemma 4 is independent during the time interval, as the to-be-deleted node must be different from the corresponding $\langle p, v\rangle$ term, otherwise the CAS sub-operation in line D15 of this operation would have failed. The linearizability point of Lemma 5 is independent, as that linearizability point depends on the head node's next pointer pointing to the tail node or not. Finally, the linearizability point of this lemma is independent, as the to-be-deleted node would be different from the corresponding $\langle p_1, v_1\rangle$ term, otherwise the CAS sub-operation in line D15 of this operation invocation would have failed.

Therefore all together, we could safely interpret the to-be-deleted node to be not present already directly after passing the verification point ($\langle p, _\rangle \notin L_{t_2}$). Consequently, the linearizability point will be the read sub-operation of the next pointer in line D11. □

Definition 6 *We define the relation $\Rightarrow$ as the total order and the relation $\Rightarrow_d$ as the direct total order between all operations in the concurrent execution. In the following formulas, $E_1 \Longrightarrow E_2$ means that if E_1 holds then E_2 holds as well, and $\oplus$ stands for exclusive or (i.e. $a \oplus b$ means $(a \vee b) \wedge \neg(a \wedge b)$):*

$$\mathbf{Op_1} \rightarrow_\mathbf{d} \mathbf{Op_2},\ \nexists \mathbf{Op_3}.\mathbf{Op_1} \Rightarrow_\mathbf{d} \mathbf{Op_3}, \\ \nexists \mathbf{Op_4}.\mathbf{Op_4} \Rightarrow_\mathbf{d} \mathbf{Op_2} \implies \mathbf{Op_1} \Rightarrow_\mathbf{d} \mathbf{Op_2} \tag{5.5}$$

$$\mathbf{Op_1} \parallel \mathbf{Op_2} \implies \mathbf{Op_1} \Rightarrow_\mathbf{d} \mathbf{Op_2} \ \oplus\ \mathbf{Op_2} \Rightarrow_\mathbf{d} \mathbf{Op_1} \tag{5.6}$$

$$\mathbf{Op_1} \Rightarrow_\mathbf{d} \mathbf{Op_2} \implies \mathbf{Op_1} \Rightarrow \mathbf{Op_2} \tag{5.7}$$

$$\mathbf{Op_1} \Rightarrow \mathbf{Op_2}, \mathbf{Op_2} \Rightarrow \mathbf{Op_3} \implies \mathbf{Op_1} \Rightarrow \mathbf{Op_3} \tag{5.8}$$

Lemma 7 *The operations that are directly totally ordered using formula 5.5, form an equivalent valid sequential execution.*

Proof: If the operations are assigned their direct total order ($Op_1 \Rightarrow_d Op_2$) by formula 5.5 then also the linearizability points of Op_1 are executed before the respective points of Op_2. In this case the operations semantics behave the same as in the sequential case, and therefore all possible executions will then be equivalent to one of the possible sequential executions. □

Lemma 8 *The operations that are directly totally ordered using formula 5.6 can be ordered unique and consistent, and form an equivalent valid sequential execution.*

Proof: Assume we order the overlapping operations according to their linearizability points. As the state before as well as after the linearizability points is identical to the corresponding state defined in the semantics of the respective sequential operations in formulas 5.1 to 5.4, we can view the operations as occurring at the linearizability point. As the linearizability points consist of atomic operations and are therefore ordered in time, no linearizability point can occur at the very same time as any other linearizability point, therefore giving a unique and consistent ordering of the overlapping operations. □

Lemma 9 *With respect to the retries caused by synchronization, one operation will always do progress regardless of the actions by the other concurrent operations.*

Proof: We now examine the possible execution paths of our implementation. There are several potentially unbounded loops that can delay the termination of the operations. We call these loops retry-loops. If we omit the conditions that are because of the operations semantics (i.e. searching for the correct position etc.), the retry-loops take place when sub-operations detect that a shared variable has changed value. This is detected either by a subsequent read sub-operation or a failed CAS. These shared variables are only changed concurrently by other CAS sub-operations. According to the definition of CAS, for any number of concurrent CAS sub-operations, exactly one will succeed. This means that for any subsequent retry, there must be one CAS that succeeded. As this succeeding CAS will cause its retry loop to exit, and our implementation does not contain any cyclic dependencies between retry-loops that exit with CAS, this means that the corresponding *Insert* or *DeleteMin* operation will progress. Consequently, independent of any number of concurrent operations, one operation will always progress. □

Theorem 2 *The algorithm implements a lock-free and linearizable priority queue.*

Proof: Following from Lemmas 7 and 8 and using the direct total order we can create an identical (with the same semantics) sequential execution that preserves the partial order of the operations in a concurrent execution. Following from Definition 3, the implementation is therefore linearizable. As the semantics of the operations are basically the same as in the skip list [91], we could use the corresponding proof of termination. This together

with Lemma 9 and that the state is only changed at one atomic statement (Lemmas 2,3,4,6,5), gives that our implementation is lock-free. □

5.5 Experiments

We have performed experiments using 1 up to 28 threads on three different platforms, each with different levels of real concurrency, architecture and operating system. Besides our implementation, we also performed the same experiments with four lock-based implementations. These are; 1) a single-lock protected skip list, 2) the implementation using multiple locks and skip lists by Lotan and Shavit [72], 3) the heap-based implementation using multiple locks by Hunt *et al.* [54], and 4) the tree-based implementation by Jones [60]. As Lotan and Shavit implements the real-time properties as presented in Section 5.6 but Hunt et al. does not, we used both the ordinary as well the real-time version of our implementation.

The key values of the inserted nodes are randomly chosen between 0 and $1000000 * n$, where n is the number of threads. Each experiment is repeated 50 times, and an average execution time for each experiment is estimated. Exactly the same sequential operations are performed for all different implementations compared. A clean-cache operation is performed just before each sub-experiment. All implementations are written in C and compiled with the highest optimization level, except from the atomic primitives, which are written in assembler.

5.5.1 Low or Medium Concurrency

To get a highly pre-emptive environment, we performed our experiments on a Compaq dual-processor Pentium II 450 MHz PC running Linux. A set of experiments was also performed on a Sun Ultra 880 with 6 processors running Solaris 9. In our experiments each concurrent thread performs 10000 sequential operations, randomly chosen with a distribution of 50% *Insert* operations versus 50% *DeleteMin* operations. The dictionaries are initialized with 100 or 1000 nodes before the start of the experiments. The implementation by Jones uses system semaphores for the mutual exclusion. All other lock-based implementations were evaluated using simple spin-locks (based on the TAS atomic primitive), and as well using system semaphores. The results from these experiments are shown in Figures 5.11 and 5.12 for the spinlock-based implementations and in Figures 5.13 5.14 for the semaphore-based, both together with the new lock-free implementation. The average

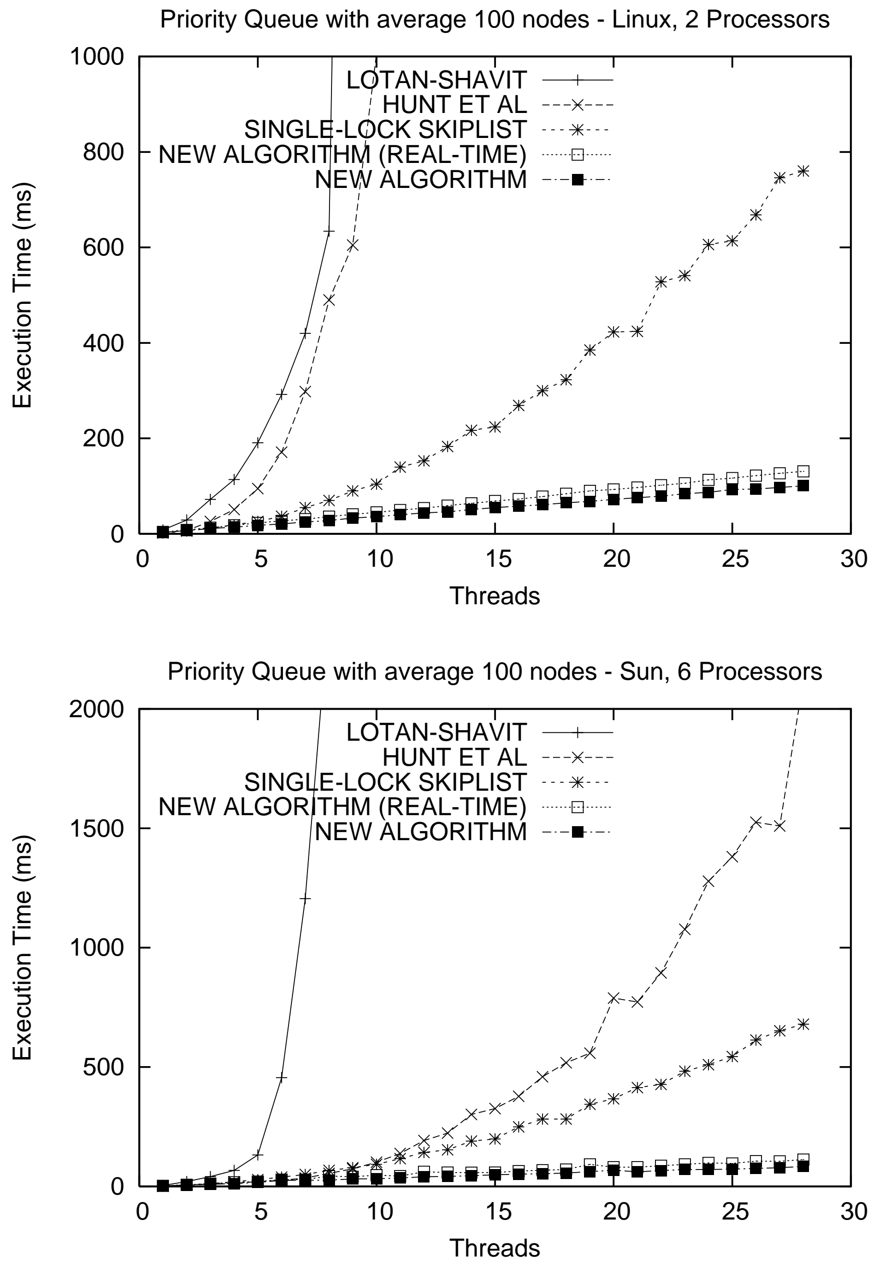


Figure 5.11: Experiment with priority queues and high contention, with initial 100 nodes, using spinlocks for mutual exclusion.

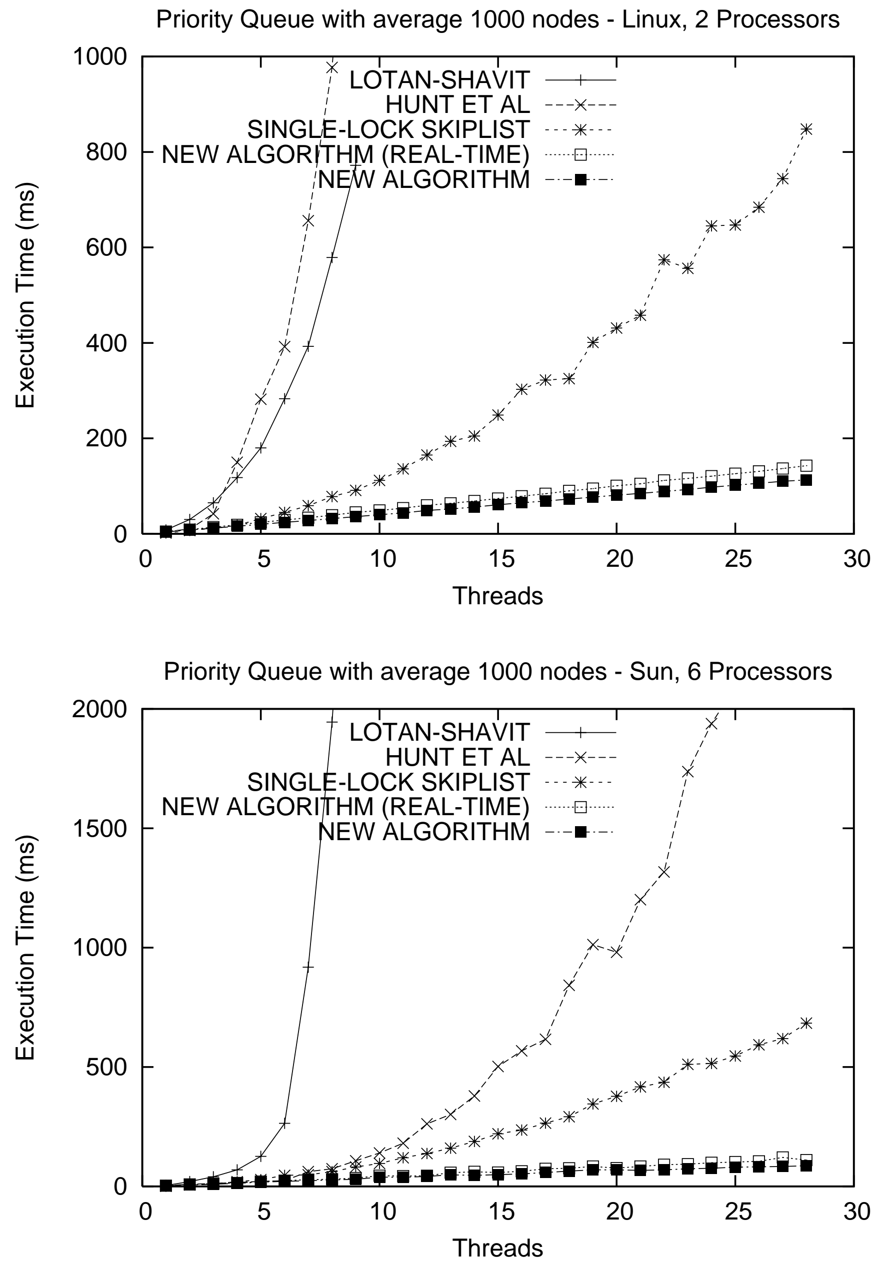


Figure 5.12: Experiment with priority queues and high contention, with initial 1000 nodes, using spinlocks for mutual exclusion.

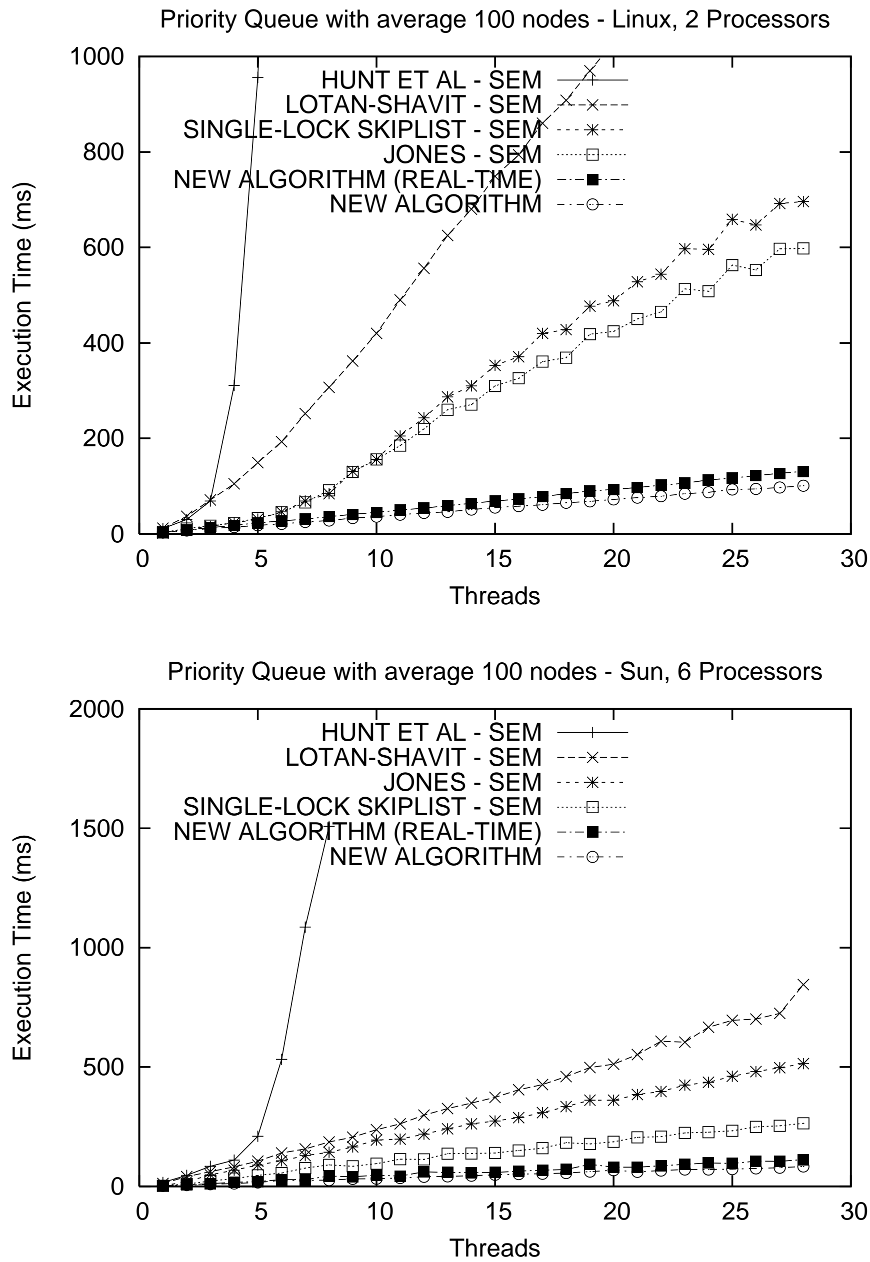


Figure 5.13: Experiment with priority queues and high contention, with initial 100 nodes, using semaphores for mutual exclusion.

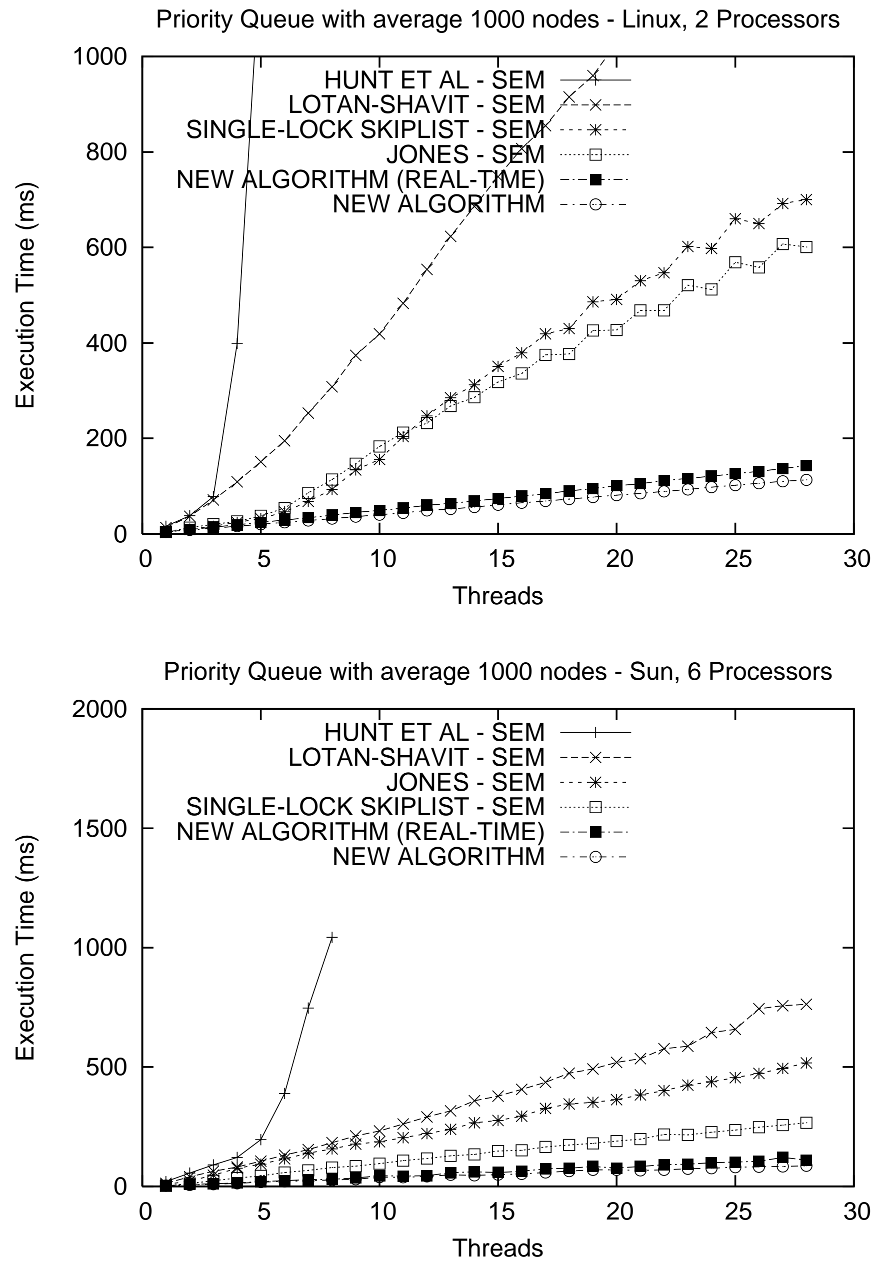


Figure 5.14: Experiment with priority queues and high contention, with initial 1000 nodes, using semaphores for mutual exclusion.

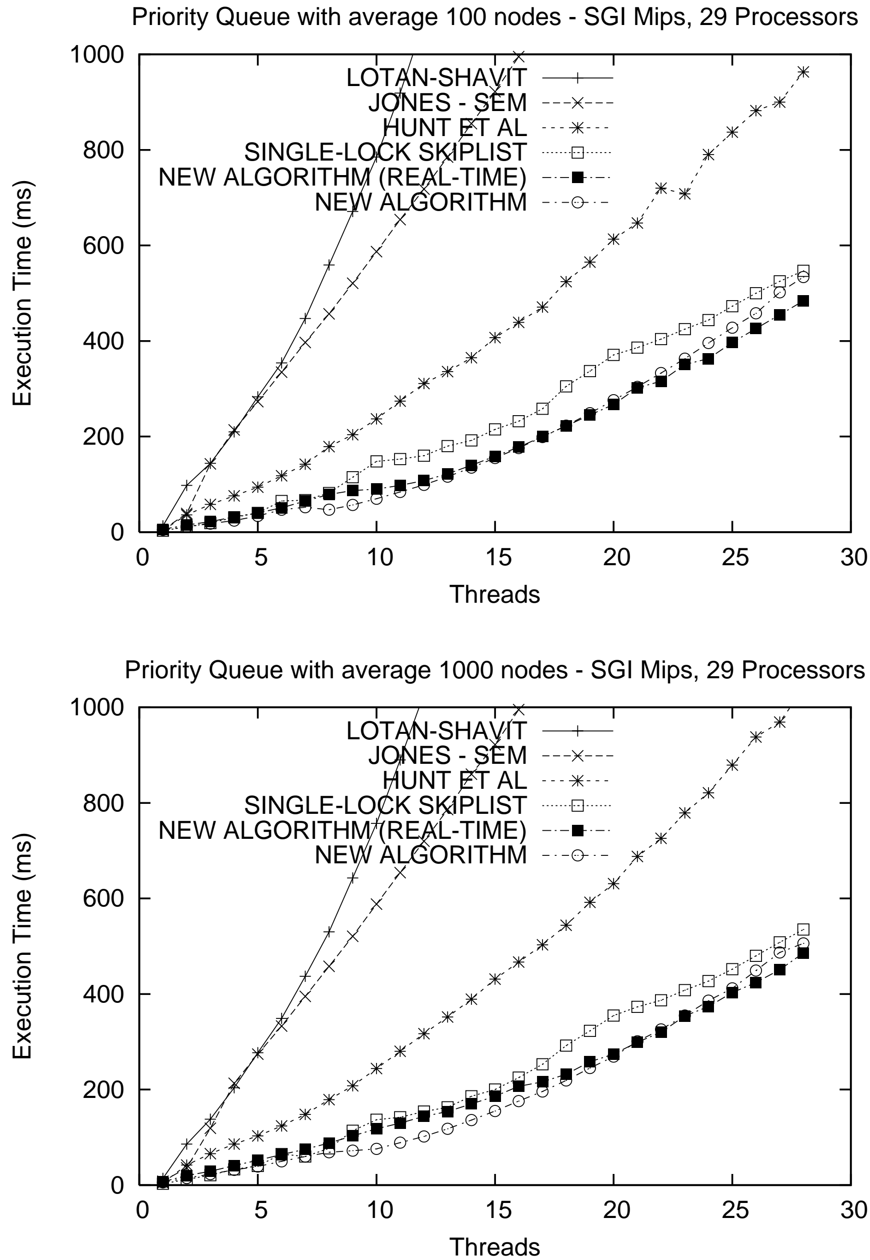


Figure 5.15: Experiment with priority queues and high contention, with initial 100 or 1000 nodes

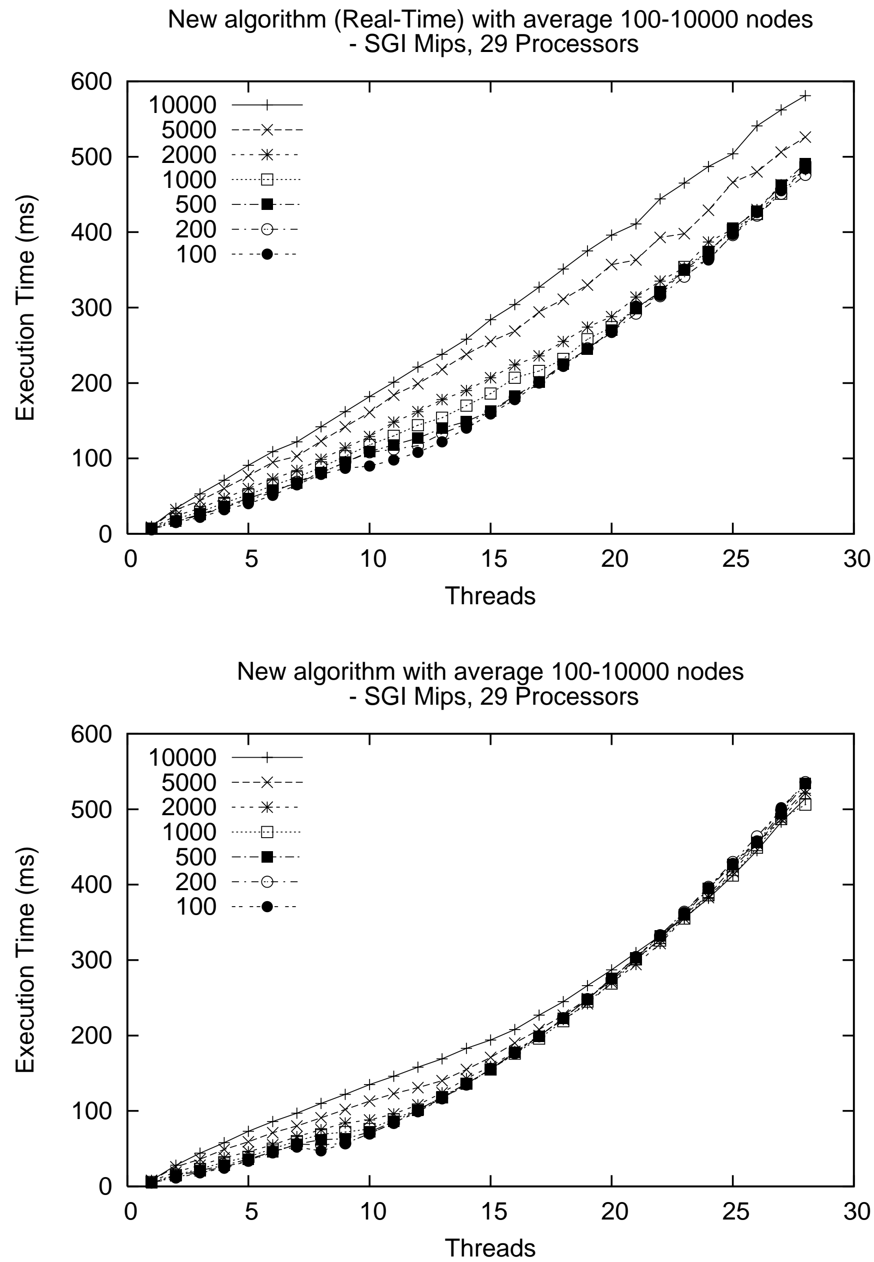


Figure 5.16: Experiment with priority queues and high contention, running with average 100-10000 nodes

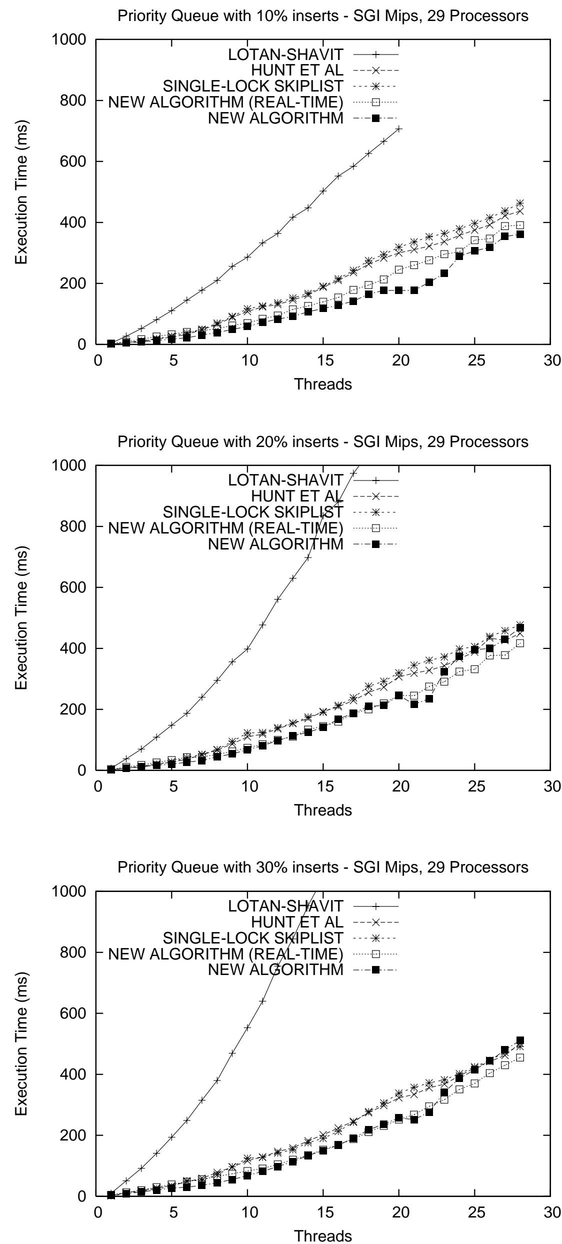


Figure 5.17: Experiment with priority queues and high contention, varying percentage of insert operations, with initial 1000 (for 10-40 %) or 0 (for 60-90 %) nodes. Part 1(3).

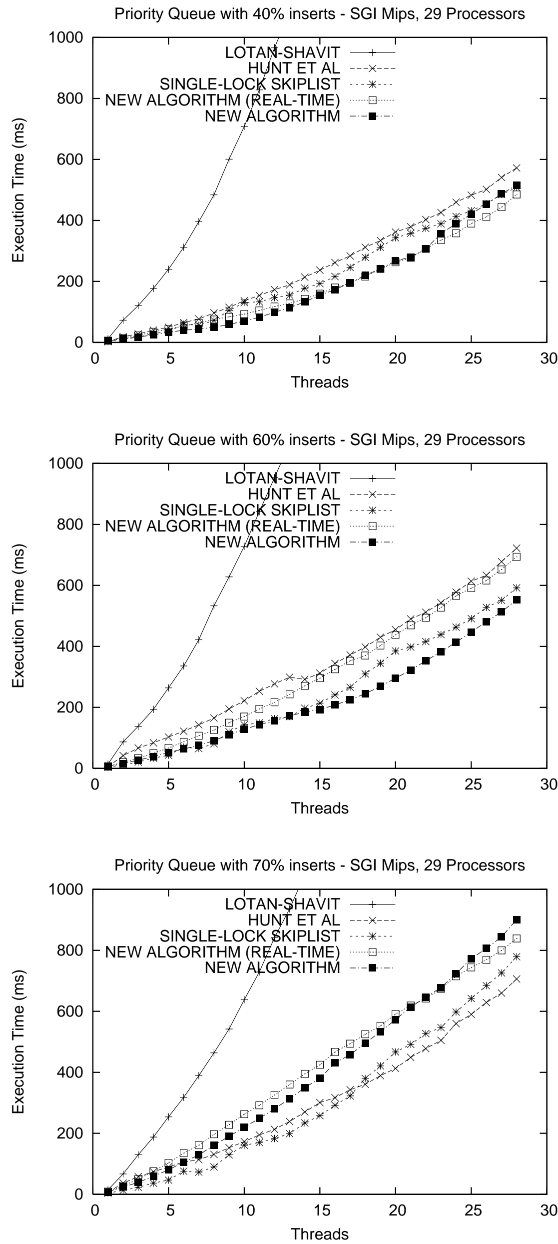


Figure 5.18: Experiment with priority queues and high contention, varying percentage of insert operations, with initial 1000 (for 10-40 %) or 0 (for 60-90 %) nodes. Part 2(3).

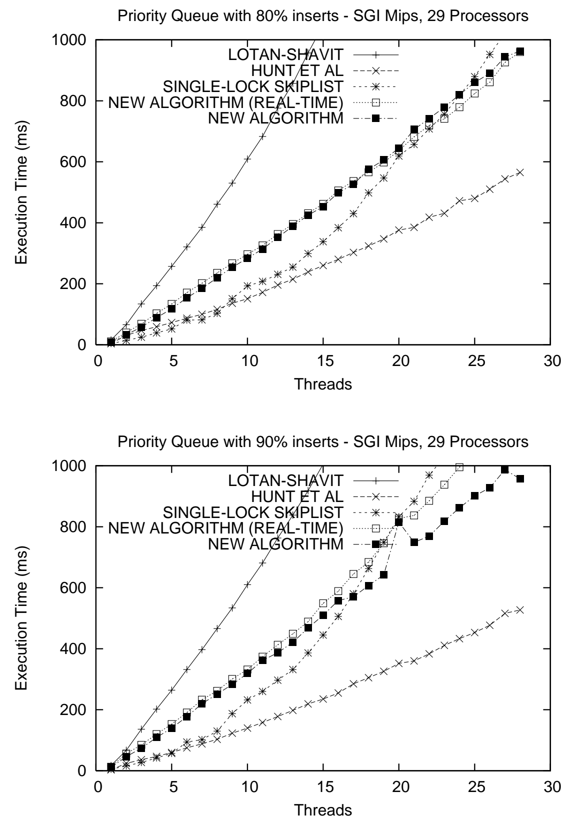


Figure 5.19: Experiment with priority queues and high contention, varying percentage of insert operations, with initial 1000 (for 10-40 %) or 0 (for 60-90 %) nodes. Part 3(3).

execution time is drawn as a function of the number of threads.

5.5.2 Full concurrency

In order to evaluate our algorithm with full concurrency we also used a SGI Origin 2000 250 MHz system running Irix 6.5 with 29 processors. With the exception for the implementation by Jones which can only use semaphores, all lock-based implementations were only evaluated using simple spin-locks, as those are always more efficient than semaphores on fully concurrent systems. In the first experiments each concurrent thread performs 10000 sequential operations, randomly chosen with a distribution of 50% *Insert* operations versus 50% *DeleteMin* operations, operating on dictionaries that are initialized with 100 or 1000 nodes. For the two implementations of our algorithm we also ran experiments, where the dictionaries were initialized with 100, 200, 500, 1000, 2000, 5000 or 10000 nodes. The results from these experiments are shown in Figure 5.15. The average execution time is drawn as a function of the number of threads.

We have also performed experiments with altered distribution of *Insert* operations, varied between 10% upto 90%. For the distribution of 10-40% inserts the dictionary was initialized with 1000 nodes, and for 60-90% inserts the dictionary was initialized as empty. The results from these experiments are shown in Figure 5.17. The average execution time is drawn as a function of the number of threads.

5.5.3 Results

From the results we can conclude that all of the implementations scale similarly with respect to the average size of the priority queue. The implementation by Lotan and Shavit [72] scales linearly with respect to increasing number of threads when having full concurrency, although when exposed to pre-emption its performance is highly dependent on the usage of semaphores, with simple spin-locks the performance decreases very rapidly. The implementation by Hunt *et al.* [54] shows better but similar behavior for full concurrency. However, it is instead severly punished by using semaphores on systems with pre-emption, because of its built-in contention management mechanism that was designed for simpler locks. The single-lock protected skip list performs better than both Lotan and Shavit and Hunt et al in all scenarios, using either semaphores or simple spin-locks. The implementation by Jones [60] shows a performance slightly worse or better than the single-lock skip list when exposed to pre-emption, though for full concurrency the

performance decreases almost to the implementation by Lotan and Shavit, because of the high overhead connected with semaphores.

Our lock-free implementation scales best compared to all other involved implementations except the single-lock skip list, having best performance already with 3 threads, independently if the system is fully concurrent or involves pre-emptions. Compared to the single-lock skip list, our lock-free implementation performs closely or slightly better for full concurrency, though having rapidly better performance with increasing level of pre-emption. Clearly, our lock-free implementation retains the logarithmic time complexity with respect to the size. However, the normal and real-time versions of our implementation shows slightly different behavior, due to several competing time factors. The factors are among many i) the real-time version can mark nodes as deleted in parallel, ii) with larger sizes the accesses get more distributed with resulting lower contention, iii) the logarithmic time complexity from the nature of the skip list, and iv) the overhead connected with the usage of time-stamps in the real-time version.

Even though the implementation by Lotan and Shavit is also based on a skip list with a similar approach as our algorithm, it performs significantly slower, especially on systems with pre-emption. This performance penalty is because of several reasons i) there are very many locks involved, each which must be implemented using an atomic primitive having almost the same contention as a CAS operation, ii) the competition for the locks is comparingly high and increases rapidly with the level in the skip list, with resulting conflicts and waiting on the execution of the blocking operations critical sections (which if possibly pre-empted gets very long), iii) the high overhead caused by the garbage collection scheme. The algorithms by Hunt et al and Jones are also penalized by the drawbacks of many locks, with increasing number of conflicts and blockings with higher level in the tree or heap structure.

For the experiments with altered distribution of *Insert* operations and full concurrency, the hierarchy among the involved implementations are quite different. For 10-40% *Insert* operations the implementation by Hunt et al shows similar performance as the single-lock skip list and our lock-free implementations, and for 60-90% it shows slightly or significantly better performance. However, neither of the altered scenarios is practically reasonable as long-term scenarios as the priority queues then will have either ever-increasing or almost average zero sizes. An ever-increasing priority queue will be highly impractical in the concern of memory, and for an average zero sized priority queue it would suffice with much simpler data structures than skip lists or trees.

5.6 Extended Algorithm

When we have concurrent *Insert* and *DeleteMin* operations we might want to have certain real-time properties of the semantics of the *DeleteMin* operation, as expressed in [72]. The *DeleteMin* operation should only return items that have been inserted by an *Insert* operation that finished before the *DeleteMin* operation started. To ensure this we are adding timestamps to each node. When the node is fully inserted its timestamp is set to the current time. Whenever the *DeleteMin* operation is invoked it first checks the current time, and then discards all nodes that have a timestamp that is after this time. In the code of the implementation (see Figures 5.6,5.7,5.8 and 5.10), the additional statements that involve timestamps are marked within the "$\langle$" and "$\rangle$" symbols. The function $getNextTimeStamp$, see Figure 5.23, creates a new timestamp. The function $compareTimeStamp$, see Figure 5.23, compares if the first timestamp is less, equal or higher than the second one and returns the values -1,0 or 1, respectively.

As we are only using the timestamps for relative comparisons, we do not need real absolute time, only that the timestamps are monotonically increasing. Therefore we can implement the time functionality with a shared counter, the synchronization of the counter is handled using CAS. However, the shared counter usually has a limited size (i.e. 32 bits) and will eventually overflow. Therefore the values of the timestamps have to be recycled. We will do this by exploiting information that are available in real-time systems, with a similar approach as in [103].

We assume that we have n periodic tasks in the system, indexed $\tau_1 ... \tau_n$. For each task τ_i we will use the standard notations T_i, C_i, R_i and D_i to denote the period (i.e. min period for sporadic tasks), worst case execution time, worst case response time and deadline, respectively. The deadline of a task is less or equal to its period.

For a system to be safe, no task should miss its deadlines, i.e. $\forall i \mid R_i \leq D_i$.

For a system scheduled with fixed priority, the response time for a task in the initial system can be calculated using the standard response time analysis techniques [17]. If we with B_i denote the blocking time (the time the task can be delayed by lower priority tasks) and with $hp(i)$ denote the set of tasks with higher priority than task τ_i, the response time R_i for task τ_i can be formulated as:

$$R_i = C_i + B_i + \sum_{j \in hp(i)} \left\lceil \frac{R_i}{T_j} \right\rceil C_j \tag{5.9}$$

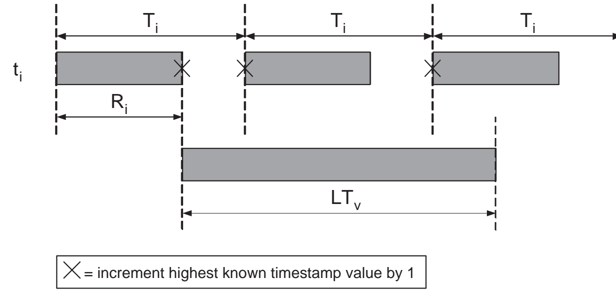


Figure 5.20: Maximum timestamp increasement estimation - worst case scenario

The summand in the above formula gives the time that task τ_i may be delayed by higher priority tasks. For systems scheduled with dynamic priorities, there are other ways to calculate the response times [17].

Now we examine some properties of the timestamps that can exist in the system. Assume that all tasks call either the *Insert* or *DeleteMin* operation only once per iteration. As each call to $getNextTimeStamp$ will introduce a new timestamp in the system, we can assume that every task invocation will introduce one new timestamp. This new timestamp has a value that is the previously highest known value plus one. We assume that the tasks always execute within their response times R with arbitrary many interruptions, and that the execution time C is comparably small. This means that the increment of highest timestamp respective the write to a node with the current timestamp can occur anytime within the interval for the response time. The maximum time for an *Insert* operation to finish is the same as the response time R_i for its task τ_i. The minimum time between two index increments is when the first increment is executed at the end of the first interval and the next increment is executed at the very beginning of the second interval, i.e. $T_i - R_i$. The minimum time between the subsequent increments will then be the period T_i. If we denote with LT_v the maximum life-time that the timestamp with value v exists in the system, the worst case scenario in respect of growth of timestamps is shown in Figure 5.20.

The formula for estimating the maximum difference in value between two existing timestamps in any execution becomes as follows:

$$MaxTag = \sum_{i=0}^{n} \left(\left\lceil \frac{\max_{v \in \{0..\infty\}} LT_v}{T_i} \right\rceil + 1 \right) \tag{5.10}$$

Now we have to bound the value of $\max_{v\in\{0..\infty\}} LT_v$. When comparing timestamps, the absolute value of these are not important, only the relative values. Our method is that we continuously traverse the nodes and replace outdated timestamps with a newer timestamp that has the same comparison result. We traverse and check the nodes at the rate of one step to the right for every invocation of an *Insert* or *DeleteMin* operation. With outdated timestamps we define timestamps that are older (i.e. lower) than any timestamp value that is in use by any running *DeleteMin* operation. We denote with $AncientVal$ the maximum difference that we allow between the highest known timestamp value and the timestamp value of a node, before we call this timestamp outdated.

$$AncientVal = \sum_{i=0}^{n} \left\lceil \frac{\max_j R_j}{T_i} \right\rceil \tag{5.11}$$

If we denote with t_{ancient} the maximum time it takes for a timestamp value to be outdated counted from its first occurrence in the system, we get the following relation:

$$AncientVal = \sum_{i=0}^{n} \left\lfloor \frac{t_{\text{ancient}}}{T_i} \right\rfloor > \sum_{i=0}^{n} \left(\frac{t_{\text{ancient}}}{T_i} \right) - n \tag{5.12}$$

$$t_{\text{ancient}} < \frac{AncientVal + n}{\sum_{i=0}^{n} \frac{1}{T_i}} \tag{5.13}$$

Now we denote with t_{traverse} the maximum time it takes to traverse through the whole list from one position and getting back, assuming the list has the maximum size N.

$$N = \sum_{i=0}^{n} \left\lfloor \frac{t_{\text{traverse}}}{T_i} \right\rfloor > \sum_{i=0}^{n} \left(\frac{t_{\text{traverse}}}{T_i} \right) - n \tag{5.14}$$

$$t_{\text{traverse}} < \frac{N + n}{\sum_{i=0}^{n} \frac{1}{T_i}} \tag{5.15}$$

The worst-case scenario is that directly after the timestamp of one node gets traversed, it gets outdated. Therefore we get:

$$\max_{v\in\{0..\infty\}} LT_v = t_{\text{ancient}} + t_{\text{traverse}} \tag{5.16}$$

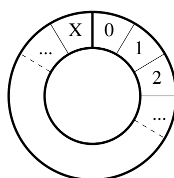


Figure 5.21: Timestamp value recycling

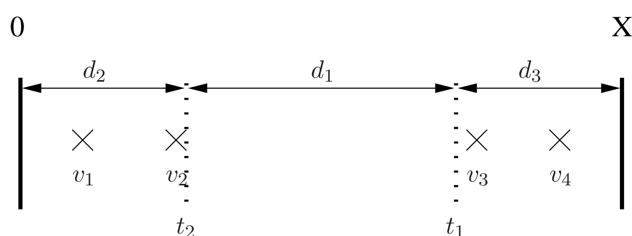


Figure 5.22: Deciding the relative order between reused timestamps

Putting all together we get:

$$MaxTag < \sum_{i=0}^{n} \left(\left\lceil \frac{N + 2n + \sum_{k=0}^{n} \left\lceil \frac{\max_j R_j}{T_k} \right\rceil}{T_i \sum_{l=0}^{n} \frac{1}{T_l}} \right\rceil + 1 \right) \tag{5.17}$$

The above equation gives us a bound on the length of the "window" of active timestamps for any task in any possible execution. In the unbounded construction the tasks, by producing larger timestamps every time they slide this window on the $[0, \ldots, \infty]$ axis, always to the right. The approach now is instead of sliding this window on the set $[0, \ldots, \infty]$ from left to right, to cyclically slide it on a $[0, \ldots, X]$ set of consecutive natural numbers, see figure 5.21. Now at the same time we have to give a way to the tasks to identify the order of the different timestamps because the order of the physical numbers is not enough since we are re-using timestamps. The idea is to use the bound that we have calculated for the span of different active timestamps. Let us then take a task that has observed v_i as the lowest timestamp at some invocation τ. When this task runs again as τ', it can conclude that the active timestamps are going to be between v_i and $(v_i + MaxTag) \mod X$. On the other hand we should make sure that in this

```
// Global variables
timeCurrent: integer
checked: pointer to Node
// Local variables
time,newtime,safeTime: integer
current,node,next: pointer to Node

function compareTimeStamp(time1:integer,
 time2:integer):integer
C1    if time1=time2 then return 0;
C2    if time2=MAX_TIME then return -1;
C3    if time1>time2 and (time1-time2)≤MAX_TAG or
       time1<time2 and (time1-time2+MAX_TIME)
       ≤MAX_TAG then return 1;
C4    else return -1;

function getNextTimeStamp():integer
G1    repeat
G2        time:=timeCurrent;
G3        if (time+1)≠MAX_TIME then newtime:=time+1;
G4        else newtime:=0;
G5    until CAS(&timeCurrent,time,newtime);
G6    return newtime;

procedure TraverseTimeStamps()
T1    safeTime:=timeCurrent;
T2    if safeTime≥ANCIENT_VAL then
T3        safeTime:=safeTime-ANCIENT_VAL;
T4    else safeTime:=safeTime+MAX_TIME-ANCIENT_VAL;
T5    while true do
T6        node:=READ_NODE(checked);
T7        current:=node;
T8        next:=ReadNext(&node,0);
T9        RELEASE_NODE(node);
T10       if compareTimeStamp(safeTime,next.timeInsert)>0 then
T11           next.timeInsert:=safeTime;
T12       if CAS(&checked,current,next) then
T13           RELEASE_NODE(current);
T14           break;
T15       RELEASE_NODE(next);
```

Figure 5.23: Creation, comparison, traversing and updating of bounded timestamps.

interval $[v_i, \ldots, (v_i + MaxTag) \mod X]$ there are no old timestamps. By looking closer to equation 5.10 we can conclude that all the other tasks have written values to their registers with timestamps that are at most $MaxTag$ less than v_i at the time that τ wrote the value v_i. Consequently if we use an interval that has double the size of $MaxTag$, τ' can conclude that old timestamps are all on the interval $[(v_i - MaxTag) \mod X, \ldots, v_i]$.

Therefore we can use a timestamp field with double the size of the maximum possible value of the timestamp.

$TagFieldSize = MaxTag * 2$

$TagFieldBits = \lceil \log_2 TagFieldSize \rceil$

In this way τ' will be able to identify that v_1, v_2, v_3, v_4 (see figure 5.22) are all new values if $d_2 + d_3 < MaxTag$ and can also conclude that:

$v_3 < v_4 < v_1 < v_2$

The mechanism that will generate new timestamps in a cyclical order and also compare timestamps is presented in Figure 5.23 together with the code for traversing the nodes. Note that the extra properties of the priority queue that are achieved by using timestamps are not complete with respect to the *Insert* operations that finishes with an update. These update operations will behave the same as for the standard version of the implementation.

Besides from real-time systems, the presented technique can also be useful in non real-time systems as well. For example, consider a system of $n = 10$ threads, where the minimum time between two invocations would be $T = 10$ ns, and the maximum response time $R = 1000000000$ ns (i.e. after 1 s we would expect the thread to have crashed). Assuming a maximum size of the list $N = 10000$, we will have a maximum timestamp difference $MaxTag < 1000010030$, thus needing 31 bits. Given that most systems have 32-bit integers and that many modern systems handle 64 bits as well, it implies that this technique is practical for also non real-time systems.

5.7 Related Work with Skip Lists

This paper describes the first[5] lock-free algorithm of a skip list data structure. Very similar constructions have appeared in the literature afterwards, by Fraser [32], Fomitchev [30] and Fomitchev and Ruppert [31]. As both Fraser's and Fomitchev's constructions appeared quite some time later in

[5] Our results were submitted for reviewing in October 2002 and published as a technical report [105] in January 2003. It was officially published in April 2003 [106], receiving a best paper award, and an extended version was also published in March 2004 [110]. Very similar constructions have appeared in the literature afterwards, by Fraser in February 2004 [32], Fomitchev in November 2003 [30] and Fomitchev and Ruppert in July 2004 [31]

the literature than ours, it was not possible to compare them in our original publications. However, we have recently studied the other's approaches and have found some significant differences, although the main ideas are essentially the same. The differences are mainly related to performance issues:

- Compared to Fraser's approach, our skip list construction does not suffer from possible restarts of the full search phase from the head level when reaching a deleted node, as our nodes also contains a backlink pointer that is set at the time of deletion. This enables us to step one step backwards when reaching a deleted node, and to directly remove the deleted node. Both Fraser's and our construction uses arrays for remembering positions, though Fraser unnecessarily remembers also the successor on each level which could incur performance penalties through the garbage collector used.

- Compared to Fomitchev's and Fomitchev and Ruppert's approach, their construction does not use an array for remembering positions, which forces their construction to perform two full search phases when inserting or deleting nodes. In addition to have backlink pointers in order to be able to step back when reaching a deleted node, their construction also uses an extra pointer mark that is set (using an extra and expensive CAS operation) on the predecessor node in order to earlier notify concurrent operations of the helping duty. In our construction we only have one backlink pointer for all levels of a node, because of a performance trade-off between the usefulness for helping operations and the cost that keeping extra pointers could incur for the garbage collection.

5.8 Conclusions

We have presented a lock-free algorithmic implementation of a concurrent priority queue. The implementation is based on the sequential skip list data structure and builds on top of it to support concurrency and lock-freedom in an efficient and practical way. Compared to the previous attempts to use skip lists for building concurrent priority queues our algorithm is lock-free and avoids the performance penalties that come with the use of locks. Compared to the previous lock-free/wait-free concurrent priority queue algorithms, our algorithm inherits and carefully retains the basic design characteristic that makes skip lists practical: simplicity. Previous lock-free/wait-free algorithms did not perform well because of their complexity, furthermore

they were often based on atomic primitives that are not available in today's systems.

We compared our algorithm with some of the most efficient implementations of priority queues known. Experiments show that our implementation scales well, and with 3 threads or more our implementation outperforms the corresponding lock-based implementations, for all cases on both fully concurrent systems as well as with pre-emption.

We believe that our implementation is of highly practical interest for multi-threaded applications. We are currently incorporating it into the NOBLE [104] library.

Chapter 6

Scalable and Lock-Free Concurrent Dictionaries[1]

Håkan Sundell, Philippas Tsigas
Department of Computing Science
Chalmers Univ. of Technol. and Göteborg Univ.
412 96 Göteborg, Sweden
E-mail: {phs, tsigas}@cs.chalmers.se

Abstract

We present an efficient and practical lock-free implementation of a concurrent dictionary that is suitable for both fully concurrent (large multi-processor) systems as well as pre-emptive (multi-process) systems. Many algorithms for concurrent dictionaries are based on mutual exclusion. However, mutual exclusion causes blocking which has several drawbacks and degrades the system's overall performance. Non-blocking algorithms avoid blocking, and are either lock-free or wait-free. Our algorithm is based on the randomized sequential list structure called skip list, and implements the full set of operations on a dictionary that is suitable for practical settings. In our performance evaluation we compare our algorithm with the most efficient non-blocking implementation of dictionaries known. The experimental

[1]This is an extended and revised version of the paper with the same title that was presented at SAC 2004 [110] and published as a technical report [107].

results clearly show that our algorithm outperforms the other lock-free algorithm for dictionaries with realistic sizes, both on fully concurrent as well as pre-emptive systems.

6.1 Introduction

Dictionaries (also called sets) are fundamental data structures. From the operating system level to the user application level, they are frequently used as basic components.

Consequently, the design of efficient implementations of dictionaries is a research area that has been extensively researched. A dictionary supports five operations, the $Insert$, the $FindKey$, the $DeleteKey$, the $FindValue$ and the $DeleteValue$ operation. The abstract definition of a dictionary is a set of key-value pairs, where the key is an unique integer associated with a value. The $Insert$ operation inserts a new key-value pair into the dictionary and the $FindKey/DeleteKey$ operation finds/removes and returns the value of the key-value pair with the specified key that was in the dictionary. The $FindValue/DeleteValue$ operation finds/removes and returns the key of the key-value pair with the specified value that was in the dictionary.

To ensure consistency of a shared data object in a concurrent environment, the most common method is to use mutual exclusion, i.e. some form of locking. Mutual exclusion degrades the system's overall performance [100] as it causes blocking, i.e. other concurrent operations can not make any progress while the access to the shared resource is blocked by the lock. Using mutual exclusion can also cause deadlocks, priority inversion (which can be solved efficiently on uni-processors [95] with the cost of more difficult analysis, although not as efficient on multiprocessor systems [92]) and even starvation.

To address these problems, researchers have proposed non-blocking algorithms for shared data objects. Non-blocking methods do not involve mutual exclusion, and therefore do not suffer from the problems that blocking can cause. Non-blocking algorithms are either lock-free or wait-free. Lock-free implementations guarantee that regardless of the contention caused by concurrent operations and the interleaving of their sub-operations, always at least one operation will progress. However, there is a risk for starvation as the progress of other operations could cause one specific operation to never finish. This is although different from the type of starvation that could be caused by blocking, where a single operation could block every other operation forever, and cause starvation of the whole system. Wait-

free [45] algorithms are lock-free and moreover they avoid starvation as well, in a wait-free algorithm every operation is guaranteed to finish in a limited number of steps, regardless of the actions of the concurrent operations. Non-blocking algorithms have been shown to be of big practical importance [115, 118], and recently NOBLE, which is a non-blocking inter-process communication library, has been introduced [104].

There exist several algorithms and implementations of concurrent dictionaries. The majority of the algorithms are lock-based, constructed with either a single lock on top of a sequential algorithm, or specially constructed algorithms using multiple locks, where each lock protects a small part of the shared data structure. However, most lock-based algorithms [22] are based on the theoretical PRAM model which is shown to be unrealistic [16]. As the search complexity of a dictionary is significant, most algorithms are based on tree or heap structures as well as tree-like structures as the skip list [91]. Previous non-blocking dictionaries are though based on arrays or ordered lists as done by Valois [122]. The path using concurrent ordered lists has been improved by Harris [44], and lately [77] presented a significant improvement by using a new memory management method [78]. However, Valois [122] presented an incomplete idea of how to design a concurrent skip list.

One common problem with many algorithms for concurrent dictionaries is the lack of precise defined semantics of the operations. Previously known non-blocking dictionaries only implements a limited set of operations, disregarding the $FindValue$ and $DeleteValue$ operations. It is also seldom that the correctness with respect to concurrency is proved, using a strong property like linearizability [50].

In this paper we present a lock-free algorithm of a concurrent dictionary that is designed for efficient use in both pre-emptive as well as in fully concurrent environments. Inspired by the incomplete attempt by Valois [122], the algorithm is based on the randomized skip list [91] data structure. It is also implemented using common synchronization primitives that are available in modern systems. The algorithm is described in detail later in this paper, and the aspects concerning the underlying lock-free memory management are also presented. The precise semantics of the operations are defined and we give a proof that our implementation is lock-free and linearizable.

Concurrent dictionaries are often used as building blocks for concurrent hash tables, where each branch (or bucket) of the hash table is represented by a dictionary. In an optimal setting, the average size of each branch is comparably low, i.e. less than 10 nodes, as in [77]. However, in practical

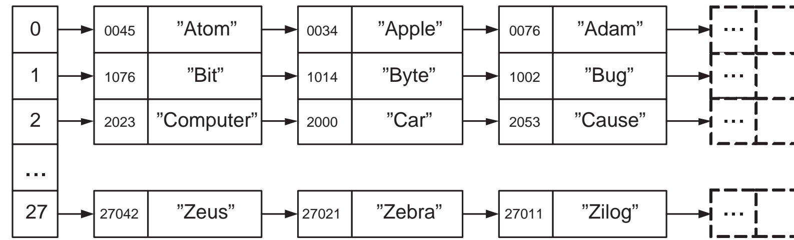


Figure 6.1: Example of a Hash Table with Dictionaries as branches.

settings the average size of each branch can vary significantly. For example, a hash table can be used to represent the words of a book, where each branch contains the words that begin with a certain letter, as in Figure 6.1. Therefore it is not unreasonable to expect dictionaries with sizes of several thousands nodes.

We have performed experiments that compare the performance of our algorithm with one of the most efficient implementations of non-blocking dictionaries known [77]. As the previous algorithm did not implement the full set of operations of our dictionary, we also performed experiments with the full set of operations, compared with a simple lock-based skip list implementation. Experiments were performed on three different platforms, consisting of a multiprocessor system using different operating systems and equipped with either 2 or 64 processors. Our results show that our algorithm outperforms the other lock-free implementation with realistic sizes and number of threads, in both highly pre-emptive as well as in fully concurrent environments.

The rest of the paper is organized as follows. In Section 6.2 we define the properties of the system that our implementation is aimed for. The actual algorithm is described in Section 6.3. In Section 6.4 we define the precise semantics for the operations on our implementations, as well showing correctness by proving the lock-free and linearizability property. The experimental evaluation that shows superior performance for our implementation is presented in Section 6.5. In Section 6.6 we discuss related work with skip lists that have appeared in the literature after our first publications. We conclude the paper with Section 6.7.

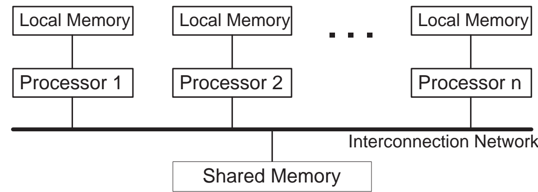


Figure 6.2: Shared memory multiprocessor system structure

6.2 System Description

A typical abstraction of a shared memory multi-processor system configuration is depicted in Figure 6.2. Each node of the system contains a processor together with its local memory. All nodes are connected to the shared memory via an interconnection network. A set of co-operating tasks is running on the system performing their respective operations. Each task is sequentially executed on one of the processors, while each processor can serve (run) many tasks at a time. The co-operating tasks, possibly running on different processors, use shared data objects built in the shared memory to co-ordinate and communicate. Tasks synchronize their operations on the shared data objects through sub-operations on top of a cache-coherent shared memory. The shared memory may not though be uniformly accessible for all nodes in the system; some processors can have slower access than the others.

6.3 Algorithm

The algorithm is an extension and modification of the parallel skip list data structure presented in [106]. The sequential skip list data structure which was invented by Pugh [91], uses randomization and has a probabilistic time complexity of $O(\log N)$ where N is the maximum number of elements in the list. The data structure is basically an ordered list with randomly distributed short-cuts in order to improve search times, see Figure 6.3. The maximum height (i.e. the maximum number of next pointers) of the data structure is $\log N$. The height of each inserted node is randomized geometrically in the way that 50% of the nodes should have height 1, 25% of the nodes should have height 2 and so on. To use the data structure as a dictionary, every node contains a key and its corresponding value. The nodes are ordered in respect of key (which has to be unique for each node), the nodes with lowest keys are located first in the list. The fields of each node item are described

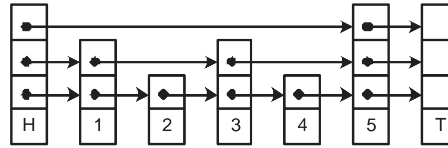


Figure 6.3: The skip list data structure with 5 nodes inserted.

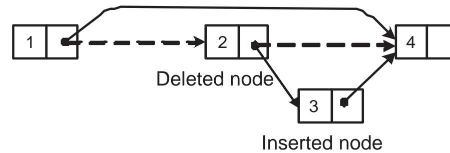


Figure 6.4: Concurrent insert and delete operation can delete both nodes.

in Figure 6.6 as it is used in this implementation. In all code figures in this section, arrays are indexed starting from 0.

In order to make the skip list construction concurrent and non-blocking, we are using two of the standard atomic synchronization primitives; Fetch-And-Add (FAA) and Compare-And-Swap (CAS). Figure 6.5 describes the specification of these primitives which are available in most modern platforms.

6.3.1 Memory Management

As we are concurrently (with possible preemptions) traversing nodes that will be continuously allocated and reclaimed, we have to consider several aspects of memory management. No node should be reclaimed and then later re-allocated while some other process is traversing this node. This can be solved for example by careful reference counting. We have selected the lock-free memory management scheme invented by Valois [122] and corrected by Michael and Scott [83], which makes use of the FAA and CAS atomic synchronization primitives.

To insert or delete a node from the list we have to change the respective set of next pointers. These have to be changed consistently, but not necessary all at once. Our solution is to have additional information on each node about its deletion (or insertion) status. This additional information will guide the concurrent processes that might traverse into one partially

```
procedure FAA(address:pointer to word, number:integer)
    atomic do
        *address := *address + number;

function CAS(address:pointer to word, oldvalue:word
 , newvalue:word):boolean
    atomic do
        if *address = oldvalue then
            *address := newvalue;
            return true;
        else return false;
```

Figure 6.5: The Fetch-And-Add (FAA) and Compare-And-Swap (CAS) atomic primitives.

deleted or inserted node. When we have changed all necessary next pointers, the node is fully deleted or inserted.

One problem, that is general for non-blocking implementations that are based on the linked-list structure, arises when inserting a new node into the list. Because of the linked-list structure one has to make sure that the previous node is not about to be deleted. If we are changing the next pointer of this previous node atomically with CAS, to point to the new node, and then immediately afterwards the previous node is deleted - then the new node will be deleted as well, as illustrated in Figure 6.4. There are several solutions to this problem. One solution is to use the CAS2 operation as it can change two pointers atomically, but this operation is not available in any existing multiprocessor system. A second solution is to insert auxiliary nodes [122] between each two normal nodes, and the latest method introduced by Harris [44] is to use one bit of the pointer values as a deletion mark. On most modern 32-bit systems, 32-bit values can only be located at addresses that are evenly dividable by 4, therefore bits 0 and 1 of the address are always set to zero. The method is then to use the previously unused bit 0 of the next pointer to mark that this node is about to be deleted, using CAS. Any concurrent *Insert* operation will then be notified about the deletion, when its CAS operation will fail.

Another memory management issue is how to de-reference pointers safely. If we simply de-reference the pointer, it might be that the corresponding node has been reclaimed before we could access it. It can also be that bit 0 of the pointer was set, thus marking that the node is deleted, and therefore the pointer is not valid. The following functions are defined for safe handling of the memory management:

```
union Link
    _: word
    ⟨p, d⟩: ⟨pointer to Node, boolean⟩

union VLink
    _: word
    ⟨p, d⟩: ⟨pointer to Value, boolean⟩

structure Node
    key, level, validLevel, version: integer
    value : union VLink
    next[level]: union Link
    prev : pointer to Node

// Global variables
head, tail : pointer to Node
// Local variables (for all functions/procedures)
node1, node2, newNode, savedNodes[maxlevel+1] : pointer to Node
prev, last, stop : pointer to Node
key1, key2, step, jump, version, version2: integer

function CreateNode(level:integer, key:integer,
 value:pointer to Value):pointer to Node
C1    node:=MALLOC_NODE();
C2    node.prev:=NULL;
C3    node.validLevel:=0;
C4    node.level:=level;
C5    node.key:=key;
C6    node.value:=⟨value,false⟩;
C7    return node;

procedure ReleaseReferences(node:pointer to Node)
R1    node.validLevel:=0;
R2    if node.prev then
R3        prev:=node.prev;
R4        node.prev:=NULL;
R5        RELEASE_NODE(prev);
```

Figure 6.6: The basic algorithm details.

```
function ReadNext(node1:pointer to pointer to Node, level:integer)
 :pointer to Node
N1    if (*node1).value.d=true then *node1:=HelpDelete(*node1,level);
N2    node2:=READ_NODE((*node1).next[level]);
N3    while node2=NULL do
N4        *node1:=HelpDelete(*node1,level);
N5        node2:=READ_NODE((*node1).next[level]);
N6    return node2;

function ScanKey(node1:pointer to pointer to Node, level:integer
 , key:integer):pointer to Node
K1    node2:=ReadNext(node1,level);
K2    while node2.key<key do
K3        RELEASE_NODE(*node1);
K4        *node1:=node2;
K5        node2:=ReadNext(node1,level);
K6    return node2;
```

Figure 6.7: The algorithm for the traversing functions.

```
    function MALLOC_NODE():pointer to Node
    function READ_NODE(address:pointer to union Link)
:pointer to Node
    function COPY_NODE(node:pointer to Node):pointer to Node
    procedure RELEASE_NODE(node:pointer to Node)
```

The function *MALLOC_NODE* allocates a new node from the memory pool of pre-allocated nodes and *RELEASE_NODE* decrements the reference counter on the corresponding given node. If the reference count reaches zero, then it calls the *ReleaseReferences* function that will call *RELEASE_NODE* on the nodes that this node has owned pointers to, and then it reclaims the node. The function *COPY_NODE* increases the reference counter for the corresponding given node and *READ_NODE* de-reference the given pointer and increase the reference counter for the corresponding node. In case the de-referenced pointer is marked, the function returns NULL.

6.3.2 Traversing

The functions for traversing the nodes are defined as follows:

function ReadNext(node1:**pointer to pointer to** Node
,level:**integer**):**pointer to** Node
function ScanKey(node1:**pointer to pointer to** Node
,level:**integer**,key:**integer**):**pointer to** Node

While traversing the nodes, processes will eventually reach nodes that are marked to be deleted. As the process that invoked the corresponding *Delete* operation might be pre-empted, this *Delete* operation has to be helped to finish before the traversing process can continue. However, it is only necessary to help the part of the *Delete* operation on the current level in order to be able to traverse to the next node. The function *ReadNext*, see Figure 6.7, traverses to the next node on the given level while helping any deleted nodes in between to finish the deletion. The function *ScanKey*, see Figure 6.7, traverses in several steps through the next pointers at the current level until it finds a node that has the same or higher key than the given key. The argument node1 in the *ReadNext* and *ScanKey* functions are continuously updated to point to the previous node of the returned node.

However, the use of the safe *ReadNext* and *ScanKey* operations for traversing the skip list, will cause the performance to be significantly lower compared to the sequential case where the next pointers are used directly. As the nodes, which are used in the lock-free memory management scheme, will be reused for the same purpose when re-allocated again after being reclaimed, the individual fields of the nodes that are not part of the memory management scheme will be intact. The *validLevel* field can therefore be used for indicating if the current node can be used for possibly traversing further on a certain level. A value of 0 indicates that this node can not be used for traversing at all, as it is possibly reclaimed or not yet inserted. As the *validLevel* field is only set to 0 directly before reclamation in line R1, a positive value indicates that the node is allocated. A value of $n + 1$ indicated that this node has been inserted up to level n. However, the next pointer of level n on the node may have been marked and thus indicating possible deletion at that level of the node. As the node is not reclaimed the *key* field is intact, and therefore it is possible to traverse from the previous node to the current position. By increasing the reference count of the node before checking the *validLevel* field, it can be assured that the node stays allocated if it was allocated directly after the increment. Because the next pointers are always updated to point (regardless of the mark) either to nothing (NULL) or to a node that is part of the memory management, allocated or reclaimed, it is possible in some scenarios to traverse directly through the next pointers. This approach is taken by the *SearchLevel* function, see

```
function SearchLevel(last:pointer to pointer to Node, lastLevel:integer,
 level:integer, key:integer): pointer to Node
S1     node1:=*last;
S2     stop:=NULL;
S3     while true do
S4         node2:=node1.next[level].p;
S5         if node2=NULL then
S6             if node1=*last then
S7                 *last:=HelpDelete(*last,lastLevel);
S8             node1:=*last;
S9         else if node2.key≥key then
S10            COPY_NODE(node1);
S11            if (node1.validLevel>level or node1=*last or node1=stop)
                       and node1.key<key and node1.key≥(*last).key then
S12                if node1.validLevel≤level then
S13                    RELEASE_NODE(node1);
S14                    node1:=COPY_NODE(*last);
S15                    node2:=ScanKey(&node1,level,key);
S16                    RELEASE_NODE(node2);
S17                return node1;
S18            RELEASE_NODE(node1);
S19            stop:=node1;
S20            if (*last).value.d = true then
S21                *last:=HelpDelete(*last,lastLevel);
S22            node1:=*last;
S23        else if node2.key≥(*last).key then
S24            node1:=node2;
S25        else
S26            if (*last).value.d = true then
S27                *last:=HelpDelete(*last,lastLevel);
S28            node1:=*last;
```

Figure 6.8: The algorithm for the SearchLevel function.

Figure 6.8, which traverses rapidly from an allocated node *last* and returns the node which *key* field is the highest key that is lower than the searched key at the current level. During the rapid traversal it is checked that the current key is within the search boundaries in line S23 and S11, otherwise the traversal restarts from the *last* node as this indicates that a node has been reclaimed and re-allocated while traversed. When the node suitable for returning has been reached, it is checked that it is allocated in line S11 and also assured that it then stays allocated in line S10. If this succeeds the node is returned, otherwise the traversal restarts at node *last*. If this fails twice, the traversal are done using the safe *ScanKey* operations in lines S12 to S16, as this indicates that the node possibly is inserted at the current level, but the *validLevel* field has not yet been updated. In case the node *last* is marked for deletion, it might have been deleted at the current level and thus it can not be used for traversal. Therefore the node *last* is checked if it is marked in lines S6, S20 and S26. If marked, the node *last* will be helped to fully delete on the current level and *last* is set to the previous node.

6.3.3 Inserting and Deleting Nodes

The implementation of the *Insert* operation, see Figure 6.9, starts in lines I4-I10 with a search phase to find the node after which the new node ($newNode$) should be inserted. This search phase starts from the head node at the highest level and traverses down to the lowest level until the correct node is found ($node1$). When going down one level, the last node traversed on that level is remembered (*savedNodes*) for later use (this is where we should insert the new node at that level). Now it is possible that there already exists a node with the same key as of the new node, this is checked in lines I12-I23, the value of the old node ($node2$) is changed atomically with a CAS. Otherwise, in lines I24-I45 it starts trying to insert the new node starting with the lowest level increasing up to the level of the new node. The next pointers of the (to be previous) nodes are changed atomically with a CAS. After the new node has been inserted at the lowest level, it is possible that it is deleted by a concurrent *Delete* operation before it has been inserted at all levels, and this is checked in lines I38 and I46. The *FindKey* operation, see Figure 6.10, basically follows the *Insert* operation.

The *Delete* operation, see Figure 6.11, starts in lines D1-D4 with a search phase to find the first node which key is equal or higher than the searched key. This search phase starts from the head node at the highest level and traverses down to the lowest level until the correct node is found ($node1$).

```
function Insert(key:integer, value:pointer to Value):boolean
I1      Choose level randomly according to the skip list distribution
I2      newNode:=CreateNode(level,key,value);
I3      COPY_NODE(newNode);
I4      savedNodes[maxLevel]:=head;
I5      for i:=maxLevel-1 to 0 step -1 do
I6          savedNodes[i]:=SearchLevel(&savedNodes[i+1],i+1,i,key);
I7          if maxLevel-1>i≥level-1 then RELEASE_NODE(savedNodes[i+1]);
I8      node1:=savedNodes[0];
I9      while true do
I10         node2:=ScanKey(&node1,0,key);
I11         ⟨value2,d⟩:=node2.value;
I12         if d=false and node2.key=key then
I13             if CAS(&node2.value,⟨value2,false⟩,⟨value,false⟩) then
I14                 RELEASE_NODE(node1);
I15                 RELEASE_NODE(node2);
I16                 for i:=1 to level-1 do
I17                     RELEASE_NODE(savedNodes[i]);
I18                 RELEASE_NODE(newNode);
I19                 RELEASE_NODE(newNode);
I20                 return true_2;
I21             else
I22                 RELEASE_NODE(node2);
I23                 continue;
I24         newNode.next[0]:=⟨node2,false⟩ ;
I25         RELEASE_NODE(node2);
I26         if CAS(&node1.next[0],⟨node2,false⟩,⟨newNode,false⟩) then
I27             RELEASE_NODE(node1);
I28             break;
I29         Back-Off
I30     newNode.version:=newNode.version+1;
I31     newNode.validLevel:=1;
I32     for i:=1 to level-1 do
I33         node1:=savedNodes[i];
I34         while true do
I35             node2:=ScanKey(&node1,i,key);
I36             newNode.next[i]:=⟨node2,false⟩;
I37             RELEASE_NODE(node2);
I38             if newNode.value.d=true then
I39                 RELEASE_NODE(node1);
I40                 break;
I41             if CAS(&node1.next[i],⟨node2,false⟩,⟨newNode,false⟩) then
I42                 newNode.validLevel:=i+1;
I43                 RELEASE_NODE(node1);
I44                 break;
I45             Back-Off
I46     if newNode.value.d = true then newNode:=HelpDelete(newNode,0);
I47     RELEASE_NODE(newNode);
I48     return true;
```

Figure 6.9: The algorithm for the Insert function.

```
function FindKey(key: integer):pointer to Value
F1    last:=COPY_NODE(head);
F2    for i:=maxLevel-1 to 0 step -1 do
F3        node1:=SearchLevel(&last,i,i,key);
F4        RELEASE_NODE(last);
F5        last:=node1;
F6    node2:=ScanKey(&last,0,key);
F7    RELEASE_NODE(last);
F8    ⟨value,d⟩:=node2.value;
F9    if node2.key≠key or d=true then
F10       RELEASE_NODE(node2);
F11       return NULL;
F12   RELEASE_NODE(node2);
F13   return value;
```

Figure 6.10: The algorithm for the FindKey function.

When going down one level, the last node traversed on that level is remembered (*savedNodes*) for later use (this is the previous node at which the next pointer should be changed in order to delete the targeted node at that level). If the found node is the correct node, it tries to set the deletion mark of the *value* field in line D8 using the CAS primitive, and if it succeeds it also writes a valid pointer (which corresponding node will stay allocated until this node gets reclaimed) to the *prev* field of the node in line D9. This *prev* field is necessary in order to increase the performance of concurrent *HelpDelete* operations, these otherwise would have to search for the previous node in order to complete the deletion. The next step is to mark the deletion bits of the next pointers in the node, starting with the lowest level and going upwards, using the CAS primitive in each step, see lines D16-D19. Afterwards in lines D20-D32 it starts the actual deletion by changing the next pointers of the previous node (*prev*), starting at the highest level and continuing downwards. The reason for doing the deletion in decreasing order of levels, is that concurrent operations that are in the search phase also start at the highest level and proceed downwards, in this way the concurrent search operations will sooner avoid traversing this node. The procedure performed by the *Delete* operation in order to change each next pointer of the previous node, is to first search for the previous node and then perform the CAS primitive until it succeeds.

```
function DeleteKey(key: integer):pointer to Value
        return Delete(key,false,NULL);
function Delete(key: integer, delval:boolean,
 value:pointer to Value):pointer to Value
D1     savedNodes[maxLevel]:=head;
D2     for i:=maxLevel-1 to 0 step -1 do
D3         savedNodes[i]:=SearchLevel(&savedNodes[i+1],i+1,i,key);
D4     node1:=ScanKey(&savedNodes[0],0,key);
D5     while true do
D6         if not delval then ⟨value,d⟩:=node1.value;
D7         if node1.key=key and (not delval or node1.value.p=value) and d=false then
D8             if CAS(&node1.value,⟩value,false⟨,⟩value,true⟨) then
D9                 node1.prev:=COPY_NODE(savedNodes[(node1.level-1)/2]);
D10                break;
D11            else continue;
D12        RELEASE_NODE(node1);
D13        for i:=0 to maxLevel-1 do
D14            RELEASE_NODE(savedNodes[i]);
D15        return NULL;
D16    for i:=0 to node1.level-1 do
D17        repeat
D18            ⟨node2,d⟩:=node1.next[i];
D19        until d=true or CAS(&node1.next[i],⟨node2,false⟩,⟨node2,true⟩);
D20    for i:=node1.level-1 to 0 step -1 do
D21        prev:=savedNodes[i];
D22        while true do
D23            if node1.next[i]=⟨NULL,true⟩ then break;
D24            last:=ScanKey(&prev,i,node1.key);
D25            RELEASE_NODE(last);
D26            if last≠node1 or node1.next[i]=⟨NULL,true⟩ then break;
D27            if CAS(&prev.next[i],⟨node1,false⟩,⟨node1.next[i].p,false⟩) then
D28                node1.next[i]:=⟨NULL,true⟩;
D29                break;
D30            if node1.next[i]=⟨NULL,true⟩ then break;
D31            Back-Off
D32        RELEASE_NODE(prev);
D33    for i:=node1.level to maxLevel-1 do
D34        RELEASE_NODE(savedNodes[i]);
D35    RELEASE_NODE(node1);
D36    RELEASE_NODE(node1);
D37    return value;
```

Figure 6.11: The algorithm for the DeleteKey function.

```
function HelpDelete(node:pointer to Node, level:integer):pointer to Node
H1    for i:=level to node.level-1 do
H2        repeat
H3            ⟨node2,d⟩:=node.next[i];
H4        until d=true or CAS(&node.next[i],⟨node2,false⟩,⟨node2,true⟩);
H5    prev:=node.prev;
H6    if not prev or level ≥ prev.validLevel then
H7        prev:=COPY_NODE(head);
H8    else COPY_NODE(prev);
H9    while true do
H10       if node.next[level]=⟨NULL,true⟩ then break;
H11       for i:=prev.validLevel-1 to level step -1 do
H12           node1:=SearchLevel(&prev,i,i,node.key);
H13           RELEASE_NODE(prev);
H14           prev:=node1;
H15       last:=ScanKey(&prev,level,node.key);
H16       RELEASE_NODE(last);
H17       if last≠node or node.next[level]=⟨NULL,true⟩ then break;
H18       if CAS(&prev.next[level],⟨node,false⟩,⟨node.next[level].p,false⟩) then
H19           node.next[level]:=⟨NULL,true⟩;
H20           break;
H21       if node.next[level]=⟨NULL,true⟩ then break;
H22       Back-Off
H23   RELEASE_NODE(node);
H24   return prev;
```

Figure 6.12: The algorithm for the HelpDelete function.

6.3.4 Helping Scheme

The algorithm has been designed for pre-emptive as well as fully concurrent systems. In order to achieve the lock-free property (that at least one thread is doing progress) on pre-emptive systems, whenever a search operation finds a node that is about to be deleted, it calls the *HelpDelete* operation and then proceeds searching from the previous node of the deleted. The *HelpDelete* operation, see Figure 6.12, tries to fulfill the deletion on the current level and returns when it is completed. It starts in lines H1-H4 with setting the deletion mark on all next pointers in case they have not been set. In lines H5-H6 it checks if the node given in the prev field is valid for deletion on the current level, otherwise it starts the search at the head node. In lines H11-H16 it searches for the correct node (*prev*). The actual deletion of this node on the current level takes place in line H18. Lines H10-H22 will be repeated until the node is deleted at the current level. This operation might execute concurrently with the corresponding *Delete* operation as well with

other *HelpDelete* operations, and therefore all operations synchronize with each other in lines D23, D26, D28, D30, H10, H17, H19 and H21 in order to avoid executing sub-operations that have already been performed.

In fully concurrent systems though, the helping strategy can downgrade the performance significantly. Therefore the algorithm, after a number of consecutive failed attempts to help concurrent *Delete* operations that stops the progress of the current operation, puts the current operation into back-off mode. When in back-off mode, the thread does nothing for a while, and in this way avoids disturbing the concurrent operations that might otherwise progress slower. The duration of the back-off is proportional to the number of threads, and for each consecutive entering of back-off mode during one operation invocation, the duration is increased exponentially.

6.3.5 Value Oriented Operations

The *FindValue* and *DeleteValue* operations, see Figure 6.13, traverse from the head node along the lowest level in the skip list until a node with the searched value is found. In every traversal step, it has to be assured that the step is taken from a valid node to a valid node, both valid at the same time. The *validLevel* field of the node can be used to safely verify the validity, unless the node has been reclaimed. The *version* field is incremented by the *Insert* operation in line I30, after the node has been inserted at the lowest level, and directly before the *validLevel* is set to indicate validity. By performing two consecutive reads of the *version* field with the same contents, and successfully verifying the validity in between the reads, it can be concluded that the node has stayed valid from the first read of the version until the successful validity check. This is done is lines V8-V13. If this fails, it restarts and traverses the safe node *last* one step using the *ReadNext* function in lines V14-V21. After a certain number (*jump*) of successful fast steps, an attempt to advance the *last* node to the current position is performed in lines V29-V38. If this attempt succeeds, the threshold *jump* is increased by 1 1/2 times, otherwise it is halved. The traversal is continued until a node with the searched value is reached in line V24 or that the tail node is reached in line V21. In case the found node should be deleted, the *Delete* operation is called for this purpose in line V26.

```
function FindValue(value: pointer to Value):integer
        return FDValue(value,false);
function DeleteValue(value: pointer to Value):integer
        return FDValue(value,true);
function FDValue(value: pointer to Value, delete: boolean):integer
V1      jump:=16;
V2      last:=COPY_NODE(head);
      next_jump:
V3      node1:=last;
V4      key1:=node1.key;
V5      step:=0;
V6      while true do
V7          ok=false;
V8          version:=node1.version;
V9          ⟨node2,d⟩:=node1.next[0];
V10         if d=false and node2≠NULL then
V11             version2:=node2.version;
V12             key2:=node2.key;
V13             if node1.key=key1 and node1.validLevel>0 and node1.next[0]=node2
                 and node1.version=version and node2.key=key2 and
                 node2.validLevel>0 and node2.version=version2 then ok:=true;
V14         if not ok then
V15             node1:=node2:=ReadNext(&last,0);
V16             key1:=key2:=node2.key;
V17             version2:=node2.version;
V18             RELEASE_NODE(last);
V19             last:=node2;
V20             step:=0;
V21         if node2=tail then
V22             RELEASE_NODE(last);
V23             return ⊥;
V24         if node2.value.p=value then
V25             if node2.version=version2 then
V26                 if not delete or Delete(key2,true,value)=value then
V27                     RELEASE_NODE(last);
V28                     return key2;
V29         else if ++step≥jump then
V30             COPY_NODE(node2);
V31             if node2.validLevel=0 or node2.key≠key2 then
V32                 RELEASE_NODE(node2);
V33                 node2:=ReadNext(&last,0);
V34                 if jump≥4 then jump:=jump/2;
V35             else jump:=jump+jump/2;
V36             RELEASE_NODE(last);
V37             last:=node2;
V38             goto next_jump;
V39         else
V40             key1:=key2;
V41             node1:=node2;
```

Figure 6.13: The algorithm for the FindValue function.

6.4 Correctness

In this section we present the proofs of correctness for our algorithm. We first prove that our algorithm is a linearizable one [50] and then we prove that it is lock-free. A set of definitions that will help us to structure and shorten the proof is first explained in this section. We start by defining the sequential semantics of our operations and then introduce two definitions concerning concurrency aspects in general.

Definition 7 *We denote with L_t the abstract internal state of a dictionary at the time t. L_t is viewed as a set of unique pairs $\langle k, v\rangle$ consisting of a unique key k and a corresponding unique value v. The operations that can be performed on the dictionary are $Insert$ (I), $FindKey$ (FK), $DeleteKey$ (DK), $FindValue$ (FV) and $DeleteValue$ (DV). The time t_1 is defined as the time just before the atomic execution of the operation that we are looking at, and the time t_2 is defined as the time just after the atomic execution of the same operation. The return value of $true_2$ is returned by an $Insert$ operation that has succeeded to update an existing node, the return value of $true$ is returned by an $Insert$ operation that succeeds to insert a new node. In the following expressions that defines the sequential semantics of our operations, the syntax is $S_1 : O_1, S_2$, where S_1 is the conditional state before the operation O_1, and S_2 is the resulting state after performing the corresponding operation:*

$$\langle k_1, _\rangle \notin L_{t_1} : \mathbf{I_1}(\langle \mathbf{k_1}, \mathbf{v_1}\rangle) = \mathbf{true}, \mathbf{L_{t_2}} = \mathbf{L_{t_1}} \cup \{\langle \mathbf{k_1}, \mathbf{v_1}\rangle\} \tag{6.1}$$

$$\begin{gathered}\langle k_1, v_{1_1}\rangle \in L_{t_1} : \mathbf{I_1}(\langle \mathbf{k_1}, \mathbf{v_{1_2}}\rangle) = \mathbf{true_2}, \\ \mathbf{L_{t_2}} = \mathbf{L_{t_1}} \setminus \{\langle \mathbf{k_1}, \mathbf{v_{1_1}}\rangle\} \cup \{\langle \mathbf{k_1}, \mathbf{v_{1_2}}\rangle\}\end{gathered} \tag{6.2}$$

$$\langle k_1, v_1\rangle \in L_{t_1} : \mathbf{FK_1}(\mathbf{k_1}) = \mathbf{v_1} \tag{6.3}$$

$$\langle k_1, v_1\rangle \notin L_{t_1} : \mathbf{FK_1}(\mathbf{k_1}) = \perp \tag{6.4}$$

$$\langle k_1, v_1\rangle \in L_{t_1} : \mathbf{DK_1}(\mathbf{k_1}) = \mathbf{v_1}, \mathbf{L_{t_2}} = \mathbf{L_{t_1}} \setminus \{\langle \mathbf{k_1}, \mathbf{v_1}\rangle\} \tag{6.5}$$

$$\langle k_1, v_1 \rangle \notin L_{t_1} : \mathbf{DK_1(k_1)} = \bot \tag{6.6}$$

$$\langle k_1, v_1 \rangle \in L_{t_1} : \mathbf{FV_1(v_1)} = \mathbf{k_1} \tag{6.7}$$

$$\langle k_1, v_1 \rangle \notin L_{t_1} : \mathbf{FV_1(v_1)} = \bot \tag{6.8}$$

$$\langle k_1, v_1 \rangle \in L_{t_1} : \mathbf{DV_1(v_1)} = \mathbf{k_1}, \mathbf{L_{t_2}} = \mathbf{L_{t_1}} \setminus \{\langle \mathbf{k_1}, \mathbf{v_1} \rangle\} \tag{6.9}$$

$$\langle k_1, v_1 \rangle \notin L_{t_1} : \mathbf{DV_1(v_1)} = \bot \tag{6.10}$$

Note that the operations will work correctly also if relaxing the condition that values are unique. However, the results of the $FindValue$ and $DeleteValue$ operations will be undeterministic in the sense that it is not decidable which key value that will be returned in the presence of several key-value pairs with the same value. In the case of the $DeleteValue$ operation, still only one pair will be removed.

Definition 8 *In a global time model each concurrent operation Op "occupies" a time interval $[b_{Op}, f_{Op}]$ on the linear time axis ($b_{Op} < f_{Op}$). The precedence relation (denoted by '$\rightarrow$') is a relation that relates operations of a possible execution, $Op_1 \rightarrow Op_2$ means that Op_1 ends before Op_2 starts. The precedence relation is a strict partial order. Operations incomparable under $\rightarrow$ are called* overlapping*. The overlapping relation is denoted by $\|$ and is commutative, i.e. $Op_1 \| Op_2$ and $Op_2 \| Op_1$. The precedence relation is extended to relate sub-operations of operations. Consequently, if $Op_1 \rightarrow Op_2$, then for any sub-operations op_1 and op_2 of Op_1 and Op_2, respectively, it holds that $op_1 \rightarrow op_2$. We also define the direct precedence relation $\rightarrow_d$, such that if $Op_1 \rightarrow_d Op_2$, then $Op_1 \rightarrow Op_2$ and moreover there exists no operation Op_3 such that $Op_1 \rightarrow Op_3 \rightarrow Op_2$.*

Definition 9 *In order for an implementation of a shared concurrent data object to be linearizable [50], for every concurrent execution there should exist an equal (in the sense of the effect) and valid (i.e. it should respect the semantics of the shared data object) sequential execution that respects the partial order of the operations in the concurrent execution.*

Next we are going to study the possible concurrent executions of our implementation. First we need to define the interpretation of the abstract internal state of our implementation.

Definition 10 *The pair $\langle k, v\rangle$ is present ($\langle k, v\rangle \in L$) in the abstract internal state L of our implementation, when there is a next pointer from a present node on the lowest level of the skip list that points to a node that contains the pair $\langle k, v\rangle$, and this node is not marked as deleted with the mark on the value.*

Lemma 10 *The definition of the abstract internal state for our implementation is consistent with all concurrent operations examining the state of the dictionary.*

Proof: As the next and value pointers are changed using the CAS operation, we are sure that all threads see the same state of the skip list, and therefore all changes of the abstract internal state seems to be atomic. □

As we are using a lock-free memory management scheme with a fixed memory size and where reclaimed nodes can only be allocated again for the same purpose, we know that there is a fixed number of nodes that will be used with the skip list, and that the individual fields (like key, value, next etc.) of the nodes are only changed by the operations of this algorithm.

Definition 11 *A node that is used in the skip list is defined as **valid** if the node is inserted at the lowest level, i.e. there is a next pointer on any other valid node that points (disregarding the eventual mark) to this node, or the node the validLevel field set to higher than zero and has been fully deleted but not yet been reclaimed. All other nodes are defined as **invalid**, i.e. the node is reclaimed, or has been allocated but not yet inserted at the lowest level. A node that is used in the skip list is defined as **valid at level i** if the node is inserted at level i, i.e. there is a next pointer at level i on any other valid node that points (disregarding the eventual mark) to this node, or the node has the validLevel field set higher than i and has been fully deleted but not yet been reclaimed.*

An interesting observation of the previous definition is that if a node is present in the abstract internal state L then it is also valid, and that if a node is valid then also the individual fields of the node are valid.

Lemma 11 *A valid node with a increased reference count, can always be used to continue traversing from, even if the node is deleted.*

Proof: For every instruction in the algorithm that increments the reference count (i.e. the *READ_NODE* and *COPY_NODE* functions),

there exists a corresponding instruction that decrements the reference count equally much (i.e. the *RELEASE_NODE* function). This means that if the reference count has been incremented, that the reference count can not reach zero (and thus be reclaimed) until the corresponding decrement instruction is executed. A node with a increased reference count can thus not be reclaimed, unless it already was reclaimed before the reference count was incremented. As the node is valid, the key field is also valid, which means that we know the absolute position in the skip list. If the node is not deleted the next pointers can be used for traversing. Otherwise it is always possible to get to the previous node by searching from the head of the skip list using the key, and traverse from there. □

Lemma 12 *The node node1 that is found in line S17 of the SearchLevel function, is a valid node with a increased reference count and will therefore not be reclaimed by concurrent tasks, and is (or was) the node with the nearest key that is lower than the searched key.*

Proof: The reference count is incremented in line S10 before the check for validity in line S11, which means that if the validity test succeeds then the node will stay valid. The *validLevel* field of a node is set in lines I31 and I42 to the current level plus one after each successful step of the *Insert* operation, and is set to zero in line R1 just before the node is fully deleted and will be reclaimed. This means that if the *validLevel* field is more than the current level, that the node is valid. Alternatively if the node is the same as a known valid node *last*, then it is also valid. If the node is valid, it is also checked that the key value is lower than was searched for. Before the validity check, the next node of *node*1 was read as *node*2 in line S4 and its key was checked to be more than or equal to the searched key in line S9. This means that the node *node*1 in line S17 is valid and was (or still is) the node with the nearest key that is lower than the searched key. □

Lemma 13 *The node node2 that is found in line V26 of the FindValue and DeleteValue functions, was present during the read of its value and this value was the same as searched for.*

Proof: The *Delete* operation marks the value before it starts with marking the next pointers and removing the node from the skip list. A node that is valid and the value is nonmarked, is therefore present in the dictionary as the node must be inserted at the lowest level and not yet deleted. The node was valid in line V13 as the *validLevel* field was positive. The *version* field

is only incremented in line I30, directly before the *validLevel* field becomes positive. As the version was the same before the check for validity in line V13 as well as after the check for equalness and validity in V24, this means that the node has been valid all the time. □

Lemma 14 *The node node2 that is found in line V37 of the FindValue and DeleteValue functions, is a valid node at the current, or between the previously known safe node last and the current position along the search-path. The node also has a increased reference count and will therefore not be reclaimed by concurrent tasks.*

Proof: The reference count is incremented in line V30 before the check for validity in line V31, which means that if the validity test succeeds then the node will stay valid. The validity check follows Lemma 12. If the node was not valid or the key of the node did not match the current position in the searchpath (i.e. the key field has changed due to reclamation of the node before the increment of the reference count) then *node*2 will be set to the next node of *last* using the *ReadNext* function. □

Lemma 15 *The functions FindValue and DeleteValue will not skip any nodes while traversing the skip list from left to right, and will therefore traverse through all nodes that was present in the skip list at the start of the traversing and which was still present while traversed.*

Proof: In order to safely move from *node*1 to *node*2, it has to be assured that both nodes are valid and that *node*1 has been pointing to *node*2 at the lowest level while both were valid, and that *node*1 is at the current position (i.e. the *key* field). If this holds we can conclude that *node*2 is, or was at the time of starting the traversal, the very next node of *node*1. These properties are checked in line V13, where the validity is confirmed to have hold during the check of the other properties by that the *version* fields of both nodes were the same as in lines V8 and V11 before checking the validity using the *validLevel* field. If the check in line V13 failed, then *node*2 will be set to the next node of *last* using the *ReadNext* function, which position is between the previously known safe node *last* and the current position along the searchpath. Given by Lemma 14, when the node *last* is updated, it is always set to a valid node with a position between the next node of the previously known safe node *last* and the current position along the searchpath. Consequently, the functions *FindValue* and *DeleteValue* will not skip any nodes while traversing the skip list from left to right. □

Definition 12 *The decision point of an operation is defined as the atomic statement where the result of the operation is finitely decided, i.e. independent of the result of any sub-operations after the decision point, the operation will have the same result. We define the state-read point of an operation to be the atomic statement where the state of the dictionary, which result the decision point depends on is read. We also define the state-change point as the atomic statement where the operation changes the abstract internal state of the dictionary after it has passed the corresponding decision point.*

We will now use these points in order to show the existance and location in execution history of a point where the concurrent operation can be viewed as it occured atomically, i.e. the *linearizability point*.

Lemma 16 *An $Insert$ operation which succeeds ($I(\langle k, v\rangle) = true$), takes effect atomically at one statement.*

Proof: The decision point for an $Insert$ operation which succeeds ($I(\langle k, v\rangle) = true$), is when the CAS sub-operation in line I26 (see Figure 6.9) succeeds, all following CAS sub-operations will eventually succeed, and the $Insert$ operation will finally return $true$. The state of the list (L_{t_1}) directly before the passing of the decision point must have been $\langle k, _\rangle \notin L_{t_1}$, otherwise the CAS would have failed. The state of the list directly after passing the decision point will be $\langle k, v\rangle \in L_{t_2}$. Consequently, the linearizability point will be the CAS sub-operation in line I26. □

Lemma 17 *An $Insert$ operation which updates ($I(\langle k, v\rangle) = true_2$), takes effect atomically at one statement.*

Proof: The decision point for an $Insert$ operation which updates ($I(\langle k, v\rangle) = true_2$), is when the CAS will succeed in line I13. The state of the list (L_{t_1}) directly before passing the decision point must have been $\langle k, _\rangle \in L_{t_1}$, otherwise the CAS would have failed. The state of the list directly after passing the decision point will be $\langle k, v\rangle \in L_{t_3}$. Consequently, the linearizability point will be the CAS sub-operation in line I13. □

Lemma 18 *A $FindKey$ operation which succeeds ($FK(k) = v$), takes effect atomically at one statement.*

Proof: The decision point for a $FindKey$ operation which succeeds ($FK(k) = v$), is when the check for marked value in line F9 fails. The state-read point

is when the value of the node is read in line F8. As the *key* field of the node can not change concurrently, the state of the list (L_{t_1}) directly before passing the state-read point must have been $\langle k, v \rangle \in L_{t_1}$. Consequently, the linearizability point will be the read sub-operation of the *value* field in line F8. □

Lemma 19 *A FindKey operation which fails ($FK(k) = \bot$), takes effect atomically at one statement.*

Proof: The decision point for a *FindKey* operation which fails ($FK(k) = \bot$), is when the check for key equality fails or when the check for marked value in line F9 succeeds. If the key equality in line F9 fails, the state-read point is the read sub-operation of *READ_NODE* in line N2 or N5 (from K1 or K5, from F6) when the next pointer at lowest level of the previous node is read. If the check for marked value in line F9 succeeds, the state-read point is the read sub-operation of the *value* field in line F8. In both cases, the state of the list (L_{t_1}) directly before passing the state-read point must have been $\langle k, v \rangle \notin L_{t_1}$. Consequently, the linearizability point will be either of the state-read points. □

Lemma 20 *A DeleteKey operation which succeeds ($DK(k) = v$), takes effect atomically at one statement.*

Proof: The decision point for a *DeleteKey* operation which succeeds ($DK(k) = v$) is when the CAS sub-operation in line D8 (see Figure 6.11) succeeds. The state of the list (L_t) directly before passing of the decision point must have been $\langle k, v \rangle \in L_t$, otherwise the CAS would have failed. The state of the list directly after passing the decision point will be $\langle k, v \rangle \notin L_t$. Consequently, the linearizability point will be the CAS sub-operation in line D8. □

Lemma 21 *A DeleteKey operations which fails ($DK(k) = \bot$), takes effect atomically at one statement.*

Proof: The decision point for a *DeleteKey* operation which fails ($DK(k) = \bot$), is when the check for key equality fails or when the check for non-marked value in line D7 fails. If the key equality in line D7 fails, the state-read point is the read sub-operation of *READ_NODE* in line N2 or N5 (from K1 or K5, from D4) when the next pointer at lowest level of the previous node is read. If the check for non-marked value in line D7 fails, the state-read point is the read sub-operation of the *value* field in line D6. In both cases, the

state of the list (L_{t_1}) directly before passing the state-read point must have been $\langle k, v\rangle \notin L_{t_1}$. Consequently, the linearizability point will be either of the state-read points. □

Lemma 22 *A FindValue operation which succeeds ($FV(v) = k$), takes effect atomically at one statement.*

Proof: The decision point for a $FindValue$ operation which succeeds ($FV(v) = k$), is when the check for valid node (and also a valid *value* field in line V24) in line V25 succeeds. The state-read point is when the *value* field is read in line V24. As the *key* field of the node can not change concurrently and as given by Lemma 13, the state of the list (L_{t_1}) directly before passing the state-read point must have been $\langle k, v\rangle \in L_{t_1}$. Consequently, the linearizability point will be the read sub-operation of the *value* field in line V24. □

Lemma 23 *A FindValue operation which fails ($FV(v) = \bot$), takes effect atomically at one statement.*

Proof: For a $FindValue$ operation which fails ($FV(v) = \bot$), all checks for value equality in line V24 fails. Because of the uniqueness of values, there can be at most one pair $\langle k_1, v_1\rangle$ present in the dictionary at one certain moment of time where $v = v_1$. Given by Lemma 15 we know that the algorithm will pass by the node with key k_1 if $\langle k_1, _\rangle \in L_{t_1}$ at the time of traversal, and that all keys in the possible range of keys will be passed by as we start traversing from the lowest key and that the skip list is ordered.

If during the execution, $key1 < k_1 < key2$, then if the check in line V13 succeeds, the state-read point is the read sub-operation in line V9, otherwise if the check in line V13 fails, the state-read point is the hidden read sub-operation of the next pointer of node $node1$ in the $READ_NODE$ function in line N2 or N5 (from V15). The state of the list (L_{t_1}) directly before passing the state-read point must have been $\langle k_1, v_1\rangle \notin L_{t_1}$. Consequently, the linearizability point will be the state-read point.

If during the execution, $key2 = k_1$ and the *value* field of node $node2$ was not equal to v in line V24, then the state-read point will be the read sub-operation of the *value* field in line V24. The state of the list (L_{t_1}) directly before passing the state-read point must have been $\langle k_1, v_1\rangle \notin L_{t_1}$. Consequently, the linearizability point will be the state-read point.

As all operations on shared memory as read, write and atomic primitives, are atomic, they can be totally ordered. If during the execution, $key2 = k_1$

and the *value* field of node *node*2 was marked in line V24, the linearizability point will be the concurrent successful CAS sub-operation on the same *value* field in line D8 that can be ordered before the read sub-operation in line V24, and after the read sub-operation of the *head* node in line V2. If no such concurrent CAS sub-operation exists, the linearizability point will be the read sub-operation of the *head* node in line V24. The state of the list (L_{t_1}) directly after passing the linearizability point must have been $\langle k_1, v_1 \rangle \notin L_{t_1}$. □

Lemma 24 *A DeleteValue operation which succeeds ($DV(v) = k$), takes effect atomically at one statement.*

Proof: The decision point for a *DeleteValue* operation which succeeds ($DV(v) = k$) is when the CAS sub-operation in line D8 (from V26) succeeds. The state of the list (L_t) directly before passing of the decision point must have been $\langle k, v \rangle \in L_t$, otherwise the CAS would have failed. The state of the list directly after passing the decision point will be $\langle k, v \rangle \notin L_t$. Consequently, the linearizability point will be the CAS sub-operation in line D8. □

Lemma 25 *A DeleteValue operation which fails ($DV(v) = \perp$), takes effect atomically at one statement.*

Proof: The proof is the same as for *FindValue*, see Lemma 23. □

Definition 13 *We define the relation $\Rightarrow$ as the total order and the relation $\Rightarrow_d$ as the direct total order between all operations in the concurrent execution. In the following formulas, $E_1 \Longrightarrow E_2$ means that if E_1 holds then E_2 holds as well, and $\oplus$ stands for exclusive or (i.e. $a \oplus b$ means $(a \vee b) \wedge \neg(a \wedge b)$):*

$$\begin{gathered}\mathbf{Op_1} \rightarrow_{\mathbf{d}} \mathbf{Op_2}, \nexists \mathbf{Op_3}.\mathbf{Op_1} \Rightarrow_{\mathbf{d}} \mathbf{Op_3}, \\ \nexists \mathbf{Op_4}.\mathbf{Op_4} \Rightarrow_{\mathbf{d}} \mathbf{Op_2} \implies \mathbf{Op_1} \Rightarrow_{\mathbf{d}} \mathbf{Op_2}\end{gathered} \tag{6.11}$$

$$\mathbf{Op_1} \parallel \mathbf{Op_2} \implies \mathbf{Op_1} \Rightarrow_{\mathbf{d}} \mathbf{Op_2} \oplus \mathbf{Op_2} \Rightarrow_{\mathbf{d}} \mathbf{Op_1} \tag{6.12}$$

$$\mathbf{Op_1} \Rightarrow_{\mathbf{d}} \mathbf{Op_2} \implies \mathbf{Op_1} \Rightarrow \mathbf{Op_2} \tag{6.13}$$

$$\mathbf{Op_1} \Rightarrow \mathbf{Op_2}, \mathbf{Op_2} \Rightarrow \mathbf{Op_3} \implies \mathbf{Op_1} \Rightarrow \mathbf{Op_3} \tag{6.14}$$

Lemma 26 *The operations that are directly totally ordered using formula 6.11, form an equivalent valid sequential execution.*

Proof: If the operations are assigned their direct total order ($Op_1 \Rightarrow_d Op_2$) by formula 6.11 then also the linearizability point of Op_1 is executed before the respective point of Op_2. In this case the operations semantics behave the same as in the sequential case, and therefore all possible executions will then be equivalent to one of the possible sequential executions. □

Lemma 27 *The operations that are directly totally ordered using formula 6.12 can be ordered unique and consistent, and form an equivalent valid sequential execution.*

Proof: Assume we order the overlapping operations according to their linearizability points. As the state before as well as after the linearizability points is identical to the corresponding state defined in the semantics of the respective sequential operations in formulas 6.1 to 6.10, we can view the operations as occurring at the linearizability point. As the linearizability points consist of atomic operations and are therefore ordered in time, no linearizability point can occur at the very same time as any other linearizability point, therefore giving a unique and consistent ordering of the overlapping operations. □

Lemma 28 *With respect to the retries caused by synchronization, one operation will always do progress regardless of the actions by the other concurrent operations.*

Proof: We now examine the possible execution paths of our implementation. There are several potentially unbounded loops that can delay the termination of the operations. We call these loops retry-loops. If we omit the conditions that are because of the operations semantics (i.e. searching for the correct position etc.), the loop retries when sub-operations detect that a shared variable has changed value. This is detected either by a subsequent read sub-operation or a failed CAS. These shared variables are only changed concurrently by other CAS sub-operations. According to the definition of CAS, for any number of concurrent CAS sub-operations, exactly one will succeed. This means that for any subsequent retry, there must be one CAS that succeeded. As this succeeding CAS will cause its retry loop to exit, and our implementation does not contain any cyclic dependencies between retry-loops that exit with CAS, this means that the corresponding

Insert, *FindKey*, *DeleteKey*, *FindValue* or *DeleteValue* operation will progress. Consequently, independent of any number of concurrent operations, one operation will always progress. □

Theorem 3 *The algorithm implements a lock-free and linearizable dictionary.*

Proof: Following from Lemmas 16, 17, 18, 19, 20, 21, 22, 23, 24, 25, 26 and 27 and using the direct total order we can create an identical (with the same semantics) sequential execution that preserves the partial order of the operations in a concurrent execution. Following from Definition 9, the implementation is therefore linearizable. As the semantics of the operations are basically the same as in the skip list [91], we could use the corresponding proof of termination. This together with Lemma 28 and that the state is only changed at one atomic statement (Lemmas 10,16,17,20,24), gives that our implementation is lock-free. □

6.5 Experiments

We have performed experiments on both the limited set of operations on a dictionary (i.e. the *Insert*, *FindKey* and *DeleteKey* operations), as well as on the full set of operations on a dictionary (i.e. also including the *FindValue* and *DeleteValue* operations).

In our experiments with the limited set of operations on a dictionary, each concurrent thread performed 20000 sequential operations, whereof the first 50 up to 10000 of the totally performed operations are *Insert* operations, and the remaining operations was randomly chosen with a distribution of 1/3 *Insert* operations versus 1/3 *FindKey* and 1/3 *DeleteKey* operations. For the systems which also involves preemption, a synchronization barrier was performed between the initial insertion phaes and the remaining operations. The key values of the inserted nodes was randomly chosen between 0 and $1000000*n$, where n is the number of threads. Each experiment was repeated 50 times, and an average execution time for each experiment was estimated. Exactly the same sequential operations were performed for all different implementations compared. Besides our implementation, we also performed the same experiment with the lock-free implementation by Michael [77] which is the most recently claimed to be one of the most efficient concurrent dictionaries existing.

Our experiments with the full set of operations on a dictionary, was performed similarly to the experiments with the limited set of operations,

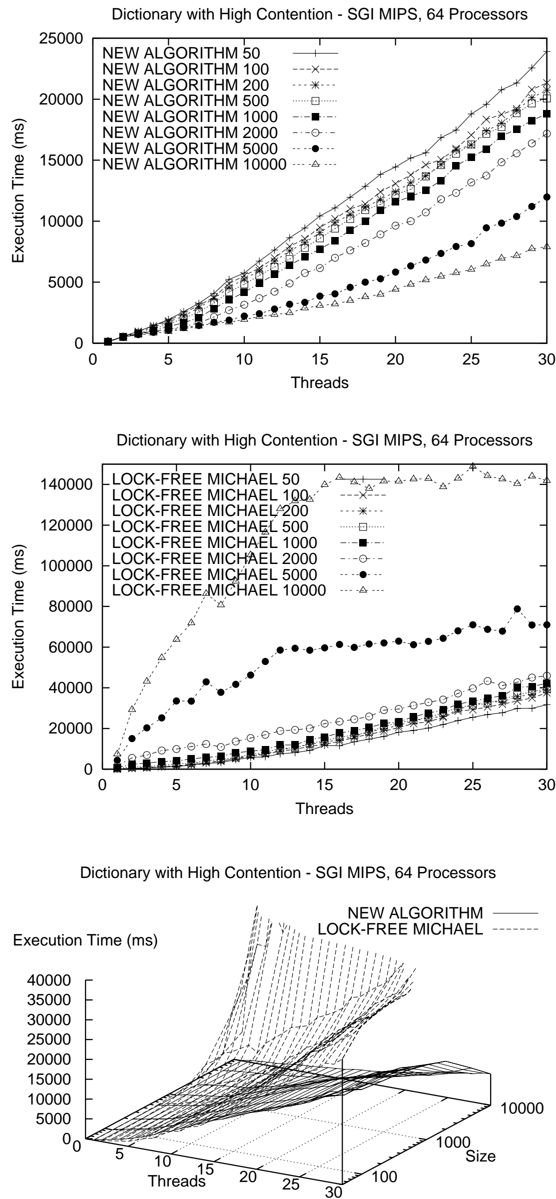


Figure 6.14: Experiment with dictionaries and high contention on SGI Origin 2000, initialized with 50,100,...,10000 nodes

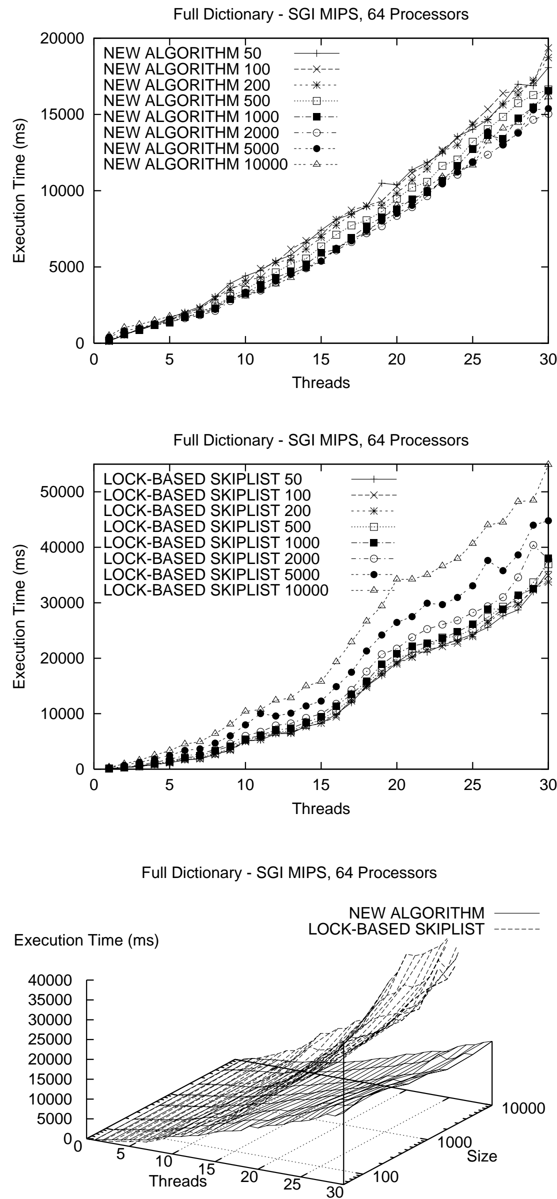


Figure 6.15: Experiment with full dictionaries and high contention on SGI Origin 2000, initialized with 50,100,...,10000 nodes

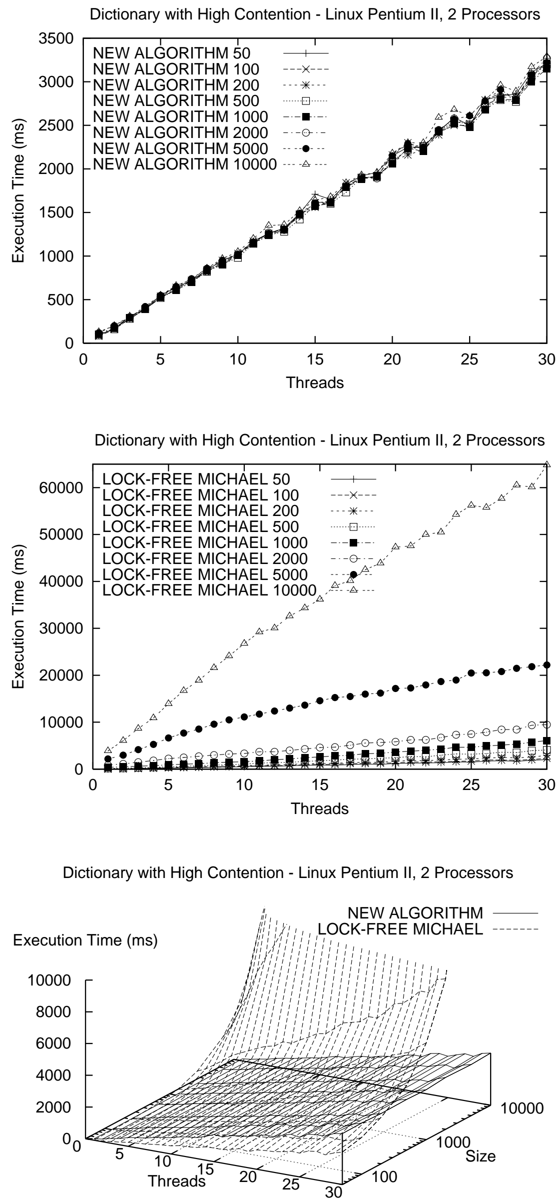


Figure 6.16: Experiment with dictionaries and high contention on Linux Pentium II, initialized with 50,100,...,10000 nodes

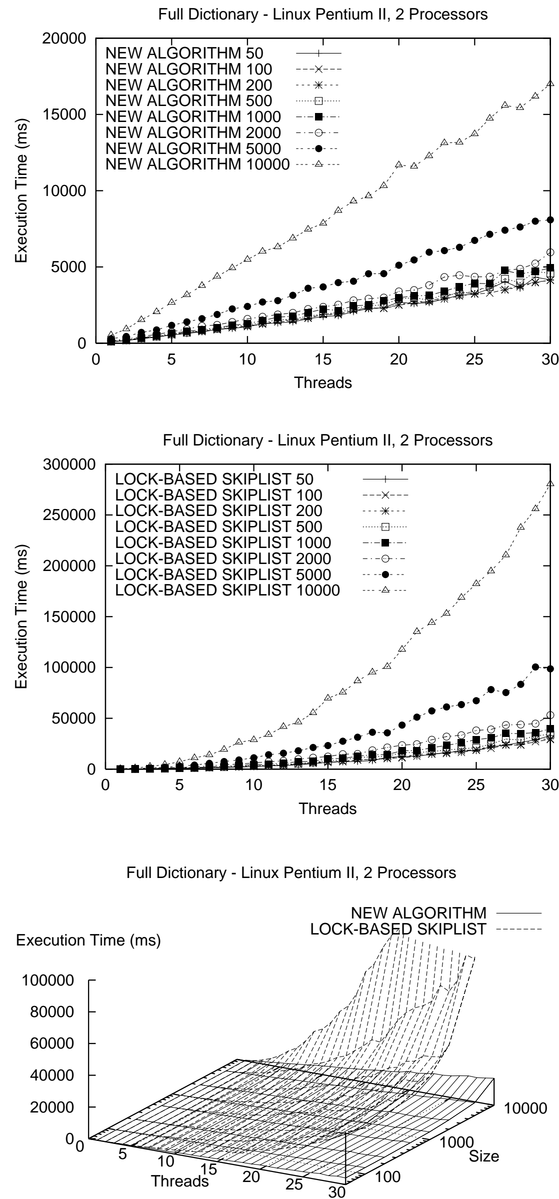


Figure 6.17: Experiment with full dictionaries and high contention on Linux Pentium II, initialized with 50,100,...,10000 nodes

except that the remaining operations after the insertion phase was randomly chosen with a distribution of 1/3 $Insert$ operations versus 15/48 $FindKey$, 15/48 $DeleteKey$, 1/48 $FindValue$ and 1/48 $DeleteValue$ operations. Each experiment was repeated 10 times. Besides our implementation, we also performed the same experiment with a lock-based implementation of skip lists using a single global lock.

The skip list based implementations have a fixed level of 10, which corresponds to an expected optimal performance with an average of 1024 nodes. All lock-based implementations are based on simple spin-locks using the TAS atomic primitive. A clean-cache operation was performed just before each sub-experiment using a different implementation. All implementations are written in C and compiled with the highest optimization level, except from the atomic primitives, which are written in assembler.

The experiments were performed using different number of threads, varying from 1 to 30. To get a highly pre-emptive environment, we performed our experiments on a Compaq dual-processor 450 MHz Pentium II PC running Linux. In order to evaluate our algorithm with full concurrency we also used a SGI Origin 2000 system running Irix 6.5 with 64 195 MHz MIPS R10000 processors. The results from these experiments are shown in Figures 6.14 and 6.16 . The average execution time is drawn as a function of the number of threads. Observe that the scale is different on each figure in order to clearify the experiments on the individual implementations as much as possible. For the SGI system and the limited set of operations, our lock-free algorithm shows a inversily proportional time complexity with respect to the size, though for the full set of operations the performance conforms to be averagely the same independently of the size. Our conjecture for this behavior is that the performance of the ccNUMA memory model of the SGI system increases significantly when the algorithm works on disjoint parts of the memory (as will occur with large sizes of the dictionary), while the time spent by the search phase of the operation will vary insignificantly because of the expected logarithmic time complexity. On the other hand, for the full set of operations, there will be corresponding performance degradation because of the linear time complexity for the value oriented operations. However, for the algorithm by Michael [77] the benefit for having disjoint access to the memory is insignificant compared to the performance degradation caused by the linear time complexity.

Our lock-free implementation scales best compared to the other implementation, having best performance for realistic sizes and any number of threads, i.e. for sizes larger or equal than 500 nodes, independently if the system is fully concurrent or involves a high degree of pre-emptions. On

scenarios with the full set of operations our algorithm performs better than the simple lock-based skip list for more than 3 threads on any system.

6.6 Related Work with Skip Lists

This paper describes an extension of the first[2] lock-free algorithm of a skip list data structure. Very similar constructions have appeared in the literature afterwards, by Fraser [32], Fomitchev [30] and Fomitchev and Ruppert [31]. As both Fraser's and Fomitchev's constructions appeared quite some time later in the literature than ours, it was not possible to compare them in our original publications. However, we have recently studied the other's approaches and have found some significant differences, although the main ideas are essentially the same. The differences are mainly related to performance issues:

- Compared to Fraser's approach, our skip list construction does not suffer from possible restarts of the full search phase from the head level when reaching a deleted node, as our nodes also contains a backlink pointer that is set at the time of deletion. This enables us to step one step backwards when reaching a deleted node, and to directly remove the deleted node. Both Fraser's and our construction uses arrays for remembering positions, though Fraser unnecessarily remembers also the successor on each level which could incur performance penalties through the garbage collector used.

- Compared to Fomitchev's and Fomitchev and Ruppert's approach, their construction does not use an array for remembering positions, which forces their construction to perform two full search phases when inserting or deleting nodes. In addition to have backlink pointers in order to be able to step back when reaching a deleted node, their construction also uses an extra pointer mark that is set (using an extra and expensive CAS operation) on the predecessor node in order to earlier notify concurrent operations of the helping duty. In our construction we only have one backlink pointer for all levels of a node, because of a performance trade-off between the usefulness for helping

[2]Our results were submitted for reviewing in October 2002 and published as a technical report [105] in January 2003. It was officially published in April 2003 [106], receiving a best paper award, and an extended version was also published in March 2004 [110]. Very similar constructions have appeared in the literature afterwards, by Fraser in February 2004 [32], Fomitchev in November 2003 [30] and Fomitchev and Ruppert in July 2004 [31]

operations and the cost that keeping extra pointers could incur for the garbage collection.

6.7 Conclusions

We have presented a lock-free algorithmic implementation of a concurrent dictionary. The implementation is based on the sequential skip list data structure and builds on top of it to support concurrency and lock-freedom in an efficient and practical way. Compared to the previous attempts to use skip lists for building concurrent dictionaries our algorithm is lock-free and avoids the performance penalties that come with the use of locks. Compared to the previous non-blocking concurrent dictionary algorithms, our algorithm inherits and carefully retains the basic design characteristic that makes skip lists practical: logarithmic search time complexity. Previous non-blocking algorithms did not perform well on dictionaries with realistic sizes because of their linear or worse search time complexity. Our algorithm also implements the full set of operations that is needed in a practical setting.

An interesting future work would be to investigate if it is suitable and how to change the skip list level reactively to the current average number of nodes. Another issue is how to choose and change the lengths of the fast jumps in order to get maximum performance of the *FindValue* and *DeleteValue* operations.

We compared our algorithm with the most efficient non-blocking implementation of dictionaries known. Experiments show that our implementation scales well, and for realistic number of nodes our implementation outperforms the other implementation, for all cases on both fully concurrent systems as well as with pre-emption.

We believe that our implementation is of highly practical interest for multi-threaded applications.

Chapter 7

Lock-Free and Practical Deques and Doubly Linked Lists using Single-Word Compare-And-Swap[1]

Håkan Sundell, Philippas Tsigas
Department of Computing Science
Chalmers Univ. of Technol. and Göteborg Univ.
412 96 Göteborg, Sweden
E-mail: {phs, tsigas}@cs.chalmers.se

Abstract

We present an efficient and practical lock-free implementation of a concurrent deque that supports parallelism for disjoint accesses and uses atomic primitives which are available in modern computer systems. Previously known lock-free algorithms of deques are either based on non-available atomic synchronization primitives, only implement a subset of the functionality, or are not designed for disjoint accesses. Our algorithm is based on a general lock-free doubly linked list, and only requires single-word compare-and-swap

[1] This is a revided and extended version of the paper that appeared as a technical report [109].

atomic primitives. It also allows pointers with full precision, and thus supports dynamic deque sizes. We have performed an empirical study using full implementations of the most efficient known algorithms of lock-free deques. For systems with low concurrency, the algorithm by Michael shows the best performance. However, as our algorithm is designed for disjoint accesses, it performs significantly better on systems with high concurrency and non-uniform memory architecture. In addition, the proposed solution also implements a general doubly linked list, the first lock-free implementation that only needs the single-word compare-and-swap atomic primitive.

7.1 Introduction

A deque (i.e. double-ended queue) is a fundamental data structure. For example, deques are often used for implementing the ready queue used for scheduling of tasks in operating systems. A deque supports four operations, the *PushRight*, the *PopRight*, the *PushLeft*, and the *PopLeft* operation. The abstract definition of a deque is a list of values, where the *PushRight*/*PushLeft* operation adds a new value to the right/left edge of the list. The *PopRight*/*PopLeft* operation correspondingly removes and returns the value on the right/left edge of the list.

To ensure consistency of a shared data object in a concurrent environment, the most common method is mutual exclusion, i.e. some form of locking. Mutual exclusion degrades the system's overall performance [100] as it causes blocking, i.e. other concurrent operations can not make any progress while the access to the shared resource is blocked by the lock. Mutual exclusion can also cause deadlocks, priority inversion and even starvation.

In order to address these problems, researchers have proposed non-blocking algorithms for shared data objects. Non-blocking algorithms do not involve mutual exclusion, and therefore do not suffer from the problems that blocking could generate. Lock-free implementations are non-blocking and guarantee that regardless of the contention caused by concurrent operations and the interleaving of their sub-operations, always at least one operation will progress. However, there is a risk for starvation as the progress of some operations could cause some other operations to never finish. Wait-free [45] algorithms are lock-free and moreover they avoid starvation as well, as all operations are then guaranteed to finish in a limited number of their own steps. Recently, some researchers also include obstruction-free [48] implementations to the non-blocking set of implementations. These kinds of implementations are weaker than the lock-free ones and do not guarantee

progress of any concurrent operation.

The implementation of a lock-based concurrent deque is a trivial task, and can preferably be constructed using either a doubly linked list or a cyclic array, protected by either a single lock or by multiple locks where each lock protects a part of the shared data structure. To the best of our knowledge, there exists no implementations of wait-free deques, but several lock-free implementations have been proposed. However, all previously lock-free deques lack in several important aspects, as they either only implement a subset of the operations that are normally associated with a deque and have concurrency restrictions[2] like Arora et al. [15], or are based on atomic hardware primitives like Double-Word Compare-And-Swap (CAS2)[3] which is not available in modern computer systems. Greenwald [37] presented a CAS2-based deque implementation as well as a general doubly linked list implementation [38], and there is also a publication series of a CAS2-based deque implementation [3],[28] with the latest version by Martin et al. [74]. Valois [122] sketched out an implementation of a lock-free doubly linked list structure using Compare-And-Swap (CAS)[4], though without any support for deletions and is therefore not suitable for implementing a deque. Michael [79] has developed a deque implementation based on CAS. However, it is not designed for allowing parallelism for disjoint accesses as all operations have to synchronize, even though they operate on different ends of the deque. Secondly, in order to support dynamic maximum deque sizes it requires an extended CAS operation that can atomically operate on two adjacent words, which is not available[5] on all modern platforms.

In this paper we present a lock-free algorithm for implementing a concurrent deque that supports parallelism for disjoint accesses (in the sense that operations on different ends of the deque do not necessarily interfere with each other). The algorithm is implemented using common synchronization primitives that are available in modern systems. It allows pointers with full precision, and thus supports dynamic maximum deque sizes (in the presence of a lock-free dynamic memory handler with sufficient garbage collection support), still using normal CAS-operations. The algorithm is

[2]The algorithm by Arora et al. does not support push operations on both ends, and does not allow concurrent invocations of the push operation and a pop operation on the opposite end.

[3]A CAS2 operations can atomically read-and-possibly-update the contents of two non-adjacent memory words. This operation is also sometimes called DCAS in the literature.

[4]The standard CAS operation can atomically read-and-possibly-update the contents of a single memory word

[5]It is available on the Intel IA-32, but not on the Sparc or MIPS microprocessor architectures. It is neither available on any currently known and common 64-bit architecture.

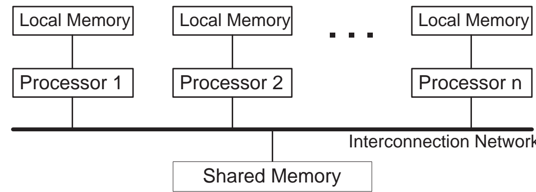


Figure 7.1: Shared Memory Multiprocessor System Structure

described in detail later in this paper, together with the aspects concerning the underlying lock-free memory management. In the algorithm description the precise semantics of the operations are defined and a proof that our implementation is lock-free and linearizable [50] is also given. We also give a detailed description of all the fundamental operations of a general doubly linked list data structure.

We have performed experiments that compare the performance of our algorithm with two of the most efficient algorithms of lock-free deques known; [79] and [74], the latter implemented using results from [27] and [43]. Experiments were performed on three different multiprocessor systems equipped with 2,4 or 29 processors respectively. All three systems used were running different operating systems and were based on different architectures. Our results show that the CAS-based algorithms outperforms the CAS2-based implementations[6] for any number of threads and any system. In non-uniform memory architectures with high contention our algorithm, because of its disjoint access property, performs significantly better than the algorithm in [79].

The rest of the paper is organized as follows. In Section 7.2 we describe the type of systems that our implementation is aiming for. The actual algorithm is described in Section 7.3. In Section 7.4 we define the precise semantics for the operations on our implementation, and show the correctness of our algorithm by proving the lock-free and linearizability properties. The experimental evaluation is presented in Section 7.5. In Section 7.6 we give the detailed description of the fundamental operations of a general doubly linked list. We conclude the paper with Section 7.7.

[6]The CAS2 operation was implemented in software, using either mutual exclusion or the results from [43], which presented an software CASn (CAS for n non-adjacent words) implementation.

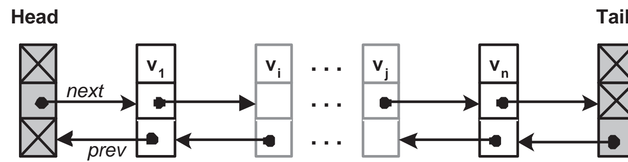


Figure 7.2: The doubly linked list data structure.

7.2 System Description

A typical abstraction of a shared memory multi-processor system configuration is depicted in Figure 7.1. Each node of the system contains a processor together with its local memory. All nodes are connected to the shared memory via an interconnection network. A set of co-operating tasks is running on the system performing their respective operations. Each task is sequentially executed on one of the processors, while each processor can serve (run) many tasks at a time. The co-operating tasks, possibly running on different processors, use shared data objects built in the shared memory to co-ordinate and communicate. Tasks synchronize their operations on the shared data objects through sub-operations on top of a cache-coherent shared memory. The shared memory may not though be uniformly accessible for all nodes in the system; processors can have different access times on different parts of the memory.

7.3 The Algorithm

The algorithm is based on a doubly linked list data structure, see Figure 7.2. To use the data structure as a deque, every node contains a value. The fields of each node item are described in Figure 7.6 as it is used in this implementation. Note that the doubly linked list data structure always contains the static head and tail dummy nodes.

In order to make the doubly linked list construction concurrent and non-blocking, we are using two of the standard atomic synchronization primitives, Fetch-And-Add (FAA) and Compare-And-Swap (CAS). Figure 7.3 describes the specification of these primitives which are available in most modern platforms.

To insert or delete a node from the list we have to change the respective set of prev and next pointers. These have to be changed consistently, but

```
procedure FAA(address:pointer to word, number:integer)
    atomic do
        *address := *address + number;

function CAS(address:pointer to word, oldvalue:word,
  newvalue:word):boolean
    atomic do
        if *address = oldvalue then
            *address := newvalue;
            return true;
        else return false;
```

Figure 7.3: The Fetch-And-Add (FAA) and Compare-And-Swap (CAS) atomic primitives.

not necessarily all at once. Our solution is to treat the doubly linked list as being a singly linked list with auxiliary information in the prev pointers, with the next pointers being updated before the prev pointers. Thus, the next pointers always form a consistent singly linked list, but the prev pointers only give hints for where to find the previous node. This is possible because of the observation that a "late" non-updated prev pointer will always point to a node that is directly or some steps before the current node, and from that "hint" position it is always possible to traverse[7] through the next pointers to reach the directly previous node.

One problem, that is general for non-blocking implementations that are based on the singly linked list data structure, arises when inserting a new node into the list. Because of the linked list structure one has to make sure that the previous node is not about to be deleted. If we are changing the next pointer of this previous node atomically with the CAS operation, to point to the new node, and then immediately afterwards the previous node is deleted - then the new node will be deleted as well, as illustrated in Figure 7.4. There are several solutions to this problem. One solution is to use the CAS2 operation as it can change two pointers atomically, but this operation is not available in any modern multiprocessor system. A second solution is to insert auxiliary nodes [122] between every two normal nodes, and the latest method introduced by Harris [44] is to use a deletion mark. This deletion mark is updated atomically together with the next pointer.

[7] As will be shown later, we have defined the deque data structure in a way that makes it possible to traverse even through deleted nodes, as long as they are referenced in some way.

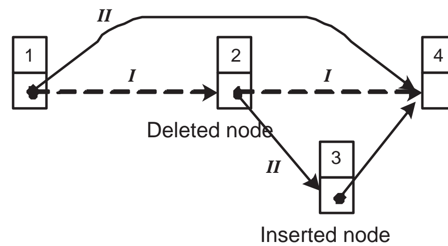


Figure 7.4: Concurrent insert and delete operation can delete both nodes.

Any concurrent insert operation will then be notified about the possibly set deletion mark, when its CAS operation will fail on updating the next pointer of the to-be-previous node. For our doubly linked list we need to be informed also when inserting using the prev pointer.

In order to allow usage of a system-wide dynamic memory handler (which should be lock-free and have garbage collection capabilities), all significant bits of an arbitrary pointer value must be possible to be represented in both the next and prev pointers. In order to atomically update both the next and prev pointer together with the deletion mark as done by Michael [79], the CAS-operation would need the capability of atomically updating at least $30+30+1 = 61$ bits on a 32-bit system (and $62+62+1 = 125$ bits on a 64-bit system as the pointers are then 64 bit). In practice though, most current 32 and 64-bit systems only support CAS operations of single word-size.

However, in our doubly linked list implementation, we never need to change both the prev and next pointers in one atomic update, and the pre-condition associated with each atomic pointer update only involves the pointer that is changed. Therefore it is possible to keep the prev and next pointers in separate words, duplicating the deletion mark in each of the words. In order to preserve the correctness of the algorithm, the deletion mark of the next pointer should always be set first, and the deletion mark of the prev pointer should be assured to be set by any operation that have observed the deletion mark on the next pointer, before any other updating steps are performed. Thus, full pointer values can be used, still by only using standard CAS operations.

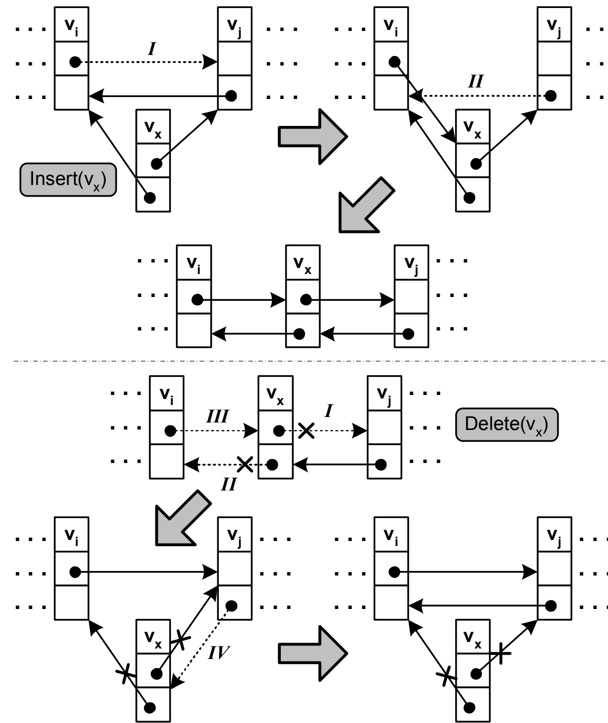


Figure 7.5: Illustration of the basic steps of the algorithms for insertion and deletion of nodes at arbitrary positions in the doubly linked list.

7.3.1 The Basic Steps of the Algorithm

The main algorithm steps, see Figure 7.5, for inserting a new node at an arbitrary position in our doubly linked list will thus be like follows: *I*) Atomically update the next pointer of the to-be-previous node, *II*) Atomically update the prev pointer of the to-be-next node. The main steps of the algorithm for deleting a node at an arbitrary position are the following: *I*) Set the deletion mark on the next pointer of the to-be-deleted node, *II*) Set the deletion mark on the prev pointer of the to-be-deleted node, *III*) Atomically update the next pointer of the previous node of the to-be-deleted node, *IV*) Atomically update the prev pointer of the next node of the to-be-deleted node. As will be shown later in the detailed description of the algorithm, helping techniques need to be applied in order to achieve the lock-free property, following the same steps as the main algorithm for inserting and deleting.

7.3.2 Memory Management

As we are concurrently (with possible preemptions) traversing nodes that will be continuously allocated and reclaimed, we have to consider several aspects of memory management. No node should be reclaimed and then later re-allocated while some other process is (or will be) traversing that node. For efficiency reasons we also need to be able to trust the prev and next pointers of deleted nodes, as we would otherwise be forced to re-start the traversing from the head or tail dummy nodes whenever reaching a deleted node while traversing and possibly incur severe performance penalties. This need is especially important for operations that try to help other delete operations in progress. Our demands on the memory management therefore rules out the SMR or ROP methods by Michael [78] and Herlihy et al. [47] respectively, as they can only guarantee a limited number of nodes to be safe via the hazard pointers, and these guarantees are also related to individual threads and never to an individual node structure. However, stronger memory management schemes as for example reference counting would be sufficient for our needs. There exists a general lock-free reference counting scheme by Detlefs et al. [27], though based on the non-available CAS2 atomic primitive.

For our implementation, we selected the lock-free memory management scheme invented by Valois [122] and corrected by Michael and Scott [83], which makes use of the FAA and CAS atomic synchronization primitives. Using this scheme we can assure that a node can only be reclaimed when there is no prev or next pointer in the list that points to it. One problem though with this scheme, a general problem with reference counting, is that it can not handle cyclic garbage (i.e. 2 or more nodes that should be recycled but reference each other, and therefore each node keeps a positive reference count, although they are not referenced by the main structure). Our solution is to make sure to break potential cyclic references directly before a node is possibly recycled. This is done by changing the next and prev pointers of a deleted node to point to active nodes, in a way that is consistent with the semantics of other operations.

The memory management scheme should also support means to de-reference pointers safely. If we simply de-reference a next or prev pointer using the means of the programming language, it might be that the corresponding node has been reclaimed before we could access it. It can also be that the deletion mark that is connected to the prev or next pointer was set, thus marking that the node is deleted. The scheme by Valois et al. supports lock-free pointer de-referencing and can easily be adopted to handle deletion

marks.

The following functions are defined for safe handling of the memory management:

function MALLOC_NODE() :**pointer to** Node
function READ_NODE(address:**pointer to** Link) :**pointer to** Node
function READ_DEL_NODE(address:**pointer to** Link) :**pointer to** Node
function COPY_NODE(node:**pointer to** Node) :**pointer to** Node
procedure RELEASE_NODE(node:**pointer to** Node)

The functions *READ_NODE* and *READ_DEL_NODE* atomically de-references the given link and increases the reference counter for the corresponding node. In case the deletion mark of the link is set, the *READ_NODE* function then returns NULL. The function *MALLOC_NODE* allocates a new node from the memory pool of pre-allocated nodes. The function *RELEASE_NODE* decrements the reference counter on the corresponding given node. If the reference counter reaches zero, the function then calls the *ReleaseReferences* function that will recursively call *RELEASE_NODE* on the nodes that this node has owned pointers to, and then it reclaims the node. The *COPY_NODE* function increases the reference counter for the corresponding given node.

As the details of how to efficiently apply the memory management scheme to our basic algorithm are not always trivial, we will provide a detailed description of them together with the detailed algorithm description in this section.

7.3.3 Pushing and Popping Nodes

The *PushLeft* operation, see Figure 7.7, inserts a new node at the leftmost position in the deque. The algorithm first repeatedly tries in lines L4-L14 to insert the new node (*node*) between the head node (*prev*) and the leftmost node (*next*), by atomically changing the next pointer of the head node. Before trying to update the next pointer, it assures in line L5 that the *next* node is still the very next node of head, otherwise *next* is updated in L6-L7. After the new node has been successfully inserted, it tries in lines P1-P13 to update the prev pointer of the next node. It retries until either i) it succeeds with the update, ii) it detects that either the next or new node is deleted, or iii) the next node is no longer directly next of the new node. In any of the two latter, the changes are due to concurrent Pop or Push operations, and the responsibility to update the prev pointer is then left to those. If the update succeeds, there is though the possibility that the new node was

```
union Link
    _: word
    ⟨p, d⟩: ⟨pointer to Node, boolean⟩

structure Node
    value: pointer to word
    prev: union Link
    next: union Link

// Global variables
head, tail: pointer to Node
// Local variables
node,prev,prev2,next,next2: pointer to Node
link1,lastlink: union Link

function CreateNode(value: pointer to word):pointer to Node
C1      node:=MALLOC_NODE();
C2      node.value:=value;
C3      return node;

procedure ReleaseReferences(node: pointer to Node)
RR1      RELEASE_NODE(node.prev.p);
RR2      RELEASE_NODE(node.next.p);
```

Figure 7.6: The basic algorithm details.

deleted (and thus the prev pointer of the *next* node was possibly already updated by the concurrent Pop operation) directly before the CAS in line P5, and then the prev pointer is updated by calling the *HelpInsert* function in line P10.

The *PushRight* operation, see Figure 7.8, inserts a new node at the rightmost position in the deque. The algorithm first repeatedly tries in lines R4-R13 to insert the new node (*node*) between the rightmost node (*prev*) and the tail node (*next*), by atomically changing the next pointer of the *prev* node. Before trying to update the next pointer, it assures in line R5 that the *next* node is still the very next node of *prev*, otherwise *prev* is updated by calling the *HelpInsert* function in R6, which updates the the prev pointer of the *next* node. After the new node has been successfully inserted, it tries in lines P1-P13 to update the prev pointer of the next node, following the same scheme as for the *PushLeft* operation.

The *PopLeft* operation, see Figure 7.9, tries to delete and return the

```
procedure PushLeft(value: pointer to word)
L1    node:=CreateNode(value);
L2    prev:=COPY_NODE(head);
L3    next:=READ_NODE(&prev.next);
L4    while true do
L5       if prev.next ≠ ⟨next,false⟩ then
L6          RELEASE_NODE(next);
L7          next:=READ_NODE(&prev.next);
L8          continue;
L9       node.prev:=⟨prev,false⟩;
L10      node.next:=⟨next,false⟩;
L11      if CAS(&prev.next,⟨next,false⟩,⟨node,false⟩) then
L12         COPY_NODE(node);
L13         break;
L14      Back-Off
L15   PushCommon(node,next);
```

Figure 7.7: The algorithm for the PushLeft operation.

value of the leftmost node in the deque. The algorithm first repeatedly tries in lines PL2-PL22 to mark the leftmost node (*node*) as deleted. Before trying to update the next pointer, it first assures in line PL4 that the deque is not empty, and secondly in line PL9 that the node is not already marked for deletion. If the deque was detected to be empty, the function returns. If *node* was marked for deletion, it tries to update the next pointer of the *prev* node by calling the *HelpDelete* function, and then *node* is updated to be the leftmost node. If the prev pointer of *node* was incorrect, it tries to update it by calling the *HelpInsert* function. After the node has been successfully marked by the successful CAS operation in line PL13, it tries in line PL14 to update the next pointer of the *prev* node by calling the *HelpDelete* function, and in line PL16 to update the prev pointer of the *next* node by calling the *HelpInsert* function. After this, it tries in line PL23 to break possible cyclic references that includes *node* by calling the *RemoveCrossReference* function.

The *PopRight* operation, see Figure 7.10, tries to delete and return the value of the rightmost node in the deque. The algorithm first repeatedly tries in lines PR2-PR19 to mark the rightmost node (*node*) as deleted. Before trying to update the next pointer, it assures i) in line PR4 that the node is not already marked for deletion, ii) in the same line that the prev pointer of the tail (*next*) node is correct, and iii) in line PR7 that the deque is not empty. If the deque was detected to be empty, the function returns. If *node* was marked for deletion or the prev pointer of the *next* node was

```
procedure PushRight(value: pointer to word)
R1      node:=CreateNode(value);
R2      next:=COPY_NODE(tail);
R3      prev:=READ_NODE(&next.prev);
R4      while true do
R5          if prev.next ≠ ⟨next,false⟩ then
R6              prev:=HelpInsert(prev,next);
R7              continue;
R8          node.prev:=⟨prev,false⟩;
R9          node.next:=⟨next,false⟩;
R10         if CAS(&prev.next,⟨next,false⟩,⟨node,false⟩) then
R11             COPY_NODE(node);
R12             break;
R13         Back-Off
R14     PushCommon(node,next);

procedure PushCommon(node, next: pointer to Node)
P1      while true do
P2          link1:=next.prev;
P3          if link1.d = true or node.next ≠ ⟨next,false⟩ then
P4              break;
P5          if CAS(&next.prev,link1,⟨node,false⟩) then
P6              COPY_NODE(node);
P7              RELEASE_NODE(link1.p);
P8              if node.prev.d = true then
P9                  prev2:=COPY_NODE(node);
P10                 prev2:=HelpInsert(prev2,next);
P11                 RELEASE_NODE(prev2);
P12             break;
P13         Back-Off
P14     RELEASE_NODE(next);
P15     RELEASE_NODE(node);
```

Figure 7.8: The algorithm for the PushRight operation.

```
function PopLeft(): pointer to word
PL1   prev:=COPY_NODE(head);
PL2   while true do
PL3       node:=READ_NODE(&prev.next);
PL4       if node = tail then
PL5           RELEASE_NODE(node);
PL6           RELEASE_NODE(prev);
PL7           return ⊥;
PL8       link1:=node.next;
PL9       if link1.d = true then
PL10          HelpDelete(node);
PL11          RELEASE_NODE(node);
PL12          continue;
PL13      if CAS(&node.next,link1,⟨link1.p,true⟩) then
PL14          HelpDelete(node);
PL15          next:=READ_DEL_NODE(&node.next);
PL16          prev:=HelpInsert(prev,next);
PL17          RELEASE_NODE(prev);
PL18          RELEASE_NODE(next);
PL19          value:=node.value;
PL20          break;
PL21      RELEASE_NODE(node);
PL22      Back-Off
PL23  RemoveCrossReference(node);
PL24  RELEASE_NODE(node);
PL25  return value;
```

Figure 7.9: The algorithm for the PopLeft function.

```
function PopRight(): pointer to word
PR1    next:=COPY_NODE(tail);
PR2    node:=READ_NODE(&next.prev);
PR3    while true do
PR4        if node.next ≠ ⟨next,false⟩ then
PR5            node:=HelpInsert(node,next);
PR6            continue;
PR7        if node = head then
PR8            RELEASE_NODE(node);
PR9            RELEASE_NODE(next);
PR10           return ⊥;
PR11       if CAS(&node.next,⟨next,false⟩,⟨next,true⟩) then
PR12           HelpDelete(node);
PR13           prev:=READ_DEL_NODE(&node.prev);
PR14           prev:=HelpInsert(prev,next);
PR15           RELEASE_NODE(prev);
PR16           RELEASE_NODE(next);
PR17           value:=node.value;
PR18           break;
PR19       Back-Off
PR20   RemoveCrossReference(node);
PR21   RELEASE_NODE(node);
PR22   return value;
```

Figure 7.10: The algorithm for the PopRight function.

incorrect, it tries to update the prev pointer of the *next* node by calling the *HelpInsert* function, and then *node* is updated to be the rightmost node. After the node has been successfully marked it follows the same scheme as the *PopLeft* operation.

7.3.4 Helping and Back-Off

The *HelpDelete* sub-procedure, see Figure 7.11, tries to set the deletion mark of the prev pointer and then atomically update the next pointer of the previous node of the to-be-deleted node, thus fulfilling step 2 and 3 of the overall node deletion scheme. The algorithm first ensures in line HD1-HD4 that the deletion mark on the prev pointer of the given node is set. It then repeatedly tries in lines HD8-HD34 to delete (in the sense of a chain of next pointers starting from the head node) the given marked node (*node*) by changing the next pointer from the previous non-marked node. First, we can safely assume that the next pointer of the marked node is always referring to a node (*next*) to the right and the prev pointer is always referring to a node (*prev*) to the left (not necessarily the first). Before trying to update the next pointer with the CAS operation in line HD30, it assures in line HD9 that *node* is not already deleted, in line HD10 that the *next* node is not marked, in line HD16 that the *prev* node is not marked, and in HD24 that *prev* is the previous node of *node*. If *next* is marked, it is updated to be the next node. If *prev* is marked we might need to delete it before we can update *prev* to one of its previous nodes and proceed with the current deletion, but in order to avoid unnecessary and even possibly infinite recursion, *HelpDelete* is only called if a next pointer from a non-marked node to *prev* has been observed (i.e. $lastlink.d$ is false). Otherwise if *prev* is not the previous node of *node* it is updated to be the next node.

The *HelpInsert* sub-function, see Figure 7.12, tries to update the prev pointer of a node and then return a reference to a possibly direct previous node, thus fulfilling step 2 of the overall insertion scheme or step 4 of the overall deletion scheme. The algorithm repeatedly tries in lines HI2-HI27 to correct the prev pointer of the given node (*node*), given a suggestion of a previous (not necessarily the directly previous) node (*prev*). Before trying to update the prev pointer with the CAS operation in line HI22, it assures in line HI4 that the *prev* node is not marked, in line HI13 that *node* is not marked, and in line HI16 that *prev* is the previous node of *node*. If *prev* is marked we might need to delete it before we can update *prev* to one of its previous nodes and proceed with the current insertion, but in order to avoid unnecessary recursion, *HelpDelete* is only called if a next pointer

```
procedure HelpDelete(node: pointer to Node)
HD1   while true do
HD2      link1:=node.prev;
HD3      if link1.d = true or
HD4        CAS(&node.prev,link1,⟨link1.p,true⟩) then break;
HD5   lastlink.d:=true;
HD6   prev:=READ_DEL_NODE(&node.prev);
HD7   next:=READ_DEL_NODE(&node.next);
HD8   while true do
HD9      if prev = next then break;
HD10     if next.next.d = true then
HD11        next2:=READ_DEL_NODE(&next.next);
HD12        RELEASE_NODE(next);
HD13        next:=next2;
HD14        continue;
HD15     prev2:=READ_NODE(&prev.next);
HD16     if prev2 = NULL then
HD17        if lastlink.d = false then
HD18           HelpDelete(prev);
HD19           lastlink.d:=true;
HD20        prev2:=READ_DEL_NODE(&prev.prev);
HD21        RELEASE_NODE(prev);
HD22        prev:=prev2;
HD23        continue;
HD24     if prev2 ≠ node then
HD25        lastlink.d:=false;
HD26        RELEASE_NODE(prev);
HD27        prev:=prev2;
HD28        continue;
HD29     RELEASE_NODE(prev2);
HD30     if CAS(&prev.next,⟨node,false⟩,⟨next,false⟩) then
HD31        COPY_NODE(next);
HD32        RELEASE_NODE(node);
HD33        break;
HD34     Back-Off
HD35  RELEASE_NODE(prev);
HD36  RELEASE_NODE(next);
```

Figure 7.11: The algorithm for the HelpDelete sub-operation.

```
function HelpInsert(prev, node: pointer to Node)
  :pointer to Node
HI1     lastlink.d:=true;
HI2     while true do
HI3         prev2:=READ_NODE(&prev.next);
HI4         if prev2 = NULL then
HI5             if lastlink.d = false then
HI6                 HelpDelete(prev);
HI7                 lastlink.d:=true;
HI8             prev2:=READ_DEL_NODE(&prev.prev);
HI9             RELEASE_NODE(prev);
HI10            prev:=prev2;
HI11            continue;
HI12        link1:=node.prev;
HI13        if link1.d = true then
HI14            RELEASE_NODE(prev2);
HI15            break;
HI16        if prev2 ≠ node then
HI17            lastlink.d:=false;
HI18            RELEASE_NODE(prev);
HI19            prev:=prev2;
HI20            continue;
HI21        RELEASE_NODE(prev2);
HI22        if CAS(&node.prev,link1,⟨prev,false⟩) then
HI23            COPY_NODE(prev);
HI24            RELEASE_NODE(link1.p);
HI25            if prev.prev.d = true then continue;
HI26            break;
HI27        Back-Off
HI28    return prev;
```

Figure 7.12: The algorithm for the HelpInsert sub-function.

from a non-marked node to *prev* has been observed (i.e. *lastlink.d* is false). If *node* is marked, the procedure is aborted. Otherwise if *prev* is not the previous node of *node* it is updated to be the next node. If the update in line HI22 succeeds, there is though the possibility that the *prev* node was deleted (and thus the prev pointer of *node* was possibly already updated by the concurrent Pop operation) directly before the CAS operation. This is detected in line HI25 and then the update is possibly retried with a new *prev* node.

Because the *HelpDelete* and *HelpInsert* are often used in the algorithm for "helping" late operations that might otherwise stop progress of other concurrent operations, the algorithm is suitable for pre-emptive as well as fully concurrent systems. In fully concurrent systems though, the helping strategy as well as heavy contention on atomic primitives, can downgrade the performance significantly. Therefore the algorithm, after a number of consecutive failed CAS operations (i.e. failed attempts to help concurrent operations) puts the current operation into back-off mode. When in back-off mode, the thread does nothing for a while, and in this way avoids disturbing the concurrent operations that might otherwise progress slower. The duration of the back-off is initialized to some value (e.g. proportional to the number of threads) at the start of an operation, and for each consecutive entering of the back-off mode during one operation invocation, the duration of the back-off is changed using some scheme, e.g. increased exponentially.

7.3.5 Avoiding Cyclic Garbage

The *RemoveCrossReference* sub-procedure, see Figure 7.13, tries to break cross-references between the given node (*node*) and any of the nodes that it references, by repeatedly updating the prev and next pointer as long as they reference a marked node. First, we can safely assume that the prev or next field of *node* is not concurrently updated by any other operation, as this procedure is only called by the main operation that deleted the node and both the next and prev pointers are marked and thus any concurrent update using CAS will fail. Before the procedure is finished, it assures in line RC3 that the previous node (*prev*) is not marked, and in line RC9 that the next node (*next*) is not marked. As long as *prev* is marked it is traversed to the left, and as long as *next* is marked it is traversed to the right, while continuously updating the prev or next field of *node* in lines RC5 or RC11.

```
procedure RemoveCrossReference(node: pointer to Node)
RC1    while true do
RC2        prev:=node.prev.p;
RC3        if prev.next.d = true then
RC4            prev2:=READ_DEL_NODE(&prev.prev);
RC5            node.prev:=⟨prev2,true⟩;
RC6            RELEASE_NODE(prev);
RC7            continue;
RC8        next:=node.next.p;
RC9        if next.next.d = true then
RC10           next2:=READ_DEL_NODE(&next.next);
RC11           node.next:=⟨next2,true⟩;
RC12           RELEASE_NODE(next);
RC13           continue;
RC14       break;
```

Figure 7.13: The algorithm for the RemoveCrossReference sub-operation.

7.4 Correctness Proof

In this section we present the correctness proof of our algorithm. We first prove that our algorithm is a linearizable one [50] and then we prove that it is lock-free. A set of definitions that will help us to structure and shorten the proof is first described in this section. We start by defining the sequential semantics of our operations and then introduce two definitions concerning concurrency aspects in general.

Definition 14 *We denote with Q_t the abstract internal state of a deque at the time t. $Q_t = [v_1, \ldots, v_n]$ is viewed as an list of values v, where $|Q_t| \geq 0$. The operations that can be performed on the deque are* PushLeft*(L),* PushRight*(R),* PopLeft*(PL) and* PopRight*(PR). The time t_1 is defined as the time just before the atomic execution of the operation that we are looking at, and the time t_2 is defined as the time just after the atomic execution of the same operation. In the following expressions that define the sequential semantics of our operations, the syntax is $S_1 : O_1, S_2$, where S_1 is the conditional state before the operation O_1, and S_2 is the resulting state after performing the corresponding operation:*

$$Q_{t_1} : \mathbf{L}(\mathbf{v_1}), Q_{t_2} = [v_1] + Q_{t_1} \tag{7.1}$$

$$Q_{t_1} : \mathbf{R}(\mathbf{v_1}), Q_{t_2} = Q_{t_1} + [v_1] \tag{7.2}$$

$$Q_{t_1} = \emptyset : \mathbf{PL}() = \bot, Q_{t_2} = \emptyset \tag{7.3}$$

$$Q_{t_1} = [v_1] + Q_1 : \mathbf{PL}() = \mathbf{v_1}, Q_{t_2} = Q_1 \tag{7.4}$$

$$Q_{t_1} = \emptyset : \mathbf{PR}() = \bot, Q_{t_2} = \emptyset \tag{7.5}$$

$$Q_{t_1} = Q_1 + [v_1] : \mathbf{PR}() = \mathbf{v_1}, Q_{t_2} = Q_1 \tag{7.6}$$

Definition 15 *In a global time model each concurrent operation Op "occupies" a time interval* $[b_{Op}, f_{Op}]$ *on the linear time axis* ($b_{Op} < f_{Op}$). *The precedence relation (denoted by '*$\rightarrow$*') is a relation that relates operations of a possible execution,* $Op_1 \rightarrow Op_2$ *means that* Op_1 *ends before* Op_2 *starts. The precedence relation is a strict partial order. Operations incomparable under* $\rightarrow$ *are called* overlapping. *The overlapping relation is denoted by* $\|$ *and is commutative, i.e.* $Op_1 \| Op_2$ *and* $Op_2 \| Op_1$. *The precedence relation is extended to relate sub-operations of operations. Consequently, if* $Op_1 \rightarrow Op_2$, *then for any sub-operations* op_1 *and* op_2 *of* Op_1 *and* Op_2, *respectively, it holds that* $op_1 \rightarrow op_2$. *We also define the direct precedence relation* $\rightarrow_d$, *such that if* $Op_1 \rightarrow_d Op_2$, *then* $Op_1 \rightarrow Op_2$ *and moreover there exists no operation* Op_3 *such that* $Op_1 \rightarrow Op_3 \rightarrow Op_2$.

Definition 16 *In order for an implementation of a shared concurrent data object to be* linearizable *[50], for every concurrent execution there should exist an equal (in the sense of the effect) and valid (i.e. it should respect the semantics of the shared data object) sequential execution that respects the partial order of the operations in the concurrent execution.*

Next we are going to study the possible concurrent executions of our implementation. First we need to define the interpretation of the abstract internal state of our implementation.

Definition 17 *The value v is* present *(*$\exists i.Q[i] = v$*) in the abstract internal state Q of our implementation, when there is a connected chain of next pointers (i.e. prev.next) from a* present *node (or the head node) in the doubly linked list that connects to a node that contains the value v, and this node is not marked as deleted (i.e. node.next.d=false).*

Definition 18 *The* decision point *of an operation is defined as the atomic statement where the result of the operation is finitely decided, i.e. independent of the result of any sub-operations after the decision point, the operation will have the same result. We define the* state-read point *of an operation to be the atomic statement where a sub-state of the priority queue is read, and this sub-state is the state on which the decision point depends. We also define the* state-change point *as the atomic statement where the operation changes the abstract internal state of the priority queue after it has passed the corresponding decision point.*

We will now use these points in order to show the existence and location in execution history of a point where the concurrent operation can be viewed as it occurred atomically, i.e. the *linearizability point*.

Lemma 29 *A* PushRight *operation ($R(v)$), takes effect atomically at one statement.*

Proof: The decision, state-read and state-change point for a *PushRight* operation which succeeds ($R(v)$), is when the CAS sub-operation in line R10 (see Figure 7.8) succeeds. The state of the deque was ($Q_{t_1} = Q_1$) directly before the passing of the decision point. The prev node was the very last present node as it pointed (verified by R5 and the CAS in R10) to the tail node directly before the passing of the decision point. The state of the deque directly after passing the decision point will be $Q_{t_2} = Q_1 + [v]$ as the next pointer of the prev node was changed to point to the new node which contains the value v. Consequently, the linearizability point will be the CAS sub-operation in line R10. □

Lemma 30 *A* PushLeft *operation ($L(v)$), takes effect atomically at one statement.*

Proof: The decision, state-read and state-change point for a *PushLeft* operation which succeeds ($L(v)$), is when the CAS sub-operation in line L11 (see Figure 7.7) succeeds. The state of the deque was ($Q_{t_1} = Q_1$) directly before the passing of the decision point. The state of the deque directly after passing the decision point will be $Q_{t_2} = [v] + Q_1$ as the next pointer of the head node was changed to point to the new node which contains the value v. Consequently, the linearizability point will be the CAS sub-operation in line L11. □

Lemma 31 *A* PopRight *operation which fails ($PR() = \bot$), takes effect atomically at one statement.*

Proof: The decision point for a *PopRight* operation which fails ($PR() = \bot$) is the check in line PR7. Passing of the decision point together with the verification in line PR4 gives that the next pointer of the head node must have been pointing to the tail node ($Q_{t_1} = \emptyset$) directly before the read sub-operation of the prev field in line PR2 or the next field in line HI3, i.e. the state-read point. Consequently, the linearizability point will be the read sub-operation in line PR2 or line HI3. □

Lemma 32 *A* PopRight *operation which succeeds ($PR() = v$), takes effect atomically at one statement.*

Proof: The decision point for a *PopRight* operation which succeeds ($PR() = v$) is when the CAS sub-operation in line PR11 succeeds. Passing of the decision point together with the verification in line PR4 gives that the next pointer of the to-be-deleted node must have been pointing to the tail node ($Q_{t_1} = Q_1 + [v]$) directly before the CAS sub-operation in line PR11, i.e. the state-read point. Directly after passing the CAS sub-operation (i.e. the state-change point) the to-be-deleted node will be marked as deleted and therefore not present in the deque ($Q_{t_2} = Q_1$). Consequently, the linearizability point will be the CAS sub-operation in line PR11. □

Lemma 33 *A* PopLeft *operation which fails ($PL() = \bot$), takes effect atomically at one statement.*

Proof: The decision point for a *PopLeft* operation which fails ($PL() = \bot$) is the check in line PL4. Passing of the decision point gives that the next pointer of the head node must have been pointing to the tail node ($Q_{t_1} = \emptyset$) directly before the read sub-operation of the next pointer in line PL3, i.e. the state-read point. Consequently, the linearizability point will be the read sub-operation of the next pointer in line PL3. □

Lemma 34 *A* PopLeft *operation which succeeds ($PL() = v$), takes effect atomically at one statement.*

Proof: The decision point for a *PopLeft* operation which succeeds ($PL() = v$) is when the CAS sub-operation in line PL13 succeeds. Passing of the decision point together with the verification in line PL9 gives that the next

pointer of the head node must have been pointing to the present to-be-deleted node ($Q_{t_1} = [v] + Q_1$) directly before the read sub-operation of the next pointer in line PL3, i.e. the state-read point. Directly after passing the CAS sub-operation in line PL13 (i.e. the state-change point) the to-be-deleted node will be marked as deleted and therefore not present in the deque ($\neg\exists i.Q_{t_2}[i] = v$). Unfortunately this does not match the semantic definition of the operation.

However, none of the other concurrent operations linearizability points is dependent on the to-be-deleted node's state as marked or not marked during the time interval from the state-read to the state-change point. Clearly, the linearizability points of Lemmas 29 and 30 are independent as the to-be-deleted node would be part (or not part if not present) of the corresponding Q_1 terms. The linearizability points of Lemmas 31 and 33 are independent, as those linearizability points depend on the head node's next pointer pointing to the tail node or not. Finally, the linearizability points of Lemma 32 as well as this lemma are independent, as the to-be-deleted node would be part (or not part if not present) of the corresponding Q_1 terms, otherwise the CAS sub-operation in line PL13 of this operation would have failed.

Therefore all together, we could safely interpret the to-be-deleted node to be not present already directly after passing the state-read point (($Q_{t_2} = Q_1$). Consequently, the linearizability point will be the read sub-operation of the next pointer in line PL3. □

Lemma 35 *When the deque is idle (i.e. no operations are being performed), all next pointers of present nodes are matched with a correct prev pointer from the corresponding present node (i.e. all linked nodes from the head or tail node are present in the deque).*

Proof: We have to show that each operation takes responsibility for that the affected prev pointer will finally be correct after changing the corresponding next pointer. After successfully changing the next pointer in the *PushLeft* (*PushRight*) in line L11 (R10) operation, the corresponding prev pointer is tried to be changed in line P5 repeatedly until i) it either succeeds, ii) either the next or this node is deleted as detected in line P3, iii) or a new node is inserted as detected in line P3. If a new node is inserted the corresponding *PushLeft* (*PushRight*) operation will make sure that the prev pointer is corrected. If either the next or this node is deleted, the corresponding *PopLeft* (*PopRight*) operation will make sure that the prev pointer is corrected. If the prev pointer was successfully changed it is possible that this node was

deleted before we changed the prev pointer of the next node. If this is detected in line P8, then the prev pointer of the next node is corrected by the *HelpInsert* function.

After successfully marking the to-be-deleted nodes in line PL13 (PR11), the *PopLeft* (*PopRight*) functions will make sure that the connecting next pointer of the prev node will be changed to point to the closest present node to the right, by calling the *HelpDelete* procedure in line PL14 (PR12). It will also make sure that the corresponding prev pointer of the next code will be corrected by calling the *HelpInsert* function in line PL16 (PR14).

The *HelpDelete* procedure will repeatedly try to change the next pointer of the prev node that points to the deleted node, until it either succeeds changing the next pointer in line HD30 or some concurrent *HelpDelete* already succeeded as detected in line HD9.

The *HelpInsert* procedure will repeatedly try to change the prev pointer of the node to match with the next pointer of the prev node, until it either succeeds changing the prev pointer in line HI22 or the node is deleted as detected in line HI13. If it succeeded with changing the prev pointer, the prev node has possibly been deleted directly before changing the prev pointer, and therefore it is detected if the prev node is marked in line HI25 and then the procedure will continue trying to correctly change the prev pointer. □

Lemma 36 *When the deque is idle, all previously deleted nodes are garbage collected.*

Proof: We have to show that each *PopRight* or *PopLeft* operation takes responsibility for that the deleted node will finally have no references to it. The possible references are caused by other nodes pointing to it. Following Lemma 35 we know that no present nodes will reference the deleted node. It remains to show that all paths of references from a deleted node will finally reference a present node, i.e. there are no cyclic referencing. After the node is deleted in lines PL14 and PL16 (PR12 and PR14), it is assured by the *PopLeft* (*PopRight*) operation by calling the *RemoveCrossReference* procedure in line PL23 (PR20) that both the next and prev pointers are pointing to a present node. If any of those present nodes are deleted before the referencing deleted node is garbage collected in line PL24 (PR21), the *RemoveCrossReference* procedures called by the corresponding *PopLeft* or *PopRight* operation will assure that the next and prev pointers of the previously present node will point to present nodes, and so on recursively. The *RemoveCrossReference* procedure repeatedly tries to change prev pointers to point to the previous node of the referenced node until the referenced node

is present, detected in line RC3 and possibly changed in line RC5. The next pointer is correspondingly detected in line RC9 and possibly changed in line RC11. □

Lemma 37 *The path of prev pointers from a node is always pointing a present node that is left of the current node.*

Proof: We will look at all possibilities where the prev pointer is set or changed. The setting in line L9 (R8) is clearly to the left as it is verified by L5 and L11 (R5 and R10). The change of the prev pointer in line P5 is to the left as verified by P3 and that nodes are never moved relatively to each other. The change of the prev pointer in line HI22 is to the left as verified by line HI3 and HI16. Finally, the change of the prev pointer in line RC5 is to the left as it is changed to the prev pointer of the previous node. □

Lemma 38 *All operations will terminate if exposed to a limited number of concurrent changes to the deque.*

Proof: The amount of changes an operation could experience is limited. Because of the reference counting, none of the nodes which are referenced to by local variables can be garbage collected. When traversing through prev or next pointers, the memory management guarantees atomicity of the operations, thus no newly inserted or deleted nodes will be missed. We also know that the relative positions of nodes that are referenced to by local variables will not change as nodes are never moved in the deque. Most loops in the operations retry because a change in the state of some node(s) was detected in the ending CAS sub-operation, and then retry by re-reading the local variables (and possibly correcting the state of the nodes) until no concurrent changes was detected by the CAS sub-operation and therefore the CAS succeeded and the loop terminated. Those loops will clearly terminate after a limited number of concurrent changes. Included in that type of loops are L4-L14, R4-R13, P1-P13, PL2-PL22 and PR3-PR19.

The loop HD8-HD34 will terminate if either the prev node is equal to the next node in line HD9 or the CAS sub-operation in line HD30 succeeds. From the start of the execution of the loop, we know that the prev node is left of the to-be-deleted node which in turn is left of the next node. Following from Lemma 37 this order will hold by traversing the prev node through its prev pointer and traversing the next node through its next pointer. Consequently, traversing the prev node through the next pointer will finally cause the prev node to be directly left of the to-be-deleted node if this is

not already deleted (and the CAS sub-operation in line HD30 will finally succeed), otherwise the prev node will finally be directly left of the next node (and in the next step the equality in line HD9 will hold). As long as the prev node is marked it will be traversed to the left in line HD20, and if it is the left-most marked node the prev node will be deleted by recursively calling *HelpDelete* in line HD18. If the prev node is not marked it will be traversed to the right. As there is a limited number of changes and thus a limited number of marked nodes left of the to-be-deleted node, the prev node will finally traverse to the right and either of the termination criteria will be fulfilled.

The loop HI2-HI27 will terminate if either the to-be-corrected node is marked in line HI13 or if the CAS sub-operation in line HI22 succeeds and prev node is not marked. From the start of the execution of the loop, we know that the prev node is left of the to-be-corrected node. Following from Lemma 37 this order will hold by traversing the prev node through its prev pointer. Consequently, traversing the prev node through the next pointer will finally cause the prev node to be directly left of the to-be-corrected node if this is not deleted (and the CAS sub-operation in line HI22 will finally succeed), otherwise line HI13 will succeed. As long as the prev node is marked it will be traversed to the left in line HI8, and if it is the left-most marked node the prev node will be deleted by calling *HelpDelete* in line HI6. If the prev node is not marked it will be traversed to the right. As there is a limited number of changes and thus a limited number of marked nodes left of the to-be-corrected node, the prev node will finally traverse to the right and either of the termination criteria will be fulfilled.

The loop RC1-RC14 will terminate if both the prev node and the next node of the to-be-deleted node is not marked in line RC3 respectively line RC9. We know that from the start of the execution of the loop, the prev node is left of the to-be-deleted node and the next node is right of the to-be-deleted node. Following from Lemma 37, traversing the prev node through the next pointer will finally reach a not marked node or the head node (which is not marked), and traversing the next node through the next pointer will finally reach a not marked node or the tail node (which is not marked), and both of the termination criteria will be fulfilled. □

Lemma 39 *With respect to the retries caused by synchronization, one operation will always do progress regardless of the actions by the other concurrent operations.*

Proof: We now examine the possible execution paths of our implementation. There are several potentially unbounded loops that can delay the termination of the operations. We call these loops retry-loops. If we omit the conditions that are because of the operations semantics (i.e. searching for the correct criteria etc.), the loop retries when sub-operations detect that a shared variable has changed value. This is detected either by a subsequent read sub-operation or a failed CAS. These shared variables are only changed concurrently by other CAS sub-operations. According to the definition of CAS, for any number of concurrent CAS sub-operations, exactly one will succeed. This means that for any subsequent retry, there must be one CAS that succeeded. As this succeeding CAS will cause its retry loop to exit, and our implementation does not contain any cyclic dependencies between retry-loops that exit with CAS, this means that the corresponding *PushRight*, *PushLeft*, *PopRight* or *PopLeft* operation will progress. Consequently, independent of any number of concurrent operations, one operation will always progress. □

Theorem 4 *The algorithm implements a correct, memory stable, lock-free and linearizable deque.*

Proof: Following from Lemmas 29, 30, 31, 32, 33 and 34 and by using the respective linearizability points, we can create an identical (with the same semantics) sequential execution that preserves the partial order of the operations in a concurrent execution. Following from Definition 16, the implementation is therefore linearizable.

Lemmas 38 and 39 give that our implementation is lock-free.

Following from Lemmas 38, 29, 30, 31, 32, 33 and 34 we can conclude that all operations will terminate with the correct result.

Following from Lemma 36 we know that the maximum memory usage will be proportional to the number of present values in the deque.

□

7.5 Experimental Evaluation

In our experiments, each concurrent thread performed 1000 randomly chosen sequential operations on a shared deque, with a distribution of 1/4 *PushRight*, 1/4 *PushLeft*, 1/4 *PopRight* and 1/4 *PopLeft* operations. Each experiment was repeated 50 times, and an average execution time for each experiment was estimated. Exactly the same sequence of operations was

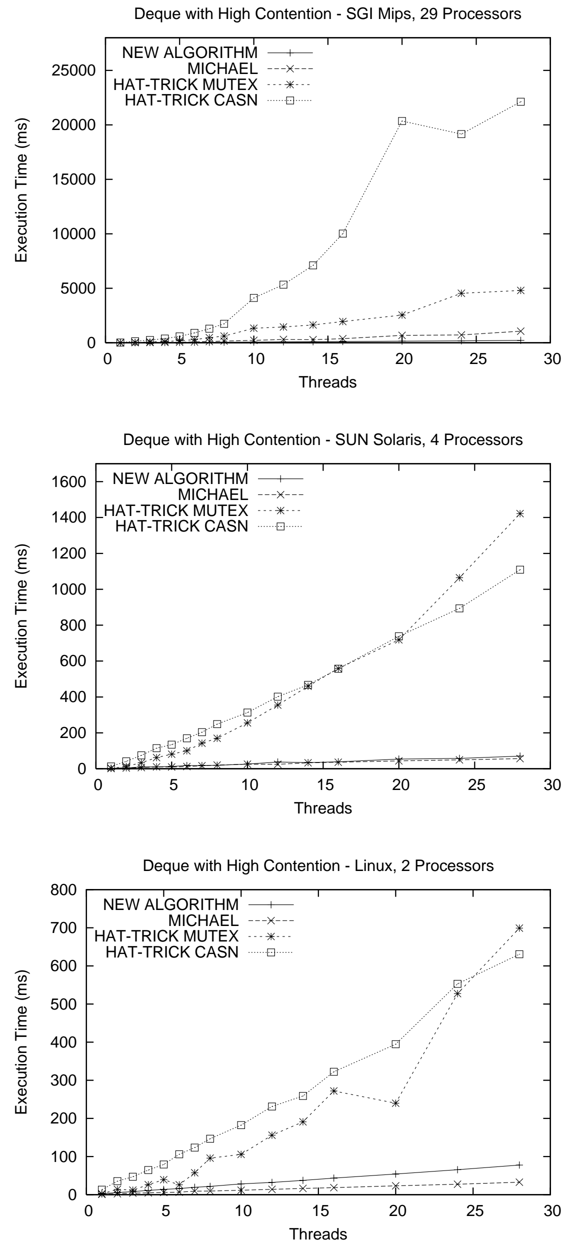


Figure 7.14: Experiment with deques and high contention.

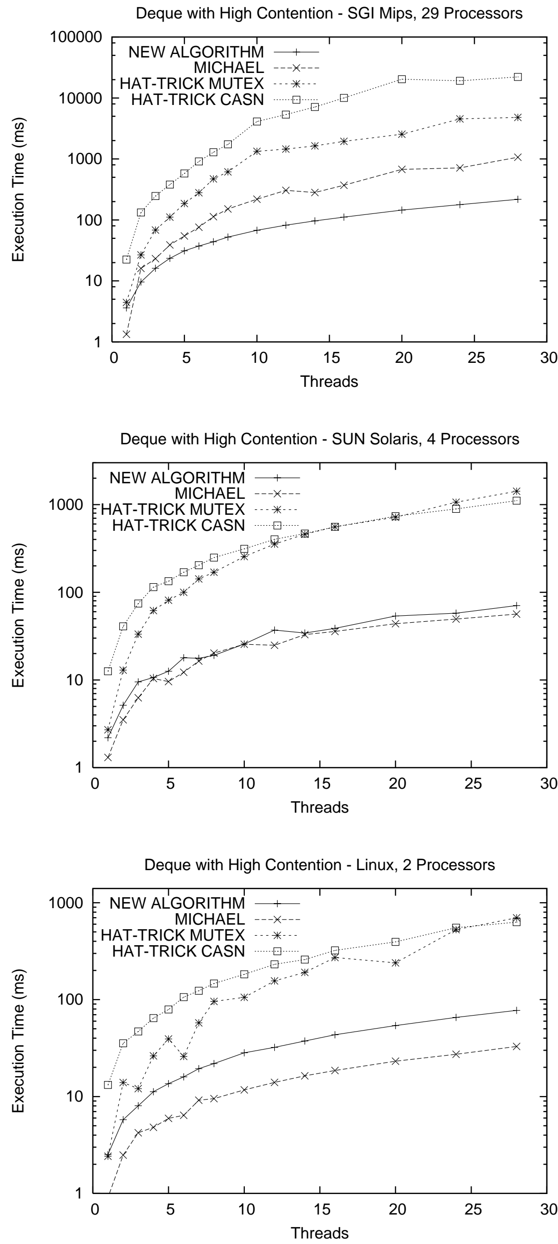


Figure 7.15: Experiment with deques and high contention, logarithmic scales.

performed for all different implementations compared. Besides our implementation, we also performed the same experiment with the lock-free implementation by Michael [79] and the implementation by Martin et al. [74], two of the most efficient lock-free deques that have been proposed. The algorithm by Martin et al. [74] was implemented together with the corresponding memory management scheme by Detlefs et al. [27]. However, as both [74] and [27] use the atomic operation CAS2 which is not available in any modern system, the CAS2 operation was implemented in software using two different approaches. The first approach was to implement CAS2 using mutual exclusion (as proposed in [74]), which should match the optimistic performance of an imaginary CAS2 implementation in hardware. The other approach was to implement CAS2 using one of the most efficient software implementations of CASN known that could meet the needs of [74] and [27], i.e. the implementation by Harris et al. [43].

A clean-cache operation was performed just before each sub-experiment using a different implementation. All implementations are written in C and compiled with the highest optimization level. The atomic primitives are written in assembly language.

The experiments were performed using different number of threads, varying from 1 to 28 with increasing steps. Three different platforms were used, with varying number of processors and level of shared memory distribution. To get a highly pre-emptive environment, we performed our experiments on a Compaq dual-processor Pentium II PC running Linux, and a Sun Ultra 80 system running Solaris 2.7 with 4 processors. In order to evaluate our algorithm with full concurrency we also used a SGI Origin 2000 system running Irix 6.5 with 29 250 MHz MIPS R10000 processors. The results from the experiments are shown in Figure 7.14. The average execution time is drawn as a function of the number of threads.

Our results show that both the CAS-based algorithms outperform the CAS2-based implementations for any number of threads. For the systems with low or medium concurrency and uniform memory architecture, [79] has the best performance. However, for the system with full concurrency and non-uniform memory architecture our algorithm performs significantly better than [79] from 2 threads and more, as a direct consequence of the nature of our algorithm to support parallelism for disjoint accesses.

7.6 General Operations for a Lock-Free Doubly Linked List

In this section we provide the details for the general operations of a lock-free doubly linked list, i.e. traversing the data structure in any direction and inserting and deleting nodes at arbitrary positions. Note that the linearizability points for these operations are defined without respect to the deque operations[8]. For maintaining the current position we adopt the *cursor* concept by Valois [122], that is basically just a reference to a node in the list.

In order to be able to traverse through deleted nodes, we also have to define the position of deleted nodes that is consistent with the normal definition of position of active nodes for sequential linked lists.

Definition 19 *The* position *of a cursor that references a node that is present in the list is the referenced node. The position of a cursor that references a deleted node, is represented by the node that was directly to the next of the deleted node at the very moment of the deletion (i.e. the setting of the deletion mark). If that node is deleted as well, the position is equal to the position of a cursor referencing that node, and so on recursively. The actual position is then interpreted to be at an imaginary node directly previous of the representing node.*

The *Next* function, see Figure 7.16, tries to change the cursor to the next position relative to the current position, and returns the status of success. The algorithm repeatedly in line NT2-NT11 checks the next node for possible traversal until the found node is present and is not the tail dummy node. If the current node is the tail dummy node, false is returned in line NT2. In line NT3 the next pointer of the current node is de-referenced and in line NT4 the deletion state of the found node is read. If the found node is deleted and the current node was deleted when directly next of the found node, this is detected in line NT5 and then the position is updated according to Definition 19 in line NT10. If the found node was detected as present in line NT5, the cursor is set to the found node in line NT10 and true is returned (unless the found node is the tail dummy node when instead false is returned) in line NT11. Otherwise it is checked if the found

[8]The general doubly linked list operation and the deque operations are compatible in the respect that the underlying data structure will be consistent. However, the linearizability point of the *PopLeft* operation is only defined with respect to the other deque operations and not with respect to the genaral doubly linked list operations.

```
function Next(cursor: pointer to pointer to Node): boolean
NT1   while true do
NT2      if *cursor = tail then return false;
NT3      next:=READ_DEL_NODE(&(*cursor).next);
NT4      d := next.next.d;
NT5      if d = true and (*cursor).next ≠ ⟨next,true⟩ then
NT6         if (*cursor).next.p = next then HelpDelete(next);
NT7         RELEASE_NODE(next);
NT8         continue;
NT9      RELEASE_NODE(*cursor);
NT10     *cursor:=next;
NT11     if d = false and next ≠ tail then return true;
```

Figure 7.16: The algorithm for the Next operation.

node is not already fully deleted in line NT6 and then fulfils the deletion by calling the *HelpDelete* procedure after which the algorithm retries at line NT2. The linearizability point of a *Next* function that succeeds is the read sub-operation of the next pointer in line NT3. The linearizability point of a *Next* function that fails is line NT2 if the node positioned by the original cursor was the tail dummy node, and the read sub-operation of the next pointer in line NT3 otherwise.

The *Prev* function, see Figure 7.17, tries to change the cursor to the previous position relative to the current position, and returns the status of success. The algorithm repeatedly in line PV2-PV11 checks the next node for possible traversal until the found node is present and is not the head dummy node. If the current node is the head dummy node, false is returned in line PV2. In line PV3 the prev pointer of the current node is de-referenced. If the found node is directly previous of the current node and the current node is present, this is detected in line PV4 and then the cursor is set to the found node in line PV6 and true is returned (unless the found node is the head dummy node when instead false is returned) in line PV7. If the current node is deleted then the cursor position is updated according to Definition 19 by calling the *Next* function in line PV8. Otherwise the prev pointer of the current node is updated by calling the *HelpInsert* function in line PV10 after which the algorithm retries at line PV2. The linearizability point of a *Prev* function that succeeds is the read sub-operation of the prev pointer in line PV3. The linearizability point of a *Prev* function that fails is line PV2 if the node positioned by the original cursor was the head dummy

```
function Prev(cursor: pointer to pointer to Node): boolean
PV1   while true do
PV2     if *cursor = head then return false;
PV3     prev:=READ_DEL_NODE(&(*cursor).prev);
PV4     if prev.next = ⟨*cursor,false⟩ and (*cursor).next.d = false then
PV5       RELEASE_NODE(*cursor);
PV6       *cursor:=prev;
PV7       if prev ≠ head then return true;
PV8     else if (*cursor).next.d = true then Next(cursor);
PV9     else
PV10      prev:=HelpInsert(prev,*cursor);
PV11      RELEASE_NODE(prev);
```

Figure 7.17: The algorithm for the Prev operation.

```
function Read(cursor: pointer to pointer to Node): pointer to word
RD1   if *cursor = head or *cursor = tail then return ⊥;
RD2   value:=(*cursor).value;
RD3   if (*cursor).next.d = true then return ⊥;
RD4   return value;
```

Figure 7.18: The algorithm for the Read function.

node, and the read sub-operation of the prev pointer in line PV3 otherwise.

The *Read* function, see Figure 7.18, returns the current value of the node referenced by the cursor, unless this node is deleted or the node is equal to any of the dummy nodes when the function instead returns a non-value. In line RD1 the algorithm checks if the node referenced by the cursor is either the head or tail dummy node, and then returns a non-value. The value of the node is read in line RD2, and in line RD3 it is checked if the node is deleted and then returns a non-value, otherwise the value is returned in line RD4. The linearizability point of a *Read* function that returns a value is the read sub-operation of the next pointer in line RD3. The linearizability point of a *Read* function that returns a non-value is the read sub-operation of the next pointer in line RD3, unless the node positioned by the cursor was the head or tail dummy node when the linearizability point is line RD1.

The *InsertBefore* operation, see Figure 7.19, inserts a new node directly before the node positioned by the given cursor and later changes the cursor to position the inserted node. If the node positioned by the cursor is the head dummy node, the new node will be inserted directly after the head dummy

```
procedure InsertBefore(cursor: pointer to pointer to Node,
 value: pointer to word)
IB1    if *cursor = head then return InsertAfter(cursor,value);
IB2    node:=CreateNode(value);
IB3    while true do
IB4       if (*cursor).next.d = true then Next(cursor);
IB5       prev:=READ_DEL_NODE(&(*cursor).prev);
IB6       node.prev:=⟨prev,false⟩;
IB7       node.next:=⟨(*cursor),false⟩;
IB8       if CAS(&prev.next,⟨(*cursor),false⟩,⟨node,false⟩) then
IB9          COPY_NODE(node);
IB10         break;
IB11      if prev.next ≠ ⟨(*cursor),false⟩ then prev:=HelpInsert(prev,*cursor);
IB12      RELEASE_NODE(prev);
IB13      Back-Off
IB14   next:=(*cursor);
IB15   *cursor:=COPY_NODE(node);
IB16   node:=HelpInsert(node,next);
IB17   RELEASE_NODE(node);
IB18   RELEASE_NODE(next);
```

Figure 7.19: The algorithm for the InsertBefore operation.

node. The algorithm checks in line IB1 if the cursor position is equal to the head dummy node, and consequently then calls the *InsertAfter* operation to insert the new node directly after the head dummy node. The algorithm repeatedly tries in lines IB4-IB13 to insert the new node (*node*) between the previous node (*prev*) of the cursor and the cursor positioned node, by atomically changing the next pointer of the prev node to instead point to the new node. If the node positioned by the cursor is deleted this is detected in line IB4 and the cursor is updated by calling the *Next* function. If the update of the next pointer of the prev node by using the CAS operation in line IB8 fails, this is because either the prev node is no longer the directly previous node of the cursor positioned node, or that the cursor positioned node is deleted. If the prev node is no longer the directly previous node this is detected in line IB11 and then the *HelpInsert* function is called in order to update the prev pointer of the cursor positioned node. If the update using CAS in line IB8 succeeds, the cursor position is set to the new node in line IB15 and the prev pointer of the previous cursor positioned node is updated by calling the *HelpInsert* function in line IB16. The linearizability

```
procedure InsertAfter(cursor: pointer to pointer to Node,
  value: pointer to word)
IA1    if *cursor = tail then return InsertBefore(cursor,value);
IA2    node:=CreateNode(value);
IA3    while true do
IA4        next:=READ_DEL_NODE(&(*cursor).next);
IA5        node.prev:=⟨(*cursor),false⟩;
IA6        node.next:=⟨next,false⟩;
IA7        if CAS(&(*cursor).next,⟨next,false⟩,⟨node,false⟩) then
IA8            COPY_NODE(node);
IA9            break;
IA10       RELEASE_NODE(next);
IA11       if (*cursor).next.d = true then
IA12           RELEASE_NODE(node);
IA13           return InsertBefore(cursor,value);
IA14       Back-Off
IA15   *cursor:=COPY_NODE(node);
IA16   node:=HelpInsert(node,next);
IA17   RELEASE_NODE(node);
IA18   RELEASE_NODE(next);
```

Figure 7.20: The algorithm for the InsertAfter operation.

point of the *InsertBefore* operation is the successful CAS operation in line IB8, or equal to the linearizability point of the *InsertBefore* operation if that operation was called in line IB1.

The *InsertAfter* operation, see Figure 7.20, inserts a new node directly after the node positioned by the given cursor and later changes the cursor to position the inserted node. If the node positioned by the cursor is the tail dummy node, the new node will be inserted directly before the tail dummy node. The algorithm checks in line IA1 if the cursor position is equal to the tail dummy node, and consequently then calls the *InsertBefore* operation to insert the new node directly after the head dummy node. The algorithm repeatedly tries in lines IA4-IA14 to insert the new node (*node*) between the cursor positioned node and the next node (*next*) of the cursor, by atomically changing the next pointer of the cursor positioned node to instead point to the new node. If the update of the next pointer of the cursor positioned node by using the CAS operation in line IA7 fails, this is because either the next node is no longer the directly next node of the cursor positioned node, or that the cursor positioned node is deleted. If the cursor positioned

```
function Delete(cursor: pointer to pointer to Node): pointer to word
D1      if *cursor = head or *cursor = tail then return ⊥;
D2      while true do
D3          link1:=(*cursor).next;
D4          if link1.d = true then return ⊥;
D5          if CAS(&(*cursor).next,link1,⟨link1.p,true⟩) then
D6              HelpDelete(*cursor);
D7              prev:=COPY_NODE((*cursor).prev.p);
D8              prev:=HelpInsert(prev,link1.p);
D9              RELEASE_NODE(prev);
D10             value:=(*cursor).value;
D11             RemoveCrossReference(*cursor);
D12             return value;
```

Figure 7.21: The algorithm for the Delete function.

node is deleted, the operation to insert directly after the cursor position now becomes the problem of inserting directly before the node that represents the cursor position according to Definition 19. It is detected in line IA11 if the cursor positioned node is deleted and then it calls the InsertBefore operation in line IA13. If the update using CAS in line IA7 succeeds, the cursor position is set to the new node in line IA15 and the prev pointer of the previous cursor positioned node is updated by calling the *HelpInsert* function in line IA16. The linearizability point of the *InsertAfter* operation is the successful CAS operation in line IA7, or equal to the linearizability point of the *InsertAfter* operation if that operation was called in line IA1 or IA13.

The *Delete* operation, see Figure 7.21, tries to delete the non-dummy node referenced by the given cursor and returns the value if successful, otherwise a non-value is returned. If the cursor positioned node is equal to any of the dummy nodes this is detected in line D1 and a non-value is returned. The algorithm repeatedly tries in line D3-D5 to set the deletion mark of the next pointer of the cursor positioned node. If the deletion mark is already set, this is detected in line D4 and a non-value is returned. If the CAS operation in line D5 succeeds, the deletion process is completed by calling the *HelpDelete* procedure in line D6 and the *HelpInsert* function in line D8. In order to avoid possible problems with cyclic garbage the *RemoveCrossReference* procedure is called in line D11. The value of the deleted node is read in line D10 and the value returned in line D12. The linearizability point of a *Delete* function that returns a value is the successful CAS operation in

line D5. The linearizability point of a *Delete* function that returns a non-value is the the read sub-operation of the next pointer in line D3, unless the node positioned by the cursor was the head or tail dummy node when the linearizability point instead is line D1.

The remaining necessary functionality for initializing the cursor positions like *First()* and *Last()* can be trivially derived by using the dummy nodes. If an *Update()* functionality is necessary, this could easily be achieved by extending the value field of the node data structure with a deletion mark, and throughout the whole algorithm interpret the deletion state of the whole node using this mark when semantically necessary, in combination with the deletion marks on the next and prev pointers.

7.7 Conclusions

We have presented the first lock-free algorithmic implementation of a concurrent deque that has all the following features: i) it supports parallelism for disjoint accesses, ii) uses a fully described lock-free memory management scheme, iii) uses atomic primitives which are available in modern computer systems, and iv) allows pointers with full precision to be used, and thus supports dynamic deque sizes. In addition, the proposed solution also implements all the fundamental operations of a general doubly linked list data structure in a lock-free manner. The doubly linked list operations also support deterministic and well defined traversals through even deleted nodes, and are therefore suitable for concurrent applications of linked lists in practice.

We have performed experiments that compare the performance of our algorithm with two of the most efficient algorithms of lock-free deques known, using full implementations of those algorithms. The experiments show that our implementation performs significantly better on systems with high concurrency and non-uniform memory architecture.

We believe that our implementation is of highly practical interest for multi-processor applications. We are currently incorporating it into the NOBLE [104] library.

Chapter 8

Conclusions

The overall focus of our work has been on designing efficient and practical shared data structures with related non-blocking synchronization methods for use within concurrent programs. One sub-goal was to find applications of wait- and lock-free data structures to real-time systems.

We have studied how timing information available in real-time systems can be used to improve and simplify wait-free algorithms for shared data structures. Our first attempt was to simplify a wait-free algorithm for snapshots that do not use any strong atomic primitives. This is especially suitable for embedded real-time systems, as many of those platforms are built on simple processors which lack support for more advanced features like strong synchronization primitives. In addition to be more suitable for embedded systems, the evaluation of the resulting simplified algorithm also show significant improvements in performance compared to the previously known algorithms for snapshots that instead use strong atomic primitives.

Our next attempt on using timing information was to study a wait-free algorithm for a shared register. Many distributed real-time systems have limited support for shared memory or have inter-process communication based on message passing. The simplified version of the algorithm show significant improvements in the aspect of usability; the size of the register can be increased as a larger part can be utilized of the involved components needed for the algorithm.

We have been doing studies and implementation work with known lock- and wait-free data structures. Several of these data structures have been put together into a software library called NOBLE, together with the algorithms presented in this thesis. The functionality of the library is designed in a way to be easily accessible for non-experts in non-blocking protocols, and at the

same time efficient and portable. Experiments on multi-processor platforms show significant improvements for the non-blocking implementations. As most algorithms have different performance characteristics and benefits for different environments, the diversity of choices in the library for each data structure, enables the user to always select the implementations that best fits the current environment. The NOBLE library has been released to the public, and has gained interest from both academia and industry.

We have presented a lock-free algorithmic implementation of a concurrent priority queue. The implementation is based on the sequential skip list data structure and builds on top of it to support concurrency and lock-freedom in an efficient and practical way. Compared to the previous attempts to use skip lists for building concurrent priority queues our algorithm is lock-free and avoids the performance penalties that come with the use of locks. Compared to the previous lock-free/wait-free concurrent priority queue algorithms, our algorithm inherits and carefully retains the basic design characteristic that makes skip lists practical: simplicity. Previous lock-free/wait-free algorithms did not perform well because of their complexity; furthermore they were often based on atomic primitives that are not available in today's systems. We compared our algorithm with some of the most efficient implementations of priority queues known. Experiments show that our implementation scales well, and with 3 threads or more our implementation outperforms the corresponding lock-based implementations, for all cases on both fully concurrent systems as well as with pre-emption.

We have presented a lock-free algorithmic implementation of a concurrent dictionary. The implementation is based on the sequential skip list data structure and builds on top of it to support concurrency and lock-freedom in an efficient and practical way. Compared to the previous attempts to use skip lists for building concurrent dictionaries our algorithm is lock-free and avoids the performance penalties that come with the use of locks. Compared to the previous non-blocking concurrent dictionary algorithms, our algorithm inherits and carefully retains the basic design characteristic that makes skip lists practical; logarithmic search time complexity. Previous non-blocking algorithms did not perform well on dictionaries with realistic sizes because of their linear or worse search time complexity. Our algorithm also implements the full set of operations that is needed in a practical setting. We compared our algorithm with the most efficient non-blocking implementation of dictionaries known. Experiments show that our implementation scales well, and for realistic number of nodes our implementation outperforms the other implementation, for all cases on both fully concurrent systems as well as with pre-emption.

We have presented the first lock-free algorithmic implementation of a concurrent deque that has all the following features: i) it supports parallelism for disjoint accesses, ii) uses a fully described lock-free memory management scheme, iii) uses atomic primitives which are available in modern computer systems, and iv) allows pointers with full precision to be used, and thus supports dynamic maximum sizes. In addition, the proposed solution also implements all the fundamental operations of a general doubly linked list data structure in a lock-free manner. The doubly linked list operations also support deterministic and well defined traversals through even deleted nodes, and are therefore suitable for concurrent applications of linked lists in practice. We have performed experiments that compare the performance of our algorithm with two of the most efficient algorithms of lock-free deques known, using full implementations of those algorithms. The experiments show that our implementation performs significantly better on systems with high concurrency and non-uniform memory architecture.

In overall, we have showed that the use of non-blocking synchronization methods can significantly improve the performance of shared data structures. Moreover, although it is known to be very complex, we have showed that it is possible to design specific non-blocking algorithms for most common data structures and abstract data types, and that these algorithms can be implemented practically and efficiently on modern computer systems. We have also showed that non-blocking synchronization can be efficiently applied to real-time systems, with the focus on practical wait-free algorithms especially suitable for hard real-time systems.

The experiments performed on the selected multi-processor platforms in the form of micro-benchmarks, do not guarantee the performance outside the scope of those experiments. However, the showed significant performance benefits in the randomized and general experimental scenarios, together with the fact that non-blocking algorithms can significantly improve the scalability of parallel applications as showed by Tsigas and Zhang [115] [118], give strong hints for a good performance on real applications as well. For our lock-free skip list construction, we already know of two American companies that are incorporating our algorithms into their environment with so far very promising results.

Chapter 9

Future Work

Our success with using asymmetric pointer updates on our design of the doubly linked list as well as the skip list shared data structures, opens the question whether this approach can be used also on other advanced data structures. Asymmetric updates can possibly make it possible in general to avoid atomic updates of 2 or more non-adjacent memory words. The key idea is to only rely on the standard and commonly available compare-and-swap (CAS) atomic primitive for updating the part of the shared data structure that is absolutely necessary for achieving correctness and consensus in the form of linearizability. The data structures of focus are tree and graph structures in general. It would also of interest to research if it is possible to relax the consensus requirement of the subsequent updates of an asymmetric update, to replace the CAS updates that are not absolutely necessary for linearizability with write operations followed by verifications using read operations.

A significant part of the properties that influence the actual performance of real implementations of the shared data structures that are presented in this thesis, is connected with the use of the lock-free reference counting technique by Valois et al. [122]. Although the alternative garbage collection method of using hazard pointers by Michael [78][80] could achieve significantly better performance in the respect of garbage collection, it is not directly applicable to our presented shared data structure algorithms as it can not support reliable global pointers from within the data structure. A possible adaptation of our algorithm to make it compatible with hazard pointers would therefore force all operations to retry from static nodes whenever the operation approaches a part of the data structure that is put for possible garbage collection, and thus incur severe performance penalties

that are proportional to the level of concurrency. Moreover, it would also make the concept of reliable traversal through linked list structures impossible. An interesting aspect to research would be to possibly combine the methods of hazard pointers with reference counting, thus achieving a more efficient and practical garbage collection scheme that is generally applicable.

In the literature about non-blocking synchronization, besides from the impractical and inefficient general schemes, there are very few wait-free algorithms of advanced common data structures available that are dynamic in size. One very significant reason is the lack of available algorithms for wait-free garbage collection and dynamic memory management. Thus, an interesting research topic would be to design methods for wait-free reference counting, and also possibly wait-free free-lists for dynamic allocation. These kinds of building blocks would significantly improve and facilitate the process of designing wait-free data structures as such based on dynamically changing linked lists. Wait-free garbage collection schemes would possibly also be of great interest to hard real-time systems, as it would enable dynamic memory management to be used in those systems.

Even though it may be possible to design shared data structures that require multiple pointer updates, using only the standard common CAS primitive and rely on asymmetric updates, there are likely design cases when more powerful atomic primitives could be very helpful. Therefore, it would be of great importance to research new wait-free and more efficient multi-word compare-and-swap (CASN) primitives. Besides from being very useful in the design of advanced data structures like trees, the possibility of being able to do wait-free transactions of arbitrary lengths would possibly be of great interest to real-time systems and especially real-time database systems.

Bibliography

[1] S. V. Adve and K. Gharachorloo, "Shared memory consistency models: A tutorial," *Computer*, vol. 29, no. 12, pp. 66–76, 1996.

[2] Y. Afek, D. Dolev, H. Attiya, E. Gafni, M. Merritt, and N. Shavit, "Atomic snapshots of shared memory," in *Proceedings of the ninth annual ACM symposium on Principles of distributed computing*, 1990, pp. 1–13.

[3] O. Agesen, D. Detlefs, C. H. Flood, A. Garthwaite, P. Martin, N. Shavit, and G. L. Steele Jr., "DCAS-based concurrent deques," in *ACM Symposium on Parallel Algorithms and Architectures*, 2000, pp. 137–146.

[4] B. Allvin, A. Ermedahl, H. Hansson, M. Papatriantafilou, H. Sundell, and P. Tsigas, "Evaluating the performance of wait-free snapshots in real-time systems," in *SNART'99 Real Time Systems Conference*, Aug. 1999, pp. 196–207.

[5] B. Alpern, L. Carter, E. Feig, and T. Selker, "The uniform memory hierarchy model of computation," *Algorithmica*, no. 12, pp. 72–109, 1994.

[6] J. Anderson, "Composite registers," in *Proceedings of the ninth annual ACM symposium on Principles of distributed computing*, 1990, pp. 15–29.

[7] ——, "Composite registers," *Distributed Computing*, no. 6, pp. 141–154, 1993.

[8] ——, "Multi-writer composite registers," *Distributed Computing*, no. 7, pp. 175–195, 1994.

[9] J. Anderson, R. Jain, and K. Jeffay, "Efficient object sharing in quantum-based real-time systems," in *Proceedings of the 19th IEEE Real-Time Systems Symposium*, Dec. 1998, pp. 346–355.

[10] J. Anderson, R. Jain, and S. Ramamurthy, "Wait-free object-sharing schemes for real-time uniprocessors and multiprocessors," in *Proceedings of the 18th IEEE Real-Time Systems Symposium*, Dec. 1997, pp. 111–122.

[11] J. Anderson and S. Ramamurthy, "Using lock-free objects in hard real-time applications," in *Proceedings of the 14th Annual ACM Symposium on Principles of Distributed Computing*, Aug. 1995, p. 272.

[12] J. Anderson, S. Ramamurthy, and K. Jeffay, "Real-time computing with lock-free shared objects," *ACM Transactions on Computer Systems*, vol. 15, no. 2, pp. 134–165, May 1997.

[13] J. H. Anderson and M. Moir, "Universal constructions for large objects," in *Proceedings of the 9th International Workshop on Distributed Algorithms*. Springer-Verlag, 1995, pp. 168–182.

[14] ——, "Universal constructions for multi-object operations," in *Proceedings of the 14th Annual ACM Symposium on the Principles of Distributed Computing*, Aug. 1995.

[15] N. S. Arora, R. D. Blumofe, and C. G. Plaxton, "Thread scheduling for multiprogrammed multiprocessors," in *ACM Symposium on Parallel Algorithms and Architectures*, 1998, pp. 119–129.

[16] J. Aspnes and M. Herlihy, "Wait-free data structures in the asynchronous PRAM model," in *ACM Symposium on Parallel Algorithms and Architectures*, 2000, pp. 340–349.

[17] N. Audsley, A. Burns, R. Davis, K. Tindell, and A. Wellings, "Fixed priority pre-emptive scheduling: An historical perspective," *Real-Time Systems*, vol. 8, no. 2/3, pp. 129–154, 1995.

[18] T. Baker, "Stack-based scheduling on real-time processes," *Real-Time Systems*, vol. 3, no. 1, pp. 97–69, Mar. 1991.

[19] G. Barnes, "Wait-free algorithms for heaps," Computer Science and Engineering, University of Washington, Tech. Rep., Feb. 1992.

[20] ——, "A method for implementing lock-free shared-data structures," in *Proceedings of the fifth annual ACM symposium on Parallel algorithms and architectures*. ACM Press, 1993, pp. 261–270.

[21] G. Bollella, J. Gosling, B. Brosgol, P. Dibble, S. Furr, and M. Turnbull, *The Real-Time Specification for Java*. Addison Wesley, 2000.

[22] L. Boug, J. Gabarr, and X. Messeguer, "Concurrent AVL revisited: Self-balancing distributed search trees," LIP, ENS Lyon, Research Report RR95-45, 1995.

[23] A. Charlesworth, "Starfire: Extending the SMP envelope," *IEEE Micro*, Jan. 1998.

[24] A. Charlesworth, N. Aneshansley, M. Haakmeester, D. Drogichen, G. Gilbert, R. Williams, and A. Phelps, "The starfire SMP interconnect," in *Proceedings of the 1997 ACM/IEEE conference on Supercomputing (CDROM)*. ACM Press, 1997, pp. 1–20.

[25] J. Chen and A. Burns, "Loop-free asynchronous data sharing in multiprocessor real-time systems based on timing properties," in *Proceedings of the 6th International Conference on Real-Time Computing Systems and Applications (RTCSA 99)*, Nov. 1999.

[26] R. David, N. Merriam, and N. Tracey, "How embedded applications using an RTOS can stay within on-chip memory limits," in *Proc. of the Industrial Experience Session, the 12th Euromicro Conference on Real-Time Systems*, June 2000.

[27] D. Detlefs, P. Martin, M. Moir, and G. Steele Jr, "Lock-free reference counting," in *Proceedings of the 20th Annual ACM Symposium on Principles of Distributed Computing*, Aug. 2001.

[28] D. Detlefs, C. H. Flood, A. Garthwaite, P. Martin, N. Shavit, and G. L. Steele Jr., "Even better DCAS-based concurrent deques," in *International Symposium on Distributed Computing*, 2000, pp. 59–73.

[29] A. Ermedahl, H. Hansson, M. Papatriantafilou, and P. Tsigas, "Wait-free snapshots in real-time systems: Algorithms and their performance," in *Proceedings of the 5th International Conference on Real-Time Computing Systems and Applications (RTCSA '98)*, 1998, pp. 257–266.

[30] M. Fomitchev, "Lock-free linked lists and skip lists," Master's thesis, York University, Nov. 2003.

[31] M. Fomitchev and E. Ruppert, "Lock-free linked lists and skip lists," in *Proceedings of the twenty-third annual symposium on Principles of Distributed Computing*, July 2004, pp. 50–59.

[32] K. A. Fraser, "Practical lock-freedom," Ph.D. dissertation, University of Cambridge, Feb. 2004, technical Report 579.

[33] H. Gao, J. F. Groote, and W. H. Hesselink, "Almost wait-free resizable hashtables," in *18th International Parallel and Distributed Processing Symposium (IPDPS'04)*, Apr. 2004, p. 50a.

[34] A. Gidenstam, M. Papatriantafilou, and P. Tsigas, "Allocating memory in a lock-free manner," Computing Science, Chalmers University of Technology, Tech. Rep. 2004-04, 2004.

[35] C. Gong and J. Wing, "A library of concurrent objects and their proofs of correctness," Computer Science Department, Carnegie Mellon University, Tech. Rep. CMU-CS-90-151, July 1990.

[36] M. Grammatikakis and S. Liesche, "Priority queues and sorting for parallel simulation," *IEEE Transactions on Software Engineering*, vol. SE-26, no. 5, pp. 401–422, 2000.

[37] M. Greenwald, "Non-blocking synchronization and system design," Ph.D. dissertation, Stanford University, Palo Alto, CA, 1999.

[38] ——, "Two-handed emulation: how to build non-blocking implementations of complex data-structures using DCAS," in *Proceedings of the twenty-first annual symposium on Principles of distributed computing*. ACM Press, 2002, pp. 260–269.

[39] M. Greenwald and D. Cheriton, "The synergy between non-blocking synchronization and operating system structure," in *Proceedings of the Second Symposium on Operating System Design and Implementation*, 1996, pp. 123–136.

[40] P. H. Ha and P. Tsigas, "Reactive multi-word synchronization for multiprocessors," in *12th International Conference on Parallel Architectures and Compilation Techniques (PACT'03).*

[41] ——, "Reactive multi-word synchronization for multiprocessors," *Journal of Instruction-Level Parallelism*, vol. 6, Apr. 2004.

[42] E. Hagersten and M. Koster, "Wildfire: A scalable path for SMPs," in *Proceedings of the Fifth IEEE Symposium on HighPerformance Computer Architecture*, Feb. 1999, pp. 172–181.

[43] T. Harris, K. Fraser, and I. Pratt, "A practical multi-word compare-and-swap operation," in *Proceedings of the 16th International Symposium on Distributed Computing*, 2002.

[44] T. L. Harris, "A pragmatic implementation of non-blocking linked lists," in *Proceedings of the 15th International Symposium of Distributed Computing*, Oct. 2001, pp. 300–314.

[45] M. Herlihy, "Wait-free synchronization," *ACM Transactions on Programming Languages and Systems*, vol. 11, no. 1, pp. 124–149, Jan. 1991.

[46] ——, "A methodology for implementing highly concurrent data objects," *ACM Transactions on Programming Languages and Systems*, Nov. 1993.

[47] M. Herlihy, V. Luchangco, and M. Moir, "The repeat offender problem: A mechanism for supporting dynamic-sized, lock-free data structure," in *Proceedings of 16th International Symposium on Distributed Computing*, Oct. 2002.

[48] ——, "Obstruction-free synchronization: Double-ended queues as an example," in *Proceedings of the 23rd International Conference on Distributed Computing Systems*, 2003.

[49] M. Herlihy and J. Moss, "Transactional memory: Architectural support for highly concurrent data structures," Digital Equipment Corp., Cambridge Research Laboratory, Tech. Rep., Apr. 1992.

[50] M. Herlihy and J. Wing, "Linearizability: a correctness condition for concurrent objects," *ACM Transactions on Programming Languages and Systems*, vol. 12, no. 3, pp. 463–492, 1990.

[51] T. Herman and V. Damian-Iordache, "Space-optimal wait-free queues," in *Proceedings of the sixteenth annual ACM symposium on Principles of distributed computing.* ACM Press, 1997, p. 280.

[52] W. H. Hesselink and J. F. Groote, "Waitfree distributed memory management by create and read until deletion (CRUD)," CWI, Amsterdam, Tech. Rep. SEN-R9811, 1998.

[53] ——, "Wait-free concurrent memory management by create and read until deletion (CaRuD)," *Distributed Computing*, vol. 14, no. 1, pp. 31–39, Jan. 2001.

[54] G. Hunt, M. Michael, S. Parthasarathy, and M. Scott, "An efficient algorithm for concurrent priority queue heaps," *Information Processing Letters*, vol. 60, no. 3, pp. 151–157, Nov. 1996.

[55] A. Israeli and L. Rappoport, "Efficient wait-free implementation of a concurrent priority queue," in *Proceedings of the 7th International Workshop on Distributed Algorithms*, ser. Lecture Notes in Computer Science, vol. 725. Springer Verlag, Sept. 1993, pp. 1–17.

[56] ——, "Disjoint-access-parallel implementations of strong shared memory primitives," in *Proceedings of the thirteenth annual ACM symposium on Principles of Distributed Computing*, Aug. 1994.

[57] A. Israeli and A. Shaham, "Optimal multi-writer multi-reader atomic register," in *Proceedings of the eleventh annual ACM symposium on Principles of distributed computing.* ACM Press, 1992, pp. 71–82.

[58] P. Jayanti, "A complete and constant time wait-free implementation of cas from ll/sc and vice versa," in *DISC 1998*, 1998, pp. 216–230.

[59] P. Jayanti and S. Petrovic, "Efficient and practical constructions of LL/SC variables," in *Proceedings of the twenty-second annual symposium on Principles of distributed computing.* ACM Press, 2003, pp. 285–294.

[60] D. W. Jones, "Concurrent operations on priority queues," *Commun. ACM*, vol. 32, no. 1, pp. 132–137, 1989.

[61] A. Karlin, K. Li, M. Manasse, and S. Owicki, "Empirical studies of competitive spinning for a shared-memory multiprocessor," in *Proceedings of the 13th ACM Symposium on Operating Systems Principles*, Oct. 1991, pp. 41–55.

[62] S. Kelly-Bootle and B. Fowler, *68000, 68010, 68020 Primer.* Howard W. Sams & Co., 1985.

[63] L. M. Kirousis, P. Spirakis, and P. Tsigas, "Reading many variables in one atomic operation: Solutions with linear or sublinear complexity," *IEEE Transactions on Parallel and Distributed Systems*, vol. 5, no. 7, pp. 688–696, July 1994.

[64] H. Kopetz and J. Reisinger, "The non-blocking write protocol nbw: A solution to a real-time synchronization problem," in *Proc. of the 14th Real-Time Systems Symp.*, 1993, pp. 131–137.

[65] L. Lamport, “Concurrent reading and writing,” *Communications of the ACM*, vol. 20, no. 11, pp. 806–811, Nov. 1977.

[66] ——, “How to make a multiprocessor computer that correctly executes multiprocess programs,” *IEEE Transactions on Computers*, vol. C.-28, no. 9, pp. 690–691, 1979.

[67] ——, “A fast mutual exclusion algorithm,” *ACM Transactions on Computer Systems*, vol. 5, no. 1, pp. 1–11, 1987.

[68] R. P. LaRowe Jr, “Page placement for non-uniform memory access time (numa) shared memory multiprocessors,” Ph.D. dissertation, Duke University, Durham, North Carolina, 1991.

[69] A. Larsson, A. Gidenstam, P. H. Ha, M. Papatriantafilou, and P. Tsigas, “Multi-word atomic read/write registers on multiprocessor systems,” in *Proceedings of the 12th Annual European Symposium on Algorithms (ESA 2004)*, Sept. 2004, pp. 736–748.

[70] J. Laudon and D. Lenoski, “The SGI origin: A ccNUMA highly scalable server,” in *Proceedings of the 24th Annual International Symposium on Computer Architecture (ISCA-97)*, vol. 25, no. 2. ACM Press, 1997, pp. 241–251.

[71] J. Lehoczky, L. Sha, and J. Strosnider, “Aperiodic responsiveness in hard real-time environments,” in *IEEE Real-Time Systems Symposium*, 1987, pp. 262–270.

[72] I. Lotan and N. Shavit, “Skiplist-based concurrent priority queues,” in *Proceedings of the International Parallel and Distributed Processing Symposium 2000*. IEEE press, 2000, pp. 263–268.

[73] S. Lumetta and D. Culler, “Managing concurrent access for shared memory active messages,” in *Proceedings of the First Merged International Parallel Processing Symposium and Symposium on Parallel and Distributed Processing*.

[74] P. Martin, M. Moir, and G. Steele, “DCAS-based concurrent deques supporting bulk allocation,” Sun Microsystems, Tech. Rep. TR-2002-111, 2002.

[75] H. Massalin and C. Pu, “A lock-free multiprocessor OS kernel,” Computer Science Department, Columbia University, Tech. Rep. CUCS-005-91, June 1991.

[76] J. M. Mellor-Crummey and M. L. Scott, “Algorithms for scalable synchronization on shared-memory multiprocessors,” *ACM Transactions on Computer Systems*, vol. 9, no. 1, pp. 21–65, Feb. 1991.

[77] M. M. Michael, “High performance dynamic lock-free hash tables and list-based sets,” in *Proceedings of the 14th ACM Symposium on Parallel Algorithms and Architectures*, 2002, pp. 73–82.

[78] ——, "Safe memory reclamation for dynamic lock-free objects using atomic reads and writes," in *Proceedings of the 21st ACM Symposium on Principles of Distributed Computing*, 2002, pp. 21–30.

[79] ——, "CAS-based lock-free algorithm for shared deques," in *Proceedings of the 9th International Euro-Par Conference*, ser. Lecture Notes in Computer Science. Springer Verlag, Aug. 2003.

[80] ——, "Hazard pointers: Safe memory reclamation for lock-free objects," *IEEE Transactions on Parallel and Distributed Systems*, vol. 15, no. 8, Aug. 2004.

[81] ——, "Practical lock-free and wait-free LL/SC/VL implementations using 64-bit CAS," in *The 18th Annual Conference on Distributed Computing (DISC 2004)*, Oct. 2004.

[82] ——, "Scalable lock-free dynamic memory allocation," in *Proceedings of the 2004 ACM SIGPLAN Conference on Programming Language Design and Implementation*, June 2004, pp. 35–46.

[83] M. M. Michael and M. L. Scott, "Correction of a memory management method for lock-free data structures," Computer Science Department, University of Rochester, Tech. Rep., 1995.

[84] ——, "Simple, fast, and practical non-blocking and blocking concurrent queue algorithms," in *Proceedings of the fifteenth annual ACM symposium on Principles of distributed computing*. ACM Press, 1996, pp. 267–275.

[85] ——, "Relative performance of preemption-safe locking and non-blocking synchronization on multiprogrammed shared memory multiprocessors," in *Proceedings of the 11th International Parallel Processing Symposium*, 1997, pp. 267–273.

[86] ——, "Nonblocking algorithms and preemption-safe locking on multiprogrammed shared memory multiprocessors," *Journal of Parallel and Distributed Computing*, vol. 51, no. 1, pp. 1–26, 1998.

[87] M. Moir, "Practical implementations of non-blocking synchronization primitives," in *Proceedings of the 15th Annual ACM Symposium on the Principles of Distributed Computing*, Aug. 1997.

[88] ——, "Transparent support for wait-free transactions," in *Proceedings of the 11th International Workshop on Distributed Algorithms*, Sept. 1997.

[89] S. Prakash, Y. H. Lee, and T. Johnson, "Non-blocking algorithms for concurrent data structures," University of Florida, Tech. Rep. 91–002, July 1991.

[90] ——, "A nonblocking algorithm for shared queues using compare-and-swap," *IEEE Trans. Comput.*, vol. 43, no. 5, pp. 548–559, 1994.

[91] W. Pugh, "Skip lists: a probabilistic alternative to balanced trees," *Communications of the ACM*, vol. 33, no. 6, pp. 668–676, 1990.

[92] R. Rajkumar, "Real-time synchronization protocols for shared memory multiprocessors," in *Proceedings of the 10th International Conference on Distributed Computing Systems*, 1990, pp. 116–123.

[93] ——, *Synchronization in Real-Time Systems: A Priority Inheritance Approach*. Kluwer Academic Publishers, 1991.

[94] S. Ramamurthy, M. Moir, and J. Anderson, "Real-time object sharing with minimal system support," in *Proc. of the 15th Annual ACM Symp. on Principles of Distributed Computing*, May 1996, pp. 233–242.

[95] L. Sha, R. Rajkumar, and J. Lehoczky, "Priority inheritance protocols: An approach to real-time synchronization," *IEEE Transactions on Computers*, vol. 39, no. 9, pp. 1175–1185, Sept. 1990.

[96] O. Shalev and N. Shavit, "Split-ordered lists: lock-free extensible hash tables," in *Proceedings of the twenty-second annual symposium on Principles of distributed computing*. ACM Press, 2003, pp. 102–111.

[97] C.-H. Shann, T.-L. Huang, and C. Chen, "A practical nonblocking queue algorithm using compare-and-swap," in *Proceedings of the Seventh International Conference on Parallel and Distributed Systems*, 2000, pp. 470–475.

[98] N. Shavit and D. Touitou, "Software transactional memory," in *Proceedings of the fourteenth annual ACM symposium on Principles of distributed computing*. ACM Press, 1995, pp. 204–213.

[99] N. Shavit and A. Zemach, "Scalable concurrent priority queue algorithms," in *Symposium on Principles of Distributed Computing*, 1999, pp. 113–122.

[100] A. Silberschatz and P. Galvin, *Operating System Concepts*. Addison Wesley, 1994.

[101] P. Sorensen, "A methodology for real-time system development," Ph.D. dissertation, University of Toronto, 1974.

[102] P. Sorensen and V. Hemacher, "A real-time system design methodology," *INFOR*, vol. 13, no. 1, pp. 1–18, Feb. 1975.

[103] H. Sundell and P. Tsigas, "Space efficient wait-free buffer sharing in multiprocessor real-time systems based on timing information," in *Proceedings of the 7th International Conference on Real-Time Computing Systems and Applications (RTCSA 2000)*. IEEE press, 2000, pp. 433–440.

[104] ——, "NOBLE: A non-blocking inter-process communication library," in *Proceedings of the 6th Workshop on Languages, Compilers and Run-time Systems for Scalable Computers*, ser. Lecture Notes in Computer Science. Springer Verlag, 2002.

[105] ——, "Fast and lock-free concurrent priority queues for multi-thread systems," Computing Science, Chalmers University of Technology, Tech. Rep. 2003-01, Jan. 2003.

[106] ——, "Fast and lock-free concurrent priority queues for multi-thread systems," in *Proceedings of the 17th International Parallel and Distributed Processing Symposium*. IEEE press, Apr. 2003, p. 11.

[107] ——, "Scalable and lock-free concurrent dictionaries," Computing Science, Chalmers University of Technology, Tech. Rep. 2003-10, Dec. 2003.

[108] ——, "Simple wait-free snapshots for real-time systems with sporadic tasks," Computing Science, Chalmers University of Technology, Tech. Rep. 2003-02, Jan. 2003.

[109] ——, "Lock-free and practical deques using single-word compare-and-swap," Computing Science, Chalmers University of Technology, Tech. Rep. 2004-02, Mar. 2004.

[110] ——, "Scalable and lock-free concurrent dictionaries," in *Proceedings of the 19th ACM Symposium on Applied Computing*. ACM press, Mar. 2004, pp. 1438–1445.

[111] ——, "Simple wait-free snapshots for real-time systems with sporadic tasks," in *Proceedings of the 10th International Conference on Real-Time and Embedded Computing Systems and Applications*, Aug. 2004, pp. 325–340.

[112] H. Sundell, P. Tsigas, and Y. Zhang, "Simple and fast wait-free snapshots for real-time systems," in *Proceedings of the 4th International Conference On Principles Of Distributed Systems (OPODIS 2000)*, ser. Studia Informatica Universalis, 2000, pp. 91–106.

[113] H. Thane, "Asterix the t-rex among real-time kernels: Timely, reliable, efficient and extraordinary," Mälardalen Real-Time Research Centre, Mälardalen University, Tech. Rep., 2000, in preparation.

[114] P. Tsigas and Y. Zhang, "Non-blocking data sharing in multiprocessor real-time systems," in *Proceedings of the 6th International Conference on Real-Time Computing Systems and Applications (RTCSA'99)*. IEEE press, 1999, pp. 247–254.

[115] ——, "Evaluating the performance of non-blocking synchronization on shared-memory multiprocessors," in *Proceedings of the international conference on Measurement and modeling of computer systems*. ACM Press, 2001, pp. 320–321.

[116] ——, "A simple, fast and scalable non-blocking concurrent FIFO queue for shared memory multiprocessor systems," in *Proceedings of the 13th Annual ACM Symposium on Parallel Algorithms and Architectures (SPAA '01)*. ACM press, 2001, pp. 134–143.

[117] ——, "Efficient wait-free queue algorithms for real-time synchronization," Computing Science, Chalmers University of Technology, Tech. Rep. 2002-05, 2002.

[118] ——, “Integrating non-blocking synchronisation in parallel applications: Performance advantages and methodologies,” in *Proceedings of the 3rd ACM Workshop on Software and Performance.* ACM Press, 2002, pp. 55–67.

[119] ——, “Lock-free object-sharing for shared memory real-time multiprocessors,” Computing Science, Chalmers University of Technology, Tech. Rep. 2003-03, 2003.

[120] J. Turek, D. Shasha, and S. Prakash, “Locking without blocking: Making lock based concurrent data structure algorithms nonblocking,” in *Proceedings of the 11th ACM Symposium on Principles of Database Systems*, Aug. 1992, pp. 212–222.

[121] J. D. Valois, “Implementing lock-free queues,” in *Proceedings of the Seventh International Conference on Parallel and Distributed Computing Systems*, 1994, pp. 64–69.

[122] ——, “Lock-free data structures,” Ph.D. dissertation, Rensselaer Polytechnic Institute, Troy, New York, 1995.

[123] ——, “Lock-free linked lists using compare-and-swap,” in *Proceedings of the 14th Annual Principles of Distributed Computing*, 1995, pp. 214–222.

[124] P. Vitanyi and B. Awerbuch, “Atomic shared register access by asynchronous hardware,” in *27th IEEE Annual Symposium on Foundations of Computer Science*, Oct. 1986, pp. 233–243.

[125] J. Zahorjan, E. D. Lazowska, and D. L. Eager, “The effect of scheduling discipline on spin overhead in shared memory parallel systems,” *IEEE Transactions on Parallel and Distributed Systems*, vol. 2, no. 2, pp. 180–198, Apr. 1991.

Index

ABA problem, *5*, 20, 25
atomic, 4, 10, 15, 22, 43, 61, 64, 71, 72
atomic primitive, *4–6*, 10–14, 16–18, 25, 97, 171, 205, 209

back-off, *12*, 82, 84, 104, 185
blocking, *9*, 12, 41, 57, 58, 65, 74, 92, 132, 168

CAS, *5, 6*, 11, 13, 16, 19, 31, 33, 97, 136, 171, 209, 210
CAS2, *6*, 24–26, 34, 35, 98, 169, 175, 197
CASN, 19, 34, 35, 197, 210
concurrent, 1, 8, 27, 40, 93, 132, 146, 205
consensus number, *14*, 16

deadlock, *9*, 12, 41, 59, 92
deque, *25*, 33, 168, 207
dictionary, *26*, 132, 149, 159, 206
doubly linked list, *23*, 33, 169, 171, 174, 198, 207, 209

experiment, 37, 52, 83, 110, 159, 194, 206, 207

FAA, *5*, 21, 97, 136, 171
fault tolerant, 10, 41, 59

garbage collection, 19, *20*, 22, 34, 169, 173, 175, 191, 203, 209, 210

hash-table, *26*, 133
hazard pointer, 19, *20*, 24–26, 35, 175, 209
helping, *13*, 104, 185

IPCP, 9

linearizability, *10*, 44, 62, 68, 94, 104, 105, 149, 159, 186, 209
LL/SC, *5*, 16, 19
lock-based, 59, 75, 83, 84, 110
lock-free, *10, 11*, 12, *12*, 23, 31, 41, 59, 60, 74, 84, 92, 104, 132, 159, 168, 205–207

memory management, *19*, 27, 31, 32, 34, 77, 97, 98, 133, 136, 137, 140, 151, 175, 210
message passing, *1*, 9, 205
multi-processor, 1, 16, 17, 21, 40, 42, 57, 58, 60, 89, 95, 206
mutual exclusion, 4, *9*, 10, 18, 40, 57–59, 92, 132, 168

non-blocking, *10*, 11, 14–19, 22, 41, 57–60, 62, 74, 92, 132, 168, 172, 205, 207, 210
NUMA, *2*, 37, 207

obstruction-free, 10, 168

parallel, 1, 8, 74, 121
PCP, 9, 13

PIP, 58
pre-emption, 1, 15, 94, 98, 136, 146, 164, 206
priority inversion, *9*, 12, 40, 41, 58, 59, 92
priority queue, *26*, 31, 32, 92, 206
process, 1, 41, 43, 53

queue, *24*, 25, 76

real-time, 7, 25, 122
  embedded systems, 22, 205
  hard, *7*, 12, 28, 74, 207, 210
  operating system, 60
  soft, *7*
  system, *6*, 7, 9, 15–17, 22, 25, 27, 42, 57–60, 62, 64, 65, 69, 72, 127
reference counting, *20*, 35, 97, 136, 175, 209

schedulability, 58
shared data structure, *18*, 30, 40, 74, 205, 207, 209
shared memory, 1, *2*, 9, 18, 42, 60, 61, 69, 75, 95, 205
singly linked list, *23*, 76, 84, 137, 172
skip list, 27, 30, 31, 94, *96*, 110, 121, 127, 133, 135, 140, 164, 165, 206, 207, 209
snapshot, *22*, 30, 42, 43, 53, 76, 205
software library, 36, 75, 205
stack, *24*, 76, 82
starvation, 12, 92
synchronization, *8*, 40, 41, 74, 122, 205, 207

TAS, *5*, 16, 84, 110
task, *7*, 8, 40, 61, 95
  aperiodic, *7*
  periodic, *7*, 30, 46, 65, 122
  sporadic, *7*, 30, 46
thread, 1, 83, 104, 110, 206
time-stamp, 28–31, 60, 63, 122
timing information, 15, 17, 27, 53, 58, 60, 64, 72, 205

UMA, *2*, 37, 38, 53
uni-processor, 1, 15, 16, 41, 58

wait-free, *10*, *12*, *13*, 14, *14*, 17, 19, 20, 22, 24, 25, 30, 41, 53, 57, 59, 60, 72, 74, 93, 133, 168, 205, 207, 210