Thesis for the Degree of Licentiate of Philosophy

Reactive Shared Objects for Interprocess Synchronization

Phuong Ha

CHALMERS | GÖTEBORG UNIVERSITY

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

Göteborg, 2004

Reactive Shared Objects for Interprocess Synchronization
Phuong Hoai Ha

©Phuong Ha, 2004.

Technical Report no. 35 L
ISSN 1651-4963
School of Computer Science and Engineering

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

Printed at Chalmers, Göteborg, Sweden, 2004

Reactive Shared Objects for Interprocess Synchronization.

Phuong Hoai Ha

Department of Computing Science
Chalmers University of Technology and Göteborg University.

Abstract

In parallel processing environments such as multiprocessor systems, processes are synchronized using concurrent objects, which allow many concurrent processes to access them at the same time. The performance of these concurrent objects heavily relies on the load conditions of the surrounding environment (e.g. OS, other applications, interconnection network, etc.), which are variant and unpredictable due to indeterministic interferences among processes. Therefore, in order to minimize synchronization cost, a major overhead in parallel applications, we need to construct efficient concurrent objects that can react to the environment variations in order to achieve good performance under all possible system conditions. This thesis presents two new reactive shared objects: the *reactive multi-word compare-and-swap object* and the *self-tuning reactive tree*.

The *multi-word* compare-and-swap objects are powerful constructs, which make the design of concurrent data objects much more effective and easier. The *reactive* multi-word compare-and-swap objects are advanced objects that 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. We present two algorithms that are non-blocking (lock-free), allowing in this way a fast and dynamical behavior.

The self-tuning reactive trees distribute a set of processes to smaller groups accessing different parts of the memory in a coordinated manner, and moreover reactively adjust their size in order to attain efficient performance across different levels of contention. We present a data structure that in an on-line manner balances the trade-off between the tree traversal latency and the latency due to contention at the tree leaves.

The experiments on the SGI Origin2000, a well-known commercial cc-NUMA multiprocessor, showed that the new reactive objects achieve significant improvements compared to previous results.

Keywords: *non-blocking, lock-free, reactive synchronization, online algorithms, practical, distributed data structures, trees.*

List of included papers

This thesis contains the following technical papers:

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

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

3. Phuong Hoai Ha, Marina Papatriantafilou and Philippas Tsigas, " Self-Adjusting Trees". *Technical report no. 2003-09*, Department of Computing Science, Chalmers University of Technology, Gothenburg, 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 honored to have professor Roger Wattenhofer from ETH Zürich as the discussion leader in my Licentiate seminar.

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

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, October 2004.

Contents

List of Figures

Chapter 1

Introduction

Multiprocessor systems aim to support parallel computing environments, where processes are running concurrently. In such parallel processing environments, interferences among processes are inevitable, for instance, many concurrent processes may cause high traffic on the bus/network, high contention on a memory module or high load on a processor. These interferences generate a variant and unpredictable environment to each process. On the other hand, these processes widely interact with each other via shared *concurrent* objects, which allow many concurrent processes to access them at the same time. The performance of these concurrent objects heavily relies on the surrounding environment as well as the number of concurrent accesses. For instance some complex objects like queue-locks are good for high load environments, whereas others like test-and-set locks are good for low load environments [27]. Therefore, it raises a question on constructing objects that can react to the environment variations in order to achieve good performance in all cases. We call such objects *reactive shared objects*. This thesis presents two such objects: the *reactive multi-word compare-and-swap object* and the *self-tuning reactive tree*.

This chapter briefly presents the background for the thesis. The next section describes shared memory multiprocessor systems. Section 1.2 presents synchronization issues and concepts in such multiprocessor systems. Section 1.3 describes some data structures used to alleviate contention on shared objects. Section 1.4 presents an approach to construct high performance shared objects. Section 1.5 presents the contributions of this thesis.

1.1 Shared Memory Multiprocessor Systems

Nowadays, multiprocessor systems are widely spread. They appear not only in supercomputers but also in desktops. Intuitively, a multiprocessor system can be considered as "a collection of processing elements that communicate and cooperate to solve large problems fast" [9]. This description shows the significant role of communication architecture, which can be based on *shared memory* or *message passing*. In shared memory multiprocessor systems, processors communicate by writing and reading shared variables in the common memory address space, which is shared among processors. In message passing systems, the communication is done via sending and receiving messages.

In general, the architecture of shared memory multiprocessor systems can be depicted by Figure 1.1

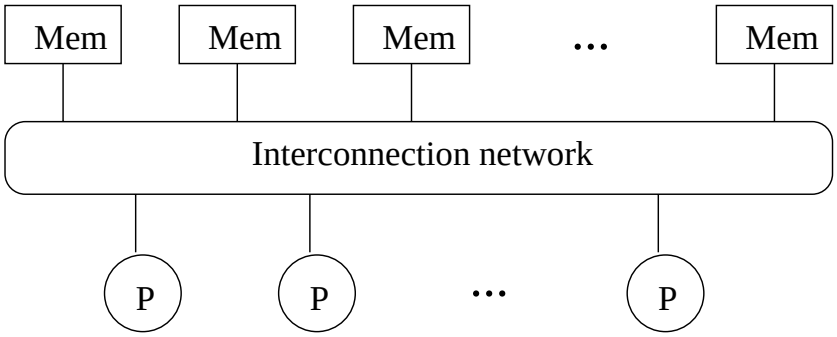


Figure 1.1: The shared memory multiprocessor, where P is a processor and Mem is a memory bank

Depending on the features of interconnection network and the distribution of memory banks, we can classify shared memory multiprocessor systems into two subclasses: uniform memory access (UMA) and non-uniform memory access (NUMA).

1.1.1 Uniform Memory Access (UMA)

In the UMA systems, the latency of accesses from any processor to any memory bank is the same. The interconnection networks often used in UMA systems are of bus and crossbar-switch types.

In bus-based UMA systems as depicted in Figure 1.2, all processors access the memory banks via a central bus and thus the bus becomes a bottleneck when the number of processors increases. Because the bandwidth of the

bus is still fixed when adding more processors to the bus, the average bandwidth for each processor decreases as the number of processors increases. Therefore, the bus-based architecture limits the scalability of these systems.

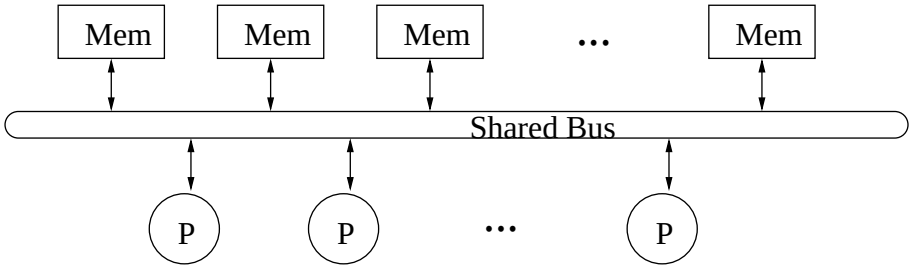


Figure 1.2: The bus-based UMA systems

Unlike bus-based UMA systems, crossbar-switch based UMA systems increase the aggregate bandwidth when adding new processors to the systems. As depicted in Figure 1.3, each processor in such systems has an interconnection to each memory bank. Therefore, bandwidth is not a problem to the crossbar-switch based systems when adding new processors, but the problem is to expand the switch, where the cost increment is quadratic to the number of ports.

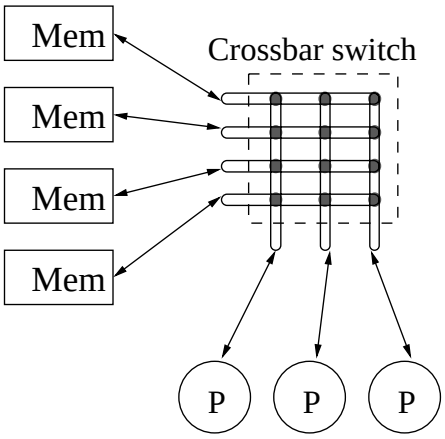


Figure 1.3: The crossbar-switch based UMA systems

Because it is expensive to design scalable UMA systems, an alternative design called non-uniform memory access systems was suggested.

1.1.2 Non-Uniform Memory Access (NUMA)

As its name mentions, the NUMA systems have different latencies from processors to memory banks among the processors. In the systems, the shared memory banks are distributed among processors so that accessing local memory is faster than accessing remote memory. Even though the physical memory banks are distributed, the same virtual address space is shared by the whole system and the memory controllers take care of 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 network 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 are replicated into caches and the systems support means to keep the caches consistent. Such a system is called cache-coherent non-uniform memory access (ccNUMA).

The SGI Origin 2000 [26], which is used for the experiments in this thesis, 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 1.4

1.1.3 Synchronization Primitives and Consensus Numbers

In order to synchronize processes running on the shared memory multiprocessor systems, the systems support some *strong* synchronization primitives. The concept "*consensus number*" is used to determine how strong a synchronization primitive is.

Consensus number: A synchronization primitive with consensus number n can achieve consensus among n processes even if up to $n-1$ processes stop [38].

According to the consensus classification, read/write registers have consensus number 1, i.e. they cannot tolerate any faulty processes. 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

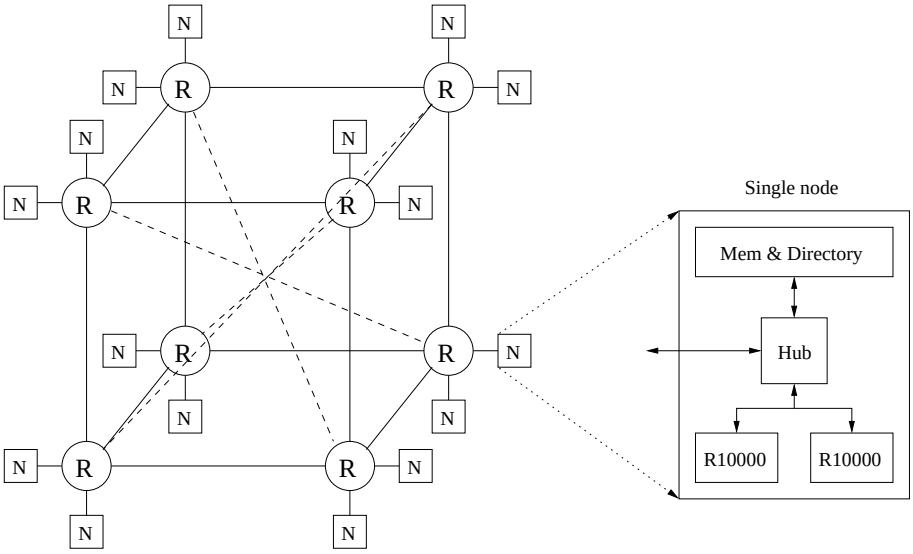


Figure 1.4: The SGI Origin 2000 architecture with 32 processors, where R is a router.

with consensus number 2: *test-and-set (TAS)* and *fetch-and-op (FAO)* and two 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 1.5, where x is a variable, v, old, new are values and *op* can be the operator *add* or *xor*.

For the systems that support *weak LL/SC*[1] 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 [30]. Because both *LL/SC* and *CAS* are primitives with infinite consensus number, or universal primitives, one primitive can be implemented from the other.

Multi-word compare-and-swap operation (CASN): 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*

[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.

```
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 1.5: Synchronization primitives

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 [15, 13, 14].

1.2 Synchronization

In shared memory multiprocessor systems, processes communicate by writing and reading shared variables or shared objects. In order to guarantee the consistency of the object state/value, the conventional method is to use

mutual exclusion, which allows only one process to access the object at one time.

Figure 1.6 illustrates a scenario where the shared object value becomes inconsistent due to parallel processing. The procedure *increment*() is used to count how many processes that have run so far and each process calls the procedure at the beginning. The figure shows points in time when events are executed on process 1 and process 2, where process 2 starts after process 1. The result that process 2 gets is 0, which is incorrect because there is already a process, process 1, starting before process 2. This problem can be solved by using procedure *increment′*() instead of *increment*(). The procedure *increment′*() uses a *lock* to protect the shared variable *counter*: only one process that has successfully acquired the lock can access the shared variable. All other processes have to wait until the lock is released. The method using locks to protect the critical section is called mutual exclusion.

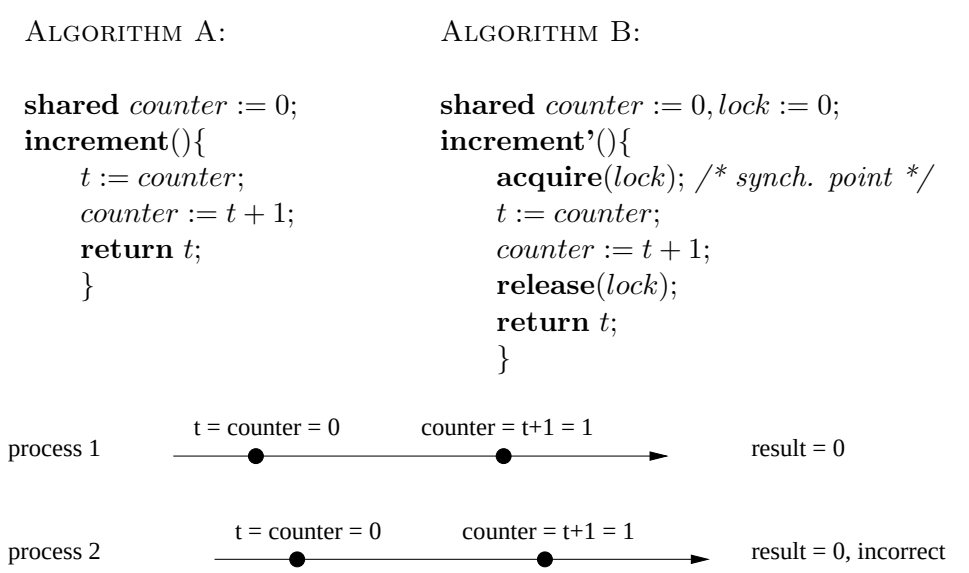


Figure 1.6: An inconsistency scenario and its solution using mutual exclusion

1.2.1 Mutual Exclusion

The purpose of mutual exclusion methods is to resolve conflicting accesses to shared objects. A program using mutual exclusion normally consists of 4

sections:

Noncritical section: contains the code not accessing the shared objects.

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 leaves the critical section.

Critical section: contains the code used when accessing the shared objects.

Exit section: contains the code to inform other waiting processes that the winner has left the critical section.

Therefore, the mutual exclusion algorithm consists of two sections *Entry* and *Exit*. All mutual exclusion algorithms must satisfy the two following requirements [2]:

Exclusion : only one process can enter the critical section at one 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, the waiting process yields the processor to other processes to execute useful tasks, but the system has to pay the context-switching cost to switch the waiting process from the running state to the suspended state. In busy-waiting mode, the waiting process continuously spins on the lock protecting the critical section so that it can acquire the lock as soon as the lock is released. However, the busy-waiting protocol wastes processor time for a useless task: spinning on the lock. Therefore, an efficient spin-lock algorithm should be in spinning mode when the waiting time is short and should switch to blocking mode when the waiting time is long. However, the waiting time, the time the lock will be held, is normally unknown in prior to the decision point. There are many research results on this problem [23, 1, 5, 24], especially Karlin et al. [24] presented five competitive strategies for determining whether and how long to spin before blocking.

In systems where processes are executed on different processors, the blocking mode does not seem necessary, i.e. context switching is not necessary. However, spinning may cause another problem if the protocol continuously executes read-modify-write operations on the lock, as it generates a lot of network traffic. A simple implementation of a spin-lock is depicted in Figure 1.7. This implementation will generate a lot of network traffic because the TAS primitive requires a "write" permission and even if it fails to update the variable *lock*, it also causes a *read-miss* in cache-coherent multiprocessor systems. All processors with the read-miss will concurrently access the variable to read its unchanged value again, causing a burst on the memory module containing the variable and the memory bus. Therefore, continuously executing TAS inside the while-loop will generate very high load on the memory module and thus will slow down all accesses to other non-conflicting variables on this memory module as well as to the interconnection network.

---

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

Figure 1.7: Spin-lock using *test-and-set*

Test-and-test-and-set: In order to reduce the surge load caused by TAS, another implementation is suggested, which is depicted in Figure 1.8. The difference between the two implementations is that the latter executes TAS only if the lock is available, i.e. $lock \neq 1$. In cache-coherent multiprocessor systems, the *lock* variable will be cached at each processor and thus reading the variable does not cause any network traffic. This change significantly improves performance of the spin-lock.

---

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

Figure 1.8: Spin-lock using *test-and-test-and-set*

Backoff: However, continuously reading the cached variable still causes a surge load on the memory module containing the variable when its value is changed. When the lock is released, all other spinning processors will *concurrently* try to execute TAS on the variable, causing a huge network traffic like the spin-lock using *test-and-set*. 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 still an interesting issue. Anderson [5] suggested the exponential backoff scheme, where the delay on each processor will be doubled up to a limit every time the processor reads the unavailable lock. The backoff scheme seems to work in practice, but it needs manually tuned parameters for the starting delay and the upper limit of the delay. Inaccurately chosen parameters will significantly affect the performance of the spin-lock.

Queue-lock: Even though the spin-lock protocol has been improved a lot to reduce the network traffic, it still needs to spin on a common lock. Many research results on local spin-locks have been published, and the most practical ones are queue-locks [5, 12, 28]. The idea is that each processor spins on a separate variable and thus 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 will inform the first processor in the waiting queue by writing a value to the processor's spin-variable. Because only one processor is informed that the lock is available, no conflict among waiting processors occurs when releasing the lock.

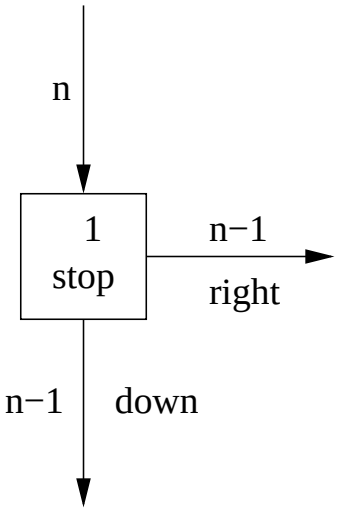
Algorithms using only Read/Write: All aforementioned mutual exclusion algorithms exploit strong synchronization primitives such as TAS, FAO and CAS. Another major research trend on mutual exclusion is to design mutual exclusion algorithms using only *Read* and $Write$ operations. The trend was initiated by Lamport's fast mutual exclusion algorithm [25], which then was generalized to the *splitter* element [2]. Figure 1.9 describes the splitter and its implementation. The 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.

Drawbacks: Even though many researchers have tried to make the mutual exclusion algorithm efficient, the concept of lock-based synchronization itself contains inevitable drawbacks:

Risk for deadlock: If the process that is keeping the lock suddenly crashes or cannot proceed to the point where it releases the lock, all other processes waiting for the lock to be released will wait forever.

Risk for lock convoy: In lock-based synchronization, one process slowing down can make the whole system consisting of many processors slow down. If the process that is keeping the lock is delayed or preempted,




```
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 1.9: The splitter element

other processes waiting for the lock have to suffer the delay or the preemption even though they may run on other independent and fast processors.

Risk for 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 acquires the lock and thus T_1 will be 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.

Because of the drawbacks of the lock-based synchronization, an alternative called *non-blocking synchronization* was suggested and introduced major research problems.

1.2.2 Non-Blocking Synchronization

To address the inherent drawbacks of locking, researchers have proposed *non-blocking algorithms* for implementing 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.

A universal methodology to construct a lock-free implementation from the sequential code of a shared object was suggested in [20]. Each process generates a copy of the shared object, manipulates on the copy and eventually makes the copy become the current shared object if there is no interference with other processes on the object since the time it generated the copy. If there was interference, the update will fail and the process has to start the procedure again until the update succeeds. The update normally uses universal primitives such as *compare-and-swap* or *load-linked/store-conditional*.

However, there is a risk for starvation as the progress of other operations could cause one specific operation to never finish.

Wait-free algorithms [19] 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.

A technique called *helping* [19] is often used to avoid starvation and achieve waitfreeness. Briefly, each process announces what it will be doing on the shared object and before executing its own operations, it has to check if there are any announced operations that have not completed yet. If so, it has to help the pending operations finish first.

The disadvantages of the universal methodology are the high cost for copying large objects and the loss of disjoint-access-parallelism. The *disjoint-access-parallelism* [22] means that processes accessing no common portion

of the shared data should be able to progress in parallel. Therefore, many research results have been on constructing efficient lock-free/wait-free implementations for specific data structures [39, 29, 22, 18, 3, 32, 31].

Linearizability : In non-blocking synchronization, many object operations can be executed concurrently, so it is more complicated to reason about the correctness of a design than in the lock-based ones, where the object operations are obviously sequential according to the order of the lock acquirements. Therefore, the concept of *linearizability* [21] is used as the correctness condition for non-blocking objects. Basically, the *linearizability* requires that the object operation appears to take effect instantaneously at some point in time between its invocation and its response. Therefore, given a history of concurrently executed operations on the shared object, we can always find a sequential execution of the operations that generates the same result.

Non-blocking algorithms have been shown to be of big practical importance [36, 37], and recently NOBLE, which is a non-blocking inter-process communication library, has been introduced [35].

1.3 Counting and Balancing Networks

As we have seen above, the performance of operations on the shared objects is much improved if the contention on those objects is not too high. With the idea in mind, many researchers have focused on constructing some special data structures that can alleviate the contention on the shared objects by evenly distributing processes into small groups, each of which accesses different objects in a coordinated manner.

Many counting and balancing networks [6, 10, 16, 34] aim at the multiprocessor coordination problems. The counter is a nice illustration for the coordination problems. Aspnes et al. presented two counting networks called *Bitonic counting network* and *Periodic counting network* in [6]. The bitonic counting network $Bitonic[k]$ depicted in Figure 1.10 is inductively built from two components $Bitonic[k/2]$ and one component $Merger[k]$, where $Bitonic[1]$ directly passes its inputs to its outputs. $Merger[k]$ is inductively built from two $Merger[k/2]$ and $k/2$ balancer B, where $Merger[2]$ consists of one balancer. The balancer is a toggle mechanism, which, given a stream of input tokens, repeatedly sends one token to the top output and one to the bottom output. The top (bottom) $Merger[k/2]$ receives $k/2$ even (odd) numbered outputs from the top $Bitonic[k/2]$ and $k/2$ odd (even)

numbered outputs from the bottom $Bitonic[k/2]$, respectively. Each pair of the i^{th} outputs from the top $Merger[k/2]$ and the bottom $Merger[k/2]$ are inputs of a balancer.

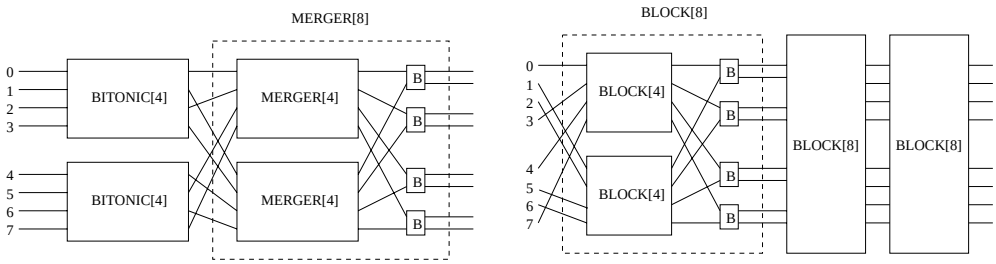


Figure 1.10: (A): The bitonic counting network with 8 inputs/outputs, $Bitonic[8]$; (B): The periodic counting network, $Periodic[8]$. B is the balancer

The periodic counting network $Periodic[k]$ depicted in Figure 1.10 consists of $log(k)$ $Block[k]$ networks, where the i^{th} output of one is the i^{th} input of the next. Each $Block[k]$ network is inductively built from two $Block[k/2]$ and $k/2$ balancers. Given a sequence, input of the top $Block[k/2]$ is the subsequence whose indices have two low-order bits 00 or 11. Similarly, input of the bottom $Block[k/2]$ is the subsequence whose indices have two low-order bits 01 or 10. Each pair of the i^{th} outputs from the top $Block[k/2]$ and the bottom $Block[k/2]$ are inputs of a balancer.

Shavit and Zemach presented counting networks called *Diffracting trees* [34]. Wattenhofer and Widmayer also proved a lower bound on the number of messages that processors must exchange in a distributed asynchronous counting mechanism, where the counter is increased by broadcasting a request message [40]. Since Chapter 3 of this thesis relates to diffracting trees, we focus on this data structure here.

Diffracting trees [34] are well-known distributed data structures. Their most significant advantage is the ability to distribute a set of concurrent processes to many small groups locally accessing shared data in a coordinated manner. Each process(or) accessing the tree can be considered as leading a *token* that follows a path from the root to the leaves. Each node is a computing element receiving tokens from its single input (coming from its parent node) and sending out tokens to its outputs; it is called *balancer* and acts as a *toggle mechanism* which, given a stream of input tokens, alternately forwards them to its outputs, from left to right (sending them to

the left and right child nodes, respectively). The result is an even distribution of tokens at the leaf nodes. Diffracting trees have been introduced for *counting-problems*, and hence the leaf nodes are counters, assigning numbers to each token that exits from them.

However, the fixed-size diffracting tree is optimal only for a small range of contention levels. To solve this problem Della-Libera and Shavit proposed the *reactive diffracting trees*, where each node can shrink (to a counter) or grow (to a subtree with counters as leaves) according to the current load, in order to attain optimal performance [10].

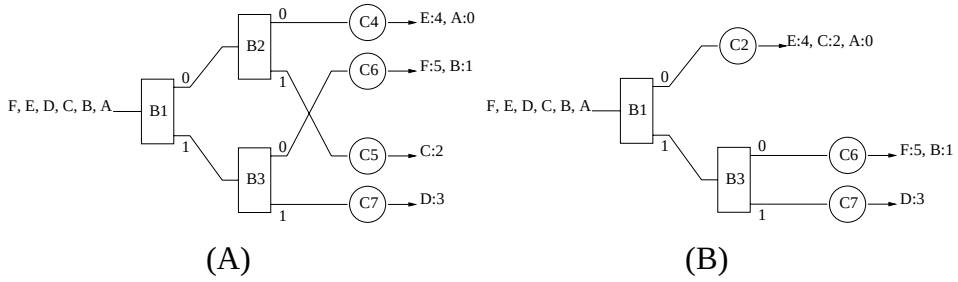


Figure 1.11: Diffracting tree (A) and reactive diffracting tree (B)

Figure 1.11(A) shows an example of a diffracting tree. The set of processors A, B, C, D, E, F is balanced on all counters where counters $C4, C6$ are accessed by only two processors and counters $C5, C7$ by only one processor. Tokens passing one of these counters receive integers i, $i + 4$, $i + 2 * 4$, $\cdots$ where i is initial value of the counter. In tree (A), processors A, B, C, D, E, F receive integers $0, 1, 2, 3, 4, 5$, respectively. Even though the processors access separate shared data (counters), they still receive numbers that form a consecutive sequence of integers as if they would have accessed a centralized counter.

Trees (A) and (B) in Figure 1.11 depict the folding action of a reactive diffracting tree. Assume that at the beginning the reactive diffracting tree has the shape like tree (A). If the loads on two counters $C4$ and $C5$ are small, the sub-tree whose root is $B2$ shrinks to counter $C2$ as in tree (B). If the sequence of processors A, B, C, D, E, F traverse the reactive diffracting tree (B), three processors A, C, E will visit counter $C2$. That is, the latency for processors to go from the root to the counter decreases and the load on counter $C2$ increases.

1.4 Reactive Shared Objects

Multiprocessor systems aim to support parallel computing environments, where processes run concurrently. In such parallel processing environments, interferences among processes are inevitable, for instance, many concurrent processes may cause high traffic on the bus/network, high contention on a memory module or high load on a processor. These interferences generate a variant and unpredictable environment to each process. On the other hand, these processes widely interact with each other via shared *concurrent* objects, which allow many concurrent processes to access them at the same time. The performance of these concurrent objects heavily relies on the surrounding environment as well as the number of concurrent accesses. For instance some complex objects like queue-locks are good for high load environments, whereas others like test-and-set locks are good for low load environments [27]. Therefore, it raises the question of constructing objects that can react to the environment variations in order to achieve good performance in all cases. We call such objects *reactive shared objects*.

A simple example of reactive synchronization is the spin-lock using *test-and-test-and-set* with exponential backoff ($TTSE$) [5]. In the spin-lock, every time a waiting process reads a busy-lock, i.e. there is probably high contention, it will double its delay in order to reduce contention. Five competitive lock-constructing strategies for determining whether and how long to spin before blocking by Karlin et al. [24] are also reactive. A reactive spin-lock that can switch from spin-lock using $TTSE$ to a complex local-spin queue-lock when the contention is considered high was suggested by Lim [27]. As mentioned in Section 1.3, Della-Libera and Shavit proposed the *reactive diffracting trees*, where each node can shrink (to a counter) or grow (to a subtree with counters as leaves) according to the current load, in order to attain optimal performance [10].

However, most of these reactive policies rely on either some manually tuned thresholds or known probability distributions of some unpredictable inputs. The backoff scheme in the $TTSE$ algorithm requires two manually tuned parameters called *base delay* and *delay upper limit*. Karlin's competitive strategies in [24] rely on either $SpinThreshold$, which may be tuned using process context-switching time on a specific system, or a probability distribution D_L of lock-waiting time. Lim's reactive spin-lock [27] relies on two thresholds called tts_retry_limit and $queue\_empty\_limit$ to switch the current protocol from $TTSE$ to queue-lock and vice versa. Lim's reactive waiting algorithm [27] relies on a waiting time distribution. The reactive diffracting trees [10] rely on three manually tuned parameters, namely

folding/unfolding thresholds and the time-intervals for consecutive reaction checks.

Unfortunately, these parameter values depend on the multiprocessor system in use, the applications using the objects and, in a multiprogramming environment, on the system utilization by the other programs that run concurrently. The programmer has to fix these parameters manually, using experimentation and information that is commonly not easily available (future load characteristics). The *fixed* parameters cannot support good reactive schemes in *dynamic* environments such as multiprocessor systems, where the traffic on buses, the contention on memory modules, and the load on processors are unpredictable to one process. For those relying on assumed distributions, the algorithms use too strong assumptions. Generally, the inputs in the dynamic environment like waiting time distributions for a shared object are unpredictable.

Ideally, reactive algorithms should be able to observe the changes in the environment and react accordingly. Based on that purpose, online algorithms and competitive analysis seem to be a promising approach.

1.4.1 Online Algorithms

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 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 *competitive ratio* is used.

Competitive ratio : An online algorithm ALG is considered *c-competitive* if there exists a constant α such that for any finite input I [8]:

$$ALG(I) \leq c \cdot OPT(I) + \alpha \tag{1.1}$$

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 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 whose processing cost is very expensive for the online algorithm but inexpensive for the optimal offline algorithm.

Adversary : For deterministic online algorithms, the adversary with the knowledge of the online algorithm 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 online player to construct the next input, we classify the adversary into two categories:

Oblivious: the adversary constructs the whole input sequence in advance regardless of the output produced by the online player. A randomized online algorithm is *c-competitive* to an oblivious adversary if

$$E[ALG(I)] \leq c \cdot OPT(I) + \alpha \tag{1.2}$$

where $E[ALG(I)]$ is the expected cost of the randomized online algorithm ALG on the input I.

Adaptive: the adversary observes the output produced by the online player so far and then based on that information constructs the next input element. If the adversary processes the input sequence, which is determined after the online player finished, using an optimal offline algorithm, it is called *adaptive-offline* adversary. A randomized online algorithm is *c-competitive* to an adaptive-offline adversary if

$$E[ALG(I) - c \cdot OPT(I)] \leq \alpha \tag{1.3}$$

If the adversary processes the input sequence online, it is called *adaptive-online* adversary. A randomized online algorithm is *c-competitive* to an adaptive-online adversary if

$$E[ALG(I) - c \cdot ADV(I)] \leq \alpha \tag{1.4}$$

where $ADV(I)$ is the adversary cost to process input I.

In conclusion, the competitive analysis is 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 that kind of information.

1.5 Contributions

We have accomplished the followings:

- We have designed 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), thereby allowing a fast and dynamical behavior. Experiments on thirty processors of an SGI Origin2000 multiprocessor show that both our algorithms react quickly to contention variations and outperform the best-known alternatives under almost all contention conditions.

- We have developed self-tuning reactive distributed trees, which not only distribute a set of processes to smaller groups accessing different parts of the memory in a coordinated manner, but also reactively adjust their size in order to attain efficient performance across different levels of contention. We present a data structure that in an on-line manner balances the trade-off between the tree traversal latency and the latency due to contention at the tree leaves. Moreover, the fact that our method can expand or shrink a subtree several levels in any adjustment step has a positive effect on the efficiency. This feature helps the self-tuning reactive tree minimize the adjustment time, which affects not only the execution time of the process adjusting the size of the tree, but also the latency of all other processes traversing the tree at the same time. Our experimental study compared the new trees with the reactive diffracting ones on the SGI Origin2000, a well-known commercial ccNUMA multiprocessor. This study showed that the self-tuning reactive trees i) select the same tree depth as the reactive diffracting trees do; ii) perform better and iii) react faster.

Chapter 2

Reactive Multi-word Synchronization for Multiprocessors [1]

Phuong Hoai Ha, Philippas Tsigas
Department of Computing Science
Chalmers Univ. of Technol. and Göteborg Univ.
412 96 Göteborg, Sweden
E-mail: {phuong, tsigas}@cs.chalmers.se

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 the 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 to the* multi-word *objects.*

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

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

2.1 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 [15, 13, 14]. 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 [3, 4, 18, 22, 31, 32]. 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* [3, 18, 22], 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* [31, 32] 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 2.2. Section 2.3 presents our algorithms at an abstract level. The algorithms in detail are described in Section 2.4. Section 2.5 presents the correctness proofs of our algorithms. In Section 2.6 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 2.7 concludes the paper.

2.2 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* [19] 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 [36, 37], and recently NOBLE, which is a non-blocking inter-process communication library, has been introduced [35].

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 portion changed by each process [20]. The disadvantages of Herlihy's methodology are the high cost for copying large objects and the 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 [7] 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 be doing. 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, process p_1 will cooperate with p_2 to complete the work of process p_2.

Israeli and Rappoport transformed this technique into one more applicable in practice in [22], 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 the new values to the portions one by one. An *owner* field is assigned to every portion of the shared data to inform the processes about which process is the owner of the portion at that time.

Harris, Fraser and Pratt [18] 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 [3]. The cooperative technique was employed in the aforementioned results as well.

However, the disadvantage of the *cooperative technique* is that the process, which is blocked by another process, does not release the words it owns when it helps the blocking process, even though many other processes blocked on these words may be able to make progress if these words are 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. Moreover, the longer the response time of p_1, the bigger the number of processes blocked by p_1. The processes blocked by p_1 will first help process p_1 and then continue to help process p_2 even when they and process p_2 access disjoint parts of the data structure. This problem will be solved if process p_1 does not conservatively keep its words and releases them while it is helping the blocking process p_2.

The left part in Figure 2.1 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.

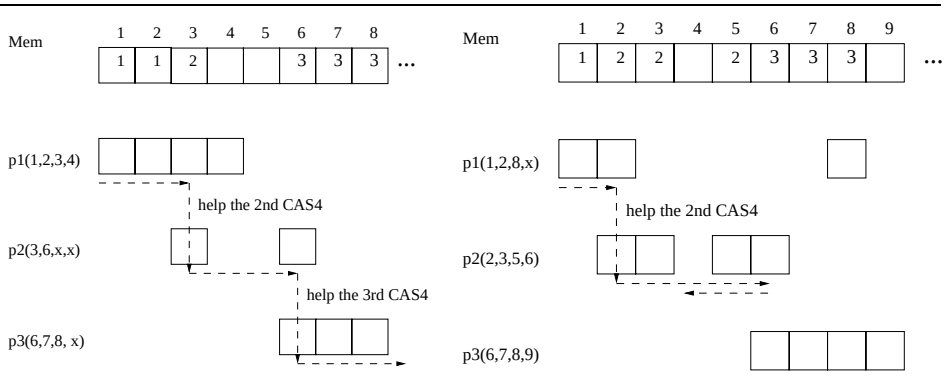


Figure 2.1: Recursive helping policy and software transactional memory

Shavit and Touitou realized the problem above and presented the *software transactional memory* (STM) in [32]. In STM, a process p_1 that was blocked by another process p_2 releases the words it owns immediately before helping blocking process p_2. Moreover, a blocked process helps at most one blocking process, so *recursive helping* does not occur. STM then was improved by Moir [31], 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 2.1 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.

2.2.1 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 2.6, 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.

2.3 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 [3, 18, 22, 31, 32] are based on different strategies that are described in Section 2.2.

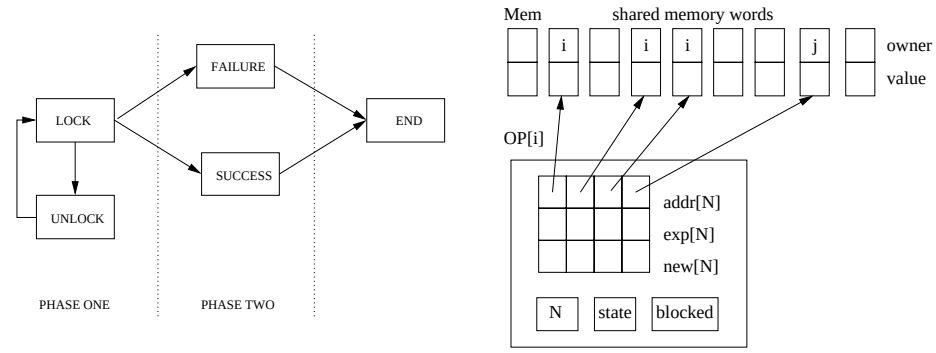


Figure 2.2: Reactive-CASN states and reactive-CAS4 data structure

The variable $OP[i]$ described in Figure 2.2 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 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 [18].

Each CASN operation consists of two phases as described in Figure 2.2. 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 2.3 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 2.3: 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 2.4: The term definitions

2.3.1 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* [11]. 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* [11].

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.

2.3.2 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* [11]. 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 2.3.1. *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 [11], 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 2.3.1, 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 [11], 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 2.3.1. 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.

2.4 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 2.5, 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 [30].

2.4.1 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 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 2.3.1. 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)

```
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 2.5: 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 2.6: Data structures in our first reactive multi-word compare-and-swap algorithm

```
/* 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 2.7: 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 2.8: 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 2.9: Procedures Updating and Unlocking/Releasing in our first reactive multi-word compare-and-swap algorithm

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 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 2.6.

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 2.10. 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.

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

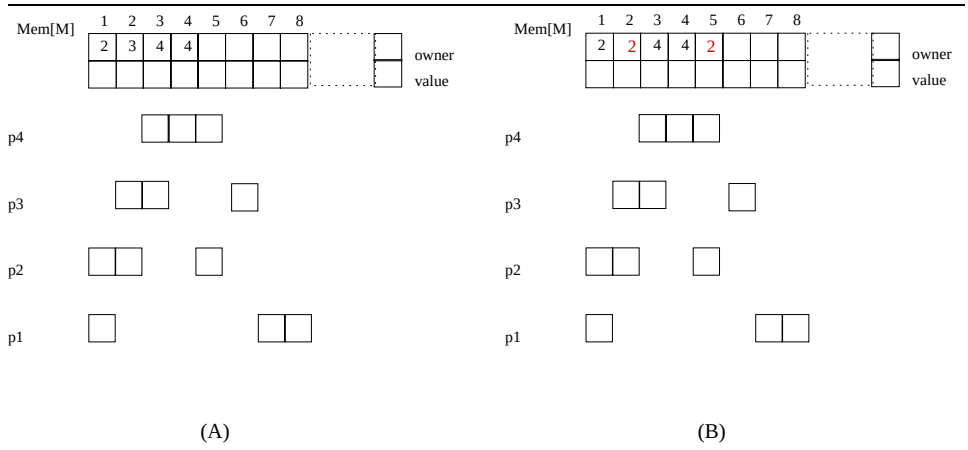


Figure 2.10: Circle-helping problem: (A) Before helping; (B) After p_1 helps $CAS3_2$ acquire $Mem[2]$ and $Mem[5]$.

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 $Help$).

2.4.2 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 2.3.2 , 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 2.11 and Figure 2.12, 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 $OP[i].blocked$[2] 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 2.3.1.

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

[2]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 2.11: 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 2.12: Procedures CheckingR in our second reactive multi-word compare-and-swap algorithm and the procedure for Read operation.

Unlocking. 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 2.11 and Figure 2.12.

2.5 Correctness Proof

In this section, we prove the correctness of our methods. Figure 2.13 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 2.13: 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 2.5.1. *Every helping process reads correct values of variable $OP[i]$.*

Proof. In our pseudo-code described in Figure 2.7, Figure 2.8, Figure 2.9, Figure 2.11 and Figure 2.12, 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[3] is linearizable to RCASN. The *Read* operation is similar to those in [18][22].

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 2.5.2. *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

[3]The *Read* procedure described in figure 2.12 is used in both reactive CASN algorithms

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 $OP[i].state$.

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 2.9) 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 2.8) *before* p_l^2 executes $SC(\&Mem[x], (OP[i].exp[y], nil))$ (line 4 in *Unlocking/ Releasing*, Figure 2.9), 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 2.5.3. *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 2.5.2 and Lemma 2.5.3, we can infer the following corollary:

Corollary 2.5.1. *The interferences among the procedures do not affect the correctness of our algorithms.*

Lemma 2.5.4. *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 2.5.5. *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 [11].

Lemma 2.5.6. *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 2.11. □

Lemma 2.5.6 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 2.5.7. *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. □

2.6 Evaluation

We compared our algorithms to the two best previously known alternatives: i) the lock-free algorithm presented in [22] that is the best representative of the *recursive helping* policy (RHP), and ii) the algorithm presented in [31] that is an improved version of the *software transactional memory* [32] (iSTM). In the latter, a dummy function that always returns zero is passed to CASN. Note that the algorithm in [31] 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 [18], 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 [17], but was not available during the time we performed our experiments. However, relying on the experimental data of the paper [18], 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 [30], 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 2.14 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.

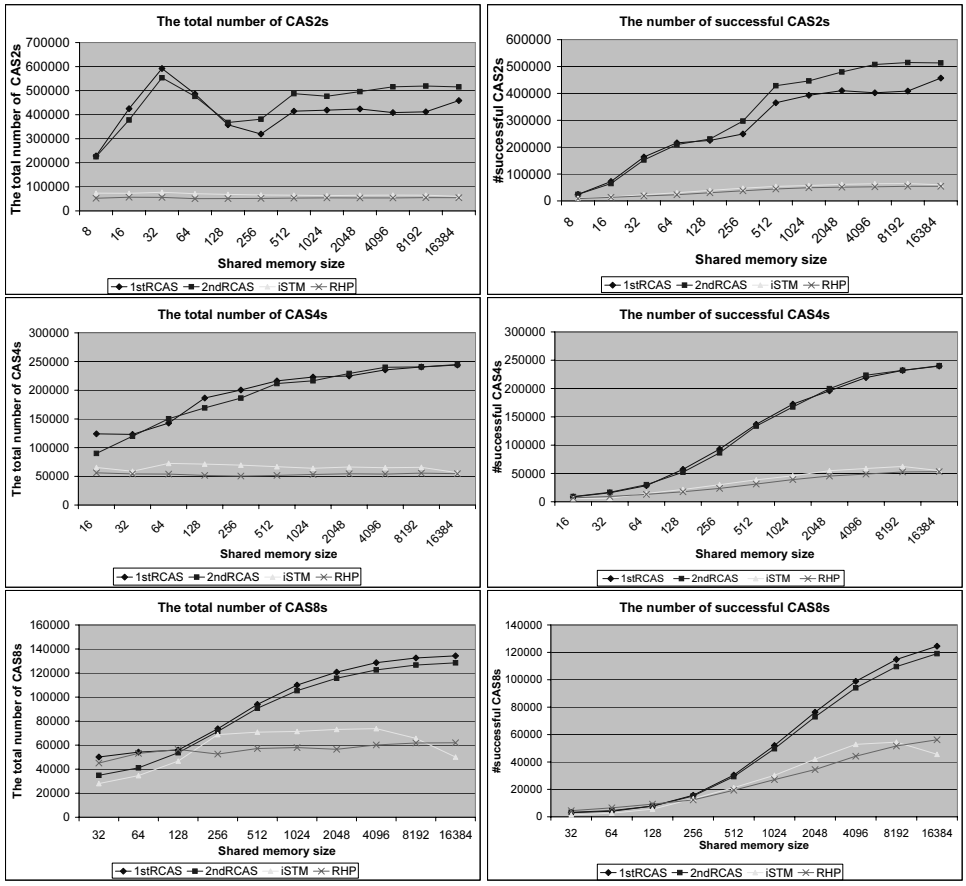


Figure 2.14: The numbers of CAS2s, CAS4s and CAS8s and the number of *successful* CAS2s, CAS4s and CAS8s in one second

2.6.1 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 2.14 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 recursive 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 2.14 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 2.14 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 2.14:

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 2.14, both reactive compare-and-swap operations perform faster than RHP and iSTM from two to nine times.

2.7 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 3

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

Phuong Hoai Ha, Marina Papatriantafilou, Philippas Tsigas
Department of Computer Science,
Chalmers University of Technology
S-412 96 Göteborg, Sweden
Email: {phuong,ptrianta,tsigas}@cs.chalmers.se

Abstract

The reactive diffracting trees *are known efficient distributed data structures for supporting synchronization. They not only distribute a set of processes to smaller groups accessing different parts of the memory in a global coordinated manner, but also adjust their size in order to attain efficient performance across different levels of contention. However, the existing reactive adjustment policy of these trees is sensitive to parameters that have to be manually set in an optimal way and be determined after experimentation. Because these parameters depend on the application as well as on the system configuration, determining their optimal values is hard in practice.*

[1] This paper appeared as a Technical Report No. 2003-09 at the department of Computer Science, Chalmers University of Technology in 2003.

Moreover, because the reactive diffracting trees expand or shrink one level at a time, the cost of a multi-adjustment phase on a reactive tree can become high.

The main contribution of this paper is that it shows that it is *possible to have reactive distributed trees for counting and balancing with no need for the user to fix manually any parameters. We present a data structure that in an on-line manner balances the trade-off between the tree traversal latency and the latency due to contention at the tree nodes. Moreover, the fact that our method can expand or shrink a subtree several levels in any adjustment step, has a positive effect in the efficiency: this feature helps the self-tuning reactive tree minimize the adjustment time, which affects not only the execution time of the process adjusting the size of the tree but also the latency of all other processes traversing the tree at the same time with no extra memory requirements. Our experimental study compared the new trees with the reactive diffracting ones on the SGI Origin2000, a well-known commercial ccNUMA multiprocessor. This study showed that the self-tuning reactive trees i) select the same tree depth as the reactive diffracting trees do; ii) perform better and iii) react faster.*

3.1 Introduction

Distributed data structures suitable for synchronization that perform efficiently across a wide range of contention conditions are hard to design. Typically, "small", "centralized" such data structures fit better low contention levels, while "bigger", "distributed" such data structures can help in distributing concurrent processor accesses to memory banks and in alleviating memory contention.

Diffracting trees [34] are well-known distributed data structures. Their most significant advantage is the ability to distribute a set of concurrent process accesses to many small groups locally accessing shared data, in a coordinated manner. Each process(or) accessing the tree can be considered as leading a *token* that follows a path from the root to the leaves. Each node is a computing element receiving tokens from its single input (coming from its parent node) and sending out tokens to its outputs; it is called *balancer* and acts as a *toggle mechanism* which, given a stream of input tokens, alternately forwards them to its outputs, from left to right (sending them to the left and right child nodes, respectively). The result is an even distribution of tokens at the leaf nodes. Diffracting trees have been introduced for *counting-problems*, and hence the leaf nodes are counters, assigning numbers

to each token that exits from them. Moreover, the number of tokens that are output at the leaves, 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 leaf-nodes such that $i < j$ (i.e. if one makes a drawing of the tokens that have exited from each counter as a stack of boxes, the combined outcome will have the shape of a single step).

The fixed-size diffracting tree is optimal only for a small range of contention levels. To solve this problem, Della-Libera and Shavit proposed the *reactive diffracting trees*, where each node can shrink (to a counter) or grow (to a subtree with counters as leaves) according to the current load, in order to attain optimal performance [10].

The algorithm in [10] uses a set of parameters to make its decisions, namely folding/unfolding thresholds and the time-intervals for consecutive reaction checks. The parameter values depend on the multiprocessor system in use, the applications using the data structure and, in a multiprogramming environment, on the system utilization by the other programs that run concurrently. The programmer has to fix these parameters manually, using experimentation and information that is commonly not easily available (future load characteristics). A second characteristic of this scheme is that the reactive part is allowed to shrink or expand the tree only one level at a time.

In this work we show that reactiveness and these two characteristics are not tied together: in particular, we present a tree-type distributed data structure that has the same semantics as the reactive trees that can expand or shrink many levels at a time, without need for manual tuning. To circumvent the need for manually setting parameters, we have analyzed the problem of balancing the trade-off between the two key measures, namely the contention level and the depth of the tree, in a way that enabled the use of efficient on-line methods for its solution. The new data structure is also considerably faster than the reactive diffracting trees, because of the low-overhead, multilevel reaction part: the new reactive trees can shrink and expand many levels at a time without using clock readings. The self-tuning reactive trees[2], like the reactive diffracting trees, are aimed in general for applications where such distributed data structures are needed. Since the latter were introduced in the context of counting problems, we use similar terms in our description, for reasons of consistency.

[2]We do not use term *diffracting* in the title of this paper since our algorithmic implementation does not use the *prism* construct, which is in the core of the algorithmic design of the (reactive) diffracting trees.

The rest of this paper is organized as follows. The next section provides basic background information about (reactive) diffracting trees. Section 3.3 presents the key idea and the algorithm of the self-tuning reactive tree. Section 3.4 describes the implementation of the tree. Section 3.5 shows the correctness of our algorithm. Section 3.6 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 3.7 concludes this paper.

3.2 Diffracting and Reactive-Diffracting Trees

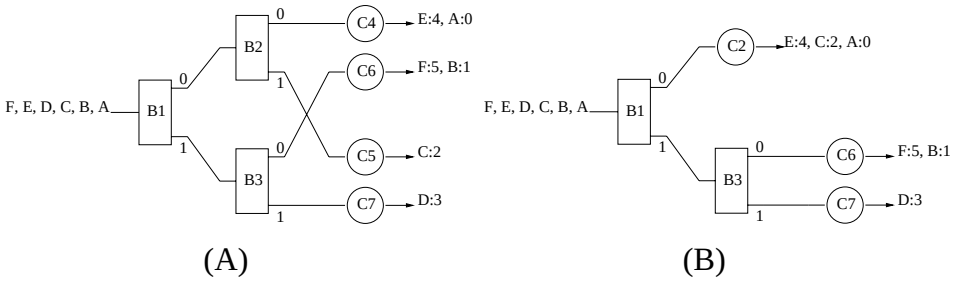


Figure 3.1: Diffracting tree A and reactive diffracting tree B

Figure 3.1(A) shows an example of diffracting tree. The set of processors A, B, C, D, E, F is balanced on all counters where counters $C4, C6$ are accessed by only two processors and counters $C5, C7$ by only one processor. Tokens passing one of these counters receive integers i, $i{+}4$, $i{+}2{*}4, \cdots$ where i is initial value of the counter. In the figure, processors A, B, C, D, E, F receive integers $0, 1, 2, 3, 4, 5$, respectively. Even though the processors access separate shared data (counters), they still receive numbers that form a consecutive sequence of integers as if they would have accessed a centralized counter.

Della-Libera and Shavit improved these trees to the *reactive diffracting trees* where each counter can independently shrink or grow according to the current load in order to attain optimal performance[10]. Trees (A) and (B) in Firgure 3.1 depict the folding action of a reactive diffracting tree. Assume at the beginning the reactive diffracting tree has the shape like tree (A). If the loads on two counters $C4$ and $C5$ are small, the sub-tree whose root is $B2$ shrinks to counter $C2$ as in tree (B). If the sequence of processors A, B, C, D, E, F traverse the reactive diffracting tree (B), three processors

A, C, E will visit counter $C2$. That is, the latency for processors to go from the root to the counter decreases and the load on counter $C2$ increases.

3.3 Self-tuning reactive trees

3.3.1 Problem descriptions

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

1. It must evenly distributes a set of concurrent process accesses to many small groups locally accessing shared data (counters at leaves), in a coordinated manner like the (reactive) diffracting trees. The step-property must be guaranteed.

2. Moreover, it must automatically and efficiently adjust its size according to its load in order to gain performance. It must not require any manually tuning parameters.

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

1. Design a dynamic mechanism that would allow the tree to predict when and how much it should resize in order to obtain good performance whereas the load on it changes unpredictably. Moreover, the overhead that this mechanism will introduce should not exceed the performance benefits that the dynamic behavior itself will bring.

2. This dynamic mechanism should not only adjust the size of the tree in order to improve performance, but, more significantly, adjust it in a way that the tree still guarantees the fundamental properties of the structure, such as the step property.

3.3.2 Key idea

The ideal reactive tree is the one in which each leaf is accessed by only one process(or) –or token[3]– at a time and the cost to traverse it from the root to the leaves is kept minimal. 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 cost to traverse from the root to the leaves increases.

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

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 traversal* from the root to the leaves, is minimal and with *no manual tuning*. In addition to this, an algorithm that can achieve the above, must also be able to cope with the following difficulties: If the tree expands immediately when the contention level increases, then it will pay the expensive cost for travel and this cost is going to be unnecessary if after that the contention level suddenly decreases. On the other hand, if the tree does not expand in time when the contention-level increases, it has to pay the large cost of contention. If the algorithm knew in advance about the changes of contention-levels at the leaves in the whole time-period that the tree operates, it could adjust the tree-size at each time-point in a way such that the overall overhead is minimized. As the contention-levels change unpredictably, there is no way for the algorithm to know this kind of information, i.e. the information about the future.

To overcome this problem, we have designed a reactive algorithm based on the online techniques that are used to solve the online currency trading problem [11].

Definition 3.3.1. *Let* surplus *denote the number of processors that exceeds the number of leaves of the self-tuning reactive tree, i.e. the subtraction of the number of the leaves from the maximal number of processors in the system that potentially want to access the tree. The* surplus *represents the contention level on the tree because the surplus processors cause contention on the leaves.*

Definition 3.3.2. *Let* latency *denote the latency due to traversal from the root to the leaves.*

Our challenge is to balance the trade-off between *surplus* and *latency*. Our solution for the problem is based on an optimal competitive algorithm called *threat-based algorithm* [11]. 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 without knowledge on how the exchange rate will vary in the future.

3.3.3 The new algorithm

In the self-tuning reactive trees, to adapt to the changes of the contention efficiently, a leaf should be free to shrink or grow to any level suggested by

the reactive scheme in one adjustment step. With this in mind, we designed a data structure for the trees such that the time used for the adjustment and the time in which other processors are blocked by the adjustment are kept minimal. Figure 3.2 illustrates the self-tuning reactive tree data structure. Each balancer has a *matching* leaf with corresponding identity. Symmetrically, each leaf that is not at the lowest level of the tree has a *matching* balancer with corresponding identity. The squares in the figure are balancers and the circles are leaves. The numbers in the squares and circles are their identities. Each balancer has two outputs, $left$ and $right$, each of them being 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 the balancer to the matching leaf or from the leaf to the matching balancer, respectively). The solid arrows in the figure represent the present pointer contents.

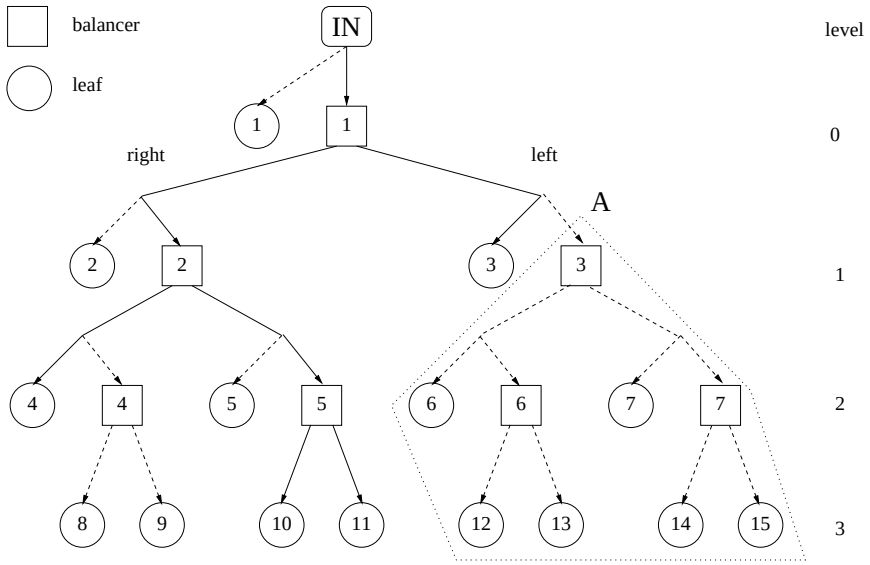


Figure 3.2: self-tuning reactive tree

Assume the tree has the shape as in Figure 3.2, where the solid arrows are the pointers' current contents. A processor p_i first visits the tree at its root IN, then following the root pointer visits balancer 1. When visiting a balancer, p_i switches the balancer's toggle-bit to the other position (i.e. from left to right and vise-versa) and then continues visiting the next node according to the toggle-bit. When visiting a leaf L, p_i before taking an appropriate counter value and exiting, checks the *reaction condition* according to the current load at L. The reaction condition estimates which tree level

is the best for the current load.

The reaction procedure

In order to balance the trade-off between *surplus* and *latency*, the procedure can be described as a game, which evolves in *load-rising* and *load-dropping transaction phases.*

Definition 3.3.3. *A* load-rising (resp. load-dropping) transaction phase *is a maximal sequence of subsequent visits at a leaf-node with monotonic non-decreasing (resp. 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.*

During a load-rising phase, a processor traversing that leaf, may decide to expand the leaf to a subtree of depth that depends on the amount of the rising contention-level. That value is computed using the *threat-based on-line* method of [11], following the principle: "expand *just enough* to guarantee a bounded competitive ratio, even in the case that contention may drop to minimum at the next measurement". Symmetric is the case during a load-dropping phase, where the reaction is to shrink a subtree to the appropriate level, depending on the measurement. The computation of the level to shrink to or to expand to uses the number of processors in the system as an upper bound of contention. The reaction procedure is described in detail in Section 3.4.3.

Depending on the result of checking the reaction condition, the processor acts as follows:

Recommended reaction: Grow to level l_{lower}, i.e. the current load is too high for the leaf L and L should expand to level l_{lower}. The processor, before exiting the tree through L, must help in carrying out the expansion task. To do so, the corresponding subtree must be constructed (if it was not already existent), the subtree's counters' (leaves') values must be set, and the pointer pointing to L must switch to point to its corresponding balancer, which is the root of the subtree resulting from the expansion.

Recommended reaction: Shrink to level l_{higher}, the current load at the leaf L is too low and thus L would like to cause a shrink operation to a higher level l_{higher}, in order to reduce the latency of traversing from the root to the present level. This means that the pointer to the corresponding balancer (i.e. ancestor of L) at level l_{higher} must switch to point to the matching

counter (leaf) and the value of that counter must be set appropriately. Let B denote that balancer. The sub-tree with B as a root contains more leaves than just L, which might not have decided to shrink to l_{higher}, and thus the processor must take this into account. To enable processors do this check, the algorithm uses an asynchronous vote-collecting scheme: when a leaf L decides to shrink to level l_{higher}, it adds its *weighted vote* for that shrinkage to a corresponding vote-array at balancer B.

Definition 3.3.4. *The weight of the vote of leaf L is the number of lowest-level leaves in the subtree rooted at the balancer matching L.*

As an example in Figure 3.2 the weight of the vote of leaf 4 is 2. Note that when voting for balancer B, the leaf L is not concerned about whether B has shrunk into its matching leaf or not. The processor that helps L write its vote to B's vote-array, will then check whether there are enough votes collected at B's vote-array. If there are enough votes collected at B' vote-array, i.e. if the sum of their weights are more than half of the total possible weight of the sub-tree rooted at B (i.e. if more than half of that subtree wants to shrink to the leaf matching B), the shrinkage will happen. After completing the shrinkage task, the processor increases and returns the counter value of L, thus exiting the tree. In the checking process, the processor will abort if the balancer B has shrunk already by a concurrent operation.

In the shrinkage procedure, the leaf matching B and the leaves of the sub-tree rooted at B must be locked in order to (i) collect their counters' values, (ii) compute the next counter value for the leaf matching B 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 it should be shrunk to a leaf.* Therefore, locking the subtree in this case effectively behaves as if locking a leaf (i.e. as it is done in the classical reactive diffracting trees) from the performance point of view.

Example of executing grow : Consider a processor p_i visiting leaf 3 in Figure 3.2, and let the result of the check be that the leaf should grow to sub-tree A with leaves $12, 13, 14$ and 15: The processor first constructs the sub-tree, whereas at the same time other processors may continue to access leaf 3 to get the counter values and then exit the tree without any disturbance. After that, it locks leaf 3 in order to (i) switch the pointer to balancer 3 and (ii) assign the proper values to counters $12, 13, 14$ and 15, then it releases leaf 3. At this point, the new processors following the left

pointer of balancer 1 will traverse through the new sub-tree, whereas the old processors that were directed to leaf 3 before, will continue to access leaf 3's counter and exit the tree. After completing the expansion task, p_i continues its normal task to access leaf 3's counter and exits the tree.

Example of executing shrink : Consider a processor p_i visiting leaf 10 in Figure 3.2 and let the result of the reaction condition be that the subtree should shrink to leaf 2. Because the sub-tree rooted at balancer 2 contains more leaves besides 10, which might not have decided to shrink to 2, processor p_i will check the votes collected at 2 for shrinking to that level. Assume that leaf 4 has voted for balancer 2, too. The weight of leaf 4's vote is two because the vote represents leaves 8 and 9 at the lowest level. Leaf 10's vote has weight 1. Therefore, the sum of the weights of the votes collected at balancer 2 is 3. In this case, processor p_i will help balancer 2 to perform the shrinkage task because the weight of votes, 3, is more than half of the total possible weight of the sub-tree (i.e. more than half of 4, which is the number of the leaves at the lowest level of the subtree –$8, 9, 10$ and 11). Then p_i locks leaf 2 and all the leaves of the sub-tree rooted at balancer 2, collects the counter values at them, computes the next counter value for leaf 2 and 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, the new processors going along the right pointer of balancer 1 can access the counter and exit the tree whereas the old processors are traversing in the old sub-tree. After completing the shrinkage task, the processor exits the tree, returning the value from counter 10.

Space needs of the algorithm

In a system with n processors, the algorithm needs $n-1$ balancer nodes and $2n-1$ leaf nodes. Note that it may seem that the data structure for the self-tuning reactive trees uses more memory space than the data structure for the reactive diffracting trees, since it introduces an auxiliary node (matching leaf) for each balancer of the tree. However, this is actually splitting the functionality of a node in the reactive diffracting trees into two components, one that is enabled when the node plays the role of a balancer and another that is enabled when the node plays the role of a leaf (cf. also Section 3.4.4 and Section 3.4.5). In other words, the corresponding memory requirements are similar. From the structure point of view, splitting the node functionality is a fundamental difference between the self-tuning trees and the reactive

diffracting trees. The voting arrays, space needs at each balancer, are $O(k)$, similar to the space needs for the prism at each balancer of the reactive diffracting trees, where k is the number of leaves of the subtree rooted at the balancer.

3.4 Implementation

3.4.1 Preliminaries

Data structure and shared variables: Figure 3.3 describes the tree data structure and the shared variables used in the implementation.

---

type $NodeType$ = **record** /*stored in one word*/
    Nid : [1..$MaxNodeId$]; $kind$: {$BALANCER, LEAF$};
    $mask$: **bit**; **end**; /*the last bit of the word, init =0*/
  $BalancerType$ = **record**
    $state$: {$ACTIVE, OLD$}; $level$: **int**; $toggleBit$: **boolean**;
    $parent$: [1..$MaxNodeId$]; $leftChild, rightChild$: $NodeType$;
    $votes$: **array**[1..$SizeOfSubtreeRootedAtBalancer$] **of int**; **end**;
  $LeafType$ = **record**
    $state$: {$ACTIVE, OLD$}; $level$: **int**; $count$: **int**;
    $parent$: [1..$MaxNodeId$];
    $lock$: {0..$MaxNodeId$}; /*init =0, else Nid holding the lock*/
    $contention$: **int**; $totLoadEst$: **int**;
    $transPhase$: {$RISING, DROPPING$};
    $latency, baseLatency, surplus, baseSurplus$ **int**;
    /* init $latency = baseLatency = 0$,
        $surplus = baseSurplus = MaxProcs - 1$ */
    $oldSugLevel, sugLevel$: **int**; **end**;/*initialized to level*/
shared variables
  $Balancers$: **array**[0..$MaxNodeId$] **of** $BalancerType$;
  /*Balancers[0] is "IN" node, whose rightChild is unused, cf. fig 3.2*/
  $Leaves$: **array**[1..$MaxNodeId$] **of** $LeafType$;
  $TokenToReact$: **array**[1..$MaxNodeId$] **of boolean**;
  $Tracing$: **array**[1..$MaxProcs$] **of** [1..$MaxNodeId$];
private variables
  $MyPath$: **array**[1..$MaxLevel$] **of** $NodeType$; /*one for each processor*/

Figure 3.3: The tree data structure

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 3.4, where x is a variable and v, old, new are values.

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

Figure 3.4: Synchronization primitives

Moreover, in order to simplify the presentation and implementation of our algorithm, we define, implement and use two advanced synchronization operations: *read-and-follow-link* and *conditionally acquiring lock*. The *read-and-follow-link* operation and the *conditionally acquiring lock* operation are described in the following paragraphs and are outlined in pseudo-code in Figure 3.5.

Read-and-follow-link operation: An auxiliary array $Tracing$ is used to keep track where each processor is in the tree (Figure 3.6). Processor p_i writes to its corresponding entry $Tracing[i]$ the node it will go before it visits the node. Each element $Tracing[i]$ can be updated by only one processor p_i via procedure $Assign(*trace_i, *child)$ and can be read by many other processors via procedure $Read(*trace_i)$ (Figure 3.5). 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 $trace_i$. They are the only procedures that can access array $Tracing$. The two-word assignment operation $Assign()$ is atomic to $Read()$ operation, which is responsible for reading the word[4]. The two operations are lock-free and thus they improve parallelism and performance of the tree. In the design of our tree, we use lock-based synchronizations only at the leaves, where the contention is much lower than that at the higher-level balancers. At all the balancers, we use lock-free operations to synchronize processes.

[4]The proof is given in Lemma 3.5.1

```
NodeType ASSIGN(NodeType * trace_i, NodeType * child)
A0  *trace_i := child;/*mark trace_i under update,clearing mask-bit*/
A1  temp := *child; /*get the expected value*/
A2  temp.mask := 1; /*set the mask-bit*/
A3  if (local := CAS(trace_i, child, temp)) = child then return temp;
A4  else return local;

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 corresponding Assign() ...*/
R4        temp.mask := 1;
R5        CAS(trace_i, local, temp);
R6  while(local.mask = 0); /*... until the Assign() completes*/
R7  return local;

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;
```

Figure 3.5: The Assign, Read and AcquireLock_cond procedures

Conditionally acquiring lock operations: During both the expansion and shrinkage procedures, a processor has to use $AcquireLock\_cond(lock, Nid)$ to acquire the lock of a leaf *on behalf of a node* Nid by writing the node identity Nid to the lock. If that lock is occupied on behalf of an ancestor of node Nid, the procedure returns $Fail$ (line AL1, Figure 3.5).

In the implementation of this procedure, we employed a busy-waiting with exponential back-off technique instead of queue-lock because the contention at the leaf of our self-tuning reactive tree is kept low. In the low-contention environment, the busy-waiting with exponential back-off technique achieves better performance than the queue-lock, which uses a more complicated data structure [28].

For acquiring the lock of a leaf to increase its counter value (i.e. not perform a self-adjustment task), processors use procedure *AcquireLock* without condition. The procedure is similar to *AcquireLock_cond* but does not check the ancestor-condition (line AL1, Figure 3.5). Procedure *AcquireLock* always returns *Succeed*.

The way these locking mechanisms interact and ensure safety and liveness in our data structure accesses is explained in the descriptions of the implementations of the *Grow* and *Shrink* procedures and is proven in Sec-

tion 3.5.

3.4.2 Traversing self-tuning reactive trees

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

NodeType TraverseB(BalancerType B, intPid)
B0      if ((k := FAX(B.toggleBit, 1)) = 0) then
B1        return (Assign(&Tracing[Pid], &B.rightChild));
B2      else return (Assign(&Tracing[Pid], &B.leftChild));

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;
```

Figure 3.6: The TraverseTree, TraverseB and TraverseL procedures

Every processor *Pid* wanting to get a counter value has to traverse the tree by calling function *TraverseTree*(), Figure 3.6. Firstly, the processor visits *Balancer*[0], which is a special node (the "IN" node in Figure 3.2),

the "tree entrance", whose right child is unused and left child points to the root balancer ($Balancers[1]$) or to the corresponding leaf ($Leaves[1]$) if the whole tree degenerated to one leaf. Before visiting *root* balancer or *root* leaf, the processor *pid* updates its new location in $Tracing[Pid]$ (line T0). The path along which the processor traverses the tree is recorded in the processor's private variable $MyPath$ (cf. Figure 3.3), where $MyPath[i]$ is the balancer or leaf the processor visited at level i. Moreover, the processor assigns to the entry $Tracing[i]$ its current position in the tree each time (so that it is traceable). According to the kinds of nodes the processor visits, it will behave correspondingly:

Balancer: the processor calls procedure $TraverseB$ to read and switch the balancer's toggle-bit (line T2 in $TraverseTree()$) as well as to follow the corresponding child link. According to the value of the toggle-bit, the processor updates it new location to $Tracing[pid]$ (lines B1, B2 in TraverseB()).

Leaf: In all cases, the processor calls procedure $TraverseL$ to read and increase the leaf counter $L.count$ (line T8) and resets $Tracing[pid]$ to the value of *Root*, node IN in Figure 3.2 (line T10). If the processor exits the tree through a leaf whose state is $ACTIVE$[5] , the processor must actively execute the reaction procedure. To do that, it needs to get hold of the $TokenToReact$ (if that is not available, then some other processor is executing the reactive procedure on behalf of the leaf, which imples for processor *Pid* that it does not need to do the same job). Subsequently, procedure $CheckCondition$ needs to be invoked. Depending on the outcome, the processor will invoke procedure *Grow* - which expands the leaf to a subtree - or procedure $Elect2Shrink$, which votes for the appropriate level to shrink to, and, if the votes collected for that decision are adequate, also performs the actual shrinkage. Subsections 3.4.4 and 3.4.5 give the details of the actions and synchronization needed during the procedures *Grow* and *Shrink*.

3.4.3 Reaction conditions

As mentioned in section 3.3.3, each leaf L of the self-tuning reactive tree estimates which level is the best for the current load. The leaf estimates the total load of tree by using the following formula:

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

```
int CheckCondition(LeafType L)
C0   TotLoadEst := MIN(MaxProcs, L.contention * 2^L.level);
C1   FirstInPhase := False;
C2   if (L.transPhase = RISING) and (TotLoadEst < L.totLoadEst) then
       L.transPhase := DROPPING;
       L.baseLatency := L.latency; FirstInPhase := True;
C3   else if (L.transPhase = DROPPING) and
       (TotLoadEst > L.totLoadEst) then
       L.transPhase := RISING;
       L.baseSurplus := L.surplus; FirstInPhase := True;
C4   if L.transPhase = RISING then
       Surplus2Latency(L, TotLoadEst, FirstInPhase);
C5   else
       Latency2Surplus(L, 1/TotLoadEst, FirstInPhase);
C6   L.totLoadEst := TotLoadEst;
C7   L.oldSugLevel := L.sugLevel;
C8   L.sugLevel := log2(MaxProcs - L.surplus);
C9   if L.sugLevel < L.level then return SHRINK;
C10  else if L.sugLevel > L.level then return GROW;
C11  else return NONE;

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

Latency2Surplus(L, 1/TotLoadEst, FirstInPhase)
/* symmetric to the above with:
       X := L.latency; baseX := L.baseLatency; Y := L.surplus;
       rXY := 1/TotLoadEst; LrXY := 1/L.totLoadEst;*/
```

Figure 3.7: The CheckCondition, Surplus2Latency and Latency2Surplus procedures

$$TotLoadEst = L.contention * 2^{L.level} \tag{3.1}$$

line C0 in $CheckCondition()$ in Figure 3.7, where $MaxProcs$ is the maximum number of processors potentially wanting to access the tree and $L.contention$, the contention of a leaf, is the number of processors that currently visit the leaf. $L.contention$ is increased by one every-time a processor visits the leaf L (line T3 in Figure 3.6) and is decreased by one when a processor leaves the leaf (line T9 in Figure 3.6). Because the number of processors accessing the tree cannot be greater than $MaxProcs$ we have an upper bound for the load: $TotLoadEst \leq MaxProcs$.

At the beginning, the initial tree is just a leaf, so the the initial *surplus*, *baseSurplus*, is $MaxProcs - 1$ and the initial *latency*, *baseLatency*, is 0. Then, based on the contention variation on each leaf, the values of *surplus* and *latency* will be changed according to the online trading algorithm. Procedure $Surplus2Latency()$ (respectively $Latency2Surplus()$) is invoked (lines C4, C5), to adjust the number of surplus processors that the tree should have at that time. The surplus value will be used to compute the number of leaves the tree should have and consequently the level the leaf L should shrink/grow to.

Procedure $Surplus2Latency(L, TotLoadEst, FirstInPhase)$ in Figure 3.7 exchanges $L.surplus$ to $L.latency$ according to the *threat-based algorithm* [11] using $TotLoadEst$ as exchange rate. For self-containment, the computation implied by this algorithm is explained below. In a load-rising transaction phase, the following rules must be followed:

1. The tree is expanded only when the estimated current total load is the highest so far in the present transaction phase.

2. When expanding, expand *just enough* to keep the competitive ratio $c = \varphi - \frac{\varphi-1}{\varphi^{1/(\varphi-1)}}$, where $\varphi = \frac{MaxProcs}{2}$, even if the total load drops to the minimum possible in the next measurement.

The number of leaves the tree should have more is

$$deltaSurplus = baseSurplus.\frac{1}{C}.\frac{TotLoadEst - TotLoadEst^{-}}{TotLoadEst - 2}$$

where $TotLoadEst^{-}$ is the highest estimated total load before the present measurement and *baseSurplus* is the number of surplus processors at the beginning of the present transaction phase (line SL3, where mXY is the lower bound of the estimated total load). Everytime a new transaction

phase starts, the value *baseSurplus* is set to the last value of *surplus* in the previous transaction phase (line C3). The parameter $FirstInPhase$ is used to identify whether this is the first exchange of the transaction phase. At the beginning,

$$surplus = baseSurplus = MaxProcs - 1$$

i.e. the tree degenerates to a node. Both variables $TotLoadEst^-$ and *baseSurplus* are stored in fields $TotLoadEst$ and *baseSurplus* of the leaf data structure, respectively.

Symmetrically, when the tree should shrink to reduce the traversal latency, the exchange rate is the inverse of the total load, $rXY = \frac{1}{TotLoadEst}$, which is increasing. In this case, the value of *surplus* increases and that of *latency* decreases.

3.4.4 Expanding a leaf to a sub-tree

A grow operation of a leaf L to a subtree T, whose root is L's matching balancer B and whose depth is $L.SugLevel - L.level$, essentially needs to (i) set the counters at the new leaves in T to proper values to ensure the step property; (ii) switch the corresponding child pointer of L's parent from L to B; and (iii) activate the nodes in T. (Figure 3.9 illustrates the steps taken in procedure grow, which is given in pseudocode in Figure 3.8.) Towards (i), it needs to:

- make sure there are no pending tokens in T. If there are any, *Grow* aborts (step G1 in *Grow*), since it should not cause "old" tokens get "new" values (that would cause "holes" in the sequence of numbers received by all tokens in the end). A new grow operation will be activated anyway by subsequent tokens visiting L, since L has high contention.

- acquire the locks for the new leaves, to be able to assign proper counter values to them (step G3 in *Grow*) to ensure the step property.

- make a consitent measurement of the number of pending processors in L and $L.count$ to use in the computation of the aforementioned values for the counters. Consistency is ensured by acquiring L's lock (step G4) and by switching L's parent's pointer from L to B (i.e. perfoming action (ii) described above; step G5 in *Grow*), since the latter leaves a "non-interfered" set of processors in L.

```
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]) /* Can't miss any processors since
                                    the current ones go to Leaves[Nid]*/
        if ∃ pending processors in the subtree rooted at B then
          return; /*abort*/
G2    for each balancer B' in the subtree rooted at B, up to level L.sugLevel − 1
        forall entries i : B'.votes[i] := 0; B'.toggleBit = 0;
G3    for each leaf L' at level L.sugLevel of the subtree rooted at 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 operation*/
        /*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]) /*Can't miss any since the new ones go to B*/
        ppL := #(pending processors at L);
G7    CurCount := L.count; L.state := OLD;
G8    Release(L.lock);
G9    for each balancer B' as described in step G2 do B'.state := ACTIVE;
G10   for each leaf L' as described in step G3 do
        update L'.count using ppL and CurCount; L'.state := ACTIVE;
        Release(L'.lock);
      return;/*Success*/
```

Figure 3.8: The Grow procedure

Each of these locks' acquisition is *conditional*, i.e. if some anscestor of L holds it, the attempt to lock will return fail. In such a case the grow procedure aborts, since the failure to get the lock means that there is an overlapping shrink operation by an ancestor of L. (Note that overlapping grow operations by an ancestor of L would have aborted, due to the existence of the token (processor) at L (step G1 in *Grow*).) Furthermore, the new leaves' locks are requested in *decreasing* order of node-id, followed by the request of $L.lock$, to avoid deadlocks.

Towards action (iii) from above, the grow procedure needs to reset the tree's T balancers' toggle bits and vote arrays (before switching L's parent's pointer from L to B; step G2) and set the state values of all balancers

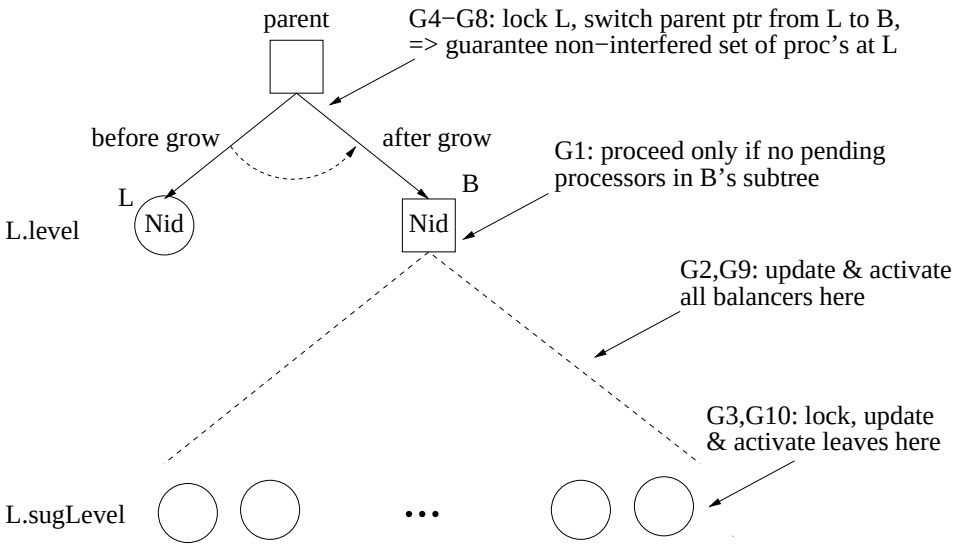


Figure 3.9: Illustration for Grow procedure

and bottom-level leaves in T to ACTIVE (after having made sure that the growing will not abort; step G9-G10).

Assigning new values to the leaf-counters: (procedure $Grow()$, line G10). Because all toggle-bits of the balancers in the new sub-tree are reset to 0, the first processor passing through the new subtree will arrive at the most right leaf[6] at level $L.sugLevel$. The code for assigning counter values for these new leaves is as in Figure 3.10, where $CurCount$ and ppL are the

---

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

Figure 3.10: Assigning counter values for the new leaves

values read at lines G7 and G6 in $Grow()$. Local variable $next\_count\_value$ is the successive counter value after ppL pending processors leave leaf L. Field $init$ in the leaf data structure (Fig. 3.3) is the base to calculate counter values, which is unchanged. For example, in Fig. 3.1 the base $init$ of counter $C6$ is 1 and the n^{th} counter value is $1 + n * 4$.

[6]The right side is shown in Figure 3.2

3.4.5 Shrinking a sub-tree to a leaf

Towards a decision of whether and where to shrink to, the token at a leaf L_0 with recommended reaction to shrink to level $L_0.SugLevel$ must add L_0's vote in the vote arrays of the balancers of its path from the root, starting from level $L_0.SugLevel$, up to level $L_0.level - 1$ (it must also take care to remove potentially existing older votes at layers above that; step E1 in $Elect2Shrink$ in Figure 3.11). When a balancer with enough votes is reached, the shrink operation will start (steps E3-E5 in $Elect2Shrink$). Figure 3.12 and figure 3.11 illustrate and give the pseudocode of the steps taken towards shrinking.

Symmetrically to a grow operation, a shrink from a subtree T rooted at balancer B (with enough votes) to B's matching leaf L, essentially needs to (i) set the counter at L to the proper value to ensure the step property; (ii) switch the corresponding child pointer of B's parent from B to L; and (iii) de-activate the nodes in T. Towards (i), it needs to:

- make sure there are no pending tokens in L. If there are any, shrink aborts (step S2 in $Shrink$), since it should not cause "old" tokens get "new" values. Subsequent tokens' checking of the reaction condition may reinitiate the shrinking later on anyway.

- acquire L's lock (step S3), to be able to assign an appropriate counter value to it, to ensure the step property.

- make a consitent measurement of (1) the number of pending processors in T and (2) the values of counters of each leaf L' in T. Consistency is ensured by acquiring $L'.lock$ for all L' in T (step S5) and by switching B's parent's pointer from B to L (i.e. perfoming action (ii) described above; step S6 in $Shrink$), since the latter leaves a "non-interfered" set of processors in T.

Similarly to procedure grow, these locks' acquisition is conditional. Symmetrically with grow, the requests are made first to $L.lock$ and then to the locks of the leaves in T, in *increasing* order of node-id, to avoid deadlocks. Failure to get $L.lock$ implies an overlapping shrink operation by an ancestor of L. Note that overlapping grow operations by an ancestor of L would have aborted, due to the existence of the token at B (step G1 in $Grow$). Note also that an overlapping shrink by some of L's ancestors cannot cause any of the attempts to get some $L'.lock$ to fail, since that shrink operation would have to first acquire the lock for L (and if it had succeeded in getting that,

```
ELECT2SHRINK( int Nid, NodeType MyPath[])
E0   L := Leaves[Nid];/*the leaf asks to shrink*/
     if L.oldSugLevel < L.sugLevel then /*new suggested level is
                                          lower than older suggestion*/
       for(i := L.oldSugLevel; i < L.sugLevel; i++) do
E1       Balancers[MyPath[i].Nid].votes[Nid] := 0;
     else for (i := L.sugLevel; i < L.oldSugLevel; 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   if (TAS(TokenToReact[Nid]) = 1) then return; /*abort*/
       /*someone else is doing the shrinkage*/
S2   forall i : Read(Tracing[i]) /*can't miss any since the current
                                   ones go to B*/
       if ∃ pending processor at L then return;/*abort*/
S3   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*/
S4   L.state := OLD; /*avoid reactive adjustment at L*/
S5   forall leaf L' in B's subtree, in increasing order of nodeId do
       AcquireLock_cond(L'.lock, Nid); /*No fails expected since
             Grow operations by ancestors will abort at G1*/
S6   Switch the parent's pointer from B to L
S7   forall i : Read(Tracing[i])
       eppB := #(effective pending processors in B's subtree;
       /*can't miss any since the new ones go to L*/
S8   for each balancer B' in the subtree rooted at B do
       B'.state := OLD;
     SL := ∅; SLCount := ∅;
S9   for each leaf L' in the subtree rooted at B do
       if (L.state = ACTIVE) then
         Compute SL := ∪L' and SLCount := ∪L'.count;
         L'.state := OLD;
       Release(L'.lock);
S10  L.count := f(eppB, SL, SLCount);
S11  L.state := ACTIVE;
S12  Release(L.lock);
S13  Reset(TokenToReact[Nid]);
```

Figure 3.11: The Elect2Shrink and Shrink procedures

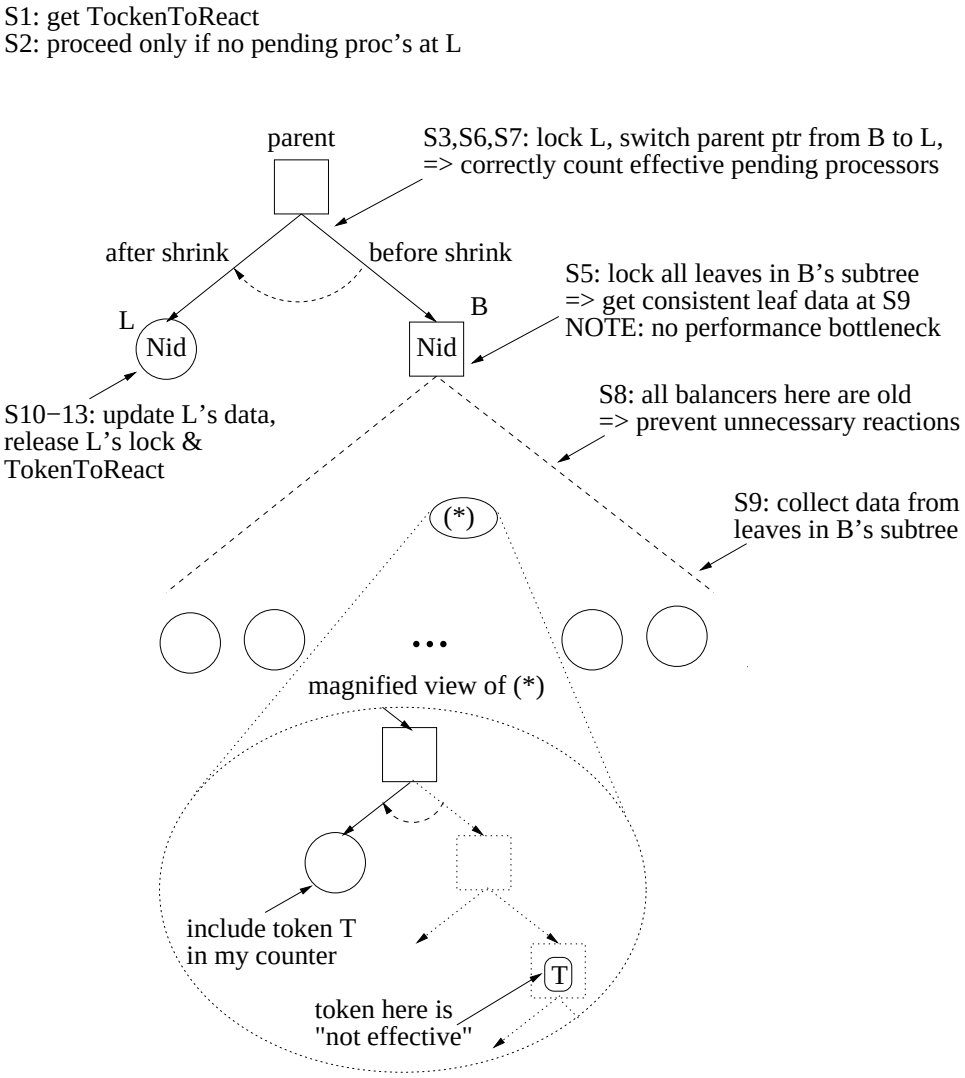


Figure 3.12: Illustration for Shrink procedure

it would have caused the shrink from B to L to abort earlier, at step S3 of $Shrink()$).

Towards action (iii) from above, the shrink procedure sets the balancers' and leaves' states in T to OLD (steps S8-S9 in $Shrink$), after having made sure that the shrink will not abort.

Assigning new values to the leaf-counters: The counter value for the new leaf is calculated based on the list of active leaves SL, their counter values $SLCount$ and the number of effective pending processors $eppB$ in the balancer's sub-tree (line S10). Function $f(eppB, SL, SLCount)$ is used to calculate the new counter value for the new leaf (line G10). The function is implemented as in Figure 3.13.

```
int F ( int eppB, list_tSL, list_tSLCount)
    Convert leaves in B to the same level, the lowest level
    ⇒ new lists of leaves SL' and their 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;
    result = lastL.count − 2^(lastL.level) + 2^(B.level);
    return result;
```

Figure 3.13: Calculating the counter value for the new leaf

For example, in Figure 3.2 if the sub-tree of balancer 2 shrinks to leaf 2 and the list 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 list of leaves SL' is $8, 9, 10, 11$. After converting, the sub-tree becomes balanced and the step-property must be satisfied on the sub-tree. The following feature of a tree satisfying step-property was exploited to calculate the new counter value in our implementation. Call the highest counter value of leaves at the lowest level $MaxValue$. The counter values of the leaves must be in range $(MaxValue - 2^{LowestLevel}, MaxValue]$.

3.5 Correctness Proof

Firstly, we prove that procedure $Assign()$ is atomic to $Read()$ as mentioned in subsection 3.4.1.

Lemma 3.5.1. Assign() *is atomic to* Read().

Proof. In procedure $Assign(*trace_i, *child)$, the variable pointed by $trace_i$ is first locked by writing the pointer *child* to it (line A0). We exploit the last bit in the pointer ($mask-bit$ in $NodeType$), which is unused because of word-alignment memory architecture, to identify if the variable is locked or not. If the last bit of the variable $*trace_i$ is zero, the variable is locked and its value is the address of the other variable whose value must be written to $*trace_i$. After reading the expected value, $*child$, we set the last bit to 1 so as to unlock the variable $*trace_i$ and then write this value to the variable $*trace_i$ by *compare_and_swap* operations (lines A1-A3).

In procedure $Read(*trace_i)$, when reading the value of the variable $*trace_i$, we check if the variable contains the expected value (line R2). If the last bit of the variable is zero, the procedure will help the corresponding procedure $Assign()$ to write the expected value to the variable before trying to read again (lines R3-R5). The linearization point of procedure $Read()$ is the point the procedure reads the value with the non-zero last bit and the linearization point of procedure $Assign()$ is the point the procedure writes a zero-last-bit value to the variable $*trace_i$ (line A0 in $Assign()$). □

As mentioned in subsection 3.4.4 and subsection 3.4.5, the leaves are locked in decreasing order of leaf identities in $Grow()$ but in increasing order in $Shrink()$ and thus we need to prove that deadlock never occurs. Further, the common assumption that processors traversing the reactive trees do not fail [10] is adopted here, as well.

Lemma 3.5.2. *Self-tuning reactive trees are deadlock-free.*

Proof. The interference between two balancers who are trying to lock leaves (recall that processors lock leaves on behalf of balancers) occurs only if one of the balancers is the other's ancestor in the *family* tree. Let us consider 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 tasks that shrink their sub-trees to leaves, both will lock leaves in increasing order of leaf identities by using the procedure $AcquireLock\_cond()$. If the leaf with smallest identity that b_j needs is locked by b_i, procedure $AcquireLock\_cond()$ 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 itself, b_i must wait at the leaf until b_j completes its own work and then b_i continues locking all necessary leaves. If no processor locking the leaves on behalf of a balancer fails, no deadlock will occur.

Case 2: b_i executes a shrinkage task, which shrinks its sub-tree to a leaf, and b_j executes an expansion task, which expands its leaf to a sub-tree. 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 locked so far (lines G3, G4 in *Grow*()). Therefore, b_i can lock 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 an expansion task. This is because in its sub-tree there is at least one pending processor that helps b_j do its reaction task.

□

Corollary 3.5.1. *In the shrinkage process, if the corresponding balancer successfully acquired the necessary leaf with smallest identity, it will then successfully lock all the leaves it needs.*

Therefore, procedure *Shrink*() returns *Fail* only if the leaf corresponding to the balancer is locked by an ancestor of that balancer.

Lemma 3.5.3. *There is no interference between any two expansion tasks in our 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 the ancestor of the other. □

Lemma 3.5.3 is the reason why we need not to lock balancers before resetting their variables, except for their states, in *Grow*() (line G2 in *Grow*()). The balancers' states are updated only when the corresponding expansion/shrinkage task is ensured to complete successfully.

Further, we need to show that the number of processors used to calculate the counter value for the new leaves in both *Grow*() and *Shrink*() is counted accurately by using the global array *Tracing*.

Definition 3.5.1. Old balancers/leaves are the balancers/leaves whose states are OLD

Definition 3.5.2. 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 new counter values calculated for the new leaves. An illustration to enhance the understanding of this definition is given in Figure 3.12, where the token marked as "T" at the lower part of the figure is not effective.

Lemma 3.5.4. *The number of* effective processors *that are pending in a locked sub-tree or a locked leaf is counted accurately by procedures Grow and Shrink.*

Proof. A processor p_i executing an adjustment task switches a pointer from a branch of the tree to a new branch before counting the pending processors in the old branch (lines G5, G6 in $Grow()$ and lines S6, S7 in $Shrink()$). Because of this and because procedure $Assign()$ is atomic to procedure $Read()$ (by lemma 3.5.1), the processor p_i will count 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 it (lines S8, S9 in $Shrink()$ and line G7 in $Grow()$), and (ii) after locking all necessary balancers and leaves, processors continue processing the corresponding shrinkage/expansion tasks only if the switching balancers/leaves are still in an active state (line G4 in $Grow()$ and line S3 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 visiting, we can know whether the processor p_j is effective (line S7 in $Shrink()$). In conclusion, the number of effective processors pending in a locked sub-tree is counted accurately by procedure $Shrink()$ also in the case of tree shrinkage. □

Because the number of effective pending processes is counted accurately according to Lemma 3.5.4, the counter values that are set at the leaves after the adjustment steps in the self-tuning trees are correct, i.e. the step-property is guaranteed. That means the requirements mentioned in subsection 3.3.1 are satisfied. This implies the following theorem:

Theorem 1. *The self-tuning tree obtains the following properties:*

1. *Evenly distribute a set of concurrent process accesses to many small groups locally accessing shared data, in a coordinated manner like the (reactive) diffracting trees. The step-property is guaranteed.*

2. *Automatically and efficiently adjust its size according to its load in order to gain performance. No manually tuning parameters are needed.*

3.6 Evaluation

In this section, we evaluate the performance of the self-tuning reactive trees proposed here. We used the reactive diffracting trees of [10] as a basis of comparison since they are the most effiecient reactive counting constructions in the literature.

The source code of [10] is not publicly available and we implemented it following exactly the algorithm as it is presented in the paper. We used the full-contention benchmark, the index distribution benchmark and the surge load benchmark on the SGI Origin2000, a popular commercial cc-NUMA multiprocessor. The index distribution benchmark and the surge load benchmark are the same as those used in [10].

In [10], besides running the benchmarks on a non-commercially availiable machine with 32 processors (Alewife), the authors also ran them on the simulator simulating a multiprocessor system similar to Alewife with up to 256 processors.

The most difficult issue in implementing the reactive diffracting 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 [10]. However, subsection *Load Surge Benchmark* in [10] described that the reactive diffracting tree sized to a depth 3 tree

when they ran index-distribution benchmark [34] with 32 processors in the highest possible load ($work = 0$) and the number of consecutive timings was set at 10. According to the description, we run our implementation of the reactive diffracting tree on the ccNUMA Origin 2000 with 32 MIPS R10000 processors and the result is that folding and unfolding thresholds are 4 and 14 microseconds, respectively. This selection of parameters did not only keep our experiments consistent with the ones presented in [34] but also gave the best performance for the diffracting trees in our system. Regarding the prism size (prism is an algorithmic construct used in diffracting process in the reactive diffracting trees), each node has $c2^{(d-l)}$ prism locations, where $c = 0.5$, d is the average value of the reactive diffracting tree depths estimated by processors passing the tree and l is the level of the node [10, 33]. The upper bound for adaptive spin *MAXSPIN* is 128 as mentioned in [34].

In order to show this is a fair comparison, we look at synchronization primitives used in the reactive diffracting tree *RD-tree* and in the self-tuning reactive tree *ST-tree*. The RD-tree uses three primitives *compare-and-swap*, *swap* and *test-and-set* (page 862 in [10]) and the ST-tree uses three primitives *compare-and-swap*, *fetch-and-xor* and *test-and-set* (Figure 3.4). Therefore, the synchronization primitives used in the ST-tree are comparable with those used in the RD-tree.

In order to make the properties and the performance of the self-tuning reactive tree algorithm presented here accessible to other researchers and practicians and help also reproducibility of our results, C code for the tested algorithms is available at
`http://www.cs.chalmers.se/~phuong/sat4lic.tar.gz`.

3.6.1 Full-contention and index distribution benchmarks

The system used for our experiments was a ccNUMA SGI Origin2000 with sixty four 195MHz MIPS R10000 CPUs with 4MB L2 cache each. The system ran IRIX 6.5. We ran the reactive diffracting tree *RD-tree* and the self-tuning reactive tree *ST-tree* in the full-contention benchmark, in which each thread continuously executed only the function to traverse the respective tree, and in the index distribution benchmark with $work = 500\mu s$ [10][34]. Each experiment ran for one minute and we counted the average number of operations for one second.

Results: The results are shown in Figure 3.14 and Figure 3.15. The right charts in both the figures show the average depth of the ST-tree compared to

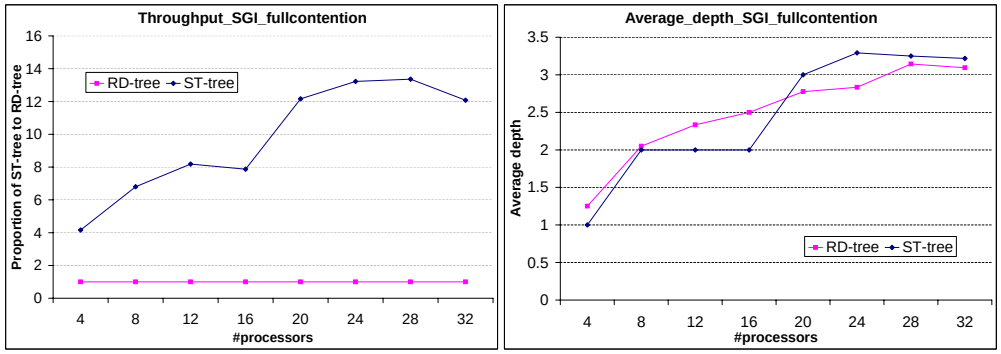


Figure 3.14: Throughput and average depth of trees in the full-contention benchmark on SGI Origin2000.

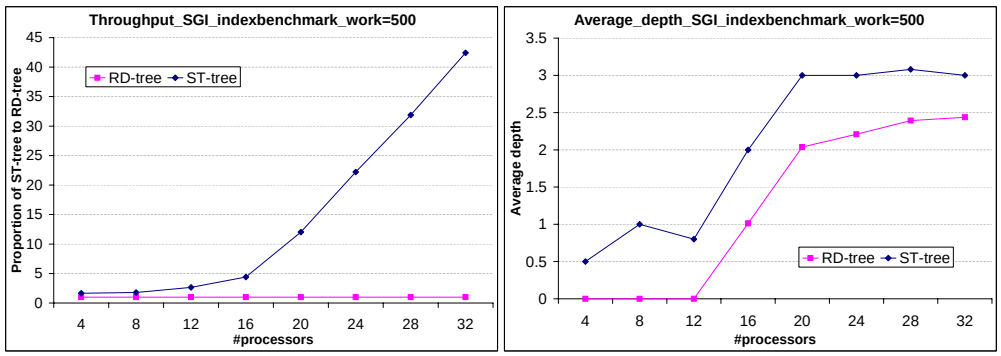


Figure 3.15: Throughput and average depth of trees in the index distribution benchmark with $work = 500\mu s$ on SGI Origin2000.

the RD-tree. The left charts show the proportion of the ST-tree throughput to that of the RD-tree.

The most interesting result is that when the contention on the leaves increases, the ST-tree automatically adjusts its size close to that of the RD-tree that requires three experimental parameters for each specific system.

Regarding throughput and scalability, we observed that the ST-tree performs better than the RD-tree. This is because the ST-tree has a faster and more efficient reactive scheme than RD-tree. The surge load benchmark in the next subsection shows that the reactive trees *continuously* adjust their current size slightly around the average size corresponding to a certain load (cf. Figure 3.16). Therefore, an efficient adjustment procedure will significantly improve the performance of the trees. There are three algorithmic

differences between the ST-tree and the RD-tree:

- The ST-tree reacts to the contention variation faster with lower overhead as described in Section 3.3.3, whereas in the RD-tree the leaves shrink or grow only one level in one reaction step and then the leaves have to wait for a given number of processors passing through themselves before shrinking or growing again.

- In the RD-tree, whenever a leaf shrinks or grows, all processors visiting the leaf are blocked completely until the reaction task completes. Moreover, some processors may be forced to go back to higher nodes many times before exiting the RD-tree. In the ST-tree, this is avoided with the introduction of the matching leaves.

- Algorithmically, the two constructions do not only differ on the reactive schemes that they use, but also on the shared variables and the way that these are accessed (cf. Section 3.3.3 and in particular the data structure splitting the functionality of a node in ST-tree into a balancer and a leaf.)

Studing the figures closer, in the full-contention benchmark (Figure 3.14), we can observe the scalability properties of the ST-tree, which shows increased throughput with increasing number of processors (as expected using the aforementioned arguments) in the left chart. The right chart shows that the average depth of the ST-trees is nearly the same as that of the RD-tree, i.e. the reaction decisions are pretty close.

In the index distribution benchmark with $work = 500\mu s$, which provides a lower-load environment, the ST-tree can be observed to show very desirable scalability behaviour as well, as shown in Figure 3.15. The charts of the average depths of both trees have approximately the same shapes again, but the ST-tree expands from half to one depth unit more than RD-tree. This is because the throughput of the former was larger, hence the contention on the ST-tree leaves was higher than that on RD-tree leaves, and this made the ST-tree expand more.

The surge load benchmark in the next subsection shows that the trees continuously adjust their current size around the average (cf. Figure 3.16). Therefore, an efficient adjustment procedure will significantly improve the performance of the trees.

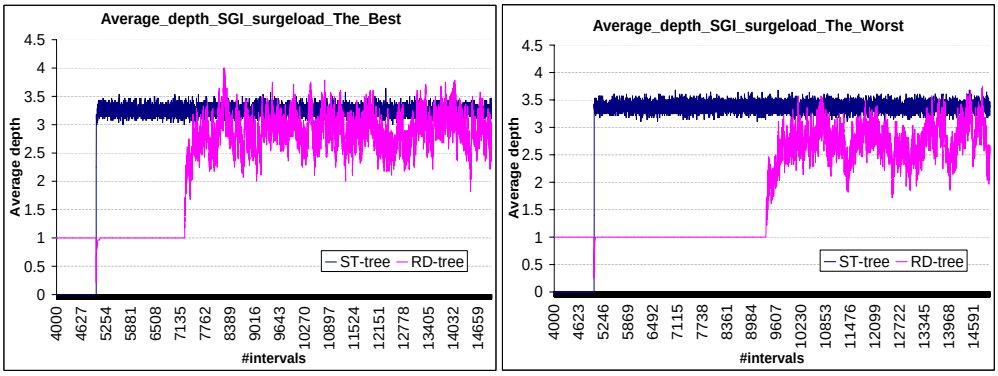


Figure 3.16: Average depths of trees in the surge benchmark on SGI Origin2000, best and worst measurements. In a black-and-white printout, the darker line is the ST-tree.

3.6.2 Surge load benchmark

The benchmark shows how fast the trees react to contention variations. The benchmark is run on a smaller but faster machine [7], ccNUMA SGI2000 with thirty 250MHz MIPS R10000 CPUs with 4MB L2 cache each. On the machine the optimal folding and unfolding thresholds, which keep our experiments consistent with the ones presented in [34], are 3 and 10 microseconds, respectively. All other parameters are kept the same as the benchmarks discussed in the previous subsection.

In this benchmark, we measured the average depth of each tree in each interval of 400 microseconds. The measurement was done by a monitor thread. At interval 5000, the number of threads was changed from four to twenty eight. The average depth of the trees at the interval 5001 was measured after synchronizing the monitor threads with all the new threads, i.e. the period between the end of interval 5000 and the beginning of interval 5001 was not 400 microsecond. Figure 3.16 shows the average depth of both trees from interval 4000 to interval 15000. The left chart shows the best reaction time figures for the RD-tree and the ST-tree; the right one shows the worst reaction time figures for the RD-tree and the ST-tree. In the benchmark, the ST-tree reached the suitable depth 3 for the case of 28 threads at interval 5004 in the best case and 5008 in the worst case, i.e only after 5 to 8 intervals since the time all 28 threads started to run. The

[7] This is because the first machine was replaced with that one at our computer center while this experimental evaluation was still in progress.

RD-tree reached level 3 at interval 7447 in the best case and at interval 9657 in the worst case. That means the reactive scheme introduced in this paper and used by the ST-tree makes the same decisions as the RD-tree, and, moreover, it reacts to contention variations much faster than the latter.

3.7 Conclusion

The self-tuning reactive trees presented in this work distribute the set of processors that are accessing them, to many smaller groups accessing disjoint critical sections in a coordinated manner. They collect information about the contention at the leaves (critical sections) and then they adjust themselves to attain adaptive performance. The self-tuning reactive trees extend a successful result in the area of reactive concurrent data structures, the reactive diffracting trees, in the following way:

- The reactive adjustment policy does not use parameters which have to be set manually and which depend on experimentation.
- The reactive adjustment policy is based on an efficient adaptive algorithmic scheme.
- They can expand or shrink many levels at a time with small overhead.
- Processors pass through the tree in only one direction, from the root to the leaves and are never forced to go back.

Moreover, the self-tuning reactive trees:

- have space needs comparable with that of the classical reactive diffracting trees
- exploit low contention cases on subtrees to make their locking process as efficient as in the classical reactive diffracting trees although the locking process locks more nodes at the same time.

Therefore, the self-tuning reactive trees can react quickly to changes of the contention levels, and at the same time offer a good latency to the processes traversing them and good scalability behaviour. We have also presented an experimental evaluation of the new trees, on the SGI Origin2000, a well-known commercial ccNUMA multiprocessor. We think that it is of big interest to do a performance evaluation on modern multiprocessor systems that are widely used in practice.

Last, we would like to emphasize an important point. Although the new trees have better performance than the classical ones in the experimental evaluation conducted and presented here, this is not the main contribution of this paper. What we consider as main contribution is the ability of the new trees to efficiently self-tune their size without any need of manual tuning.

Chapter 4

Conclusions

Designing reactive shared objects is a promising approach to synchronize concurrent processes in shared memory multiprocessor systems. This thesis has presented two such objects called the *reactive multi-word compare-and-swap object* and the *self-tuning reactive tree*.

In Chapter 2, two reactive, lock-free algorithms that implement multi-word compare-and-swap operations (CASN) are presented. 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 the contention and reacts accordingly is efficient and does not cancel out the benefits in most contention levels. 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 to contention variantions and significantly outperform the best-known alternatives in most contention conditions.

In Chapter 3, the self-tuning reactive trees are presented. The trees distribute the set of processors that are accessing them, to many smaller groups accessing disjoint critical sections in a coordinated manner. They collect information about the contention at the leaves (critical sections) and then they adjust themselves to attain optimal performance. The self-tuning reactive

trees extend the very successful result in the area of reactive concurrent data structures, the reactive diffracting trees, in the following way:

- The reactive adjustment policy does not use parameters that have to be set manually and depend on experimentation.
- The reactive adjustment policy is based on an efficient algorithmic scheme.
- They can expand or shrink many levels at a time with small overhead.
- Processors pass through the tree in only one direction, from the root to the leaves and they are never forced to go back.

Moreover, the self-tuning reactive trees:

- have space requirement comparable with that of the classical reactive diffracting trees
- exploit the low contention cases on subtrees to make their locking process as efficient as in the classical reactive diffracting trees although the locking process locks more nodes at the same time.

Therefore, the self-tuning reactive trees can react quickly to changes of the contention level, and at the same time offer good latency to the processes traversing them.

In conclusion, the reactive shared objects, which can react to environment variations, are a promising research trend to synchronize processes in multiprocessor systems. This thesis has presented two such objects and has showed that they are feasible for real systems. We believe that the techniques used in this thesis can be applied for constructing other concurrent objects in the future.

Chapter 5

Future Work

The interference among processes in the multiprocessor systems generates a variant and unpredictable environment to concurrent objects, whose performance heavily relies on the surrounding environment. Therefore, the problems on constructing reactive concurrent objects, which can react to the environment variations in order to achieve good performance, are attractive to many researchers.

However, most of available reactive policies rely on either some manually tuned thresholds or known probability distributions of some unpredictable inputs. These parameter values depend on the multiprocessor system in use, the applications using the objects and, in a multiprogramming environment, on the system utilization by the other programs that run concurrently. The programmer has to fix these parameters manually, using experimentation and information that is commonly not easily available (future load characteristics). Furthermore, the *fixed* parameters cannot support good reactive schemes in *dynamic* environments such as multiprocessor systems, where the traffic on buses, the contention on memory modules and the load on processors are unpredictable to one process. For those relying on assumed distributions, the algorithms use too strong assumptions. Generally, the inputs in the dynamic environment such as waiting time distributions on a shared object are unpredictable.

Ideally, reactive algorithms should be able to observe the changes in the environment and react accordingly. Based on that purpose, online algorithms and competitive analysis seem to be a promising approach. With this idea in mind, we have presented two reactive shared objects that react fast to the environment variations with no need of manually tuned parameters nor any assumptions on probability distributions of inputs. The experiments

on a real commercial system, the SGI Origin2000, show promising results for these reactive shared objects.

In the future, we plan to look into new reactive schemes that may further improve the performance of reactive shared objects. The reactive schemes used in this thesis are based on competitive online techniques that provide good behavior against a malicious adversary. In the high performance setting, a different 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.

Bibliography

[1] A. Agarwal and M. Cherian. Adaptive backoff synchronization techniques. In *Proceedings of the 16th Annual International Symposium on Computer Architecture*, pages 396–406, Jerusalem, Israel, May 28–June 1, 1989. IEEE Computer Society TCCA and ACM SIGARCH.

[2] 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.

[3] J. H. Anderson and M. Moir. Universal constructions for large objects. In *Proceedings of the 9th International Workshop on Distributed Algorithms*, pages 168–182. Springer-Verlag, 1995.

[4] J. H. Anderson, S. Ramamurthy, and R. Jain. Implementing wait-free objects on priority-based systems. In *Proceedings of the sixteenth annual ACM symposium on Principles of distributed computing*, pages 229–238. ACM Press, 1997.

[5] T. E. Anderson. The performance of spin lock alternatives for shared-money multiprocessors. *IEEE Trans. Parallel Distrib. Syst.*, 1(1):6–16, 1990.

[6] J. Aspnes, M. Herlihy, and N. Shavit. Counting networks. *J. ACM*, 41(5):1020–1048, 1994.

[7] G. Barnes. A method for implementing lock-free shared-data structures. In *Proceedings of the fifth annual ACM symposium on Parallel algorithms and architectures*, pages 261–270. ACM Press, 1993.

[8] A. Borodin and R. El-Yaniv. Online computation and competitive analysis. *Cambridge University Press*, 1998.

[9] D. E. Culler, J. P. Singh, and A. Gupta. Parallel architecture: A hardware/software approach. *Morgan Kaufmann Publishers*, 1998.

[10] G. Della-Libera and N. Shavit. Reactive diffracting trees. *J. Parallel Distrib. Comput.*, 60(7):853–890, 2000.

[11] 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.

[12] G. Granunke and S. Thakkar. Synchronization algorithms for shared-memory multiprocessors. *Computer*, 23(6):60–69, 1990.

[13] M. Greenwald. Non-blocking synchronization and system design. *PhD thesis*, 1999.

[14] M. Greenwald. Two-handed emulation: how to build non-blocking implementations of complex data-structures using dcas. In *Proceedings of the twenty-first annual symposium on Principles of distributed computing*, pages 260–269. ACM Press, 2002.

[15] M. Greenwald and D. Cheriton. The synergy between non-blocking synchronization and operating system structure. In *Proceedings of the second USENIX symposium on Operating systems design and implementation*, pages 123–136. ACM Press, 1996.

[16] P. H. Ha, M. Papatriantafilou, and P. Tsigas. Self-adjusting trees. Technical Report 2003-09, Computing Science, Chalmers University of Technology, October 2003.

[17] T. L. Harris. In *Personal Communication*, August 2002.

[18] T. L. Harris, K. Fraser, and I. A. Pratt. A practical multi-word compare-and-swap operation. In *Proceedings of the 16th International Conference on Distributed Computing*, pages 265–279. Springer-Verlag, 2002.

[19] M. Herlihy. Wait-free synchronization. *ACM Transaction on Programming and Systems*, 11(1):124–149, Jan. 1991.

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

[21] 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, July 1990.

[22] A. Israeli and L. Rappoport. Disjoint-access-parallel implementations of strong shared memory primitives. In *Proceedings of the thirteenth annual ACM symposium on Principles of distributed computing*, pages 151–160. ACM Press, 1994.

[23] D. N. Jayasimha. Parallel access to synchronization variables. In *Proceedings of the 1987 International Conference on Parallel Processing (ICPP'87)*, pages 97–100, University Park, Penn., Aug. 1987. Penn State. CSRD TR#630 jan. '87.

[24] A. R. Karlin, K. Li, M. S. Manasse, and S. Owicki. Empirical studies of competitve spinning for a shared-memory multiprocessor. In *Proceedings of the thirteenth ACM symposium on Operating systems principles*, pages 41–55. ACM Press, 1991.

[25] L. Lamport. A fast mutual exclusion algorithm. *ACM Trans. Comput. Syst.*, 5(1):1–11, 1987.

[26] J. Laudon and D. Lenoski. The sgi origin: A ccnuma highly scalable server. In *Proceedings of the 24th Annual International Symposium on Computer Architecture (ISCA-97)*, Computer Architecture News, pages 241–251. ACM Press, 1997.

[27] B. Lim. Reactive synchronization algorithms for multiprocessors. *PhD. Thesis*, 1995.

[28] 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.

[29] M. M. Michael and M. L. Scott. Nonblocking algorithms and preemption-safe locking on multiprogrammed shared memory multiprocessors. *J. Parallel Distrib. Comput.*, 51(1):1–26, 1998.

[30] M. Moir. Practical implementations of non-blocking synchronization primitives. In *Proceedings of the sixteenth annual ACM symposium on Principles of distributed computing*, pages 219–228. ACM Press, 1997.

[31] M. Moir. Transparent support for wait-free transactions. In *Proceedings of the 11th International Workshop on Distributed Algorithms*, pages 305–319. Springer-Verlag, 1997.

[32] N. Shavit and D. Touitou. Software transactional memory. In *Proceedings of the fourteenth annual ACM symposium on Principles of distributed computing*, pages 204–213. ACM Press, 1995.

[33] N. Shavit, E. Upfal, and A. Zemach. A steady state analysis of diffracting trees. *Theory of Computing Systems*, 31(4):403–423, 1998.

[34] N. Shavit and A. Zemach. Diffracting trees. *ACM Trans. Comput. Syst.*, 14(4):385–428, 1996.

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

[36] P. Tsigas and Y. Zhang. Evaluating the performance of non-blocking synchronization on shared-memory multiprocessors. In *Proceedings of the 2001 ACM SIGMETRICS international conference on Measurement and modeling of computer systems*, pages 320–321. ACM Press, 2001.

[37] P. Tsigas and Y. Zhang. Integrating non-blocking synchronisation in parallel applications: Performance advantages and methodologies. In *Proceedings of the 3rd ACM Workshop on Software and Performance (WOSP'02)*, pages 55–67. ACM press, July 2002.

[38] J. Turek and D. Shasha. The many faces of consensus in distributed systems. *IEEE Computer*, 25(2):8–17, 1992.

[39] J. Valois. Lock-free data structures. *PhD. Thesis*, 1995.

[40] R. Wattenhofer and P. Widmayer. An inherent bottleneck in distributed counting. *Journal of Parallel and Distributed Computing*, 49(1):135–145, 25 Feb. 1998.