Thesis for the Degree of Doctor of Philosophy

Reactive Concurrent Data Structures and Algorithms for Synchronization

Phuong Ha

CHALMERS | GÖTEBORG UNIVERSITY

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

Göteborg, Sweden 2006

Reactive Concurrent Data Structures and Algorithms for Synchronization
Phuong Hoai Ha
ISBN 91-7291-780-6

Doktorsavhandlingar vid Chalmers tekniska högskola
Ny series nr 2462
ISSN 0346-718X

Technical Report no. 17D
ISSN 1651-4971
Department of Computer Science and Engineering

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

Chalmers Reproservice,
Göteborg, Sweden, 2006

Abstract

Parallelism plays a significant role in high-performance computing systems, from large clusters of computers to chip-multithreading (CMT) processors. Performance of the parallel systems comes not only from concurrently running more processing hardware but also from utilizing the hardware efficiently. The hardware utilization is strongly influenced by how processors/processes are synchronized in the system to maximize parallelism. Synchronization between concurrent processes usually relies on shared data structures. The data structures that enhance parallelism by allowing processes to access them concurrently are known as *concurrent data structures*. The thesis aims at developing efficient concurrent data structures and algorithms for synchronization in asynchronous shared-memory multiprocessors.

Generally speaking, simple data structures perform well in the absence of contention but perform poorly in high-contention situations. Contrarily, sophisticated data structures that can scale and perform well in the presence of high contention usually suffer unnecessary high latency when there is no contention. Efficient concurrent data structures should be able to adapt their algorithmic complexity to varying contention. This has motivated us to develop fundamental concurrent data structures like trees, multi-word compare-and-swap and locks into reactive ones that timely adapt their size or algorithmic behavior to the contention level in execution environments. While the contention is varying rapidly, the reactive data structures must keep the cost of reaction below its benefit, avoiding unnecessary reaction due to the contention oscillation. This is quite challenging since the information of how the contention will vary in the future is usually not available in multiprogramming multiprocessor environments. To deal with the uncertainty, we have successfully synthesized non-blocking synchronization techniques and advanced on-line algorithmic techniques, in the context of reactive concurrent data structures. On the other hand, we have developed a new optimal on-line algorithm for one-way trading with time-varying exchange-rate bounds. The algorithm extends the set of practical problems that can be transformed to the one-way trading so as to find an optimal solution. In this thesis, the new algorithm demonstrates its applicability by solving the freshness problem in the context of concurrent data structures.

Keywords: *synchronization, reactive, non-blocking, concurrent data structures, distributed data structures, multi-word atomic primitives, spin-locks, shared memory, online algorithms, online financial problems, randomization.*

List of Included Papers and Reports

This thesis is based on the work contained in the following publications.

1. Phuong Hoai Ha & Philippas Tsigas. Reactive multi-word synchronization for multiprocessors. *Proceedings of the 12th IEEE/ACM International Conference on Parallel Architectures and Compilation Techniques (PACT '03)*, Sept. 2003, pp. 184-193, IEEE press.

2. Phuong Hoai Ha & Philippas Tsigas. Reactive multi-word synchronization for multiprocessors. *The Journal of Instruction-Level Parallelism*, Vol. 6, No. *(Special issue with selected papers from the 12th IEEE/ACM International Conference on Parallel Architectures and Compilation Techniques)*, April 2004, AI Access Foundation and Morgan Kaufmann Publishers.

3. Phuong Hoai Ha, Marina Papatriantafilou & Philippas Tsigas. Self-tuning Reactive Distributed Trees for Counting and Balancing. *Proceedings of the 8th International Conference on Principles of Distributed Systems (OPODIS '04)*, Dec. 2004, LNCS 3544, pp. 213-228, Springer-Verlag.

4. Phuong Hoai Ha, Philippas Tsigas, Mirjam Wattenhofer & Roger Wattenhofer. Efficient Multi-Word Locking Using Randomization. *Proceedings of the 24th Annual ACM SIGACT-SIGOPS Symposium on Principles Of Distributed Computing (PODC '05)*, Jul. 2005, pp. 249-257, ACM Press.

5. Phuong Hoai Ha, Marina Papatriantafilou & Philippas Tsigas. Reactive Spin-locks: A Self-tuning Approach. *Proceedings of the 8th IEEE International Symposium on Parallel Architectures, Algorithms and Networks (I-SPAN '05)*, Dec. 2005, pp. 33-39, IEEE press.

6. Peter Damaschke, Phuong Hoai Ha & Philippas Tsigas. One-Way Trading with Time-Varying Exchange-Rate Bounds. *Technical report: CS:2005-17*, Department of Computer Science and Engineering, Chalmers University of Technology, Sweden.

7. Peter Damaschke, Phuong Hoai Ha & Philippas Tsigas. Competitive Freshness Algorithms for Wait-free Data Objects. *Technical report: CS:2005-18*, Department of Computer Science and Engineering, Chalmers University of Technology, Sweden.

I together with the co-authors of the respective papers contributed to the design, analysis and experimental evaluation of the algorithms presented in these papers, as well as to the writing of these papers in a nice research environment.

ACKNOWLEDGMENTS

First of all, I wish to thank Philippas Tsigas, my supervisor, and Marina Papatriantafilou for their enthusiasm and constant support. They continuously inspire my research with excellent advices. I have learnt an enormous amount on how to do research from working with them and have really enjoyed that. I am honored to be one of their students. Further, I would like to thank Björn von Sydow, a member of my committee, for following my research with helpful comments. I am also grateful to Peter Dybjer for kindly agreeing to be my examiner.

I am very honored to have Prof. Michael L. Scott from University of Rochester as my faculty opponent. I am also honored to my grading committee: Prof. Otto J. Anshus, Prof. Björn Lisper, and Prof. Per Stenström.

I feel privileged to be a member of the distributed system group with nice colleagues: Niklas Elmqvist, Anders Gidenstam, Boris Koldehofe, Andreas Larsson, Elad Schiller, Håkan Sundell and Yi Zhang. They gave me many helpful comments and advices not only on my work but also on the Swedish life.

Frankly, I would not have gone so far without the support of all staffs and other PhD students at the Department of Computing Science. I would like to take this chance to thank them all. Many thanks to Bror Bjerner and Sven-Arne Andréasson for suitably coordinating my teaching duty with my research, which helps me have better insights in my research fields.

It would be a mistake if I forget to thank my good friends. Without them, my life would have been more difficult and less enjoyable. I could not thank them personally here, but they know who they are. Thanks to them all!

Last but not least, I wish to give many thanks to my family for their constant love, support and encouragement. They are forever my motivation to overcome any hardships and challenges in my life.

Phuong Hoai Ha

Göteborg, June 2006.

Contents

1 Introduction **1**
- 1.1 Our Contributions . . . 2
  - 1.1.1 Universal Constructions for Concurrent Data Structures . 2
  - 1.1.2 Reactive Spin-locks . . . 3
  - 1.1.3 Self-Tuning Diffracting Trees . . . 4
  - 1.1.4 Freshness Algorithms for Wait-free Data Objects . . . 4
  - 1.1.5 One-Way Trading with Time-Varying Bounds . . . 5

2 Background **7**
- 2.1 Shared Memory Multiprocessors . . . 7
  - 2.1.1 Synchronization Primitives . . . 9
- 2.2 Mutual Exclusion . . . 10
- 2.3 Non-blocking Synchronization . . . 15
  - 2.3.1 Universal Constructions . . . 16
  - 2.3.2 Freshness . . . 17
- 2.4 Distributed Data Structures for Counting and Balancing . . . 19
- 2.5 Online Algorithms . . . 21

3 Reactive Multi-Word Synchronization **25**
- 3.1 Introduction . . . 26
- 3.2 Problem Description, Related Work and Our Contribution . . . 27
  - 3.2.1 Our Contribution . . . 30
- 3.3 Algorithm Informal Description . . . 31
  - 3.3.1 The First Algorithm . . . 33
  - 3.3.2 The Second Algorithm . . . 34
- 3.4 Implementations . . . 35
  - 3.4.1 First Reactive Scheme . . . 35
  - 3.4.2 Second Reactive Scheme . . . 43
- 3.5 Correctness Proof . . . 46

3.6 Evaluation . . . 51
3.6.1 Results . . . 52
3.7 Conclusions . . . 54

4 Efficient Multi-Word Locking 57
4.1 Introduction . . . 58
4.2 Related Work . . . 59
4.3 Problem and Model . . . 60
4.4 Randomized Registers . . . 61
4.4.1 The Algorithm . . . 61
4.4.2 Length of Directed Paths . . . 61
4.4.3 Length of Waiting Chains . . . 63
4.4.4 Execution Time . . . 65
4.5 Evaluation . . . 70
4.5.1 The micro-benchmark . . . 71
4.5.2 The application . . . 73
4.6 Conclusions . . . 75

5 Reactive Spin-locks: A Self-Tuning Approach 77
5.1 Introduction . . . 78
5.1.1 Contributions . . . 79
5.2 Problem analysis . . . 81
5.2.1 Tuning parameters and system characteristics . . . 81
5.2.2 Experimental studies . . . 82
5.3 Modeling the problem . . . 83
5.4 The algorithm . . . 86
5.5 Correctness . . . 88
5.6 Estimating the delay base . . . 90
5.7 Evaluation . . . 93
5.7.1 Results . . . 95
5.8 Conclusions . . . 98

6 Self-Tuning Diffracting Trees 99
6.1 Introduction . . . 100
6.2 Background . . . 102
6.2.1 Diffracting and Reactive-Diffracting Trees . . . 102
6.2.2 Online Algorithms . . . 103
6.3 Self-tuning reactive trees . . . 104
6.3.1 Problem description . . . 104
6.3.2 Key ideas . . . 104

6.3.3 The tree structure . . . 105
6.3.4 The reactive scheme . . . 106
6.3.5 Space needs of the tree . . . 108
6.4 Implementation . . . 108
6.4.1 Preliminaries . . . 108
6.4.2 Traversing self-tuning reactive trees . . . 109
6.4.3 Reaction conditions . . . 111
6.4.4 Expanding a leaf to a sub-tree . . . 112
6.4.5 Shrinking a sub-tree to a leaf . . . 114
6.4.6 Efficiency enhancement . . . 116
6.5 Correctness Proof . . . 118
6.6 Evaluation . . . 121
6.6.1 Full contention benchmark . . . 123
6.6.2 Surge load benchmark . . . 124
6.7 Conclusion . . . 125

7 Competitive Freshness Algorithms 127
7.1 Introduction . . . 128
7.2 Preliminaries . . . 129
7.3 Problem and Model . . . 131
7.4 Optimal Deterministic Algorithm . . . 133
7.5 Competitive Randomized Algorithm . . . 136
7.6 Conclusions . . . 139

8 One-Way Trading with Time-Varying Bounds 141
8.1 Introduction . . . 142
8.1.1 Freshness of Concurrent Data Objects . . . 142
8.1.2 Our contributions . . . 144
8.2 The Lower Bound of Competitive Ratios . . . 145
8.3 Optimal threat-based policy for the second model . . . 148
8.4 Conclusions . . . 159

9 Conclusions and Future Research 161

List of Figures

2.1 The bus-based and crossbar-switch-based UMA systems 8
2.2 The SGI Origin 2000 architecture with 32 processors, where R is a router. 9
2.3 Synchronization primitives 10
2.4 The structure of concurrent processes in the mutual exclusion . . . 11
2.5 The *test-and-set* lock 12
2.6 The *test-and-test-and-set* lock 12
2.7 The splitter element 14
2.8 Freshness problem 18
2.9 A diffracting tree (A) and a reactive diffracting tree (B). 20

3.1 Recursive helping policy and software transactional memory . . . 29
3.2 Reactive-CASN states and reactive-CAS4 data structure 32
3.3 Reactive CASN description 32
3.4 The term definitions 33
3.5 Synchronization primitives 36
3.6 Data structures in our first reactive multi-word compare-and-swap algorithm 36
3.7 Procedures CASN and Help in our first reactive multi-word compare-and-swap algorithm 38
3.8 Procedures Locking and CheckingR in our first reactive multi-word compare-and-swap algorithm 39
3.9 Procedures Updating and Unlocking/Releasing in our first reactive multi-word compare-and-swap algorithm 40
3.10 Circle-helping problem: (A) Before helping; (B) After p_1 helps $CAS3_2$ acquire $Mem[2]$ and $Mem[5]$. 42
3.11 Procedures Help and Unlocking in our second reactive multi-word compare-and-swap algorithm. 44

3.12 Procedures CheckingR in our second reactive multi-word compare-and-swap algorithm and the procedure for Read operation. 45
3.13 Shared variables with procedures reading or directly updating them 46
3.14 The numbers of CAS2s, CAS4s and CAS8s and the number of *successful* CAS2s, CAS4s and CAS8s in one second 53

4.1 Delation of p. 64
4.2 t_1, t_2 and t_3 for a process. 65
4.3 Process p, q_1, q_2, q_3, q_4 are in p's delay graph. The depth of p is 3, q_1's depth is also 3. 66
4.4 Depth(p)=1. 68
4.5 The single-word compare-and-swap primitive 70
4.6 The distributions of the longest wait-queue lengths in the micro-benchmark on the SGI Origin2000. 72
4.7 The micro-benchmark execution times on the SGI Origin2000. . . 73
4.8 The algorithm for a thread t_k in computing one result/output . . . 75
4.9 The application execution times on the SGI Origin2000. 75

5.1 The structure for parallel applications 79
5.2 Synchronization primitives, where x is a variable and v, old, new are values. 80
5.3 The execution time and the lock fairness of the Spark98 benchmark on an SGI Origin3800. 82
5.4 The transaction phases of contention variations on the lock. 86
5.5 The Acquire and Release procedures 89
5.6 The table of manually tuned parameters for $TTSE$ and $TicketP$ in Spark98, Volrend and Radiosity applications on the SGI Origin2000 and the Intel Xeon workstation, where b_e, l_e are respectively $TTSE$'s *delay base* and *delay upper limit* for exponential backoff, and b_p is $TicketP$'s *delay base* for proportional backoff delays. The b_e, l_e and b_p are measured by the number of null-loops. 95
5.7 The execution time of Spark98, Volrend and Radiosity applications on the SGI Origin2000. 95
5.8 The execution time of Spark98, Volrend and Radiosity applications on a workstation with 2 Intel Xeon processors. 96

6.1 A diffracting tree (A) and a reactive diffracting tree (B). 102
6.2 A self-tuning reactive tree 105
6.3 The tree basic data structure and the synchronization primitives . . 109

6.4 The TraverseTree, TraverseB, TraverseL, CheckCondition, Surplus2Latency and Latency2Surplus procedures 110
6.5 The $Grow$, $Elect2Shrink$ and $Shrink$ procedures 113
6.6 The $NextCount$ function in the $Grow$ procedure 114
6.7 Illustrations for $Grow$ and $Shrink$ procedures 114
6.8 The $NextCount$ function in $Shrink$ procedure. 115
6.9 The BasicAssign, Assign, Read, and AcquireLock_cond operations 116
6.10 An illustration for the need of *read-and-follow-link* operation . . . 117
6.11 Throughput and average depth of trees in the full-contention benchmark on SGI Origin2000. 122
6.12 Average depths of trees in the surge load benchmark on SGI Origin2000, the fastest and the average reactions. 124
6.13 Throughput of trees in the surge load benchmark on SGI Origin2000.125

7.1 Illustrations for concurrent reading/writing and freshness problem 131
7.2 Illustrations for Theorem 2 and the randomized algorithm 136

8.1 Freshness problem 143
8.2 Illustration for the proof of Theorem 4 147
8.3 Numerical comparison of competitive ratios among different algorithms. The last row shows values of k corresponding to the ratios c in the improved TBP. 158

Chapter 1

Introduction

Parallel systems aim at supporting computation capacity for large parallel applications in many research areas like high-energy physics, biomedical sciences and earth sciences. Such applications consist of tasks/processes that run concurrently and share common data/resources. Since most of the shared resources do not allow more than one process to access them concurrently in a predictive manner, the processes need to be synchronized efficiently. The conventional method to synchronize concurrent processes is *mutual exclusion* (e.g. semaphore, monitor). Mutual exclusion degrades the system's overall performance as it causes blocking (cf. Section 2.2). To address the drawbacks of mutual exclusion, a concept called *non-blocking synchronization* has been proposed for shared resources (cf. Section 2.3). Non-blocking synchronization entails implementations of fundamental data structures that allow many processes to access them concurrently. The implementations are known as *concurrent data structures*.

On one hand, concurrent data structures aim at improving performance by maximizing parallelism. On the other hand, since the data structures allow many processes to concurrently access them, the contention level on them is high, subsequently degrading performance. Generally, simple data structures perform well in the absence of contention but perform poorly in high-congestion situations. Contrarily, sophisticated data structures that can scale and perform well in the presence of high contention usually suffer unnecessary high latency when there is no contention. Efficient concurrent data structures should be able to adapt their algorithmic complexity to contention variation. This fact raises a question on constructing reactive concurrent data structures and algorithms that can react to contention variation so as to achieve good performance in all conditions.

There exist reactive concurrent data structures and algorithms in the literature [2, 7, 25, 66, 75]. However, their reactive schemes rely on either some experi-

mentally tuned thresholds or known probability distributions of some inputs. Such *fixed* experimental thresholds may frequently become inappropriate in variable and unpredictable environments such as multiprogramming systems. The assumption on known probability distributions of some inputs is usually not feasible.

These issues motivated us to research and develop efficient reactive concurrent data structures and algorithms that require neither experimentally tuned thresholds nor probability distributions of inputs. In order to develop such concurrent data structures and algorithms, we need to deal with unpredictability in execution environments. Our approach is to synthesize non-blocking synchronization techniques and advanced on-line algorithmic techniques (e.g. on-line trading), in the context of reactive concurrent data structures. Working on this research direction, we have achieved the following results.

1.1 Our Contributions

1.1.1 Universal Constructions for Concurrent Data Structures

We have developed and implemented new reactive multi-word compare-and-swap objects [39, 40]. The multi-word compare-and-swap objects are powerful constructions, which make the design of concurrent data structures much more convenient and efficient.

Shared memory multiprocessor systems typically provide a set of hardware primitives in order to support synchronization. Generally, they provide *single-word* read-modify-write hardware primitives such as compare-and-swap, load-linked/store-conditional and fetch-and-op, from which higher-level synchronization objects are then implemented in software. Although the *single-word* hardware primitives are conceptually powerful enough to support higher-level synchronization, from the programmer's point of view they are not as useful as their generalizations, the *multi-word* objects. This has motivated us to develop two fast and reactive lock-free *multi-word* compare-and-swap algorithms. The algorithms dynamically measure the level of contention as well as the memory conflicts of the *multi-word* compare-and-swap objects, and in response, they react accordingly in order to guarantee good performance in a wide range of system conditions. The algorithms are non-blocking (lock-free), allowing in this way fast dynamical behavior. Experiments on thirty processors of an SGI Origin2000 multiprocessor have showed that both our algorithms react quickly to the contention variations and outperform the best-known alternatives in almost all contention conditions. The algorithms, together with their implementation details, are described in Chapter 3.

Subsequently, we have developed a general multi-word locking method that allows processes to efficiently multi-lock arbitrary registers [41]. Multi-word lock-

ing methods, together with appropriate helping schemes, are the core of universal constructions like multi-word compare-and-swap objects. We have examined the general multi-word locking problem, where processes are allowed to multi-lock arbitrary registers. Aiming for a highly efficient solution, we have proposed a randomized algorithm that successfully breaks long dependency chains, the crucial factor for slowing down an execution. In the analysis, we have focused on the 2-word locking problem and showed that in this special case an execution of our algorithm takes with high probability at most time $O(\Delta^3 \log n / \log\log n)$, where n is the number of registers and Δ is the maximal number of processes interested in the same register (the contention). Furthermore, we have implemented our algorithm for the general multi-word lock problem on an SGI Origin2000 machine, demonstrating that our algorithm is not only of theoretical interest. The algorithm, together with its analysis, is described in Chapter 4.

1.1.2 Reactive Spin-locks

We have developed and implemented a new reactive lock-based synchronization algorithm that can automatically react to contention variation on the lock [38]. The algorithm is based on synchronization structures of applications and competitive online algorithms.

Reactive spin-lock algorithms that can automatically react to contention variation on the lock have received great attention in the field of multiprocessor synchronization. This results from the fact that the algorithms help applications achieve good performance in all possible contention conditions. However, to make decisions, the reactive schemes in the existing algorithms rely on (i) some *fixed* experimentally tuned thresholds, which may frequently become inappropriate in dynamic environments like multiprogramming/multiprocessor systems, or (ii) known probability distributions of inputs, which are usually not available. This has motivated us to develop a new reactive spin-lock algorithm that is completely self-tuning, which means no experimentally tuned parameter nor probability distribution of inputs is needed. The new spin-lock is built on a competitive online algorithm. Our experiments, which use the Spark98 kernels and the SPLASH-2 applications as application benchmarks, on a multiprocessor machine SGI Origin2000 and on an Intel Xeon workstation have showed that the new self-tuning spin-lock helps applications with different characteristics to achieve good performance in a wide range of contention levels. The algorithm, together with its implementation details, is described in Chapter 5.

1.1.3 Self-Tuning Diffracting Trees

We have developed and implemented new self-tuning reactive trees that distribute a set of memory accesses to different memory banks in a coordinated manner [37]. The trees reactively adapt their size to contention variation, attaining good performance at all contention levels.

Reactive diffracting trees [25] are efficient distributed data structures that supports synchronization. The trees distribute a set of processes to smaller subsets that access different parts of memory in a global coordinated manner. They also adjust their size to attain good performance in the presence of different contention levels. However, their adjustment is sensitive to parameters that have to be manually tuned and determined after experimentation. Since these parameters depend on the application as well as on the system configuration, determining their optimal value is hard in practice. On the other hand, as the trees grow or shrink by only one level at a time, the cost of multi-level adjustments is high. This has motivated us to develop new reactive diffracting trees for counting and balancing without the need to tune parameters manually. The new trees, in an on-line manner, balance the trade-off between the tree traversal latency and the latency due to contention at the tree nodes. Moreover, the trees can grow or shrink by several levels in one adjustment step, improving their efficiency. Their efficiency is illustrated via experiments, which compare the new trees with the traditional reactive diffracting trees. The experiments, which have been conducted on an SGI Origin2000, a well-known commercial ccNUMA multiprocessor, have showed that the new trees select the same tree depth, perform better and react faster than the traditional trees. The tree algorithm, together with its implementation details, is described in Chapter 6.

1.1.4 Freshness Algorithms for Wait-free Data Objects

We have analyzed the freshness problem of concurrent data objects [1] as an online problem and subsequently developed two online algorithms for the problem: an optimal deterministic algorithm and a competitive randomized algorithm [23].

Wait-free concurrent data objects are widely used in multiprocessor systems and real-time systems. Their popularity results from the fact that they avoid locking and that concurrent operations on such data objects are guaranteed to finish in a bounded number of steps regardless of the other operations interference. The data objects allow high access parallelism and guarantee correctness of the concurrent access with respect to its semantics. In such a highly concurrent environment, where write-operations update the object state concurrently with read-operations, the age/freshness of the state returned by a read-operation is a significant measure

[1] A data structure that is considered as a basic building block is called *data object* in this thesis.

of the object quality, especially in reactive/detective real-time systems. To address the problem, we have first proposed a freshness measure for wait-free concurrent data objects. Subsequently, we have modeled the freshness problem as an online problem and developed two algorithms for the problem. The first one is a deterministic algorithm with freshness competitive ratio $\sqrt{\alpha}$, where α is a function of the execution-time upper-bound of the wait-free operations. Moreover, we have proved that $\sqrt{\alpha}$ is asymptotically the optimal freshness competitive ratio for deterministic algorithms, implying that the first algorithm is optimal. The second algorithm is a competitive randomized algorithm with freshness competitive ratio $\frac{\ln \alpha}{1+\ln 2-\frac{2}{\sqrt{\alpha}}}$. The algorithms, together with their analysis, are described in Chapter 7.

1.1.5 One-Way Trading with Time-Varying Bounds

We have developed new online trading models to which more practical problems in parallel/distributed systems can be transformed in order to find an optimal solution [24]. For the models, we have proved lower bounds of competitive ratios and suggested an optimal competitive algorithm.

One-way trading is a basic online problem in finance. Since its optimal solution is given by a simple formula (however with difficult analysis), the problem is attractive as a target to which other practical online problems can be transformed. However, there are still natural online problems that do not fit in the known variants of one-way trading. We have developed some new models where the bounds of exchange rates are not constant but vary with time in certain ways. The first model, where the (logarithmic) exchange rate has limited decay speed, arises from an issue in distributed data structures, namely to maximize the freshness of values in concurrent objects. For this model we have proved a lower bound on competitive ratios which is nearly optimal, i.e., up to a small constant factor. Clearly, the lower bound holds also against stronger adversaries. Subsequently, we have presented an optimal algorithm in a model where only the maximal allowed exchange rate decreases with time, but the actual exchange rates may jump arbitrarily within this frame. We have chosen this more powerful adversary model afterwards because some applications do not make use of the limited decay speed. Our numerical experiments have suggested that this algorithm is still not too far from the lower bound for the weaker adversary. This is explained by the observation that slowly increasing exchange rates seems to be the worst case for the online player. The new models and the algorithm, together with their analysis, are described in Chapter 8.

Chapter 2

Background

2.1 Shared Memory Multiprocessors

In this section, we give a brief overview of shared memory multiprocessors, the computer architecture that most of the work in this thesis concerns.

A multiprocessor system can be considered as "a collection of processing elements that communicate and cooperate to solve large problems fast" [22]. This description shows the significant role of communication architecture which can be based on either *message passing* or *shared memory*. In message passing systems, the communication is done via sending and receiving messages. In shared memory multiprocessor systems, processors communicate by writing and reading shared variables in a common memory address space. Depending on the features of processor-to-memory interconnection, shared memory multiprocessor systems can be classified into two subclasses: uniform memory access (UMA) and non-uniform memory access (NUMA).

Uniform Memory Access (UMA): In the UMA systems, references from processors to memory banks have the same latency. The interconnection networks often used in UMA systems are bus and crossbar-switch types. In bus-based UMA systems as depicted in Figure 2.1, all processors access memory banks via a central bus and consequently the bus becomes a bottle-neck when the number of processors increases. Since the bandwidth of the bus is still fixed when more processors are connected to the bus, the average bandwidth for each processor decreases as the number of processors increases. Therefore, the bus-based architecture limits the system's scalability. Unlike bus-based UMA systems, crossbar-switch based UMA systems increase the aggregate bandwidth when adding new processors. As depicted in Figure 2.1, each processor in the system has an interconnection to each

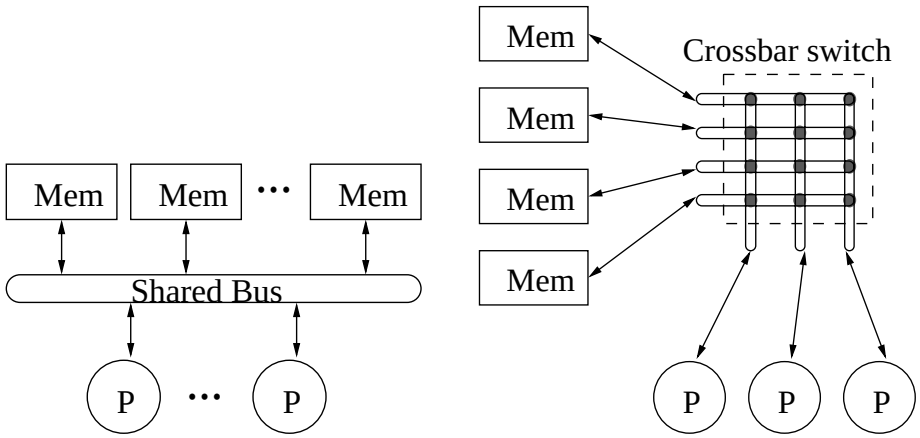

Figure 2.1: The bus-based and crossbar-switch-based UMA systems

memory bank. Therefore, bandwidth is not a problem to the system when adding new processors, but the problem is to expand the switch: the cost increment is quadratic to the number of ports.

Since it is expensive to design a scalable UMA system, an alternative design called non-uniform memory access (NUMA) has been proposed.

Non-Uniform Memory Access (NUMA): As their name mentions, NUMA systems have different latency of references from a processor to memory banks. In the system, the shared memory banks are distributed among processors in order to alleviate traffic on the processor-to-memory interconnection: references from a processor to its memory bank do not incur traffic on the interconnection. This makes local-memory references faster than remote-memory ones. Although the physical memory banks are distributed, the same virtual address space is shared by the whole system and memory controllers handle the mapping from the virtual address space to the physical distributed memory banks. Such memory architecture is called distributed shared memory (DSM). The NUMA architecture reduces both the average access time and the bandwidth demand on the interconnection because requests to local memory banks are executed locally.

Moreover, in order to increase performance and reduce the memory access time, cache is exploited in modern systems. The data from memory is replicated into caches of processors and the system supports means to keep the caches consistent. Such a system is called cache-coherent non-uniform memory access (cc-NUMA).

The SGI Origin 2000 [64], which has been used for the experiments in this the-

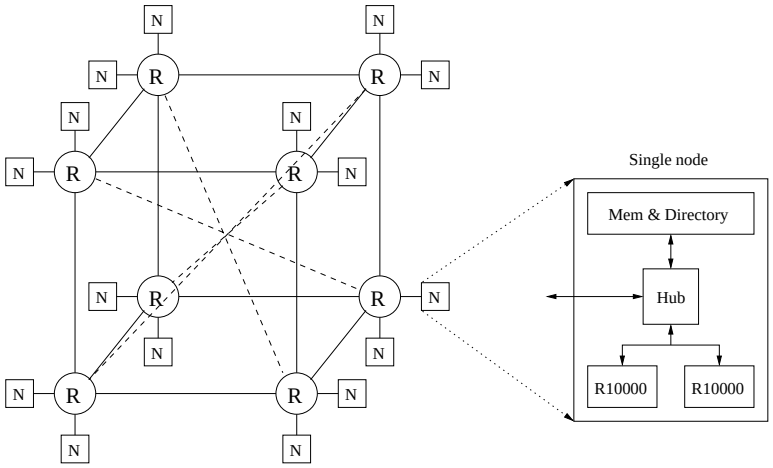

Figure 2.2: The SGI Origin 2000 architecture with 32 processors, where R is a router.

sis, is a commercial ccNUMA machine. The machine can support up to 512 nodes, which each contains 2 MIPS R10000 processors and up to 4GB of memory. The machine uses distributed shared memory (DSM) and maintains cache-coherence via a directory-based protocol, which keeps track of the cache-lines from the corresponding memory. A cache-line consists of 128 bytes. The machine architecture is depicted in Figure 2.2

2.1.1 Synchronization Primitives

In order to synchronize processes efficiently, modern shared memory multiprocessors support some *strong* synchronization primitives. The primitives are classified according to their *consensus number* [46], the maximum number of processes for which the primitives can be used to solve a *consensus problem* in a fault tolerant manner. In the consensus problem, a set of n asynchronous processes, each with a given input, communicate to achieve an agreement on one of the inputs. A primitive with a consensus number n can achieve consensus among n processes even if up to $n-1$ processes stop [102].

According to the consensus classification, read/write registers have consensus number 1, i.e. they cannot tolerate any faulty processes in the consensus setting. There are some primitives with consensus number 2 and some with infinite consensus number. In this subsection, we present only the primitives used in this thesis. The synchronization primitives related to our algorithms are two primitives with consensus number 2: *test-and-set (TAS)* and *fetch-and-op (FAO)* and two

```
TAS(x) /* init: x ← 0 */
atomically{
    oldx ← x;
    x ← 1;
    return oldx;
    }

FAO(x, v)
atomically {
    oldx ← x;
    x ← op(x, v);
    return(oldx)
    }

CAS(x, old, new)
atomically {
    if(x = old)
      x ← new;
      return(true);
    else return(false);
    }

LL(x){
    return the value of x so that it may
    be subsequently used with SC
    }

VL(x)
atomically {
    if (no other process has written to x
    since the last LL(x))
      return(true);
    else return(false);
    }

SC(x, v)
atomically {
    if (no other process has written to x
    since the last LL(x))
      x ← v; return(true);
    else return(false);
    }
```

Figure 2.3: Synchronization primitives

primitives with infinite consensus number: *compare-and-swap (CAS)* and *load-linked/validate/store-conditional (LL/VL/SC)*. The definitions of the primitives are described in Figure 2.3, where x is a variable, v, old, new are values and op can be operators add, sub, or, and and xor.

For the systems that support *weak LL/SC*[1] or the systems that support *CAS*, we can implement the *LL/VL/SC* instructions algorithmically [80]. Since both *LL/SC* and *CAS* are primitives with infinite consensus number, or universal primitives, one primitive can be implemented from the other.

2.2 Mutual Exclusion

In shared memory multiprocessor systems, processes communicate by writing and reading shared variables or shared objects. In order to guarantee the consistency of

[1]The weak LL/SC instructions allow the SC instruction to return $false$ even though there was no update on the corresponding variable since the last LL.

```
while true do
    Noncritical section;
    Entry section;
    Critical section;
    Exit section;
od
```

Figure 2.4: The structure of concurrent processes in the mutual exclusion

the object state/value, the conventional method is to use mutual exclusion, which allows only one process to access the object at one time. A brief description of mutual exclusion research is given in this section.

In mutual exclusion, each concurrent process repeatedly executes a *noncritical section* and a *critical section* as depicted in Figure 2.4, where *Noncritical section* contains the code not accessing the shared object; *Entry section* contains the code responsible for resolving the conflict among concurrent processes so that only one can pass by this section and all other processes have to wait here until the winner exits the critical section; *Critical section* contains the code accessing the shared object; and *Exit section* contains the code to inform other waiting processes that the winner has left the critical section. A mutual exclusion algorithm consists of the *Entry* and *Exit* sections. Mutual exclusion algorithms must satisfy the two following requirements [4]:

Exclusion : at most one process can be at the critical section at any point in time.

Livelock-freedom : if there are some processes in the entry section, one process will eventually enter the critical section.

Moreover, an efficient mutual exclusion algorithm should generate small overhead, i.e delay time, due to the execution of entry and exit sections.

In systems where many processes can be executed concurrently by the same processor, there are two options for waiting processes in the entry section: *blocking* or *busy-waiting*. In the blocking mode, waiting processes relinquish their processors , allowing other processes to execute useful tasks. This mode, however, incurs the context-switching cost: waiting processes' contexts must be saved and contexts of other processes must be restored. In the busy-waiting mode, waiting processes continuously check shared variables (or a lock) protecting the critical section; these are known as *spinning* algorithms. In this mode, processes waste processor time on checking the lock. Therefore, an efficient mutual exclusion algorithm should keep waiting processes in the busy-waiting mode when the waiting time is short

and should switch them to the blocking mode when the waiting time is long. However, the waiting time in which the lock will be held is normally unknown prior to the decision point. This challenge has attracted a lot of attention to this research field [2, 7, 54, 58]; especially Karlin et al. [58] presented five competitive strategies for determining whether and how long to busy-wait before blocking.

In systems where each process is executed on a separate processor, the blocking mode is no longer necessary since no other process will use the relinquished processor. However, even in this case, the busy-waiting mode still encounters another challenge: if processes/processors continuously execute read-modify-write operations on the lock, this may incur high network traffic and high memory contention. A simple mutual exclusion algorithm known as a *test-and-set* lock is depicted in Figure 2.5. Since the TAS primitive requires a "write" permission even when it fails to update the $lock$ variable, it causes a *read-miss* in cache-coherent multiprocessor systems. All processors experiencing the read-miss will concurrently access the variable in order to read its unchanged value again, causing an access burst on both the processor-to-memory interconnection and the memory module containing the variable. Therefore, continuously executing TAS inside the while-loop incurs a very high load, slowing down all accesses to the interconnection and to other non-contended variables on this memory module.

---

shared $lock := 0$;
acquire()$\{$ **while** $(TAS(lock) = 1)$; $\}$
release()$\{$ $lock := 0$; $\}$;

Figure 2.5: The *test-and-set* lock

Test-and-test-and-set locks: In order to reduce the surge load caused by TAS, an improved version of the test-and-set lock has been suggested, which is known as a *test-and-test-and-set* lock (cf. Figure 2.6). The improvement is to execute TAS only if the lock is available, i.e. $lock \neq 1$. In cache-coherent multiprocessor systems, the $lock$ variable is cached at each processor and thus reading the variable does not incur any network traffic.

---

acquire()$\{$ **while** $(lock = 1$ **or** $TAS(lock) = 1)$; $\}$

Figure 2.6: The *test-and-test-and-set* lock

Backoff technique: Although the improved version significantly reduces network traffic and memory conflict, continuously checking the cached variable still incurs a surge load when the lock is released. At that time, all waiting processors

will *concurrently* try to execute TAS on the variable as in the test-and-set lock. In order to avoid the situation where all processors concurrently realize that the lock is available, a delay is inserted between two consecutive iterations. How long the delay should be is an interesting issue. Agarwal et al. [2] suggested an exponential backoff scheme, where the delay on each processor will be doubled up to a limit every time the processor checks and finds the lock unavailable. The backoff scheme has been showed to work in practice, but it needs manually tuned parameters, namely the initial delay and the upper limit of the delay. Inaccurately chosen parameters will significantly affect the lock performance.

Queue-lock: In the aforementioned algorithms, busy-waiting loops need to access a common lock variable. The variable consequently becomes a bottleneck, limiting the scalability of the system. This has motivated researchers to develop *local-spin* algorithms [7, 21, 32, 73, 77, 110] in which busy-waiting loops access only shared variables that are *locally accessible*. Such algorithms are especially efficient in cache-coherent multiprocessors and in distributed shared-memory processors. A variable is locally accessible if either a copy of the variable is in the local cache (for the former) or the variable is stored in the local partition of the shared memory (for the latter). The most practical local-spin algorithms in the literature are queue-locks [7,21,32,73,77]. The idea is that each processor spins on a separate variable and subsequently the variable will be cached at the corresponding processor. These spin-variables are linked together to construct a waiting queue. When the winner releases the lock, it informs the first processor in the waiting queue by writing a value to the processors spin-variable. Since only one processor is informed that the lock is available, no conflict among waiting processors occurs when the lock is released.

Algorithms using only Read/Write: All aforementioned mutual exclusion algorithms exploit strong synchronization primitives such as TAS, FAO and CAS. Another major research trend on mutual exclusion is to design mutual exclusion algorithms using only $Read$ and $Write$ operations that are faster than the strong synchronization primitives. The trend has been initiated by Lamport's fast mutual exclusion algorithm [63] that has been generalized to the *splitter* element [10]. Figure 2.7 describes the splitter and its implementation. A beautiful feature of the splitter is that if n concurrent processes enter the splitter, the splitter will split the set of concurrent processes such that: i) at most one stops at the splitter, ii) at most $n-1$ processes go right and iii) at most $n-1$ processes go down. That means if we have a complete binary tree of splitters with depth $n-1$, all n concurrent processes entering at the root of the tree will be kept within the tree and each splitter will keep at most one process.

```
shared Y := 0, X;
splitter(pid){
    X := pid;
    if Y ≠ 0 then return Right;
    else
        Y := pid;
        if X ≠ pid then return Down;
        else return Stop;
    }
```

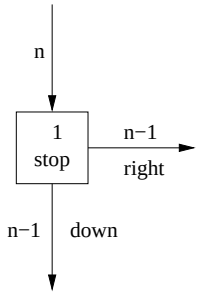

Figure 2.7: The splitter element

Drawbacks

Although substantial research effort has been devoted to the improvement of mutual exclusion algorithms, the concept of lock-based synchronization itself contains inevitable drawbacks:

- *Risk of lock convoy and deadlock:* In lock-based synchronization, one process slowing down can make the whole system consisting of many processes slow down. If the process that is holding the lock is delayed for some reason, e.g. due to preemption, page faults, cache misses or interrupt handling, other processes waiting for the lock must suffer the delay even though they are running on other independent and fast processors. Moreover, if the lock holder suddenly crashes or cannot proceed to the point where it releases the lock, all processes waiting for the lock to be released will wait forever.

- *Risk of priority inversion:* In real-time systems, a high-priority task must be executed before lower-priority tasks in order to meet its deadline. However, if the tasks communicate using lock-based synchronization, a low-priority task can delay a higher-priority task even if they do not share any objects.

  For example, assume there are three tasks T_1, T_2, T_3, where T_1 has the highest priority and T_3 has the lowest priority. T_1 and T_3 share a resource protected by a lock. Assume that T_3 is holding the lock and thus T_1 is delayed by T_3 until T_3 releases the lock. Before T_3 releases the lock, T_2 with higher priority than T_3 will delay T_3 and thus delay T_1. We see that even though T_1 and T_2 do not share any resource, the lower priority task T_2 can delay the higher priority task T_1.

These risks can be eliminated using complicated operation system supports like blocking-aware schedulers and priority-inheritance protocols [89], which entail ad-

ditional overhead. Because of the drawbacks of the lock-based synchronization, an alternative called *non-blocking synchronization* has been suggested and it has become a major research trend in the area.

2.3 Non-blocking Synchronization

To address the drawbacks of lock-based synchronization, a new concept of *non-blocking synchronization* has been proposed for shared resources. Non-blocking synchronization does not involve mutual exclusion, and therefore does not suffer from the problems that blocking can cause. Non-blocking algorithms are either lock-free or wait-free.

A *lock-free* implementation of a concurrent data structure guarantees that always *at least one* of the concurrent operations executed by processes progresses regardless of the interleaving caused by the other operations. A general approach is that each operation makes its own copy of the shared data, updates the copy and finally makes the copy become the current data. The final step is usually done by switching a pointer using strong synchronization primitives like compare-and-swap or load-linked/store-conditional to avoid race conditions. This approach has drawbacks like the high cost of copying the whole data and loss of *disjoint-access-parallelism*. Disjoint-access-parallelism [53] means that processes accessing no common portion of the shared data should be able to progress in parallel. Another approach is to use a set of locks to protect the shared data, each associated with a small portion of the data. To avoid blocking problems arising from the mutual exclusion, a process executing an operation o_i must *help* the contending operation o_j complete before continuing to execute o_i.

However, lock-free implementations encounter a risk of starvation since the progress of other processes could cause one specific process to never finish. *Wait-free* [46] implementations are lock-free and moreover they avoid starvation. In a wait-free implementation, *each* operation is guaranteed to finish in a bounded number of steps, regardless of the actions of other concurrent operations on the same data. The basic idea is to improve the fairness (with respect to response time) among contending operations. Since lock-free/wait-free algorithms are guaranteed to progress regardless of process failures, the algorithms are strongly fault-tolerant. Lock-free/wait-free algorithms have been shown to be of big practical importance [70, 79, 100, 101].

Recently, a concept of *obstruction-freedom* has been proposed due to excessive helping overhead in lock-free/wait-free implementations. Obstruction-free implementations guarantee termination only in the absence of *step contention* [11], the number of concurrent processes whose steps are interleaved. The absence of

step contention does not preclude scenarios in which other processes are failed, swapped-out or have outstanding operations on the same data but are not accessing the data. Obstruction-free implementations avoid deadlock and priority-inversion arising from the mutual exclusion, but they encounter live-lock in the presence of step contention. The present solution for the live-lock is to use a *contention manager* [11, 36, 48, 108].

Since non-blocking algorithms do not involve mutual exclusion and allow processes to access shared data concurrently, their criteria for consistency correctness are more complex than those of mutual exclusion. A well-known correctness condition is *linearizability* [50], which means that a concurrent execution is correct if there is an equivalent sequential execution that preserves the partial order of the real execution. Linearizability requires that operations on the shared data appear to take effect atomically at a point of time in their execution interval.

2.3.1 Universal Constructions

Since implementing data structures in a non-blocking manner is notoriously sophisticated, substantial research effort has been devoted to develop *universal constructions* like *multi-word* atomic objects and software transactional memory, which alleviate the burden of transforming sequential implementations of data structures into non-blocking concurrent ones.

Herlihy [47] proposed a methodology for implementing concurrent data objects where interferences among processes are prevented by generating a private copy of the portions changed by each process. In the methodology, large objects are partitioned into portions (by designers) that are linked together using pointers. Each operation makes a *logical* copy of the object in which read-only portions are *physically* shared among the copies to minimize overhead. However, for common objects like FIFO queues implemented as linked lists of portions, operations like *enqueue* must copy the whole object since changing the next pointer of a portion results in copying the portion. The disadvantages of the methodology are the high cost of copying large objects and loss of *disjoint-access-parallelism* [53]. The disjoint-access-parallelism means that processes accessing no common portion of the shared data should be able to progress in parallel.

Barnes [14] later suggested a *cooperative technique* which allows many processes to access the same data structure concurrently as long as the processes write down exactly what they will do. Before modifying a portion of the shared data, a process p_1 checks whether this portion is used by another process p_2. If this is the case, p_1 will first cooperate with p_2 to complete p_2's work. Israeli and Rappoport [53] transformed this technique into more applicable one, where the concept of disjoint-access-parallelism was introduced. All processes try to lock all portions

of the shared data they need before writing back new values to the portions one by one. An *owner* field is assigned to every portion of the shared data to inform the processes which process is the current owner of the portion. The technique has subsequently been used to develop wait-free universal constructions [1, 5].

However, the disadvantage of the *cooperative technique* is that the process, which is being blocked by another process, does not release the portions of the shared data it is holding when helping the blocking process. If these portions were released, processes blocked on them would be able to progress. This *cooperative technique* uses a *recursive helping policy*, and the time needed for a blocked process p_1 to help another process p_2 may be long. The longer the response time of p_1, the larger the number of processes blocked by p_1. The processes blocked by p_1 will first help p_1 and then continue to help p_2 even when they and p_2 access disjoint parts of the shared data. This problem will be solved if p_1 does not conservatively keep its portions while helping the blocking process p_2.

Shavit and Touitou [92] realized the problem and presented a lock-free *software transactional memory* (STM). In STM, a process p_1 that is being blocked by a process p_2 releases its portions immediately before helping p_2. Moreover, a blocked process helps at most one blocking process, so *recursive helping* does not occur. Lock-free STM has been improved with respect to performance, space overhead and dynamic collections of shared data [29]. Recently, obstruction-free STM has been proposed [49, 75].

However, from efficiency point of view, STM is not an ideal technique to implement concurrent data structures since the blocked process releases its portions regardless of the contention level on them. That is, even if there is no other process requiring the portions at this time, it still releases them, and after helping the blocking process, it may have to compete with other processes to acquire the portions again. Moreover, even if a process is holding all portions it needs except for the last one, which is being held by another process, it still releases all its portions and then starts from scratch. In this case, it should realize that not many processes are waiting for its portions and that it is almost successful, so it should try to keep its current portions as in the *cooperative technique*.

2.3.2 Freshness

Concurrent data-structures/data-objects play a significant role in distributed computing. As a result, many of their aspects have been researched deeply such as consistency conditions [13,50,90], concurrency hierarchy [30] and fault-tolerance [74]. In this section, we introduce another aspect of concurrent data objects: freshness of states/values returned by read-operations of read-write objects.

Freshness is a significant property for shared data in general and has received

great attention in databases [18, 56, 85] as well as in caching systems [60, 65, 68]. Briefly, freshness is a yardstick to evaluate how fresh/new a value of a concurrent object returned by its read-operation is, when the object is updated and read concurrently. For concurrent data objects, although read-operations are allowed to return any value written by other concurrent operations, they are preferred to return the most fresh/latest one of these valid values, especially in reactive/detective systems. For instance, monitoring sensors continuously concurrently input data via a concurrent object and the processing unit periodically reads the data to make the system react accordingly. In such systems, the freshness of data influences how fast the system reacts to environment changes.

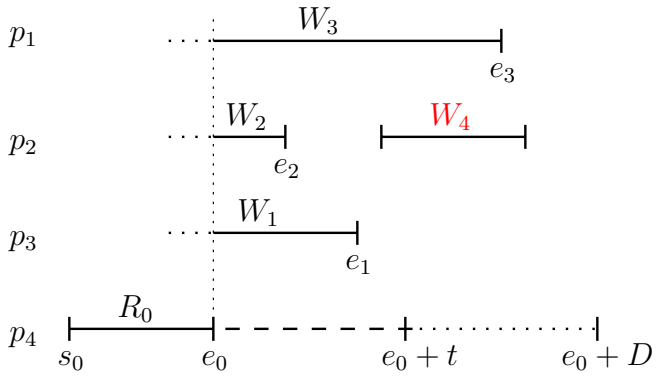

Figure 2.8: Freshness problem

Figure 2.8 illustrates the freshness problem. A read-operation R_0 runs concurrently with three write-operations W_1, W_2 and W_3 on the same object, where the execution-time upper-bound D of the (wait-free) read/write operations on the object is known. Each operation takes effect at an endpoint e_i (i.e. linearization point [50]) that happens at an unpredictable time before time D. At the endpoint e_0, the number of ongoing write-operations M is given. The freshness problem is to find a delay t, a real number in $[0, D]$, so that the new endpoint $e'_0 = e_0 + t$ of the read-operation R_0 has an optimal freshness value $f_t = \frac{k(\#we_t)}{h(t)}$, where $\#we_t$ is the number of further write-operation endpoints earned by the delay t, e.g. $\#we_D = M$; and k, h are increasing functions that reflect the relation between freshness and latency in real applications. The read-operation is only allowed to read the object data and check the number of ongoing write-operations. The write-operation is only allowed to write data to the object.

In Figure 2.8, read-operation R_0 earns two more endpoints e_1, e_2 of concurrent write-operations W_1, W_2 due to delaying endpoint e_0 by t, and thus returns a fresher value. Intuitively, if R_0 delays the endpoint e_0 by D, it will return the

freshest value at endpoint e_3[2]. However, from the application point of view the read-operation R_0 in this case will respond most slowly. Therefore, the goal in the freshness problem is to design read-operations that *respond fast* as well as *return fresh values*. Since there are two conflicting objectives for read-operations: *fast response* and *fresh value*, we define a measure of freshness as a function that is monotone increasing in the number of earned endpoints and decreasing in time.

The freshness problem is especially interesting in reactive/detective systems where monitoring sensors continuously and concurrently input data to the system via a concurrent data object and a processing unit periodically reads the data to make the system react accordingly. The period of reading data is denoted by T. If the read-operation R_0 of the processing unit returns data at e_0, the system will change its state according to the old data and will keep this state until the next period. If R_0 delays its endpoint e_0 by time $t < T$, the system will change its state according to the data at e_1, which means the system in this case may react faster to environment changes.

2.4 Distributed Data Structures for Counting and Balancing

As we have seen above, performance of concurrent data structures is much improved if the contention on the shared data is not too high. With this idea in mind, many researchers have focused on developing distributed data structures that can alleviate the contention on the shared data. These data structures distribute processes into small groups, each of which accesses a different part of the distributed shared data in a coordinated manner, for instance queue-locks [7, 32, 77], combining trees [31,111], counting pyramids [106,107], combining funnels [95], counting networks [8, 9] and diffracting trees [93, 94]. Whereas queue-locks aim at reducing contention on locks generally, the other sophisticated data structures focus on specific problems such as counting and balancing in order to enhance efficiency. Combining trees [31, 111] implement low-contention `fetch-and-`Φ operations by combining requests along paths upward to their root and subsequently distributing results downward to their leaves. The idea has been developed to counting pyramids [106, 107] that allow nodes to randomly forward their requests to a node on the next higher level and allow processors to select their initial level according to their request frequency. A similar idea has been used to develop combining funnels [95]. Opposite to the idea of combining requests, diffracting trees [93, 94] reduce contention for counting problems by distributing requests downward to the

[2]Note that R_0 only needs to consider write-endpoints of write operations that occur concurrently to R_0 in its original execution interval $[s_0, e_0]$, e.g. R_0 will ignore W_4.

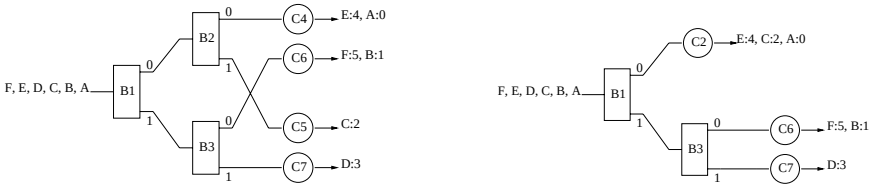

Figure 2.9: A diffracting tree (A) and a reactive diffracting tree (B).

leaves which each works as a counter in a coordinated manner. The trees have been developed to elimination trees [91] that are suited for stack and pool constructions. Another approach to solve counting problems is to use counting networks [8, 9], which ensure low contention at each node. The networks have been extended to *linearizable* [50] counting networks with timing assumptions [71, 76]. Empirical studies on the Proteus [17], a multiprocessor simulator, have showed that diffracting trees are superior to counting networks and combining trees under high loads [94].

Diffracting trees [93, 94] are well-known distributed data structures with the ability to distribute concurrent memory accesses to different memory banks in a coordinated manner. Each process(or) accessing the tree can be considered as leading a *token* that follows a path with intermediate nodes from the root to a leaf. Each node receives tokens from its single input (coming from its parent node) and sends tokens to its outputs. The node is called *balancer* and acts as a *toggle mechanism* that, given a stream of input tokens, alternately forwards them to its outputs, from left to right (i.e. send them to the left and right child nodes, respectively). The result is an even distribution of tokens at the leaves. In the trees, the contention at the root and balancers is alleviated using an *elimination technique* that evenly balances each pair of incoming tokens left and right without accessing the *toggle bit*. Diffracting trees have been introduced for *counting problems*, and hence their leaves are counters. The trees also guarantee the *step property*, which states that: when there are no tokens present inside the tree and if out_i denotes the number of tokens that have been output at leaf i, $0 \leq out_i - out_j \leq 1$ for any pair i and j of leaves such that $i < j$ (i.e. if one draws the tokens that have exited from each counter as a stack of boxes, the combined outcome will have the shape of a single step). Yet the fixed-size diffracting tree is optimal only for a small range of contention levels. To solve this problem, Della-Libera and Shavit [25] proposed *reactive diffracting trees*, where nodes can shrink (to a single counter) or grow (to a subtrees with counters as leaves) according to their local load.

Figure 2.9(A) depicts a diffracting tree. A set of processors $\{A, B, C, D, E, F\}$

is balanced on all leaves. Tokens passing one of these counters receive integer i, $i + 4$, $i + 2 * 4$, $\cdots$ where i is initial value of the counter. In the figure, processors A, B, C, D, E and F receive integers $0, 1, 2, 3, 4$ and 5, respectively. Even though the processors access separate shared data (counters), they still receive numbers that form a consecutive sequence of integers as if they accessed a centralized counter.

Trees (A) and (B) in Figure 2.9 depict the folding action of a reactive diffracting tree. Assume at the beginning the reactive diffracting tree has a shape like tree (A). If loads on two counters $C4$ and $C5$ are small, the sub-tree whose root is $B2$ shrinks to counter $C2$ as depicted in tree (B). After that, if processors A, B, C, D, E and F sequentially traverse the tree (B), three processors A, C and E will visit counter $C2$. That is, the latency for processors to go from the root to the counter decreases whereas the load on each counter is still kept low.

However, the reactive diffracting tree [25] uses a set of parameters to make its reactive decisions, namely folding/unfolding thresholds and the time interval for consecutive reactions. The parameter values depend on the multiprocessor system in use, the applications using the data structure and, in a multiprogramming environment, the system utilization by other concurrent programs. The parameters must be manually tuned using experimentation and information that is not easily available (e.g. future load characteristics). On the other hand, the tree can shrink or grow by only one level at a time, making multi-level adjustments costly.

As we know, the main challenge in designing reactive data structures in multiprocessor/ multiprogramming systems is to deal with unpredictable regular changes in the execution environment. The changes may make reactions obsolete at the time they take effect and consequently slow the system down. Therefore, reactive schemes using *fixed* parameters cannot be an optimal approach in *dynamic* environments such as multiprogramming systems. An ideal reactive data structure should not rely on experimentally tuned parameters and should react fast.

2.5 Online Algorithms

As mentioned in Chapter 1, our approach for reactive concurrent data structures to deal with unpredictability in execution environments is to synthesize non-blocking synchronization techniques and algorithmic techniques in the area of online algorithms. In this section, we give a brief introduction to online algorithms and their competitive analysis.

Online problems are optimization problems, where the input is received online and the output is produced online so that the cost of processing the input is minimum or the outcome is best. If we know the whole input in advance, we may

find an *optimal offline algorithm* OPT processing the whole input with the minimum cost. In order to evaluate how good an online algorithm is, the concept of *competitive ratio* has been suggested.

Competitive ratio: An online algorithm ALG is considered competitive with a competitive ratio c (or *c-competitive*) if there exists a constant β such that for any finite input I [16]:

$$ALG(I) \leq c \cdot OPT(I) + \beta \tag{2.1}$$

where $ALG(I)$ and $OPT(I)$ are the costs of the online algorithm ALG and the optimal offline algorithm OPT to service input I, respectively. The competitive ratio is a well-established concept and the comparison with the optimal off-line algorithm is natural in scenarios where either absolute performance measures are meaningless or assumptions on known probability distributions of some inputs are not feasible.

A common way to analyze an online algorithm, called *competitive analysis*, is to consider a game between an *online player* and a malicious *adversary*. In this game, i) the online player applies the online algorithm on the input generated by the adversary and ii) the adversary with the knowledge of the online algorithm tries to generate the worst possible input for the player. The input processing costs are very expensive for the online algorithm but relatively inexpensive for the optimal offline algorithm.

Adversary: For deterministic online algorithms, the adversary with knowledge of the online algorithms can generate the worst possible input to maximize the competitive ratio. However, the adversary cannot do that if the online player uses randomized algorithms. In randomized algorithms, depending on whether the adversary can observe the output from the online player to construct the next input, we classify the adversary into different categories. The adversary that constructs the whole input sequence in advance regardless of the output produced by the online player is called *oblivious adversary*. A randomized online algorithm is *c-competitive* against an oblivious adversary if

$$E[ALG(I)] \leq c \cdot OPT(I) + \beta \tag{2.2}$$

where $E[ALG(I)]$ is the expected cost of the randomized online algorithm ALG with the input I. The other adversary that observes the output produced by the online player so far and then based on that information constructs the next input element is called *adaptive adversary*. The adaptive adversary, however, is less natural and less practical for modeling real problems than the oblivious adversary.

The competitive analysis that uses the competitive ratio as a yardstick to evaluate algorithms is a valuable approach to resolve the problems where i) if we had

some information about the future, we could find an optimal solution, and ii) it is impossible to obtain such information.

Chapter 3

Reactive Multi-word Synchronization for Multiprocessors[1]

Phuong Hoai Ha[2], Philippas Tsigas[2]

Abstract

Shared memory multiprocessor systems typically provide a set of hardware primitives in order to support synchronization. Generally, they provide single-word *read-modify-write hardware primitives such as compare-and-swap, load-linked/store-conditional and fetch-and-op, from which higher-level synchronization operations are then implemented in software. Although the* single-word *hardware primitives are conceptually powerful enough to support higher-level synchronization, from the programmer's point of view they are not as useful as their generalizations, the* multi-word *objects.*

This paper presents two fast and reactive lock-free multi-word *compare-and-swap algorithms. The algorithms dynamically measure the level of contention as well as the memory conflicts of the* multi-word *compare-and-swap operations, and in response, they react accordingly in order to guarantee good performance in a wide range of system conditions. The algorithms are non-blocking (lock-free), allowing in this way fast dynamical behavior. Experiments on thirty processors*

[1]This paper appeared in the Journal of Instruction-Level Parallelism, Vol. 6, No. (Special Issue with selected papers from the 12th IEEE/ACM International Conference on Parallel Architectures and Compilation Techniques), Apr. 2004, AI Access Foundation and Morgan Kaufmann Publishers.

[2]Department of Computer Science and Engineering, Chalmers University of Technology, S-412 96 Gothenburg, Sweden. Email:{*phuong, tsigas*}*@cs.chalmers.se*

of an SGI Origin2000 multiprocessor have showed that both our algorithms react quickly to the contention variations and outperform the best known alternatives in almost all contention conditions.

3.1 Introduction

Synchronization is an essential point of hardware/software interaction. On one hand, programmers of parallel systems would like to be able to use high-level synchronization operations. On the other hand, the systems can support only a limited number of hardware synchronization primitives. Typically, the implementation of the synchronization operations of a system is left to the system designer, who has to decide how much of the functionality to implement in hardware and how much in software in system libraries. There has been a considerable debate about how much hardware support and which hardware primitives should be provided by the systems.

Consider the multi-word compare-and-swap operations (CASNs) that extend the single-word compare-and-swap operations from one word to many. A single-word compare-and-swap operation (CAS) takes as input three parameters: the address, an *old* value and a *new* value of a word, and atomically updates the contents of the word if its current value is the same as the *old* value. Similarly, an N-word compare-and-swap operation takes the addresses, *old* values and *new* values of N words, and if the current contents of these N words all are the same as the respective *old* values, the CASN will update the new values to the respective words atomically. Otherwise, we say that the CAS/CASN fails, leaving the variable values unchanged. It should be mentioned here that different processes might require different number of words for their compare-and-swap operations and the number is not a fixed parameter. Because of this powerful feature, CASN makes the design of concurrent data objects much more effective and easier than the single-word compare-and-swap [33–35]. On the other hand most multiprocessors support only single word compare-and-swap or compare-and-swap-like operations e.g. Load-Linked/Store-Conditional in hardware.

As it is expected, many research papers implementing the powerful CASN operation have appeared in the literature [5, 6, 44, 53, 81, 92]. Typically, in a CASN implementation, a CASN operation tries to lock all words it needs one by one. During this process, if a CASN operation is blocked by another CASN operation, then the process executing the blocked CASN may decide to help the blocking CASN. Even though most of the CASN designs use the helping technique to achieve the lock-free or wait-free property, the helping strategies in the designs are different. In the *recursive helping policy* [5, 44, 53], the CASN operation, which has been

blocked by another CASN operation, does not release the words it has acquired until its failure is definite, even though many other not conflicting CASNs might have been blocked on these words. On the other hand, in the *software transactional memory* [81,92] the blocked CASN operation immediately releases all words it has acquired regardless of whether there is any other CASN in need of these words at that time. In low contention situations, the release of all words acquired by a blocked CASN operation will only increase the execution time of this operation without helping many other processes. Moreover, in any contention scenario, if a CASN operation is close to acquiring all the words it needs, releasing all its acquired words will not only significantly increase its execution time but also increase the contention in the system when it tries to acquire these words again. The disadvantage of these strategies is that both of them are not adaptable to the different memory access patterns that different CASNs can trigger, or to frequent variations of the contention on each individual word of shared data. This can actually have a large impact on the performance of these implementations.

The idea behind the work described in this paper is that giving the CASN operation the possibility to adapt its helping policy to variations of contention can have a large impact on the performance in most contention situations. Of course, dynamically changing the behavior of the protocol comes with the challenge of performance. The overhead that the dynamic mechanism will introduce should not exceed the performance benefits that the dynamic behavior will bring.

The rest of this paper is organized as follows. We give a brief problem description, summarize the related work and give more detailed description of our contribution in Section 3.2. Section 3.3 presents our algorithms at an abstract level. The algorithms in detail are described in Section 3.4. Section 3.5 presents the correctness proofs of our algorithms. In Section 3.6 we present the performance evaluation of our CASN algorithms and compare them to the best known alternatives, which also represent the two helping strategies mentioned above. Finally, Section 3.7 concludes the paper.

3.2 Problem Description, Related Work and Our Contribution

Concurrent data structures play a significant role in multiprocessor systems. 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 as it causes blocking, i.e. other concurrent operations cannot make any progress while the access to the shared resource is blocked by the lock. Using mutual exclusion can also cause deadlocks,

priority inversion 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. *Wait-free* [46] 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 [100, 101], and recently NOBLE, which is a non-blocking inter-process communication library, has been introduced [98].

The main problem of lock/wait-free concurrent data structures is that many processes try to read and modify the same portions of the shared data at the same time and the accesses must be atomic to one another. That is why a multi-word compare-and-swap operation is so important for such data structures.

Herlihy proposed a methodology for implementing concurrent data structures where interferences among processes are prevented by generating a private copy of the portions changed by each process [47]. The disadvantages of the methodology are the high cost of copying large objects and loss of disjoint-access-parallelism. The *disjoint-access-parallelism* means that processes accessing no common portion of the shared data should be able to progress in parallel.

Barnes [14] later suggested a *cooperative technique* which allows many processes to access the same data structure concurrently as long as the processes write down exactly what they will do. Before modifying a portion of the shared data, a process p_1 checks whether this portion is used by another process p_2. If this is the case, p_1 will first cooperate with p_2 to complete p_2's work.

Israeli and Rappoport transformed this technique into more applicable one [53], where the concept of disjoint-access-parallelism was introduced. All processes try to lock all portions of the shared data they need before writing back new values to the portions one by one. An *owner* field is assigned to every portion of the shared data to inform the processes which process is the current owner of the portion.

Harris, Fraser and Pratt [44] aiming to reduce the per-word space overhead eliminated the owner field. They exploited the data word for containing a special value, a pointer to a CASNDescriptor, to pass the information of which process is the owner of the data word. However, in their paper the memory management problem is not discussed clearly.

A wait-free multi-word compare-and-swap was introduced by Anderson and

Moir in [5]. The cooperative technique was employed in the aforementioned results as well.

However, the disadvantage of the *cooperative technique* is that the process, which is being blocked by another process, does not release the words it is holding when helping the blocking process, even though many other processes blocked on these words might be able to make progress if these words were released. This *cooperative technique* uses a *recursive helping policy*, and the time needed for a blocked process p_1 to help another process p_2 may be long. The longer the response time of p_1, the larger the number of processes blocked by p_1. The processes blocked by p_1 will first help p_1 and then continue to help p_2 even when they and p_2 access disjoint parts of the data structure. This problem will be solved if p_1 does not conservatively keep its words while helping the blocking process p_2.

The left part in Figure 3.1 illustrates the helping strategy of the *recursive helping policy*. There are three processes executing three CAS4: p_1 wants to lock words 1,2,3,4; p_2 wants to lock words 3,6 and two other words; and p_3 wants to lock words 6,7,8 and another word. At that time, the first CAS4 acquired words 1 and 2, the second CAS4 acquired word 3 and the third CAS4 acquired words 6,7 and 8. When process p_1 helps the first CAS4, it realizes that word 3 was acquired by the second CAS4 and thus it helps the second CAS4. Then, p_1 realizes that word 6 was acquired by the third CAS4 and it continues to help the third CAS4 and so on. We observe that i) the time for a process to help other CASN operations may be long and unpredictable and ii) if the second CAS4 did not conservatively keep word 3 while helping other CAS4, the first CAS4 could succeed without helping other CAS4s, especially the third CAS4 that did not block the first CAS4. Note that helping causes more contention on the memory. Therefore, the less helping is used, the lower the contention level on the memory is.

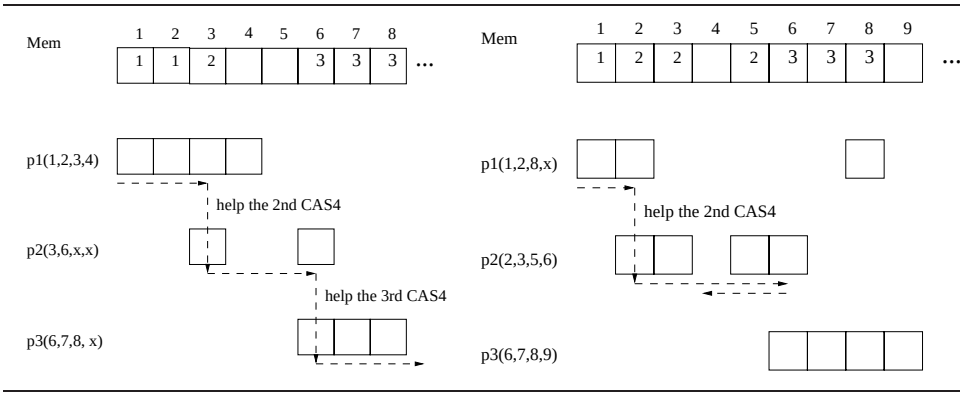

Figure 3.1: Recursive helping policy and software transactional memory

Shavit and Touitou realized the problem above and presented the *software transactional memory* (STM) in [92]. In STM, a process p_1 that is being blocked by a process p_2 releases the words it owns immediately before helping p_2. Moreover, a blocked process helps at most one blocking process, so *recursive helping* does not occur. STM then was improved by Moir [81], who introduced a design of a *conditional* wait-free multi-word compare-and-swap operation. An evaluating function passed to the CASN by the user will identify whether the CASN will retry when the contention occurs. Nevertheless, both STM and the improved version (iSTM) also have the disadvantage that the blocked process releases the words it owns regardless of the contention level on the words. That is, even if there is no other process requiring the words at that time, it still releases the words, and after helping the blocking process, it may have to compete with other processes to acquire the words again. Moreover, even if a process acquired the whole set of words it needs except for the last one, which is owned by another process, it still releases all the words and then starts from scratch. In this case, it should realize that not many processes require the words and that it is almost successful, so it would be best to try to keep the words as in the *cooperative technique*.

The right part of Figure 3.1 illustrates the helping strategy of STM. At that time, the first CAS4 acquired word 1, the second CAS4 acquired words 2,3 and 5 and the third CAS4 acquired words 6,7 and 8. When process p_1 helps the first CAS4, it realizes that word 2 was acquired by the second CAS4. Thus, it releases word 1 and helps the second CAS4. Then, when p_1 realizes that the second CAS4 was blocked by the third CAS4 on word 6, it, on behalf of the second CAS4, releases word 5,3 and 2 and goes back to help the first CAS4. Note that i) p_1 could have benefited by keeping word 1 because no other CAS4 needed the word; otherwise, after helping other CAS4s, p_1 has to compete with other processes to acquire word 1 again; and ii) p_1 should have tried to help the second CAS4 a little bit more because this CAS4 operation was close to success.

Note that most algorithms require the N words to be sorted in addresses and this can add an overhead of $O(logN)$ because of sorting. However, most applications can sort these addresses before calling the CASN operations.

3.2.1 Our Contribution

All available CASN implementations have their weak points. We realized that the weaknesses of these techniques came from their static helping policies. These techniques do not provide the ability to CASN operations to measure the contention that they generate on the memory words, and more significantly to reactively change their helping policy accordingly. We argue that these weaknesses are not fundamental and that one can in fact construct multi-word compare-and-swap algorithms

where the CASN operations: i) measure in an efficient way the contention that they generate and ii) reactively change the helping scheme to help more efficiently the other CASN operations.

Synchronization methods that perform efficiently across a wide range of contention conditions are hard to design. Typically, *small* structures and *simple* methods fit better low contention levels while *bigger* structures and more *complex* mechanisms can help to distribute processors/processes among the memory banks and thus alleviate memory contention.

The key to our first algorithm is for every CASN to release the words it has acquired only if the average contention on the words becomes too high. This algorithm also favors the operations closer to completion. The key to our second algorithm is for a CASN to release not *all* the words it owns at once but *just enough* so that most of the processes blocked on these words can progress. The performance evaluation of the proposed algorithms on thirty processors of an SGI Origin2000 multiprocessor, which is presented in Section 3.6, matches our intuition. In particular, it shows that both our algorithms react fast according to the contention conditions and significantly outperform the best-known alternatives in all contention conditions.

3.3 Algorithm Informal Description

In this section, we present the ideas of our reactive CASN operations at an abstract level. The details of our algorithms are presented in the next section.

In general, practical CASN operations are implemented by locking all the N words and then updating the value of each word one by one accordingly. Only the process having acquired all the N words it needs can try to write the new values to the words. The processes that are blocked, typically have to *help* the blocking processes so that the lock-free feature is obtained. The helping schemes presented in [5,44,53,81,92] are based on different strategies that are described in Section 3.2.

The variable $OP[i]$ described in Figure 3.2 is the shared variable that carries the data of $CASN_i$. It consists of three arrays with N elements each: $addr_i$, exp_i and new_i and a variable $blocked_i$ that contain the addresses, the old values, the new values of the N words that need to be compared-and-swapped atomically and the number of CASN operations blocked on the words, respectively. The N elements of array $addr_i$ must be increasingly sorted in addresses to avoid live-lock in the helping process. Each entry of the shared memory Mem, a normal 32-bit word used by the real application, has two fields: the $value$ field (24 bits) and the $owner$ field (8 bits). The $owner$ field needs $log_2 P + 1$ bits, where P is the number of processes in the system. The $value$ field contains the real value of the word while

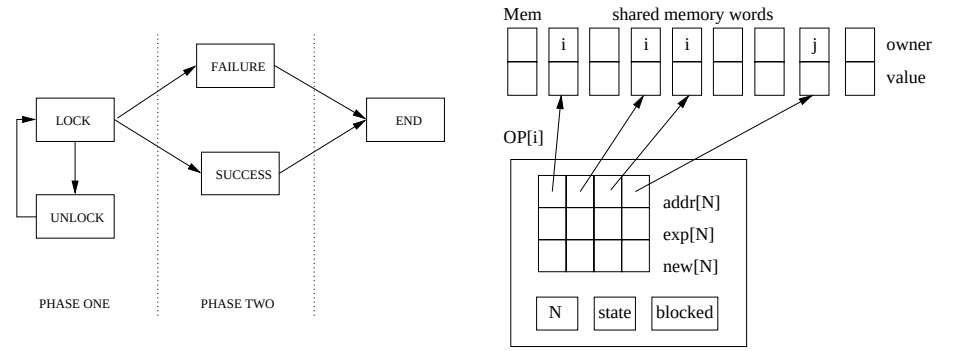

Figure 3.2: Reactive-CASN states and reactive-CAS4 data structure

the $owner$ field contains the identity of the CASN operation that has acquired the word. For a system supporting 64-bit words, the value field can be up to 56 bits if $P < 256$. However, the value field cannot contain a pointer to a memory location in some modern machines where the size of pointer equals the size of the largest word. For information on how to eliminate the owner field, see [44].

Each CASN operation consists of two phases as described in Figure 3.2. The first phase has two states $Lock$ and $Unlock$ and it tries to lock all the necessary words according to our reactive helping scheme. The second one has also two states $Failure$ and $Success$. The second phase updates or releases the words acquired in the first phase according to the result of phase 1.

Figure 3.3 describes our CASN operation at a high level.

CASN$(OP[i])$
try_again :
Try to lock all N necessary words;
if *manage to lock all the N words* **then**
write new values to these words one by one; **return** $Success$;
else if *read an unexpected value* **then**
release all the words that have been locked by $CASN_i$; **return** $Failure$;
else if *contention is "high enough"* **then**
release some/all $CASN_i$*'s words to reduce contention*; **goto** try_again;

Figure 3.3: Reactive CASN description

In order to know whether the contention on $CASN_i$'s words is high, each $CASN_i$ uses variable $OP[i].blocked$ to count how many other CASNs are being blocked on its words. Now, which contention levels should be considered *high*?

The $CASN_i$ has paid a price (execution time) for the number of words that it has acquired and thus it should not yield these words to other CASNs too generously as the *software transactional memory* does. However, it should not keep these words egoistically as in the *cooperative technique* because that will make the whole system slowdown. Let w_i be the number of words currently kept by $CASN_i$ and n_i be the estimated number of CASNs that will go ahead if some of w_i words are released. The $CASN_i$ will consider releasing its words only if it is blocked by another CASN. The challenge for $CASN_i$ is to *balance the trade-off* between its own progress w_i and the potential progress n_i of the other processes. If $CASN_i$ knew how the contention on its w_i words will change in the future from the time $CASN_i$ is blocked to the time $CASN_i$ will be unblocked, as well as the time $CASN_i$ will be blocked, $CASN_i$ would have been able to make an optimal trade-off. Unfortunately, there is no way for $CASN_i$ to have this kind of information.

The terms used in the section are summarized in the following table:

Terms	Meanings
P	the number of processes in the system
N	the number of words needing to be updated atomically
w_i	the number of words currently kept by $CASN_i$
$blocked_i$	the number CASN operations blocked by $CASN_i$
r_i	the average contention on words currently kept by $CASN_i$
m	the lower bound of average contention r_i
M	the upper bound of average contention r_i
c	the competitive ratio

Figure 3.4: The term definitions

3.3.1 The First Algorithm

Our first algorithm concentrates on the question of when $CASN_i$ should release all the w_i words that it has acquired. A simple solution is as follows: if the average contention on the w_i words, $r_i = \frac{blocked_i}{w_i}$, is greater than a certain threshold, $CASN_i$ releases all its w_i words to help the *many* other CASNs to go ahead. However, how big should the threshold be in order to optimize the trade-off? In our first reactive CASN algorithm, the threshold is calculated in a similar way to the *reservation price policy* [28]. This policy is an optimal deterministic solution for the online search problem where a player has to decide whether to exchange his dollars to yens at the current exchange rate or to wait for a better one without knowing how the exchange rate will vary.

Let P and N be the number of processes in the system and the number of words needing to be updated atomically, respectively. Because $CASN_i$ only checks the release-condition when: i) it is blocked and ii) it has locked at least a word and blocked at least a CASN, we have that $1 \leq blocked_i \leq (P-2)$ and $1 \leq w_i \leq (N-1)$. Therefore, $m \leq r_i \leq M$ where $m = \frac{1}{N-1}$ and $M = P-2$. Our *reservation contention policy* is as follows:

Reservation contention policy: $CASN_i$ releases its w_i words when the average contention r_i is greater than or equal to $R^* = \sqrt{Mm}$.

The policy is $\sqrt{\frac{M}{m}}$-competitive. For the proof and more details on the policy, see *reservation price policy* [28].

Beside the advantage of reducing collision, the algorithm also favors CASN operations that are closer to completion, i.e w_i is larger, or that cause a small number of conflicts, i.e. $blocked_i$ is smaller. In both cases, r_i becomes smaller and thus $CASN_i$ is unlikely to release its words.

3.3.2 The Second Algorithm

Our second algorithm decides not only when to release the $CASN_i$'s words but also how many words need to be released. It is intuitive that the $CASN_i$ does not need to release all its w_i words but releases *just enough* so that most of the CASN operations blocked by $CASN_i$ can go ahead.

The second reactive scheme is influenced by the idea of the *threat-based algorithm* [28]. The algorithm is an optimal solution for the one-way trading problem, where the player has to decide whether to accept the current exchange rate as well as how many of his/her dollars should be exchanged to yens at the current exchange rate.

Definition 3.3.1. *A* transaction *is the interval from the time a CASN operation is blocked to the time it is unblocked and acquires a new word*

In our second scheme, the following rules must be satisfied in a *transaction*. According to the threat-based algorithm [28], we can obtain an optimal competitive ratio for unknown duration variant $c = \varphi - \frac{\varphi-1}{\varphi^{1/(\varphi-1)}}$ if we know only φ, where $\varphi = \frac{M}{m}$; m and M are the lower bound and upper bound of the average contention on the words acquired by the CASN as mentioned in subsection 3.3.1, respectively:

1. Release words only when the current contention is greater than $(m * c)$ and is the highest so far.

2. When releasing, release *just enough* words to keep the competitive ratio c.

Similar to the threat-based algorithm [28], the number of words which should be released by a blocked $CASN_i$ at time t is $d_i^t = D_i * \frac{1}{c} * \frac{r_i^t - r_i^{t-1}}{r_i^t - m}$, where r_i^{t-1} is the highest contention until time $(t-1)$ and D_i is the number of words acquired by the $CASN_i$ at the beginning of the transaction. In our algorithm, r_i stands for the average contention on the words kept by a $CASN_i$ and is calculated by the following formula: $r_i = \frac{blocked_i}{w_i}$ as mentioned in Section 3.3.1. Therefore, when $CASN_i$ releases words with contention smaller than r_i, the average contention at that time, the next average contention will increase and $CASN_i$ must continue releasing words in decreasing order of word-indices until the word that made the average contention increase is released. When this word is released, the average contention on the words locked by $CASN_i$ is going to reduce, and thus according to the first of the previous rules, $CASN_i$ does not release its remaining words anymore at this time. That is how *just enough* to help most of the blocked processes is defined in our setting.

Therefore, beside the advantage of reducing collision, the second algorithm favors to release the words with high contention.

3.4 Implementations

In this section, we describe our reactive multi-word compare-and-swap implementations.

The synchronization primitives related to our algorithms are *fetch-and-add (FAA), compare-and-swap (CAS)* and *load-linked/validate/store-conditional (LL/VL/SC)*. The definitions of the primitives are described in Figure 3.5, where x is a variable and v, old, new are values.

For the systems that support *weak LL/SC* such as the SGI Origin2000 or the systems that support *CAS* such as the SUN multiprocessor machines, we can implement the *LL/VL/SC* instructions algorithmically [80].

3.4.1 First Reactive Scheme

The part of a CASN operation ($CASN_i$) that is of interest for our study is the part that starts when $CASN_i$ is blocked while trying to acquire a new word after having acquired some words; and ends when it manages to acquire a new word. This word could have been locked by $CASN_i$ before $CASN_i$ was blocked, but then released by $CASN_i$. Our first reactive scheme decides whether and when the $CASN_i$ should release the words it has acquired by measuring the *contention* r_i on the words it has acquired, where $r_i = \frac{blocked_i}{kept_i}$ and $blocked_i$ is the number of processes blocked on the $kept_i$ words acquired by $CASN_i$. If the contention

```
FAA(x, v)
atomically {
    oldx ← x;
    x ← x + v;
    return(oldx)
    }

CAS(x, old, new)
atomically {
    if(x = old)
      x ← new;
      return(true);
    else return(false);
    }
```

```
LL(x){
    return the value of x such that it may
    be subsequently used with SC
    }

VL(x)
atomically {
    if (no other process has written to x
    since the last LL(x))
      return(true);
    else return(false);
    }

SC(x, v)
atomically {
    if (no other process has written to x
    since the last LL(x))
      x ← v; return(true);
    else return(false);
    }
```

Figure 3.5: Synchronization primitives

```
type word_type = record value; owner; end; /*normal 32-bit words*/
     state_type = {Lock, Unlock, Succ, Fail, Ends, Endf};
     para_type = record N: integer; addr: array[1..N] of *word_type ;
                        exp, new: array[1..N] of word_type; /*CASN*/
                        state: state_type; blocked : 1..P; end; /*P: #processes*/
     return_type = record kind:{Succ, Fail, Circle}; cId:1..P; end;
                        /*cId: Id of a CASN participating in a circle-help*/
shared var Mem: set of word_type; OP: array[1..P] of para_type;
           Version: array[1..P] of unsigned long;
private var casn_l: array[1..P] of 1..P;
           /*keeping CASNs currently helped by the process*/
           l: of 1..P; /*the number of CASNs currently helped by the process*/
```

Figure 3.6: Data structures in our first reactive multi-word compare-and-swap algorithm

r_i is higher than a *contention threshold* R^*, process p_i releases all the words. The *contention threshold* R^* is computed according to the *reservation contention policy* in Section 3.3.1. One interesting feature of this reactive helping method is that it favors processes closer to completion as well as processes with a small number of conflicts.

At the beginning, the CASN operation starts phase one in order to lock the N words. Procedure $Casn$ tries to lock the words it needs by setting the state of the CASN to $Lock$ (line 1 in $Casn$). Then, procedure $Help$ is called with four parameters: i) the identity of helping-process $helping$, ii) the identity of helped-CASN i, iii) the position from which the process will help the CASN lock words pos, and iv) the version ver of current variable $OP[i]$. In the $Help$ procedure, the $helping$ process chooses a correct way to help $CASN_i$ according to its state. At the beginning, $CASN_i$'s state is $Lock$. In the $Lock$ state, the $helping$ process tries to help $CASN_i$ lock all necessary words:

- If the $CASN_i$ manages to lock all the N words successfully, its state changes into $Success$ (line 7 in $Help$), then it starts phase two in order to conditionally write the new values to these words (line 10 in $Help$).

- If the $CASN_i$, when trying to lock all the N words, discovers a word having a value different from its old value passed to the CASN, its state changes into $Failure$ (line 8 in $Help$) and it starts phase two in order to release all the words it locked (line 11 in Help).

- If the $CASN_i$ is blocked by another $CASN_j$, it checks the unlock-condition before helping $CASN_j$ (line 6 in $Locking$). If the unlock-condition is satisfied, $CASN_i$'s state changes into $Unlock$ (line 3 in $CheckingR$) and it starts to release the words it locked (line 3 in $Help$).

Procedure $Locking$ is the main procedure in phase one, which contains our first reactive scheme. In this procedure, the process called $helping$ tries to lock all N necessary words for $CASN_i$. If one of them has a value different from its expected value, the procedure returns $Fail$ (line 4 in $Locking$). Otherwise, if the value of the word is the same as the expected value and it is locked by another CASN (lines 6-15 in $Locking$) and at the same time $CASN_i$ satisfies the unlock-condition, its state changes into $Unlock$ (line 3 in $CheckingR$). That means that other processes whose CASNs are blocked on the words acquired by $CASN_i$ can, on behalf of $CASN_i$, unlock the words and then acquire them while the $CASN_i$'s process helps its blocking CASN operation, $CASN_{x.owner}$ (line 13 in $Locking$).

Procedure $CheckingR$ checks whether the average contention on the words acquired by $CASN_i$ is high and has passed a threshold: the unlock-condition. In

```
/* Version[i] must be increased by one before OP[i] is used to contain parameters
addr[], exp[] and new[] for Casn(i) */
state_type CASN(i)
begin
1:   OP[i].blocked := 0; OP[i].state := Lock; for j := 1 to P do casn_l[j] = 0;
2:   l := 1; casn_l[l] := i; /*record CASN_i as a currently helped one*/
3:   Help(i, i, 0, Version[i]);
     return OP[i].state;
end.

return_type HELP(helping, i, pos, ver)
begin
start :
1:   state := LL(&OP[i].state);
2:   if (ver ≠ Version[i]) then return (Fail, nil);
     if (state = Unlock) then /*CASN_i is in state Unlock*/
3:     Unlocking(i);
       if (helping = i) then /*helping is CASN_i's original process*/
4:       SC(&OP[i].state, Lock); goto start; /*help CASN_i*/
       else /*otherwise, return to previous CASN*/
5:       FAA(&OP[i].blocked, −1); return (Succ, nil);
     else if (state = Lock) then
6:     result := Locking(helping, i, pos);
       if (result.kind = Succ) then /*Locking N words successfully*/
7:       SC(&OP[i].state, Succ); /*change its state to Success*/
       else if (result.kind = Fail) then /*Locking unsuccessfully*/
8:       SC(&OP[i].state, Fail); /*change its state to Failure*/
       else if (result.kind = Circle) then /*the circle help occurs*/
9:       FAA(&OP[i].blocked, −1);
         return result; /*return to the expected CASN*/
       goto start;
     else if (state = Succ) then
10:    Updating(i); SC(&OP[i].state, Ends); /*write new values*/
     else if (state = Fail) then
11:    Releasing(i); SC(&OP[i].state, Endf); /*release its words*/
     return (Succ, nil);
end.
```

Figure 3.7: Procedures CASN and Help in our first reactive multi-word compare-and-swap algorithm

```
return_type LOCKING(helping, i, pos)
begin
start:
      for j := pos to OP[i].N do /*only help from position pos*/
1:        e.addr = OP[i].addr[j]; e.exp = OP[i].exp[j];
      again:
2:        x := LL(e.addr);
3:        if (not VL(&OP[i].state)) then
            return (nil, nil); /*return to read its new state*/
4:        if (x.value ≠ e.exp) then return (Fail, nil); /*x was updated*/
          else if (x.owner = nil) then /*x is available*/
5:          if (not SC(e.addr, (e.exp, i)) then goto again;
          else if (x.owner ≠ i) then /*x is locked by another CASN*/
6:          CheckingR(i, OP[i].blocked, j − 1); /*check unlock-condition*/
7:          if (x.owner in casn_l) then return (Circle, x.owner); /*circle-help*/
8:          Find index k: OP[x.owner].addr[k] = e.addr;
9:          ver = Version[x.owner];
10:         if ( not VL(e.addr)) then goto again;
11:         FAA(&OP[x.owner].blocked, 1);
12:         l := l + 1; casn_l[l] := x.owner; /*record x.owner*/
13:         r := Help(helping, x.owner, k, ver);
14:         casn_l[l] := 0; l := l − 1; /*omit x.owner's record*/
15:         if ((r.kind = Circle) and (r.cId ≠ i)) then
              return r; /*CASN_i is not the expected CASN in the circle help*/
            goto start;
      return (Succ, nil);
end.

CHECKINGR(owner, blocked, kept)
begin
1:    if ((kept = 0) or (blocked = 0)) then return;
2:    if (not VL(&OP[owner].state)) then return;
3:    if ((blocked/kept) > R*) then SC(&OP[owner].state, Unlock);
      return;
end.
```

Figure 3.8: Procedures Locking and CheckingR in our first reactive multi-word compare-and-swap algorithm

```
UPDATING(i)
begin
    for j := 1 to OP[i].N do
1:      e.addr = OP[i].addr[j]; e.exp = OP[i].exp[j];
2:      e.new = OP[i].new[j];
    again :
3:      x := LL(e.addr);
4:      if (not VL(&OP[i].state)) then return;
        if (x = (e.exp, i)) then /*x is expected value & locked by CASN_i*/
5:        if ( not SC(e.addr, (e.new, nil)) then goto again;
    return;
end.

UNLOCKING(i)/RELEASING(i)
begin
    for j := OP[i].N downto 1 do
1:      e.addr = OP[i].addr[j]; e.exp = OP[i].exp[j];
    again :
2:      x := LL(e.addr);
3:      if not VL(&OP[i].state) then return;
        if (x = (e.exp, nil)) or (x = (e.exp, i)) then
4:        if (not SC(e.addr, (e.exp, nil)) then goto again;
    return;
end.
```

Figure 3.9: Procedures Updating and Unlocking/Releasing in our first reactive multi-word compare-and-swap algorithm

this implementation, the *contention threshold* is R^*, $R^* = \sqrt{\frac{P-2}{N-1}}$, where P is the number of concurrent processes and N is the number of words that need to be updated atomically by CASN.

At time t, $CASN_i$ has created average contention r_i on the words that it has acquired, $r_i = \frac{blocked_i}{kept_i}$, where $blocked_i$ is the number of CASNs currently blocked by $CASN_i$ and $kept_i$ is the number of words currently locked by $CASN_i$. $CASN_i$ only checks the unlock-condition when: i) it is blocked and ii) it has locked at least a word and blocked at least a process (line 1 in $CheckingR$). The unlock-condition is to check whether $\frac{blocked_i}{kept_i} \geq R^*$. Every process blocked by $CASN_i$ on word $OP[i].addr[j]$ increases $OP[i].blocked$ by one before helping $CASN_i$ using a fetch-and-add operation (FAA) (line 11 in $Locking$), and decreases the variable by one when it returns from helping the $CASN_i$ (line 5 and 9 in $Help$). The variable is not updated when the state of the $CASN_i$ is $Success$ or $Failure$ because

in those cases $CASN_i$ no longer needs to check the unlock-condition.

There are two important variables in our algorithm, the $Version$ and $casn\_l$ variables. These variables are defined in Figure 3.6.

The variable $Version$ is used for memory management purposes. That is when a process completes a CASN operation, the memory containing the CASN data, for instance $OP[i]$, can be used by a new CASN operation. Any process that wants to use $OP[i]$ for a new CASN must firstly increase the $Version[i]$ and pass the version to procedure Help (line 3 in $Casn$). When a process decides to help its blocking CASN, it must identify the current version of the CASN (line 9 and 10 in $Locking$) to pass to procedure $Help$ (line 13 in $Locking$). Assume process p_i is blocked by $CASN_j$ on word $e.addr$, and p_i decides to help $CASN_j$. If the version p_i reads at line 9 is not the version of $OP[j]$ at the time when $CASN_j$ blocked p_i, that is $CASN_j$ has ended and $OP[j]$ is re-used for another new CASN, the field $owner$ of the word has changed. Thus, command $VL(e.addr)$ at line 10 returns failure and p_i must read the word again. This ensures that the version passed to $Help$ at line 13 in procedure $Locking$ is the version of $OP[j]$ at the time when $CASN_j$ blocked p_i. Before helping a CASN, processes always check whether the CASN version has changed (line 2 in $Help$).

The other significant variable is $casn\_l$, which is local to each process and is used to trace which CASNs have been helped by the process in order to avoid the circle-helping problem. Consider the scenario described in Figure 3.10. Four processes p_1, p_2, p_3 and p_4 are executing four CAS3 operations: $CAS3_1$, $CAS3_2$, $CAS3_3$ and $CAS3_4$, respectively. The $CAS3_i$ is the CAS3 that is initiated by process p_i. At that time, $CAS3_2$ acquired $Mem[1]$, $CAS3_3$ acquired $Mem[2]$ and $CAS3_4$ acquired $Mem[3]$ and $Mem[4]$ by writing their original helping process identities in the respective owner fields (recall that Mem is the set of *separate* words in the shared memory, not an array). Because p_2 is blocked by $CAS3_3$ and $CAS3_3$ is blocked by $CAS3_4$, p_2 helps $CAS3_3$ and then continues to help $CAS3_4$. Assume that while p_2 is helping $CAS3_4$, another process discovers that $CAS3_3$ satisfies the unlock-condition and releases $Mem[2]$, which was blocked by $CAS3_3$; p_1, which is blocked by $CAS3_2$, helps $CAS3_2$ acquire $Mem[2]$ and then acquire $Mem[5]$. Now, p_2, when helping $CAS3_4$ lock $Mem[5]$, realizes that the word was locked by $CAS3_2$, its own CAS3, that it has to help now. Process p_2 has made a cycle while trying to help other CAS3 operations. In this case, p_2 should return from helping $CAS3_4$ and $CAS3_3$ to help its own CAS3, because, at this time, the $CAS3_2$ is not blocked by any other CAS3. The local arrays $casn_l$s are used for this purpose. Each process p_i has a local array $casn\_l_i$ with size of the maximal number of CASNs the process can help at one time. Recall that at one time each process can execute only one CASN, so the number is not greater than P, the number of processes in the system. In our implementation, we set the size

of arrays $casn\_l$ to P, i.e. we do not limit the number of CASNs each process can help.

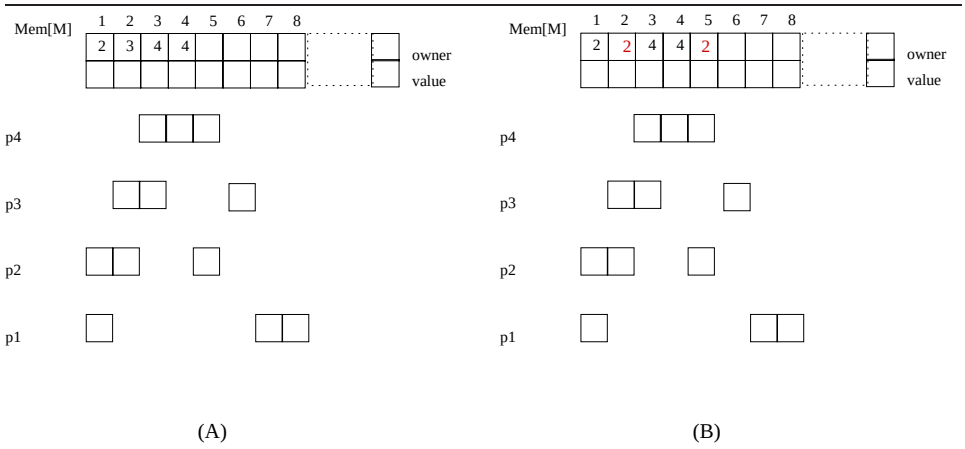

Figure 3.10: Circle-helping problem: (A) Before helping; (B) After p_1 helps $CAS3_2$ acquire $Mem[2]$ and $Mem[5]$.

An element $casn\_l_i[l]$ is set to j when process p_i starts to help a $CASN_j$ initiated by process p_j and the $CASN_j$ is the l^{th} CASN that process p_i is helping at that time. The element is reset to 0 when process p_i completes helping the l^{th} CASN. Process p_i will realize the circle-helping problem if the identity of the CASN that process p_i intends to help has been recorded in $casn\_l_i$.

In procedure $Locking$, before process $p_{helping}$ helps $CASN_{x.owner}$, it checks whether it is helping the CASN currently (line 7 in $Locking$). If yes, it returns from helping other CASNs until reaching the unfinished helping task on $CASN_{x.owner}$ (line 15 in $Locking$) by setting the returned value $(Circle, x.owner)$ (line 7 in $Locking$). The l^{th} element of the array is set to $x.owner$ before the process helps $CASN_{x.owner}$, its l^{th} CASN, (line 12 in $Locking$) and is reset to zero after the process completes the help (line 14 in $Locking$).

In our methods, a process helps another CASN, for instance $CASN_i$, *just enough* so that its own CASN can progress. The strategy is illustrated by using the variable $casn\_l$ above and helping the $CASN_i$ unlock its words. After helping the $CASN_i$ release all its words, the process returns immediately because at this time the CASN blocked by $CASN_i$ can go ahead (line 5 in $Help$). After that, no process helps $CASN_i$ until the process that initiated it, p_i, returns and helps it progress (line 4 in $Help$).

3.4.2 Second Reactive Scheme

In the first reactive scheme, a $CASN_i$ must release all its acquired words when it is blocked and the average contention on these words is higher than a threshold, R^*. It will be more flexible if the $CASN_i$ can release only some of its acquired words on which many other CASNs are being blocked.

The second reactive scheme is more adaptable to contention variations on the shared data than the first one. An interesting feature of this method is that when $CASN_i$ is blocked, it only releases *just enough* words to reduce most of CASNs blocked by itself.

According to rule 1 of the second reactive scheme as described in Section 3.3.2 , contention r_i is considered for adjustment only if it increases, i.e. when either the number of processes blocked on the words kept by $CASN_i$ increases or the number of words kept by $CASN_i$ decreases. Therefore, in this implementation, which is described in Figure 3.11 and Figure 3.12, the procedure $CheckingR$ is called not only from inside the procedure $Locking$ as in the first algorithm, but also from inside the procedure $Help$ when the number of words locked by $CASN_i$ reduces (line 5). In the second algorithm, the variable $OP[i].blocked$[1] is an array of size N, the number of words need to be updated atomically by CASN. Each element of the array $OP[i].blocked[j]$ is updated in such a way that the number of CASNs blocked on each word is known, and thus a process helping $CASN_i$ can calculate how many words need to be released in order to release *just enough* words. To be able to perform this task, besides the information about contention r_i, which is calculated through variables $blocked_i$ and $kept_i$, the information about the highest r_i so far and the number of words locked by $CASN_i$ at the beginning of the transaction is needed. This additional information is saved in two new fields of $OP[i].state$ called r_{max} and $init$, respectively. While the $init$ is updated only one time at the beginning of the transaction (line 3 in $CheckingR$), the r_{max} field is updated whenever the unlock-condition is satisfied (line 3 and 5 in $CheckingR$). The beginning of a transaction is determined by comparing the number of word currently kept by the CASN, $kept$, and its last unlock-position, $gs.ul\_pos$ (line 2 in $CheckingR$). The values are different only if the CASN has acquired more words since the last time it was blocked, and thus in this case the CASN is at the beginning of a new transaction according to definition 3.3.1.

After calculating the number of words to be released, the position from which the words are released is saved in field ul_pos of $OP[i].state$ and it is called ul_pos_i. Consequently, the process helping $CASN_i$ will only release the words from $OP[i].addr[ul\_pos_i]$ to $OP[i].addr[N]$ through the procedure $Unlocking$.

[1] In our implementation, the array *blocked* is simple a 64-bit word such that it can be read in one atomic step. In general, the whole array can be read atomically by a snapshot operation.

```
type state_type = record init; r_max; ul_pos; state; end;
     para_type = record N: integer; addr: array[1..N] of *word_type ;
                        exp, new: array[1..N] of word_type;
                        state: {Lock, Unlock, Succ, Fail, Ends, Endf};
                        blocked: array[1..N] of 1..P; end;
                        /* P: #processes; N-word CASN */

return_type HELP(helping, i, pos)
begin
start :
1:   gs := LL(&OP[i].state);
2:   if (ver ≠ Version[i]) then return (Fail, nil);
3:   if (gs.state = Unlock) then
4:     Unlocking(i, gs.ul_pos);
5:     cr = CheckingR(i, OP[i].blocked, gs.ul_pos, gs);
6:     if (cr = Succ) then goto start;
7:     else SC(&OP[i].state.state, Lock);
8:     if (helping = i) then goto start;
9:     else FAA(&OP[i].blocked[pos], −1); return (Succ, nil);
     else if (state = Lock) then
10:    result := Locking(helping, i, pos);
11:    if (result.kind = Succ) then SC(&OP[i].state, (0, 0, 0, Succ));
12:    else if (result = Fail) then SC(&OP[i].state, (0, 0, 0, Fail));
       else if (result.kind = Circle) then
13:      FAA(&OP[i].blocked[pos], −1); return result;
       goto start;
     ...
end.

UNLOCKING(i, ul_pos)
begin
     for j := OP[i].N downto ul_pos do
       e.addr = OP[i].addr[j]; e.exp = OP[i].exp[j];
     again :
       x := LL(e.addr);
       if (not VL(&OP[i].state)) then return;
       if (x = (e.exp, nil)) or (x = (e.exp, i)) then
         if (not SC(e.addr, (e.exp, nil)) then goto again;
     return;
end.
```

Figure 3.11: Procedures Help and Unlocking in our second reactive multi-word compare-and-swap algorithm.

```
boolean CHECKINGR(owner, blocked, kept, gs)
begin
      if (kept = 0) or (blocked = {0..0})) then return Fail;
1:    if ( not VL(&OP[owner].state)) then return Fail;
      for j := 1 to kept do nb := nb + blocked[j];
1':   r := nb/kept; /*r is the current contention*/
      if (r < m * C) then return Fail; /* m = 1/(N-1) */
2:    if (kept ≠ gs.ul_pos) then /*At the beginning of transaction*/
        d = kept * 1/C * (r-m*C)/(r-m); ul_pos := kept - d + 1;
3:      SC(&OP[owner].state, (kept, r, ul_pos, Unlock));
        return Succ;
4:    else if (r > gs.r_max) then /*r is the highest contention so far*/
        d = gs.init * 1/C * (r-gs.r_max)/(r-m); ul_pos := kept - d + 1;
5:      SC(&OP[owner].state, (gs.init, r, ul_pos, Unlock));
        return Succ;
      return Fail;
end.

value_type READ(x)
begin
start :
      y := LL(x);
      while (y.owner ≠ nil) do
            Find index k: OP[y.owner].addr[k] = x;
            ver = Version[y.owner];
            if ( not VL(x)) then goto start;
            Help(self, y.owner, k, ver); y := LL(x);
      return (y.value);
end.
```

Figure 3.12: Procedures CheckingR in our second reactive multi-word compare-and-swap algorithm and the procedure for Read operation.

If $CASN_i$ satisfies the unlock-condition even after a process has just helped it unlock its words, the same process will continue helping the $CASN_i$ (line 5 and 6 in $Help$). Otherwise, if the process is not process p_i, the process initiating $CASN_i$, it will return to help the CASN that was blocked by $CASN_i$ before (line 9 in $Help$). The changed procedures compared with the first implementation are $Help$, $Unlocking$ and $CheckingR$, which are described in Figure 3.11 and Figure 3.12.

3.5 Correctness Proof

In this section, we prove the correctness of our methods. Figure 3.13 briefly describes the shared variables used by our methods and the procedures reading or directly updating them.

	Mem	OP[i].state	OP[i].blocked
Help(helping, i, pos, ver)		LL, *SC*	*FAA*
Unlocking(i, ul_point)	LL, *SC*	VL	
Releasing(i)	LL, *SC*	VL	
Updating(i)	LL, *SC*	VL	
Locking(helping, i, pos)	LL, *SC*	VL	*FAA*
CheckingR(owner, blocked, kept, gs)		VL, *SC*	
Read(x)	LL		

Figure 3.13: Shared variables with procedures reading or directly updating them

The array OP consists of P elements, each of which is updated by only one process, for example $OP[i]$ is only updated by process p_i. Without loss of generality, we only consider an element of array OP, $OP[i]$, on which many concurrent helping processes get necessary information to help a CASN, $CASN_i^j$. The symbol $CASN_i^j$ denotes that this CASN uses the variable $OP[i]$ and that it is the j^{th} time the variable is re-used, i.e. $j = Version[i]$. The value of $OP[i]$ read by process p_k is *correct* if it is the data of the CASN that blocks the CASN helped by p_k. For example, p_k helps $CASN_{i1}^{j1}$ and realizes the CASN is blocked by $CASN_i^j$. Thus, p_k decides to help $CASN_i^j$. But if the value p_k read from $OP[i]$ is the value of the next CASN, $CASN_i^{j+1}$, i.e. $CASN_i^j$ has completed and $OP[i]$ is re-used for another new CASN, the value that p_k read from $OP[i]$, is not correct for p_k.

Lemma 3.5.1. *Every helping process reads correct values of variable $OP[i]$.*

Proof. In our pseudo-code described in Figure 3.7, Figure 3.8, Figure 3.9, Figure 3.11 and Figure 3.12, the value of $OP[i]$ is read before the process checks $VL(\&OP[i].state)$ (line 1 in $Unlocking$, $Releasing$, $Updating$ and $Locking$). If $OP[i]$ is re-used for $CASN_i^{j+1}$, the value of $OP[i].state$ has certainly changed since p_k read it at line 1 in procedure $Help$ because $OP[i]$ is re-used only if the state of $CASN_i^j$ has changed into $Ends$ or $Endf$. In this case, p_k realizes the change and returns to procedure $Help$ to read the new value of $OP[i].state$ (line 3 in $Unlocking$, $Releasing$, $Locking$ and line 4 in $Updating$). In procedure $Help$, p_k will realize that $OP[i]$ is reused by checking its version (line 2 in $Help$) and return, that is p_k does not use incorrect values to help CASNs. Moreover, when

p_k decides to help $CASN_i^j$ at line 13 in procedure $Locking$, the version of $OP[i]$ passed to the procedure $Help$ is the correct version, that is the version corresponding to $CASN_i^j$, the CASN blocking the current CASN on word $e.addr$. If the version p_k read at line 9 in procedure $Locking$ is incorrect, that is $CASN_i^j$ has completed and $OP[i]$ is re-used for $CASN_i^{j+1}$, p_k will realize this by checking $VL(e.addr)$. Because if $CASN_i^j$ has completed, the $owner$ field of word $e.addr$ will change from i to nil. Therefore, p_k will read the value of word $e.addr$ again and realize that $OP[i].state$ has changed. In this case, p_k will return and not use the incorrect data as argued above.

□

From this point, we can assume that the value of $OP[i]$ used by processes is correct. In our reactive compare-and-swap (RCASN) operation, the linearization point is the point its state changes into $Succ$ if it is successful or the point when the process that changes the state into $Fail$ reads an unexpected value. The linearization point of $Read$ is the point when $y.owner == nil$. It is easy to realize that our specific $Read$ operation[2] is linearizable to RCASN. The $Read$ operation is similar to those in [44] [53].

Now, we prove that the interferences among the procedures do not affect the correctness of our algorithms.

We focus on the changes of $OP[i].state$. We need to examine only four states: $LOCK$, $UNLOCK$, $SUCCESS$ and $FAILURE$. State END is only used to inform whether the CASN has succeeded and it does not play any role in the algorithm. Assume the latest change occurs at time t_0. The processes helping $CASN_i$ are divided into two groups: group two consists of the processes detecting the latest change and the rest are in group one. Because the processes in the first group read a wrong state (it fails to detect the latest change), the states they can read are $LOCK$ or $UNLOCK$, i.e. only the states in phase one.

Lemma 3.5.2. *The processes in group one have no effect on the results made by the processes in group two.*

Proof. To prove the lemma, we consider all cases where a process in group two can be interfered by processes in group one. Let p_l^j denote process p_l in group j. Because the shared variable $OP[i].block$ is only used to estimate the contention level, it does not affect the correctness of CASN returned results. Therefore, we only look at the two other shared variables Mem and $OP[i].state$.

[2]The $Read$ procedure described in figure 3.12 is used in both reactive CASN algorithms

Case 1.1 : Assume p_k^1 interferes with p_l^2 while p_l^2 is executing procedure $Help$ or $CheckingR$. Because these procedures only use the shared variable $OP[i].state$, in order to interfere with p_l^2 p_k^1 must update this variable, i.e. p_k^1 must also execute one of the procedures $Help$ or $CheckingR$. However, because p_k^1 does not detect the change of $OP[i].state$ (it is a member of the first group), the change must happen after p_k^1 read $OP[i].state$ by LL at line 1 in $Help$, and consequently it will fail to update $OP[i].state$ by SC. Therefore, p_k^1 cannot interfere p_l^2 while p_l^2 is executing procedure $Help$ or $CheckingR$, or in other words, p_l^2 cannot be interfered through the shared variable $OP[i].state$.

Case 1.2 : p_l^2 is interfered through a shared variable $Mem[x]$ while executing one of the procedures $Unlocking$, $Releasing$, $Updating$ and $Locking$. Because $OP[i].state$ changed after p_k^1 read it and in the new state of $CASN_i$ p_l^2 must update $Mem[x]$, the state p_k^1 read can only be $Lock$ or $Unlock$. Thus, the value p_k^1 can write to $Mem[x]$ is $(OP[i].exp[y], i)$ or $(OP[i].exp[y], nil)$, where $OP[i].addr[y]$ points to $Mem[x]$. On the other hand, p_k^1 can update $Mem[x]$ only if it is not acquired by another CASN, i.e. $Mem[x] = (OP[i].exp[y], nil)$ or $Mem[x] = (OP[i].exp[y], i)$.

- If p_k^1 wants to update $Mem[x]$ from $(OP[i].exp[y], i)$ to $(OP[i].exp[y], nil)$, the state p_k^1 read is $Unlock$. Because only state $Lock$ is the subsequent state from $Unlock$, the correct state p_l^2 read is $Lock$. Because some processes must help the $CASN_i$ successfully release necessary words, which include $Mem[x]$, before the $CASN_i$ could change from $Unlock$ to $Lock$, p_k^1 fails to execute $SC(\&Mem[x], (OP[i].exp[y], nil))$ (line 4 in $Unlocking$, Figure 3.9) and retries by reading $Mem[x]$ again (line 2 in $Unlocking$). In this case, p_k^1 observes that $OP[i].state$ changed and gives up (line 3 in $Unlocking$).
- If p_k^1 wants to update $Mem[x]$ from $(OP[i].exp[y], nil)$ to $(OP[i].exp[y], i)$, the state p_k^1 read is $Lock$. Because the current value of $Mem[x]$ is $(OP[i].exp[y], nil)$, the current state p_l^2 read is $Unlock$ or $Failure$.
  - If p_k^1 executes $SC(\&Mem[x], (OP[i].exp[y], i))$ (line 5 in $Locking$, Figure 3.8) *before* p_l^2 executes $SC(\&Mem[x], (OP[i].exp[y], nil))$ (line 4 in $Unlocking$/ $Releasing$, Figure 3.9), p_l^2 retries by reading $Mem[x]$ again (line 2 in $Unlocking$/ $Releasing$) and eventually updates $Mem[x]$ successfully.
  - If p_k^1 executes $SC(\&Mem[x], (OP[i].exp[y], i))$ (line 5 in $Locking$) *after* p_l^2 executes $SC(\&Mem[x], (OP[i].exp[y], nil))$ (line 4 in $Unlocking/Releasing$), p_k^1 retries by reading $Mem[x]$ again (line

2 in $Locking$). Then, p_k^1 observes that $OP[i].state$ changed and gives up (line 3 in $Locking$).

Therefore, we can conclude that p_k^1 cannot interfere with p_l^2 through the shared variable $Mem[x]$, which together with case 1.1 results in that the processes in group one cannot interfere with the processes in group two via the shared variables. □

Lemma 3.5.3. *The interferences between processes in group two do not violate linearizability.*

Proof. On the shared variable $OP[i].state$, only the processes executing procedure $Help$ or $CheckingR$ can interfere with one another. In this case, the linearization point is when the processes modify the variable by SC. On the shared variable $Mem[x]$, all processes in group two will execute the same procedure such as $Unlocking$, $Releasing$, $Updating$ and $Locking$, because they read the latest state of $OP[i].state$. Therefore, the procedures are executed as if they are executed by one process without any interference. In conclusion, the interferences among the processes in group two do not cause any unexpected result. □

From Lemma 3.5.2 and Lemma 3.5.3, we can infer the following corollary:

Corollary 3.5.1. *The interferences among the procedures do not affect the correctness of our algorithms.*

Lemma 3.5.4. *A $CASN_i$ blocks another $CASN_j$ at only one position in Mem for all times $CASN_j$ is blocked by $CASN_i$.*

Proof. Assume towards contradiction that $CASN_j$ is blocked by $CASN_i$ at two position x_1 and x_2 on the shared variable Mem, where $x_1 < x_2$. At the time when $CASN_j$ is blocked at x_2, both $CASN_i$ and $CASN_j$ must have acquired x_1 already because the CASN tries to acquire an item on Mem only if all the lower items it needs have been acquired by itself. This is a contradiction because an item can only be acquired by one CASN. □

Lemma 3.5.5. *The algorithms are lock-free.*

Proof. We prove the lemma by contradiction. Assume that no CASN in the system can progress. Because in our algorithms a CASN operation cannot progress only if it is blocked by another CASN on a word, each CASN operation in the systems must be blocked by another CASN on a memory word. Let $CASN_i$ be the CASN that acquired the word w_h with highest address among all the words acquired by all CASNs. Because the N words are acquired in the increasing order of their addresses, $CASN_i$ must be blocked by a $CASN_j$ at a word w_k where $address(w_h) < address(w_k)$. That mean $CASN_j$ acquired a word w_k with the address higher than that of w_h, the word with highest address among all the words acquired by all CASN. This is contradiction. □

The following lemmas prove that our methods satisfy the requirements of online-search and one-way trading algorithms [28].

Lemma 3.5.6. *Whenever the* average contention on acquired words *increases during a transaction, the unlock-condition is checked.*

Proof. According to our algorithm, in procedure $Locking$, every time a process increases $OP[owner].blocked$, it will help $CASN_{owner}$. If the $CASN_{owner}$ is in a transaction, i.e. being blocked by another CASN, for instance $CASN_j$, the process will certainly call $CheckingR$ to check the unlock-condition. Additionally, in our second method the average contention can increase when the CASN releases some of its words and this increase is checked at line 5 in procedure $Help$ in figure 3.11. □

Lemma 3.5.6 has the important consequence that the process always detects the *average contention on the acquired words* of a CASN whenever it increases, so applying the online-search and one-way trading algorithms with the value the process obtains for the average contention is correct according to the algorithm.

Lemma 3.5.7. *Procedure $CheckingR$ in the second algorithm computes unlock-point ul_point correctly.*

Proof. Assume that process p_m executes $CASN_i$ and then realizes that $CASN_i$ is blocked by $CASN_j$ on word $OP[i].addr[x]$ at time t_0 and read $OP[i].blocked$ at time t_1. Between t_0 and t_1 the other processes which are blocked by $CASN_i$ can update $OP[i].blocked$. Because $CheckingR$ only sums on $OP[i].blocked[k]$, where $k = 1, .., x-1$, only processes blocked on words from $OP[i].addr[1]$ to $OP[i].addr[x-1]$ are counted in $CheckingR$. These processes updating $OP[i].blocked$ is completely independent of the time when $CASN_i$ was blocked on word $OP[i].addr[x]$. Therefore, this situation is similar to one where all the updates happen before t_0,

i.e. the value of $OP[i].blocked$ used by $CheckingR$ is the same as one in a sequential execution without any interference between the two events that $CASN_i$ is blocked and that $OP[i].blocked$ is read. Therefore, the unlock-condition is checked correctly. Moreover, if $CASN_i$'s state is changed to Unlock, the words from $OP[i].addr[x]$ to $OP[i].addr[N]$ acquired by $CASN_i$ after time t_0 due to another process's help, will be also released. This is the same as a sequential execution: if $CASN_i$'s state is changed to Unlock at time t_0, no further words can be acquired. □

3.6 Evaluation

We compared our algorithms to the two best previously known alternatives: i) the lock-free algorithm presented in [53] that is the best representative of the *recursive helping* policy (RHP), and ii) the algorithm presented in [81] that is an improved version of the *software transactional memory* [92] (iSTM). In the latter, a dummy function that always returns zero is passed to CASN. Note that the algorithm in [81] is not intrinsically wait-free because it needs an evaluating function from the user to identify whether the CASN will stop and return when the contention occurs. If we pass the above dummy function to the CASN, the algorithm is completely lock-free.

Regarding the multi-word compare-and-swap algorithm in [44], the lock-free memory management scheme in this algorithm is not clearly described. When we tried to implement it, we did not find any way to do so without facing live-lock scenarios or using blocking memory management schemes. Their implementation is expected to be released in the future [43], but was not available during the time we performed our experiments. However, relying on the experimental data of the paper [44], we can conclude that this algorithm performs approximately as fast as iSTM did in our experiments, in the shared memory size range from 256 to 4096 with sixteen threads.

The system used for our experiments was an ccNUMA SGI Origin2000 with thirty two 250MHz MIPS R10000 CPUs with 4MB L2 cache each. The system ran IRIX 6.5 and it was used exclusively. An extra processor was dedicated for monitoring. The Load-Linked (LL), Validate (VL) and Store-Conditional (SC) instructions used in these implementations were implemented from the LL/SC instructions supported by the MIPS hardware according to the implementation shown in Figure 5 of [80], where fields tag and val of $wordtype$ were 32 bits each. The experiments were run in 64-bit mode.

The shared memory Mem is divided into N equal parts, and the i^{th} word in N words needing to be updated atomically is chosen randomly in the i^{th} part of the

shared memory to ensure that words pointed by $OP[i].addr[1]...OP[i].addr[N]$ are in the increasing order of their indices on Mem. Paddings are inserted between every pair of adjacent words in Mem to put them on separate cache lines. The values that will be written to words of Mem are contained in a two-dimensional array $Value[3][N]$. The value of $Mem[i]$ will be updated to $Value[1][i]$, $Value[2][i]$, $Value[3][i]$, $Value[1][i]$, and so on, so that we do not need to use the procedure Read, which also uses the procedure Help, to get the current value of $Mem[i]$. Therefore, the time in which only the CASN operations are executed is measured more accurately. The CPU time is the average of the useful time on each thread, the time only used for CASNs. The useful time is calculated by subtracting the overhead time from the total time. The number of successful CASNs is the sum of the numbers of successful CASNs on each thread. Each thread executing the CASN operations precomputes N vectors of random indices corresponding to N words of each CASN prior to the timing test. In each experiment, all CASN operations concurrently ran on thirty processors for one minute. The time spent on CASN operations was measured.

The contention on the shared memory Mem was controlled by its size. When the size of shared memory was 32, running eight-word compare-and-swap operations caused a high contention environment. When the size of shared memory was 16384, running two-word compare-and-swap operations created a low contention environment because the probability that two CAS2 operations competed for the same words was small. Figure 3.14 shows the total number of CASN and the number of *successful* CASN varying with the shared memory size. We think this kind of chart gives the reader a good view on how each algorithm behaves when the contention level varies by comparing the total number of CASN and the number of *successful* CASN.

3.6.1 Results

The results show that our CASN constructions compared to the previous constructions are significantly faster for almost all cases. The left charts in Figure 3.14 describes the number of CASN operations performed in one second by the different constructions.

In order to analyze the improvements that are because of the reactive behavior, let us first look at the results for the extreme case where there is almost no contention and the reactive part is rarely used: CAS2 and the shared memory size of 16384. In this extreme case, only the efficient design of our algorithms gives the better performance. In the other extreme case, when the contention is high, for instance the case of CAS8 and the shared memory size of 32, the brute force approach of the recursive helping scheme (RHP) is the best strategy to use. The re-

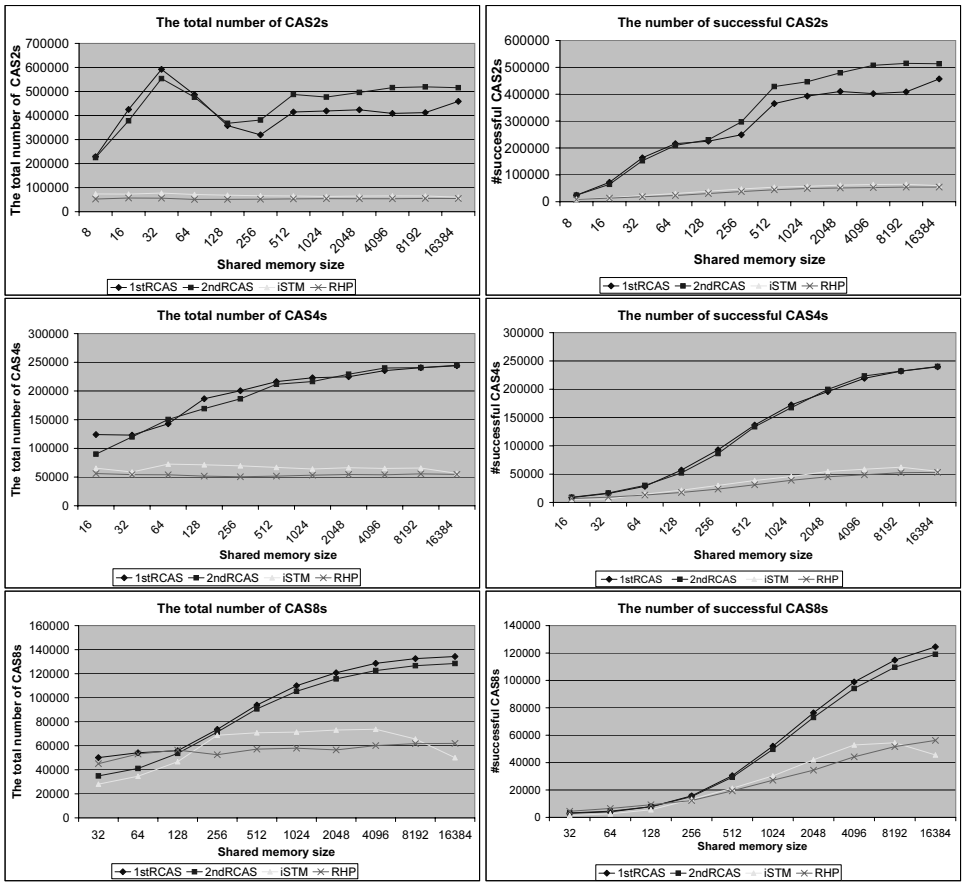

Figure 3.14: The numbers of CAS2s, CAS4s and CAS8s and the number of *successful* CAS2s, CAS4s and CAS8s in one second

cursive scheme works quite well because in high contention the conflicts between different CASN operations can not be really solved locally by each operation and thus the serialized version of the recursive help is the best that we can hope for. Our reactive schemes start helping the performance of our algorithms when the contention coming from conflicting CASN operations is not at its full peak. In these cases, the decision on whether to release the acquired words plays the role in gaining performance. The benefits from the reactive schemes come quite early and drive the performance of our algorithms to reach their best performance rapidly. The left charts in Figure 3.14 shows that the chart of RHP is nearly flat regardless of the contention whereas those of our reactive schemes increase rapidly with the decrease of the contention.

The right charts in Figure 3.14 describes the number of *successful* CASN operations performed in one second by the different constructions. The results are similar in nature with the results described in the previous paragraph. When the contention is not at its full peak, our reactive schemes catch up fast and help the CASN operations to solve their conflicts locally.

Both figures show that our algorithms outperform the best previous alternatives in almost all cases. At the memory size 16384 in the left charts of Figure 3.14:

CAS2 : the first reactive compare-and-swap (1stRCAS) and the second one (2ndRCAS) are about seven times and nine times faster than both RHP and iSTM, respectively.

CAS4 : both RCAS are four times faster than both RHP and iSTM.

CAS8 : both RCAS are two times faster than both RHP and iSTM.

Regarding the number of successful CASN operations, our RCAS algorithms still outperform RHP and iSTM in almost all cases. Similar to the above results, at the memory size of 16384 in the right charts of Figure 3.14, both reactive compare-and-swap operations perform faster than RHP and iSTM from two to nine times.

3.7 Conclusions

Multi-word synchronization constructs are important for multiprocessor systems. Two reactive, lock-free algorithms that implement multi-word compare-and-swap operations are presented in this paper. The key to these algorithms is for every CASN operation to measure in an efficient way the contention level on the words it has acquired, and reactively decide whether and how many words need to be released. Our algorithms are also designed in an efficient way that allows high parallelism —both algorithms are lock-free— and most significantly, guarantees that the new operations spend significantly less time when accessing coordination shared variables usually accessed via expensive hardware operations. The algorithmic mechanism that measures contention and reacts accordingly is efficient and does not cancel the benefits in most cases. Our algorithms also promote the CASN operations that have higher probability of success among the CASNs generating the same contention. Both our algorithms are linearizable. Experiments on thirty processors of an SGI Origin2000 multiprocessor show that both our algorithms react quickly according to the contention conditions and significantly outperform the best-known alternatives in all contention conditions.

In the near future, we plan to look into new reactive schemes that may further improve the performance of reactive multi-word compare-and-swap implementations. The reactive schemes used in this paper are based on competitive online

techniques that provide good behavior against a malicious adversary. In the high performance setting, a weaker adversary model might be more appropriate. Such a model may allow the designs of schemes to exhibit *more active* behavior, which allows faster reaction and better execution time.

Chapter 4

Efficient Multi-Word Locking Using Randomization[1]

Phuong Hoai Ha[2], Philippas Tsigas[2], Mirjam Wattenhofer[3], Roger Wattenhofer[4]

Abstract

In this paper we examine the general multi-word locking problem, where processes are allowed to multi-lock arbitrary registers. Aiming for a highly efficient solution we propose a randomized algorithm which successfully breaks long dependency chains, the crucial factor for slowing down an execution. In the analysis we focus on the 2-word lock problem and show that in this special case an execution of our algorithm takes with high probability at most time $O(\Delta^3 \log n / \log\log n)$*, where* n *is the number of registers and* Δ *the maximal number of processes interested in the same register (the contention). Furthermore, we implement our algorithm for the general multi-word lock problem on an SGI Origin2000 machine, demonstrating that our algorithm is not only of theoretical interest.*

[1]This paper appeared in the Proceedings of the 24th Annual ACM SIGACT-SIGOPS Symposium on Principles Of Distributed Computing (PODC '05), Jul. 2005, pp. 249-257, ACM Press.

[2]Department of Computer Science and Engineering, Chalmers University of Technology, S-412 96 Gothenburg, Sweden. Email:{*phuong, tsigas*}*@cs.chalmers.se*

[3]Department of Computer Science, ETH Zurich, 8092 Zurich, Switzerland. Email:*mirjam.wattenhofer@inf.ethz.ch*

[4]Computer Engineering and Networks Laboratory, ETH Zurich, 8092 Zurich, Switzerland. Email:*wattenhofer@tik.ee.ethz.ch*

4.1 Introduction

Edsger Dijkstra's dining philosophers problem is widely recognized as a prototypical resource allocation instance. We are given n philosophers, sitting at a round table. Each philosopher is an asynchronous process who cycles through the three states thinking, hungry, and eating. Between each neighboring pair of philosophers, there is a fork. When becoming hungry, a philosopher tries to grab her left and right fork. After having acquired both forks, the philosopher eats. When finished eating, the philosopher returns her forks and goes back to thinking mode.

We can represent the classic dining philosophers problem in a shared memory multi-processor system by having n shared registers (the forks) and n processes (the philosophers). The two registers ("forks") which are of interest to process p_i (with $i = 1, 2, \ldots, n$) are registers i and $i + 1$ (with the notable exception that the "right fork" of processor p_n is register 1, and not $n + 1$, to achieve the desired ring topology). In shared memory dining philosophers, each process repeatedly and asynchronously tries to lock its two registers (hungry), then performs some atomic operation on these two registers, such as multi-word compare-and-swap (eating), and then continues with other operations (thinking).

The dining philosophers problem perfectly illustrates typical multi-process synchronization difficulties. If all philosophers become hungry at the same time, and pick up their left fork simultaneously, we have a *deadlock*, since no philosopher can grab her right fork as well. Similarly, if a process crashes (or behaves awfully slow) after locking its registers, the two neighbor processes cannot make progress; as a remedy the research community has proposed *non-blocking* protocols, such as recursive helping schemes, or transactional memory.

In this paper we focus on a third fundamental multi-process synchronization issue, *efficiency*. In dining philosophers, "even" philosophers (processes with even process id) do not have a conflict of interest among themselves. An efficient implementation striving for maximum concurrency would therefore always let even and odd processes eat in turns, thus maximizing the available resources.

In this paper we examine the general multi-word lock problem, where processes are allowed to multi-lock arbitrary registers. The remainder of the paper is organized as follows: In Section 2 we set our paper into context of prior art. The model is then formally introduced in Section 3. In Section 4 we present our algorithm and analyze it; in particular we show that a process has to wait at most $O(\Delta^3 \log n / \log \log n)$ time until it can eat, where n is the number of registers and Δ the maximal number of processes interested in the same register (the contention). In Section 5 we present extensive results from our implementation on an SGI Origin2000 machine, proving that our idea is not only of theoretical interest. Finally, in Section 4.6 we conclude the paper.

4.2 Related Work

Since processes without a conflict can proceed concurrently, it seems promising to first compute a minimum coloring of the conflict graph. Yet solving the multi-lock problem using coloring remained theory.

In a generalized variant of dining philosophers, a process shares d forks and can only eat if it has obtained all d forks. For this generalized problem [72] gave a solution with waiting chains of length $O(c)$, assuming that an oracle has colored the conflict graph[1] with c colors. The waiting chain length was reduced to $O(\log c)$ in [97]. Assuming that a vertex coloring with $d + 1$ colors is known, in [20] this length was further reduced to 3. For a simplified version of dining philosophers [83] managed to have waiting chains of length at most 4 in constant time.

Unfortunately, even in a powerful message passing model, coloring is a tough problem. It was proven in [69] that such colorings cannot be found in constant time. In fact, even simpler problems (such as independent sets) have logarithmic lower bounds [69]. More severely, the conflict graph is not available straightforwardly. To compute the conflict graph, processes need some form of synchronization. We believe that this synchronization is as hard to achieve as the original multi-lock problem.

In this paper we present an efficient but blocking algorithm for the general multi-lock problem, consequently without making a detour through coloring. A blocking algorithm for the multi-lock problem can be turned into a lock-free algorithm if it is combined with a *helping technique*. The basic idea of the so called cooperative technique [14], a form of helping technique, was improved and is still developing in a series of nifty research papers [5, 40, 44, 53, 81, 92].

For readability we do not integrate our algorithm with a helping scheme; however, it can be added to our algorithm: Each process, before locking registers, somewhere notes what it wanted to do with that register. In case the process crashes while holding the lock, others can help it finish by following its steps. The implementation of our algorithm used in Section 4.5 of this paper includes a helping scheme, rendering our implementation lock-free.

In distributed graph algorithms (message passing), randomization techniques are widely used. With a few exceptions [12], the shared memory community does generally not apply randomization, presumably because its alleged overhead. Our experiments show that the overhead due to randomization is less than 1% of the total execution time.

[1] Here the conflict graph is a graph where each node represents a process and each edge represents a resource which is shared by the two endpoint processes. Our conflict graph is different, see also Section 4.3.

4.3 Problem and Model

In this section we recapitulate the problem we consider and formally define the model used in the next sections.

We study the *multi-lock problem*, which is a generalization of dining philosophers. In the multi-lock problem each participating process needs to lock multiple registers in order to do some operation on the locked registers, like an N-word compare-and-swap (CASN).

Typically, in a multi-lock implementation a process tries to lock all its registers one by one. To avoid deadlocks, the registers are totally ordered, conventionally by their identifiers (id). When executing a k-lock, a process p locks its registers $r_1, \ldots, r_k$ according to their total order.

As discussed in Section 4.2, there exist several schemes which can be employed once a process is blocked by other processes from locking its registers. In the analysis we assume that processes simply wait (spin-lock) until the block is resolved, yet for the implementation (Section 4.5) we include a helping scheme.

We consider m *asynchronous* processes which can access n shared registers. In the analysis we concentrate on 2-locks. Each process only executes a single 2-lock and then goes to sleep.

The dependencies between the processes are modeled by a directed acyclic *conflict graph* $G = (V, E)$. In G each node represents a register and each edge represents a process. In the following, we will use the terms node/register and edge/process interchangeably. There is a directed edge p from node r_1 to node r_2 iff process p tries to lock register r_1 first and after being successful tries to lock register r_2, meaning that r_1 comes before r_2 in the total order of registers. Since all directed edges point from nodes with lower id to nodes with higher id the resulting graph G is acyclic.

Following the conventions for asynchronous processes, in the analysis we assume that each atomic operation, like reading, writing, or locking a register, incurs a delay of *at most* one time unit. An operation on multiple locked registers (e.g. *CAS2*) incurs a delay. For convenience let c be the longest time which elapses from the moment a process has locked its last register until it releases the lock on all its registers.

In the remainder of this section, to illustrate our model, we quickly analyze a classical implementation of dining philosophers. We show that it is a factor $\Omega(n)$ less efficient than an optimal implementation. The classical implementation proceeds as follows: the processes try to lock their registers one by one, each starting with the register with smaller identifier. Consider the following execution: First, each process p_i, $i < n$, locks register r_i, whereas p_n fails to lock r_1 (due to p_1). Then, each process tries to lock register r_{i+1}, yet only process p_{n-1} succeeds. The

second register of all other processes is locked by another process. Thus, process p_i has to wait until process p_{i+1} releases its lock on r_{i+1}. By induction, after having waited $\Theta(cn)$ time units, p_1 releases its lock on r_1 and p_n may lock both its registers. Thus, the execution time is $\Omega(cn)$. An optimal implementation needs only $O(c)$ time and hence the classical algorithm is a factor $\Omega(n)$ less efficient than an optimal algorithm.

4.4 Randomized Registers

In this section we present a more efficient algorithm for multi-lock and analyze it according to two standard criteria for the special case of 2-lock.

4.4.1 The Algorithm

Alerted by the poor execution time of the classical algorithm for dining philosophers due to its long dependency chain, we aim at breaking dependency chains. A promising yet simple (allowing for an efficient implementation) approach is randomization. Specifically, we suggest to randomly permute the order of the registers. Let Π be a permutation on the registers, chosen uniformly at random. The permutation represents the new total ordering of the registers. For details on how the randomization can be implemented we refer to Section 4.5.

In short we henceforth write $p = (r_i, r_j)$ meaning that process p wants to acquire register r_i and r_j and that $\Pi(\text{id}(r_i)) < \Pi(\text{id}(r_j))$. Thus, r_i is p's *first* register and r_j is p's *second* register.

In the next sections we analyze the efficiency of the suggested 2-lock algorithm which uses a randomized total ordering of registers. As in [97] we evaluate two related properties: the maximum length of a waiting chain and the longest time a process needs until it successfully performs a 2-lock. Towards this goal, we first prove some basic properties of the conflict graph[2]. Thereafter, we analyze the length of waiting chains and finally show that with high probability after $O(c\Delta^3 \log n / \log\log n)$ time the execution is finished.

4.4.2 Length of Directed Paths

Henceforth, we denote by G the conflict graph as obtained by the random permutation of the registers. Let the maximum degree in G be Δ. In this section we analyze the length of a directed path in G. The following facts are used for the analysis, the proofs of which can be found in standard mathematical textbooks.

[2]Note, that the conflict graph is only needed for analysis purposes. The processes do not know the conflict graph.

Fact 4.4.1 (Stirling).

$$k! \geq 2\frac{\sqrt{k}k^k}{e^k}.$$

Fact 4.4.2 (Markov).

$$\mathbb{P}(X \geq t) \leq \mathbb{E}[X]/t.$$

Throughout the paper $\log n$ denotes the logarithm with base two.

To estimate the length of a directed path in G, we first upper bound the number of distinct *undirected* paths of length k in G: To obtain an undirected path of length k one can choose one out of n nodes in G as start node. In any node there are at most Δ neighbor nodes to continue the path. Therefore:

Observation 4.4.3. *There are at most $n \cdot \Delta^k$ distinct undirected paths of length k in G.*

As a next step we give the probability that a given path of length k in G is directed.

Observation 4.4.4. *The probability that a given path of length k in G is directed is $\frac{2}{(k+1)!}$.*

Proof. In a path of length k there are $k+1$ nodes $u_1, \ldots, u_{k+1}$. For the path to be directed, it must hold that either $\Pi(\text{id}(u_1)) < \Pi(\text{id}(u_2)) < \ldots < \Pi(\text{id}(u_{k+1}))$ or $\Pi(\text{id}(u_1)) > \Pi(\text{id}(u_2)) > \ldots > \Pi(\text{id}(u_{k+1}))$. Hence, there are exactly two good out of $(k+1)!$ possible choices. □

Thus, the probability that a path is directed decreases exponentially with increasing path-length. Combining both Observation 4.4.3 and Observation 4.4.4 gives an upper bound on the number of directed paths of length k.

Lemma 4.4.1. *Let C be the number of directed paths of length k. Then, $\mathbb{E}[C] < \frac{1}{n^\Delta}$, for $k \geq 3\Delta\frac{\log n}{\log\log n}$.*

Proof. Let p_i denote a path of length k and let X_{p_i} be defined as follows

$$X_{p_i} = \begin{cases} 1, & \text{if } p_i \text{ directed} \\ 0, & \text{otherwise.} \end{cases}$$

Then by linearity of expectation,

$$\begin{aligned} \mathbb{E}[C] &= \mathbb{E}[\sum_{\forall p_i} X_{p_i}] \\ &= \sum_{\forall p_i} \mathbb{E}[X_{p_i}] \\ &\leq n\Delta^k \frac{2}{(k+1)!}, \end{aligned}$$

by Observation 4.4.3 and Observation 4.4.4. Applying Stirling's formula (Fact 4.4.1) and substituting $3\Delta \log n / \log\log n$ for k yields the following inequalities

$$\begin{aligned}
\mathbb{E}[C] &< n\Delta^k \frac{e^k}{\sqrt{k}k^k} \\
&\leq n\Delta^k \frac{e^k}{(3\Delta \log n/\log\log n)^k} \\
&< n\frac{1}{(\log n/\log\log n)^k} \\
&= n\frac{1}{2^{3\Delta \log n/\log\log n(\log\log n - \log\log\log n)}} \\
&= n\frac{1}{n^{3\Delta(1-\log\log\log n/\log\log n)}} \\
&\leq \frac{1}{n^{\Delta}}.
\end{aligned}$$

□

Finally, we bound the probability that there exists a directed path in G by applying Markov's inequality.

Corollary 4.4.5. *With probability at most $1/n^{\Delta}$ there exists a directed path of length at least $3\Delta \log n/\log\log n$.*

Proof. By Fact 4.4.2 and Lemma 4.4.1

$$\mathbb{P}(C \geq 1) \leq \mathbb{E}[\mathbb{C}] \leq 1/n^{\Delta}.$$

□

4.4.3 Length of Waiting Chains

Following the notation of [97] we define a *waiting chain* as a series of processes such that each process in the chain is waiting for some action by the next process in the chain.

A process p is delayed by another process q if q can, by slowing down or stopping, cause p to have a longer total waiting time than if q stayed in its remainder section. The maximum length of a waiting chain is the maximum distance between two processes such that one process can delay the other.

We want to intuitively depict the concept of delaying with the example of Figure 4.1. Process p can be delayed by all processes in the figure: E.g. process q_2 can delay p if q_1 locks r_1 before p and q_2 locks r_6 before q_1. Thus, q_1 has to wait

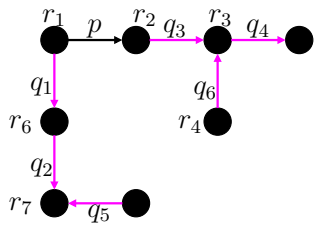

Figure 4.1: Delation of p.

for q_2 until it can acquire (and release) both its locks and consequently p (which waits for q_1 to release the lock on r_1) also has to wait for q_2. Process q_6 can also delay p: q_6 locks r_3, q_3 locks r_2 and p locks r_1. Thus, p is waiting for q_3 to release r_2 which itself is waiting for q_6 to release r_3. On the other hand, process p could not be delayed by a (not shown) process $q_7 = (r_4, x)$, where x is some register not depicted in Figure 4.1. This is because process q_7 may acquire register r_4 before q_6 and thus q_6 has to wait until q_7 releases this register again before it can proceed. Yet, this does in no way delay p since it does not impose a longer waiting time on p. In general:

Lemma 4.4.2. *Process $q = (R_1, R_2)$ can delay process $p = (r_1, r_2)$ if and only if R_2 lies on a directed path starting in either r_1 or r_2.*

Proof. By definition, if process q delays process p, the waiting time of p must be longer if q slows down or stops than if q stayed in its remainder section. If processes q's second register R_2 lies on a directed path starting in r_i, $i \in \{1, 2\}$, then there exists a directed path of processes $q_1 = (r_i, r_j), \ldots, q_k = (r_l, R_2)$, with possibly $q_k = q$. In the case that each process in this path locked its first register - and given that $q_k \neq q$ q also locked its second register - none of processes $q_1, \ldots, q_k$ makes any progress as long as q does not make any progress. Thus, process p will not be able to lock its register r_i and hence its waiting time is longer than if q stayed in its remainder section, showing that q delays p.

In order to show the other direction of the lemma we let Q be the set of processes which do not lie with their second register on a directed path from $r_i, i \in [1, 2]$. Between any arbitrary process q in Q and any directed path $\mathcal{P}_i$ from r_i there is at least one process $\overline{q}$ which breaks this directed path, that is $\overline{q}$'s second register lies on $\mathcal{P}_i$ whereas its first register does not. By slowing down or stopping its execution q may hinder $\overline{q}$ in acquiring its first register, yet it does not hinder $\overline{q}$ in acquiring its second register, otherwise one of q's registers would also lie on a directed path from r_i, a contradiction to the assumption that q in Q. Thus, by slowing down or stopping q either does not affect $\overline{q}$ or $\overline{q}$ cannot participate in the

execution at all as long as q does not make any progress. Hence, process q cannot delay p via $\bar{q}$ and consequently it cannot delay p via any process on $\mathcal{P}_i$. Furthermore, q is not incident to p and thus it cannot delay p directly. Since q cannot delay p indirectly via a process on a directed path nor directly, p is not affected if q slows down or stops. This concludes the proof □

Corollary 4.4.6. *The maximum length of a waiting chain in the randomized registers algorithm is with probability at least* $1-1/n^{\Delta}$ *at most* $3\Delta \log n/\log\log n+1$.

Proof. The maximal number of edges between a process p and a process q which delays p is at most the length of the longest directed path plus one, since by Lemma 4.4.2 a process which delays p must be incident with its second register to a directed path. Hence, we can directly apply Corollary 4.4.5. □

4.4.4 Execution Time

Though the length of a waiting chain is an indicator of the efficiency of an algorithm, it is only a lower bound for the execution time. For the execution time we must bound two values: First, we need to bound the time until a process is able to lock its first register, then we need to bound the time until it can lock its second register. Towards this goal we introduce some helpful definitions.

The execution starts at time zero. A process $p = (r_1, r_2)$ locks its first register at time $t_1(p)$ and its second register at time $t_2(p)$. Using the definition of Section 4.3 process p releases both its locks at time $t_3(p) \leq t_2(p) + c$. (See also Figure 4.2.)

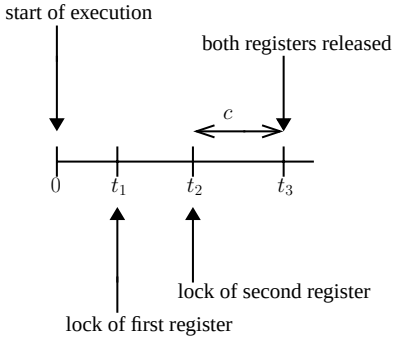

Figure 4.2: t_1, t_2 and t_3 for a process.

Definition 4.4.7 (Delay Graph). *Let p be a process with $p = (r_1, r_2)$. Then p's* delay graph, *denoted by $D(p)$ contains all processes with $q = (R_1, R_2)$ where*

R_1 lies on a directed path starting in r_1. The depth of process p depth(p) is then defined as the length of the longest directed path (number of processes in the path) in $D(p)$.

In the example of Figure 4.3 3 processes q_1, q_2, q_3, q_4 are in p's delay graph, whereas process q_5, q_6 are not, since there is no a directed path from r_1 to q_6's first register r_4, respectively q_5's first register r_8. Intuitively, processes which are incident to a directed path from p merely by their second register, do not delay p much, since those processes release their lock on the crucial register quickly after acquiring it. Note that the depth of a process is at least one since at least the process itself lies in its delay graph.

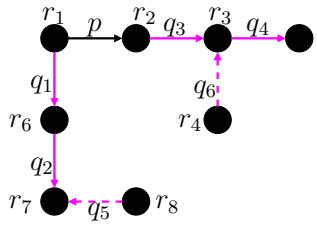

Figure 4.3: Process p, q_1, q_2, q_3, q_4 are in p's delay graph. The depth of p is 3, q_1's depth is also 3.

We now bound the maximal depth of any process by directly applying Corollary 4.4.5:

Corollary 4.4.8. *The maximum depth k^* of any process is at most $3\Delta \log n / \log\log n$ with probability at least $1 - 1/n^{\Delta}$.*

The next lemma reveals a key property of the delay graph.

Lemma 4.4.3. *Let $D(p)$ be processes $p = (r_1, r_2)$ delay graph and let $q = (R_1, R_2)$ be a process in $D(p)$. Then,*

$$\textit{depth}(q) \leq \textit{depth}(p).$$

Furthermore, if $R_1 \neq r_1$ then

$$\textit{depth}(q) < \textit{depth}(p).$$

Proof. Assume without loss of generality that $\mathcal{P}_{R_j u} = (R_j, u_1, \ldots, u_k, u)$, $j \in \{1, 2\}$, is a longest directed path in q's delay graph. Then, depth(q)$= |\mathcal{P}_{R_j u}| = |\{R_j, u_1, \ldots, u_k, u\}| - 1$. Since $q \in D(p)$ there is a directed path $\mathcal{P}_{r_1 R_1} =$

$(r_1, v_1, \ldots, v_l, R_1)$ from r_1 to R_1, where $|\mathcal{P}_{r_1R_1}| = |\{r_1, v_1, \ldots, v_l, R_1\}| - 1$ is the length of this path. Consequently, there exists a directed path $\mathcal{P}_{r_1u} = (r_1, \ldots, R_1, R_j, \ldots, u)$ from r_1 to u. Thus,

$$\begin{aligned}\text{depth}(p) &\geq |\mathcal{P}_{r_1u}| \\ &= |\{r_1, \ldots, v_l, R_1, R_j, \ldots, u\}| - 1 \\ &\geq |\{R_j, \ldots, u\}| - 1 \\ &= \text{depth}(q).\end{aligned}$$

If furthermore, $r_1 \neq R_1$ then

$$\begin{aligned}\text{depth}(p) &= |\{r_1, \ldots, v_l, R_1, R_j, \ldots, u\}| - 1 \\ &\geq |\{r_1, \ldots, v_l\}| + |\{R_1, R_j \ldots, u\}| - 1 \\ &\geq 1 + |\mathcal{P}_{R_ju}| \\ &> \text{depth}(q).\end{aligned}$$

□

The following corollary shows that along a directed path the depth of the processes is strictly decreasing.

Corollary 4.4.9. *Let $\mathcal{P} = (r_1, r_2, \ldots, r_{k+1})$ be a directed path and let $p_i = (r_i, r_{i+1}), 1 \leq i \leq k$, be the processes on this path. Then, depth$(p_i) >$ depth$(p_{i+1}), 1 \leq i \leq k-1$.*

Proof. By the definition of a delay graph, a process $q = (R_1, R_2)$ lies in the delay graph $D(p)$ of process $p = (r_1, r_2)$ iff there is a directed path between r_1 and R_1. Thus, process p_{i+1} lies in the delay graph of process p_i since by the assumption r_i and r_{i+1} lie on a directed path. We hence may apply Lemma 4.4.3 which states that the depth of a process q which lies in the delay graph $D(p)$ of process p is strictly smaller than p's depth if p's first register is not equal to q's first register. Since in our case the first register of p_{i+1} is r_{i+1} and the first register of p_i is r_i, this condition holds and thus the depth of p_{i+1} is strictly smaller than p_i's depth. □

Corollary 4.4.10. *There is a process with depth one in any conflict graph G.*

Proof. Let $p_1 = (r_1, r_2)$ be a process in G with depth k. Then, there exists a directed path $\mathcal{P} = (r_1, r_2, , \ldots, r_{k+1})$ from r_1 to some node r_{k+1} of length (number of processes in $\mathcal{P}$) k. By Corollary 4.4.9 the depth of the processes $p_i = (r_i, r_{i+1}), 1 \leq i \leq k$, in this path is strictly decreasing. Thus, depth$(p_{i+1}) \leq$ depth$(p_i) - 1, 1 \leq i \leq k-1$, and since depth$(p_1) = k$, we have depth$(p_k) \leq 1$. The depth of any process is at least one and consequently depth$(p_k) = 1$. □

In the next lemma we upper bound $t_3(p)$ for a process p with depth one.

Lemma 4.4.4. *For a process $p = (r_1, r_2)$ with depth(p)=1 we have $t_3(p) \leq 4c\Delta^2$.*

Proof. A process p has depth one iff the following two conditions hold: A process q_j incident to p's *first* register r_1 is either incoming in r_1 (type a), that is $q_j = (x, r1)$, x an arbitrary register, or $q_j = (r_1, x)$ and all processes incident to q_j's second register x are incoming in x (type b). A process q_i incident to p's *second* register r_2 is incoming in r_2, that is $q_i = (x, r_2)$, x an arbitrary register. (See also Figure 4.4.)

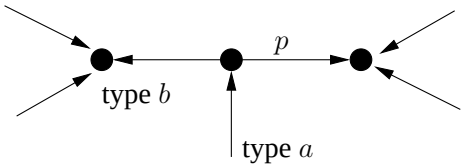

Figure 4.4: Depth(p)=1.

We first concentrate on type a processes: Processes of type a release their lock on r_1 at most c time units after acquiring it. The next process acquires the lock on r_1 at most one time unit later. Thus each process of type a adjacent to r_1 delays p for at most $c + 1 \leq 2c$ time units.

A process $q_j = (r_1, x)$ of type b must wait at the utmost for all processes incident to its second register x until it can acquire the lock on x and thereafter release r_1. A process incident to x releases its lock on x at most c time units after acquiring it and the next process acquires it at most one time unit later. Thus, each process incident to x delays q_j for at most $c + 1 \leq 2c$ time units. Besides q_j there are at most $\Delta - 1$ processes incident to x. We thus immediately get that q_j acquires its lock on x after at most $2c(\Delta - 1) + 1$ time units and releases its locks after at most c more time units. Thus each process of type b adjacent to r_1 delays p for at most $2c(\Delta - 1) + 1 + c \leq 2c\Delta$ time units.

Besides p there are at most $\Delta - 1$ processes incident to r_1, each of which releases its lock on r_1 at most $2c\Delta$ time units after acquiring it. Therefore, we immediately get

$$t_1(p) = 2c\Delta(\Delta - 1) + 1$$

The time until p can lock r_2 is by the same argument as the argument for type a processes at most $2c(\Delta - 1) + 1$ and hence

$$\begin{aligned} t_3(p) &\leq t_2(p) + c \\ &\leq 2c\Delta(\Delta - 1) + 1 + 2c(\Delta - 1) + 1 + c \\ &\leq 4c\Delta^2. \end{aligned}$$

□

Lemma 4.4.5. *For a process p with depth(p)=k we have*

$$t_3(p) \leq 4c\Delta^2 k.$$

Proof. We prove the lemma by induction on the depth of a process $p = (r_1, r_2)$. By Corollary 4.4.10 there always exists a process of depth one in the conflict graph G and thus we may base the induction in this case.

Base Case: In case that depth(p)= 1 $t_3(p) \leq 4c\Delta^2$ by Lemma 4.4.4.

Induction: We henceforth assume that for a process q with depth(q)$\leq (k-1)$ it holds that $t_3(q) \leq 4c\Delta^2(k-1)$ and consider process p with depth k. By Lemma 4.4.3 all processes in p's dependency graph $D(p)$ which do not have r_1 as their first register have depth less than k and thus –by the induction hypothesis– finished their operations at time $4c\Delta^2(k-1)$ at latest. Thus, the only processes in $D(p)$ which are still active are those which have r_1 as their first register and consequently at time $4c\Delta^2(k-1)$ p's depth is at most one. We then apply Lemma 4.4.4 and get

$$t_3(p) \leq 4c\Delta^2(k-1) + 4c\Delta^2 = 4c\Delta^2 k.$$

□

Theorem 4.4.11. *A process p finishes its operations after time $O(c\Delta^3 \log n/\log\log n)$ with probability at least $1 - 1/n^{\Delta}$.*

Proof. By Corollary 4.4.8 the depth of any process is at most $3\Delta \log n/\log\log n$ with probability at least $1 - 1/n^{\Delta}$. Thus, using Lemma 4.4.5,

$$\begin{aligned} t_3(p) &\leq 4c\Delta^2 \cdot 3\Delta \log n/\log\log n \\ &\in O(c\Delta^3 \log n/\log\log n). \end{aligned}$$

□

In a model where an operation takes exactly time c, clearly an optimal algorithm needs at least the time c to finish. Therefore:

Corollary 4.4.12. *With probability at least $1 - 1/n^{\Delta}$ the randomized registers algorithm is $O(\Delta^2 \log n/\log\log n)$ competitive.*

4.5 Evaluation

We have proposed a *multi-lock* algorithm, where the operation performed after all registers are locked can be defined arbitrarily by the programmer. To evaluate the algorithm, we chose the operation specifically to be a single-word compare-and-swap on each register. With this choice, our algorithm became a multi-word compare-and-swap algorithm.

The multi-word compare-and-swap operations (CASN) extend the single-word compare-and-swap operations from one word to many. A single-word compare-and-swap operation (CAS) takes as input three parameters: the address, an *old* value and a *new* value of a word, and atomically updates the contents of the word if its current value is the same as the *old* value (cf. Figure 4.5). Similarly, an N-word compare-and-swap operation takes the addresses, *old* values and *new* values of N words, and if the current contents of these N words all are the same as the respective *old* values, the CASN will write the new values to the respective words atomically. Otherwise, we say that the CAS/CASN fails, leaving the variable values unchanged.

```
CAS(x, old, new)
atomically {
    if (x = old) {x ← new; return (true)};
    else return (false);
    }
```

Figure 4.5: The single-word compare-and-swap primitive

The multi-word compare-and-swap operations are powerful constructs, which make the design of concurrent data structures more effective and easier. As expected, they attracted the attention of many researchers, consequently many CASN implementations appear in the literature [5, 40, 44, 53, 81, 92]. One approach suggested to construct CASN operations is *cooperative technique*, which allows processes to concurrently access the shared data as long as they write down what they are doing. Before changing a portion of the shared data that was locked by another process p_j, a process p_i must help p_j complete its task first. The technique was first theoretically suggested by Barnes [14] and then was transformed into a more applicable one by Israeli et al. [53], which was used to implement a lock-free multi-word compare-and-swap operation. This implementation was later improved by Harris et al. [44] to reduce the per-word space overhead. A wait-free multi-word compare-and-swap was developed by Anderson based on this technique [5]. However, this *cooperative technique* uses a recursive helping policy, where a process has to help many other processes before completing its own task. The helping chains, where

process p_i helps p_{i+1}, may be very long. All processes related to a chain may do the same task, the task of the last process in the chain, which reduces parallelism and creates high collision levels on the shared data needed by the common task.

In order to evaluate the performance of our algorithm (randomized CAS, in short RaCASN) and also check its feasibility in a real setting we implemented it and ran it on a ccNUMA SGI Origin2000 multiprocessor that was equipped with 30 CPUs. As discussed in Section 4.2 we equipped our randomized registers algorithm with a helping policy [53]. In order to see in practice the performance benefits of the randomization we also implemented the deterministic recursive helping policy (DeCASN) presented in [53]. The implementation of RaCASN was similar to that of DeCASN except that the order of registers/words[3] chosen to be locked was random in RaCASN. In other words, both algorithms are lock-free. For the tests we used a micro-benchmark and a small application. The micro-benchmark was designed to generate an execution environment with high contention on the shared registers. The application was a parallel-prefix application with continuous input feed and space constraints.

4.5.1 The micro-benchmark

The micro-benchmark aims at generating an environment with high contention on shared registers. In the micro-benchmark, a set of $N + k$ virtual registers v_i, $1 \leq i \leq N+k$, are mapped on k system registers $r_1, r_2, \cdots, r_k$, where N is the number of registers to be updated atomically by the CASN operations. The mapping used is

$$v_i = \begin{cases} r_i & \text{if } 1 \leq i \leq k \\ r_{i-k} & \text{if } k+1 \leq i \leq k+N. \end{cases}$$

The virtual registers are accessed by k N-word compare-and-swap operations $CASN_1$, $CASN_2$, $\cdots$, $CASN_k$. During the execution of this benchmark, each $CASN_i$ operation tries to update virtual registers $v_i, v_{i+1}, \cdots, v_{i+N-1}$ atomically. Note that two consecutive CASNs $CASN_i$ and $CASN_{i+1}$ have $N-1$ system registers in common and thus the micro-benchmark can generate helping chains of length up to k, where a helping chain is a chain of CASN operations that a thread has to help before completing its own CASN.

In our experiment, we ran the micro-benchmark with DeCASN and RaCASN. In the RaCASN implementation, k random numbers corresponding to the k system registers were precomputed and stored in a shared array. For each execution, we measured the longest helping chain and then computed the distribution of the

[3]Terms *register* and *word* can be used interchangeably.

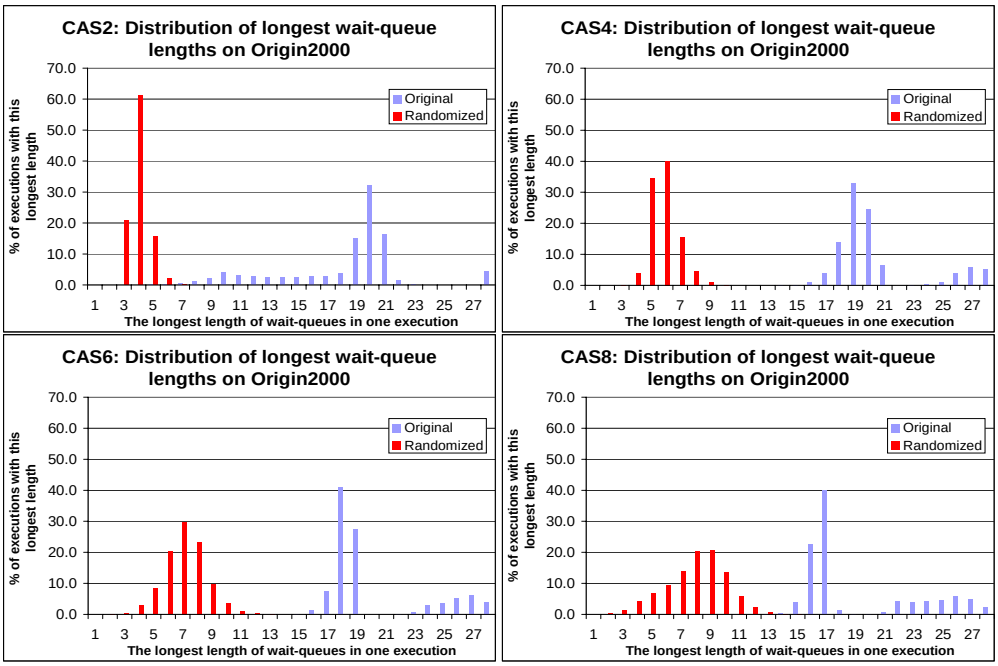

Figure 4.6: The distributions of the longest wait-queue lengths in the micro-benchmark on the SGI Origin2000.

chain lengths over one million executions. We also measured the average execution time of the micro-benchmark using DeCASN and RaCASN. Our experiment ran the micro-benchmark with 28 threads on 28 processors of the SGI Origin2000 machine. We tested the benchmark with $N = 2, 4, 6$ and 8, i.e. $CAS2$, $CAS4$, $CAS6$ and $CAS8$. The results are presented in Figure 4.6 and Figure 4.7.

Results: Figure 4.6 shows that RaCASN breaks the helping chains much better than DeCASN, thus making themselves faster. Long helping chains degrade the efficiency of the whole system since all processors related to a chain try to lock the same registers of the last CASN in the chain, which generates high collision levels on these registers.

In the case of CAS2 in Figure 4.6, RaCASN exhibits executions with the longest helping chain of length 4 in 61% of the total number of executions, of length 3 in 21% of the total number of executions and of length 5 in 16% of the total number of executions. The RaCASN longest helping chain over one million executions has length 7 in 0.2% of the total number of executions. Regarding DeCASN, it exhibits executions with longest helping chains of length 20 in 32% of

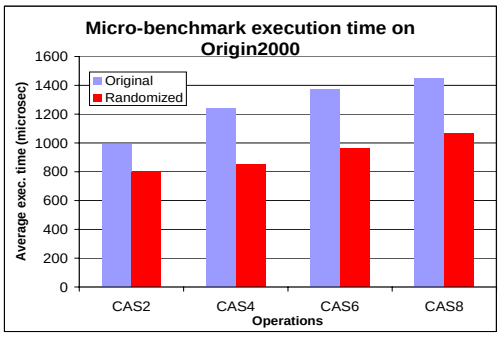

Figure 4.7: The micro-benchmark execution times on the SGI Origin2000.

the total number of executions, of length 21 in 16% of the total number of executions and of length 19 in 15% of the total number of executions. The DeCASN exhibits executions with longest helping chain of length 28, the maximal number of CASN operations, in 4% of the total number of executions.

When the number of registers to be updated increases, the distribution of RaCASN longest chain lengths shifts to the right slowly but is still much better than that of DeCASN as shown in the charts of CAS4, CAS6 and CAS8 (cf. Figure 4.6). Note that the probability that one CASN must help another grows with N. However, the length of the longest helping chain may not increase since a successful CASN can reduce this length by at least N. We can observe this effect in Figure 4.6 where the highest bar in the DeCASN longest length distribution shifts to the left slowly when N increases from 2 to 8.

Since RaCASN helps the micro-benchmark break long helping chains, which by itself reduces collision on memory and increases parallelism, RaCASN achieves better performance on the benchmark as shown in Figure 4.7. The RaCASN is from 20% to 31% faster than DeCASN. The overhead of computing k random numbers in RaCASN implementation is not significant, which consumed only 0.07 percent of the execution time.

4.5.2 The application

As we have experienced, an algorithm that gains good performance on a micro-benchmark may not keep such performance on a real application. This motivated us to do another comparison between RaCASN and DeCASN on an application.

The application comes from the following problem:

The problem: There are n registers r_1, r_2, $\cdots$, r_n, each of which belongs to one of n agents a_1, a_2, $\cdots$, a_n. The agents communicate with the underlying

computational system via these registers: agent a_k reads a result in register r_k written by the system before writing there a new input i_k for the system. Input values i_k are put in register r_k randomly and independently of other agents. The input values change all the time dynamically. (We can think that they are inputs from sensors.)

The computational system computes an output/result o_k for agent a_k from the prefix $i_1, i_2, \cdots, i_k$. For simplicity, we assume that it computes a prefix-sum

$$o_k = i_1 + i_2 + \cdots + i_k = o_{k-1} + i_k$$

for all k in $[2, n]$. The system writes the result o_k back to register r_k only if the values used to compute o_k have not changed yet. That means:

- either all registers $r_1, r_2, \cdots, r_k$ have not changed yet if o_k is computed from $i_1, i_2, \cdots, i_k$, or
- registers r_{k-1} and r_k have not changed yet if o_k is computed from o_{k-1} and i_k, where o_{k-1} had been written successfully to register r_{k-1} and no new input i_{k-1} has been put in this register since o_{k-1} was written back.

The efficiency of the computational system is evaluated by the number of results written successfully. The more results are written successfully, the better the system is.

A simple algorithm solving the problem: The following two observations can be made:

- The results must be computed as fast as possible in order to write them back to the registers before new inputs are put in them.
- Using o_{k-1} and i_k to compute o_k has higher probability of success than using $i_1, i_2, \cdots, i_k$.

Therefore, we use n threads $t_1, t_2, \cdots, t_n$, where the main task of thread t_k is to compute o_k fast. The algorithm is illustrated in Figure 4.8.

In our experiment, the CASN operation in the algorithm was in turn replaced by RaCASN and DeCASN and then the average execution times of the application were measured over one million executions. The number of registers or threads n was varied from 4 to 28. The experiment with higher n generates higher collision level on the registers due to the helping policy. In the experiment, each thread ran exclusively on one processor of the Origin2000 machine. The result is presented in Figure 4.9.

while $True$ **do**
Read registers from k to k', where register r'_k is the first reg. that is observed containing an output/result. Note $o_1 = i_1$.
Compute o_k: $o_{k'+1} := o_{k'} + i_{k'+1}; \cdots; o_k := o_{k-1} + i_k$;
if $CASN(\langle r_{k'}, r_{k'+1}, \cdots, r_k\rangle, \langle o_{k'}, i_{k'+1}, ..., i_k\rangle,$
$\langle o_{k'}, o_{k'+1}, ..., o_k\rangle) = Success)$ **then break**;
done

Figure 4.8: The algorithm for a thread t_k in computing one result/output

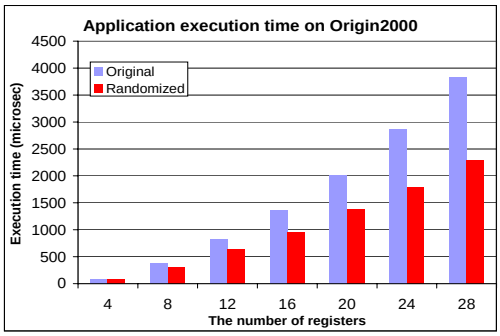

Figure 4.9: The application execution times on the SGI Origin2000.

Results: The experimental result shows that RaCASN helps the application run faster compared to DeCASN. It is up to 40% faster in the case of 28 registers or 28 threads. Figure 4.9 shows that with more threads the DeCASN/ReCASN speed-up relation grows. This implies that the randomization in RaCASN plays a significant role in reducing collisions on the shared registers, thus helping the application achieve better performance. The overhead of pre-computing n random numbers corresponding to n registers is not significant: it takes at most 0.7% of the execution time. (This worst case is measured in the case where the number of register is 4.)

4.6 Conclusions

In this paper we advocated randomization for implementing multi-locking such as CASN efficiently. We showed that our approach is efficient, in theory as well as in practice.

In the past, multi-lock algorithms were usually evaluated by random simulations. That is, in an evaluation/simulation of an algorithm it was assumed that *randomly* chosen registers were accessed by the processes. We believe that this

is a conceptual *faux pas*. In fact, shared memory processes operate on shared data structures (e.g. search trees, linked lists) which are accessed anything but randomly. In reality, as in dining philosophers, access is not random but well-structured. For example, in a shared ordered linked list a process needs to multi-lock the two *neighbor* records in order to insert a new record.

By shifting the randomization from the simulation to the actual implementation our system is efficient in any application, as worst-case as it may be.

Chapter 5

Reactive Spin-locks: A Self-tuning Approach[1]

Phuong Hoai Ha[2], Marina Papatriantafilou[2], Philippas Tsigas[2]

Abstract

Reactive spin-lock algorithms that can automatically react to contention variation on the lock have received great attention in the field of multiprocessor synchronization. This results from the fact that the algorithms help applications achieve good performance in all possible contention conditions. However, to make decisions, the reactive schemes in the existing algorithms rely on (i) some fixed *experimentally tuned thresholds, which may get frequently inappropriate in dynamic environments like multiprogramming/multiprocessor systems, or (ii) known probability distributions of inputs, which are not usually feasible.*

This paper presents a new reactive spin-lock algorithm that is completely self-tuning, which means no experimentally tuned parameter nor probability distribution of inputs are needed. The new spin-lock is based on both synchronization structures of applications and a competitive online algorithm. Our experiments, which use the Spark98 kernels and the SPLASH-2 applications as application benchmarks, on a multiprocessor machine SGI Origin2000 and an Intel Xeon workstation have showed that the new self-tuning spin-lock helps the applications

[1]Expanded version of a preliminary result published in the Proceedings of the 8th IEEE International Symposium on Parallel Architectures, Algorithms and Networks (I-SPAN '05), Dec. 2005, pp. 33-39, IEEE press.

[2]Department of Computer Science and Engineering, Chalmers University of Technology, S-412 96 Gothenburg, Sweden. Email:{*phuong,ptrianta,tsigas*}*@cs.chalmers.se*.

with different characteristics nearly achieve the best performance in a wide range of contention levels.

5.1 Introduction

Multiprocessor systems aim at supporting parallel computing environments, where processes are running concurrently. In such parallel processing environments the interferences among processes are inevitable. Many concurrent processes may cause high traffic on the system bus (or network), high contention on memory modules and high load on processors; all these slow down process executions. These interferences generate a variable and unpredictable environment to each process. Such a variable environment consequently affects interprocess-synchronization methods like spin-locks. Some sophisticated spin-locks such as the MCS queue-lock [77] are good for high-load environments, whereas others such as the *test-and-test-and-set* lock [2, 7] are good for low-load environments [66]. This fact raises a question on constructing reactive spin-locks that can adapt to load variation in their environment so as to achieve good performance in all conditions.

There exist reactive spin-lock algorithms in the literature [2, 7, 66, 67]. Spin-lock using the *test-and-test-and-set* operation with exponential backoff ($TTSE$) [2, 7] is an example: every time a waiting process reads a busy lock, i.e. there is probably high contention on the lock, it will double its backoff delay in order to reduce the contention. Another reactive spin-lock that can switch from spin-lock using $TTSE$ to a sophisticated local-spin queue-lock when the contention is considered high was suggested in [66, 67].

However, these reactive spin-locks suffer some drawbacks. First of all, their reactive schemes rely on either some experimentally tuned thresholds or known probability distributions of some inputs. Such *fixed* experimental threshold-values may frequently become inappropriate in variable and unpredictable environments such as multiprogramming systems. Assumption on known probability distributions of some inputs is not usually feasible. Further, the reactive spin-locks do not adapt to synchronization characteristics of applications and thus they are inefficient for different applications. We observe that characteristics of applications such as delays inside/outside the critical sections have a large impact on which spin-lock will help the applications achieve the best performance. Lim's reactive spin-lock [66], which switches to $TTSE$ when contention is low and to the MCS queue-lock when contention is high, was showed inefficient to some real applications [59]. A good reactive spin-lock should not only react to the contention variation on the lock, but also adapt to a variety of applications with different characteristics.

These issues motivated us to design a new reactive spin-lock that requires nei-

ther experimentally tuned thresholds nor probability distributions of inputs. The new spin-lock moreover adapts itself to applications, keeping its good performance on different applications.

---

while $true$ **do** *Noncritical section*; *Entry section*; *Critical section*; *Exit section*; **od**

Figure 5.1: The structure for parallel applications

We classify spin-locks into two categories: *arbitrating locks* such as ticket-locks [61] and queue-locks [21, 32, 73, 77] and *non-arbitrating locks* such as TAS locks [2, 7]. *Arbitrating locks* are locks that identify who is the next lock holder in advance. The rest of spin-locks are *non-arbitrating locks*.

Arbitrating locks and non-arbitrating locks each have their own advantages. Arbitrating locks prevent processors from causing bursts in network traffic as well as high contention on the lock. This is because they avoid the situation that many processors concurrently realize the lock available and thus concurrently try to acquire the lock [4, 7, 55, 59, 77]. Although the advantages of arbitrating spin-locks have been studied so widely, the following advantages of non-arbitrating spin-locks have not been studied deeply. Non-arbitrating locks have two interesting properties: i) tolerance to crash failures in the lock-competing phase, the *Entry section* in Figure 5.1, and ii) ability of exploiting *cache affinity* [57, 96, 99, 104, 105] and the underlying system supports such as page migration [64]. The lock holder can re-acquire the lock and re-use the exclusive shared data many times before the lock is acquired by another processor, saving time used for transferring the lock and the shared data from one to another. From experiments we observe that the non-arbitrating locks is favored by applications with the critical section much larger than the non-critical section (cf. Figure 5.1) to exploit locality/cache whereas the arbitrating locks is favored by ones with the critical section much smaller than the non-critical section to avoid bursts both in network traffic and in memory contention. This implies that characteristics of a specific application can decide which kind of locks helps the application achieve better performance. (Further discussions on the advantages of both lock categories continue in Section 5.2.)

5.1.1 Contributions

We designed and implemented a new reactive spin-lock with the following properties:

- It is completely self-tuning: neither experimentally tuned parameters nor probability distributions of inputs are needed. The new reactive scheme automatically adjusts its backoff delay reasonably according to contention on

the lock as well as characteristics of applications. The scheme is built on a competitive online algorithm. What it needs from the system is only the ratio of the latency of remote memory references to the latency of level 1 cache references, which is available in documents about the system architecture.

- It combines the advantages of both arbitrating and non-arbitrating spin-locks. In order to achieve this property, the new spin-lock does not use *strict* arbitrations like ticket-locks, but instead introduces a *loose* form of arbitration. This allows the spin-lock to be able to exploit locality. Combining a *loose* arbitration with a suitable reactive backoff scheme helps the new spin-lock achieve the advantages of the both categories.

In addition to proving the correctness of the new spin-lock, in order to test its feasibility we ran experiments using the Spark98 kernels [84] and the SPLASH-2 applications [109] as application benchmarks on an SGI Origin2000, a well-known commercial ccNUMA system, and a popular workstation with two Intel Xeon processors. These experiments showed that in a wide range of contention levels the new reactive spin-lock performed nearly as well as the best, which was manually tuned for each benchmark on each system.

The synchronization primitives related to our algorithms are *fetch-and-add (FAA)* and *compare-and-swap (CAS)*, which are available in most recent systems either in hardware like Intel, Sun machines or in software like SGI machines. The definitions of the primitives are described in Figure 5.2.

TAS(x) $atomically\{$ $oldx \leftarrow x$; $x \leftarrow 1$; **return** $oldx$; $\}$ /* *init:* $x \leftarrow 0$ */
FAA(x, v) $atomically$ $\{$ $oldx \leftarrow x$; $x \leftarrow x + v$; **return**$(oldx)$ $\}$
CAS(x, old, new) $atomically$ $\{$
if$(x = old)$ **then** $x \leftarrow new$; **return**$(true)$; **else return**$(false)$; $\}$

Figure 5.2: Synchronization primitives, where x is a variable and v, old, new are values.

The rest of this paper is organized as follows. Section 5.2 describes our problem analysis, which led and motivated this work. Section 5.3 models the spin-lock problem as an online problem. Section 5.4 presents a new competitive algorithm for reactive spin-locks. Section 5.5 presents correctness proofs of the new spin-lock. Section 5.6 presents a heuristic for the new reactive spin-lock to adapt to synchronization characteristics of applications. Section 5.7 presents the performance evaluation of the new reactive spin-lock and compares the spin-lock with representatives of arbitrating and non-arbitrating spin-locks using the application benchmarks. Finally, Section 5.8 concludes this paper.

5.2 Problem analysis

5.2.1 Tuning parameters and system characteristics

In general, besides the cost for experimentally tuning the parameters, the reactive spin-locks using tuned parameters cannot always achieve good performance because the parameters depend on the system utilization, which in turn is affected by other applications running concurrently. Thus, tuned parameters at some point of time may become obsolete at a later point of time when they are used. Further, reactive spin-locks may also need to take care of properties of applications such as delays inside/outside the critical section when choosing locking protocols.

Regarding the algorithm-system interplay, there is also the issue of arbitrating vs. non-arbitrating locks, which implies different benefits, as explained in the introduction. In arbitrating locks, the lock and the data used in the critical section must be transferred from one processor to another according to their order in the waiting queue, regardless of how far the distance between these two processors is in the system. This generates high transmission cost. Contrarily, in non-arbitrating locks, the processors closest to the current lock owner, for instance processors in the same node in NUMA systems, have higher probability to acquire the lock because they will realize the lock available sooner. Moreover, when there are many requests for the lock from processors in the same node, the system may move the memory page containing the lock to the local memory of that node, giving these processors higher probabilities to acquire the lock the next time. Unlike the arbitrating locks, the non-arbitrating locks also have the ability to tolerate faults in the Entry section (cf. Figure 5.1). In the Entry section, the non-arbitrating locks prevent slow or crashed processors from blocking other fast processors.

Although arbitrating locks such as ticket locks and queue-locks are considered as fair locks in the literature, their fairness may still depends on the applications using the locks as well as on the architecture of the system on which the applications are running. Regarding the ticket lock, if processors in the same node of a NUMA system, whose local memory is storing the ticket variable, execute the iteration in Figure 5.1 so fast that they continuously get new tickets again before the ticket variable can be accessed by processors on other nodes, the ticket lock becomes unfair. Similar for the queue-lock, if processors in the same node of a NUMA system, whose local memory is storing the pointer used to enqueue the waiting queue, execute the iteration in Figure 5.1 so fast that they continuously enter the waiting queue before processors on other nodes have a chance to so, the queue lock may become unfair.

5.2.2 Experimental studies

To see whether the above concerns have a sound basis, we conducted an experimental study. We used the Spark98 shared memory program lmv [84] on an SGI Origin3800. The system has 31 500MHz MIPS R14000 CPUs with 8MB L2 cache each. These experiments confirmed our observations. In order to compare performance among spin-lock algorithms, we need benchmarks where the contention level on the lock is high. Therefore, we used only one lock to synchronize updates of the result array in the Spark98 kernel. We used the largest pack file $sf5.1.pack$ [84] as input.

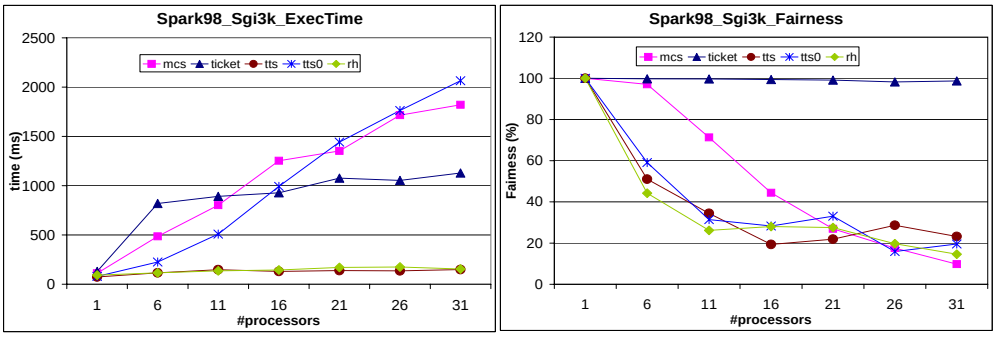

Figure 5.3: The execution time and the lock fairness of the Spark98 benchmark on an SGI Origin3800.

The left and the right charts in Figure 5.3 show the execution times and the fairness of the Spark98 kernel using the MCS queue-lock (mcs), the ticket lock with proportional backoff tuned for the SGI Origin3800 ($ticket$), TTSE with backoff parameters tuned for the SGI Origin3800 (tss), TTSE with backoff parameters mentioned in [88] ($tts0$) and the RH lock [86] with backoff parameters tuned for the SGI Origin3800 (rh). The contention level on the lock is adjusted by changing the number of processors accessing it. For instance, in the case of 31 processors, there is the highest contention level on the lock. The source codes for $TTSE$ and MCS are from [88]. The implementation of $ticket$ is similar to Figure 2 in [77].

From the left chart in Figure 5.3, we can see that the non-arbitrating locks such as $TTSE$ and RH both with tuned parameters outperform the arbitrating locks such as the MCS queue-lock and the ticket lock when the contention level increases. That is because the $TTSE$ and RH exploit the locality/caching among processors within the same node. Moreover, they do not suffer the *lock convoy* problem in the entry section.

The left chart also shows the problem of existing reactive spin-locks such as

$TTSE$: their performance strongly depends on the experimentally tuned parameters. Inaccurately chosen parameters will lead to bad performance as depicted in the left chart between the TTSE with parameters tuned for SGI Origin3800 (tss) and the TTSE with parameters mentioned in [88] ($tts0$). The latter is about 14 times slower than the former in the case of 31 processors, which is a big difference on application performance.

The right chart in Figure 5.3 shows the fairness of these spin-locks. Here, the processor first finishing its own task will send a signal to all other processors to stop and to count the number of times each processor has successfully acquired the lock. In this chart, the most interesting is the fairness of the MCS queue-lock, which is normally considered as a fair lock. Beyond a certain number of processors, from fairness point of view, the MCS queue-lock does not seem better than other non-arbitrating locks such as $TTSE$ and RH. From the log file of the experiment in the case of 31 processors, we saw that a group of 16 processors connected together via the same router, had a number of lock accesses much greater than those of other processors connected via another router. That means that the group of 16 processors connected via the same router executed their own tasks so fast that they continuously successfully updated the pointer to enqueue before the pointer could be updated by other processors of another router. That means that even fair arbitrating spin locks cannot always ensure fairness for arbitrary applications running on arbitrary systems.

All these factors are considered seriously in the design of our new reactive spin-lock.

5.3 Modeling the problem

In this section we model the spin-lock problem as an online problem. The theoretical model of parallel applications in our research is typically described as a set of threads with the structure shown in Figure 5.1 [4]. We consider a system with P sequential processes running on P processors. We assume that each process runs on one processor, which is common in recent systems such as SGI Origin2000. In this case, we do not need to switch the process state from spinning to blocking in the *Entry section* (cf. Figure 5.1), i.e. there is no context-switching cost in the spin-lock overhead [58].

First of all, we determine the upper/lower bounds of backoff delays between two consecutive spins. Let "delay base" $base_l$ of a lock l be the average interval in which the lock holder keeps the lock locally before yielding it to another process/processor. In order to obtain a high probability of spinning a free lock, a backoff delay $delay_i$ between two consecutive spins of a process p_i on the lock l

should not be smaller than $base_l$, $base_l \leq delay_i$. On the other hand, according to Anderson [7] the upper bound of backoff delays should equal the number of processes potentially interested in acquiring the lock so that the backoff has the same performance as statically assigned slots when there are many spinning processes. This implies $delay_i \leq P \cdot base_l$, where P is the number of processes potentially interested in acquiring the lock. In conclusion,

$$base_l \leq delay_i \leq P \cdot base_l \tag{5.1}$$

where $delay_i$ is a time-varying measure.

Secondly, we look at the problem of how to compute a reasonable $delay_i$ for the next backoff every time a waiting process p_i observes a busy lock. In the $TTSE$ spin-lock [7], the backoff delay $delay_i$ is doubled up to some limit every time a waiting process reads a busy lock. In fact, the backoff scheme in the $TTSE$ spin-lock comes from Ethernet's backoff scheme for networks with characteristics different from those of spin-locks. In networks the cost to a collision is equal and independent of the number of processes whereas in spin-locks the cost depends on the number of participating processes [7]. Therefore, the backoff scheme in the $TTSE$ spin-lock is not competitive and its performance strongly depends on how well its base/limit values are chosen.

Let "delay surplus" $surplus_i$ of a process p_i be

$$surplus_i = (P \cdot base_l - delay_i) \tag{5.2}$$

We have $0 \leq surplus_i \leq (P-1) \cdot base_l$. Like $delay_i$, $surplus_i$ is a time-varying measure.

Definition 5.3.1. *A* load-rising (resp. load-dropping) transaction phase *is a maximal sequence of processes' subsequent visits at the lock with monotonic non-decreasing (resp. non-increasing) contention level on the lock*[1]*. A load-rising phase ends when a decrease in contention is observed. At that point, a load-dropping phase begins.*

Our goal at this moment is to design a reactive non-arbitrating spin-lock whose backoff delay (or delay in short) is dynamically and optimally adjusted to contention variation on the lock. This implies that we need to minimize two opposite factors: i) the delay between a pair of lock release and lock acquisition due to the backoff and ii) the communication bandwidth used by spinning processes as well as the load on the lock.

[1]The contention level on a lock is measured by the number of processes that are competing for the lock, cf. Section 5.4.

This is an online problem. Whenever a spinning process p_i observes a load increase on the lock, it has to decide whether it should increase its $delay_i$ now. If it increases its delay too soon, it will waste time on a long backoff delay when the lock becomes available. If it does not increase its delay in time, it will cause the same problems as the spin-lock using TTS like high network traffic and high contention on the lock, which consequently delay the lock holder to release the lock. If the process knew in advance how contention on the lock would vary in the whole competing period, it would be able to find an optimal solution. However, there is no way for processes to know that information, the information about the future in an unpredictable environment.

We are interested in designing a deterministic online algorithm against a malicious adversary for the spin-lock problem. In such kind of problems, randomization cannot improve competitive performance [28]. For deterministic online algorithms the adversary with the knowledge of the algorithms generates the worst possible input to maximize the competitive ratio. The adversary creates transaction phases that fool the player, a process competing for the lock, to increase/decrease his delay incorrectly. This makes the player end up with a bad result whereas the adversary still achieves the best result.

Figure 5.4 illustrates how the adversary can create such transaction phases. Assume that the adversary designs A as an optimal load-point to increase the delay and B as an optimal load-point to decrease the delay. Since the adversary has both knowledge of the deterministic algorithm used by the player and full control on creating load inputs, the malicious adversary can add a sequence of load-rising points $\cdots \leq a_1 \leq a_2 \leq \cdots \leq a_n < A$ that fools the player to increase his delay up to the maximum before the load reaches A (i.e. to fool the player to increase his delay too soon). When the player observes a load increase on the lock, he will increase his delay according to his deterministic algorithm, and eventually his delay reaches the maximum at some point a_i before the load reaches point A.

The goal of online/offline algorithms is to maximize $\mathcal{P} = \sum_{t \in T_j} \Delta surplus_{i,t} \cdot l_t$ for each transaction phase T_j, where l_t is the load at time $t \in T_j$ and $\Delta surplus_{i,t}$ is the additional amount of surplus that the player/process p_i spends at load l_t. The idea behind this goal is to put a longer delay at a higher contention level reasonably. For the game in Figure 5.4, the adversary achieves the best value $\mathcal{P}$ at A since he will use all his surplus "budget", $(P-1) \cdot base_l$, at the suitable load-point A where l_t becomes maximum in the load-rising transaction phase T_j. That means the player increases his delay too soon, wasting time on a long backoff delay when the lock becomes available.

Similarly, the adversary can fool the player on the load-dropping phase from A to B by adding a sequence of load-dropping points $b_1 \geq b_2 \geq \cdots \geq b_m > B$. When the player observes a load decrease on the lock, he decreases his delay, and

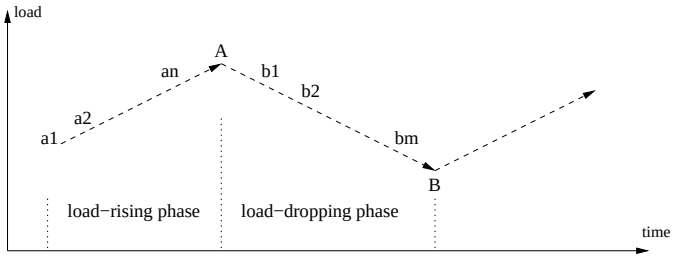

Figure 5.4: The transaction phases of contention variations on the lock.

eventually his delay reaches the minimum at some point b_j before the load reaches point B. That means the player decreases his delay too soon, causing high network traffic and high contention on the lock.

Lastly, we determine upper/lower bounds of loads l_t on the lock. The load is the number of processes currently waiting for the lock, i.e. $l_t \leq P$. On the other hand, a process needs to delay only if it could not acquire the lock, so we have $1 \leq l_t \leq P$.

In summary, the spin-lock problem can be described as the following online game. With known upper/lower bounds of loads l_t on the lock,$1 \leq l_t \leq P$, the player (a process i_i) needs to spend his initial delay surplus (e.g. $(P-1) \cdot base_l$) at l_t efficiently. Loads l_t are unfolded on-the-fly and when a new value l_t is observed, a new period starts. Given a current load value, the player has to decide how much of his delay surplus should be spent at the current load, i.e. how much his current backoff delay should be lengthened at the current load.

5.4 The algorithm

In order to play against the malicious adversary, the player needs a *competitive* online algorithm for computing his backoff delay. When the load on the lock increases, the player has to reduce his delay surplus, $surplus$, by exchanging it with another asset called $savings$. When the load decreases, he increases $surplus$ by exchanging this $savings$ back to $surplus$.

The idea of our spin-lock algorithm is as follows. During a load-rising phase T_j, when the player observes a load increase on the lock, he increases his delay *just enough* to keep a bounded competitive ratio even if the load suddenly drops to the minimum in the next observation. The amount of time by which the player's delay increases is computed similarly to the *threat-based* method of [28]. The online algorithm for computing the delay can be described by the following rules:

- The delay is increased only when the load is the highest so far in the present transaction phase.
- When increasing delay, increase *just enough* to keep the competitive ratio $c = P - \frac{P-1}{P^{1/(P-1)}}$, even if the load drops to the minimum in the next observation.

The amount of time by which the delay should increase is:

$$\Delta delay = \Delta surplus = initSurplus \cdot \frac{1}{c} \cdot \frac{load - load^{-}}{load - 1} \tag{5.3}$$

where $initSurplus$ is $surplus$ at the beginning of a load-rising transaction phase, $load$ is the present load on the lock observed by the player, and $load^{-}$ is the highest load on the lock before the present observation (cf. procedure $Surplus2Savings$ in Figure 5.5).

The online algorithm is presented via pseudo-code in Figure 5.5. Every time a new load-rising transaction phase starts, the value $initSurplus$ is set to the last value of $surplus$ in the previous transaction phase (lines C2, C3). At the beginning of a transaction, the load is initialized to $counter$ and $delay = counter \cdot base_l$, where $counter$, a sort of ordering tickets, shows how many processes are competing for the lock. The $counter$ is obtained when the process reads the lock at the first time (line A1). Each process chooses an initial $surplus$ with respect to its own ticket/counter (line A2)

$$initSurplus = (P - counter) \cdot base_l \tag{5.4}$$

This helps the new spin-lock partly prevent processes from concurrently observing a free lock, the worst situation for non-arbitrating spin-locks.

Symmetrically, in a load-dropping phase the amount of time by which the player's delay should decrease is computed by applying the same method with only one change, namely that the value of load on the lock $load$, which is decreasing, is replaced by the inverse $\frac{1}{load}$ (cf. procedure $Savings2Surplus$).

Finally, we briefly explain the whole spin-lock algorithm via the pseudo-code in Figure 5.5. In order to know the load on a lock, we need a counter to count how many processes are concurrently competing for the lock. If we used a separate counter, we would generate additional bottleneck beside the lock. Therefore, we use a single-word variable to contain both the lock and the counter (cf. $LockType$ in Figure 5.5).

A process p_i calls procedure $Acquire(L)$ when it wants to acquire lock L. The structure of the procedure is similar to the spin-lock using TTS except for the

ways to compute the delay and to update the lock. First, p_i increases both values $\langle lock, counter\rangle$ by 1 (line $A1$). The lock L has been occupied if $L.lock \neq 0$. When spinning the lock locally (line A5), if p_i observes a free lock, i.e $L.lock = 0$, it will try to acquire the lock by increasing only field $L.lock$ by 1 (field $L.counter$ is kept intact, line A7). It will successfully acquire the lock if no other processes have acquired the lock in this interval, i.e. $cond.lock = 0$ (line A8).

Process p_i calls procedure $Release()$ when releasing the lock. The procedure has to do two tasks atomically: i) reset the $lock$ field and ii) decrease the $counter$ field by 1. The CAS primitive can do these tasks atomically (line R2).

Lemma 5.4.1. *In each load-rising/load-dropping phase, the new deterministic spin-lock algorithm is competitive with a competitive ratio* $c = P - \frac{P-1}{P^{1/(P-1)}} = \Theta(\log P)$*, where* P *is the number of processes potentially interested in the lock.*

Proof. The proof is similar to that of the threat-based policy in [28]. □

Theorem 5.4.2. *The new spin-lock algorithm guarantees mutual exclusion and non-livelock. Its space complexity is* $\Theta(\log P)$ *for systems with* P *processors.*

Proof. The proof can be found Section 5.5. □

5.5 Correctness

The correctness of the new algorithm follows almost straightforward from its description. In particular, due to the atomicity properties of FAA and CAS, we have that:

Lemma 5.5.1. *The number of processors currently waiting for the lock* $L.counter$ *is counted correctly.*

Lemma 5.5.2. *The space need for the lock field of* $LockType$ *is* $\log(P)$ *for systems with* P *processors.*

Proof. Let Δt denote an interval since the $lock$ field of a lock L is increased to 1 at line $A1$ or $A7$ until it is reset to 0 at line $R2$. In Δt, each processor p_i can increase the $lock$ field by at most one. Indeed, if p_i increases $lock$ by 1 at line $A1$ or $A7$, it no longer increases $lock$ at line $A7$ because line $A7$ is executed only if $lock = 0$ (line $A6$); it cannot also increase $lock$ at line $A1$ because each processor only executes $A1$ once at the beginning of procedure $Acquire$.

Therefore, in Δt the $lock$ field is increased by at most P. That means the value of the lock field is never greater than P, the number of processors. □

```
type LockType = record lock, counter : [1..MaxProcs]; end; /*stored in one word*/
     LockStruct = record L : LockType; base : int; end;
     InfoType = record load⁻ : [1..MaxProcs]; phase : {Rising, Dropping};
                       surplus, initSurplus : int; savings, initSavings : int; end;
private variables info : InfoType;

ACQUIRE(LockStruct pL)
A1  L := FAA(&pL.L, ⟨1,1⟩); /*increase counter,try to take lock*/
    if L.lock then /*lock is occupied*/
A2    info.initSurplus := info.surplus := (P − L.counter) · pL.base; /*initialize variables*/
      info.initSavings := info.savings := (L.counter · pL.base) · L.counter;
A3    delay := ComputeDelay(info, L.counter, pL.base);
      cond := ⟨1,0⟩; /*conditional variable for while loop*/
      do
A4      sleep(delay);
A5      L = pL.L; /*read lock again*/
A6      if L.lock then /*lock is still occupied*/
          delay := ComputeDelay(info, L.counter); continue;
A7      cond = FAA(&pL.L, ⟨1,0⟩); /*try to take lock*/
A8    while cond.lock;

int COMPUTEDELAY (InfoType I, int load, int base)
    FirstInPhase := False;
    if I.phase = Rising and load < I.load⁻ then
C1    I.phase := Dropping; I.initSavings := I.savings; FirstInPhase := True;
    else if I.phase = Dropping and load > I.load⁻ then
C2    I.phase := Rising; I.initSurplus := I.surplus; FirstInPhase := True;
C3  if I.phase = Rising then Surplus2Savings(I, load, FirstInPhase);
C4  else Savings2Surplus(I, 1/load, FirstInPhase);
C5  I.load⁻ := load;
C6  return (P · base − I.surplus);

SURPLUS2SAVINGS (InfoType I, int load, bool FirstInPhase)
    X := I.surplus; initX := I.initSurplus; Y := I.savings; rXY := load; rXY⁻ := I.load⁻;
    if FirstInPhase then
      if rXY > mXY · C then /*mXY: lower bound of rXY*/
S1      ΔX := initX · 1/C · (rXY − mXY · C)/(rXY − mXY); /*C: comp. ratio*/
    else
S2    ΔX := initX · 1/C · (rXY − rXY⁻)/(rXY − mXY);
S3  I.surplus := I.surplus − ΔX; I.savings := I.savings + ΔX · rXY;

SAVINGS2SURPLUS (InfoType I, 1/load, bool FirstInPhase)
/* Symmetric to procedure Surplus2Savings with:
    X := I.savings; initX := I.initSavings; Y := I.surplus; rXY := 1/load; rXY⁻ := 1/I.load⁻; */

RELEASE (LockType pL)
R1  do L := pL.L;
R2  while not CAS(&pL.L, L, ⟨0, L.counter − 1⟩); /*release lock & decrease counter*/
```

Figure 5.5: The Acquire and Release procedures

Lemma 5.5.3. *The new reactive spin-lock allows only one processor to enter the critical section at a point of time.*

Proof. A processor p_i can enter the critical section only if the lock field of the value that p_i gets from the FAA primitive at line $A1$ or $A7$ is 0. Due to the atomicity properties of FAA primitives, at one point of time at most one processor can observe that the lock field is 0, and can become the lock holder. Only when the lock holder exits the critical section, the field is reset to 0 (line $R2$), allowing another processor to enter the critical section. □

These lemmas imply the following theorem:

Theorem 5.5.4. *The new spin-lock algorithm guarantees mutual exclusion and non-livelock. Its space complexity is $\Theta(\log P)$ for systems with P processors.*

5.6 Estimating the delay base

So far we have assumed that the basic interval $base_l$ in which a process p_i keeps the lock l locally before yielding it to other processes is known. This section describes how the new spin-lock estimates the $base_l$ based on characteristics of each parallel application such as delays outside/inside the corresponding critical section (cf. Definitions 5.6.3).

Like the *delay base* in the $TTSE$ spin-lock, the $base_l$ is just a basic value at the beginning from which the online algorithm in Section 5.4 starts to adjust the backoff delay according to contention variation. Instead of forcing programmers to tune the value manually, the new spin-lock estimates the value automatically.

First, we define terms used in this section.

Fairness: In order to evaluate fairness of spin-locks, we consider them on applications whose threads do the same task but on different data. Fairness is introduced to evaluate unbalanced situations where a thread may successfully acquire the lock many times more than the others may. Fairness is an interesting aspect of spin-lock algorithms, which may help the application gain performance in multiprocessor systems by utilizing all processors concurrently in high-load cases. Since threads cannot make any progress when waiting for the lock, only the lock holder utilizes one of system processors. If the lock holder continuously and successfully re-acquires the lock, only one processor will be used for useful task. In contrast, if the spin-lock is so fair that each thread can acquire the lock in turn, all threads will concurrently utilize system processors to execute their non-critical section task in parallel (cf. Figure 5.1).

Assume that in a period Δt there are N processors concurrently executing the code with structure as in Figure 5.1. These processors start and end outside Δt.

That means we are only interested in the fairness for periods Δt in which all N processors *are concurrently and continuously competing* for the lock.

Definition 5.6.1. *Call n_i the number of times each of N processors p_i has successfully acquired a lock in a period Δt.* Fairness *of the lock in the period can be computed using the following formula:*

$$fairness_{\Delta t} = \frac{\sum_i n_i}{\max_i n_i \cdot N} \tag{5.5}$$

The lock that can keep its fairness in a shorter Δt is the better.

Overhead and delay: In most systems, the latencies of memory references vary with memory levels. Let the latency of accessing L1 cache be a time unit, we have the following definition:

Definition 5.6.2. Overhead *of yielding a cached variable such as a lock to another processor in order to achieve good fairness is:*

$$overhead = \frac{latency\ of\ remote\ memory\ reference}{latency\ of\ L1\ cache\ reference} \tag{5.6}$$

Definition 5.6.3. Delay outside a critical section (DoCS) *is the interval since the lock holder releases the lock in the Exit section until the first attempt to re-acquire it in the Entry section (cf. Figure 5.1).* Delay inside a critical section (DiCS) *is the interval when the lock holder is in the Critical section.*

In the new spin-lock, the delays outside/inside a critical section ($DoCS$/$DiCS$) are estimated by an individual process when needed. In fact, $DoCS$ and $DiCS$ influence the backoff delay strongly. If the backoff delay is chosen inaccurately, application performance degrades significantly (cf. tts ($TTSE$ with tuned thresholds) and $tts0$ ($TTSE$ without tuned thresholds) in Figure 5.3).

We have found a reasonable heuristic to estimate the delay base by $DoCS$.

The heuristic: *The delay base for a lock l, $base_l$, can be estimated by the delay outside the corresponding critical section, $DoCS$, using the following formula:*

$$base_l = \frac{a \cdot DoCS + b}{DoCS^2} \tag{5.7}$$

where a and b are constants.

Indeed, if the DoCS approaches 0, i.e. the whole execution time of the application is inside the critical section, the application should be executed by only one processor to reduce the cost of transferring data among processors, i.e. $base_l \to \infty$. In this case, the profit of concurrently executing non-critical sections on processors

is too small compared to the cost of transferring critical data from one processor to another. On the other hand, if the DoCS approaches infinite, a processor after finishing its current iteration (cf. Figure 5.1) should immediately yield the lock to other processors so that other processors can use the lock, i.e. $base_l$ is approximately zero[2]. In this case, without knowledge of interference pattern among processes/processors the lock holder should immediately yield the lock to others since he will almost not acquire the lock again due to $DoCS \rightarrow \infty$. Therefore, the function $g(x)$ to compute the delay base $base_l$ from $DoCS$ has the following form

$$y = g(x) = \frac{f_n(x)}{f_{n+k}(x)} \tag{5.8}$$

where $k \geq 1$ is an integer and $f_n(x) = a_n x^n + \cdots + a_1 x^1 + a_0$

On the other hand, the benefit of successfully acquiring the lock, i.e. the period of using the lock locally, should not be smaller than the overhead of yielding the lock to another processor in order to support the fairness (cf. Definition 5.6.2). Therefore, $overhead \leq base_l$. If all processors keep the lock in the minimum time $base_l = overhead$ to minimize Δt in Definition 5.6.1, the time for the lock to visit all $P - 1$ other processors and then come back to p_i is

$$\begin{aligned}&(base_l + transmission\ delay) \cdot P \\ &= (overhead + overhead) \cdot P = 2 \cdot overhead \cdot P\end{aligned}$$

If $DoCS = 2 \cdot overhead \cdot P$, this is an optimal situation. This is because each processor p_i always successfully acquires the lock when it needs, i.e. the lock comes back to p_i after $DoCS$, and all other processors can exploit p_i's interval $DoCS$ to successfully acquire the lock. Therefore, the chart of function $g(x)$ must contain a point $M = (2 \cdot overhead \cdot P, overhead)$.

Moreover, when $DoCS = overhead$, which is small, in order to support the fairness the $base_l$ should be long enough so that the ticket/counter in the new algorithm (Figure 5.5) can be accessed by $P - 1$ other processors before the current lock holder gets another ticket, i.e.

$$\begin{aligned}base_l &= transmission\ delay \cdot (P - 1) \\ &= overhead \cdot (P - 1)\end{aligned}$$

Therefore, the chart of function $g(x)$ must contain a point $N = (overhead, overhead \cdot (P - 1))$.

Since the chart of $g(x)$ must contain both points M and N, the simplest form of $g(x)$ that can satisfy this requirement is

$$y = g(x) = \frac{a \cdot x + b}{x^2} \tag{5.9}$$

[2]Accurately, $base_l = DiCS$. Nevertheless, because $DoCS \rightarrow \infty$, i.e $DiCS$ is too small compared with $DoCS$, we ignore $DiCS$, i.e $base_l \approx 0$

where a, b are constants and can be found via points M and N. Each lock l in a parallel application has its own base $base_l$, which is estimated once at the beginning via the delay outside the corresponding critical section $DoCS$.

Applications using many small locks[3]: Timing functions are costly and thus the new spin-lock should estimate $base_l$ only for locks l with significant impact on the application performance, i.e. those are accessed many times during application execution. Moreover, in order to avoid oscillation at the beginning of application, which may make $base_l$ be estimated inaccurately, the new spin-lock starts to estimate $base_l$ after an interval that is long enough for all processes/processors to be able to acquire the lock l once. As discussed above, the benefit of acquiring the lock should be greater than the overhead of transferring the lock, so each processor should keep the lock in a period not smaller than $overhead$. Therefore, the new spin-lock starts to estimate $base_l$ after an interval of $2 \cdot overhead \cdot P$ since the beginning of the execution. In this initial interval, the new spin-lock uses the ticket lock with proportional backoff and $base_l$ is initialized to $overhead$. After the $base_l$ is estimated, the new spin-lock uses the reactive spin-lock in Figure 5.5.

5.7 Evaluation

Choosing non-arbitrating/arbitrating representatives : To keep graphs uncluttered we chose an efficient representative for each category (i.e. *arbitrating* and *non-arbitrating*).

We chose the ticket lock with proportional backoff ($TicketP$) as the representative for the *arbitrating lock* since:

- the $TicketP$ performs better than the MCS queue-lock[4] when using application benchmarks Spark98 (cf. Figure 5.3) and SPLASH-2 (cf. [59]), and
- from the fairness point of view, the $TicketP$ is better than the queue lock (cf. Figure 5.3).

Although the ticket lock is considered not as scalable as the MCS queue-lock since in the former processes spin on centralized variables, this is not a performance issue for the ticket lock on recent machines with cache-coherent support as long as the backoff delay of the ticket lock is tuned well. Moreover, the ticket lock gains further performance due to its simplicity and fairness. The implementation of $ticketP$ was similar to Figure 2 in [77], where the time unit was experimentally tuned for both the benchmarks and the evaluation systems to achieve the best performance.

[3]The *small* locks are locks that are used very few times and on which contention level is low.

[4]The MCS queue-lock performance is comparable with those of other queue-locks [77, 88]

For non-arbitrating spin-locks, we chose as the representative the $TTSE$ with backoff parameters tuned for both the benchmarks and the evaluation systems. The RH lock in [86] shows its advantages compared to the $TTSE$ lock only if the system has two nodes and the latency of local memory references within a node is much smaller than the latency of remote memory references to the other node, which is not the case in our experiments. The source code for $TTSE$ was from [88].

Choosing application benchmarks: In order to compare performance among different spin-lock algorithms, the application benchmark chosen should have highly contended lock, which will noticeably promote efficient lock algorithms (cf. Performance Goals for Locks in [22]). Therefore, we chose as our application benchmarks the shared memory program using locks lmv from the Spark98 kernel [84] and the applications from the SPLASH-2 suite [109]: Volrend, which uses one lock, instead of an array of locks $QLock$, to protect a global queue, and Radiosity. Both Volrend and Radiosity have highly unstructured access patterns to irregular data structures [109]. The Radiosity application has a special feature different from the Spark98 and the Volrend: it has too many *small* locks besides some high contention locks. Therefore, the Radiosity is a "malicious" benchmark for sophisticated spin-lock algorithms like the new reactive spin-lock. The input data for the benchmarks were $sf5.1.pack$ for the Spark98, $head.den$ for the Volrend and *-room* option for the Radiosity, which are the largest data sets available for the Spark98 and the Volrend, and the recommended data set for the Radiosity.

Platforms used in the evaluation: The main system used for our experiments was a ccNUMA SGI Origin2000 with twenty eight 250MHz MIPS R10000 CPUs with 4MB L2 cache each. The system ran IRIX 6.5 and it was used exclusively. In the system, each thread ran exclusively on one processor. The system latencies of memory references are available in [64].

We also used as an evaluation platform a popular workstation with two Intel Xeon 3GHz CPUs with 1MB L2 cache each. The workstation ran Linux kernel 2.6.8. Since each Xeon processor with hyper-threading technology can concurrently execute two threads, the workstation can concurrently execute four threads without preemption. The system latencies of memory references are available in [15].

We compared our new reactive spin-lock with $TTSE$ and $TicketP$, both of which were *manually tuned* for each application benchmark on each platform. The tuned parameters for both are presented in Figure 5.6. Contention on the lock was varied by changing the number of participating processors/threads. The execution times of the application benchmarks were measured.

	Spark98	Volrend	Radiosity
$TTSE$/Origin2k	$b_e = 50000$ $l_e = 650000$	$b_e = 400$ $l_e = 1400$	$b_e = 200$ $l_e = 1200$
$TicketP$/Origin2k	$b_p = 100$	$b_p = 50$	$b_p = 130$
$TTSE$/Xeon	$b_e = 80$ $l_e = 700$	$b_e = 50$ $l_e = 350$	$b_e = 120$ $l_e = 1100$
$TicketP$/Xeon	$b_p = 60$	$b_p = 30$	$b_p = 90$

Figure 5.6: The table of manually tuned parameters for $TTSE$ and $TicketP$ in Spark98, Volrend and Radiosity applications on the SGI Origin2000 and the Intel Xeon workstation, where b_e, l_e are respectively $TTSE$'s *delay base* and *delay upper limit* for exponential backoff, and b_p is $TicketP$'s *delay base* for proportional backoff delays. The b_e, l_e and b_p are measured by the number of null-loops.

5.7.1 Results

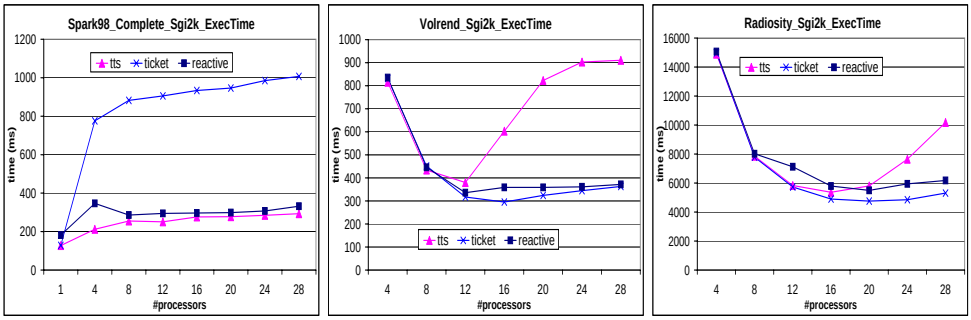

Figure 5.7: The execution time of Spark98, Volrend and Radiosity applications on the SGI Origin2000.

The new reactive spin-lock in Figure 5.5 involved in all locks with high contention[5]. Such locks play significant roles in application execution time and promote efficient spin-lock algorithms. Working on such high contention locks, processes always have to delay between two consecutive accesses. The new reactive spin-lock utilizes the delay interval to compute a reasonable value for the next delay. This is reason why even though the new reactive spin-lock appears quite heavy compared with the non-arbitrating/arbitrating representatives, it is actually efficient.

[5]The new reactive spin-lock algorithm does not involve in locks with low contention (cf. the last paragraph in subsection 5.6)

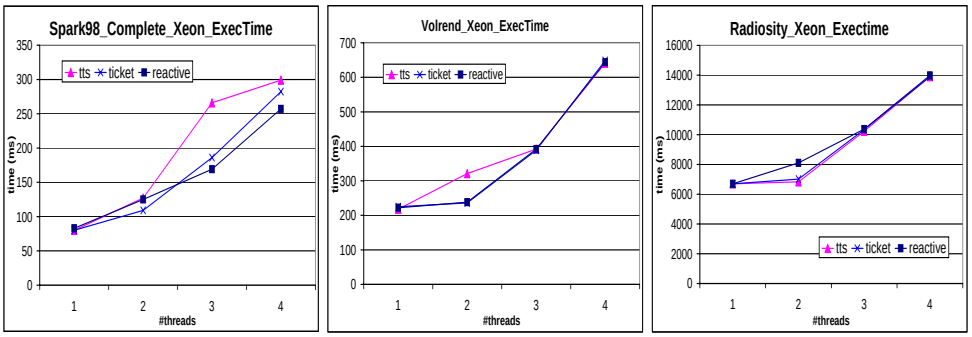

Figure 5.8: The execution time of Spark98, Volrend and Radiosity applications on a workstation with 2 Intel Xeon processors.

Figure 5.7 shows average execution times of applications Spark98, Volrend and Radiosity using $TTSE$ (tts), $TicketP$ ($ticket$) and the new reactive spin-lock ($reactive$) on the SGI platform. All the three charts show that the new reactive spin-lock approaches the best performances, which are the tts performance in the case of Spark98 and the $ticket$ performance in the cases of Volrend and Radiosity. Note that the new reactive algorithm *without tuning* performed similarly to the better of two representatives *with manual tuning* of non-arbitrating and arbitrating categories.

In the left chart on the Spark98 execution times, the reactive spin-lock approaches the best one, the $TTSE$. The reason why on the Spark98 $TTSE$ is better than $TicketP$ is as follows. In the Spark98, the $DoCS$ is not large and thus the Spark98 benchmark favors the spin-lock that exploits the *locality*, i.e. non-arbitrating spin-locks. With the large values $b_e = 50000$ and $l_e = 650000$ (cf. Figure 5.6), contention on the lock was kept low and the lock holder could re-acquire the lock and re-use the shared resource many times before the other processors retried to acquire the lock. This saved the time for transferring the lock as well as the shared data to another processor, the time for reading and writing data and the time for re-acquiring the lock because everything was cached locally. For the arbitrating spin-lock such as $TicketP$, all processors were in a waiting queue. Regardless of whether the distance between two consecutive processors in the waiting queue was too far, the lock and the shared data were transferred back and forth on the interconnect network, degrading the performance of $TicketP$ on the Spark98.

In the new spin-lock, the necessary backoff delay was computed reasonably by a competitive online algorithm that increased/decreased the backoff delay *just enough* to alleviate contention on the lock. That means the algorithm tried to kept changes as small as possible compared with the initial value. The initial value was

large due to the small delay outside the critical section (cf. Section 5.6). Since the new reactive spin-lock is a non-arbitrating spin-lock, it got benefit from exploiting the locality like $TTSE$.

In the middle chart on the Volrend execution times, the reactive spin-lock still approaches the best one, the $TicketP$. The reason why on the Volrend $TicketP$ is better than $TTSE$ is as follows. Since the high contention lock in the Volrend has large $DoCS$ and small $DiCS$, $TTSE$'s backoff delay had to be small to minimize the interval from the last lock release to the next lock acquisition. Therefore, the Volrend had $b_e = 400$ and $l_e = 1400$ (cf. Figure 5.6), which are too small compared with those in the Spark98. $TTSE$ spinning the lock with such a high frequency generated high contention on the lock, degrading performance of the whole system as mentioned in [4, 7, 55, 59, 77]. Therefore, the Volrend benchmark favored arbitrating locks such as $TicketP$, which reduced overhead due to the arbitration among processors and thus reduced contention on the lock.

However, the Volrend did not degrade the new reactive spin-lock performance, a non-arbitrating spin-lock. This is because the reactive spin-lock automatically and reasonably adjusted backoff delay $delay_i$ for each processor p_i according to contention on the lock, keeping contention on the lock low. On the other hand, the fact that the initial delay for each processor p_i was proportional to the $ticket$ that p_i obtained prevented partly processors from concurrently observing a free lock. These helped the new reactive spin-lock solve problems caused by high contention situation on the lock, which degraded the $TTSE$ performance.

Similar to the Volrend, the Radiosity benchmark shows that even applications with many *small* locks as Radiosity could not stop the reactive spin-lock algorithm from approaching the best performance, the $TicketP$ performance.

Experiments on the Intel platform showed a similar result: the new spin-lock performed as well as the best representative (cf. Figure 5.8). On this platform, performances of well-tuned $TTSE$ and $TicketP$ were similar for Volrend and Radiosity and were slightly different for Spark98. In the Spark98 benchmark, the new spin-lock still performed as the best. Although the benchmarks did not scale on the Intel platform, the result is still interesting since it shows how well the new spin-lock automatically tuned itself on different architectures compared with manually-tuned spin-locks.

In summary, the experiments on different platforms showed that the new reactive spin-lock without need of manually tuned parameters reacts well to contention variation as well as to a variety of applications. This helped the applications using the new reactive spin-lock approach the best performance gained by $TTSE$ and $TicketP$, the spin-locks that were *manually* tuned for both each application and each platform.

5.8 Conclusions

We have presented a new reactive spin-lock that is completely self-tuning, namely neither experimentally tuned thresholds nor probability distributions of inputs are required. The new spin-lock combines advantages of both arbitrating and non-arbitrating spin-locks. These features are achieved by a competitive algorithm for adjusting backoff delay reasonably to contention on the lock. Moreover, the new spin-lock also adapts itself to synchronization characteristics of applications to keep its good performance on different applications. Experimental results showed that the new spin-lock almost achieved the best performance on different platforms.

Chapter 6

Self-Tuning Reactive Diffracting Trees for Counting and Balancing[1]

Phuong Hoai Ha[2], Marina Papatriantafilou[2], Philippas Tsigas[2]

Abstract

Reactive diffracting trees *are an efficient distributed data structure that supports synchronization. They distribute a set of processes to smaller subsets that access different parts of memory in a global coordinated manner. They also adjust their size to attain good performance in the presence of different contention levels. However, their adjustment is sensitive to parameters that have to be manually tuned and determined after experimentation. Since these parameters depend on the application as well as on the system configuration, determining their optimal value is hard in practice. On the other hand, as the trees grow or shrink by only one level at a time, the cost of multi-level adjustments is high.*

This paper presents new reactive diffracting trees for counting and balancing without the need to fix parameters manually. The new trees balance the trade-off between the tree traversal latency and the latency due to contention at the tree nodes in an on-line manner. Moreover, the trees can grow or shrink by several levels in one adjustment step, improving their efficiency. Their efficiency is illustrated

[1]Expanded version of a preliminary result published in the Proceedings of the 8th International Conference on Principles of Distributed Systems (OPODIS '04), Dec. 2004, LNCS 3544, pp. 213-228, Springer-Verlag.

[2]Department of Computer Science and Engineering, Chalmers University of Technology, S-412 96 Gothenburg, Sweden. Email: {*phuong, ptrianta, tsigas*}*@cs.chalmers.se*.

via experiments on the SGI Origin2000, a well-known commercial ccNUMA multiprocessor, which compare the new trees with the traditional reactive diffracting trees. The experiments have showed that the new trees select the same tree depth, perform better and react faster than the traditional trees.

6.1 Introduction

It is hard to design distributed data structures for synchronization that perform efficiently over a wide range of contention conditions. Typically, simple centralized data structures are sufficient for low concurrency levels, for instance test-and-test-and-set locks with backoff [2, 7, 32, 58]; however, at higher levels of concurrency, sophisticated data structures are needed to reduce contention by distributing concurrent memory accesses to different banks, for instance queue-locks [7, 32, 77], combining trees [31, 111], counting pyramids [106, 107], combining funnels [95], counting networks [8, 9] and diffracting trees [93, 94]. Whereas queue-locks aim at reducing contention on locks generally, the other sophisticated data structures focus on specific problems such as counting and balancing in order to enhance efficiency. Combining trees [31, 111] implement low-contention `fetch-and-`Φ operations by combining requests along paths upward to their root and subsequently distributing results downward to their leaves. The idea has been developed to counting pyramids [106, 107] that allow nodes to randomly forward their requests to a node on the next higher level and also allow processors to select their initial level according to their request frequency. A similar idea has been used to develop combining funnels [95]. Opposite to the idea of combining requests, diffracting trees [93, 94] reduce contention for counting problems by distributing requests downward to the leaves, of which each works as a counter in a coordinated manner. The trees have been developed to elimination trees [91] that are suited for stack and pool constructions. Another approach for counting problems is counting networks [8, 9], which ensure low contention at each node. Linearizability [50] in counting network in general and with timing assumptions has been studied in [71, 76]. Empirical studies on the Proteus [17], a multiprocessor simulator, have showed that diffracting trees are superior to counting networks and combining trees under high loads [94].

Diffracting trees [93, 94] are well-known distributed data structures with the ability to distribute concurrent memory accesses to different memory banks in a coordinated manner. Each process(or) accessing the tree can be considered as leading a *token* that follows a path with mediate nodes from the root to a leaf. Each node receives tokens from its single input (coming from its parent node) and sends tokens to its outputs. The node is called *balancer* and acts as a *toggle mechanism* that, given a stream of input tokens, alternately forwards them to its outputs, from

left to right (i.e. send them to the left and right child nodes, respectively). The result is an even distribution of tokens at the leaves. In the trees, the contention at the root and balancers is alleviated using an *elimination technique* that evenly balances each pair of incoming tokens left and right without accessing the *toggle bit*. Diffracting trees have been introduced for *counting problems*, and hence their leaves are counters. The trees also satisfy the *step property*, which states that: when there are no tokens present inside the tree and if out_i denotes the number of tokens that have been output at leaf i, $0 \leq out_i - out_j \leq 1$ for any pair i and j of leaves such that $i < j$ (i.e. if one draws the tokens that have exited from each counter as a stack of boxes, the combined outcome will have the shape of a single step). Yet the fixed-size diffracting tree is optimal only for a small range of contention levels. To solve this problem, Della-Libera and Shavit proposed *reactive diffracting trees*, where nodes can shrink (to a counter) or grow (to subtrees with counters as leaves) according to their local load [25].

However, the reactive diffracting tree [25] uses a set of parameters to make its reactive decisions, namely folding/unfolding thresholds and the time interval for consecutive reactions. The parameter values depend on the multiprocessor system in use, the applications using the data structure and, in a multiprogramming environment, the system utilization by the other concurrent programs. The parameters must be manually tuned using experimentation and information that is not easily available (e.g. future load characteristics). Besides, the tree can shrink or grow by only one level at a time, making multi-level adjustments costly.

As we know, the main challenge in designing reactive objects in multiprocessor/ multiprogramming systems is to deal with unpredictable regular changes in execution environments. Therefore, reactive schemes using *fixed* parameters cannot be an optimal approach in *dynamic* environments such as multiprogramming systems. An ideal reactive object should not rely on experimentally tuned parameters and should react fast.

In this work we show that it is possible to construct such ideal reactive objects. In particular, we present a tree-type distributed data structure that has the same semantics as the reactive diffracting trees and moreover can react fast without the need of manual tuning. To circumvent the need of manually tuned parameters, we analyze the problem of balancing the trade-off between the two key measures, namely the contention level and the depth of the tree, as an online problem and subsequently develop an efficient on-line solution. The new reactive tree is also considerably faster than the reactive diffracting tree because of the low-overhead multilevel reaction: it can shrink and grow by many levels at a time without using expensive system clock reading. The new tree like the reactive diffracting tree is generally aimed at applications where such distributed data structures are needed. Since the latter were introduced in the context of counting problems, we use similar

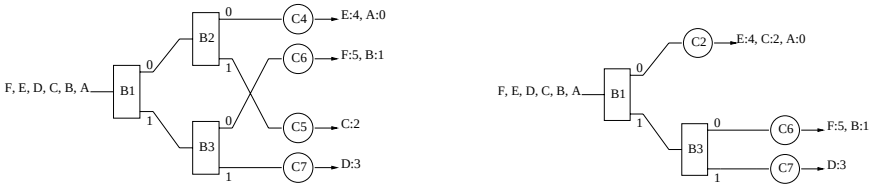

Figure 6.1: A diffracting tree (A) and a reactive diffracting tree (B).

terms in our description.

The rest of this paper is organized as follows. The next section provides basic background information about (reactive) diffracting trees. Section 6.3 presents the key idea and the algorithm of the self-tuning reactive tree. Section 6.4 describes the implementation of the tree. Section 6.5 shows the correctness of our algorithm. Section 6.6 presents an experimental evaluation of the self-tuning reactive trees, compared with the reactive diffracting trees, on the Origin2000 platform, and elaborate on a number of properties of our algorithm. Section 6.7 concludes this paper.

6.2 Background

In this section, we briefly describe (reactive) diffracting trees and present the fundamental concepts of online algorithms, which are used in the new self-tuning reactive trees.

6.2.1 Diffracting and Reactive-Diffracting Trees

Figure 6.1(A) depicts a diffracting tree. A set of processors $\{A, B, C, D, E, F\}$ is balanced on all leaves in which leaves $C4$ and $C6$ are accessed by only two processors, and leaves $C5$ and $C7$ by only one processor. Leaves $C4$, $C6$, $C5$ and $C7$ return values $f(4k)$, $f(4k+1)$, $f(4k+2)$ and $f(4k+3)$ respectively, where k is the number of processors that have visited the corresponding leaf and f is an arbitrary function, which may be costly. For counting problems, $f(k)$ simply returns k. Tokens passing one of these counters receive integer i, $i+4$, $i+2*4$, $\cdots$ where i is initial value of the counter. In the figure processors A, B, C, D, E and F receive integers $0, 1, 2, 3, 4$ and 5, respectively. Even though the processors access separate shared data (counters), they still receive numbers that form a consecutive sequence of integers as if they accessed a centralized counter.

Della-Libera and Shavit extended the trees to *reactive diffracting trees* where each counter can independently shrink or grow according to its local load in order to attain optimal performance [25]. Trees (A) and (B) in Figure 6.1 depict the folding action of a reactive diffracting tree. Assume at the beginning the reactive diffracting tree has a shape like tree (A). If the load on two counters $C4$ and $C5$ is small, the sub-tree whose root is $B2$ shrinks to counter $C2$ as depicted in tree (B). After that, if processors A, B, C, D, E and F sequentially traverse the tree (B), three processors A, C and E will visit counter $C2$. That is, the latency for processors to go from the root to the counter decreases whereas the load on each counter is still kept low.

6.2.2 Online Algorithms

Online problems are optimization problems where both the input is received online and the output is produced online so that the cost of processing the input is minimal or the outcome is best. If we know the whole input in advance, we may find an *optimal offline algorithm* OPT processing the whole input with minimal cost. In order to evaluate how good an online algorithm is, the concept of *competitive ratio* has been suggested.

Competitive ratio: An online algorithm ALG is considered competitive with a competitive ratio c (or *c-competitive*) if there exists a constant α so that for any finite input I [28]:

$$ALG(I) \leq c \cdot OPT(I) + \alpha \tag{6.1}$$

where $ALG(I)$ and $OPT(I)$ are the costs of the online algorithm ALG and the optimal offline algorithm OPT to service input I, respectively.

A common way to analyze an online algorithm is to consider a game between an *online player* and a malicious *adversary*. In this game, i) the online player applies the online algorithm on the input generated by the adversary and ii) the adversary with the knowledge of the online algorithm tries to generate the worst possible input whose processing cost is very expensive for the online algorithm but relatively inexpensive for the optimal offline algorithm.

The online algorithms together with the competitive analysis are a promising approach to resolve the problems where i) if we had some information about the future, we could find an optimal solution, and ii) it is impossible to obtain such kind of information.

6.3 Self-tuning reactive trees

6.3.1 Problem description

The problem we are interested in is to construct a tree-type data structure that satisfies the following requirements:

- It must evenly distribute a set of concurrent memory accesses to many small groups locally accessing shared data (counters at leaves) in a coordinated manner like (reactive) diffracting trees. It must guarantee the step-property.
- Moreover, it must automatically and efficiently adjust its size according to its load in order to achieve good performance over a wide range of load levels. It must not require any manually tuned parameters.

In order to satisfy these requirements, we have to tackle the following algorithmic problems:

- Design a dynamic mechanism that allows the tree to predict when and how much it should resize in order to obtain good performance while its load is changing unpredictably. Moreover, the overhead this mechanism introduces should not exceed the performance benefits the dynamic behavior gains.
- This dynamic mechanism should not only adjust the tree size to improve efficiency, but, more significantly, guarantee also the tree fundamental properties such as the step property.

6.3.2 Key ideas

The ideal reactive tree is the one in which each leaf is accessed by only one process(or) –or token[1]– at a time and the costs for a token to travel from the root to a leaf are kept to a minimum. However, these two latency-related factors are opposite to each other, i.e. if we want to decrease the contention at the leaves, we need to expand the tree and so the travel costs increases.

What we are looking for is a tree where the *overall overhead*, including the *latency due to contention* at the leaves and the *latency due to travel* from the root to a leaf, is minimum. The tree must also be able to cope with the problem of how to adjust the tree size so that this reaction does not become obsolete at the time when it takes effect. If the tree grows immediately whenever the contention level increases, it will pay high costs of travel, which will be subsequently wasted if right after that the contention level suddenly decreases. On the other hand, if the tree does not grow in time when the contention level increases, it may have to pay high costs due to high contention. If the tree knew in advance the contention

[1]The terms *processor, process* and *token* are used interchangeably throughout the paper.

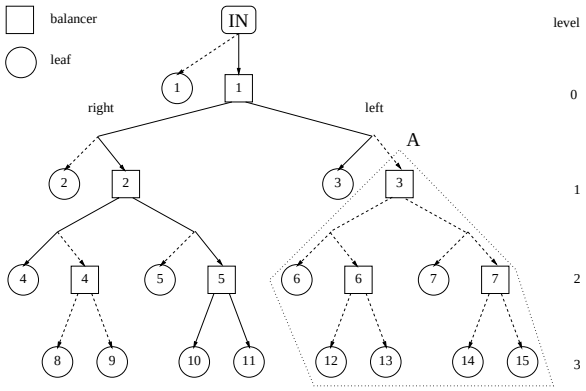

Figure 6.2: A self-tuning reactive tree

variation over its operative interval, it could adjust its size at each time point in such a way that the overall overhead is minimized. Since the contention level changes unpredictably, there is no way for the tree to collect such information.

To overcome this problem, we have designed a reactive algorithm based on the online techniques that have been used to solve the online currency trading problem [28].

Definition 6.3.1. *Let* surplus *denote the subtraction of the number of leaves from the number of processors that access the tree. The* surplus *represents the contention level on the tree since it is the surplus processors that cause contention on the leaves.*

Definition 6.3.2. *Let* latency *denote the latency due to travel from the root to a leaf.*

The challenge is to balance the trade-off between *surplus* and *latency*. Our solution for the problem is inspired by an optimal online algorithm called *threat-based algorithm* [28]. The algorithm is an optimal solution for the one-way trading problem, where the player must decide whether to accept the current exchange rate as well as how many of her dollars should be exchanged to yens at this rate without knowledge of how the exchange rate will vary in the future.

6.3.3 The tree structure

To adapt to contention variation efficiently, each leaf in the tree should be able to shrink and grow freely to any level suggested by the reactive scheme in one adjustment step. This motivates us to design a data structure for the tree in which

the adjustment time and the time in which processors are blocked due to an adjustment are kept to a minimum. Figure 6.2 illustrates the self-tuning reactive tree data structure. The squares in the figure are balancers and the circles are leaves. The numbers in the squares and circles are their labels. Each balancer has a *matching* leaf with the same label number. Symmetrically, each leaf that is not at the lowest level of the tree has a *matching* balancer. Each balancer has two outputs, $left$ and $right$, of which each is a pointer that can point to either a leaf or a balancer. A shrink or expand operation is essentially a switch of such a pointer (from a balancer to the matching leaf or vice-versa). Solid arrows in the figure represent the current pointer directions.

To traverse the tree, a processor p_i first visits the tree at its root IN and then follows the root pointer to visit balancer 1. When visiting a balancer, p_i switches the balancer toggle-bit to the other position (i.e. from left to right and vise-versa) and then follows the current direction of the corresponding pointer to visit the next node. If contention on the toggle-bit is high, the elimination technique [94] can be used here to alleviate the contention. When p_i visits a leaf L, before taking an appropriate counter value and exiting, it executes a *reactive scheme* with respect to the current load at L. The reactive scheme estimates which tree level is the best for the current load.

6.3.4 The reactive scheme

Definition 6.3.3. *A* load-rising (or load-dropping) transaction phase *is the longest sequence of subsequent visits at a leaf node with a monotonic non-decreasing (or non-increasing) estimated contention level over all the tree. A load-rising phase ends when a decrease in contention is observed; at that point a load-dropping phase begins.*

The trade-off between *surplus* and *latency* can be described as a game consisting of *load-rising* and *load-dropping transaction phases*. During a load-rising phase, a processor p_i visiting a leaf L may decide to expand the leaf to a subtree whose depth depends on the rising contention level. The depth is computed using the *threat-based online* method [28] with a principle: "expand *just enough* to guarantee a bounded competitive ratio even in the case that the contention level drops to the minimum in the next measurement step". The case of a load-dropping phase is symmetric: a reaction is to shrink a subtree to an appropriate level depending on the contention level. The results of the online method trigger the corresponding reaction.

If the recommended action is to grow to a level l_{lower}, i.e. the current load at the leaf L is too high and L should grow to a level l_{lower}, the processor p_i,

before exiting the tree, must help L carry out the growth task. The task consists of constructing the corresponding subtree (if it did not already exist) and switching the corresponding pointer from L to its matching balancer, which is the root of the subtree. For instance, assume that a processor p_i is visiting leaf 3 in Figure 6.2 and the recommendation is to grow to a sub-tree A with a depth of 3. The processor first constructs the sub-tree while other processors normally access the leaf 3 and exit the tree without any disturbance. After that, p_i locks leaf 3 in order to (i) switch the pointer to balancer 3 and (ii) assign proper values to counters $12, 13, 14$ and 15; then it releases leaf 3. At this point, the new coming processors following the left pointer of balancer 1 traverse the new sub-tree while the old processors that were already directed to leaf 3 continue to access leaf 3 and exit the tree. After completing the task, p_i increases the counter value of leaf 3 and exits the tree.

If the recommended action is to shrink to a level l_{higher}, i.e. the current load at the leaf L is too low and L should shrink to a higher level l_{higher} so as to reduce the travel latency, the pointer to the ancestral balancer of L at level l_{higher} must switch to the matching leaf whose counter value must be set properly. Let B denote that balancer. Since the sub-tree with B as a root contains other leaves that may have not decided to shrink to l_{higher}, an asynchronous vote-collecting scheme is needed. The leaf L votes for the level l_{higher} by adding its vote to the balancer B "vote box".

Definition 6.3.4. *The weight of a leaf's vote is the number of leaves at the lowest level in the subtree rooted at the matching balancer.*

For instance, in Figure 6.2 the weight of leaf 4 vote is 2 since the vote represent two leaves 8 and 9 at the lowest level.

The processor that helps L vote for B checks if there are enough votes for B. If more than half leaves of the subtree B vote to shrink to the matching leaf, the processor will execute the shrinkage task. It locks the matching leaf and the leaves of the sub-tree B in order to (i) collect their counter values, (ii) compute the proper counter value for the matching leaf and (iii) switch the pointer from B to its matching leaf. Note that all the leaves of subtree B need to be locked *only if* the load on the subtree is *so small that the subtree should shrink to a leaf.* Therefore, from the performance point of view locking the subtree in this case affects as locking a leaf in the traditional reactive diffracting tree.

For instance, assume that a processor p_i visits leaf 10 in Figure 6.2 and the recommendation is to shrink to level 1. Assume that leaf 4 has voted for balancer 2 too. The weight of leaf 4's vote is two since the vote represents leaves 8 and 9 at the lowest level. Leaf 10's vote has a weight of 1. Therefore, the sum of the vote weights at balancer 2 is 3. In this case, processor p_i helps balancer 2 execute the shrinkage task since three of four leaves at the subtree lowest level, leaves $8, 9$ and

10, have voted for the balancer. Then p_i locks leaf 2 and all the leaves of the sub-tree rooted at balancer 2, collects the counter values, computes the next counter value for leaf 2 and finally switches the pointer from balancer 2 to leaf 2. After that, all the leaves of the sub-tree are released immediately so that other processors can continue to access their counters. As soon as the counter at leaf 2 is assigned the new value, new processors going along the right pointer of balancer 1 access the new leaf and exit the tree while old processors continue traversing the old sub-tree. After completing the shrinkage task, the processor increases the counter value of leaf 10 and exits the tree.

Both the growing and shrinking processes support high parallelism: new coming processors follow the new sub-tree/leaf while pending processors continue traversing the old leaf/sub-tree.

6.3.5 Space needs of the tree

In a system with n processors, the algorithm needs $n-1$ balancer nodes and $2n-1$ leaf nodes. Although it seems that the self-tuning reactive tree requires more memory than the traditional reactive diffracting tree since the former introduces an auxiliary node (matching leaf) for each balancer of the tree, this is not the case. Actually, the former only splits the function of each node into two components: one is enabled when the node plays the role of a balancer and another is enabled when the node plays the role of a leaf (cf. Section 6.4.4 and Section 6.4.5). In other words, their memory requirements are similar.

6.4 Implementation

6.4.1 Preliminaries

Data structure and shared variables: Figure 6.3 describes the tree basic data structure and shared variables used in the implementation.

Synchronization primitives: The synchronization primitives used for the implementation are *test-and-set (TAS)*, *fetch-and-xor (FAX)* and *compare-and-swap (CAS)*. The definitions of the primitives are described in Figure 6.3, where x is a variable and v, old, new are values .The synchronization primitives used in the tree are comparable with those used in the traditional reactive diffracting tree, which are *test-and-set*, *swap* and *compare-and-swap* [25].

```
type NodeType =record Nid : [1..MaxNodeId]; kind : {BALANCER, LEAF}; mask: bit; end;
                /*in one word*/
     BalancerType = record state : {ACTIVE, OLD}; level : int; toggleBit : boolean;
                left, right : NodeType; end;
     LeafType = record state : {ACTIVE, OLD}; level, count, init : int;
                transPhase : {RISING, DROPPING}; end;
shared variables
     Balancers : array[0..MaxNodeId] of BalancerType; Leaves : array[1..MaxNodeId] of LeafType;
     TokenToReact : array[1..MaxNodeId] of boolean; Tracing : array[1..MaxProcs] of [1..MaxNodeId];

TAS(x) atomically{ oldx := x; x := 1; return oldx; } /* init: x := 0 */
FAX(x, v) atomically{ oldx := x; x := x xor v; return oldx; }
CAS(x, old, new) atomically{ oldx := x; if(x = old) then {x := new}; return oldx; }
```

Figure 6.3: The tree basic data structure and the synchronization primitives

6.4.2 Traversing self-tuning reactive trees

A processor Pid traverses the tree by calling function $TraverseTree()$ in Figure 6.4. First, it visits $Balancer[0]$ (the "IN" node in Figure 6.2), whose left child points to either $Balancers[1]$ or $Leaves[1]$. Before visiting a node on the next level, it updates its new location in $Tracing[Pid]$ (line T0). It records to its private variable $MyPath$ the path along which it traverses the tree, where $MyPath[i]$ is the node visited at level i. At each node, its behavior depends on the node type.

If the node is a balancer, the processor calls $TraverseB$ procedure to follow a proper child link (line T2 in $TraverseTree()$) and subsequently updates its new location to $Tracing[pid]$ (lines B1, B2 in $TraverseB()$). In $TraverseB$ procedure, the toggle mechanism for toggle-bits can be implemented using either advanced techniques like elimination techniques [94] to alleviate contention on the toggle-bits or low-contention hardware primitives like *fetchop* primitives in the SGI Origin2000 [64].

If the node is a leaf, in all cases the processor calls $TraverseL$ procedure to read and increase the leaf counter $L.count$ (line T8) and resets $Tracing[pid]$ to the value of $Root$. If exiting the tree through a leaf whose state is $ACTIVE$[2] , the processor must actively execute a reactive process. It acquires the leaf $TokenToReact$ and, upon succeeding, it invokes the $CheckCondition$ procedure. According to the procedure result the processor invokes either $Grow$ procedure to expand the leaf or $Elect2Shrink$ procedure to shrink the tree. The $Grow$ and $Elect2Shrink$ procedures are presented in subsections 6.4.4 and 6.4.5.

[2]Meaning that its state is not OLD; intuitively, old leaves (and old balancers) are those that a new processor traveling from the root cannot visit at that time.

```
int TraverseTree( int Pid)
T0 n := Assign(&Tracing[Pid], &Balancers[0].left);
   for(i := 1; ; i++) do
T1    MyPath[i] := n;
T2    if IsBalancer(n) then
        n := TraverseB(Balancers[n.Nid], Pid)
      else /*IsLeaf*/
T3      Leaves[n.Nid].contention++;
T4      if (Leaves[n.Nid].state = ACTIVE) and
          (TAS(TokenToReact[n.Nid]) = 0) then
          react := CheckCondition(Leaves[n.Nid]);
T5        if react = SHRINK then
            Elect2Shrink(n.Nid, MyPath);
T6        else if react = GROW then
            Grow(n.Nid);
T7        Reset(TokenToReact[n.Nid]);
T8      result := TraverseL(n.Nid);
T9      Leaves[n.Nid].contention--;
T10     Assign(&Tracing[Pid], &Balancers[0]);
        /*reset Tracing[Pid]*/
T11     return result;

NodeType TraverseB(BalancerType B, intPid)
B0 if ((k := Toggle(B.toggleBit)) = 0) then
B1    return (Assign(&Tracing[Pid], &B.right));
B2 else
      return (Assign(&Tracing[Pid], &B.left));

int TraverseL( int Nid)
L0  L := Leaves[Nid];
L1  AcquireLock(L.lock, Nid); /*lock the leaf*/
L2  result := L.count;
    L.count := L.count + 2^L.level;
L3  Release(L.lock); /*release the leaf*/
L4  return result;

int CheckCondition(LeafType L)
C0  Load := MIN(MaxProcs, L.cont * 2^L.level);
C1  FirstInPhase := False;
C2  if (L.transPhase = RISING) and
      (Load < L.load) then
      L.transPhase := DROPPING;
      FirstInPhase := True;
C3  else if (L.transPhase = DROPPING) and
      (Load > L.load) then
      L.transPhase := RISING;
      FirstInPhase := True;
C4  if L.transPhase = RISING then
      Surplus2Latency(L, Load, FirstInPhase);
C5  else
      Latency2Surplus(L, 1/Load, FirstInPhase);
C6  L.newLevel := log2(MaxProcs - L.surplus);
C7  if L.newLevel < L.level then return SHRINK;
C8  else if L.newLevel > L.level then return GROW;
C9  else return NONE;

Surplus2Latency(L, Load, FirstInPhase)
SL0 X := L.surplus; baseX := L.baseSurplus;
    Y := L.latency;
SL1 rXY := Load; LrXY := L.totLoadEst;
SL2 if FirstInPhase then
      if rXY > mXY * C then
      /* mXY: lower bound of rXY, C: comp. ratio*/
        ΔX := baseX * 1/C * (rXY - mXY*C)/(rXY - mXY);
SL3 else
      ΔX := baseX * 1/C * (rXY - LrXY)/(rXY - mXY);
SL4 L.surplus := L.surplus - ΔX;
SL5 L.latency := L.latency + ΔX * rXY;

Latency2Surplus(L, 1/Load, FirstInPhase)
/* symmetric to the above with: X := L.latency;
Y := L.surplus; */
```

Figure 6.4: The TraverseTree, TraverseB, TraverseL, CheckCondition, Surplus2Latency and Latency2Surplus procedures

6.4.3 Reaction conditions

Each leaf of the self-tuning reactive tree, via visiting processors, locally estimates which level is the best for the current load. A leaf L estimates the total load on the tree using the following formula (line C0 in $CheckCondition()$ in Figure 6.4):

$$Load = L.contention * 2^{L.level} \tag{6.2}$$

where $MaxProcs$ is the maximum number of processors potentially accessing the tree and $L.contention$, contention on the leaf, is the number of processors that are currently visiting the leaf. The value of $L.contention$ is increased by one every time a processor visits the leaf L (line $T3$ in Figure 6.4) and is decreased by one when a processor leaves the leaf (line $T9$ in Figure 6.4). At the beginning, the tree degenerate to a leaf. A processor considers to expand the root leaf to a tree only if it collides with other processors at the root leaf. Therefore, $2 \leq Load \leq MaxProcs$.

Since the tree is actually a leaf at the beginning, we have the following initial values:

$$\begin{aligned} surplus &= baseSurplus = MaxProcs - 1 \\ latency &= baseLatency = 0 \end{aligned}$$

Then, according to the contention variation on each leaf, the values of $surplus$ and $latency$ will be changed using an online algorithm. The number of surplus processors that the tree should have at that time is adjusted by $Surplus2Latency$ and $Latency2Surplus$ procedures in Figure 6.4. The surplus value is subsequently used to compute the number of leaves the tree should have and consequently the level the leaf L should shrink/grow to (line $C8$).

The $Surplus2Latency$ procedure exchanging $L.surplus$ to $L.latency$ is inspired by the *threat-based algorithm* [28] using $Load$ as exchange rate. In a load-rising transaction phase, the procedure complies with the following rules:

- The tree grows only when the load is the highest so far in the present transaction phase.
- When growing, it grows *just enough* to keep the competitive ratio $C = \varphi - \frac{\varphi-1}{\varphi^{1/(\varphi-1)}}$, where $\varphi = \frac{MaxProcs}{2}$, even if the load drops to the minimum possible in the next measurement.

The number of leaves the tree should have more is

$$\Delta Surplus = baseSurplus \cdot \frac{1}{C} \cdot \frac{Load - Load^{-}}{Load - 2}$$

where $Load^{-}$ is the highest load before the present measurement and $baseSurplus$ is the number of surplus processors at the beginning of the present transaction

phase (line SL3). Every time a new transaction phase starts, $baseSurplus$ is set to the last value of $surplus$ in the previous transaction phase. At the first exchange of a new transaction phase, where $Load^-$ has not been set yet, the value is replaced by the product of the competitive ratio and the minimum load (line $SL2$). Both variables $Load^-$ and $baseSurplus$ are stored in the current leaf where the reaction occurs.

Symmetrically, the $Latency2Surplus$ procedure computes how much the tree should shrink to reduce the travel latency when the load is decreasing. In the procedure, the exchange rate is the inverse of the load, $rXY = \frac{1}{Load}$ and is increasing. In this case, the value of $surplus$ increases and that of $latency$ decreases.

Lemma 6.4.1. *In each load-rising/load-dropping transaction phase, the reactive adjustment scheme is competitive with competitive ratio $C = \Theta(\ln P)$, where P is the number of processors potentially accessing the tree.*

Proof. The proof is similar to that of the threat-based policy in [28]. □

6.4.4 Expanding a leaf to a sub-tree

An expansion of a leaf L to a subtree T whose root is L's matching balancer B and depth is $L.newLevel - L.level$ essentially needs to set the counters at the new leaves in T to proper values so as to ensure the step property. Figure 6.7 illustrates the steps taken in $Grow$ procedure whose pseudo-code is in Figure 6.5. The expansion occurs only if there are no pending tokens in T (step G1). Otherwise, it will cause "old" tokens to get "new" values, which causes "holes" in the sequence of numbers received by all tokens in the end. As soon as the condition is satisfied, an expansion will be activated by subsequent tokens visiting L since L is under a high load. The process locks leaves in the new subtree (step G3) and sets proper values to their counters with respect to the step property (step G10). The values are computed on a consistent measurement of L's counter value and the number of pending processors in L (cf. Figure 6.6). Consistency is ensured by locking L (step G4) and switching the pointer from L to B (step G5) since the latter leaves a "non-interfered" set of processors in L. The lock acquisition is *conditional*, i.e. if some ancestor of L holds a lock, L's attempt to acquire the lock will return fail. In such a case, the operation aborts since a failure to acquire the lock means that there is an overlapping shrinkage operation being executed by an ancestor of L. (Note that overlapping growth operations by an ancestor of L must abort due to the existence of the token/processor at L (step G1), which is carrying out the growth process.) The leaf-locks are acquired in a *decreasing* order of node labels to avoid deadlock. Finally, the state of nodes in the new subtree is set to $ACTIVE$ to let new coming processors traverse them (step G9,G10).

```
GROW (int Nid) /*Leaves[Nid] becomes OLD;
     Balancers[Nid] and its subtree become ACTIVE*/
G0   L := Leaves[Nid]; B := Balancers[Nid];
G1   forall i, Read(Tracing[i])
       if ∃ pending processors in the subtree B then
         return; /*abort*/
G2   for each balancer B' in the subtree rooted at B,
       B'.toggleBit = 0;
G3   for each leaf L' at level L.newLevel of the
       subtree B, in decreasing order of nodeId do
       if not AcquireLock_cond(L'.lock, Nid) then
         Release all acquired locks; return; /*abort*/
G4   if (not AcquireLock_cond(L.lock, Nid))
       or (L.state = OLD) then
/*1st: an ancestor activated an overlapping Shrink*/
/*2nd:someone already made the expansion*/
       Release all acquired locks; return; /*abort*/
G5   Switch parent's pointer from L to B;
G6   forall i, Read(Tracing[i])
       ppL := #(pending processors at L);
/*Miss no processor since the new ones go to B*/
G7   CurCount := L.count; L.state := OLD;
G8   Release(L.lock);
G9   for each balancer B' as in step G2 do
       B'.state := ACTIVE;
G10  for each leaf L' as in step G3 do
       L'.count := NextCount(ppL, CurCount);
       L'.state := ACTIVE;
       Release(L'.lock);
     return;/*Success*/

ELECT2SHRINK( int Nid, NodeType MyPath[])
E0   L := Leaves[Nid];/*the leaf asks to shrink*/
     if L.oldLevel < L.newLevel then
/*new suggested level is lower than older suggestion*/
       for(i := L.oldLevel; i < L.newLevel; i++) do
E1       Balancers[MyPath[i].Nid].votes[Nid] := 0;
     else for (i := L.newLevel; i < L.oldLevel; i++) do
E2       B := Balancers[MyPath[i].Nid];
E3       B.votes[Nid] := 2^(MaxLevel−L.level);
E4       bWeight := 2^(MaxLevel−B.level);
         /*weight of B's subtree*/
E5       if (Σ_i B.votes[i]) / bWeight > 0.5 then Shrink(i); break;

SHRINK ( int Nid)/*Leaves[Nid] becomes ACTIVE;
     Balancers[Nid] and its subtree become OLD*/
S0   B := Balancers[Nid]; L := Leaves[Nid];
S1   forall i : Read(Tracing[i])
       if ∃ pending processor at L then return;/*abort*/
S2   if ( not AcquiredLock_cond(L.lock, Nid))
       or (B.state = OLD) then
       /*1st: some ancestor is performing Shrink*/
       /*2nd: someone already made the shrinkage*/
       Release possibly acquired lock; return; /*abort*/
S3   L.state := OLD; /*avoid reactive adjustment at L*/
S4   forall leaf L' in B's subtree, in increasing order
     of nodeId do
       AcquireLock_cond(L'.lock, Nid);
S5   Switch the parent's pointer from B to L
S6   forall i : Read(Tracing[i])
       eppB := #(effective processors in B's subtree;
       /*can't miss any since the new ones go to L*/
S7   for each balancer B' in the subtree rooted at B do
       B'.state := OLD;
     SL := ∅; SLCount := ∅;
S8   for each leaf L' in the subtree rooted at B do
       if (L.state = ACTIVE) then
         SL := ∪{L'};SLCount := ∪{L'.count};
         L'.state := OLD;
       Release(L'.lock);
S9   L.count := NextCount(eppB, SL, SLCount);
S10  L.state := ACTIVE;
S11  Release(L.lock);
```

Figure 6.5: The $Grow$, $Elect2Shrink$ and $Shrink$ procedures

Computing proper values for new leaf-counters: Since toggle-bits of balancers in the new subtree are reset to 0, the first processor traversing the new subtree will arrive at the right most leaf[3] at level $L.newLevel$. Therefore, the values for the counters at new leaves (line G10) are computed as in Figure 6.6, where L's current

---

$next\_count\_value := CurCount + ppL * 2^{L.level}$; $increment = next\_count\_value - MostRightLeaf.init$;
for *every leaf L' at level L.newLevel* **do** $L'.count = L'.init + increment$;

Figure 6.6: The $NextCount$ function in the $Grow$ procedure

counter value $CurCount$ and the number of pending processors ppL are read at lines G7 and G6 in the procedure, respectively. Local variable $next\_count\_value$ is the successive counter value after all the pending processors leave leaf L. Field $init$ in the leaf data structure (Figure 6.3) is the base to calculate counter values and is unchanged. For instance, in Figure 6.1 the base $init$ of counter $C6$ is 1 and the n^{th} counter value is $1 + n * 4$.

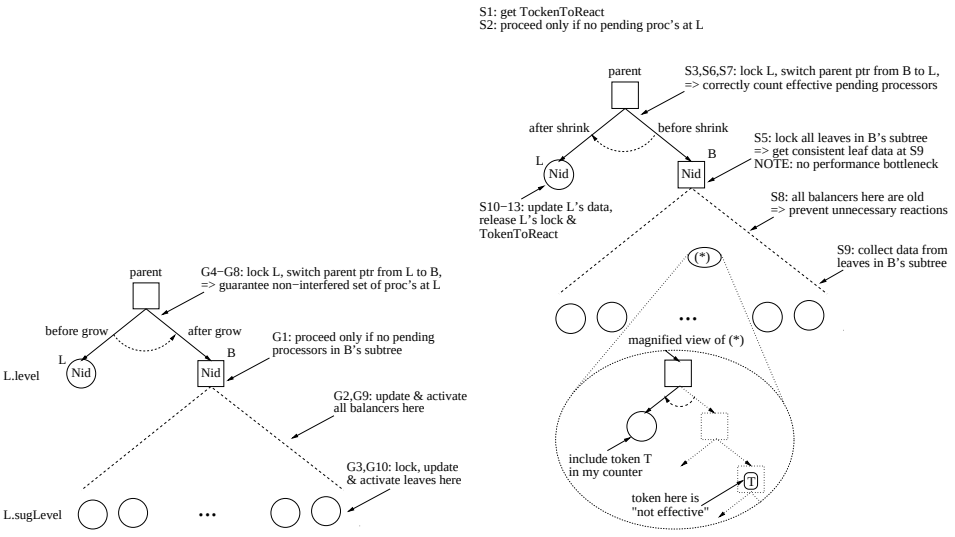

Figure 6.7: Illustrations for $Grow$ and $Shrink$ procedures

6.4.5 Shrinking a sub-tree to a leaf

A processor at a leaf L_0 with recommended reaction to shrink to level $L_0.newLevel$ adds L_0's vote to vote-boxes of balancers on its $MyPath$ from level $L_0.newLevel$

[3]The right side is shown in Figure 6.2.

down to level $L_0.level - 1$ (lines E2,E3 in Figure 6.5). It also removes the leaf old votes at levels above the new level (line E1). When reaching a balancer B with enough votes, the processor starts a shrinkage process at the balancer (line E5). The process is illustrated in Figure 6.7 and its pseudocode is in Figure 6.5

Symmetrically to the growth process, a shrinkage process of a subtree T rooted at balancer B to B's matching leaf L, essentially needs to set L's counter to a proper value so as to ensure the step property. The process occurs only if there are no pending tokens in L. Otherwise, it will cause "old" tokens get "new" values (step S1 in $Shrink$). The process lock L (step S2) and sets a proper value to its counter with respect to the step property (step S9). The value for L's counter is computed on a consistent measurement of the number of pending processors in T and the counter values of each leaf L' in T (cf. Figure 6.8). Consistency is ensured by locking leaves L' in T (step S4) and switching the pointer from B to L since the latter leaves a "non-interfered" set of processors in T. Similarly to the $Grow$ procedure, the lock acquisition is conditional. The process locks leaves in T in an *increasing* order of node labels to avoid deadlock. Note that an overlapping shrinkage process by L's ancestors cannot cause any of the attempts to lock L' to fail since the overlapping process must already lock L (and if it had succeeded, it would have caused the shrinkage from B to L to abort earlier, at step S2 in $Shrink$). Finally, the shrinkage process sets state of balancers and leaves in T to OLD (steps S7,S8 in $Shrink$).

Computing a proper value for the new leaf-counter: The counter value for the new leaf is computed on the set of active leaves SL, their counter values $SLCount$ and the number of effective pending processors $eppB$ in the subtree B (line S9). The value is the result of the $NextCount$ function, which is implemented as in Figure 6.8.

---

int NEXTCOUNT (**int** $eppB, list\_t\ SL, list\_t\ SLCount$)
Convert leaves in B to the same level, the lowest level ⇒ new sets of leaves SL' & counter values SLCount';
Distribute the number of effective processors $eppB$ on the leaves in SL' so that step-property is satisfied.;
Call the leaf last visited by the pending processors $lastL$;
return $(lastL.count - 2^{lastL.level} + 2^{B.level})$;

Figure 6.8: The $NextCount$ function in $Shrink$ procedure.

For instance, in Figure 6.2 if the subtree of balancer 2 shrinks to leaf 2 and the set of active leaves SL is $\{4, 10, 11\}$, leaf 4 needs to be converted to two leaves 8 and 9 at the same level with leaves 10 and 11, the lowest level. Thus, the new set of leaves SL' is $8, 9, 10, 11$. After converting, the subtree becomes balanced and the step-property must be satisfied on the subtree. The following feature of trees satisfying the step-property was exploited to distribute the set of effective

processors $eppB$ among leaves in SL'. Call the highest counter value of leaves at the lowest level $MaxValue$. The counter values of leaves must be in range $(MaxValue - 2^{LowestLevel}, MaxValue]$.

6.4.6 Efficiency enhancement

In order to improve the efficiency of the new tree, we define and implement two advanced synchronization operations: *read-and-follow-link* and *conditional lock acquisition*. The former is a lock-free operation that maximizes parallelism at balancers. The latter is to minimize disturbance due to reactive adjustments for the processors. The operations are described in this subsection and their pseudo-code is in Figure 6.9.

```
BASICASSIGN(NodeType * trace_i, NodeType * child)
A0  *trace_i := child;/*mark trace_i, clear mask-bit*/
A1  temp := *child; /*get the expected value*/
A2  temp.mask := 1; /*set the mask-bit*/
A3  CAS(trace_i, child, temp);

NodeType ASSIGN(NodeType * trace_i, NodeType * child)
AR0 BasicAssign(trace_i, child);
AR1 return Read(trace_i);

boolean ACQUIRELOCK_COND( int lock, int Nid)
AL0 while ((CurOccId := CAS(lock, 0, Nid)) ≠ 0) do
AL1     if IsParent(CurOccId, Nid) then return Fail;
AL2     Delay using exponential backoff;
AL3 return Success;

NodeType READ(NodeType * trace_i)
R0  do
R1      local := *trace_i;
R2      if local.mask = 0 then
        /*trace_i is marked*/
R3          temp := *local; /*help Assign() ...*/
R4          temp.mask := 1;
R5          CAS(trace_i, local, temp);
R6  while(local.mask = 0);
    /*... until it completes*/
R7  return local;

RELEASE( int lock)
    lock := 0;
```

Figure 6.9: The BasicAssign, Assign, Read, and AcquireLock_cond operations

The read-and-follow-link operation: In order to support high parallelism, we apply a non-blocking synchronization technique, instead of mutual exclusion, for balancers, where the collision among visiting processors is high. The technique allows all processors to concurrently pass a balancer, helping the tree achieve better performance. However, this may make the number of pending processors be counted incorrectly, which consequently causes the tree output to not satisfy the step-property.

Figure 6.10 illustrates the problem of incorrectly counting the number of pending processors. A processor p_2 reads a pointer ptr that shows which node the processor is going to visit. Before the processor updates its new location in $Tracing[2]$, another processor p_1 executes a reactive adjustment that switches the pointer to another position and counts the number of pending processors in the old branch via

$Tracing$ array. Since p_1 reads a obsolete value of $Tracing[2]$, it does not count p_2. Then, p_2 follows the old value of ptr and visit the old branch. That means the number of pending processors in the old branch is higher than what p_1 has counted, leading to compute incorrect values for the new leaf-counters in the $Grow$ and $Shrink$ procedures (step G10 and S9 in Figure 6.5).

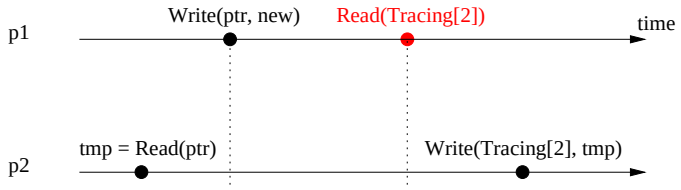

Figure 6.10: An illustration for the need of *read-and-follow-link* operation

Intuitively, if both $Read(ptr)$ and $Write(Tracing[2])$ on p_2 occur atomically to $Read(Tracing[2])$ on p_1, the problem will be solved. This motivates us to implement advanced operations called $BasicAssign$ and $Read$(cf. Figure 6.9). Each element $Tracing[i]$ can be updated by only one processor p_i via the $BasicAssign$ operation and can be read by many other processors via the $Read$ operation. The former reads the variable pointed by $child$ and then writes its value to the variable pointed by $trace_i$. The latter reads the variable at $trace_i$. The $BasicAssign()$ operation is atomic to $Read()$ operation[4] and only the operations can access array $Tracing$. The two operations are lock-free [46] and thus they improve parallelism and performance of the tree.

Conditional lock-acquisition operation: In order to minimize adjustment delay for processors, we need to minimize locking intervals for *working* leaves: the $Grow$ procedure acquires necessary leaves in *decreasing* order of their labels so as to acquire the working leaf L at latest, and the $Shrink$ procedure acquires necessary leaves in *increasing* order of their labels. Moreover, since the tree allow concurrent adjustments at any level, adjustments at high levels should have higher priorities than ones at lower levels when there is collision among adjustments. However, this optimization may lead to deadlock due to interferences between growth and shrikage processes. We solve this problem by designing an advanced operation called *conditional lock acquisition*.

During both the growth and shrinkage processes, a processor on behalf of a node Nid must invoke $AcquireLock\_cond$ to acquire a leaf lock $lock$ by writing the node label Nid to the lock. If the lock is occupied by an ancestor of the node, the procedure returns $Fail$ (line AL1, Figure 6.9). The procedure is built on the

[4]The proof is given in Lemma 6.5.1

busy-waiting with exponential backoff technique instead of queue-lock because the contention at the leaf of the tree is kept low. In a low-contention environment, the busy-waiting with exponential back-off technique achieves better performance than the queue-lock, which requires more complicated data structures [77].

For acquiring leaf locks to increase counter values (i.e. not perform adjustment tasks), processors invoke $AcquireLock$ procedure without condition. The procedure is similar to $AcquireLock\_cond$ but does not check the ancestor-condition (line AL1, Figure 6.9). Procedure $AcquireLock$ always returns $Succeed$.

The way these locking mechanisms interact and ensure safety and liveness for the tree accesses is explained in descriptions of the $Grow$ and $Shrink$ procedures and is proven in Section 6.5.

6.5 Correctness Proof

We first prove that the $BasicAssign$ operation is atomic to the $Read$ operation (cf. Figure 6.9) as mentioned in subsection 6.4.6.

Lemma 6.5.1. *The* BasicAssign *operation is atomic to the* Read *operation.*

Proof. In the $BasiceAssign$ operation, the variable pointed by $trace_i$ is first locked by writing a pointer $child$ to it (line A0, Figure 6.9). The last bit in a pointer (*mask* bit in $NodeType$, Figure 6.3), which is unused because of word-alignment memory architecture, is exploited to determine if a variable is locked or not. If the last bit of the $*trace_i$ variable is zero, the variable is locked and its value is the address of another variable whose value must be written to $*trace_i$. After reading the expected value $*child$, the operation sets the last bit to 1 so as to unlock $*trace_i$ and subsequently writes the value to $*trace_i$ using a *compare_and_swap* primitive (lines A1-A3). If $*trace_i$ still contains the pointer $child$, the primitive will successfully write the value to $*trace_i$. Otherwise, another operation has helped the operation complete the assignment.

When reading the value of a variable $*trace_i$, the $Read$ operation checks if the variable is locked (line R2). If the last bit of the variable is zero, the operation will help the corresponding $BasicAssign$ operation to write the expected value to the variable before trying to read it again (lines R3-R5). Therefore, the $BasicAssign$ operation is atomic to the $Read$ operation. The linearization point [50] of the $Read$ operation is the point it reads a value with a non-zero last bit (line R1); the linearization point of the $BasicAssign()$ operation is the point it writes a pointer with a zero last bit to $*trace_i$ (line A0). □

Second, since leaves are locked in decreasing order of leaf identities in the $Grow$ procedure but in increasing order in the $Shrink$ procedure, we need to

prove that deadlock never occurs due to interferences between growth and shrinkage processes.

Lemma 6.5.2. *Self-tuning reactive trees are deadlock-free.*

Proof. Interferences between two balancers that are trying to lock leaves[5] occur only if one of the balancers is the other's ancestor in the *family* tree. Assume that there are two balancers b_i and b_j, where b_i is b_j's ancestor.

- *Case 1:* If both balancers b_i and b_j execute shrinkage processes that shrink their sub-trees to leaves, both will lock leaves in increasing order of leaf identities by using the $AcquireLock\_cond$ operation. If the leaf with smallest identity that b_j needs is locked by b_i, the operation called by b_j will return $Fail$ immediately. This is because the leaf is locked by an ancestor of b_j. If the leaf is locked by b_j, b_i must wait at the leaf until b_j completes its own work and then b_i continues locking necessary leaves. If no processor locking the leaves on behalf of a balancer crashes, no deadlock will occur.

- *Case 2:* b_i executes a shrinkage process, which shrinks its subtree to a leaf, and b_j executes a growth process, which expands its matching leaf to a subtree. In this case, b_i tries to lock all necessary leaves in increasing order of leaf identities and b_j does that in decreasing order of leaf identities. Assume that b_i locked leaf k successfully and is now trying to lock leaf $(k+1)$ whereas b_j locked leaf $(k+1)$ successfully and is trying to lock leaf k. Because b_i and b_j use the procedure $AcquireLock\_cond$ that conditionally acquires the locks, b_j will fail to lock leaf k, which is locked by its ancestor, and will release all the leaves it has locked so far (lines G3 and G4, Figure 6.5). Eventually, b_i successfully locks leaf $k+1$ and continues locking other necessary leaves. That is, deadlock does not occur in this case either.

Note that there is no the case that b_i executes a growth process. This is because in its subtree there is at least one pending processor that helps b_j execute its adjustment process. □

Corollary 6.5.3. *In the shrinkage process, if the corresponding balancer successfully locked the necessary leaf with smallest identity, it will successfully lock all the leaves it needs.*

Therefore, the $Shrink$ procedure returns $Fail$ only if the matching leaf of the corresponding balancer is locked by an ancestor of the balancer.

[5] recall that processors lock leaves on behalf of balancers

Lemma 6.5.4. *There is no interference between any two growth processes in the self-tuning reactive tree.*

Proof. Similarly as in the proof of the previous lemma: i) the interference between two balancers who are trying to lock leaves occurs only if one of the balancers is the other's ancestor in the *family* tree and ii) there is no case that two expansion phases are executed at the same time and one of the two corresponding balancers is an ancestor of the other. □

Lemma 6.5.4 explains why the $Grow$ procedure does not lock balancers before resetting their variables (line G2, Figure 6.5).

Finally, we prove that the number of processors used to calculate counter value for new leaves in both $Grow$ and $Shrink$ procedures is counted accurately via the global array $Tracing$.

Definition 6.5.5. Old balancers/leaves are the balancers/leaves whose states are OLD

Definition 6.5.6. Effective processors/tokens *are processors/tokens that are not in old balancers nor in old leaves of a locked sub-tree.*

Only the *effective processors* affect the next counter values calculated for new leaves. An illustration to enhance the understanding of this definition is given in Figure 6.7-$Shrink$, where the token marked as "T" at the lower part of the figure is not effective.

Lemma 6.5.7. *The number of* effective processors *that are pending in a locked sub-tree or a locked leaf is counted accurately in the $Grow$ and $Shrink$ procedures.*

Proof. A processor p_i executing an adjustment process switches a pointer from one branch of the tree to the other before counting pending processors in the old branch (lines G5, G6 in $Grow$ and lines S5, S6 in $Shrink$, Figure 6.5). Since the $BasiceAssign()$ operation is atomic to the $Read$ operation (by Lemma 6.5.1), the processor counts the number of pending processors accurately. Recall that the old branch is locked as a whole so that no processor can leave the tree from the old branch as well as no other adjustment can concurrently take place in the old branch until the counting completes. The pending processors counted include *effective* processors and *ineffective* processors.

In the case of tree expansion, the number of pending processors in the locked leaf is the number of effective processors.

In the case of tree shrinkage, we lock both old and active leaves of the locked sub-tree so that no pending processor in the sub-tree can switch any pointers. Recall that to switch a pointer, a processor has to successfully lock the leaf corresponding to that pointer. On the other hand, (i) we set states of all balancers and leaves in a locked sub-tree/locked leaf to Old before releasing them (lines S7, S8 in $Shrink$ and line G7 in $Grow$), and (ii) after locking all necessary balancers and leaves, processors continue processing the corresponding shrinkage/growth processes only if the switching balancers/leaves are still in an active state (line G4 in $Grow$ and line S2 in $Shrink$). Therefore, a pending processor in the locked sub-tree that visited an Old balancer or an Old leaf will never visit an $Active$ one in this locked sub-tree. Similarly, a pending processor in the locked sub-tree that visited an $Active$ balancer or an $Active$ leaf will never visit an Old one in this locked sub-tree. Hence, by checking the state of the node that a pending processor p_j in the locked sub-tree is currently visiting, we can know whether the processor p_j is effective (line S6 in $Shrink$). In conclusion, the number of effective processors in a locked sub-tree is counted accurately in $Shrink$ procedure in the case of tree shrinkage. □

Since the number of effective pending processors is counted accurately (by Lemma 6.5.7), the counter values that are set at leaves after adjustment steps in the self-tuning trees are correct, i.e. the step-property is guaranteed. That means the requirements mentioned in subsection 6.3.1 are satisfied. This implies the following theorem:

Theorem 6.5.8. *The self-tuning tree obtains the following properties:*

- *Evenly distribute a set of concurrent memory accesses to different banks in a coordinated manner like the (reactive) diffracting tree. The step-property is guaranteed.*
- *Automatically and efficiently adjust its size according to its load in order to gain performance. No manually tuned parameters are needed.*

6.6 Evaluation

In this section, we evaluate the performance of the self-tuning reactive tree. We implemented two versions of the tree: one called *ST-Tree(P)* uses the elimination technique [93, 94] to alleviate contention at toggle-bits as the traditional reactive diffracting tree does, and the other called *ST-Tree* uses a low-contention hardware primitive $fetchop$ supported by the SGI Origin2000 [64] instead. It is interesting to see how much the new tree can speed up if the system supports low-contention synchronization primitives.

We used the traditional reactive diffracting tree [25] as a basis of comparison since they are the most efficient reactive counting constructions in the literature. The most difficult issue in implementing the tree is to find the best folding and unfolding thresholds as well as the number of consecutive timings called *UNFOLDING_LIMIT, FOLDING_LIMIT* and *MINIMUM_HITS* in [25]. Subsection *Load Surge Benchmark* in [25] has described that the tree sized to a depth 3 tree when the authors ran the index-distribution benchmark [94] with 32 processors in the highest possible load ($work = 0$) and the number of consecutive timings was set to 10. Following the description, we ran our implementation of the tree on the ccNUMA Origin2000 with thirty 250MHz MIPS R10000 processors. The result is that folding and unfolding thresholds are 3 and 10 microseconds, respectively. This selection of parameters did not only keep our experiments consistent with the ones presented in [25] but also gave the best performance for the reactive diffracting tree in our system. Regarding the prism size, an algorithmic construct to implement the elimination technique, each node has $c2^{(d-l)}$ prism locations, where $c = 0.5$, d is the average value of the tree depths estimated by processors traversing the tree and l is the level of the node [25, 93]. The upper bound for adaptive spin *MAXSPIN* is 128 as mentioned in [94].

We used the full-contention benchmark and the surge-load benchmark that are similar to the index-distribution benchmark with $work = 0$ and the surge-load benchmark in [25]. The benchmarks ran on a ccNUMA SGI Origin 2000 with thirty 250MHz MIPS R10000 processors. The system ran IRIX 6.5. In order to make these empirical results accessible to other researchers and practitioners, C code for the tested algorithms is available at `http://www.cs.chalmers.se/~phuong/satNov05.tar.gz`.

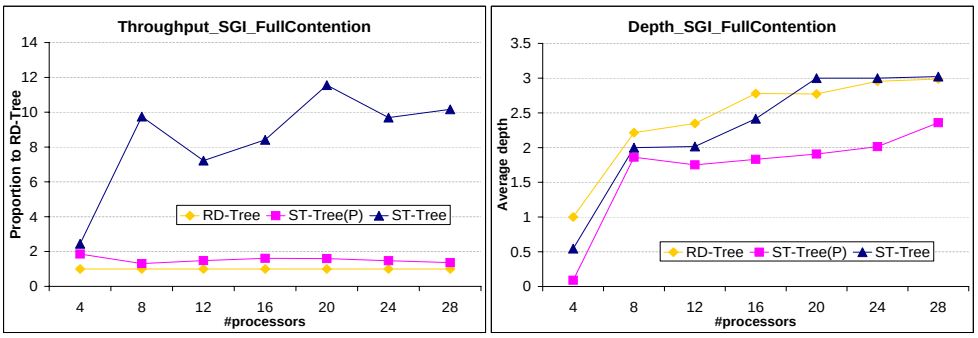

Figure 6.11: Throughput and average depth of trees in the full-contention benchmark on SGI Origin2000.

6.6.1 Full contention benchmark

In the benchmark, each processor continuously traverses the respective tree and gets a counter value. The benchmark was run for the different numbers of processors from 4 to 28, which simulates different loads on trees. We ran the benchmark for one minute and measured the average size of trees and the average number of traversing operations (or the number of tokens passing a tree) in one second. The results are shown in Figure 6.11, where the right charts show the tree average depths and the left charts show the proportion of the ST-tree throughput to that of the RD-tree. The tree with higher throughput is the better.

The right shows that both ST-Tree(P) and ST-Tree perform better than RD-Tree. In the case of 28 processors, the ST-Tree(P), which uses the same elimination technique as RD-Tree, is 36% faster than RD-Tree and ST-Tree is 10 times faster. Since each tree *continuously* adjusts its current size around the average value due to load variation on its leaves even in the case that the number of participating processors is fixed (cf. Figure 6.12), a tree with more efficient adjustment will achieve better performance in the full-contention benchmark. The reactive adjustments of the ST-Tree(P)/ST-Tree and RD-Tree have algorithmic differences:

- The former reacts to load variation faster with lower overhead as described in Section 6.3.3, whereas in the latter leaves shrink or grow only one level in one reaction step and then have to wait for a given number of processors traversing themselves before shrinking or growing again.
- In the latter, whenever a leaf shrinks or grows, all processors visiting the leaf are blocked completely until the reaction process completes. Moreover, some processors may be forced to go back to higher nodes many times before exiting the tree. In the former, this problem is avoided with the introduction of matching leaves, which provide high parallelism.

Another interesting result is that when the load on trees increases, the ST-Tree automatically adjusts its size close to that of the RD-tree that requires three experimental parameters for each specific system. Since the RD-Tree throughput is lower than the ST-Tree throughput, the result implies that the longer lock-based adjustments at leaves of the RD-Tree block more processors at leaves, causing loads on them as high as those on the ST-Tree leaves. On the other hand, the size difference between ST-Tree and ST-Tree(P) implies that the elimination technique [93,94] not only alleviates contention on toggle-bits but also delays processors at balancers, consequently reducing loads on leaves. However, the low loads at leaves are at the cost of low throughput.

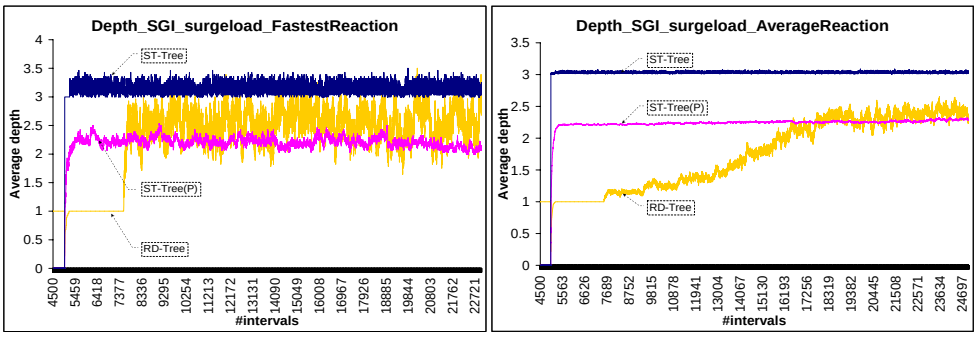

Figure 6.12: Average depths of trees in the surge load benchmark on SGI Origin2000, the fastest and the average reactions.

6.6.2 Surge load benchmark

The benchmark shows how fast the trees react to load variation. In this benchmark, we measured the average depth of each tree in each interval of 400 microseconds. The measurement was done by a monitor processor. At interval 5000, the number of processors was changed from 4 to 28. The depth of the trees at the interval 5001 was measured after synchronizing the monitor processor with all the new processors, i.e. the period between the end of interval 5000 and the beginning of interval 5001 was not 400 microseconds.

Figure 6.12 shows depths of the trees from interval 4500 to interval 25000. The left chart shows the fastest reaction experiment for each tree over 15 experiments. It also shows the amplitude in which the tree sizes vary when the number of processors is fixed to 28. The RD-Tree size amplitude is showed to be the largest. The right chart shows the average reaction time figures for the trees over 15 experiments. In the case of 28 processors, the ST-tree reached a depth 3 at interval 5009, i.e only after 9 intervals since the time all 28 processors started to run. The ST-Tree(P) reached a depth 2.2 at interval 5362 and the RD-tree reached level 2.4 at interval 17770. The difference between the average reaction times of ST-Tree and ST-Tree(P) implies that the elimination technique delays processors at balancers when the load surges, making loads at leaves increase gradually. The difference between ST-Tree(P) and RD-Tree reaction delays re-confirms the advantage of the fast multi-level adjustment scheme used in ST-Tree/ST-Tree(P).

Moreover, in the surge-load benchmark, it is interesting to see not only how fast the trees adjust their size but also how efficient they are with respect to throughput. We extended the benchmark so that the number of processors accessing the trees changes from 4 to 28 or from 28 to 4 for each period of 0.1 second. The benchmark was run in 1 minute or in 600 cycles. We measured the average number of travers-

ing operations, i.e. the number of tokens passing a tree, in one second. The result is showed in Figure 6.13. As expected, the ratio of ST-Tree/ST-Tree(P) throughput to RD-Tree throughput in the benchmark is higher than in the full-load benchmark in which loads on leaves vary slightly due to the number of processors fixed. In the surge-load benchmark, the ST-Tree(P) throughput is 2.4 times higher than that of RD-Tree and ST-Tree throughput is 15.3 times higher.

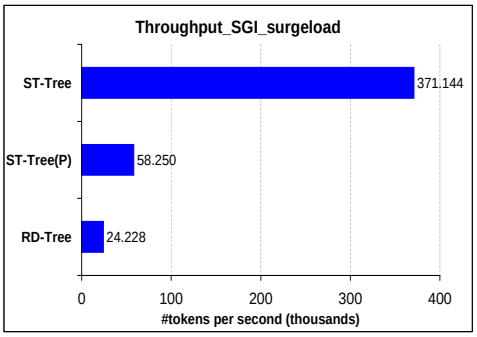

Figure 6.13: Throughput of trees in the surge load benchmark on SGI Origin2000.

6.7 Conclusion

This paper has presented the self-tuning reactive tree, a data structure that distributes concurrent memory accesses to different banks in a coordinated manner. The tree extends the reactive diffracting tree, a successful result in the area of reactive concurrent data structures, in many aspects. The new tree is *completely* reactive: its reactive adjustment does not need any tuned parameters. To circumvent the need of manually tuned parameters, the trade-off between tree depth and loads at leaves is analyzed as an online problem and subsequently an efficient online solution is suggested. The solution is inspired by an optimal online algorithm for an online financial problem [28], which helps the tree make precise reactive decisions. The precise decisions contribute a significant factor to the tree efficiency. Another considerable factor to the tree efficiency is the new construction that allows the tree to freely grow and shrink by several levels in just one adjustment step. The new construction is designed to reduce overhead due to expensive system calls (e.g. timing calls) and adjustment delays (e.g. delays of locking tokens or moving tokens upward and downward) in the traditional reactive diffracting tree. Moreover, the new construction has space complexity comparable with that of the traditional reactive diffracting trees. It also exploits low-contention occasions on subtrees to make its locking process as efficient as in the traditional reactive diffracting trees

although its locking process locks more nodes at the same time. As a result, the new tree can react quickly to load variations, and at the same time offers good latency to the traversing processors and good scalability behavior.

We have also presented an experimental evaluation of the new tree on the SGI Origin2000, a well-known commercial ccNUMA multiprocessor. We think that it is of interest to evaluate the tree performance on real multiprocessor systems that are widely used in practice.

In the near future, we plan to look into new reactive schemes that may further improve the performance of reactive shared objects. Ideally, reactive shared objects should be able to observe the changes in execution environments and react accordingly in time. In unpredictable environments such as multiprocessor/multiprogramming systems, online algorithms and competitive analysis seem to be a promising approach for designing such reactive objects. In the approach, choosing appropriate adversary models may allow faster reaction and better execution time.

Chapter 7

Competitive Freshness Algorithms for Wait-free Data Objects[1]

Peter Damaschke[2], Phuong Hoai Ha[2], Philippas Tsigas[2]

Abstract

Wait-free concurrent data objects are widely used in multiprocessor systems and real-time systems. Their popularity results from the fact that they avoid locking and that concurrent operations on such data objects are guaranteed to finish in a bounded number of steps regardless of the other operations interference. The data objects allow high access parallelism and guarantee correctness of the concurrent access with respect to its semantics. In such a highly-concurrent environment, where many wait-free write-operations updating the object state can overlap a single read-operation, the age/freshness of the state returned by this read-operation is a significant measure of the object quality, especially for real-time systems.

In this paper, we first propose a freshness measure for wait-free concurrent data objects. Subsequently, we model the freshness problem as an online problem and present two algorithms for the problem. The first one is a deterministic algorithm with freshness competitive ratio $\sqrt{\alpha}$, where α is a function of execution-time upper-bound of wait-free operations. Moreover, we prove that $\sqrt{\alpha}$ is asymptotically the

[1]Expanded version of a preliminary result published in Technical report no. 2005-17, Department of Computer Science and Engineering, Chalmers University of Technology, Gothenburg, Sweden, Oct. 2005.

[2]Department of Computer Science and Engineering, Chalmers University of Technology, S-412 96 Gothenburg, Sweden. Email:{*ptr, phuong, tsigas*}*@cs.chalmers.se*

optimal freshness competitive ratio for deterministic algorithms, implying that the first algorithm is optimal. The second algorithm is a competitive randomized algorithm with freshness competitive ratio $\frac{\ln \alpha}{1+\ln 2-\frac{2}{\sqrt{\alpha}}}$.

7.1 Introduction

Concurrent data objects play a significant role in multiprocessor systems, but also create challenges on consistency. In concurrent environments like multiprocessor systems, consistency of a shared data object is guaranteed mostly by mutual exclusion, a form of locking. However, mutual exclusion degrades the system's overall performance due to lock convoying, i.e. other concurrent operations cannot make any progress while the access to the shared object is blocked. Mutual exclusion also contains risks of deadlock and priority inversion. 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 the problems that blocking can cause. Non-blocking algorithms are either lock-free or wait-free. *Lock-free* [47] algorithms guarantee that regardless of both the contention caused by concurrent operations and the interleaving of their suboperations, always at least one operation will progress. However, there is a risk for starvation as progress of other operations could cause one specific operation to never finish. *Wait-free* [46] algorithms are lock-free and moreover they avoid starvation. In a wait-free algorithm every operation is guaranteed to finish in a limited number of steps, regardless of actions of other concurrent operations. Non-blocking algorithms have been shown to be of big practical importance [42,45,78], and recently NOBLE, which is a non-blocking inter-process communication library, has been introduced [98]. As a result, many aspects of concurrent data objects have been researched deeply such as consistency conditions [13, 50, 90], concurrency hierarchy [30] and fault-tolerance [74].

In this paper, we look at another aspect of concurrent data objects: the freshness of the object states returned by read-operations. Freshness is a significant property for shared data in general and has achieved great concerns in databases [18,56,85] as well as in caching systems [60, 65, 68]. Briefly, freshness is a yardstick to evaluate how fresh/new a value of a concurrent object returned by its read-operation is, when the object is updated and read concurrently. For concurrent data objects, although read-operations are allowed to return any value written by other concurrent operations, they are preferred to return the freshest/latest one of these valid values, especially in reactive/detective systems. For instance, monitoring sensors continuously concurrently input data via a concurrent object and the processing unit periodically reads the data to make the system react accordingly. In such sys-

tems, the freshness of data influences how fast the system reacts to environment changes.

However, there are few results on the freshness problem in the literature. Simpson [51, 52] suggested a freshness specification for a single-writer-to-single-reader asynchronous communication mechanism, which is different from atomic register suggested by Lamport [62]. Simpson's communication model with a single writer and a single reader is not suitable for fully concurrent shared objects that many readers and many writers can concurrently access.

These issues motivate us to define and attack the freshness problem for wait-free shared objects. We model the problem as an online problem and then present two algorithms for it. The first one is a deterministic algorithm, which is a non-trivial adaptation from an online search algorithm called *reservation price policy* [28]. The algorithm achieves a competitive ratio $\sqrt{\alpha}$, where α is a function of execution-time upper-bound of wait-free operations. Subsequently, we prove that the algorithm is optimal by proving that $\sqrt{\alpha}$ is the best freshness competitive ratio for deterministic algorithms. The second is a new competitive randomized algorithm with competitive ratio $\frac{\ln \alpha}{1+\ln 2-\frac{2}{\sqrt{\alpha}}}$. The randomized algorithm is nearly optimal since our results [24] from an elaboration on the EXPO search algorithm [28] showed that $O(\ln \alpha)$ is an asymptotically optimal competitive ratio for randomized freshness algorithms.

The rest of this paper is organized as follows. Section 7.2 briefly introduces the concept of competitive ratio, which will be used throughout the paper. Section 7.3 describes the freshness problem and models it as an online problem. Section 7.4 presents the optimal deterministic algorithm for the freshness problem. Section 7.5 presents the randomized algorithm. Finally, Section 7.6 concludes the paper.

7.2 Preliminaries

In this section, we give a brief introduction to the competitive ratio of online algorithms that will appear frequently in this paper.

Online problems are optimization problems, where the input is received online and the output is produced online so that the cost of processing the input is minimum or the outcome is best. If we know the whole input in advance, we may find an *optimal offline algorithm* OPT processing the whole input with the minimum cost. In order to evaluate how good an online algorithm is, the concept of *competitive ratio* is suggested.

Competitive ratio : An online algorithm ALG is considered competitive with a competitive ratio c (or *c-competitive*) if there exists a constant β such that for any finite input I [16]:

$$ALG(I) \leq c \cdot OPT(I) + \beta \tag{7.1}$$

where $ALG(I)$ and $OPT(I)$ are the costs of the online algorithm ALG and the optimal offline algorithm OPT to service input I, respectively. The competitive ratio is a well-established concept and the comparison with the optimal off-line algorithm is natural in scenarios where either absolute performance measures are meaningless or assumption on known probability distributions of some inputs is not feasible.

A popular way to analyze an online algorithm is to consider a game between an *online player* and a malicious *adversary*. In this game, i) the online player applies the online algorithm on the input generated by the adversary and ii) the adversary with the knowledge of the online algorithm tries to generate the worst possible input for the player. The input processing costs are very expensive for the online algorithm but still inexpensive for the optimal offline algorithm.

Adversary : For deterministic online algorithms, the adversary with knowledge of the online algorithms can generate the worst possible input to maximize the competitive ratio. However, the adversary cannot do that if the online player uses randomized algorithms. In randomized algorithms, depending on whether the adversary can observe the output from the online player to construct the next input, we classify the adversary into different categories. The adversary that constructs the whole input sequence in advance regardless of the output produced by the online player is called *oblivious adversary*. A randomized online algorithm is *c-competitive* to an oblivious adversary if

$$E[ALG(I)] \leq c \cdot OPT(I) + \beta \tag{7.2}$$

where $E[ALG(I)]$ is the expected cost of the randomized online algorithm ALG on the input I. The other adversary that observes the output produced by the online player so far and then based on that information constructs the next input element is called *adaptive adversary*. Since the oblivious adversary is more natural and more practical for modeling real problems, we design a new competitive randomized freshness algorithm against an oblivious adversary.

The competitive analysis that uses the competitive ratio as a yardstick to evaluate algorithms is a valuable approach to resolve the problems where i) if we had some information about the future, we could have found an optimal solution, and ii) it is impossible to obtain that kind of information.

7.3 Problem and Model

Linearizability [50] is the correctness condition for concurrent objects. It requires that operations on the objects appear to take effect atomically at a point of time in their execution interval. This allows a read operation to return any of values written by concurrent write operations, which is illustrated by Figure 7.1.

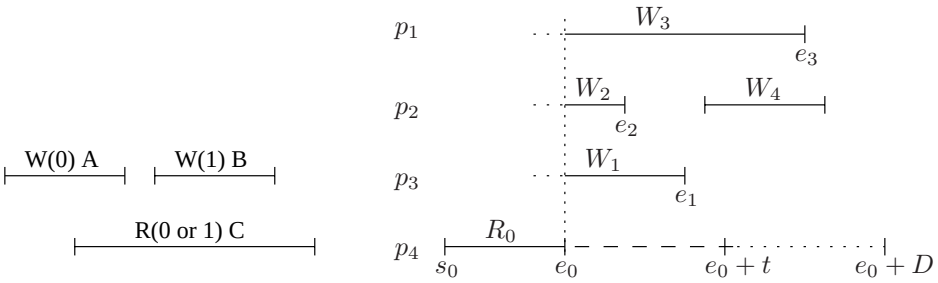

Figure 7.1: Illustrations for concurrent reading/writing and freshness problem

We use "W(x) A" ("R(x) A") to stand for a write (read) operation of value x to (from) a shared register by process A. It is correct for C to return either 0 or 1 with respect to linearizability. However, from freshness point of view we prefer C to return 1, the newer/fresher value of the register. The freshness problem is to find a solution for read operations to obtain the freshest value from a shared object. Intuitively, if a read operation lengthens its execution interval by putting some delay between the invocation and the response, it can obtain a fresher value but it will respond more slowly from application point of view. Therefore, the freshness problem is to design read-operations that both respond fast and return fresh values.

The freshness problem is especially interesting in reactive systems, where monitoring sensors continuously and concurrently input data for a processing unit via a concurrent data object. The unit periodically reads the data from the object and subsequently makes the system react to environment changes accordingly. In order to react fast, the read-operation used by the unit must both respond fast and return a value as fresh as possible. If the read-operation responds immediately at time e_0 and an environment change occurs at time $e_0 + \epsilon$, the system must wait for a period T until the next read in order to observe the change. In this scenario, the system will react faster if the read-operation delays a bit to return the fresh value at $e_0 + \epsilon$. The system will subsequently react according to the change at time $e_0 + \epsilon$ instead

of waiting until time $e_0 + T$ to be able to observe the change, where $\epsilon << T$ (Assume that processing time is negligible.).

The freshness problem is illustrated by Figure 7.1. In the illustration, a read operation R_0 runs concurrently to three write operations W_1, W_2 and W_3 on a concurrent shared object. In this paper, read/write operations imply operations on the same object. The actual execution interval of a operation i is defined from the time s_i the operation starts to the time e_i it takes effect (i.e. linearization point [50]). A time axis runs from left to right. The value returned by R_0 becomes fresher if there are more end-points e_i appear in the interval $[s_0, e_0]$. In the illustration, if R_0 delays the time-point e_0 to $e'_0 = e_0 + d$, the execution interval $[s_0, e'_0]$ will include two more end-points e_1 and e_2 and thus the value returned is newer. However, the delay will also make the read-operation respond more slowly. This implies that R_0 needs to find the time delay d so as to maximize the freshness value $f_d = \frac{k(\#we_d)}{h(d)}$, where $\#we_d$ is the number of new write-endpoints earned by delaying R_0's read-endpoint an interval d and k, h are increasing functions that depend on real applications. Each application may specify its own functions k and h according to the relation between the latency and freshness in the application.

Assume that the shared object supports a function for read operations to check how many write operations (with their timestamp) are ongoing at a time[1]. A write-timestamp wt shows the *start-point* of the corresponding write operation whereas a read-timestamp rt shows the *end-point* of the corresponding read operation. The timestamp objective is to help R_0 ignore W_4 due to $rt_0 < wt_4$. Note that R_0 only needs to consider write-endpoints of write operations that occur concurrently to R_0 in its original execution interval $[s_0, e_0]$, e.g. R_0 will ignore W_4. Therefore, in the freshness problem, the number of concurrent write operations that have not finished at the original read-endpoint e_0 is known and is called M. This number is also the total number of considered write-endpoints, i.e. $\#we \leq M$.

The most challenging issue in the freshness problem is that the end-points of concurrent write operations appear unpredictably. In order to analyze the problem, we consider it as an online game between a player and an oblivious adversary where the malicious adversary decides when to place the write-endpoints e_i on-the-fly and the player (the read operation) decides when she should stop and place her read-endpoint e'_0. The online game starts at the original read-endpoint e_0 and the player knows the total number of write-endpoints M that the adversary will use throughout the game. At a time t, the player knows how many of M end-points have been used by the adversary so far, i.e. $\#we_t$, (by comparing M with the number of ongoing write operations that ran concurrently with the original read

[1]The assumption is practical since this can be done by adding a list of timestamps of ongoing write operations to the shared object.

operation) and computes the current freshness value $f_t = \frac{k(\#we_t)}{h(t)}$. For each f_t observed, without knowledge of how the value will vary in the future, the player must decide whether she accepts this value and stops or waits for a better one. In this online game, the player's goal is to minimize the competitive ratio $c = \frac{f_{max}}{f_{chosen}}$, where f_{chosen} is the freshness value chosen by the player and f_{max} is the best value in this game, which is chosen by the adversary. The duration of this game D is the upper bound of execution time of the wait-free read/write operations and is known to the player. This implies that all the M write-endpoints must appear at a time-point in the interval, i.e. $\#we_D = M$.

In summary, we define the freshness problem as follows. Let M be the number of ongoing wait-free write operations at the original read-endpoint e_0 of a wait-free read operation and D be the execution-time upper-bound of these wait-free read/write operations. The read operation needs to find a delay $d \leq D$ for its new end-point e'_0 so as to achieve an optimal freshness value $f_d = \frac{k(\#we_d)}{h(d)}$, where $\#we_d$ is the number of write-endpoints earned by the delay d and k, h are increasing functions that reflect the relation between latency and freshness in real applications. The read-operation is only allowed to read the object data and check the number of ongoing write-operations. The write-operation is only allowed to write data to the object. We assume the time is discrete, where a time unit is the period with which the read operation regularly checks the number of ongoing write operations on the shared object. The extended read operation is still wait-free with an execution-time upper-bound $2D$.

The rest of this paper presents two competitive online algorithms for the freshness problem. The first one is an optimal deterministic algorithm with competitive ratio $\sqrt{\alpha}$, where $\alpha = \frac{h(D)}{h(1)}$. The second one is a nearly-optimal randomized algorithm with competitive ratio $\frac{\ln \alpha}{1+\ln 2-\frac{2}{\sqrt{\alpha}}}$. Note that the competitive ratios do not depend on k and M, the parameters related to the number of end-points.

7.4 Optimal Deterministic Algorithm

Modeling the freshness problem as an online game, we observe that the freshness problem is a variant of the online search problem [28]. In the online search problem, a player searches for the maximum (minimum) price in a sequence of prices that unfolds daily. For each day i, the player observes a price p_i and she must decide whether to accept this price or to wait for a better one. The search game ends when the player accepts a price, which is also the result.

Inspired by an online search algorithm called *reservation price policy* [28], we suggest a competitive deterministic algorithm for the freshness problem. In the

freshness problem, in addition to the fact that the player is searching for the best in a sequence of freshness values that unfolds sequentially in a foreknown range, there are more restrictions on the adversary, particularly on how the adversary can vary the freshness value f_t at a time t:

$$\frac{f_{t-1} * h(t-1)}{h(t)} = \frac{k(\#we_{t-1})}{h(t)} \leq f_t = \frac{k(\#we_t)}{h(t)} \leq \frac{k(M)}{h(t)} \tag{7.3}$$

The restrictions come from the fact that the adversary cannot remove the end-points she has placed, i.e. $\#we_{t-1} \leq \#we_t \leq M$, where $\#we_t$ is the number of end-points that have appeared until a time t, and the freshness value at the time t is $f_t = \frac{k(\#we_t)}{h(t)}$, where k, h are increasing functions. The restrictions make the adversary in the freshness problem weaker than the adversary in the online search problem, and intuitively the player in the freshness problem should benefit from this. However, we will prove that this is not the case for deterministic algorithms (cf. Theorem 2).

Before presenting the deterministic freshness algorithm, we need to find upper/lower bounds on freshness values f_t. Since $1 \leq t \leq D$, from Equation (7.3) it follows $f_t \leq \frac{k(M)}{h(1)}$. On the other hand, since M ongoing write-operations must end at time-points in the interval D, the player is ensured a freshness value $f_{min} = \frac{k(M)}{h(D)}$ by just waiting until $t = D$. Therefore, the player considers to stop at a freshness value f_t only if $f_t \geq \frac{k(M)}{h(D)}$. We have $\frac{k(M)}{h(D)} \leq f_t \leq \frac{k(M)}{h(1)}$.

Deterministic Algorithm: The read operation accepts the first freshness value that is not smaller than $f^* = \frac{k(M)}{\sqrt{h(1)h(D)}}$.

Indeed, let f^* be the threshold for accepting a freshness value and f_{max} be the highest value chosen by the adversary. The player (the read operation) waits for a value $f_t \geq f^*$. If such a value appears in the interval D, the player accepts it and returns it as the result. Otherwise, when waiting until the time D, the player must accept the value $f_{min} = \frac{k(M)}{h(D)}$.

Case 1: If the player chooses a big value as f^*, the adversary will choose $f_{max} < f^*$, causing the player to wait until the time D and accept the value $f_{min} = \frac{k(M)}{h(D)}$. The competitive ratio in this case is $c_1 = \frac{f_{max}}{\frac{k(M)}{h(D)}} < \frac{f^*}{\frac{k(M)}{h(D)}}$.

Case 2: If the player chooses a small value as f^*, the adversary will place f^* at a time t, causing the player to accept the value and stop. Right after that, the adversary places all M end-points, achieving a value $f_{max} = \frac{k(M)}{h(t)} \leq \frac{k(M)}{h(1)}$ (equality occurs when the adversary chooses $t = 1$). The competitive ratio in this case is $c_2 = \frac{\frac{k(M)}{h(1)}}{f^*}$.

In order not to be fooled by the adversary, the player should choose f^* so as to make $c_1 = c_2$, which results in $f^* = \frac{k(M)}{\sqrt{h(1)h(D)}}$ and the competitive ratio $c = c_1 = c_2 = \sqrt{\frac{h(D)}{h(1)}}$.
Let $\alpha = \frac{h(D)}{h(1)}$. This leads to the following theorem.

Theorem 1. *The suggested deterministic algorithm is competitive with competitive ratio $c = \sqrt{\alpha}$, where $\alpha = \frac{h(D)}{h(1)}$.*

We now prove that there is no deterministic algorithm for the freshness problem that achieves a competitive ratio better than $\sqrt{\alpha}$.

We use a logarithmic vertical axis for freshness. Let LF denote the logarithm of freshness. More specifically, we normalize the LF axis so that freshness $\frac{k(M)}{h(D)}$ corresponds to point 0 and freshness $\frac{k(M)}{h(1)}$ corresponds to point $\ln \frac{h(D)}{h(1)} = \ln \alpha$. One unit on the LF axis multiplies the freshness by factor e (Euler's number).

We also introduce some parameters that characterize the status of a game. Let t be the time, initially $t = 1$. At any moment, let f be the maximum LF the adversary has already reached during the history of the game, and g the maximum LF the adversary can still achieve at a given time. LF value $g(t)$ at time t corresponds to freshness $k(M)/h(t)$, unless f is already larger, in which case we have $g = f$. However in the latter case the game is over, without loss of generality: The adversary cannot gain more and would therefore decrease the freshness as quickly as possible, in order to make the player's position as bad as possible, hence an optimal player would stop now. (The dotted polyline in Figure 7.2 illustrates the case $f = g(t)$ in which the player should stop at time t.)

The horizontal axis is for the logarithm of $h(t)$. We normalize it so that $h(1)$ corresponds to point 0 and $h(D)$ corresponds to point $\ln \frac{h(D)}{h(1)} = \ln \alpha$). Note that, in these logarithmic coordinates, g simply decreases at unit speed, starting at point $\ln \alpha$. Finally, let c denote the current LF. We remark that c can decrease at most at unit speed but can jump upwards arbitrarily as long as $c \le g$.

Theorem 2. *The optimal deterministic competitive ratio is asymptotically (subject to lower-order terms) $\sqrt{\alpha}$, where $\alpha = \frac{h(D)}{h(1)}$.*

Proof. To prove the theorem, we only need to show one of the adversary's strategies against which no online deterministic algorithm can achieve a competitive ratio better than $\sqrt{\alpha}$. We work in the logarithmic coordinates as defined above, which makes the argument rather simple.

The adversary starts with $c = \frac{\ln \alpha}{2} = \frac{\ln \frac{h(D)}{h(1)}}{2}$. Then she decreases c at unit speed until the player stops. Immediately after this moment, c jumps to g if $c > 0$

at the stop time (Case 1), otherwise c keeps on decreasing at unit speed (Case 2). Clearly, we have constantly $g - c = \frac{\ln \alpha}{2}$ until the stop time. Let p be the player's value of LF. In Case (1) we finally get $f = g$, hence $f - p = g - c = \frac{\ln \alpha}{2}$ (cf. the dashed polyline c_1 in Figure 7.2). In Case (2), f has still its initial value $\frac{\ln \alpha}{2}$ whereas $p \leq 0$, hence $f - p \geq \frac{\ln \alpha}{2}$ (cf. the line c_2 in Figure 7.2). Thus the competitive ratio is at least $e^{\frac{\ln \alpha}{2}} = \sqrt{\alpha}$. The player can achieve this competitive ratio by applying the deterministic algorithm given above. □

This result shows that a deterministic player cannot take advantage of the constraints on the behavior of freshness in time (compared to online search on unrestricted sequences of profit values).

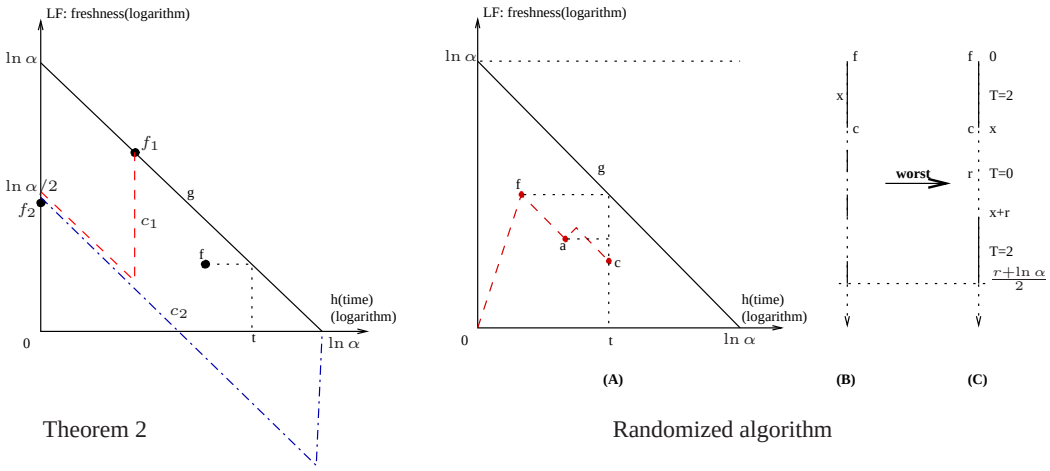

Figure 7.2: Illustrations for Theorem 2 and the randomized algorithm

7.5 Competitive Randomized Algorithm

In this section, we present a competitive randomized algorithm for the freshness problem. The algorithm achieves a competitive ratio $c = \frac{\ln \alpha}{1+\ln 2-\frac{2}{\sqrt{\alpha}}}$, where $\alpha = \frac{h(D)}{h(1)}$.

As discussed in the previous section, our problem is a restricted case of online search. We model the problem by a game between an (online) player and an adversary. The adversary's profit is the highest freshness ever reached. The player's profit is the freshness value at the moment when she stops. Note that for

a player running a randomized strategy, the profit is the expected freshness value, with respect to the distribution of stops resulting from the strategy and input. We shall make use of a known simple transformation of (randomized) online search to (deterministic) one-way trading [28]: The player has some budget of money she wants to exchange while the exchange rates may vary over time. Her goal is to maximize her gain. The transformation is given as follows: The budget corresponds to probability 1, and exchanging some fraction of money means to stop the game with exactly that probability. Note that a deterministic algorithm for online search has to exchange all money at *one* point in time. For the freshness problem, it is possible to apply a well-known competitive randomized algorithm EXPO [28]. Applying the EXPO algorithm on the freshness problem achieves a competitive ratio $\ell \frac{2^{\ell-1+1/\ln 2}}{2^{\ell-1+1/\ln 2}-\frac{1}{\ln 2}}$, where $\ell = \log_2 \alpha$. That means for the freshness problem our randomized algorithm is better than the EXPO algorithm by a constant factor $\frac{1+\ln 2}{\ln 2}$ when α becomes large.

Theorem 3. *There is a randomized algorithm for the freshness problem with competitive ratio* $\frac{\ln \alpha}{1+\ln 2-\frac{2}{\sqrt{\alpha}}}$, *where* $\alpha = \frac{h(D)}{h(1)}$

Proof. We start with some conventions. We imagine that the money, both exchanged and non-exchanged, is "distributed" on the LF axis. Formally, the allocation of money on the LF axis at any time is described by two non-negative real *density functions* S and T, where $S(x)$ is the density of not yet exchanged money in point x of the LF axis, $T(x)$ is similarly defined for the money that has been already exchanged. What functions S and T specifically are, and how they are modified by the opponents' actions, will be described below. Let the total amount of money be $\ln \alpha$ by convention. (Recall that scaling factors do not influence the competitive ratio.)

The *value* of every piece of *exchanged* money is the freshness value of its position on the LF axis. Note that the total value of exchanged money defined in this way, i.e. the integral over the value-by-density product, is the player's profit in the game. Moreover, the player can temporarily have some of the money in her *pocket*.

The idea of the strategy is to guarantee some concentration of exchanged money immediately below the final f, either some constant minimum density of T or, even better, a constant amount at one point not too far from f. We want to keep T simple in order to make the calculations simple. (The well-known δ_x symbol used below denotes the distribution with infinite density at a single point x but with integral 1 on any interval that contains x. We also use the same notations f, g, c as earlier.) Locating much money instantaneously is risky because c may jump upwards, and then this money has little value compared to the adversary's. On the other hand,

since c decreases at most with unit speed, the player may completely abstain from exchanging money as long as c is increasing, and wait until c goes down again. These preliminary thoughts lead to the following strategy.

In the beginning, let the not-yet-exchanged money be located on the LF axis on interval $[0, \ln \alpha]$ with density 1, that is, we have $S = 1$ on this interval. Remember that g decreases at unit speed. The player puts the money above g in her pocket. Whenever f increases, she also puts the money below the new f in her pocket. Hence we always have $S = 1$ on $[f, g]$, and $S = 0$ outside. The player continuously locates exchanged money on the LF axis, observing the following rule: *If you have money in your pocket and c is positive and decreasing, and $T(c) < 2$ at the current c, then set $T(c) := 2$. If the game is over (because of $f = g$) and not all money is exchanged yet, put the rest r on the current c.* Note that the adversary must set the final c nonnegative.

Filling-up density T to 2 is always possible, by the following argument: The player uses the one unit of money from S that she gets per time unit from the region above the falling g, and the money from S that she got directly from the current points c when f went upwards.

Obviously, the player produces a density function T that is constantly 2 on certain intervals and 0 outside, plus some component $r\delta_c$. We make some crucial observations regarding the final situation: (1) T has density 2 on interval $(c, f]$, or we have $c = f$. (2) The *gaps* with $T = 0$ between the "$T = 2$ intervals" have a total length not exceeding r.

These claims follow easily from the strategy: (1) Either c begins decreasing, starting from the last f, and T is filled up to 2 all the time when $c > 0$, as we saw above, or the final c equals the final f. (2) Whenever f went upwards, the player has taken from S the money corresponding to the increase of f, and later she has transferred it to T and located it at the same points again. Hence, only on intervals not "visited" again by c we have $T = 0$, and the money taken from S on these intervals is still in the player's pocket and thus contributes to r.

Figure 7.2-(A) illustrates the player's behavior. The dashed line represents a variation of c in a game; point c is the final value of c when the game ends, i.e. $f = g(t)$. For all values v on the LF axis between f and a and between a and c, the player sets $T(v) = 2$.

Using (1),(2) we now analyze the profit the player can guarantee herself. Remember that the value of exchanged money located on the LF axis decreases exponentially. Let $x = f - c$ (final values). Both r and x depend on the input, i.e., the behavior of c in time. The total amount of money is fixed, it equals $\ln \alpha$. For any fixed r, x, the worst case is now that the gaps in T sum up to the maximum length r and are as high as possible on the LF axis, that is, immediately below point c, because in this case all exchanged money outside $[c, f]$ has the least possible value.

That is, T has only one gap, namely interval $[c-r, c]$.

Figure 7.2-(C) illustrates the worst case corresponding to an instance -(B), where solid lines represent ranges on the LF axis with $T = 2$. In the worst case, the adversary shifts all solid lines except for $[c, f]$ to the lowest possible position so as to minimize the player's profit.

Hence, a lower bound on the player's profit, divided by the value at f, is given by

$$\min_{r,x} \left(2\int_0^x e^{-t}dt + re^{-x} + 2\int_{x+r}^{(r+\ln\alpha)/2} e^{-t}dt \right),$$

where we started integration (with $t = 0$) at point f and go down the LF axis (cf. Figure 7.2-(C)). Verify that, in fact, $\int Tdt = \ln\alpha$. The above expression evaluates to

$$2 + (r - 2 + 2e^{-r})e^{-x} - 2e^{-(r+\ln\alpha)/2} > 2 + (r - 2 + 2e^{-r})e^{-x} - 2/\sqrt{\alpha}.$$

For any fixed x, this is minimized if $2e^{-r} = 1$, that is, $r = \ln 2$. Since now $r-2+2e^{-r} = \ln 2-2+1 < 0$, the worst case is $x = 0$, which gives $1+\ln 2-2/\sqrt{\alpha}$. The adversary earns $\ln\alpha$ times the value at f. □

7.6 Conclusions

To the best of our knowledge, this paper is the first paper that defines the freshness problem for wait-free data objects. Within this paper, we have modeled the freshness problem as an online problem and then have presented two online algorithms to solve it. The first one is a deterministic algorithm with freshness competitive ratio $\sqrt{\alpha}$, where α is a function of execution-time upper-bound of wait-free operations. The function α is specified by real applications according to their purpose. Subsequently, we prove that $\sqrt{\alpha}$ is asymptotically the optimal freshness competitive ratio for deterministic algorithms. The second is a randomized algorithm with freshness competitive ratio $\frac{\ln\alpha}{1+\ln 2-2/\sqrt{\alpha}}$. The randomized algorithm is nearly optimal. In [24] it has been showed that $O(\ln\alpha)$ is a lower bound on competitive ratios for the one-way trading with time-varying exchange-rate bounds corresponding to the freshness problem. This gives a lower bound $O(\ln\alpha)$ to competitive ratios of randomized freshness algorithms.

This paper provides a starting point to further research the freshness problem on concurrent data objects as an online problem. The paper has presented algorithms that can apply on general wait-free data objects without any restrictions. However, wait-free data objects are just one kind of concurrent data objects while freshness itself is an interesting problem for concurrent data objects in general.

Chapter 8

Trading Latency for Freshness: One-Way Trading with Time-Varying Exchange Rate Bounds[1]

Peter Damaschke[2], Phuong Hoai Ha[2], Philippas Tsigas[2]

Abstract

This paper studies the problem of trading latency for freshness when accessing concurrent data objects. We observe that it can be modeled as variants of the one-way trading problem, a fundamental online problem in finance.The difference between these variants and the original one is that the bounds of the exchange rates are not constant but vary with time in certain ways.

The main question this paper addresses is whether these new variants can conduce to better algorithms with respect to freshness compared to the original model. The answer is "yes". The key results obtained in this paper are the followings. First, for the variants we provide an algorithm that achieves a better competitive ratio compared with the threat-based algorithm, an optimal algorithm for the original one-way trading model. Second, we prove lower bounds of competitive ratios for the variants, showing that our algorithm is optimal for one of the variants.

[1]Expanded version of a preliminary result published in Technical report no. 2005-17, Department of Computer Science and Engineering, Chalmers University of Technology, Gothenburg, Sweden, Oct. 2005.

[2]Department of Computer Science and Engineering, Chalmers University of Technology, S-412 96 Gothenburg, Sweden. Email:{*ptr, phuong, tsigas*}*@cs.chalmers.se*

8.1 Introduction

The one-way trading problem is a fundamental on-line problem in finance [26–28]. In the problem, a player is exchanging her initial wealth in one currency (e.g. dollar) to another currency (e.g. yen) so as to maximize her profit while the exchange rate (from dollar to yen) varies unpredictably. El-Yaniv et al. suggested optimal solutions for several slight variants of the problem. Since the optimal solutions for these variants are quite simple computationally (that is, the amounts to exchange are easy to compute, but the analysis is quite sophisticated), practical issues can be transformed to one-way trading in order to find optimal solutions [40].

However, there are still natural problems that cannot be transformed to any of these variants in a tight way. One of them comes from the freshness problem of concurrent data objects. The freshness problem is to design access operations for concurrent objects that quickly return the freshest/latest values of data while the objects are modified concurrently by other operations. For concurrent data objects, read-operations are allowed to return any value written by a concurrent write-operation. However, from an application point of view the read-operations are preferred to return the latest valid value, especially in reactive/detective systems. For instance, monitoring sensors continuously concurrently input data via a concurrent object and the processing unit periodically reads the data to make the system react accordingly. In such systems, the freshness of data influences how fast the system reacts to environment changes.

8.1.1 Freshness of Concurrent Data Objects

Concurrent data objects like stacks [45, 87, 103], queues [78, 82, 103] and linked lists [34,42,103] play a significant role in distributed computing. As a result, many aspects of concurrent data objects have been researched deeply such as consistency conditions [13, 50, 90], concurrency hierarchy [30] and fault-tolerance [74]. In this paper, we look at another aspect of concurrent data objects: freshness of states/values returned by read-operations of read-write objects.

Figure 8.1 illustrates the freshness problem. A read-operation R_0 runs concurrently with three write-operations W_1, W_2 and W_3 on the same object, where the execution-time upper-bound D of the wait-free read/write operations on the object is known. Each operation takes effect at an endpoint e_i (i.e. linearization point [50]) that appears unpredictably before time D. At the endpoint e_0, the number of ongoing write-operations M is given. The freshness problem is to find a delay t, a real number in $[0, D]$, so that the new endpoint $e'_0 = e_0 + t$ of the read-operation R_0 has an optimal freshness value $f_t = \frac{k(\#we_t)}{h(t)}$, where $\#we_t$ is the number of further write-operation endpoints earned by the delay t, e.g. $\#we_D = M$;

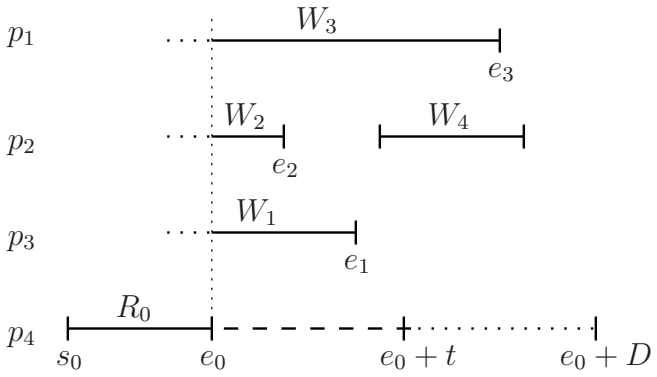

Figure 8.1: Freshness problem

and k, h are increasing functions that reflect the relation between freshness and latency in real applications. The read-operation is only allowed to read the object data and check the number of ongoing write-operations. The write-operation is only allowed to write data to the object.

In Figure 8.1, read-operation R_0 earns two more endpoints e_1, e_2 of concurrent write-operations W_1, W_2 due to delaying endpoint e_0 by t, and thus returns a fresher value. Intuitively, if R_0 delays the endpoint e_0 by duration D, it will return the freshest value at endpoint e_3[1]. However, from the application point of view the read-operation R_0 in this case will respond most slowly. Therefore, the goal in the freshness problem is to design read-operations that *respond fast* as well as *return fresh values*. Since there are two conflicting objectives for read-operation: *fast response* and *fresh value*, we define a measure of freshness as a function that is monotone increasing in the number of earned endpoints and decreasing in time. The delay is chosen by a randomized policy. In this view the freshness problem is an online search problem with freshness values as profits.

In an online search problem, the player searches for the maximum price in a sequence of prices that are unfolded sequentially. When observing a new price, she must decide whether to accept this current price or to wait for a better one. The game ends when she accepts a price. The online search problem can be transformed to an one-way trading problem in which the player exchanges all her money at once. It is well-known that randomized online search is equivalent to (deterministic) one-way trading: The amount of money exchanged at every moment corresponds to the probability to stop and to accept the current profits.

The freshness problem is especially interesting in reactive/detective systems

[1]Note that R_0 only needs to consider write-endpoints of write operations that occur concurrently to R_0 in its original execution interval $[s_0, e_0]$, e.g. R_0 will ignore W_4.

where monitoring sensors continuously concurrently input data via a concurrent data object and a processing unit periodically reads the data with period T to make the system react accordingly. If the read-operation R_0 of the processing unit returns data at e_0, the system will change its state according to the old data and will keep this state until the next period. If R_0 delays its endpoint e_0 by time $t < T$, the system will change its state according to the data at e_1, which means the system in this case may react faster to environment changes.

In particular, in this paper, we consider the freshness problem with the simple definition $f_d = \frac{\#we_d}{d}$, where d is a natural number in $[1, D]$,but our study may be extended to more general freshness functions. Since endpoints cannot disappear, the number of endpoints observed on-line cannot decrease in time. This implies

$$\frac{f_{d-1}(d-1)}{d} \leq f_d \leq \frac{M}{d}, \forall d \in [1, D] \tag{8.1}$$

Since all concurrent write-operations finish at time D at the latest, the freshness at time D is always M/D. This implies that the online player can always achieve at least the freshness M/D by just waiting until time D and thus she can ignore all freshness values smaller than M/D.

8.1.2 Our contributions

Based on the observation in the previous subsection, in this paper we consider the following new one-way trading models. The first one is a continuous model on time interval $[1, D]$ with known duration D and exchange rates r that fulfill $r(t) \leq r(u)\frac{u}{t}$ for any times $t < u$ and $r(t) \geq \frac{M}{D}$ $\forall t \in [1, D]$, where M is the maximal allowed exchange rate at $t = 1$. The model is motivated by Inequality (8.1). The second model is time-discrete with known duration D and exchange rates that are bounded from above by any decreasing function of time $M(t)$ and bounded from below by a constant m. Note that the second model "contains" the first one when D is large. Any instance of the first model can be transformed to the second model where $m = \frac{M}{D}$ and $M(t) = \frac{M}{t}$. For the first model we prove that no online algorithm can achieve a competitive ratio less than, asymptotically, $(\ln D)/2$. For the second model, we suggest an optimal threat-based algorithm with competitive ratio

$$c^* = \max_{1 \leq k \leq D} \left\{ c \;\middle|\; c = k\left(1 - \left(\frac{c-1}{\frac{M(k)}{m} - 1}\right)^{1/k}\right) \right\} \tag{8.2}$$

Since this expression is hard to evaluate analytically, we add some numerical results for $M(t) = \frac{M}{t}$ and $m = \frac{M}{D}$ in order to compare the competitive ratios in the case of the freshness problem at the end of Section 8.3. As for the relation

between the two results, since the adversary in the second model is less restricted (or stronger) than one in the first model, the lower bound of competitive ratios $(\ln D)/2$ holds also for the second model. We chose to consider the stronger adversary in the second model because the threat-based algorithm exchanges money only at increasing rates, thus it does not even exploit the limited decay speed of exchange rates in the first model. Our numerical experiments suggest that the threat-based algorithm with the new competitive ratio c^* is still not too far from the lower bound, despite the stronger adversary. This is explained by the observation that slowly increasing exchange rates seem to be the worst case for the online player. In the lower-bound proof we consider continuous time only because this simplifies the arguments. Note that when D is large, the difference between continuous and discrete time models disappears.

The rest of this paper is organized as follows. Section 8.2 presents the lower bound of competitive ratios. Section 8.3 presents an optimal threat-based algorithm for the second model. Section 8.4 concludes the paper with some remarks.

8.2 The Lower Bound of Competitive Ratios

In this section, we present the lower bound of competitive ratios for the first model. The δ notation we use once in our proof below is well-known: δ_c is the distribution with infinite density at a single point c but with integral 1 on any interval that contains c.

Theorem 4. *For every $\epsilon > 0$ there exists D_ϵ such that for all $D > D_\epsilon$, no algorithm for the first model can achieve a competitive ratio better than $(\ln D)/2 - \epsilon$.*

Proof. (1) We start with new *notations* that will make the argument easier, as they are adapted to the geometry of the problem. In particular, we work with logarithmic axes for both exchange rate and time.

In the following, let LF be the logarithm of the exchange rate (freshness, in our case). We normalize the LF axis in such a way that exchange rate M/D corresponds to point 0 and exchange rate M to point $\ln D$. This is convenient because now, going one unit upwards on the LF axis increases the exchange rate by factor e (Euler's number). We also normalize the amount of money of the online player to $\ln D$. (Note that scaling factors do not affect the competitive ratios.)

(2) Next we introduce some *parameters* that characterize the status of the game between online player and adversary at any moment. Let t denote the time, initially $t = 1$.

Defining f: Let f be the maximum LF the adversary has already reached during the history of the game. But initially we set $f = 0$, since M/D is the guar-

anteed exchange rate that the adversary can use (according to the definition of the model). In particular we have $f \geq 0$ at any time.

Defining g: Let g be the maximum LF the adversary can still achieve in at the considered time. In more detail, g corresponds to exchange rate M/t at time t, unless f is already larger, in which case we have $g = f$. In the latter case the game is over, without loss of generality: Note that the adversary cannot gain more than g and would therefore decrease the exchange rate as quickly as possible, in order to make the player's position worse, hence the player should stop immediately. We also remark that, on the logarithmic time axis, g decreases at unit speed, starting at point $\ln D$.

Defining c: Let c denote the current LF, as determined by the adversary. Hence an instance of the problem is given by c as a function of time. Note that on the logarithmic time axis, parameter c, like g, can decrease at most at unit speed, but c can jump upwards arbitrarily, according to the model.

(3) After the definition of key parameters we describe the "court" for our game. We imagine that the money, both the exchanged and non-exchanged money, is distributed on the LF axis. Moreover, the player can temporarily have some of the money in her *pocket*. Formally, the allocation of money on the LF axis at any time will be described by two non-negative real *density functions* S and T, where $S(x)$ is the density of not yet exchanged money in point x of the LF axis, and $T(x)$ is similarly defined for the money that has already been exchanged. What the functions S and T specifically are and how they are modified by the opponents' actions will be described below.

The *value* of every piece of *exchanged* money is the exchange rate corresponding to its position on the LF axis. Therefore the total value of exchanged money, which is the integral over the value-by-density product, gives the player's profit in the game.

(4) Next we describe two *primitives* that our adversary will use in her strategy specified below.

The adversary can take parts of the not-yet-exchanged money, that is, diminish $S(x)$ at certain points x. This money is handed out to the player, i.e., moved to the player's pocket. The rest is always uniformly spread out on a certain interval $[h, g]$, so that density S is constant on $[h, g]$. The number h with $f \leq h < g$ will be decided below in the adversary's strategy.

The adversary can also modify T in a special way: She can *move* pieces of exchanged money on the LF axis, and increase their amount (!) by factor e (decrease their amount by factor $1/e$) if they are moved one unit downwards (upwards): The effect is that the total value of exchanged money, i.e., the player's profit, remains the same. This manipulation is only used for easier bookkeeping, in order to simplify function T.

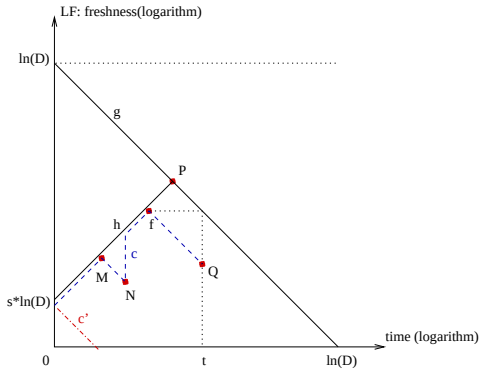

Figure 8.2: Illustration for the proof of Theorem 4

According to the one-way trading setting, the player can only place money continuously on the LF axis at the current point c and thus increase $T(c)$.

Finally, let s be some constant between 0 and 1 that we fix later. It has only technical importance.

(5) Now we are prepared to specify the *adversary's strategy*. At start let be $c = h = s \ln D$. Hence we have initially $S = 1/(1-s)$ on $[h, g]$. (Recall that the total amount of money is $\ln D$.) As g decreases with time, the adversary gives the money above point g to the player. Simultaneously she increases h at unit speed and gives the money below point h to the player, too. Thus, S on $[h, g]$ remains constantly $1/(1-s)$ all the time, and the player obtains $2/(1-s)$ units of money per time unit. As long as the player has money in her pocket, the adversary decreases c at unit speed. Whenever the player runs out of money, the adversary sets immediately $c := h$. Also, if the player "raises a loan" and takes extra money from $[h, g]$, which is of course allowed by the game, the adversary increases h accordingly, so as to keep $S = 1/(1-s)$ in all points of $[h, g]$. We remark that when $h = g$ is reached, h has to decrease together with g in the remaining time, due to the specification of h.

Figure 8.2 illustrates the adversary's strategy. At point M, the player withholds all money in her pocket and thus the adversary decreases c at unit speed. At point N, the player spends all money in her pocket and the adversary in response sets immediately $c := h$. The game ends when the player has exchanged all money at Q, where $g = f$.

(6) We show the following *invariant*: Before the game ends, the adversary can always hold $T(x) \leq 2/(1-s)$ at every point x.

This is vacuously true in the beginning, since no money has been exchanged yet. As long as the player exchanges all her money immediately, we have $c =$

$h = f$, and c increases at unit speed, hence the invariant remains true. The more complicated case is that the player withholds some money in her pocket. By the adversary's strategy, h keeps going upwards, but c goes downwards at unit speed. Now, whenever the player exchanges some money at the current c, the adversary moves it upwards and fills the gap between f and h, without ever making $T(x) > 2/(1-s)$ at any point x. This is always possible, since the pocket contains an amount of at most $2(h-f)/(1-s)$, due to the adversary's policy of adjusting h and giving money. As soon as the player decides again to exchange *all* money that is currently in her pocket, we are back to case $c = h$. Continuing in this way, the adversary always recovers the claimed invariant.

(7) The final *analysis* step of our proof compares the profits of both players. At the end of the game the player has exchanged all money, hence $g = h = f$, and all money in T is below f. Moreover, the adversary must go to $c = 0$, and the player can exchange all remaining money at point c. Since the adversary would instead exchange all money at the final $f \geq s \ln D$, and $T(x) \leq 2/(1-s)$ holds at every point $x \neq 0$, the best possible competitive ratio would be achieved in the following case: $f = s \ln D$, $T(x) = 2/(1-s)$ at every point $x > 0$, and the remaining $(1 - \frac{2s}{1-s}) \ln D$ units of exchanged money are at $x = 0$ (line c' in Figure 2), which adds a term $\frac{1-3s}{(1-s)} \ln D \delta_0$ to T. Since the value of exchanged money decreases exponentially down the LF axis, and $e^{-s \ln D} = 1/D^s$, we can now calculate the inverse competitive ratio in this best case, as the player's profit divided by $\ln D$:

$$\frac{2}{(1-s)\ln D} \int_0^{s \ln D} e^{-x} dx + \frac{1-3s}{(1-s)D^s} = \frac{2}{(1-s)\ln D} - \frac{2}{(1-s)D^s \ln D} + \frac{1-3s}{(1-s)D^s}.$$

Given D, the adversary would choose s so as to minimize this expression. Now the Theorem follows: Consider any $s > 0$, arbitrarily small but fixed. For large enough D, factor $1/D^s$ becomes negligible compared to $1/\ln D$, hence we can ignore the last two terms, and the inverse competitive ratio comes arbitrarily close to $\frac{2}{\ln D}$. □

For concrete small values of D we computed numerically the s that gives the minimum, and the resulting lower bounds (cf. Figure 8.3). We remark that our analysis can be easily extended to models with exchange rates decreasing like $1/t^B$, B any constant. The only feature we needed is linearity on the logarithmic scale.

8.3 Optimal threat-based policy for the second model

In this section, we find a new optimal competitive ratio for the second model, where the upper bound of exchange rates $M(t)$ is a decreasing function with time

t and the lower bound is a constant m. The ratio is then used in the threat-based policy [28] to create an optimal algorithm for the second model. The algorithm is computationally simple: the amount of money to exchange in every step follows a simple formula. This makes the algorithm suitable for real applications.

We repeat our second one-way trading model. With a known duration D and known upper/lower bounds for exchange rates (yen per dollar) r_i: $m \leq r_i \leq M(i)$, where *the upper bound $M(i)$ is a decreasing function of time i*, the online player or trader needs to trade her initial wealth W_0 given in dollar to yen efficiently. Exchange rates are unfolded on-the-fly over a discrete time interval and when a new exchange rate is observed, a new period starts. Given a current exchange rate, the trader has to decide how much of her dollars should be exchanged to yen at the current rate. Without loss of generality, assume that the trader's initial wealth is one dollar, $W_0 = 1$.

Obviously, we can not achieve an optimal competitive ratio by directly applying the threat-based algorithm of the original model [28], where the upper bound of exchange rates is constant, to the new model, where the upper bound decreases with time. In the new model the adversary is clearly more restricted and thus the player should benefit from that. Hence, the analysis must adapt to the new model. Although the flow of the following analysis looks similar to that in the original model, the technical details have to be adapted non-trivially at various places.

Let $k \leq D$ be the length of an *increasing* sequence of exchange rates $m \leq p_1 < p_2 < \cdots < p_k \leq M(K)$, where K is the index of p_k in the original sequence, $k \leq K$. Since $M(i)$ is a decreasing function, $M(K) \leq M(k)$. This follows $m \leq p_1 < p_2 < \cdots < p_k \leq M(k)$.

For instance, if we have a sequence R of exchange rates $\{1, 2, 4, 3, 7, 5, 6\}$ with $D = 7$, then the corresponding increasing sequence P of the exchange rates is $\{1, 2, 4, 7\}$ with $k = 4$. Note that $R[5] = 7$ is included in the increasing sequence as $P[4]$ and $R[4] = 3$ is ignored since $R[3] > R[4]$. We have $P[4] \leq M(5) < M(4)$, since $P[4]$ corresponds to $R[5]$ (i.e. time/step 5) in the original sequence R and $M(i)$ is a decreasing function.

We will prove that the optimal competitive ratio c^* is

$$c^* = \max_{k=1\cdots D} \left\{ c \;\middle|\; c = k\left(1 - \left(\frac{c-1}{\frac{M(k)}{m} - 1}\right)^{1/k}\right) \right\} \tag{8.3}$$

For each D given, we always find the ratio c^* that satisfies Equation 8.3 by simply computing c for each $k = 1, 2, \cdots, D$ and then choose the maximum c as c^*. With the competitive ratio c^* found, the player follows the threat-based policy as in [28]:

- Consider trading dollar to yen at the current exchange rate only if it is the highest seen so far;
- When converting dollar, convert *just enough* dollar at the current exchange rate to ensure the competitive ratio c^* even if the adversary then drops the rate to the minimum and keeps it there until the end.

The amount of dollar s_i that should be put at the current exchange rate r_i is:

$$s_1 = \frac{1}{c} \cdot \frac{r_1 - mc}{r_1 - m} \; and \; s_i = \frac{1}{c} \cdot \frac{r_i - r_{i-1}}{r_i - m}, \forall i \geq 2. \tag{8.4}$$

where $c = c^*$,$r_1 \geq mc^*$ and r_{i-1} is the highest exchange rate before the current one. If none of exchange rates given is larger than mc^* until the end of game, the player can achieve the competitive ratio c^* by just converting all her dollars at the minimal exchange rate m at the end of game.

Since the threat-based algorithms is influenced only by the increasing sequence of given exchange rates (cf. Rule 1), henceforth we consider threat-based algorithms on the increasing sequence $P = p_1, p_2, \cdots, p_k$, where $p_1 < p_2 < \cdots < p_k$. This implies

$$s_1 = \frac{1}{c} \cdot \frac{p_1 - mc}{p_1 - m} \; and \; s_i = \frac{1}{c} \cdot \frac{p_i - p_{i-1}}{p_i - m}, \forall i \geq 2. \tag{8.5}$$

As we know, the competitive ratio c used in formula (8.5) is the target competitive ratio that the player tries to achieve. Obviously, the ratio cannot be an arbitrary small number. For instance, if the player chooses $c = 1$, she will convert all her dollars at the first exchange rate r_1 since $s_1 = 1$. Then she will run out of dollars to convert when the adversary issues a higher exchange rate r_2 in the next step and thus the player fails to achieve the competitive ratio $c = 1$. Therefore, the player following the threat-based policy achieves a competitive ratio only if the chosen ratio is large enough.

The following lemmas are inspired by the analysis of the original threat-based policy in [28].

Definition 8.3.1. *Given a sequence R of exchange rates, a threat-based algorithm A_c as defined by formula (8.5) with a ratio c, is* c-proper *with respect to R if*

- *the sum of daily exchanged dollars s_i computed by A_c, when the exchange rate sequence is R, is not larger than 1, the initial wealth; and*
- *the resulting ratio of optimal offline return over online return $A_c(R)$ with respect to R is not larger than c.*

Lemma 8.3.1. *The threat-based algorithm following Formula (8.5) with $c = c'$ is guaranteed to achieve the competitive ratio c' as long as there are enough dollars to exchange until the end of a game.*

Proof. Let D_i and Y_i be the number of dollars and yen after the exchange at a step i of an increasing sequence P of exchange rates. We will prove that at any step i the algorithm always achieves a competitive ratio c' even if the adversary drops the rate to minimum in the next step and keeps it there until the end of a game, i.e.

$$\frac{p_i}{Y_i + mD_i} \leq c',\ \forall 1 \leq i \leq k \tag{8.6}$$

We prove this lemma by induction. For the case $i = 1$, we have

$$\frac{p_1}{Y_1 + mD_1} = \frac{p_1}{s_1 + m(1 - s_1)} = c'$$

Therefore, Inequality (8.6) is correct for $i = 1$. Assume that the inequality is correct for $i = k - 1$, i.e.,

$$\frac{p_{k-1}}{Y_{k-1} + mD_{k-1}} \leq c' \tag{8.7}$$

We will prove that the inequality is also correct for $i = k$, i.e.,

$$\frac{p_k}{Y_k + mD_k} \leq c' \tag{8.8}$$

Indeed, as long as there are enough dollars to exchange until the end, we have

$$\begin{aligned}
\frac{p_k}{Y_k + mD_k} &= \frac{p_k}{(Y_{k-1} + s_k p_k) + m(D_{k-1} - s_k)} \\
&= \frac{p_k}{(Y_{k-1} + mD_{k-1}) + s_k(p_k - m)} \\
&\leq \frac{p_k}{\frac{p_{k-1}}{c'} + \frac{p_k - p_{k-1}}{c'}} = c' \ \text{(Inequality (8.7) and Formula (8.5))}
\end{aligned}$$

□

Lemma 8.3.2. *If A_c is c-proper with respect to an exchange rate sequence R, then for any $c' \geq c$, $A_{c'}$ is c'-proper with respect to R.*

Proof. This lemma comes from Lemma 3 in [28]. Let s_i and s'_i are amount of dollars converted on day/step i by A_c and $A_{c'}$, respectively. Following formula (8.5), we have

$$s_1 - s'_1 = \frac{p_1}{p_1 - m}\left(\frac{1}{c} - \frac{1}{c'}\right) \geq 0$$

$$s_i - s'_i = \frac{p_i - p_{i-1}}{p_i - m}\left(\frac{1}{c} - \frac{1}{c'}\right) \geq 0, \forall i \geq 2$$

Therefore, $\sum_i s'_i \leq \sum_i s_i \leq 1$, i.e. $A_{c'}$ satisfies the first condition of *c'-proper*. Moreover, from Lemma 8.3.1 it follows that $A_{c'}$ achieves a competitive ratio c' with respect to R, satisfying the second condition of *c'-proper*. □

Lemma 8.3.2 implies the following Corollary.

Corollary 8.3.1. *If c^* is the* maximum *competitive ratio that is achievable by the adversary when the player follows the improved threat-based policy, A_{c^*} is c^*-proper regardless of any actual sequence of exchange rates created by the adversary.*

Indeed, for each sequence R of exchange rates, there exists the smallest competitive ratio c so that A_c is c-proper with respect to R. Since c^* is the *maximum* competitive ratio that is achievable by the adversary, $c^* \geq c$. From Lemma 8.3.2 it follows that A_{c^*} is c^*-proper with respect to R.

The main idea of the following analysis is to find the maximum competitive ratio c^* that is achievable by the adversary when the player follows the threat-based policy. The competitive ratio will then become the competitive ratio of the threat-based policy for the generalized one-way trading problem and will be known by the player since it is computed using only known information: the duration D, the lower bound m and the upper bound function $M(i)$ (cf. Equation 8.3).

Lemma 8.3.3. *For fixed $k > 1$ and p_1, the maximum competitive ratio that the adversary can achieve is*

$$c^{(k)}(p_1) = 1 + \frac{p_1 - m}{p_1} \cdot (k-1)\left(1 - \left(\frac{p_1 - m}{M(k) - m}\right)^{1/(k-1)}\right) \tag{8.9}$$

The maximum is achieved when

$$p_k = M(k) \; and \; \frac{p_i - p_{i-1}}{p_i - m} = 1 - \left(\frac{p_1 - m}{M(k) - m}\right)^{1/(k-1)}, \forall i \in [2, k]. \tag{8.10}$$

Proof. This lemma comes from Lemma 5 in [28]. Since the player spends his dollars only on the increasing sequence $p_1, p_2, \cdots, p_k$, we have $\sum_{i=1}^{k} s_i = 1$. Replace s_i using formula (8.5), we obtain

$$\frac{1}{c}\frac{p_1 - cm}{p_1 - m} + \frac{1}{c}\sum_{i=2}^{k}\frac{p_i - p_{i-1}}{p_i - m} = 1$$

which conduces towards a formula for c

$$c = 1 + \frac{p_1 - m}{p_1} \cdot \sum_{i=2}^{k} \frac{p_i - p_{i-1}}{p_i - m} \tag{8.11}$$

On the other hand,

$$\begin{aligned}
\sum_{i=2}^{k} \frac{p_i - p_{i-1}}{p_i - m} &= \sum_{i=2}^{k} \left(1 - \frac{p_{i-1} - m}{p_i - m}\right) = k - 1 - \sum_{i=2}^{k} \frac{p_{i-1} - m}{p_i - m} \\
&\leq k - 1 - (k-1) \left(\prod_{i=2}^{k} \frac{p_{i-1} - m}{p_i - m}\right)^{1/(k-1)} \quad \text{(geometric-arithmetic mean inequality)} \\
&= (k-1) \left(1 - \left(\frac{p_1 - m}{p_k - m}\right)^{1/(k-1)}\right)
\end{aligned}$$

Equality occurs if and only if $\frac{p_{i-1}-m}{p_i-m} = \left(\frac{p_1-m}{p_k-m}\right)^{1/(k-1)} \forall i \in [2, k]$

Applying the inequality on Equations 8.11 follows

$$c \leq 1 + \frac{p_1 - m}{p_1} \cdot (k-1) \left(1 - \left(\frac{p_1 - m}{p_k - m}\right)^{1/(k-1)}\right)$$

Since the right side increases with p_k and $p_k \leq M(k)$, we have

$$c \leq 1 + \frac{p_1 - m}{p_1} \cdot (k-1) \left(1 - \left(\frac{p_1 - m}{M(k) - m}\right)^{1/(k-1)}\right)$$

The equality occurs when

$$p_k = M(k) \; and \; \frac{p_i - p_{i-1}}{p_i - m} = 1 - \left(\frac{p_1 - m}{p_k - m}\right)^{1/(k-1)}, \forall i \in [2, k].$$

□

Now the adversary needs to find a value for p_1 so as to maximize $c^{(k)}(p_1)$ in Lemma 8.3.3 whereas k is still considered as a constant.

Lemma 8.3.4. *For fixed $k > 1$, there exists a unique number p^* in $[m, M(k)]$ such that $c^{(k)}(p^*) = \max_{p_1} c^{(k)}(p_1)$ and*

$$c^{(k)}(p^*) = \frac{kp^*}{km + (p^* - m)} \tag{8.12}$$

Proof. This lemma comes from Lemma 6 in [28]. Let $u = (p_1 - m)^{1/(k-1)} \geq 0$ and $v = (M(k) - m)^{1/(k-1)} > 0$, Equation 8.9 becomes

$$c^{(k)}(p_1) = 1 + (k-1)\left(\frac{u^{k-1}v - u^k}{p_1 v}\right)$$

The derivative of $c^{(k)}(p_1)$ can be written as follows

$$\frac{dc^{(k)}(p_1)}{dp_1} = -\frac{u^k + mku - m(k-1)v}{p_1^2 v}$$

Let $f(u) = u^k + mku - m(k-1)v$. Since i) $f(u)$ increases with $u \geq 0$ and ii) $f(0) = -m(k-1)v < 0$ as well as $f(v) = v^k + mv > 0$ for all $v > 0, k > 1$, equation $f(u) = 0$ has a unique positive root u^*. Moreover,

$$\frac{d^2c^{(k)}(u^*)}{dp_1} = -\frac{k((u^*)^{k-1} + m)}{p_1^2 v(k-1)(u^*)^{k-2}} < 0$$

Therefore, $c^{(k)}(p_1)$ achieves its maximum at u^* or at $p_1 = p^* = (u^*)^{k-1} + m$.

From $f(u^*) = (u^*)^k + mku^* - m(k-1)v = 0$, we have

$$\frac{u^*}{v} = \frac{m(k-1)}{(u^*)^{k-1} + mk}, \text{or}$$
$$\left(\frac{p^* - m}{M(k) - m}\right)^{1/(k-1)} = \frac{m(k-1)}{p^* + m(k-1)}$$

Replacing p_1 by p^* in Equation 8.9 follows

$$\begin{aligned} c^{(k)}(p^*) &= 1 + \frac{p^* - m}{p^*} \cdot (k-1)\left(1 - \left(\frac{p^* - m}{M(k) - m}\right)^{1/(k-1)}\right) \\ &= 1 + \frac{p^* - m}{p^*} \cdot (k-1)\left(1 - \frac{m(k-1)}{p^* + m(k-1)}\right) \\ &= \frac{kp^*}{km + (p^* - m)} \end{aligned}$$

□

Lemma 8.3.5. *For fixed $k > 1$, against the worst sequence created by the adversary above, amount of dollars exchanged at each step by the improved threat-based algorithm is $s_i' = 1/k, \forall i \in [1, k]$.*

Proof. This lemma comes from Lemma 7 in [28]. The worst k-step sequence $P' = p'_1, \cdots, p'_k$ created by the adversary has the following properties from Lemma 8.3.3 and Lemma 8.3.4.

$$\frac{p'_i - p'_{i-1}}{p'_i - m} = 1 - \left(\frac{p'_1 - m}{M(k) - m}\right)^{1/(k-1)}, \forall i \in [2, k],\ and$$

$$p'_1 = p^*,\ where\ c^{(k)}(p^*) = \frac{kp^*}{km + (p^* - m)}$$

Since $s'_i = \frac{1}{c} \cdot \frac{p'_i - p'_{i-1}}{p'_i - m}, \forall i \in [2, k]$, we have $s'_2 = \cdots = s'_k$. On the other hand,

$$\begin{aligned} s'_1 &= \frac{1}{c} \cdot \frac{p'_1 - cm}{p'_1 - m} \\ &= \frac{1}{c^{(k)}(f^*)} \cdot \frac{p^* - c^{(k)}(p^*)m}{p^* - m} \\ &= \frac{p^* + m(k-1))}{kp^*} \cdot \frac{p^*(p^* - m)}{(p^* - m)(p^* + m(k-1))} \\ &= \frac{1}{k} \end{aligned}$$

Since $\sum_{i=1}^{k} s'_i = 1$, we have $s'_1 = s'_2 = \cdots = s'_k = 1/k$. □

Lemma 8.3.6. *For fixed $k > 1$, $c^{(k)}$ is the unique root, c, of*

$$c = k \cdot \left(1 - \left(\frac{c-1}{\frac{M(k)}{m} - 1}\right)^{1/k}\right) \tag{8.13}$$

Proof. This lemma comes from Lemma 8 in [28]. From Formula (8.5), we have:

$$\begin{aligned} & s'_i = \frac{1}{c^{(k)}} \cdot \frac{p'_i - p'_{i-1}}{p'_i - m}, \forall i \geq 2 \\ \Rightarrow\ & \frac{1}{k} = \frac{1}{c^{(k)}} \cdot \left(1 - \left(\frac{p'_1 - m}{M(k) - m}\right)^{1/(k-1)}\right) \text{ (Lemma 8.3.5 and Lemma 8.3.3)} \\ \Rightarrow\ & c^{(k)} = k\left(1 - \left(\frac{p'_1 - m}{M(k) - m}\right)^{1/(k-1)}\right) \end{aligned}$$

On the other hand, from formula (8.5) and Lemma 8.3.5, we also have:

$$\begin{aligned} \frac{1}{k} = s_1' = \frac{1}{c^{(k)}} \cdot \frac{p_1' - c^{(k)}m}{p_1' - m} \\ \Rightarrow p_1' c^{(k)} - mc^{(k)} = p_1' k - kmc^{(k)} \\ \Rightarrow p_1' = \frac{mc^{(k)}(k-1)}{k - c^{(k)}} \end{aligned}$$

Replacing p_1' in $c^{(k)}$ by the right side of the last equation follows

$$c^{(k)} = k\left(1 - \left(\frac{mk(c^{(k)}-1)}{(k-c^{(k)})(M(k)-m)}\right)^{1/(k-1)}\right) \tag{8.14}$$

Expanding Equation (8.14) follows

$$\begin{aligned} \frac{k-c^{(k)}}{k} = \left(\frac{mk(c^{(k)}-1)}{(k-c^{(k)})(M(k)-m)}\right)^{1/(k-1)} \\ \Rightarrow \frac{k-c^{(k)}}{k}\left(\frac{mk(c^{(k)}-1)}{(k-c^{(k)})(M(k)-m)}\right) = \left(\frac{mk(c^{(k)}-1)}{(k-c^{(k)})(M(k)-m)}\right)^{k/(k-1)} \\ \Rightarrow \left(\frac{c^{(k)}-1}{\frac{M(k)}{m}-1}\right)^{1/k} = \left(\frac{mk(c^{(k)}-1)}{(k-c^{(k)})(M(k)-m)}\right)^{1/(k-1)} \end{aligned}$$

Replacing the right side by the left side of the last equation in Equation (8.14), we obtain:

$$c^{(k)} = k \cdot \left(1 - \left(\frac{c^{(k)}-1}{\frac{M(k)}{m}-1}\right)^{1/k}\right)$$

Let

$$f(c) = c + k\left(\frac{c-1}{\frac{M(k)}{m}-1}\right)^{1/k} - k$$

Since i) $f(c)$ is an increasing function with $c \geq 1$ and ii) $f(1) = 1 - k < 0$ as well as $\lim_{c\to+\infty} f(c) > 0$ due to $k > 1, M(k) > m$, equation $f(c) = 0$ has a unique root $c \geq 1$. □

Up to this point, we have proved that for a fixed k, the maximum competitive ratio with respect to k that the adversary can achieve is the unique root of Equation (8.13). Therefore, the maximum competitive ratio achievable by the adversary for the whole game with a known duration D, where k is any value in range $[1, D]$, is the maximum root of Equation (8.13), where $k = 1, \cdots, D$.

Corollary 8.3.2. *The maximum competitive ratio c^* achievable by the adversary for the whole game with a known duration D is*

$$c^* = \max_{k=1\cdots D} \left\{ c \;\middle|\; c = k\left(1 - \left(\frac{c-1}{\frac{M(k)}{m} - 1}\right)^{1/k}\right) \right\} \tag{8.15}$$

For each D given, we always find the ratio c^* that satisfies Equation (8.15) by simply computing c for each $k = 1, \cdots, D$ and then choose the maximum c as c^*. Since A_{c^*} is c^*-proper regardless of any actual sequence of exchange rates created by the adversary (cf. Corollary 8.3.2), the player that uses the algorithm A_{c^*} is guaranteed to achieve the competitive ratio c^* regardless of any actual sequence of exchange rates created by the adversary.

Theorem 5. *The maximum competitive ratio c^* achievable by the adversary is the lowest possible competitive ratio for the one-way trading game with a time-decreasing upper bound.*

Proof. The proof is inspired by the proof of Theorem 5 in [28]. Let ALG be any (deterministic) algorithm.

The adversary behaves as follows. Let k^* is the k that corresponds to c^* in Equation (8.15). Let the worst sequence of exchange rates be $R' = p'_1, p'_2, \cdots, p'_{k^*}$, $m_{k^*+1}, \cdots, m_D$, where p'_i are computed as those in the worst k-step sequence P' in the proof of Lemma 8.3.5. This also implies $p'_i < p'_{i+1}, i = 1, \cdots, k^* - 1$.

At the first step, the adversary presents exchange rate p'_1 to ALG. If ALG spends less than $1/k^*$ at this rate, i.e. $s_1 < 1/k^*$, the adversary drops the exchange rate to the minimum m and keep it there until the end of game. Since p'_1 was chosen so that $1/k^*$ is the minimal amount that need to be convert so as to keep the competitive ratio c^* even if the adversary drops the exchange rate to the minimum (cf. Lemma 8.3.5), ALG with $s_1 < 1/k^*$ cannot achieve the competitive ratio c^*.

If ALG spends more than $1/k^*$ at p'_1, the adversary presents p'_2 to ALG in the next step. At each step $i = 2, \cdots, k^*$, if amount of dollars ALG has exchanged so far is smaller than i/k^*, the adversary stops (i.e., drops the exchange rate to the minimum and keeps it there until the end of game). Otherwise, she presents the next exchange rate p'_{i+1} to ALG. Clearly the adversary will stop at a step $j \leq k^*$ since the trader's initial wealth is one dollar. We have

$$\sum_{i=1}^{j-1} s_i \geq \frac{j-1}{k^*}$$

$$\sum_{i=1}^{j} s_i < \frac{j}{k^*}$$

ALG could have achieved a better competitive ratio by exchanging $1/k^*$ dollars at each step $i = 1, \cdots, j-1$ and exchanging $s_j^- = s_j + (\sum_{i=1}^{j-1} s_i - \frac{j-1}{k^*})$ at a higher exchange rate p'_j. Even in the case that ALG exchanged s_j^- at the rate p'_j, ALG could not achieve the competitive ratio c^*. This is because $s_j^- = \sum_{i=1}^{j} s_i - \frac{j-1}{k^*} < \frac{j}{k^*} - \frac{j-1}{k^*} = \frac{1}{k^*}$ and p'_j was chosen so that $\frac{1}{k^*}$ is the minimal amount of dollars to exchange at p'_j in order to ensure the competitive ratio c^* even if the adversary drops the exchange rate to the minimum and keep it there until the end of game.

That means ALG achieves the competitive ratio c^* only if ALG keeps $s_1 = s_2 = \cdots = s_{k^*} = \frac{1}{k^*}$, otherwise ALG will end up with a higher competitive ratio. □

Theorem 5 implies the following corollary

Corollary 8.3.3. *The threat-based algorithm A_{c^*} is an optimal competitive algorithm for the one-way trading problem with time-decreasing upper bound.*

Figure 8.3 shows the competitive ratio c with corresponding k for each value of D (rows Improved TBP c^* and Improved TBP k), which result from applying the improved threat-based policy to the freshness problem, that is, $M(t) = M/t$ and $m = M/D$. The competitive ratios c^* of the improved TBP turn out to be better than in the original threat-based policy (row Original TBP) and are very close to the lower bounds. Note that the discrepancy between the proved lower bound and the algorithmic result for small D comes from the fact that our lower bound refers to the continuous time model, so there is no contradiction. This discretization effect apparently disappears with growing D as expected.

	Value of D							
	2	4	8	16	32	64	128	$\cdots$
Lower bound $\mathcal{C}$	1.12	1.26	1.40	1.56	1.72	1.89	2.07	$\cdots$
Corresponding s	0.29	0.29	0.29	0.28	0.28	0.28	0.28	$\cdots$
Original TBP c	1.17	1.48	1.85	2.28	2.75	3.25	3.77	$\cdots$
Improved TBP c^*	1	1.17	1.33	1.52	1.73	1.99	2.25	$\cdots$
Corresponding k	1	2	2	3	3	4	5	$\cdots$

Figure 8.3: Numerical comparison of competitive ratios among different algorithms. The last row shows values of k corresponding to the ratios c in the improved TBP.

8.4 Conclusions

We have extended the set of practical issues that can be transformed to one-way trading by presenting new one-way trading models. Unlike the available models, they either limit the change speed of exchange rates or allow the bounds of exchange rates to vary with time. For the new models, we first proved a lower bound that is asymptotically optimal subject to a small constant factor. We have also presented an optimal competitive algorithm against a stronger adversary, where the maximum possible exchange rate decreases with time and the minimum is constant. The practicality of the new models is demonstrated by their use for the freshness problem of concurrent data objects.

Chapter 9

Conclusions and Future Research

In this thesis, we have proposed and developed a new approach for designing reactive concurrent data structures and algorithms. The approach does not require experimentally tuned thresholds nor probability distributions of inputs as previous reactive concurrent data structures in the literature do. Instead, to deal with the uncertainty, we have successfully synthesized non-blocking synchronization techniques and on-line algorithmic techniques, in the context of reactive concurrent data structures. Based on the approach, we have successfully developed fundamental concurrent data structures like trees, multi-word compare-and-swap and locks into reactive ones that efficiently adapt their size or algorithmic behavior to the contention in the system. Moreover, we have improved the applicability of our approach by developing new models for the one-way trading problem. We have also provided optimal solutions for these models. The new models extend the set of practical problems that can be transformed to the one-way trading so as to find an optimal solution. We have used the new models to provide an optimal solution for the freshness problem in the context of concurrent data structures.

In the future, we will continue researching and developing reactive non-blocking synchronization techniques.There are many fundamental concurrent data structures used in synchronization that we would like to look at and develop into reactive ones. On the other hand, we will research and develop practical online models for reactive non-blocking synchronization. Competitive analysis, which compares the performance of online algorithms to an optimal offline algorithm, is too conservative in practice. The drawback of the competitive analysis can be eliminated using either weaker adversary models (e.g. statistical adversary [19]) or risk management models [3] in which the online player forecasts what will happen and makes decision based on that forecast. If her forecast is correct, she will benefit from it. Otherwise, she can control her risk of performing too poorly. Regarding our re-

search, even though variation in execution environments is unpredictable, reactive protocols can forecast what will happen with high confidence, especially when they run at operating systems level. At that level, protocols can collect more information about processes and have more control on them.

Bibliography

[1] Y. Afek, D. Dauber, and D. Touitou. Wait-free made fast. In *Proc. of ACM Symp. on Theory of Computing (STOC)*, pages 538–547, 1995.

[2] A. Agarwal and M. Cherian. Adaptive backoff synchronization techniques. In *Proc. of the Annual Intl. Symp. on Computer Architecture*, pages 396–406, 1989.

[3] S. al Binali. The competitive analysis of risk taking with applications to online trading. In *Proc. of the IEEE Symp. on Foundations of Computer Science (FOCS)*, pages 336–344, 1997.

[4] J. H. Anderson, Y.-J. Kim, and T. Herman. Shared-memory mutual exclusion: Major research trends since 1986. *Distributed Computing*, 16(2-3):75–110, 2003.

[5] J. H. Anderson and M. Moir. Universal constructions for large objects. In *Proc. of the Intl. Workshop on Distributed Algorithms*, pages 168–182, 1995.

[6] J. H. Anderson, S. Ramamurthy, and R. Jain. Implementing wait-free objects on priority-based systems. In *Proc. of ACM Symp. on Principles of Distributed Computing (PODC)*, pages 229–238, 1997.

[7] T. E. Anderson. The performance of spin lock alternatives for shared-money multiprocessors. *IEEE Trans. Parallel Distrib. Syst.*, 1(1):6–16, 1990.

[8] J. Aspnes, M. Herlihy, and N. Shavit. Counting networks and multi-processor coordination. In *Proc. of ACM Symp. on Theory of Computing (STOC)*, pages 348–358, 1991.

[9] J. Aspnes, M. Herlihy, and N. Shavit. Counting networks. *J. ACM*, 41(5):1020–1048, 1994.

[10] H. Attiya and A. Fouren. Adaptive wait-free algorithms for lattice agreement and renaming. In *Proc. of ACM Symp. on Principles of Distributed Computing (PODC)*, pages 277–286, 1998.

[11] H. Attiya, R. Guerraoui, and P. Kouznetsov. Computing with reads and writes in the absence of step contention. In *Proc. of the Intl. Symp. on Distributed Computing (DISC)*, pages 122 – 136, 2005.

[12] H. Attiya, F. Kuhn, M. Wattenhofer, and R. Wattenhofer. Efficient adaptive collect using randomization. *Proc. of the Intl. Symp. on Distributed Computing (DISC)*, pages 159–173, 2004.

[13] H. Attiya and J. L. Welch. Sequential consistency versus linearizability. *ACM Trans. Comput. Syst.*, 12(2):91–122, 1994.

[14] G. Barnes. A method for implementing lock-free shared-data structures. *Proc. of the ACM Symp. on Parallel Algorithms and Architectures (SPAA)*, pages 261–270, 1993.

[15] D. Besedin. Detailed platform analysis in rightmark memory analyzer. part 6 - intel xeon. In *http://www.digit-life.com/articles2/rmma/rmma-nocona.html*, 2005.

[16] A. Borodin and R. El-Yaniv. Online computation and competitive analysis. *Cambridge University Press*, 1998.

[17] E. A. Brewer, C. N. Dellarocas, A. Colbrook, and W. E. Weihl. Proteus: A high-performance parallel architecture simulator. Technical report, Massachusetts Institute of Technology, Cambridge, MA, USA, 1991.

[18] J. Cho and H. Garcia-Molina. Synchronizing a database to improve freshness. In *SIGMOD '00: Proceedings of the 2000 ACM SIGMOD international conference on Management of data*, pages 117–128, 2000.

[19] A. Chou, J. Cooperstock, R. El-Yaniv, M. Klugerman, and T. Leighton. The statistical adversary allows optimal money-making trading strategies. In *Proc. of the ACM-SIAM Symp. on Discrete Algorithms (SODA)*, pages 467–476, 1995.

[20] M. Choy and A. K. Singh. Efficient Fault Tolerant Algorithms for Resource Allocation in Distributed Systems. *Proc. of ACM Symp. on Theory of Computing (STOC)*, pages 593–602, 1992.

[21] T. S. Craig. Queuing spin lock algorithms to support timing predictability. In *Proc. of Real-Time Systems Symp.*, pages 148–157, 1993.

[22] D. E. Culler, J. P. Singh, and A. Gupta. Parallel computer architecture: A hardware/software approach. *Morgan Kaufmann Publisher*, 1999.

[23] P. Damaschke, P. H. Ha, and P. Tsigas. Competitive freshness algorithms for wait-free objects. *Technical report CS:2005-18, Chalmers University of Technology, Sweden*, 2005.

[24] P. Damaschke, P. H. Ha, and P. Tsigas. One-way trading with time-varying exchange rate bounds. *Technical report CS:2005-17, Chalmers University of Technology, Sweden*, 2005.

[25] G. Della-Libera and N. Shavit. Reactive diffracting trees. *J. Parallel Distrib. Comput.*, 60(7):853–890, 2000.

[26] R. El-Yaniv. Competitive solutions for online financial problems. *ACM Comput. Surv.*, 30(1):28–69, 1998.

[27] R. El-Yaniv, A. Fiat, R. Karp, and G. Turpin. Competitive analysis of financial games. In *Proc. of the 33rd Symp. on Foundations of Computer Science*, pages 327–333, 1992.

[28] R. El-Yaniv, A. Fiat, R. M. Karp, and G. Turpin. Optimal search and one-way trading online algorithms. *Algorithmica*, 30(1):101–139, 2001.

[29] K. Fraser. Practical lock-freedom. Technical Report UCAM-CL-TR-579, University of Cambridge, Computer Laboratory, 2004.

[30] E. Gafni, M. Merritt, and G. Taubenfeld. The concurrency hierarchy, and algorithms for unbounded concurrency. In *Proc. of ACM Symp. on Principles of Distributed Computing (PODC)*, pages 161–169, 2001.

[31] J. R. Goodman, M. K. Vernon, and P. J. Woest. Efficient synchronization primitives for large-scale cache-coherent multiprocessors. In *Proc. of the Intl. Conf. on Architectural support for programming languages and operating systems(ASPLOS)*, pages 64–75, 1989.

[32] G. Graunke and S. Thakkar. Synchronization algorithms for shared-memory multiprocessors. *IEEE Computer*, 23(6):60–69, 1990.

[33] M. Greenwald. Non-blocking synchronization and system design. *PhD thesis, STAN-CS-TR-99-1624, Stanford University*, 1999.

[34] M. Greenwald. Two-handed emulation: how to build non-blocking implementations of complex data-structures using dcas. In *Proc. of ACM Symp. on Principles of Distributed Computing (PODC)*, pages 260–269, 2002.

[35] M. Greenwald and D. Cheriton. The synergy between non-blocking synchronization and operating system structure. In *Proc. of the USENIX Symp. on Operating Systems Design and Implementation*, pages 123–136, 1996.

[36] R. Guerraoui, M. Herlihy, and B. Pochon. Toward a theory of transactional contention managers. In *Proc. of ACM Symp. on Principles of Distributed Computing (PODC)*, pages 258–264, 2005.

[37] P. H. Ha, M. Papatriantafilou, and P. Tsigas. Self-tuning reactive distributed trees for counting and balancing. In *Proc. of the Intl. Conf. on Principles of Distributed Systems (OPODIS '04), LNCS 3544*, pages 213–228, 2004.

[38] P. H. Ha, M. Papatriantafilou, and P. Tsigas. Reactive spin-locks: A self-tuning approach. In *Proc. of the IEEE Intl. Symp. on Parallel Architectures, Algorithms and Networks (I-SPAN '05)*, pages 33–39, 2005.

[39] P. H. Ha and P. Tsigas. Reactive multi-word synchronization for multiprocessors. In *Proc. of the IEEE/ACM Intl. Conf. on Parallel Architectures and Compilation Techniques (PACT '03)*, pages 184–193, 2003.

[40] P. H. Ha and P. Tsigas. Reactive multi-word synchronization for multiprocessors. *The Journal of Instruction-Level Parallelism*, Special issue with selected papers from PACT'03:http://www.jilp.org/vol6/v6paper3.pdf, 2004.

[41] P. H. Ha, P. Tsigas, M. Wattenhofer, and R. Wattenhofer. Efficient multi-word locking using randomization. In *Proc. of ACM Symp. on Principles of Distributed Computing (PODC)*, pages 249–257, 2005.

[42] T. Harris. A pragmatic implementation of non-blocking linked lists. In *Proc. of the Intl. Symp. on Distributed Computing (DISC)*, pages 300–314, 2001.

[43] T. L. Harris. In *Personal Communication*, August 2002.

[44] T. L. Harris, K. Fraser, and I. A. Pratt. A practical multi-word compare-and-swap operation. In *Proc. of the Intl. Symp. on Distributed Computing (DISC)*, pages 265–279, 2002.

[45] D. Hendler, N. Shavit, and L. Yerushalmi. A scalable lock-free stack algorithm. In *Proc. of the ACM Symp. on Parallel Algorithms and Architectures (SPAA)*, pages 206–215, 2004.

[46] M. Herlihy. Wait-free synchronization. *ACM Transaction on Programming and Systems*, 11(1):124–149, 1991.

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

[48] M. Herlihy, V. Luchangco, and M. Moir. Obstruction-free synchronization: Double-ended queues as an example. In *Proc. of the IEEE Intl. Conf. on Distributed Computing Systems (ICDCS)*, pages 522–529, 2003.

[49] M. Herlihy, V. Luchangco, M. Moir, and I. William N. Scherer. Software transactional memory for dynamic-sized data structures. In *Proc. of ACM Symp. on Principles of Distributed Computing (PODC)*, pages 92–101, 2003.

[50] M. P. Herlihy and J. M. Wing. Linearizability: A correctness condition for concurrent objects. *ACM Transactions on Programming Languages and Systems*, 12(3):463–492, 1990.

[51] H.R.Simpson. Correctness analysis for class of asynchronous communication mechanisms. *Computers and Digital Techniques, IEE Proc.-*, 139(1):35– 49, 1992.

[52] H.R.Simpson. Freshness specification for a class of asynchronous communication mechanisms. *Computers and Digital Techniques, IEE Proc.-*, 151(2):110–118, 2004.

[53] A. Israeli and L. Rappoport. Disjoint-access-parallel implementations of strong shared memory primitives. In *Proc. of ACM Symp. on Principles of Distributed Computing (PODC)*, pages 151–160, 1994.

[54] D. N. Jayasimha. Parallel access to synchronization variables. In *Proc. of the Intl. Conf. on Parallel Processing (ICPP'87)*, pages 97–100, 1987.

[55] A. Kägi and D. B. J. R. Goodman. Efficient synchronization: Let them eat QOLB. In *Proc. of the Annual Intl. Symp. on Computer Architecture (ISCA-97)*, Computer Architecture News, pages 170–180, 1997.

[56] K.-D. Kang, S. H. Son, and J. A. Stankovic. Managing deadline miss ratio and sensor data freshness in real-time databases. *IEEE Transactions on Knowledge and Data Engineering*, 16(10):1200–1216, 2004.

[57] H. D. Karatza. Cache affinity and resequencing in a shared-memory multiprocessing system. *Journal of Systems and Software*, 51(1):7–18, 2000.

[58] A. R. Karlin, K. Li, M. S. Manasse, and S. Owicki. Empirical studies of competitve spinning for a shared-memory multiprocessor. In *Proc. of the ACM Symp. on Operating Systems Principles*, pages 41–55, 1991.

[59] S. Kumar, D. Jiang, J. P. Singh, and R. Chandra. Evaluating synchronization on shared address space multiprocessors: Methodology and performance. In *Proc. of the ACM SIGMETRICS Intl. Conf. on Measurement and Modeling of Computing Systems (SIGMETRICS-99)*, pages 23–34, 1999.

[60] A. Labrinidis and N. Roussopoulos. Exploring the tradeoff between performance and data freshness in database-driven web servers. *The VLDB Journal*, 13(3):240–255, 2004.

[61] L. Lamport. A new solution of dijktra's concurrent programming problem. *Commun. ACM*, 17(8):453–455, 1974.

[62] L. Lamport. On interprocess communication. part ii: Algorithms. *Distributed Computing*, 1(2):86–101, 1986.

[63] L. Lamport. A fast mutual exclusion algorithm. *ACM Trans. Comput. Syst.*, 5(1):1–11, 1987.

[64] J. Laudon and D. Lenoski. The sgi origin: A ccnuma highly scalable server. In *Proc. of the Annual Intl. Symp. on Computer Architecture (ISCA-97)*, pages 241–251, 1997.

[65] W.-S. Li, O. Po, W.-P. Hsiung, K. S. Candan, and D. Agrawal. Engineering and hosting adaptive freshness-sensitive web applications on data centers. In *Proc. of the Intl. Conf. on World Wide Web*, pages 587–598, 2003.

[66] B. Lim. Reactive synchronization algorithms for multiprocessors. *PhD. Thesis, MIT-LCS-TR-664, Massachusetts Institute of Technology*, 1995.

[67] B.-H. Lim and A. Agarwal. Reactive synchronization algorithms for multiprocessors. In *Proc. of the International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 25–35, 1994.

[68] Y. Ling and W. Chen. Measuring cache freshness by additive age. *SIGOPS Oper. Syst. Rev.*, 38(3):12–17, 2004.

[69] N. Linial. Locality in Distributed Graph Algorithms. *SIAM Journal on Computation*, 21(1):193–201, 1992.

[70] S. S. Lumetta and D. E. Culler. Managing concurrent access for shared memory active messages. In *Proc. of the Intl. Parallel Processing Symp. (IPPS)*, page 272, 1998.

[71] N. Lynch, N. Shavit, A. Shvartsman, and D. Touitou. Timing conditions for linearizability in uniform counting networks. *Theor. Comput. Sci.*, 220(1):67–91, 1999.

[72] N. A. Lynch. Upper bounds for static resource allocation in a distributed system. *Journal for Computer and System Sciences*, 23(2):254–278, 1981.

[73] P. Magnussen, A. Landin, and E. Hagersten. Queue locks on cache coherent multiprocessors. In *Proc. of the Intl. Parallel Processing Symp.*, pages 165–171, 1994.

[74] D. Malkhi, M. Merritt, M. K. Reiter, and G. Taubenfeld. Objects shared by byzantine processes. *Distrib. Comput.*, 16(1):37–48, 2003.

[75] V. J. Marathe, W. N. S. III, and M. L. Scott. Adaptive software transactional memory. In *Proc. of the Intl. Symp. on Distributed Computing (DISC)*, pages 354 – 368, 2005.

[76] M. Mavronicolas, M. Papatriantafilou, and P. Tsigas. The impact of timing on linearizability in counting networks. In *Proc. of the Intl. Symp. on Parallel Processing (IPPS)*, pages 684–688, 1997.

[77] J. M. Mellor-Crummey and M. L. Scott. Algorithms for scalable synchronization on shared-memory multiprocessors. *ACM Trans. Comput. Syst.*, 9(1):21–65, 1991.

[78] M. M. Michael and M. L. Scott. Simple, fast, and practical non-blocking and blocking concurrent queue algorithms. In *Proc. of ACM Symp. on Principles of Distributed Computing (PODC)*, pages 267–275, 1996.

[79] M. M. Michael and M. L. Scott. Relative performance of preemption-safe locking and non-blocking synchronization on multiprogrammed shared memory multiprocessors. In *Proc. of the IEEE Intl. Parallel Processing Symp. (IPPS*, pages 267–273, 1997.

[80] M. Moir. Practical implementations of non-blocking synchronization primitives. In *Proc. of ACM Symp. on Principles of Distributed Computing (PODC)*, pages 219–228, 1997.

[81] M. Moir. Transparent support for wait-free transactions. In *Proc. of the Intl. Workshop on Distributed Algorithms*, pages 305–319, 1997.

[82] M. Moir, D. Nussbaum, O. Shalev, and N. Shavit. Using elimination to implement scalable and lock-free fifo queues. In *Proc. of the ACM Symp. on Parallel Algorithms and Architectures (SPAA)*, pages 253–262, 2005.

[83] M. Naor and L. Stockmeyer. What can be computed locally? *SIAM Journal on Computation*, 24(6):1259–1277, 1995.

[84] D. R. O'hallaron. Spark98: Sparse matrix kernels for shared memory and message passing systems. Technical Report CMU-CS-97-178, Computing Science, Carnegie Mellon University, 1997.

[85] E. Pacitti and E. Simon. Update propagation strategies to improve freshness in lazy master replicated databases. *The VLDB Journal*, 8(3-4):305–318, 2000.

[86] Z. Radovic and E. Hagersten. Efficient synchronization for nonuniform communication architectures. In *Proc. of the IEEE/ACM SC2002 Conf.*, page 13, 2002.

[87] W. N. Scherer and M. L. Scott. Nonblocking concurrent data structures with condition synchronization. In *Proc. of the Intl. Symp. on Distributed Computing (DISC)*, pages 174–187, 2004.

[88] M. L. Scott and W. N. Scherer. Scalable queue-based spin locks with timeout. In *Proc. of the ACM SIGPLAN Symp. on Principles and Practices of Parallel Programming*, pages 44–52, 2001. Source code is available at `ftp://ftp.cs.rochester.edu/pub/packages/scalable_synch/PPoPP_01_trylocks.tar.gz`.

[89] L. Sha, R. Rajkumar, and J. P. Lehoczky. Priority inheritance protocols: An approach to real-time synchronization. *IEEE Transactions on Computers*, 39(9):1175–1185, 1990.

[90] C. Shao, E. Pierce, and J. L. Welch. Multi-writer consistency conditions for shared memory objects. In *Proc. of the Intl. Symp. on Distributed Computing (DISC)*, pages 106–120, 2003.

[91] N. Shavit and D. Touitou. Elimination trees and the construction of pools and stacks: preliminary version. In *Proc. of the ACM Symp. on Parallel Algorithms and Architectures (SPAA)*, pages 54–63, 1995.

[92] N. Shavit and D. Touitou. Software transactional memory. *Proc. of ACM Symp. on Principles of Distributed Computing (PODC)*, pages 204–213, 1995.

[93] N. Shavit, E. Upfal, and A. Zemach. A steady state analysis of diffracting trees. *Theory of Computing Systems*, 31(4):403–423, 1998.

[94] N. Shavit and A. Zemach. Diffracting trees. *ACM Trans. Comput. Syst.*, 14(4):385–428, 1996.

[95] N. Shavit and A. Zemach. Combining funnels: a new twist on an old tale:. In *Proc. of ACM Symp. on Principles of Distributed Computing (PODC)*, pages 61–70, 1998.

[96] M. S. Squillante and E. D. Lazowska. Using processor-cache affinity information in shared-memory multiprocessor scheduling. *IEEE Transactions on Parallel and Distributed Systems*, 4(2):131–143, 1993.

[97] E. Styer and G. L. Peterson. Improved Algorithms for Distributed Resource Allocation. *Proc. of ACM Symp. on Principles of Distributed Computing (PODC)*, pages 615–628, 1988.

[98] H. Sundell and P. Tsigas. NOBLE: A non-blocking inter-process communication library. In *Proc. of the Workshop on Languages, Compilers and Run-time Systems for Scalable Computers, LNCS*, 2002.

[99] J. Torrellas, A. Tucker, and A. Gupta. Benefits of cache-affinity scheduling in shared-memory multiprocessors a summary. In *Proc. of the 1993 ACM Sigmetrics Conf.*, pages 272–274, 1993.

[100] P. Tsigas and Y. Zhang. Evaluating the performance of non-blocking synchronization on shared-memory multiprocessors. In *Proc. of the ACM SIGMETRICS Intl. Conf. on Measurement and Modeling of Computer Systems*, pages 320–321, 2001.

[101] P. Tsigas and Y. Zhang. Integrating non-blocking synchronisation in parallel applications: Performance advantages and methodologies. In *Proc. of the ACM Workshop on Software and Performance (WOSP'02)*, pages 55–67, 2002.

[102] J. Turek and D. Shasha. The many faces of consensus in distributed systems. *IEEE Computer*, 25(2):8–17, 1992.

[103] J. Valois. Lock-free data structures. *PhD. Thesis*, 1995.

[104] R. Vaswani and J. Zahorjan. The implications of cache affinity on processor scheduling for multiprogrammed shared memory multiprocessors. In *Proc. of the ACM Symp. on Operating System Principles*, pages 26–40, 1991.

[105] Y. M. Wang, H. H. Wang, and R. C. Chang. Clustered affinity scheduling on large-scale numa multiprocessors. *Journal of Systems and Software*, 36(1):61–70, 1997.

[106] R. Wattenhofer and P. Widmayer. An adaptive distributed counting scheme. In *Proc. of Intl. Colloquium on Structural Information and Communication Complexity (SIROCCO)*, pages 145–157, 1998.

[107] R. Wattenhofer and P. Widmayer. The counting pyramid: an adaptive distributed counting scheme. *J. Parallel Distrib. Comput.*, 64(4):449–460, 2004.

[108] I. William N. Scherer and M. L. Scott. Advanced contention management for dynamic software transactional memory. In *Proc. of ACM Symp. on Principles of Distributed Computing (PODC)*, pages 240–248, 2005.

[109] S. C. Woo, M. Ohara, E. Torrie, J. P. Singh, and A. Gupta. The SPLASH-2 programs: Characterization and methodological considerations. In *Proc. of the Annual Intl. Symp. on Computer Architecture, ACM SIGARCH Computer Architecture News*, pages 24–36, 1995.

[110] J.-H. Yang and J. H. Anderson. A fast, scalable mutual exclusion algorithm. *Distributed Computing*, 9(1):51–60, 1995.

[111] P.-C. Yew, N.-F. Tzeng, and D. H. Lawrie. Distributing hot-spot addressing in large-scale multiprocessors. *IEEE Trans. Comput.*, 36(4):388–395, 1987.