Abstract

Title of Dissertation: Scalable Synchronization in Shared Memory Multiprocessing Systems

Jae-Heon Yang, Doctor of Philosophy, 1994

Dissertation directed by: Assistant Professor James H. Anderson
Department of Computer Science

It is our thesis that scalable synchronization can be achieved with only minimal hardware support, specifically read/write atomicity. This is contrary to the conventional viewpoint that stronger hardware support is required for scalable synchronization; such support not only requires additional cost, but also leads to portability problems.

As evidence in support of our thesis, we present a new scalable mutual exclusion algorithm based on read and write instructions. The performance of this algorithm is better than prior mutual exclusion algorithms based on read/write atomicity, and even rivals that of the fastest mutual exclusion algorithms that require stronger primitives. Our algorithm is based on the technique of *local spinning*, i.e., busy-waiting on variables that are locally-accessible to the waiting process. Local-spinning minimizes remote accesses of shared memory, which tend to dominate performance under heavy contention.

An obvious question left open by the algorithm described above is whether it is possible to synchronize even more efficiently using only reads and writes. We partially address this question by investigating time bounds for the mutual exclusion problem. These time bounds are based on a time complexity measure that counts only remote accesses of shared variables; local accesses are ignored.

Our time bounds establish trade-offs between time complexity and "write-" and "access-contention". The *write-* (*access-contention*) of a concurrent program is the number of processes that may be simultaneously enabled to write (access) the same shared variable. We show that, for any N-process mutual exclusion algorithm, if write-contention is w, and if at most v remote variables can be accessed atomically, then there exists an execution involving only one process in which that process executes $\Omega(\log_{vw} N)$ remote operations for entry into its critical section. We further show that, among these operations, $\Omega(\sqrt{\log_{vw} N})$ distinct remote variables are accessed. For algorithms with access-contention c, we show that the latter bound can be improved to $\Omega(\log_{vc} N)$. These results imply that our mutual exclusion algorithm is optimal if write-contention is limited to a constant; it remains an open problem whether better time complexity can be achieved with higher write-contention in algorithms based on read/write atomicity. It is worth noting that the time bounds that we establish apply to a wide class of synchronization primitives, not just reads and writes.

Since most scalable synchronization algorithms that have been proposed are based on local-spin techniques, it is natural to seek to determine the necessary level hardware support for the use of such techniques. We show that on distributed shared memory machines, only weak hardware support is required, namely atomic read and write instructions. In particular, we show that any

shared object can be implemented on such machines from single-reader, single-writer boolean variables using local-spin techniques. These results provide further evidence in support of our thesis that only minimal hardware support is required for scalable synchronization.

Scalable Synchronization in Shared Memory Multiprocessing Systems

by

Jae-Heon Yang

Dissertation submitted to the Faculty of the Graduate School
of The University of Maryland in partial fulfillment
of the requirements for the degree of
Doctor of Philosophy
1994

Advisory Committee:

Assistant Professor James H. Anderson, Chairman/Advisor
Associate Professor Clyde P. Kruskal
Associate Professor Joel Saltz
Assistant Professor Rich Gerber
Professor Moon Jhong Rhee

Dedication

To my late father, to my mother, and to my wife Hyunjoo
for their love and support.

Acknowledgements

Glory to the Lord!

I would like to thank my dissertation advisor Jim Anderson. He guided and supported my dissertation research, with great care and patience. He taught me how to do research, and did it well. Without his help, I would not have pulled it off. I would also like to thank my dissertation committee. I am very grateful to A. Udaya Shankar and Rich Gerber for their advice when I needed it most.

I would like to thank Howard Gobioff for helping with the performance studies given in Section 2.6. I would also like to acknowledge Argonne National Laboratories and Lawrence Livermore National Laboratories for providing me with access to the machines used in these studies. I am particularly grateful to Terry Gaasterland at Argonne, and Tammy Welcome and Brent Gorda at Lawrence Livermore for their help. I would also like to thank Nir Shavit for his

helpful comments on an earlier draft of Chapter 2. I would like to thank Gadi Taubenfeld for prompting me to consider the bounds for cache-coherence in Section 3.5. I would also like to thank Sanglyul Min for his helpful comments on an earlier draft of Chapter 3.

My friends in the Department of Computer Science, including Kwan-Woo Ryu, Sungzoon Cho, Bongki Moon, Sam Hyuk Noh, Kyuseok Shim, Seongsoo Hong, Tae-Hyung Kim, Soo-Mook Moon, Mark Moir, Nancy Lindley, and many others, have made my journey enjoyable. Friends in the St. Andrew Kim Church have been wonderful company in the faith. I am very thankful to Rev. Kim, Rev. Lee, Rev. Song, and Rev. Ham for their spiritual guidance.

Financial support from the Department of Computer Science, the Graduate School of the University of Maryland, and the Korean Ministry of Education is gratefully acknowledged.

My families in Korea made my pursuit of dissertation research possible through their sacrifices. I cannot thank them enough. I am very thankful to my mother who has kept her very best for me. I thank my late father, who passed away last Fall, in my prayers.

I thank my wife Hyunjoo for everything. I would not say more, because she knows it all. I love her for her love.

Table of Contents

List of Figures

Scalable Synchronization in Shared Memory Multiprocessing Systems

Jae-Heon Yang

August 5, 1994

This comment page is not part of the dissertation.

Typeset by LaTeX using the `dissertation` style by Pablo A. Straub, University of Maryland.

Chapter 1

Introduction

Advances in hardware technology have made processors faster and more affordable than ever. Although the current crop of processors are much faster than older ones, a single processor cannot always meet the ever-increasing demand for more processing power. For this reason, considerable research effort has been concentrated on the utilization of multiple processors. As a result of such efforts, a variety of multiprocessing systems are commercially available today.

If a multiprocessing system supports a single address space by hardware, then it is called a *shared-memory multiprocessor*. Otherwise, i.e., if a multiprocessing system has multiple address spaces, then it is called a *message-passing multiprocessor*, or simply a *multicomputer*. In this dissertation, we focus on shared-memory multiprocessors.

In a *shared memory concurrent program*, a collection of sequential programs called *processes* cooperate with each other by sharing variables in a commonly-accessible memory. The execution of such a program may be considered as an interleaving of the executions of its component processes. When processes interact, not all possible interleavings are desirable. In such instances, processes must

be *synchronized* to prevent unacceptable interleavings [10]. Synchronization is not without its cost; it almost always decreases the level of concurrency, and hence degrades performance. A synchronization algorithm is said to be *scalable* if increasing the number of processes to be synchronized does not degrade performance dramatically. Scalable synchronization methods are of great importance in concurrent programming. In many applications, inefficient synchronization may defeat the purpose of employing multiple processors, namely executing a program faster. In this thesis, we develop synchronization methods that can be efficiently used even when a large number of processes need to be synchronized, and investigate fundamental costs of synchronization in shared-memory multiprocessing systems.

The rest of the chapter is organized as follows. In Section 1.1 a simple description of shared-memory multiprocessors is given. Past research on synchronization is surveyed in Section 1.2. Finally, Section 1.3 summarizes the results of the thesis.

1.1 Shared-Memory Multiprocessors

A *shared-memory multiprocessing system* consists of a set of processors, a set of memory modules, and an interconnection network. In shared-memory multiprocessing systems, all memory modules form a single address space. In other words, all processors may access every location in every memory module. The single address space relieves programmers from problems of data partitioning, which are known to be some of the most difficult problems in programming parallel machines. The shared address space also provides better support for parallelizing

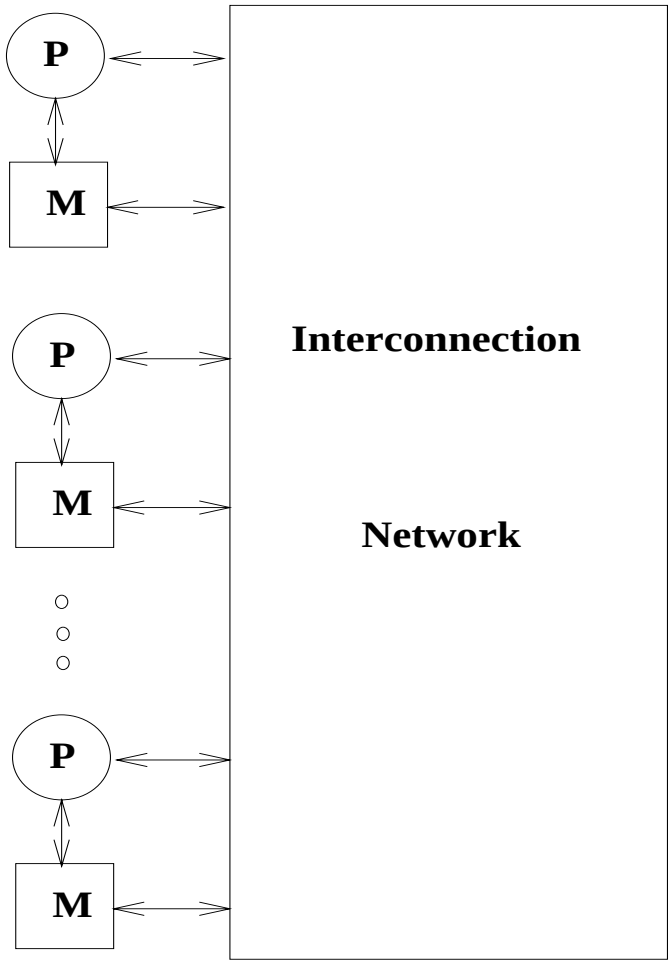

Figure 1.1: Distributed shared memory

compilers, multiprogramming, and standard operating systems. These features make a shared-memory multiprocessor easier to program than a message-passing machine [41].

In the rest of this section, we briefly discuss two architectural paradigms for shared-memory multiprocessors, namely distributed shared memory and cache-coherent memory, and examine some multiprocessors that adopt such paradigms.

Distributed Shared Memory Machines

A distributed shared-memory multiprocessor is depicted in Figure 1.1. Each processor has its own memory module, which is connected by a private data path. Other processors may access the memory module only by traversing the global interconnection network. Distributing memory with the processors is desirable because it enables programs to exploit locality. Some references to shared variables and all references to private variables and codes can be made local to each processor. These references are served without the longer latency of remote references, resulting in reduced bandwidth demands on the global interconnect [41].

The BBN TC2000 is an example of a distributed shared-memory multiprocessor [13]. The TC2000 consists of a number of nodes, each of which contains a processor and a memory unit. The nodes are connected via a multi-stage interconnection network, known as the Butterfly switch. Each access to a remote memory location (i.e., one that requires a traversal of the interconnection network) takes about 2 microseconds, whereas each local reference takes about 0.6 microseconds. Each node's processor, a Motorola 88100, provides an atomic fetch-and-store instruction called `xmem` as a synchronization primitive. The TC2000 has cache memory for private data, but does not provide a cache coherence mechanism for shared data.

Coherent-Cache Machines

In Figure 1.2, a cache-coherent multiprocessor is depicted. Each processor is equipped with its own cache. If a variable in some memory module is accessed by a processor for the first time, it is copied into the accessing processor's cache. Further accesses to the variable by the same processor may be served by the lo-

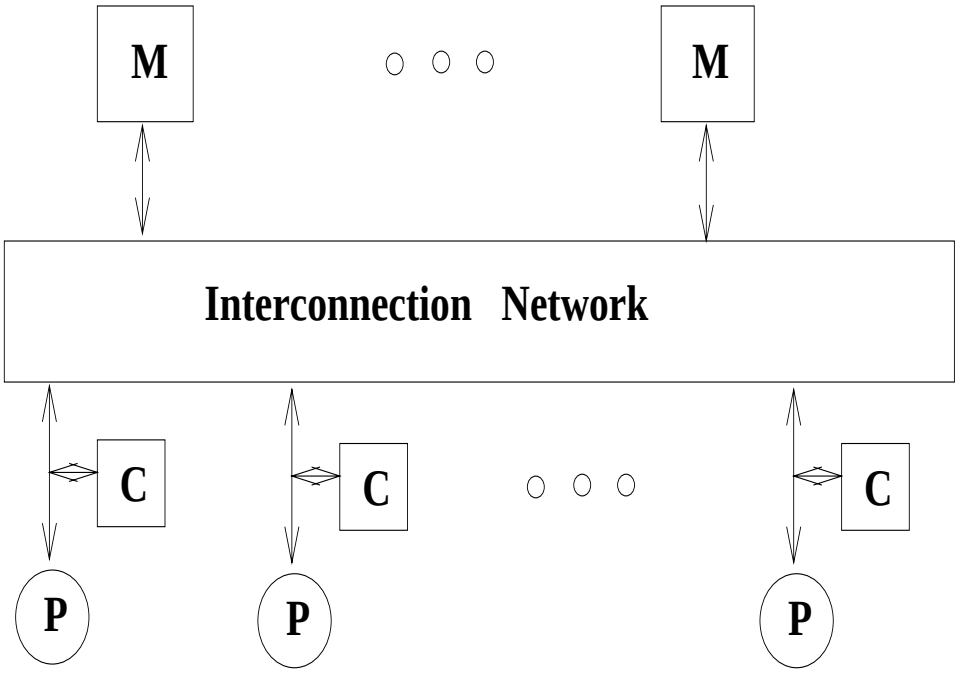

Figure 1.2: Coherent-cache

cal cache. Caching techniques enable shared-memory multiprocessors to achieve better performance by reducing memory latency. However, caching shared variables may introduce inconsistent copies of a shared variable. In order to maintain consistency among multiple copies of a variable (in a memory module and possibly multiple caches), a cache-coherence protocol must be provided by hardware, or by software, or by a combination of both [11, 15, 19]. When a new value is written to a cached variable, a cache-coherence protocol either invalidates other copies of the variable, or updates those copies to the new value.

The Sequent Symmetry is an example of a cache-coherent shared-memory multiprocessor [55]. On the Symmetry, the processors and memory nodes are interconnected via a shared bus. In other words, all processors, memory modules, and I/O controllers plug into a single bus. A processor node consists of an Intel

80386 or Intel 80486 and a 64 Kbyte, two-way set-associative cache. Cache coherence is maintained by a snoopy protocol. Snoopy protocols are almost always adopted in shared bus machines. Under this protocol, every cache snoops the traffic on the shared bus. If a variable is written, every processor that has a copy of that variable in its local cache either invalidates or updates its copy.

The Kendall Square Research KSR1 multiprocessor is another example of a cache-coherent shared-memory multiprocessor [35]. The interconnection network of the KSR1 is formed of hierarchical rings. Scalability is achieved by connecting 32 processors to a ring that operates at one GB/sec. Interconnection bandwidth within a ring scales linearly, because every ring slot may contain a transaction [14]. The current KSR1 machine uses a two level hierarchy to interconnect 34 rings, and scales up to 1088 processors. As the hierarchical ring may have an arbitrary number of levels, more processors could be added.

A unique feature of the KSR1 multiprocessor is its ALLCACHE[1] mechanism. The operation of the ALLCACHE mechanism is similar to that of other coherent caches: data is transferred to a processor's local cache when accessed by that processor. The difference is that the memory of all the processors is part of a 40-bit virtual address space managed as a cache. In other words, there is no typical main memory other than caches. This is sometimes called a cache only memory architecture (COMA). The ring is used to convey cache lines to service cache misses.

[1]ALLCACHE is a registered trademark of Kendall Square Research

Other Shared-Memory Multiprocessors

Observe that, in both architectural paradigms described above, some memory location may be closer to a processor than other memory locations. More precisely, the time to access memory is not uniform. A remote memory access requires a traversal of the global interconnect between processors and shared memory, while a local or cache access does not. So, local memory accesses or cache accesses are much faster than remote memory accesses.

In fact, the above two paradigms – distributed shared memory and cache-coherent memory – provide a means to exploit locality in programs, and both may be adopted in the same system. The DASH multiprocessor is an example of such a system [41]. The DASH machine is a shared-memory multiprocessor that provides a single address space with a distributed memory and coherent caches.

Most commercial multiprocessors with coherent caches, including the Sequent Symmetry, rely on snooping to maintain coherence. However, straightforward snooping schemes require that all caches observe every memory request from every processor. This results in limited scalability because the common bus and the individual processor caches easily saturate.

A directory-based cache coherence scheme is adopted in DASH to avoid the scalability problem of snoopy schemes. This scheme eliminates the need to broadcast every memory request to all processor caches [40]. The directory keeps a record of the processor caches that hold a copy of each memory block. Because only the caches with copies may be affected by accesses to the memory block, only those caches need be notified of such accesses. Thus, such notifications can be handled by point-to-point messages, instead of broadcasts. Unlike most snoopy schemes that depend on shared buses, the directory-based coherence mechanism

is not dependent on any specific interconnection network, and hence may employ scalable, low-latency networks such as Omega networks or k-ary n-cubes used by non-cache-coherent and message-passing machines.

1.2 A History of Synchronization

In the design of any shared-memory multiprocessing system, provisions must be made for the implementation of atomic operations. An operation is *atomic* if its execution is semantically indivisible, i.e., if it "appears" to take effect instantaneously. A related notion is the concept of "granularity": an operation is said to be *fine-grained* if it can be easily implemented in terms of low-level machine instructions, and is said to be *coarse-grained* otherwise. The notion of granularity is inherently architecture-dependent. For example, on a shared-memory multiprocessor that provides only atomic reads and writes as synchronization primitives, read and write operations that access a single memory location are usually taken to be fine-grained, whereas operations that access multiple memory locations or that perform multiple reads or writes to a given location are considered coarse-grained.

When implementing coarse-grained atomic operations from fine-grained ones, it is necessary to ensure that coarse-grained operations of different processes do not adversely interfere with one another. Typically, such interference is prevented by implementing coarse-grained operations as "critical sections". Such an implementation requires the existence of a synchronization protocol for ensuring that critical sections are executed fairly and in a mutually exclusive manner. The problem of designing such a protocol has come to be regarded as one of the most

fundamental problems in concurrent programming, i.e., the well-known *mutual exclusion problem*. Algorithms that provide mutual exclusion by busy-waiting are commonly called *spin locks*.

The mutual exclusion problem was first formally stated and solved in a seminal paper by Dijkstra [21]. In this problem, each of a set of processes repeatedly executes a program fragment known as its critical section. Before executing its critical section, a process must first execute another program fragment, its "entry section", and upon termination of its critical section, a third program fragment, its "exit section". The entry and exit sections must be designed so that (i) at most one process executes its critical section at any time, and (ii) each process in its entry section eventually executes its critical section. The former is known as the *mutual exclusion* property, and the latter is known as the *starvation-freedom* property. In some variants of the problem, starvation-freedom is replaced by the weaker requirement of *livelock-freedom*: if some process is in its entry section, then some process eventually executes its critical section.

Dijkstra's original solution is depicted in Figure 1.3. His algorithm satisfied the livelock-freedom property, but not the starvation-freedom property. The first starvation-free solution was presented by Knuth in [34]. Of the many early solutions to the mutual exclusion problem, most are quite complicated and difficult to understand. A notable exception is an especially simple solution first presented by Peterson in [50] and later refined by Kessels in [32]. The approach taken by Kessels was to first solve the mutual exclusion problem for two processes, and to then use the two-process solution in a binary arbitration tree to solve the N-process case. Kessels' algorithm was the first solution that required fewer than $O(N)$ operations per critical section execution in the absence of contention.

```
shared var B, C : array[0..N − 1] of boolean;
            K : 0..N − 1;
initially   (∀i :: B[i] = true ∧ C[i] = true)

process i
private var j : 0..N;
while true do
        Noncritical Section;
        B[i] := false;
  LOOP: if K ≠ i then
            C[i] := true;
            if B[K] then K := i fi;
            goto LOOP
        else
            C[i] := false;
            j := 0;
            while (j < N) do
                if j ≠ i ∧ ¬C[j] then goto LOOP fi;
                j := j + 1
            od
        fi;
        Critical Section;
        C[i] := true;
        B[i] := true
od
```

Figure 1.3: Dijkstra's mutual exclusion algorithm.

Early solutions to the mutual exclusion problem, including all of the algorithms mentioned so far, required only minimal hardware support, specifically atomic read and write instructions. Unfortunately, such early solutions suffered from two serious shortcomings: first, nearly all were rather daunting from a conceptual standpoint; second, most require the execution of many instructions, even when there is no contention at all between processes. The need for simpler and faster solutions to the mutual exclusion problem ultimately resulted in the design of multiprocessing systems with synchronization mechanisms that are more sophisticated than simple reads and writes, and correspondingly, solutions to the mutual exclusion problem based on these new mechanisms. Examples of such mechanisms — hereafter called *strong primitives* — include the fetch-and-store, compare-and-swap, and fetch-and-add instructions.

Early solutions based on strong primitives, such as the familiar test-and-set lock, were conceptually simple, but resulted in somewhat poor performance. More recently, queue-based spin locks have been proposed by Anderson [9], by Graunke and Thakkar [26], and by Mellor-Crummey and Scott [45] that exhibit better performance; these locks are implemented using fetch-and-add, fetch-and-store, or compare-and-swap instructions, respectively. These algorithms exhibit good scalability when used on multiprocessors that permit shared variables to be locally accessible, as is the case if coherent caching schemes are employed, or if shared variables can be allocated in a local portion of distributed shared memory. The key to their performance is the idea of *local spinning*, i.e., busy-waiting on variables that are locally-accessible to the waiting process. By relying on local spinning as the sole mechanism by which processes wait, these algorithms induce minimal memory and interconnect contention. This stands in sharp contrast to

the case of earlier locking algorithms, such as Dijkstra's in Figure 1.3, in which processes busy-wait on nonlocal memory locations.

In addition to the software-based solutions to the mutual exclusion problem described above, a number of hardware-based implementations have been proposed. Of particular interest are the solutions given by Goodman, Vernon, and Woest in [25], and by Lee and Ramachandran in [39], which exploit underlying cache coherence mechanisms. These implementations involve the construction of a distributed queue in hardware. The basic idea is to form a queue of processes contending for the lock by having each processor spin on its own cache line; this technique avoids the generation of unnecessary interconnect traffic.

1.3 Summary of Results

The goal of this dissertation is to determine the hardware support required for scalable synchronization in shared-memory multiprocessors. It is our thesis that scalable synchronization can be achieved with only minimal hardware support, specifically read/write atomicity. This is contrary to the conventional viewpoint that stronger hardware support, which resulted in additional cost and portability problems, is necessary for scalable synchronization.

In this dissertation, the time complexity of a concurrent program is measured by counting only remote accesses of shared variables; local accesses are ignored. This complexity measure is proposed as a new metric of scalability. As evidence in support of our thesis, we present a new scalable mutual exclusion algorithm based on read and write instructions whose time complexity is better than that of any other mutual exclusion algorithms based on read/write atomicity. The

performance of this algorithm rivals that of the fastest mutual exclusion algorithms that require stronger primitives. Our algorithm is based on the technique of *local spinning*, i.e., busy-waiting on variables that are locally-accessible to the waiting process.

An obvious question left open by the algorithm described above is whether it is possible to synchronize even more efficiently using only reads and writes. We partially address this question by investigating time bounds for mutual exclusion problem. Our time bounds establish trade-offs between time complexity and "write-" and "access-contention". The *write-* (*access-contention*) of a concurrent program is the number of processes that may be simultaneously enabled to write (access) the same shared variable. We show that, for any N-process mutual exclusion algorithm, if write-contention is w, and if at most v remote variables can be accessed atomically, then there exists an execution involving only one process in which that process executes $\Omega(\log_{vw} N)$ remote operations for entry into its critical section. We further show that, among these operations, $\Omega(\sqrt{\log_{vw} N})$ distinct remote variables are accessed. For algorithms with access-contention c, we show that the latter bound can be improved to $\Omega(\log_{vc} N)$. These results imply that our mutual exclusion algorithm is optimal if write-contention is limited to a constant; it remains an open problem whether better time complexity can be achieved with higher write-contention in algorithms based on read/write atomicity. Our results apply to wide classes of synchronization primitives, not just reads and writes.

Since most scalable synchronization algorithms that have been proposed are based on local-spin techniques, it is natural to seek to determine the "right" level hardware support for the use of such techniques. We show that from a *compu-*

tational standpoint, only weak hardware support is required, namely atomic read and write instructions. In particular, we show that any atomic operation can be implemented using only read and write instructions and local spinning. These results provide further evidence in support of our thesis that only minimal hardware support is required for scalable synchronization.

We now consider the main contributions of this dissertation in more details.

Scalable Mutual Exclusion Algorithms. In Chapter 2, we present a new scalable mutual exclusion algorithm based on read and write instructions. We also present an interesting extension of this algorithm. In this extension, only a constant number of memory references are required for acquiring a lock in the absence of contention. Our algorithms are based on the local-spin synchronization techniques.

In order to formally study the scalability of concurrent programs, we propose a time complexity measure for concurrent programs that captures the communication overhead in synchronization algorithms. Under our proposed measure, the complexity of a concurrent program is measured by counting only remote accesses of shared variables; local accesses are ignored. This measure satisfies two criteria that must be met by any reasonable complexity measure. First, it is conceptually simple. In fact, this measure is a natural descendent of the standard time complexity measure used in sequential programming. Second, as demonstrated by a number of published performance studies, this measure has a tangible connection with real performance. All other proposed time complexity measures for concurrent programs that we know of fail to satisfy at least one of these criteria.

Our mutual exclusion algorithm has $O(\log N)$ time complexity using the pro-

posed measure. The time complexity of this algorithm is better than that of all prior solutions to the mutual exclusion problem that are based upon atomic read and write instructions. Performance studies conducted on the BBN TC2000 and Sequent Symmetry multiprocessors indicate that our algorithms exhibit scalable performance under heavy contention. In fact, our spin lock algorithm outperforms all prior algorithms based on read/write atomicity, and its performance under heavy contention rivals that of the fastest queue-based locks that employ strong primitives such as compare-and-swap or fetch-and-add.

Time/Contention Trade-Offs for Multiprocessor Synchronization. In Chapter 3, we investigate the costs inherent to synchronization in shared-memory multiprocessing systems. In particular, we show that there are trade-offs between contention and communication, which fundamentally limit the scalability of synchronization algorithms.

The amount of communication in a concurrent program is captured by using the complexity measure proposed in Chapter 2, i.e., by counting remote memory references.

On many large scale shared-memory multiprocessors, multistage interconnection networks are employed to get a high bandwidth connection between processors and memory modules. Pfister and Norton [53] have shown that when many processors request access to the same memory location, making it a highly contended variable called *hot spot*, a tree-shaped saturation builds up in the interconnection network, resulting in performance degradation not only for those processors accessing the hot spot, but other processors as well.

To formally model contention for shared memory locations, we define the *write-* (*access-contention*) of a concurrent program as the number of processes

that may be simultaneously enabled to write (access) the same shared variable. Programs with high write- or access-contention are susceptible to hot spot contention.

In Chapter 3, we show that, for any N-process mutual exclusion algorithm, if write-contention is w, and if at most v remote variables can be accessed atomically, then there exists an execution involving only one process in which that process executes $\Omega(\log_{vw} N)$ remote operations for entry into its critical section. We further show that, among these operations, $\Omega(\sqrt{\log_{vw} N})$ distinct remote variables are accessed. For algorithms with access-contention c, we show that the latter bound can be improved to $\Omega(\log_{vc} N)$.

These results have a number of important implications. For example, the latter two bounds imply that a communication/contention trade-off exists even if coherent caching techniques are employed. Also, because the execution that establishes these bounds involves only one process, it follows that so-called fast mutual exclusion algorithms — i.e., algorithms that require a process to execute only a constant number of memory references in the absence of competition — require arbitrarily high write-contention in the worst case. Because the competition for critical section is likely to be low in many well-designed systems, achieving fast mutual exclusion in the absence of competition is of great practical interest [37].

In most shared-memory multiprocessors, an atomic operation may access only a constant number of remote variables. In fact, most commonly-available synchronization primitives (e.g., read, write, test-and-set, fetch-and-store, compare-and-swap, and fetch-and-add) access only one remote variable. In this case, the first and the last of our bounds are asymptotically tight. These results also show

that our N-process $\Theta(\log_2 N)$ mutual exclusion algorithm based on read/write atomicity is optimal. This algorithm has access-contention (and hence write-contention) two.

Hardware Support for Scalable Synchronization. In Chapter 4, we turn our attention to implementations of shared objects based on critical sections. In particular, we investigate the applicability of local-spin techniques within such implementations.

The conventional viewpoint is that powerful hardware-based mechanisms are required for scalable implementations of atomic operations. This conventional viewpoint has led to the design of multiprocessing systems with synchronization mechanisms that are stronger than atomic reads and writes, and has resulted in implementation techniques based on these mechanisms. Such mechanisms, however, require additional hardware support. For example, Silicon Graphics' PowerSeries machines are built using a multiple number of R2000/R3000 microprocessors, which do not have any strong primitives. In order to implement test-and-set, a synchronization memory with a special interconnect was inevitably developed [24]. In addition, algorithms based on strong primitives may be inefficient on machines that do not support such primitives in hardware. Because different architectures are likely to provide different sets of strong primitives, algorithms based on such primitives are often of limited portability.

In Chapter 4, we call into question the assumption that such costs are inevitable. Specifically, we show that very weak hardware support is sufficient for local-spin synchronization. In this chapter, we prove that any shared objects, no matter how complicated, can be implemented without global busy-waiting from single-reader, single-writer, boolean variables, in distributed shared mem-

ory multiprocessors.

In Chapter 5, we summarize the results obtained in this dissertation and suggest directions for future research.

Chapter 2

Scalable Mutual Exclusion Algorithms

2.1 Introduction

Most early solutions to the mutual exclusion problem required only minimal hardware support, specifically atomic read and write instructions. Although of theoretical importance, most such algorithms were judged to be impractical from a performance standpoint, leading to the development of solutions requiring stronger hardware support such as read-modify-write operations. The poor performance of early read/write algorithms stems partially from two factors. First, such algorithms are not scalable, i.e., performance degrades dramatically as the number of contending processes increases. Second, even in the absence of contention, such algorithms require a process contending for its critical section to execute many operations.

The second of these two problems has subsequently been addressed, specifically by Lamport in [37], where a read/write algorithm is given that requires only a constant number of operations per critical section acquisition in the absence of contention. Following the title of Lamport's paper, such algorithms

have come to be known simply as "fast mutual exclusion algorithms". This designation is somewhat of a misnomer, as such algorithms are not necessarily fast in the *presence* of contention. In fact, the problem of designing a scalable algorithm requiring only read/write atomicity has remained open. In this chapter, we present such an algorithm.

In a recent paper [6], Anderson presented a mutual exclusion algorithm that uses only local spins and that requires only atomic read and write operations. In his algorithm, each of N processes executes $O(N)$ remote operations to enter its critical section whether there is contention or not. All other previously published mutual exclusion algorithms that are based on atomic reads and writes employ global busy-waiting and hence induce an unbounded number of remote operations under heavy contention. Most such algorithms also require $O(N)$ remote operations in the absence of contention. Some exceptions to the latter include the algorithm given by Kessels in [32] and the previously mentioned one given by Lamport in [37]. Kessels' algorithm generates $O(\log N)$ remote operations in the absence of contention, while Lamport's generates $O(1)$. A variant of Lamport's algorithm has recently been presented by Styer in [56]. Although Styer claims that his algorithm is more scalable than Lamport's, in terms of time complexity, they are actually very similar: both generate unbounded remote operations under heavy contention and $O(1)$ operations in the absence of contention. Styer's claims of scalability are predicated upon complexity calculations that ignore operations performed within busy-waiting constructs. Because the processes in his algorithm busy-wait on remote variables, such complexity calculations do not give a true indication of scalability. Another recent algorithm of interest is one given by Michael and Scott in [47]. Although this algorithm generates $O(N)$

remote memory references in the presence of contention and $O(1)$ in the absence of contention, it requires both full- and half-word reads and writes to memory, which is a level of hardware support more powerful than ordinary read/write atomicity.

In this chapter, we present a new mutual exclusion algorithm that requires only atomic reads and writes and in which all spins are local. Our algorithm induces $O(\log N)$ remote operations under any amount of contention, and thus is an improvement over the algorithm given by Anderson in [6]. We also present a modified version of this algorithm that requires only $O(1)$ remote operations in the absence of contention. Unfortunately, in this modified algorithm, worst-case complexity rises to $O(N)$. However, we argue that this $O(N)$ behavior is rare, occurring only when transiting from a period of high contention to a period of low contention. Under high contention, this modified algorithm induces only $O(\log N)$ remote operations. It is worth noting that our algorithm and its variation are starvation-free, whereas some of the aforementioned algorithms are not.

The rest of the chapter is organized as follows. In Section 2.2, we present our model of concurrent programs. The above-mentioned mutual exclusion algorithm is then presented in Section 2.3, and its correctness proof is given in Section 2.4. In Section 2.5, we consider the modified version of the algorithm discussed above. In Section 2.6, we present results from performance studies conducted on the BBN TC2000 and Sequent Symmetry multiprocessors. These studies indicate that our mutual exclusion algorithm exhibits scalable performance under heavy contention. We end the chapter with a short discussion in Section 2.7.

2.2 Definitions

In this section, we present our model of concurrent programs and define the relations used in reasoning about such programs. A *concurrent program* consists of a set of processes and a set of variables. A *process* is a sequential program consisting of labeled statements. Each *variable* of a concurrent program is either private or shared. A *private variable* is defined only within the scope of a single process, whereas a *shared variable* is defined globally and may be accessed by more than one process. Each process of a concurrent program has a special private variable called its *program counter*: the statement with label k in process p may be executed only when the value of the program counter of p equals k. For an example of the syntax we employ for programs, see Figure 2.1.

A program's semantics is defined by its set of "fair histories". The definition of a fair history, which is given below, formalizes the requirement that each statement of a program is subject to weak fairness. Before giving the definition of a fair history, we introduce a number of other concepts; all of these definitions apply to a given concurrent program.

A *state* is an assignment of values to the variables of the program. One or more states are designated as *initial states*. If state u can be reached from state t via the execution of statement s, then we say that s is *enabled* at state t and we write $t \xrightarrow{s} u$. If statement s is not enabled at state t, then we say that s is *disabled* at t. A *history* is a sequence $t_0 \xrightarrow{s_0} t_1 \xrightarrow{s_1} \cdots$, where t_0 is an initial state. A history may be either finite or infinite; in the former case, it is required that no statement be enabled at the last state of the history. A history is *fair* if it is finite or if it is infinite and each statement is either disabled at infinitely many states of the history or is infinitely often executed in the history. Note

that this fairness requirement implies that each continuously enabled statement is eventually executed. Unless otherwise noted, we henceforth assume that all histories are fair.

With regard to complexity, we assume that each shared variable is *local* to at most one process and is *remote* to all other processes. This assumption is reflective of a distributed shared memory model. We refer to a statement execution as an *operation*. An operation is *remote* if it accesses remote variables, and is *local* otherwise.

Following [20], we define safety properties using *unless* assertions and progress properties using *leads-to* assertions. Consider two predicates P and Q over the variables of a program. The assertion P *unless* Q holds iff for any pair of consecutive states in any history of the program, if $P \wedge \neg Q$ holds in the first state, then $P \vee Q$ holds in the second. If predicate P is initially true and if P *unless false* holds, then predicate P is said to be an *invariant*. We say that predicate P *leads-to* predicate Q, denoted $P \mapsto Q$, iff for each history $t_0 \stackrel{s_0}{\rightarrow} t_1 \stackrel{s_1}{\rightarrow} \cdots$ of the program, if P is true at some state t_i, then Q is true at some state t_j where $j \geq i$.

2.3 Mutual Exclusion Algorithm

In this section, we present our mutual exclusion algorithm. First, we state more precisely the conditions that must be satisfied by such an algorithm. In the mutual exclusion problem, there are N processes, each of which has the following structure.

while *true* **do**

Noncritical Section;

Entry Section;

Critical Section;

Exit Section

od

It is assumed that each process begins execution in its noncritical section. It is further assumed that each critical section execution terminates. By contrast, a process is allowed to halt in its noncritical section. No variable appearing in any entry or exit section may be referred to in any noncritical or critical section (except, of course, program counters). A program that solves this problem is required to satisfy the mutual exclusion and starvation-freedom properties, given earlier in Section 1.2. As shown in Section 2.4, mutual exclusion can be stated formally as an invariant, and starvation-freedom as a leads-to property. We also require that each process in its exit section eventually enters its noncritical section; this requirement is trivially satisfied by our solution (and most others), so we will not consider it further.

As in [32], we first solve the mutual exclusion problem for two processes, and then apply our two-process solution in a binary arbitration tree to get an N-process solution. The two-process algorithm is depicted in Figure 2.1. The two processes are denoted u and v, which are assumed to be distinct, positive integer values.

The algorithm employs five shared variables, $C[u]$, $C[v]$, T, $P[u]$, and $P[v]$. Variable $C[u]$ ranges over $\{0, u, v\}$ and is used by process u to inform process v of its intent to enter its critical section. Observe that $C[u] = u \neq 0$ holds while

```
shared var C : array[u, v] of {0, u, v};
           P : array[u, v] of 0..2;
           T : {u, v}
initially  C[u] = 0 ∧ C[v] = 0 ∧ P[u] = 0 ∧ P[v] = 0

process u                                    process v

while true do                                /* symmetric to process id */
   0: Noncritical Section;
   1: C[u] := u;
   2: T := u;
   3: P[u] := 0;
   4: if C[v] ≠ 0 then
   5:    if T = u then
   6:       if P[v] = 0 then
   7:          P[v] := 1 fi;
   8:       while P[u] = 0 do /* null */ od;
   9:       if T = u then
  10:         while P[u] ≤ 1 do /* null */ od fi
          fi
       fi;
  11: Critical Section;
  12: C[u] := 0;
  13: if T ≠ u then
  14:    P[v] := 2 fi
od
```

Figure 2.1: Two-process mutual exclusion algorithm.

process u executes its statements 2 through 12, and $C[u] = 0$ holds otherwise. Variable $C[v]$ is used similarly. Variable T ranges over $\{u, v\}$ and is used as a tie-breaker in the event that both processes attempt to enter their critical sections at the same time. The algorithm ensures that the two processes enter their critical sections according to the order in which they update T. Variable $P[u]$ ranges over $\{0, 1, 2\}$ and is used by process u whenever it needs to busy-wait. Note that $P[u]$ is waited on only by process u, and thus can be stored in a memory location that is locally accessible to process u (in which case all spins

are local). Variable $P[v]$ is used similarly by process v.

Loosely speaking, the algorithm works as follows. When process u wants to enter its critical section, it informs process v of its intention by establishing $C[u] = u$. Then, process u assigns its identifier u to the tie-breaker variable T, and initializes its spinning location $P[u]$. If process v has not shown interest in entering its critical section, in other words, if $C[v] = 0$ holds when u executes statement 4, then process u proceeds directly to its critical section. Otherwise, u reads the tie-breaker variable T. If $T \neq u$, which implies that $T = v$, then u can enter its critical section, as the algorithm prohibits v from entering its critical section when $C[u] = u \ \wedge \ T = v$ holds (recall that ties are broken in favor of the first process to update T). If $T = u$ holds, then either process v executed statement 2 before process u, or process v has executed statement 1 but not statement 2. In the first case, u should wait until v exits its critical section, whereas, in the second case, u should be able to proceed to its critical section. This ambiguity is resolved by having process u execute statements 6 through 10. Statements 6 and 7 are executed by process u to release process v in the event that it is waiting for u to update the tie-breaker variable (i.e., v is busy-waiting at statement 8). Statements 8 through 10 are executed by u to determine which process updated the tie-breaker variable first. Note that $P[u] \geq 1$ implies that v has already updated the tie-breaker, and $P[u] = 2$ implies that v has finished its critical section. To handle these two cases, process u first waits until $P[u] \geq 1$ (i.e., until v has updated the tie-breaker), re-examines T to see which process updated T last, and finally, if necessary, waits until $P[u] = 2$ (i.e., until process v finishes its critical section).

shared var C : **array**$[0..\log_2 N - 1, 0..N-1]$ **of** $-1..N-1$;
P : **array**$[1..\log_2 N - 1, 0..N-1]$ **of** $0..2$;
T : **array**$[0..\log_2 N - 1, 0..N-1]$ **of** $0..N-1$
initially $(\forall j, i : 0 \leq j < \log_2 N :: C[j,i] = -1 \ \wedge \ P[j,i] = 0)$
define $(\forall l : 0 \leq l < N :: comp(l) \ \equiv$ **if even**(l) **then** $l+1$ **else** $l-1$ **fi**$)$

process i
private var $rival : -1..N-1$;
$j : 0..\log_2 N$;
$k : 0..(N-1)/2$;
$l : 0..N-1$
initially $j = 0 \ \wedge \ k = i/2 \ \wedge \ l = i$

Figure 2.2: Variable declarations for N-process mutual exclusion algorithm.

After executing its critical section, process u informs process v that it is finished by establishing $C[u] = 0$. If $T = v$, in which case process v is waiting to enter its critical section, then process u updates $P[v]$ in order to terminate v's busy-waiting loop.

As discussed above, the N-process case is solved by applying the above two-process algorithm in a binary arbitration tree. The resulting algorithm is shown in Figures 2.2 and 2.3. In this figures, process identifiers range over $\{0..N-1\}$, and for notational convenience, N is assumed to be a power of 2. Associated with each link in the tree is an entry section and an exit section. The entry and exit sections associated with the two links connecting a given node to its sons constitute a two-process mutual exclusion algorithm. Initially, all processes start at the leaves of the tree. To enter its critical section, a process is required to traverse a path from its leaf up to the root, executing the entry section of each

```
process i

while true do
    0: Noncritical Section;
    1: while j < log_2 N do
    2:     C[j,l] := i;
    3:     T[j,k] := i;
    4:     P[j,i] := 0;
    5:     rival := C[j,comp(l)];
    6:     if rival ≠ −1 then
    7:         if T[j,k] = i then
    8:             if P[j,rival] = 0 then
    9:                 P[j,rival] := 1 fi;
   10:             while P[j,i] = 0 do /* null */ od;
   11:             if T[j,k] = i then
   12:                 while P[j,i] ≤ 1 do /* null */ od fi
               fi
           fi;
   13:     j, k, l := j + 1, i/2^(j+2), i/2^(j+1)
       od;
   14: Critical Section;
   15: while j > 0 do
   16:     j, k, l := j − 1, i/2^j, i/2^(j−1)
   17:     C[j,l] := −1;
   18:     rival := T[j,k];
   19:     if rival ≠ i then
   20:         P[j,rival] := 2 fi
       od
od
```

Figure 2.3: N-process mutual exclusion algorithm.

link on this path. Upon exiting its critical section, a process traverses this path in reverse, this time executing the exit section of each link.

As seen in Figures 2.2 and 2.3, each variable of the two-process algorithm now has an additional subscript giving the appropriate level in the arbitration tree. As defined in Section 2.4, we use x/y to denote truncated integer division, i.e., $\lfloor x \div y \rfloor$. The expression $comp(l)$ is used to identify the C-variable of process i's two-process competitor at each level of the arbitration tree.

Although the concept of a binary arbitration tree is quite simple, one subtle problem does arise in the implementation. In particular, because each process waits on a local spin location at each level of the arbitration tree, it is important that each process knows the precise identity of any competitor within instances of the two process algorithm. For example, consider an algorithm with four processes, denoted 0 through 3. In the arbitration tree, process 0 first competes with process 1 at level 0, and then with one of processes 2 and 3 at level 1. At level 1 in the tree, both processes 2 and 3 access $C[1,1]$ and $T[1,0]$. However, at this level, each still has a unique spin location, namely $P[1,2]$ and $P[1,3]$, respectively. If process 0 encounters contention when competing at level 1, then it is important that process 0 knows precisely which of 2 or 3 it is competing against. Otherwise, for example, process 0 might update the spin location for process 2, $P[1,2]$, when in fact it is competing against process 3. This could result in a violation of either mutual exclusion or starvation-freedom. As seen in Figures 2.2 and 2.3, this "identity problem" is handled by means of the *rival* variable. Because of this problem, the two-process local-spin algorithm given by Anderson in [6] cannot be readily applied within an arbitration tree to get an N-process local-spin algorithm.

With regard to complexity, note that if variable $P[i]$ is local to process i in the two process algorithm, then process i executes a constant number of remote operations in its two-process entry and exit sections. It follows that, in the N-process algorithm, each process executes $O(\log N)$ remote operations in its (N-process) entry and exit sections.

2.4 Correctness Proof

In this section, we prove that the mutual exclusion and starvation-freedom properties hold for N-process mutual exclusion algorithm of Figures 2.2 and 2.3. A correctness proof for the simpler two-process algorithm is presented in [60]. We begin by presenting definitions and notational conventions that will be used in the remainder of the chapter. As in Section 2.3, we assume that the number of processes N is a power of 2.

Notational Conventions: Unless specified otherwise, we assume that i, p, and q range over $\{0..N-1\}$ and that $0 \leq level < log_2 N$ holds. We denote statement number k of process i as $k.i$. We use x/y to denote truncated integer division, i.e., $\lfloor x \div y \rfloor$. Let S be a subset of the statement labels in process i. Then, $i@\{S\}$ holds iff the program counter for process i equals some value in S. The following is a list of symbols we will use ordered by increasing binding power: $\equiv$, $\mapsto$, $\Rightarrow$, $\vee$, $\wedge$, $(=, \neq, >, <, \geq, \leq)$, $+$, $(\cdot, /)$, $\neg$, $(., @)$. The symbols enclosed in parentheses have the same priority. We sometimes use parentheses to override this binding rule. □

The function *comp* defined in Figure 2.2 has the following properties.

$$comp(comp(x)) \;=\; x \tag{C0}$$

(C0) directly follows from the definition of *comp*. □

$$p/2^j = comp(q/2^j) \;\Rightarrow\; comp(p/2^j) = q/2^j \tag{C1}$$

The antecedent implies that $comp(p/2^j) = comp(comp(q/2^j))$ holds. By (C0), the consequent follows. □

$$p/2^{j+1} = q/2^{j+1} \;\Rightarrow\; p/2^j = q/2^j \;\vee\; p/2^j = comp(q/2^j) \tag{C2}$$

Let x denotes the value of $p/2^{j+1}$. By the definition of $p/2^{j+1}$, $x \cdot 2^{j+1} \leq p < x \cdot 2^{j+1} + 2^{j+1}$. (Recall that '/' denotes truncated integer division.) The antecedent implies that $x \cdot 2^{j+1} \leq p < x \cdot 2^{j+1} + 2^{j+1} \;\wedge\; x \cdot 2^{j+1} \leq q < x \cdot 2^{j+1} + 2^{j+1}$. Dividing by 2^j yields $2x \leq p/2^j < 2x + 2 \;\wedge\; 2x \leq q/2^j < 2x + 2$. This implies that $(p/2^j = 2x \;\vee\; p/2^j = 2x + 1) \;\wedge\; (q/2^j = 2x \;\vee\; q/2^j = 2x + 1)$ holds. The consequent follows from the definition of $comp(q/2^j)$. □

$$p/2^j = q/2^j \;\vee\; p/2^j = comp(q/2^j) \;\Rightarrow\; p/2^{j+1} = q/2^{j+1} \tag{C3}$$

(C3) directly follows from the definition of *comp*. □

To facilitate the presentation, we define the following predicate.

$$ME(level) \;\equiv\; (\forall i :: (\mathbf{N}p : p/2^{level} = i/2^{level} :: p.j \geq level) \;\leq\; 1)$$

The following invariants, which are stated without proof, follow directly from the program text.

invariant $\quad i@\{0\} \;\Rightarrow\; (i.j = 0)$ (I0)

invariant $\quad i@\{2..13, 17..20\} \;\Rightarrow\; (0 \leq i.j < \log_2 N)$ (I1)

invariant $\quad i@\{14\} \;\Rightarrow\; (i.j = \log_2 N)$ (I2)

invariant $\quad i.k = i/2^{i.j+1} \;\wedge\; i.l = i/2^{i.j}$ (I3)

We next prove several invariants on the relation between the program variables and *comp*.

invariant $\quad C[level, i/2^{level}] = p \;\wedge\; p \neq -1 \;\Rightarrow\; p/2^{level} = i/2^{level}$ (I7)

Initially $(\forall j, i :: C[j, i] = -1)$ holds, and hence (I7) is true. To prove that (I7) is not falsified, it suffices to consider only those statements that may establish the antecedent. The antecedent may be established only by statement $2.p$. Statement $2.p$ establishes the antecedent only if $p.j = level \;\wedge\; p.l = i/2^{level}$ holds. By (I3), this implies that $p.j = level \;\wedge\; p/2^{level} = i/2^{level}$ holds. Thus, statement $2.p$ preserves (I7). □

invariant $\quad C[level, comp(i/2^{level})] = p \ \wedge \ p \neq -1 \ \Rightarrow$

$p/2^{level} = comp(i/2^{level})$ (I8)

The proof is similar to that given for (I7). □

invariant $\quad i.j = level \ \wedge \ i@\{6..9\} \ \Rightarrow \ i.rival = -1 \ \vee$

$i.rival/2^{level} = comp(i/2^{level})$ (I9)

Initially $(\forall i :: i@\{0\})$ holds, and hence (I9) is true. To prove that (I9) is not falsified, it suffices to consider only those statements that may establish the antecedent or falsify the consequent. The antecedent may be established only by statements $5.i$, $13.i$, and $16.i$. Statement $5.i$ establishes the antecedent when $i.j = level$ holds. By (I3), $i.j = level \ \wedge \ i.l = comp(i/2^{level})$ holds. (I8) implies that statement $5.i$ also establishes the consequent in that case. By the same reason, $5.i$ does not falsify the consequent when the antecedent holds. $\neg i@\{6..9\}$ holds after the execution of statements $13.i$ and $16.i$. Although statement $18.i$ may falsify the consequent, it establishes $i@\{19\}$. □

invariant $\quad (i.j > level \vee i.j = level \wedge i@\{4..20\}) \wedge T[level, i/2^{level+1}] = p \ \Rightarrow$

$p/2^{level+1} = i/2^{level+1}$ (I10)

Initially $(\forall i :: i.j = 0 \ \wedge \ i@\{0\})$ holds, and hence (I10) is true. The antecedent may be established only by statements $1.i$, $3.i$, $13.i$, $16.i$, and $3.p$. Statement $1.i$ establishes $i@\{14\}$ only if $i.j = \log_2 N$ holds, which implies that $i.j > level$. Statement $3.i$ establishes the antecedent only if $i.j = level$ holds.

(I3) implies that statement $3.i$ establishes $T[level, i/2^{level+1}] = i$ in that case. Because $i/2^{level+1} = i/2^{level+1}$ holds, this implies that statement $3.i$ preserves (I10). Statement $13.i$ establishes $i.j > level$ only when $i.j = level \wedge i@\{13\}$ holds. Statement $16.i$ establishes $i.j = level$ only when $i.j > level$ holds. It follows that, although statements $1.i$, $3.i$, and $16.i$ may preserve the antecedent, they does not establish it.

Statement $3.p$ may establish the antecedent only if $p.j = level \wedge p.k = i/2^{level+1}$ holds. By (I3), this implies that $p/2^{level+1} = i/2^{level+1}$ holds. Thus, statement $3.p$ preserves (I10). □

invariant $\quad i.j = level \wedge i@\{19, 20\} \Rightarrow i.rival/2^{level+1} = i/2^{level+1}$ (I11)

Initially $(\forall i :: i@\{0\})$ holds, and hence (I11) is true. The antecedent may be established only by statement $18.i$. Statement $18.i$ establishes the antecedent only if $i.j = level \wedge i@\{18\}$ holds. (I3) and (I10) imply that statement $18.i$ establishes the consequent in that case. □

Mutual Exclusion

We next prove that the following assertion is an invariant, which implies, by (I2), that the mutual exclusion property holds.

$$(\forall\, n : 0 \leq n \leq \log_2 N :: ME(n)) \tag{G0}$$

We use an induction on n in our proof. The induction base $ME(0)$ is an invariant by the definition of ME. Thus, it suffices to prove that the following assertion

holds.

$$(\forall\ level : 0 \leq level < \log_2 N :: ME(level)\ \Rightarrow\ ME(level+1)) \tag{G1}$$

Next, we prove several assertions that are needed to establish $ME(level+1)$. In these proofs, we assume that $ME(level)$ is an invariant.

invariant $\quad C[level, p/2^{level}] = p\ \Rightarrow\ p.j > level \vee$
$p.j = level\ \wedge\ p@\{3..14, 17\}$ (I12)

Initially $(\forall\ level, i :: C[level, i] = -1)$ holds, and hence, because p ranges over $\{0..N-1\}$, (I12) is true. The antecedent is established only by statement $2.p$ when $p.j = level$ holds. Statement $2.p$ establishes $p.j = level\ \wedge\ p@\{3\}$. Only statements $13.p$, $14.p$, $16.p$, and $17.p$ may falsify the consequent. When statement $13.p$ falsifies $p.j = level\ \wedge\ p@\{3..14, 17\}$, it establishes $p.j > level$. Statement $14.p$ does not falsify the consequent, because, by (I2), $p@\{14\}\ \Rightarrow\ p.j > level$ holds (recall that $level < \log_2 N$). When statement $16.p$ falsifies $p.j > level$, it establishes $p.j = level\ \wedge\ p@\{17\}$. Statement $17.p$ may falsify the consequent only if $p.j = level$ holds. In this case, by (I3), statement $17.p$ also falsifies the antecedent. □

invariant $\quad p.j > level\ \vee\ p.j = level\ \wedge\ p@\{3..14, 17\}\ \Rightarrow$
$C[level, p/2^{level}] = p$ (I13)

Initially $(\forall i :: i.j = 0\ \wedge\ i@\{0\})$ holds, and hence (I13) is true. Only statements

$1.p$, $2.p$, $13.p$, and $16.p$ may establish the antecedent. The consequent may be falsified by statements $2.i$ or $17.i$. Statement $1.p$ establishes $p@\{14\}$ only if $p.j = \log_2 N$ holds. This implies $p.j > level$. Thus, although statement $1.p$ may preserve the antecedent, it does not establish it. Statement $2.p$ may establish the antecedent only when $p.j = level$ holds. In this case, by (I3), it also establishes the consequent. Statement $13.p$ establishes $p.j > level$ only if $p.j = level \ \wedge \ p@\{13\}$ holds. Thus, although statement $13.p$ may preserve the antecedent, it does not establish it. Statement $16.p$ establishes $p.j = level \ \wedge \ p@\{17\}$ only if $p.j > level$ holds. Thus, it does not establish the antecedent. The consequent may be falsified by statements $2.i$ or $17.i$ only if $i.l = p/2^{level} \ \wedge \ i.j = level$ holds. This implies, by (I3), that $p/2^{level} = i/2^{level} \ \wedge \ i.j = level$ holds. If $i \neq p$ holds, then, by $ME(level)$, $p.j < level$, which implies that the antecedent of (I13) does not hold. If $i = p$, then statement $2.i$ does not falsify the consequent, and when statement $17.i$ falsifies the consequent, it also falsifies the antecedent. □

invariant $C[level, p/2^{level}] = -1 \ \Rightarrow \ (\forall i : i/2^{level} = p/2^{level} :: i.j < level \vee$
$i.j = level \ \wedge \ i@\{0, 1, 2, 15, 16, 18, 19, 20\})$ (I14)

Assume, for the sake of contradiction, that $(C[level, p/2^{level}] = -1) \ \wedge \ i/2^{level} = p/2^{level} \ \wedge \ (i.j > level \ \vee \ i.j = level \ \wedge \ i@\{3..14, 17\})$ holds. Then, by (I13), that $(C[level, p/2^{level}] = -1) \wedge (C[level, i/2^{level}] = i) \wedge i/2^{level} = p/2^{level}$ holds. This is a contradiction. Thus, (I14) is an invariant. □

invariant $(\forall i : i/2^{level} = p/2^{level} :: i.j < level \ \vee \ i.j = level \wedge$
$i@\{0, 1, 2, 15, 16, 18, 19, 20\}) \ \Rightarrow \ C[level, p/2^{level}] = -1$ (I15)

Assume, for the sake of contradiction, that $(\forall i : i/2^{level} = p/2^{level} :: i.j < level \vee i.j = level \wedge i@\{0,1,2,15,16,18,19,20\}) \wedge C[level, p/2^{level}] = r \wedge r \neq -1$ holds. Then, by (I7), $r/2^{level} = p/2^{level}$ holds, and $r.j > level \vee (r.j = level \wedge r@\{3..14,17\})$ holds by (I12). This is a contradiction. Thus, (I15) is an invariant. □

invariant $(p.j > level \vee p.j = level \wedge p@\{4..20\}) \wedge$

$q/2^{level} = comp(p/2^{level}) \wedge q.j > level \vee q.j = level \wedge q@\{4..20\}) \Rightarrow$

$T[level, p/2^{level+1}] = p \vee T[level, p/2^{level+1}] = q$ (I16)

Initially $(\forall i :: i.j = 0 \wedge i@\{0\})$ holds, and hence (I16) is true. The antecedent is established only by statements $1.p$, $1.q$, $3.p$, $3.q$, $13.p$, $13.q$, $16.p$, and $16.q$. As in the proof on (I13), although statement $1.p$ may preserve the antecedent, it does not establish it. Similar reasoning applies to $1.q$. Statement $3.p$ establishes the antecedent only when $p.j = level$ holds. By (I3), this implies that the consequent is established. Statement $3.q$ establishes the antecedent only when $q.j = level$ holds. (I3), (C3), and $q/2^{level} = comp(p/2^{level})$ implies $q.k = p/2^{level+1}$. In this case, statement $3.q$ establishes the consequent. As in the proof of (I13), although statements $13.p$ and $16.p$ may preserve the antecedent, they do not establish it. Similar reasoning applies to $13.q$ and $16.q$. (I3) implies that the consequent may be falsified only by statement $3.i$, where $i/2^{level+1} = p/2^{level+1} \wedge i.j = level \wedge i \neq p \wedge i \neq q$ holds. Then, $i/2^{level} = p/2^{level} \vee i/2^{level} = comp(p/2^{level})$ holds by (C2). When $q/2^{level} = comp(p/2^{level})$ holds, $i.j = level \wedge i \neq p \wedge i \neq q \wedge (i/2^{level} = p/2^{level} \vee i/2^{level} = q/2^{level})$ holds. This implies, by $ME(level)$, that $p.j < level \vee q.j < level$ holds, and hence, the antecedent does not hold.

□

invariant $p.j = level \wedge p@\{6..9\} \wedge C[level, comp(p/2^{level})] = q \wedge q \neq -1 \Rightarrow$

$(p.rival = q \ \wedge \ q \neq -1) \ \vee \ T[level, p/2^{level+1}] = q \vee$

$(q.j = level \ \wedge \ q@\{3\})$ (I17)

Initially $(\forall i :: i@\{0\})$ holds, and hence (I17) is true. Only statements $5.p$, $13.p$, $16.p$, and $2.q$ may establish the antecedent. The consequent may be falsified only by statements $5.p$, $18.p$, $3.q$, and $3.i$, where $i \neq q$. Statement $5.p$ establishes the antecedent only if $p.j = level \ \wedge \ C[level, comp(p/2^{level})] = q \ \wedge \ q \neq -1$ holds. By (I3), this implies that statement $5.p$ also establishes $p.rival = q \ \wedge \ q \neq -1$. Statements $13.p$ and $16.p$ establish $\neg p@\{6..9\}$. Thus, they do not establish the antecedent. Statement $2.q$ may establish the antecedent only when $q.j = level$ holds. In this case, it also establishes $q.j = level \ \wedge \ q@\{3\}$. By (I3), statement $5.p$ may falsify the consequent only if $C[level, comp(p/2^{level})] \neq q$ holds. Statement $18.p$ establishes $p@\{19\}$. Thus, statements $5.p$ and $18.p$ preserve (I17). Statement $3.q$ may falsify (I17) only if $C[level, comp(p/2^{level})] = q \ \wedge \ q \neq -1 \ \wedge \ q.j = level \ \wedge \ q@\{3\}$ holds. By (I8), this implies that $q/2^{level} = comp(p/2^{level}) \wedge q.j = level \wedge q@\{3\}$ holds. By (C3), this implies that $q/2^{level+1} = p/2^{level+1} \wedge q.j = level \wedge q@\{3\}$ holds. In this case, (I3) implies that statement $3.q$ establishes $T[level, p/2^{level+1}] = q$. (I3) implies that statement $3.i$ may falsify $T[level, p/2^{level+1}] = q$ only if $i/2^{level+1} = p/2^{level+1} \ \wedge \ i.j = level \ \wedge \ i@\{3\} \ \wedge \ i \neq q$ holds. By (C2), this implies that $(i/2^{level} = p/2^{level} \ \vee \ i/2^{level} = comp(p/2^{level})) \ \wedge \ i.j = level \ \wedge \ i@\{3\} \ \wedge \ i \neq q$ holds. Note further that statement $3.i$ may falsify (I17) only if $C[level, comp(p/2^{level})] = q \ \wedge \ q \neq -1$

holds. By (I8), this implies that $q/2^{level} = comp(p/2^{level})$ holds. Thus, statement $3.i$ may falsify (I17) only if $(i/2^{level} = p/2^{level} \ \vee \ i/2^{level} = q/2^{level}) \ \wedge \ i.j = level \ \wedge \ i@\{3\} \ \wedge \ i \neq q$ holds. If $i/2^{level} = p/2^{level} \ \wedge \ i.j = level \ \wedge \ i@\{3\}$ holds, then, by *ME*$(level)$, this implies that $p.j = level$ does not hold. Otherwise, $i/2^{level} = q/2^{level} \ \wedge \ i.j = level \ \wedge \ i \neq q$ holds. By *ME*$(level)$, this implies that $q.j < level$ holds. In that case, (I12) implies that $C[level, comp(p/2^{level})] = q$ does not hold. □

invariant $\quad p.j = level \ \wedge \ (p@\{6\} \ \wedge \ p.rival \neq -1 \ \vee \ p@\{7..12\}) \ \wedge$

$q/2^{level} = comp(p/2^{level}) \ \wedge \ q.j = level \ \wedge \ q@\{19, 20\} \ \Rightarrow$

$q.rival = p \ \vee \ q.rival = q$ (I18)

Initially $(\forall i :: i@\{0\})$ holds, and hence (I18) is true. The antecedent may be established only by statements $5.p$, $6.p$, $13.p$, $16.p$, $18.p$, $13.q$, $16.q$, and $18.q$. Only statements $5.q$ and $18.q$ may falsify the consequent. By (I3), statement $5.p$ may establish the antecedent only if $p.j = level \ \wedge \ C[level, comp(p/2^{level})] = i \ \wedge \ i \neq -1 \ \wedge \ q/2^{level} = comp(p/2^{level})$ holds for some i. By (I8), this implies that $C[level, comp(p/2^{level})] = i \ \wedge \ i/2^{level} = comp(p/2^{level}) \ \wedge \ i/2^{level} = q/2^{level}$ holds. By (I12), this implies that $(i.j > level \ \vee \ i.j = level \ \wedge \ i@\{3..14, 17\}) \ \wedge \ i/2^{level} = q/2^{level}$ holds. By *ME*$(level)$, this implies that $q.j = level$ does not hold. Statement $6.p$ may establish $p.j = level \ \wedge \ p@\{7..12\}$ only if $p.j = level \ \wedge \ p@\{6\} \ \wedge \ p.rival \neq -1$ holds. Thus, although statement $6.p$ may preserve, it does not establish the antecedent. Statements $13.p$, $16.p$, and $18.p$ establish $\neg p@\{6..12\}$, and hence, do not establish the antecedent. Statements $13.q$ and $16.q$ establish $\neg q@\{19, 20\}$, and hence, do not establish the antecedent.

Statement 18.q establishes the antecedent only if $p.j = level \ \wedge \ (p@\{6\} \ \wedge \ p.rival \neq -1 \ \vee \ p@\{7..12\}) \ \wedge \ q.j = level \ \wedge \ q@\{18\}$ holds. By (I16), this implies that $T[level, p/2^{level+1}] = p \ \vee \ T[level, p/2^{level+1}] = q$ holds. In that case, by (I3), statement 18.q establishes the consequent. By the same reason, statement 18.q may falsify the consequent only if the antecedent does not hold. Finally, statement 5.q establishes $q@\{6\}$. □

invariant $p.j = level \ \wedge \ (p@\{6\} \ \wedge \ p.rival \neq -1 \ \vee \ p@\{7..12\}) \ \wedge$
$q/2^{level} = comp(p/2^{level}) \ \wedge \ q.j = level \ \wedge \ q@\{20\} \ \Rightarrow \ q.rival = p$ (I19)

Initially $(\forall i :: i@\{0\})$ holds, and hence (I19) is true. The antecedent may be established only by statements 5.p, 6.p, 13.p, 16.p, 18.p, 13.q, 16.q, and 19.q. Only statements 5.q and 18.q may falsify the consequent. The reasoning for statements 5.p, 6.p, 13.p, 16.p, 18.p, 13.q, and 16.q is the same as that given in the proof of (I18). Statement 19.q establishes the antecedent only if $p.j = level \wedge (p@\{6\} \wedge p.rival \neq -1 \vee p@\{7..12\}) \wedge q/2^{level} = comp(p/2^{level}) \wedge q.j = level \ \wedge \ q@\{19\} \ \wedge \ q.rival \neq q$ holds. This implies, by (I18), that $q.rival = p$ holds. Finally, statements 5.q and 18.q establish $\neg q@\{20\}$. □

invariant $p.j = level \ \wedge \ (p@\{6\} \ \wedge \ p.rival \neq -1 \ \vee \ p@\{7..12\}) \ \wedge$
$q/2^{level} = comp(p/2^{level}) \ \wedge \ q.j = level \ \wedge \ q@\{19\} \ \wedge \ q.rival = q \ \Rightarrow$
$T[level, p/2^{level+1}] = q$ (I20)

Initially $(\forall i :: i@\{0\})$ holds, and hence (I20) is true. The antecedent may be established only by statements 5.p, 6.p, 13.p, 16.p, 18.p, 5.q, 13.q, 16.q, and 18.q.

Only statement $3.i$ may falsify the consequent. The reasoning for statements $5.p$, $6.p$, $13.p$, $16.p$, and $18.p$ is the same as that given in the proof of (I18). Statements $5.q$, $13.q$, and $16.q$ establish $\neg q@\{19\}$, and hence, do not establish the antecedent. If $q/2^{level} = comp(p/2^{level})$ holds, then, by (C3), $q/2^{level+1} = p/2^{level+1}$ also holds. Thus, (I3) implies that statement $18.q$ establishes the antecedent only if the consequent holds.

(I3) implies that statement $3.i$ may falsify $T[level, p/2^{level+1}] = q$ only if $i.j = level \ \wedge \ i/2^{level+1} = p/2^{level+1} \ \wedge \ i@\{3\} \ \wedge \ i \neq q$ holds. By (C2), $i.j = level \ \wedge \ (i/2^{level} = p/2^{level} \ \vee \ i/2^{level} = comp(p/2^{level})) \ \wedge \ i@\{3\} \ \wedge \ i \neq q$ holds. Note further that statement $3.i$ may falsify (I20) only if $q/2^{level} = comp(p/2^{level})$ holds. Thus, statement $3.i$ may falsify (I20) only if $(i/2^{level} = p/2^{level} \ \vee \ i/2^{level} = q/2^{level}) \ \wedge \ i.j = level \ \wedge \ i@\{3\} \ \wedge \ i \neq q$ holds. If $i/2^{level} = p/2^{level} \ \wedge \ i.j = level \ \wedge \ i@\{3\}$ holds, then, by $ME(level)$, this implies that $p.j = level \wedge p@\{6..12\}$ does not hold. Otherwise, $i/2^{level} = q/2^{level} \wedge i.j = level \wedge i \neq q$ holds, which by $ME(level)$, implies that $q.j = level$ does not hold.

□

invariant $\quad (p.j > level \ \vee \ p.j = level \ \wedge \ p@\{5..14\}) \ \wedge$

$P[level, p] = 2 \ \wedge \ C[level, comp(p/2^{level})] = q \ \wedge \ q \neq -1 \ \Rightarrow$

$T[level, p/2^{level+1}] = q \ \vee \ q.j = level \ \wedge \ q@\{3\}$ (I21)

Initially $(\forall i :: i.j = 0 \ \wedge \ i@\{0\})$ holds, and hence (I21) is true. The antecedent is established only by statements $1.p$, $4.p$, $13.p$, $16.p$, $20.i$, and $2.q$. The consequent may be falsified only by statements $3.q$, $13.q$, $16.q$, and $3.i$, where $i \neq q$. Statement $1.p$ establishes $p@\{14\}$ only if $p.j = \log_2 N$ holds. This implies

$p.j > level$. Thus, although statement $1.p$ may preserve the antecedent, it does not establish it. Statement $4.p$ may establish the antecedent only if $p.j = level$ holds. In this case, it also establishes $P[level, p] = 0$. Statement $13.p$ establishes $p.j > level$ only when $p.j = level \ \wedge \ p@\{13\}$ holds, and statement $16.p$ establishes $p.j = level$ only when $p.j > level$ holds. Thus, although statements $13.p$ and $16.p$ may preserve the antecedent, they do not establish it.

(I11) implies that statement $20.i$ may establish $P[level, p] = 2$ only if $i.j = level \ \wedge \ i@\{20\} \ \wedge \ i/2^{level+1} = p/2^{level+1}$ holds. By (C2), this implies that $i.j = level \wedge i@\{20\} \wedge (i/2^{level} = p/2^{level} \vee i/2^{level} = comp(p/2^{level})$ holds. Note further that statement $20.i$ may falsify (I21) only if $C[level, comp(p/2^{level})] = q \ \wedge \ q \neq -1$ holds. By (I8), this implies that $q/2^{level} = comp(p/2^{level})$ holds. Thus, statement $20.i$ may falsify (I21) only if $(i/2^{level} = p/2^{level} \ \vee \ i/2^{level} = q/2^{level}) \wedge i.j = level \wedge i@\{20\}$ holds. If $i/2^{level} = p/2^{level} \wedge i.j = level \wedge i@\{20\}$ holds, then, by $ME(level)$, this implies that $p.j < level \vee p.j = level \wedge p@\{20\}$ holds. Otherwise, $i/2^{level} = q/2^{level} \ \wedge \ i.j = level \ \wedge \ i@\{20\}$ holds, which, by $ME(level)$ and (I12), implies that $C[level, q/2^{level}] = q$ does not hold. Because $q/2^{level} = comp(p/2^{level})$, this implies that $C[level, comp(p/2^{level})] = q$ does not hold. Statement $2.q$ may establish the antecedent only if $q.j = level$ holds. In this case, it also establishes $q.j = level \ \wedge \ q@\{3\}$.

Statement $3.q$ may falsify (I21) only if $C[level, comp(p/2^{level})] = q \ \wedge \ q \neq -1 \ \wedge \ q.j = level \ \wedge \ q@\{3\}$ holds. By (I8), this implies that $q/2^{level} = comp(p/2^{level}) \ \wedge \ q.j = level \ \wedge \ q@\{3\}$ holds. By (C3), this implies that $q/2^{level+1} = p/2^{level+1} \ \wedge \ q.j = level \ \wedge \ q@\{3\}$ holds. In that case, (I3) implies that statement $3.q$ establishes $T[level, p/2^{level+1}] = q$. Statements $13.q$ and $16.q$ do not falsify $q.j = level \ \wedge \ q@\{3\}$. Finally, (I3) implies that statement $3.i$

may falsify $T[level, p/2^{level+1}] = q$ only if $i/2^{level+1} = p/2^{level+1} \ \wedge \ i.j = level \ \wedge \ i@\{3\} \ \wedge \ i \neq q$ holds. By (C2), this implies that $(i/2^{level} = p/2^{level} \ \vee \ i/2^{level} = comp(p/2^{level})) \wedge i.j = level \wedge i@\{3\} \wedge i \neq q$ holds. Note further that statement $3.i$ may falsify (I21) only if $C[level, comp(p/2^{level})] = q \wedge q \neq -1$ holds. By (I8), this implies that $q/2^{level} = comp(p/2^{level})$ holds. Thus, statement $3.i$ may falsify (I21) only if $(i/2^{level} = p/2^{level} \vee i/2^{level} = q/2^{level}) \wedge i.j = level \wedge i@\{3\} \wedge i \neq q$ holds. If $i/2^{level} = p/2^{level} \ \wedge \ i.j = level \ \wedge \ i@\{3\}$ holds, then this implies, by $ME(level)$, that $p.j < level \ \vee \ p.j = level \ \wedge \ p@\{3\}$ holds. Otherwise, $i/2^{level} = q/2^{level} \wedge i.j = level \wedge i \neq q$ holds, which, by $ME(level)$ and (I12), implies that $C[level, q/2^{level}] = q$ does not hold. Because $q/2^{level} = comp(p/2^{level})$, this implies that $C[level, comp(p/2^{level})] = q$ does not hold. Thus, statement $3.i$ preserves (I21). □

invariant $(p.j > level \ \vee \ p.j = level \ \wedge \ p@\{13..20\}) \wedge$

$C[level, comp(p/2^{level})] = q \ \wedge \ q \neq -1 \Rightarrow$

$T[level, p/2^{level+1}] = q \ \vee \ q.j = level \ \wedge \ q@\{3\}$ (I22)

Initially $(\forall i :: i.j = 0 \ \wedge \ i@\{0\})$ holds, and hence (I22) is true. The antecedent may be established only by statements $1.p$, $6.p$, $7.p$, $11.p$, $12.p$, $13.p$, $16.p$, and $2.q$. Only statements $3.q$, $13.q$, $16.q$, and $3.i$, where $i \neq q$, may falsify the consequent. The reasoning for statements $1.p$, $13.p$, $16.p$, $3.q$, $13.q$, and $16.q$ is the same as that given in the proof of (I21).

Statement $6.p$ may establish the antecedent only if $p.j = level \ \wedge \ p@\{6\} \ \wedge \ p.rival = -1$ holds. By (I17), this implies that $\neg(C[level, comp(p/2^{level})] = q \ \wedge \ q \neq -1) \ \vee \ T[level, p/2^{level+1}] = q \ \vee \ (q.j = level \ \wedge \ q@\{3\})$ holds. This

implies that either the antecedent does not hold or the consequent holds after the execution of statement $6.p$.

(I3) implies that statements $7.p$ and $11.p$ establish the antecedent only if $p.j = level \wedge T[level, p/2^{level+1}] \neq p \wedge p@\{7,11\} \wedge C[level, comp(p/2^{level})] = q \wedge q \neq -1$ holds. By (I8), this implies that $p.j = level \wedge T[level, p/2^{level+1}] \neq p \wedge p@\{7,11\} \wedge q/2^{level} = comp(p/2^{level}) \wedge C[level, q/2^{level}] = q$ holds. In that case, (I12) implies that $p.j = level \wedge T[level, p/2^{level+1}] \neq p \wedge p@\{7,11\} \wedge q/2^{level} = comp(p/2^{level}) \wedge (q.j > level \vee q.j = level \wedge q@\{3..14,17\})$ holds. By predicate calculus, this implies that $T[level, p/2^{level+1}] \neq p \wedge p.j = level \wedge p@\{7,11\} \wedge q/2^{level} = comp(p/2^{level}) \wedge ((q.j > level \vee q.j = level \wedge q@\{4..17\}) \vee (q.j = level \wedge q@\{3\}))$ holds. This implies, by (I16), that the consequent holds.

Statement $12.p$ establishes the antecedent only if $C[level, comp(p/2^{level})] = q \wedge q \neq -1 \wedge p.j = level \wedge p@\{12\} \wedge P[level, p] = 2$ holds. By (I21), this implies that the consequent holds. Statement $2.q$ may establish the antecedent only if $q.j = level$ holds. In this case, it also establishes $q.j = level \wedge q@\{3\}$.

(I3) implies that statement $3.i$ may falsify $T[level, p/2^{level+1}] = q$ only if $i/2^{level+1} = p/2^{level+1} \wedge i.j = level \wedge i@\{3\} \wedge i \neq q$ holds. By (C2), this implies that $(i/2^{level} = p/2^{level} \vee i/2^{level} = comp(p/2^{level})) \wedge i.j = level \wedge i@\{3\} \wedge i \neq q$ holds. Note further that statement $3.i$ may falsify (I22) only if $C[level, comp(p/2^{level})] = q \wedge q \neq -1$ holds. By (I8), this implies that $q/2^{level} = comp(p/2^{level})$ holds. Thus, statement $3.i$ may falsify (I22) only if $(i/2^{level} = p/2^{level} \vee i/2^{level} = q/2^{level}) \wedge i.j = level \wedge i@\{3\} \wedge i \neq q$ holds. If $i/2^{level} = p/2^{level} \wedge i.j = level \wedge i@\{3\}$ holds, then, this implies, by $ME(level)$, that $p.j < level \vee p.j = level \wedge p@\{3\}$ holds. Otherwise,

$i/2^{level} = q/2^{level} \wedge i.j = level \wedge i \neq q$ holds, which, by $ME(level)$ and (I12), implies that $C[level, q/2^{level}] = q$ does not hold. Because $q/2^{level} = comp(p/2^{level})$, this implies that $C[level, comp(p/2^{level})] = q$ does not hold. Thus, statement $3.i$ preserves (I22). □

invariant $(\forall i :: (\mathbf{N}p : p/2^{level+1} = i/2^{level+1} :: p.j \geq level + 1) \leq 1)$ (I23)

Assume, for the sake of contradiction, that $p.j > level \wedge q.j > level \wedge p/2^{level+1} = i/2^{level+1} \wedge q/2^{level+1} = i/2^{level+1} \wedge p \neq -1 \wedge q \neq -1$ holds for some i. By $ME(level)$ and (C2), we can assume, that $p.j > level \wedge q.j > level \wedge p/2^{level} = i/2^{level} \wedge q/2^{level} = comp(i/2^{level}) \wedge p \neq -1 \wedge q \neq -1$ holds. This implies, by (I13), that $p.j > level \wedge q.j > level \wedge C[level, p/2^{level}] = p \wedge C[level, q/2^{level}] = q \wedge p/2^{level} = i/2^{level} \wedge q/2^{level} = comp(i/2^{level}) \wedge p \neq -1 \wedge q \neq -1$ holds. Then, by (C1), $p.j > level \wedge q.j > level \wedge C[level, comp(p/2^{level})] = q \wedge C[level, comp(q/2^{level})] = p \wedge p \neq -1 \wedge q \neq -1$ holds. By (I22), this implies that $T[level, p/2^{level+1}] = q \wedge T[level, q/2^{level+1}] = p$ holds, which is a contradiction. Thus, (I23) is an invariant. □

Observe that we have completed the proof of (G1), and thus proved that (G0) holds. Because $ME(level)$ has been proved to hold for any $level$, the invariants from (I12) to (I23) hold for any $level$.

Progress

We next prove a number of invariants that are needed to establish starvation-freedom.

invariant $p.j = level \ \wedge \ (p@\{6\} \ \wedge \ p.rival \neq -1 \ \vee \ p@\{7..12\}) \ \wedge$

$(\forall i : i/2^{level} = comp(p/2^{level}) :: i.j < level \vee i.j = level \wedge i@\{0, 1, 2, 15, 16\})$

$\Rightarrow \ P[level, p] = 2$ (I24)

Initially $(\forall i :: i@\{0\})$ holds, and hence (I24) is true. The antecedent may be established only by statements $5.p$, $6.p$, $13.p$, $16.p$, $18.p$, $13.i$, $14.i$, $15.i$, $16.i$, $19.i$, and $20.i$. Only statements $4.p$ and $9.i$ may falsify the consequent.

As in the proof of (I18), statement $5.p$ may establish the antecedent only if $i/2^{level} = comp(p/2^{level}) \wedge \ (i.j > level \ \vee \ i.j = level \ \wedge \ i@\{3..14, 17\})$ holds, which implies that the antecedent of (I24) does not hold. The reasoning for statements $6.p$, $13.p$, $16.p$, and $18.p$ is the same as that given in the proof of (I18).

(I2) implies that when statement $14.i$ establishes $i@\{15\}$, $i.j > level$ holds. Thus, statement $14.i$ does not establish the antecedent.

Statement $13.i$ establishes $i.j = level \ \wedge \ i@\{1\}$ only if $i.j < level$ holds. Statement $15.i$ may establish $i.j = level \ \wedge \ i@\{0\}$ only if $i.j = level \ \wedge \ i@\{15\}$ holds. Statement $16.i$ establishes $i.j < level$ only if $i.j = level \ \wedge \ i@\{16\}$ holds. Thus, although statements $13.i$, $15.i$, and $16.i$ may preserve the antecedent, they do not establish it.

Statement $19.i$ may establish the antecedent only if $p.j = level \ \wedge \ (p@\{6\} \ \wedge \ p.rival \neq -1 \ \vee \ p@\{7..12\}) \ \wedge \ i/2^{level} = comp(p/2^{level}) \wedge i.j = level \wedge i@\{19\} \ \wedge \ i.rival = i$ holds. This implies, by (I20), that $p.j = level \ \wedge \ (p@\{6\} \ \wedge \ p.rival \neq -1 \ \vee \ p@\{7..12\}) \ \wedge \ i/2^{level} = comp(p/2^{level}) \ \wedge \ i.j = level \ \wedge \ i@\{19\} \ \wedge \ T[level, p/2^{level+1}] = i$ holds. By (I13), this implies that $C[level, p/2^{level}] = p \ \wedge \ T[level, p/2^{level+1}] \neq p \ \wedge \ i/2^{level} = comp(p/2^{level}) \ \wedge \ i.j = level \ \wedge \ i@\{19\}$

holds. This implies, by (C1) and (C3), that $C[level, comp(i/2^{level})] = p \ \wedge$ $T[level, i/2^{level+1}] \neq p \ \wedge \ i.j = level \ \wedge \ i@\{19\}$ holds. By (I22), this implies that $p.j = level \ \wedge \ p@\{3\}$ holds, which implies that the antecedent does not hold. Thus, statement 19.i does not establish the antecedent.

Statement 20.i establishes the antecedent only if $p.j = level \ \wedge \ (p@\{6\} \ \wedge$ $p.rival \neq -1 \ \vee \ p@\{7..12\}) \ \wedge \ i/2^{level} = comp(p/2^{level}) \ \wedge \ i.j = level \ \wedge \ i@\{20\}$ holds. By (I19), this implies that $i.j = level \ \wedge \ i@\{20\} \ \wedge \ i.rival = p$ holds. In this case, statement 20.i establishes the consequent.

Statement 4.p establishes $p@\{5\}$. (I9) implies that statement 9.i may falsify the consequent only if $i.j = level \wedge i@\{9\} \ \wedge \ p/2^{level} = comp(i/2^{level})$ holds. By (C1), this implies that $i.j = level \ \wedge \ i@\{9\} \ \wedge \ i/2^{level} = comp(p/2^{level})$ holds, which implies that the antecedent does not hold. □

invariant $\quad p.j = level \ \wedge \ p@\{8..10\} \ \wedge \ q/2^{level} = comp(p/2^{level}) \ \wedge$

$q.j = level \ \wedge \ q@\{6..10\} \ \wedge \ T[level, q/2^{level+1}] = q \ \Rightarrow \ q.rival = p \qquad$ (I25)

Initially $(\forall i :: i@\{0\})$ holds, and hence (I25) is true. The antecedent may be established only by statements 7.p, 13.p, 16.p, 3.q, 5.q, 13.q, and 16.q. Only statements 5.q and 18.q may falsify the consequent. (I3) implies that statement 7.p may establish the antecedent only if $q/2^{level} = comp(p/2^{level}) \wedge$ $T[level, p/2^{level+1}] = p$ holds. By (C3), this implies that $T[level, q/2^{level+1}] = p$ holds, which implies that the antecedent does not hold. $\neg p@\{8..10\}$ holds after the execution of statements 13.p and 16.p. $\neg q@\{6..10\}$ holds after the execution of statements 3.q, 13.q, and 16.q. Statement 5.q may falsify (I25) only if $p.j = level \wedge p@\{8..10\} \wedge q/2^{level} = comp(p/2^{level}) \wedge q.j = level \wedge q@\{5\}$ holds.

By (I13), this implies that $C[level, p/2^{level}] = p \ \wedge \ q/2^{level} = comp(p/2^{level}) \ \wedge \ q.j = level \wedge q@\{5\}$ holds. This implies, by (C1), that $C[level, comp(q/2^{level})] = p \ \wedge \ q.j = level \ \wedge \ q@\{5\}$ holds. In this case, (I3) implies that statement $5.q$ establishes the consequent. Finally, statement $18.q$ establishes $q@\{19\}$. □

invariant $p.j = level \ \wedge \ p@\{8..10\} \ \wedge \ q/2^{level} = comp(p/2^{level}) \wedge$

$q.j = level \ \wedge \ q@\{10..12\} \ \wedge \ T[level, q/2^{level+1}] = q \ \Rightarrow \ P[level, p] \geq 1$ (I26)

Initially $(\forall i :: i@\{0\})$ holds, and hence (I26) is true. The antecedent may be established only by statements $7.p$, $13.p$, $16.p$, $3.q$, $8.q$, $9.q$, $13.q$, and $16.q$. Only statement $4.p$ may falsify the consequent. The reasoning for statements $7.p$, $13.p$, $16.p$, $3.q$, $13.q$, and $16.q$ is similar to that given in the proof of (I25) Statement $8.q$ may establish the antecedent only if $p.j = level \ \wedge \ p@\{8..10\} \ \wedge \ q/2^{level} = comp(p/2^{level}) \ \wedge \ q.j = level \ \wedge \ q@\{8\} \ \wedge \ T[level, q/2^{level+1}] = q \wedge P[level, q.rival] \geq 1$ holds. By (I25), this implies that the consequent holds. Statement $9.q$ may establish the antecedent only if $p.j = level \ \wedge \ p@\{8..10\} \ \wedge \ q/2^{level} = comp(p/2^{level}) \wedge q.j = level \wedge q@\{9\} \wedge T[level, q/2^{level+1}] = q$ holds. By (I25), this implies that statement $9.q$ establishes the consequent. Finally, statement $4.p$ establishes $p@\{5\}$. □

invariant $p.j = level \ \wedge \ p@\{10\} \ \wedge \ q/2^{level} = comp(p/2^{level}) \wedge$

$q.j = level \ \wedge \ q@\{10\} \ \Rightarrow \ P[level, p] \geq 1 \ \vee \ P[level, q] \geq 1$ (I27)

The antecedent implies, by (I16), that $p.j = level \ \wedge \ p@\{10\} \ \wedge \ q/2^{level} = comp(p/2^{level}) \ \wedge \ q.j = level \ \wedge \ q@\{10\} \ \wedge \ (T[level, p/2^{level+1}] = p \ \vee$

$T[level, p/2^{level+1}] = q$) holds. By (C1) and (C3), this implies that $p.j = level \wedge p@\{10\} \wedge q/2^{level} = comp(p/2^{level}) \wedge p/2^{level} = comp(q/2^{level}) \wedge q.j = level \wedge q@\{10\} \wedge (T[level, p/2^{level+1}] = p \vee T[level, q/2^{level+1}] = q)$ holds. By (I26), this implies that the consequent holds. □

invariant $i.j = level \wedge i@\{11, 12\} \Rightarrow P[level, i] \geq 1$ (I28)

Initially $(\forall i :: i@\{0\})$ holds, and hence (I28) is true. The antecedent may be established only by statements $10.i$, $13.i$, and $16.i$. Only statement $4.i$ may falsify the consequent. Statement $10.i$ establishes the antecedent only if the consequent holds. $\neg i@\{11, 12\}$ holds after the execution of statements $4.i$, $13.i$, and $16.i$. □

invariant $p.j = level \wedge p@\{12\} \wedge q/2^{level} = comp(p/2^{level}) \wedge$
$q.j = level \wedge q@\{5..12\} \wedge T[level, q/2^{level+1}] = q \Rightarrow P[level, q] = 0$ (I29)

Initially $(\forall i :: i@\{0\})$ holds, and hence (I29) is true. The antecedent may be established only by statements $11.p$, $13.p$, $16.p$, $3.q$, $4.q$, $13.q$, and $16.q$. Only statements $9.i$ and $20.i$ may falsify the consequent. (I3) implies that statement $11.p$ may establish the antecedent only if $p.j = level \wedge p@\{11\} \wedge q/2^{level} = comp(p/2^{level}) \wedge T[level, p/2^{level+1}] = p$ holds. By (C3), this implies that $T[level, q/2^{level+1}] = p$ holds, which implies that the antecedent does not hold. $\neg p@\{12\}$ holds after the execution of statements $13.p$ and $16.p$. $\neg q@\{5..12\}$ holds after the execution of statements $3.q$, $13.q$, and $16.q$. When statement $4.q$ establishes $q.j = level \wedge q@\{5\}$, it also establishes the consequent.

(I9) implies that statement $9.i$ may falsify (I29) only if $q/2^{level} =$

$comp(p/2^{level}) \wedge q/2^{level} = comp(i/2^{level}) \wedge i.j = level \wedge i@\{9\}$ holds. By (C1), this implies that $p/2^{level} = i/2^{level} \wedge i.j = level \wedge i@\{9\}$ holds. In that case, $ME(level)$ implies that $p.j = level \wedge p@\{12\}$ does not hold. (I11) implies that statement $20.i$ may falsify the consequent only if $q/2^{level} = comp(p/2^{level}) \wedge q/2^{level+1} = i/2^{level+1} \wedge i.j = level \wedge i@\{20\}$ holds. By (C1), this implies that $comp(q/2^{level}) = p/2^{level} \wedge q/2^{level+1} = i/2^{level+1} \wedge i.j = level \wedge i@\{20\}$ holds. By (C2), this implies that $(i/2^{level} = p/2^{level} \vee i/2^{level} = q/2^{level}) \wedge i.j = level \wedge i@\{20\}$ holds. If $i = p$, then statement $20.i$ establishes $\neg p@\{12\}$. If $i/2^{level} = p/2^{level} \wedge i \neq p \wedge i.j = level \wedge i@\{20\}$ holds, then $ME(level)$ implies that $p.j = level$ does not hold. If $i = q$, then statement $20.i$ establishes $\neg q@\{5..12\}$. If $i/2^{level} = q/2^{level} \wedge i \neq q \wedge i.j = level \wedge i@\{20\}$ holds, then $ME(level)$ implies that $q.j = level$ does not hold. □

invariant $\neg(p.j = level \wedge p@\{12\} \wedge q/2^{level} = comp(p/2^{level}) \wedge$
$q.j = level \wedge q@\{12\})$ (I30)

Assume, for the sake of contradiction, that $p.j = level \wedge p@\{12\} \wedge q/2^{level} = comp(p/2^{level}) \wedge q.j = level \wedge q@\{12\}$ holds. By (I28), this implies that $p.j = level \wedge p@\{12\} \wedge q/2^{level} = comp(p/2^{level}) \wedge q.j = level \wedge q@\{12\} \wedge P[level, p] \geq 1 \wedge P[level, q] \geq 1$ holds. By (I16), this implies that $p.j = level \wedge p@\{12\} \wedge q/2^{level} = comp(p/2^{level}) \wedge q.j = level \wedge q@\{12\} \wedge P[level, p] \geq 1 \wedge P[level, q] \geq 1 \wedge (T[level, p/2^{level+1}] = p \vee T[level, p/2^{level+1}] = q)$ holds. In that case, (C1) and (C3) imply that $P[level, p] \geq 1 \wedge P[level, q] \geq 1 \wedge p.j = level \wedge p@\{12\} \wedge q.j = level \wedge q@\{12\} \wedge q/2^{level} = comp(p/2^{level}) \wedge p/2^{level} = comp(q/2^{level}) \wedge (T[level, q/2^{level+1}] = q \vee T[level, p/2^{level+1}] = p)$ holds. By

(I29), this implies that $P[level, p] \geq 1 \ \wedge \ P[level, q] \geq 1 \ \wedge \ (P[level, p] = 0 \ \vee \ P[level, q] = 0)$ holds, which is a contradiction. Thus (I30) is an invariant.

□

invariant $p.j = level \ \wedge \ p@\{4..12\} \ \wedge \ q/2^{level} = comp(p/2^{level}) \ \wedge$

$(q.j > level \ \vee \ q.j = level \ \wedge \ q@\{13..20\}) \ \Rightarrow \ T[level, q/2^{level+1}] = p$ (I31)

Initially $(\forall i :: i@\{0\})$ holds, and hence (I31) is true. The antecedent may be established only by statements $3.p$, $13.p$, $16.p$, $1.q$, $6.q$, $7.q$, $11.q$, $12, q$, $13.q$, and $16.q$. Only statement $3.i$ may falsify the consequent. Statement $3.p$ may establish the antecedent only if $p.j = level \ \wedge \ p@\{3\} \ \wedge \ q/2^{level} = comp(p/2^{level})$ holds. By (C3), this implies that $p.j = level \ \wedge \ p@\{3\} \ \wedge \ p/2^{level+1} = q/2^{level+1}$ holds. In this case, by (I3), statement $3.p$ establishes the consequent. Statements $13.p$ and $16.p$ preserve $\neg p@\{4..12\}$, and hence do not establish the antecedent. When statement $1.q$ establishes $q@\{14\}$, $q.j \geq \log_2 N$ holds, which implies that $q.j > level$ holds. Thus, although statement $1.q$ may preserve the antecedent, it does not establish it.

Statement $6.q$ may establish the antecedent only if $p.j = level \wedge p@\{4..12\} \wedge q/2^{level} = comp(p/2^{level}) \wedge q.j = level \wedge q@\{6\} \wedge q.rival = -1$ holds. By (I13), this implies that $p.j = level \ \wedge \ p@\{4..12\} \ \wedge \ C[level, p/2^{level}] = p \ \wedge \ q/2^{level} = comp(p/2^{level}) \ \wedge \ q.j = level \ \wedge \ q@\{6\} \ \wedge \ q.rival = -1$ holds. By (C1), this implies that $p.j = level \ \wedge \ p@\{4..12\} \ \wedge \ C[level, comp(q/2^{level})] = p \ \wedge \ p \neq -1 \ \wedge \ q/2^{level} = comp(p/2^{level}) \ \wedge \ q.j = level \ \wedge \ q@\{6\} \ \wedge \ q.rival = -1$ holds. In that case, (I17) implies that the consequent holds.

By (I3), statements $7.q$ and $11.q$ may establish the antecedent only if $p.j =$

$level \wedge p@\{4..12\} \wedge q/2^{level} = comp(p/2^{level}) \wedge q.j = level \wedge q@\{7,11\} \wedge T[level, q/2^{level+1}] \neq q$ holds. By (C3), this implies that $p.j = level \wedge p@\{4..12\} \wedge q/2^{level} = comp(p/2^{level}) \wedge q.j = level \wedge q@\{7,11\} \wedge T[level, q/2^{level+1}] \neq q \wedge p/2^{level+1} = q/2^{level+1}$ holds. By (I16), this implies that $T[level, q/2^{level+1}] = p$ holds.

Statement 12.q may establish the antecedent only if $p.j = level \wedge p@\{4..12\} \wedge q/2^{level} = comp(p/2^{level}) \wedge q.j = level \wedge q@\{12\} \wedge P[level, q] = 2$ holds. By (I13), this implies that $p.j = level \wedge p@\{4..12\} \wedge C[level, p/2^{level}] = p \wedge p \neq -1 \wedge q/2^{level} = comp(p/2^{level}) \wedge q.j = level \wedge q@\{12\} \wedge P[level, q] = 2$ holds. By (C1), this implies that $p.j = level \wedge p@\{4..12\} \wedge C[level, comp(q/2^{level})] = p \wedge p \neq -1 \wedge q/2^{level} = comp(p/2^{level}) \wedge q.j = level \wedge q@\{12\} \wedge P[level, q] = 2$ holds. In that case, by (I21), $T[level, q/2^{level+1}] = p$ holds. This implies that the consequent of (I31) holds. Statement 13.q establishes $q.j > level$ only when $q.j = level \wedge q@\{13\}$ holds, and statement 16.q establishes $q.j = level$ only when $q.j > level$ holds. Thus, although statements 13.q and 16.q may preserve the antecedent, they do not establish it.

(I3) implies that statement 3.i may falsify $T[level, q/2^{level+1}] = p$ only if $i/2^{level+1} = p/2^{level+1} \wedge i.j = level \wedge i@\{3\} \wedge i \neq p$ holds. By (C2), this implies that $(i/2^{level} = p/2^{level} \vee i/2^{level} = comp(p/2^{level})) \wedge i.j = level \wedge i@\{3\} \wedge i \neq p$ holds. Note further that statement 3.i may falsify (I31) only if $q/2^{level} = comp(p/2^{level})$ holds. Thus, statement 3.i may falsify (I31) only if $(i/2^{level} = p/2^{level} \vee i/2^{level} = q/2^{level}) \wedge i.j = level \wedge i@\{3\} \wedge i \neq p$ holds. If $i/2^{level} = p/2^{level} \wedge i.j = level \wedge i \neq p$ holds, then $ME(level)$ implies that $p.j = level$ does not hold. If $i/2^{level} = q/2^{level} \wedge i.j = level \wedge i@\{3\}$ holds, then, by $ME(level)$, this implies that $(q.j > level \vee q.j = level \wedge q@\{13..20\})$

does not hold. □

invariant $p.j = level \ \wedge \ p@\{5..12\} \ \wedge \ q/2^{level} = comp(p/2^{level}) \wedge$

$q.j = level \ \wedge \ q@\{19,20\} \ \Rightarrow$

$q.rival = p \ \vee \ P[level,p] = 0 \ \vee \ P[level,p] = 2$ (I32)

Initially $(\forall i :: i@\{0\})$ holds, and hence (I32) is true. The antecedent is established only by statements $4.p$, $13.p$, $16.p$, $13.q$, $16.q$, and $18.q$. Only statements $5.q$, $18.q$, and $9.i$ may falsify the consequent. Statement $4.p$ may establish the antecedent only if $p.j = level$ holds. In this case, it also establishes $P[level,p] = 0$. Statements $13.p$ and $16.p$ preserve $\neg p@\{5..12\}$, and hence do not establish the antecedent. Statements $13.q$ and $16.q$ preserve $\neg q@\{19,20\}$, and hence do not establish the antecedent. Statement $18.q$ may falsify (I32) only if $p.j = level \wedge p@\{5..12\} \wedge q/2^{level} = comp(p/2^{level}) \wedge q.j = level \wedge q@\{18\}$ holds. By (I31), this implies that $q.j = level \ \wedge \ q@\{18\} \ \wedge \ T[level, q/2^{level+1}] = p$ holds. In this case, (I3) implies that statement $18.q$ establishes $q.rival = p$. Thus, statement $18.q$ preserves (I32). Although statement $5.q$ may falsify the consequent, it establishes $q@\{6\}$. (I9) implies that statement $9.i$ may falsify the consequent only if $q/2^{level} = comp(p/2^{level}) \wedge p/2^{level} = comp(i/2^{level}) \wedge i.j = level \wedge i@\{9\}$ holds. By (C1), this implies that $q/2^{level} = i/2^{level} \ \wedge \ i.j = level \ \wedge \ i@\{9\}$ holds. In that case, $ME(level)$ implies that $q.j = level \ \wedge \ q@\{19,20\}$ does not hold. □

invariant $p.j = level \ \wedge \ p@\{5..12\} \ \wedge \ (\forall i : i/2^{level} = comp(p/2^{level}) ::$

$i.j < level \ \vee \ i.j = level \ \wedge \ i@\{0..3,15,16\}) \ \Rightarrow$

$$P[level, p] = 0 \ \vee \ P[level, p] = 2 \qquad \text{(I33)}$$

Initially $(\forall i :: i@\{0\})$ holds, and hence (I33) is true. The antecedent may be established only by statements $4.p$, $13.p$, $16.p$, $13.i$, $14.i$, $15.i$, $16.i$, $19.i$, and $20.i$. Only statement $9.i$ may falsify the consequent. The reasoning for statements $4.p$, $13.p$, and $16.p$ is the same as in the proof of (I32). (I2) implies that when statement $14.i$ establishes $i@\{15\}$, $i.j > level$ holds. Thus, statement $14.i$ does not establish the antecedent.

Statement $13.i$ establishes $i.j = level \ \wedge \ i@\{1\}$ only if $i.j < level$ holds. Statement $15.i$ may establish $i.j = level \ \wedge \ i@\{0\}$ only if $i.j = level \ \wedge \ i@\{15\}$ holds. Statement $16.i$ establishes $i.j < level$ only if $i.j = level \ \wedge \ i@\{16\}$ holds. Thus, although statements $13.i$, $15.i$, and $16.i$ may preserve the antecedent, but they do not establish it.

Statements $19.i$ and $20.i$ establish the antecedent only if $p.j = level \ \wedge \ p@\{5..12\} \ \wedge \ i/2^{level} = comp(p/2^{level}) \ \wedge \ i.j = level \ \wedge \ i@\{19, 20\}$ holds. By (I32), this implies that $i.j = level \ \wedge \ i@\{19, 20\} \ \wedge \ (i.rival = p \ \vee \ P[level, p] = 0 \ \vee \ P[level, p] = 2)$ holds. If $i.rival \neq p$, then the consequent of (I33) holds, and is not falsified by statements $19.i$ and $20.i$. If $i.rival = p$, then statement $19.i$ establishes $i@\{20\}$, and statement $20.i$ establishes $P[level, p] = 2$.

(I9) implies that statement $9.i$ may falsify the consequent only if $p/2^{level} = comp(i/2^{level}) \ \wedge \ i.j = level \ \wedge \ i@\{9\}$ holds. This implies, by (C1), that $i/2^{level} = comp(p/2^{level}) \ \wedge \ i.j = level \ \wedge \ i@\{9\}$ holds. In that case, the antecedent does not hold. □

invariant $\quad p.j = level \ \wedge \ (p@\{6\} \ \wedge \ p.rival \neq -1 \ \vee \ p@\{7..12\}) \ \wedge$

$$q/2^{level} = comp(p/2^{level}) \ \wedge \ q.j = level \ \wedge \ q@\{19,20\} \ \Rightarrow \ q.rival = p \quad \text{(I34)}$$

Initially $(\forall i :: i@\{0\})$ holds, and hence (I34) is true. The antecedent may be established only by statements $5.p$, $6.p$, $13.p$, $16.p$, $18.p$, $13.q$, $16.q$, and $18.q$. Only statements $5.q$ and $18.q$ may falsify the consequent. As in the proof of (I18), statement $5.p$ may establish the antecedent only if $i/2^{level} = q/2^{level} \wedge (i.j > level \vee i.j = level \wedge i@\{3..14,17\})$ holds. By $ME(level)$, this implies that the antecedent does not hold. The reasoning for statements $6.p$, $13.p$, $16.p$, $18.p$, $13.q$, and $16.q$ is the same as in the proof of (I18). Statement $18.q$ may falsify (I34) only when $p.j = level \wedge (p@\{6\} \wedge p.rival \neq -1 \vee p@\{7..12\}) \wedge q/2^{level} = comp(p/2^{level}) \wedge q.j = level \wedge q@\{18\}$ holds. By (I31), this implies that $q.j = level \wedge T[level, q/2^{level+1}] = p$ holds. By (I3), this implies that statement $18.q$ establishes the consequent, and hence preserves (I34). Finally, although statement $5.q$ may falsify the consequent, it establishes $q@\{6\}$. □

invariant $p.j = level \ \wedge \ p@\{11,12\} \ \wedge \ (\forall i : i/2^{level} = comp(p/2^{level}) ::$

$$i.j < level \ \vee \ i.j = level \ \wedge \ i@\{0..3,15,16\}) \ \Rightarrow \ P[level,p] = 2 \quad \text{(I35)}$$

Initially $(\forall i :: i@\{0\})$ holds, and hence (I35) is true. The antecedent may be established only by statements $10.p$, $13.p$, $16.p$, $13.i$, $14.i$, $15.i$, $16.i$, $19.i$, and $20.i$. Only statements $4.p$ and $9.i$ may falsify the consequent. Statement $10.p$ may establish the antecedent only if $p.j = level \wedge p@\{10\} \wedge (\forall i : i/2^{level} = comp(p/2^{level}) :: i.j < level \vee i.j = level \wedge i@\{0..3,15,16\}) \wedge P[level,p] \geq 1$ holds. By (I33), this implies that $(P[level,p] = 0 \vee P[level,p] = 2) \wedge P[level,p] \geq 1$ holds. This implies that whenever statement $10.p$ establishes

the antecedent, $P[level, p] = 2$ holds. $\neg p@\{11, 12\}$ holds after the execution of statements $13.p$ and $16.p$.

The reasoning for statements $13.i$, $14.i$, $15.i$, and $16.i$ is the same as given in the proof of (I33). Statements $19.i$ and $20.i$ establish the antecedent only if $p.j = level \wedge p@\{11, 12\} \wedge i/2^{level} = comp(p/2^{level}) \wedge i.j = level \wedge i@\{19, 20\}$ holds. By (I34), this implies that $i.j = level \wedge i@\{19, 20\} \wedge i.rival = p$ holds. In that case, statement $19.i$ establishes $i@\{20\}$, and statement $20.i$ establishes $P[level, p] = 2$.

Although statement $4.p$ may falsify the consequent, it establishes $p@\{5\}$. (I9) implies that statement $9.i$ may falsify the consequent only if $p/2^{level} = comp(i/2^{level}) \wedge i.j = level \wedge i@\{9\}$ holds. By (C1), this implies that $i/2^{level} = comp(p/2^{level}) \wedge i.j = level \wedge i@\{9\}$ holds, which implies that the antecedent does not hold. □

invariant $p.j = level \wedge p@\{5..12\} \wedge q/2^{level} = comp(p/2^{level}) \wedge$
$q.j = level \wedge q@\{9\} \wedge q.rival = p \Rightarrow P[level, p] = 0$ (I36)

Initially $(\forall i :: i@\{0\})$ holds, and hence (I36) is true. The antecedent may be established only by statements $4.p$, $13.p$, $16.p$, $5.q$, $8.q$, $13.q$, $16.q$, and $18.q$. Only statements $9.i$ and $20.i$ may falsify the consequent. Statement $4.p$ may establish the antecedent only if $p.j = level$ holds. In this case, it also establishes $P[level, p] = 0$. $\neg p@\{5..12\}$ holds after the execution of statements $13.p$ and $16.p$. $\neg q@\{9\}$ holds after the execution of statements $5.q$, $13.q$, $16.q$, and $18.q$. Statement $8.q$ establishes the antecedent only when $q.j = level \wedge q@\{8\} \wedge q.rival = p \wedge P[level, q.rival] = 0$ holds. This implies that the consequent

holds.

(I9) implies that statement $9.i$ may falsify (I36) only if $i.j = level \wedge i@\{9\} \wedge q/2^{level} = comp(p/2^{level}) \wedge p/2^{level} = comp(i/2^{level})$ holds. By (C1), this implies that $i.j = level \wedge i@\{9\} \wedge q/2^{level} = i/2^{level}$ holds. If $i = q$, then statement $9.i$ establishes $q@\{10\}$. If $i \neq q$, then $ME(level)$ implies that $q.j = level$ does not hold. Thus, statement $9.i$ preserves (I36). (I11) implies that statement $20.i$ may falsify (I36) only if $q/2^{level} = comp(p/2^{level}) \wedge p/2^{level+1} = i/2^{level+1} \wedge i.j = level \wedge i@\{20\}$ holds. By (C1), this implies that $comp(q/2^{level}) = p/2^{level} \wedge q/2^{level} = comp(p/2^{level}) \wedge p/2^{level+1} = i/2^{level+1} \wedge i.j = level \wedge i@\{20\}$ holds. By (C2), this implies that $(i/2^{level} = p/2^{level} \vee i/2^{level} = q/2^{level}) \wedge i.j = level \wedge i@\{20\}$ holds. If $i = p$, then statement $20.i$ establishes $p@\{15\}$. If $i/2^{level} = p/2^{level} \wedge i \neq p \wedge i.j = level \wedge i@\{20\}$ holds, then $ME(level)$ implies that $p.j = level$ does not hold. If $i = q$, then statement $20.i$ establishes $q@\{15\}$. If $i/2^{level} = q/2^{level} \wedge i \neq q \wedge i.j = level \wedge i@\{20\}$ holds, then $ME(level)$ implies that $q.j = level$ does not hold. □

invariant $p.j = level \wedge p@\{12\} \wedge q/2^{level} = comp(p/2^{level}) \wedge$

$q.j = level \wedge q@\{4..10\} \wedge T[level, p/2^{level+1}] = q \Rightarrow P[level, p] = 2$ (I37)

Initially $(\forall i :: i@\{0\})$ holds, and hence (I37) is true. The antecedent may be established only by statements $11.p$, $13.p$, $16.p$, $3.q$, $13.q$, and $16.q$. Only statements $4.p$ and $9.i$ may falsify the consequent. (I3) implies that statement $11.p$ may establish the antecedent only if $p.j = level \wedge p@\{11\} \wedge T[level, p/2^{level+1}] = p$ holds. Thus, statement $11.p$ does not establish the antecedent. $\neg p@\{12\}$ holds after the execution of statements $13.p$ and $16.p$.

Statement $3.q$ establishes the antecedent only if $p.j = level \ \wedge \ p@\{12\} \ \wedge \ q/2^{level} = comp(p/2^{level}) \ \wedge \ q.j = level \ \wedge \ q@\{3\}$ holds. By $ME(level)$, this implies that $p.j = level \ \wedge \ p@\{12\} \ \wedge \ q/2^{level} = comp(p/2^{level}) \ \wedge \ q.j = level \ \wedge \ q@\{3\} \ \wedge \ (\forall i : i/2^{level} = comp(p/2^{level}) \ \wedge \ i \neq q :: i.j < level)$ holds. In that case, (I35) implies that $P[level, p] = 2$ holds. $\neg q@\{4..10\}$ holds after the execution of statements $13.q$ and $16.q$.

Although statement $4.p$ may falsify the consequent, it establishes $p@\{5\}$. As in the proof of (I36), statement $9.i$ may falsify (I37) only if $i/2^{level} = q/2^{level} \ \wedge \ i.j = level \ \wedge \ i@\{9\}$ holds. If $i = q$, then (I36) implies that statement $9.i$ does not falsify the consequent when the antecedent holds. If $i \neq q$, then $ME(level)$ implies that $q.j = level$ does not hold. □

Now, we prove that the algorithm is free from starvation. To facilitate the presentation, we define the following predicate.

$$SF(level) \ \equiv \ (\forall i :: i.j = level \ \wedge \ i@\{1..13\} \ \mapsto \ i@\{14\})$$

We next prove that the following assertion holds, which implies that the starvation-freedom property holds.

$$(\forall n : 0 \leq n \leq \log_2 N :: SF(n)) \qquad \text{(G2)}$$

We use an induction on n in the proof. Observe that $SF(\log_2 N) \ \equiv \ (\forall i :: i.j = \log_2 N \ \wedge \ i@\{1..13\} \ \mapsto \ i@\{14\})$. By the contrapositive of (I1), $(\forall i :: i.j = \log_2 N \ \Rightarrow \ \neg i@\{2..13\})$ holds. By the definition of a fair history and the pro-

gram text, $(\forall i :: i.j = \log_2 N \ \wedge \ i@\{1\} \ \mapsto \ i@\{14\})$ holds. These two assertions imply that the induction base $SF(\log_2 N)$ holds. Thus, it suffices to prove that the following assertion holds.

$$(\forall level : 0 \leq level < \log_2 N :: (\forall j : j > level :: SF(j)) \ \Rightarrow \ SF(level)) \quad \text{(G3)}$$

Next, we prove several assertions that are needed to establish $SF(level)$. In these proofs, we assume that $(\forall j : j > level :: SF(j))$ holds.

$$i.j > level \ \mapsto \ i.j = level \ \wedge \ i@\{19\} \quad \text{(L1)}$$

By (I0), $i.j > level \ \Rightarrow \ i@\{1..20\}$. By our assumption that $(\forall j : j > level :: SF(j))$ holds, by (I1), $i.j > level \wedge i@\{1..13\} \ \mapsto \ i.j > level \wedge i@\{14\}$. By the definition of a fair history and the program text, $i.j > level \ \wedge \ i@\{14..20\} \ \mapsto \ i.j = level \ \wedge \ i@\{19\}$. □

Now, we prove two *unless* assertions.

$$i.j = level \ \wedge \ i@\{10\} \ \wedge \ P[level, i] \geq 1 \ \ \textit{unless} \ \ i.j = level \ \wedge \ i@\{11\} \quad \text{(U0)}$$

To prove that (U0) holds, it suffices to consider only those statements that may falsify $i.j = level \ \wedge \ i@\{10\} \ \wedge \ P[level, i] \geq 1$. The statements to consider are $4.i$, $10.i$, $13.i$, and $16.i$. When statements $4.i$, $13.i$, or $16.i$ are enabled for execution, $\neg i@\{10\}$ holds, which implies that these statements preserve (U0). Statement $10.i$ establishes $i.j = level \ \wedge \ i@\{11\}$, if executed when $i.j = level$

holds, and hence preserves (U0).

$$i.j = level \ \wedge \ i@\{12\} \ \wedge \ P[level, i] = 2 \ \ unless \ \ i.j = level \ \wedge \ i@\{13\} \qquad \text{(U1)}$$

To prove that (U1) holds, it suffices to consider only those statements that may falsify $i.j = level \ \wedge \ i@\{12\} \ \wedge \ P[level, i] = 2$. The statements to consider are $4.i$, $12.i$, $13.i$, $16.i$, and $9.q$. When statements $4.i$, $13.i$, or $16.i$ are enabled for execution, $\neg i@\{12\}$ holds, which implies that these statements preserve (U1). Statement $12.i$ establishes $i.j = level \ \wedge \ i@\{13\}$, if executed when $i.j = level$ holds, and hence preserves (U1). (I9) implies that statement $9.q$ may falsify (U1) only when $i.j = level \ \wedge \ i@\{12\} \ \wedge \ q.j = level \ \wedge \ q@\{9\} \ \wedge \ i/2^{level} = comp(q/2^{level})$ holds. By (C1), this implies that $i.j = level \ \wedge \ i@\{12\} \ \wedge \ q.j = level \ \wedge \ q@\{9\} \ \wedge \ q/2^{level} = comp(i/2^{level})$ holds. In this case, (I36) implies that $q.rival \neq i \ \vee \ P[level, i] = 0$ holds, which implies that statement $9.q$ does not falsify (U1).

The next two assertions follow from these *unless* assertions, the definition of a fair history, and the program text; (U0) is used to prove (L2) and (U1) is used to prove (L3).

$$i.j = level \ \wedge \ i@\{10\} \ \wedge \ P[level, i] \geq 1 \ \mapsto \ i.j = level \ \wedge \ i@\{11\} \qquad \text{(L2)}$$

$$i.j = level \ \wedge \ i@\{12\} \ \wedge \ P[level, i] = 2 \ \mapsto \ i.j = level \ \wedge \ i@\{13\} \qquad \text{(L3)}$$

The following assertions, which are stated without proof, follow directly from

the definition of a fair history and the program text.

$$i.j = level \ \wedge \ i@\{10\} \ \wedge \ q/2^{level} = comp(i/2^{level}) \ \wedge \ q.j = level \ \wedge \ q@\{3..9\} \ \mapsto$$
$$(i.j = level \ \wedge \ i@\{10\} \ \wedge \ q/2^{level} = comp(i/2^{level}) \ \wedge \ q.j = level \wedge$$
$$q@\{10, 13\}) \ \vee \ (i.j = level \ \wedge \ i@\{11\}) \qquad \text{(L4)}$$

$$i.j = level \ \wedge \ i@\{10\} \ \wedge \ q/2^{level} = comp(i/2^{level}) \ \wedge \ q.j = level \ \wedge \ q@\{11\} \ \mapsto$$
$$(i.j = level \ \wedge \ i@\{10\} \ \wedge \ q/2^{level} = comp(i/2^{level}) \ \wedge \ q.j = level \wedge$$
$$q@\{12, 13\}) \ \vee \ (i.j = level \ \wedge \ i@\{11\}) \qquad \text{(L5)}$$

$$i.j = level \ \wedge \ i@\{10\} \ \wedge \ q/2^{level} = comp(i/2^{level}) \ \wedge \ q.j = level \ \wedge \ q@\{13\} \ \mapsto$$
$$(i.j = level \ \wedge \ i@\{10\} \ \wedge \ q/2^{level} = comp(i/2^{level}) \ \wedge \ q.j > level) \vee$$
$$(i.j = level \ \wedge \ i@\{11\}) \qquad \text{(L6)}$$

Assertions (L7) through (L13), given next, easily follow from the preceding assertions. In particular, (L1) implies that (L7) holds; (I34) and (L7) imply that (L8) holds; (I34) and (L8) imply that (L9) holds; (L9) implies that (L10) holds; (L2) and (L10) imply that (L11) holds; (I27) implies that (L12) holds; and (L2) implies that (L13) holds.

$$i.j = level \ \wedge \ i@\{10\} \ \wedge \ q/2^{level} = comp(i/2^{level}) \wedge$$
$$(q.j > level \ \vee \ q.j = level \ \wedge \ q@\{17..19\}) \mapsto$$
$$(i.j = level \wedge i@\{10\} \wedge q/2^{level} = comp(i/2^{level}) \wedge q.j = level \wedge q@\{19\}) \vee$$
$$(i.j = level \ \wedge \ i@\{11\}) \qquad \text{(L7)}$$

$$i.j = level \ \wedge \ i@\{10\} \ \wedge \ q/2^{level} = comp(i/2^{level}) \ \wedge$$
$$(q.j > level \ \vee \ q.j = level \ \wedge \ q@\{17..19\}) \mapsto$$
$$(i.j = level \ \wedge \ i@\{10\} \ \wedge \ q/2^{level} = comp(i/2^{level}) \ \wedge \ q.j = level \ \wedge \ q@\{19\}$$
$$\wedge \ q.rival = i) \ \vee \ (i.j = level \ \wedge \ i@\{11\}) \tag{L8}$$

$$i.j = level \ \wedge \ i@\{10\} \ \wedge \ q/2^{level} = comp(i/2^{level}) \ \wedge$$
$$(q.j > level \ \vee \ q.j = level \ \wedge \ q@\{17..20\}) \mapsto$$
$$(i.j = level \wedge i@\{10\} \wedge q/2^{level} = comp(i/2^{level}) \wedge q.j = level \wedge q@\{20\} \wedge$$
$$q.rival = i) \ \vee \ (i.j = level \ \wedge \ i@\{11\}) \tag{L9}$$

$$i.j = level \ \wedge \ i@\{10\} \ \wedge \ q/2^{level} = comp(i/2^{level}) \ \wedge$$
$$(q.j > level \ \vee \ q.j = level \ \wedge \ q@\{17..20\}) \mapsto$$
$$(i.j = level \ \wedge \ i@\{10\} \ \wedge \ P[level, i] = 2) \vee$$
$$(i.j = level \ \wedge \ i@\{11\}) \tag{L10}$$

$$i.j = level \ \wedge \ i@\{10\} \ \wedge \ q/2^{level} = comp(i/2^{level}) \ \wedge$$
$$(q.j > level \ \vee \ q.j = level \ \wedge \ q@\{17..20\}) \mapsto$$
$$i.j = level \ \wedge \ i@\{11\} \tag{L11}$$

$$i.j = level \ \wedge \ i@\{10\} \ \wedge \ q/2^{level} = comp(i/2^{level}) \ \wedge \ q.j = level \ \wedge \ q@\{10\} \ \mapsto$$
$$i.j = level \wedge i@\{10\} \wedge q/2^{level} = comp(i/2^{level}) \wedge q.j = level \wedge q@\{10\} \wedge$$
$$(P[level, i] \geq 1 \ \vee \ P[level, q] \geq 1) \tag{L12}$$

$$i.j = level \ \wedge \ i@\{10\} \ \wedge \ q/2^{level} = comp(i/2^{level}) \ \wedge \ q.j = level \ \wedge \ q@\{10\} \wedge$$
$$(P[level, i] \geq 1 \ \vee \ P[level, q] \geq 1) \mapsto$$

$$(i.j = level \wedge i@\{10\} \wedge q/2^{level} = comp(i/2^{level}) \wedge q.j = level \wedge q@\{11\}) \vee$$
$$(i.j = level \ \wedge \ i@\{11\}) \qquad \text{(L13)}$$

invariant $\quad i.j = level \ \wedge \ i@\{10\} \ \wedge \ q/2^{level} = comp(i/2^{level}) \wedge$

$q.j = level \ \wedge \ q@\{12\} \ \Rightarrow$

$$(i.j = level \ \wedge \ i@\{10\} \ \wedge \ P[level, i] \geq 1) \vee$$
$$(i.j = level \ \wedge \ i@\{10\} \ \wedge \ q/2^{level} = comp(i/2^{level}) \wedge$$
$$q.j = level \ \wedge \ q@\{12\} \ \wedge \ P[level, q] = 2) \qquad \text{(I38)}$$

By (I16), the antecedent implies that $i.j = level \ \wedge \ i@\{10\} \ \wedge \ q/2^{level} = comp(i/2^{level}) \ \wedge \ q.j = level \ \wedge \ q@\{12\} \ \wedge \ (T[level, i/2^{level+1}] = q \ \vee \ T[level, i/2^{level+1}] = i)$ holds. By (C1) and (C3), this implies that $(i.j = level \wedge i@\{10\} \wedge q/2^{level} = comp(i/2^{level}) \wedge q.j = level \wedge q@\{12\} \wedge T[level, q/2^{level+1}] = q) \ \vee \ (i.j = level \ \wedge \ i@\{10\} \ \wedge \ q/2^{level} = comp(i/2^{level}) \ \wedge \ i/2^{level} = comp(q/2^{level}) \ \wedge \ q.j = level \ \wedge \ q@\{12\} \ \wedge \ T[level, q/2^{level+1}] = i)$ holds. By (I26) and (I37), this implies that the consequent of (I38) holds. □

$$i.j = level \ \wedge \ i@\{10\} \ \wedge \ q/2^{level} = comp(i/2^{level}) \ \wedge \ q.j = level \ \wedge \ q@\{12\} \ \mapsto$$
$$(i.j = level \wedge i@\{10\} \wedge q/2^{level} = comp(i/2^{level}) \wedge q.j = level \wedge q@\{13\}) \vee$$
$$(i.j = level \ \wedge \ i@\{11\}) \qquad \text{(L14)}$$

By (L2) and (L3), (I38) implies that (L14) holds. □

$$i.j = level \wedge i@\{10\} \wedge q/2^{level} = comp(i/2^{level}) \wedge q.j = level \wedge q@\{3..14\} \ \mapsto$$
$$(i.j = level \ \wedge \ i@\{10\} \ \wedge \ q/2^{level} = comp(i/2^{level}) \ \wedge \ q.j > level) \vee$$

$$(i.j = level \ \wedge \ i@\{11\}) \tag{L15}$$

(I2) implies that $\neg(q.j = level \ \wedge \ q@\{14\})$ holds. (Recall that $level < \log_2 N$.) Hence, (L4), (L5), (L6), (L12), (L13), (L14), and $\neg(q.j = level \ \wedge \ q@\{14\})$ imply that (L15) holds. □

$$i.j = level \wedge i@\{10\} \wedge q/2^{level} = comp(i/2^{level}) \wedge q.j = level \wedge q@\{3..14\} \ \mapsto$$
$$(i.j = level \ \wedge \ i@\{11\}) \tag{L16}$$

(L11) and (L15) imply that (L16) holds. □

$$i.j = level \ \wedge \ i@\{10\} \ \mapsto \ i.j = level \ \wedge \ i@\{11\} \tag{L17}$$

(I24) implies that $i.j = level \wedge i@\{10\} \ \Rightarrow \ (i.j = level \wedge i@\{10\} \wedge P[level, i] = 2) \ \vee \ (\exists q :: i.j = level \ \wedge \ i@\{10\} \ \wedge \ q/2^{level} = comp(i/2^{level}) \ \wedge \ (q.j > level \vee q.j = level \wedge q@\{17..20\})) \vee (\exists q :: i.j = level \wedge i@\{10\} \wedge q/2^{level} = comp(i/2^{level}) \ \wedge \ q.j = level \ \wedge \ q@\{3..14\})$. By (L2), (L11), and (L16), this implies that (L17) holds. □

By proving assertions similar to (I38) and (L4) through (L16), it is possible to establish (L18), given next, which is similar to (L17). For brevity, we omit the proof of (L18).

$$i.j = level \ \wedge \ i@\{12\} \ \mapsto \ i.j = level \ \wedge \ i@\{13\} \tag{L18}$$

Note that (L17) implies that the first busy-waiting loop of process i terminates, while (L18) implies that the second busy-waiting loop of i terminates.

$$i.j = level \ \wedge \ i@\{1..13\} \ \mapsto \ i.j = level + 1 \ \wedge \ i@\{1\} \qquad \text{(L19)}$$

By the program text and the definition of a fair history, (L17) and (L18) imply that (L19) holds. (Recall, by assumption, that $level < \log_2 N$. Thus, $i.j = level \ \wedge \ i@\{1\} \ \mapsto \ i.j = level \ \wedge \ i@\{2\}$.) □

Observe that if $(\forall j : j > level :: SF(j))$ holds, then, by (L19), that $SF(level)$ holds. This concludes the proof of (G3). Thus, we conclude that the program in Figures 2.2 and 2.3 is free from starvation.

2.5 Fast Mutual Exclusion in the Absence of Contention

As discussed in Section 2.1, most early mutual exclusion algorithms based on read/write atomicity are neither fast in the absence of contention, nor able to cope with high contention. Because Lamport's fast mutual exclusion algorithm induces $O(1)$ remote operations in the absence of contention, and our mutual exclusion algorithm requires $O(\log N)$ remote operations given any level of contention, it seems reasonable to expect a solution to exist that induces $O(1)$ remote operations when contention is absent, and $O(\log N)$ remote operations when contention is high.

The algorithm given in Figures 2.4 and 2.5 almost achieves that goal. The

shared var B : **array**$[0..N-1]$ **of boolean**;
$X, Y : -1..N-1$;
Z : **boolean**
initially $Y = -1 \ \wedge \ Z = false \ \wedge \ (\forall i :: B[i] = false)$

process i
private var $flag$: **boolean**;
$n : 1..N$

Figure 2.4: Variable declarations for fast, scalable mutual exclusion algorithm.

basic idea of this modification is to combine Lamport's fast mutual exclusion algorithm and our algorithm, specifically by placing an extra two-process version of our algorithm "on top" of the arbitration tree. The "left" entry section of this extra two-process program (i.e., process u's code in Figure 2.1) is executed by a process if that process detects no contention. The "right" entry section of this extra program (i.e., process v's code in Figure 2.1) is executed by the winning process from the arbitration tree. A process will compete within the arbitration tree (as before) if it detects any contention. As seen in Figure 2.5, the scheme used to detect contention is similar to that used in Lamport's algorithm. In this figure, we use $ENTRY_k$ and $EXIT_k$ to denote the entry and exit sections of the k-process version of our algorithm.

It should be clear that, in the absence of contention, a process enters its critical section after executing $O(1)$ remote operations. Also, in the presence of contention, a process enters its critical section after executing $O(\log N)$ remote operations. However, when a period of contention ends, N remote operations might be required in order to re-open the fast entry section — see the **while** loop at line 22 in Figure 2.5. Nonetheless, performance studies show that, under

```
process i

while true do
    0: Noncritical Section;
    1: X := i;
    2: if Y ≠ −1 then goto 14 fi;
    3: Y := i;
    4: if X ≠ i then goto 14 fi;
    5: B[i] := true;
    6: if Z then goto 14 fi;
    7: if Y ≠ i then goto 14 fi;
    8: ENTRY_2;                          /* Two-Process Entry Section */
    9:    Critical Section;
   10: EXIT_2;                           /* Two-Process Exit Section */
   11: Y := −1;
   12: B[i] := false;
   13: goto 0;
   14: ENTRY_N;                          /* Arbitration Tree */
   15:    ENTRY_2;                       /* Two-Process Entry Section */
   16:       Critical Section;
   17:       B[i] := false;
   18:       if X = i then
   19:          Z := true;
   20:          flag := true;
   21:          n := 0;
   22:          while (n < N) do
   23:             if B[n] then flag := false fi;
   24:             n := n + 1
                od;
   25:          if flag then Y := −1 fi;
   26:          Z := false
             fi;
   27:    EXIT_2;                        /* Two-Process Exit Section */
   28: EXIT_N                            /* Arbitration Tree */
od
```

Figure 2.5: Fast, scalable mutual exclusion algorithm.

high contention, these statements are rarely executed. (Under low contention, they are obviously never executed.) For example, out of the 100,000 critical section executions in one experiment, these N statements were performed after only 55 critical section executions in the four-process case, and after only one in the eight- and sixteen-process cases.

In the absence of contention, our algorithm generates about twice as many remote memory operations as Lamport's. However, under high contention, our algorithm is clearly superior, as Lamport's induces an unbounded number of remote operations. Also, our modified algorithm ensures starvation-freedom, whereas Lamport's algorithm does not.

In the rest of this section, we prove that the mutual exclusion and starvation-freedom properties hold for mutual exclusion of Figures 2.4 and 2.5. We first prove five invariants that are needed to prove that mutual exclusion holds. The first three are quite simple: (I39) follows from (G0), (I40) follows directly from the program text, and (I41) follows from (I39).

invariant $(\mathbf{N}i :: i@\{15..27\}) \leq 1$ (I39)

invariant $i@\{6..12\} \Rightarrow B[i]$ (I40)

invariant $i@\{20..26\} \Rightarrow Z$ (I41)

invariant $i.flag \wedge ((i@\{22,23\} \wedge i.n > p) \vee (i@\{24\} \wedge i.n \geq p)) \Rightarrow$
$\neg p@\{7..12\}$ (I42)

Initially $(\forall i :: i@\{0\})$ holds, and hence (I42) is true. The antecedent may be established only by statements $20.i$, $21.i$, $23.i$, and $24.i$. Only statement $6.p$ may falsify the consequent. $\neg i@\{22..24\}$ holds after the execution of statement $20.i$. Although statement $21.i$ establishes $i@\{22\}$, it also establishes $i.n = 0$, which implies that the antecedent does not hold. Statement $23.i$ may falsify (I42) only if $i.flag \ \wedge \ i@\{23\} \ \wedge \ i.n = p \ \wedge \ \neg B[i.n] \ \wedge \ p@\{7..12\}$ holds. By (I40), this is a contradiction. Thus, statement $23.i$ preserves (I42). Statement $24.i$ may establish $i@\{22\} \ \wedge \ i.n > p$ only if $i@\{24\} \ \wedge \ i.n \geq p$ holds. Thus, although statement $24.i$ may preserve the antecedent, it does not establish it. Statement $6.p$ may falsify the consequent only when Z holds. By (I41), this implies that the antecedent does not hold. □

invariant $\quad i@\{25\} \ \wedge \ i.flag \ \Rightarrow \ (\forall p :: \neg p@\{7..12\})$ (I43)

Initially $(\forall i :: i@\{0\})$ holds, and hence (I43) is true. The antecedent may be established only by statements $20.i$ and $22.i$. Only statement $6.q$ may falsify the consequent. $\neg i@\{25\}$ holds after the execution of statement $20.i$. Statement $22.i$ may establish $i@\{25\} \ \wedge \ i.flag$ only if $i@\{22\} \ \wedge \ i.flag \ \wedge \ i.n \geq N$ holds. By (I42), this implies that the consequent holds. Statement $6.q$ may falsify the consequent only when Z holds. By (I41), this implies that the antecedent does not hold. □

The following assertion implies that the mutual exclusion property holds for the fast entry section.

invariant $((\mathbf{N}i :: i@\{2\} \ \wedge \ X = i \ \wedge \ Y = -1) + (\mathbf{N}i :: i@\{3\} \ \wedge \ X = i) +$
$(\mathbf{N}i :: i@\{4\} \ \wedge \ X = i \ \wedge \ Y = i) + (\mathbf{N}i :: i@\{5..7\} \ \wedge \ Y = i) +$
$(\mathbf{N}i :: i@\{8..11\})) \leq 1 \ \wedge \ ((\exists p :: p@\{8..11\}) \ \Rightarrow \ Y \neq -1)$ (I44)

Initially $(\forall i :: i@\{0\})$ holds, and hence (I44) is true. (I44) may be falsified only by statements $1.q$, $2.q$, $3.q$, $4.q$, $7.q$, $11.q$, and $25.q$.

Statement $2.q$ may increment $(\mathbf{N}i :: i@\{3\} \ \wedge \ X = i)$ only if $q@\{2\} \ \wedge \ X = q \ \wedge \ Y = -1$ holds. Statement $3.q$ may increment $(\mathbf{N}i :: i@\{4\} \ \wedge \ X = i \ \wedge \ Y = i)$ only if $q@\{3\} \ \wedge \ X = q$ holds. Statement $4.q$ may increment $(\mathbf{N}i :: i@\{5..7\} \ \wedge \ Y = i)$ only if $q@\{4\} \ \wedge \ X = q \ \wedge \ Y = q$ holds. Statement $7.q$ may increment $(\mathbf{N}i :: i@\{8..11\})$ only if $q@\{7\} \ \wedge \ Y = q$ holds. Thus, statements $2.q$, $3.q$, $4.q$, and $7.q$ preserve (I44).

Statement $1.q$ may falsify (I44) only if $q@\{1\} \ \wedge \ Y = -1 \ \wedge \ ((\exists p :: p@\{8..11\}) \ \Rightarrow \ Y \neq -1)$ holds. This implies that $(\mathbf{N}i :: i@\{4\} \ \wedge \ X = i \ \wedge \ Y = i) = 0 \ \wedge \ (\mathbf{N}i :: i@\{5..7\} \ \wedge \ Y = i) = 0 \ \wedge \ (\mathbf{N}i :: i@\{8..11\}) = 0$ holds. Because $(\mathbf{N}i :: X = i) \leq 1$ holds, $(\mathbf{N}i :: i@\{2\} \ \wedge \ X = i \ \wedge \ Y = -1) + (\mathbf{N}i :: i@\{3\} \ \wedge \ X = i) \ \leq \ 1$ also holds, which implies that statement $1.q$ preserves (I44).

Statement $7.q$ establishes $q@\{8\}$ only if $Y = q$ holds, and hence preserves (I44). Statement $11.q$ could potentially falsify (I44) by establishing $Y = -1$. However, statement $11.q$ decrements $(\mathbf{N}i :: i@\{8..11\})$ by 1, and hence establishes $(\mathbf{N}i :: i@\{4\} \ \wedge \ X = i \ \wedge \ Y = i) = 0 \ \wedge \ (\mathbf{N}i :: i@\{5..7\} \ \wedge \ Y = i) = 0 \ \wedge \ (\mathbf{N}i :: i@\{8..11\}) = 0$. Because $(\mathbf{N}i :: X = i) \leq 1$ holds, $(\mathbf{N}i :: i@\{2\} \ \wedge \ X = i \ \wedge \ Y = -1) + (\mathbf{N}i :: i@\{3\} \ \wedge \ X = i) \ \leq \ 1$ also holds, which implies that statement $11.q$ preserves (I44).

Statement 25.q could potentially falsify (I44) by establishing $Y = -1$. However, statement 25.q may establish $Y = -1$ only if $q@\{25\} \ \wedge \ q.flag$ holds. By (I43), this implies that $Y = -1 \ \wedge \ (\forall i :: \neg i@\{7..12\})$ holds, which implies that statement 25.q establishes $(\mathbf{N}i :: i@\{4\} \ \wedge \ X = i \ \wedge \ Y = i) = 0 \ \wedge \ (\mathbf{N}i :: i@\{5..7\} \ \wedge \ Y = i) = 0 \ \wedge \ (\mathbf{N}i :: i@\{8..11\}) = 0$. Because $(\mathbf{N}i :: X = i) \leq 1$ holds, $(\mathbf{N}i :: i@\{2\} \ \wedge \ X = i \ \wedge \ Y = -1) + (\mathbf{N}i :: i@\{3\} \ \wedge \ X = i) \ \leq \ 1$ also holds, which implies that statement 25.q preserves (I44). □

ENTRY$_2$ and EXIT$_2$ satisfy the following properties.

invariant $\quad ((\mathbf{N}i :: i@\{8..10\}) \leq 1 \ \wedge \ (\mathbf{N}i :: i@\{15..27\}) \leq 1) \ \Rightarrow$
$(\mathbf{N}i :: i@\{9, 16..26\}) \leq 1$ (I45)

$$i@\{8\} \quad \mapsto \quad i@\{9\} \tag{L20}$$

$$i@\{15\} \quad \mapsto \quad i@\{16\} \tag{L21}$$

The proof of (I45) is similar to that of (G1), and is omitted for brevity. The proof of (L20) and (L21) are similar to that of (G3), and are omitted for brevity. (Note that process identifiers and function *comp* are used for convenience in the proof of (G1) and (G3).)

The following two assertions imply that the mutual exclusion and starvation-freedom properties hold for the algorithm of Figures 2.4 and 2.5.

invariant $\quad (\mathbf{N}i :: i@\{9, 16\}) \leq 1$ (I46)

(I39), (I44), and (I45) imply that (I46) holds. □

invariant $i@\{1..8, 14, 15\} \mapsto i@\{9, 16\}$ (L22)

By the program text, $i@\{1..7\} \mapsto i@\{8, 14\}$. (G2) implies that $i@\{14\} \mapsto i@\{15\}$. Hence, (L22) follows from (L20) and (L21). □

2.6 Performance Results

To compare the scalability of our mutual exclusion algorithm with that of other algorithms, we conducted a number of experiments on the BBN TC2000 and Sequent Symmetry multiprocessors. Results from some of these experiments are presented in this section.

BBN TC2000

The BBN TC2000 is a distributed shared memory multiprocessor, each node of which contains a processor and a memory unit. Each node's processor, a Motorola 88100, provides an atomic fetch-and-store instruction called `xmem`. Other strong primitives such as compare-and-swap and fetch-and-add are provided using the TC2000 hardware locking protocol [13].

We tested seven mutual exclusion algorithms on the TC2000: a simple test-and-set algorithm; the queue-based algorithm using compare-and-swap given by Mellor-Crummey and Scott in [45]; the queue-based algorithm using fetch-and-add given by T. Anderson in [9]; the fast mutual exclusion algorithm given by Lamport in [37]; the tree-based algorithm given by Styer in [56]; the tree-based

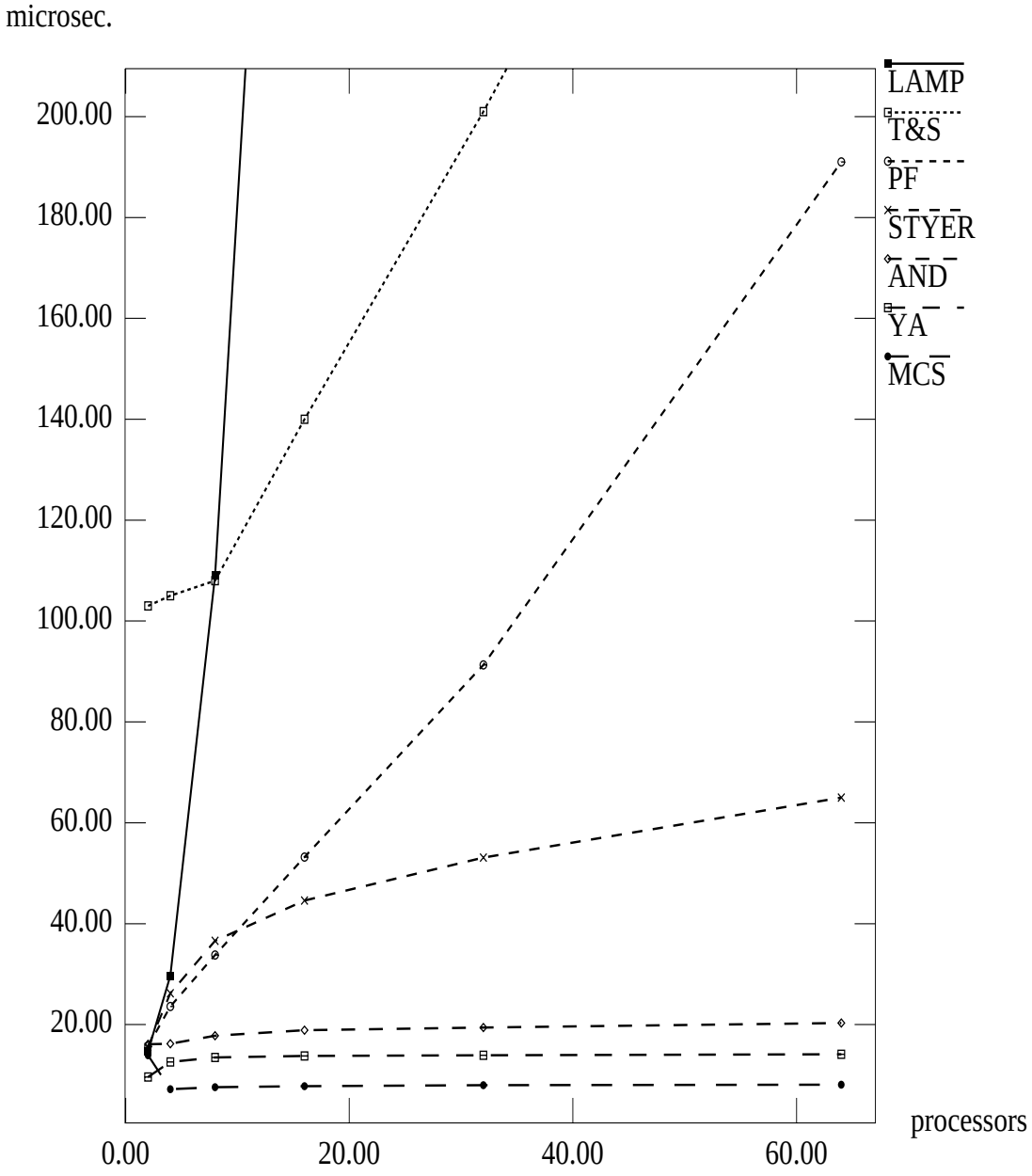

Figure 2.6: Performance results on the TC2000.

algorithm given by Peterson and Fischer in [52]; and the mutual exclusion algorithm described in Section 2.3. Performance results obtained by running these seven algorithms on the TC2000 are summarized in Figure 2.6. Each point (x, y) in each graph represents the average time y for one critical section execution with x competing processors. The timing results summarized in the graph were obtained by averaging over 10^5 critical section executions. The critical section consists of a read and an increment of shared counter. Results obtained using larger critical sections show similar performance to that depicted in Figure 2.6. The timing results presented include the execution time of critical sections.

The performance of the test-and-set algorithm is given by the graph labeled T&S, Mellor-Crummey and Scott's algorithm by the graph labeled MCS, T. Anderson's algorithm by the graph labeled AND, Lamport's algorithm by the graph labeled LAMP, Styer's algorithm by the graph labeled STYER, Peterson and Fischer's algorithm by the graph labeled PF, and our algorithm by the graph labeled YA. On the TC2000, the MCS algorithm was the best overall performer of the alternatives considered here. The graph depicted for the MCS algorithm is mostly flat, except at the point for two processors. This anomaly at two processors coincides with results reported by Mellor-Crummey and Scott on the Sequent Symmetry, and was attributed by them to the lack of a compare-and-swap instruction on the Symmetry [45]. As our implementation of their algorithm did employ compare-and-swap, we have not found a satisfying explanation for this behavior on the TC2000.

T. Anderson's algorithm requires only local spinning when implemented on a machine with coherent caches. On the Symmetry, where each process can spin on its own coherent cache, Anderson's algorithm outperforms the MCS algorithm.

However, on the TC2000, which does not support coherent caching, Anderson's algorithm requires remote spinning, slowing its performance.

The simple T&S algorithm exhibited poor scalability. The average execution time for the 64 processor case, which is not depicted in Figure 2.6, is about 330 microseconds. Where there is a possibility of contention among a large number of processors, it should be avoided, or used with good backoff scheme [2].

Three algorithms based on atomic reads and writes — Lamport's, Peterson and Fischer's, and Styer's — also showed poor scalability. In particular, the performance of Lamport's algorithm degrades dramatically as the number of contenders increases. The average execution time for the 64 processor case, which is not depicted in Figure 2.6, is about 4000 microseconds. The performance of Styer's algorithm, which is better than that of Lamport's, is due to the tree structure employed. Styer's algorithm generates $O(\log N)$ remote operations outside of busy-waiting loops. Even though Peterson and Fischer's algorithm is also tree-based, it induces $O(N)$ remote operations outside of busy-waiting loops, which results in poorer scalability.

Our mutual exclusion algorithm shows performance that is comparable to that of T. Anderson's and Mellor-Crummey and Scott's algorithms. Its good scalability emphasizes the importance of local spinning. The difference seen between our mutual exclusion algorithm and the MCS algorithm is explained by the amount of global traffic generated by each algorithm. The MCS algorithm generates $O(1)$ remote operations per critical section execution, whereas ours generates $O(\log N)$. The global traffic of the other five algorithms is unbounded, as each employs global spinning. The performance of T. Anderson's algorithm is far better than that of the simple test-and-set algorithm. Because the processes

in Anderson's algorithm spin globally on the TC2000, this might be interpreted as a counterexample to our belief that minimizing remote operations is important for good scalability. However, Mellor-Crummey and Scott reported in [45] that Anderson's algorithm produced far fewer remote operations than the test-and-set algorithm.

Sequent Symmetry

Performance results of experiments on the Sequent Symmetry are summarized in Figure 2.6. Cache coherence is maintained by a snoopy protocol. The Symmetry provides an atomic fetch-and-store instruction. Because other strong primitives are not provided, we used a version of Mellor-Crummey and Scott's algorithm that is implemented with fetch-and-store and that does not ensure starvation-freedom [45]. Fetch-and-add, which is used in T. Anderson's algorithm, was simulated by a test-and-set algorithm with randomized backoff, as Anderson did in [9].

The experiments on the Symmetry show similar results to that for the TC2000. However, on the Symmetry, T. Anderson's algorithm has the best overall performance, mainly because the availability of coherent caches makes all spins in his algorithm local. The performance of Lamport's algorithm on the Symmetry is far better than that on the TC2000. This seems partly due to the fact that his algorithm is not starvation-free. Specifically, when a process enters its critical section, it can keep all needed variables in its own cache and repeatedly enter its critical section, without yielding to the other processes. In one of our tests for the two-process case, one process executed 50,000 critical sections during a period of time in which the other process executed only 120 critical sections.

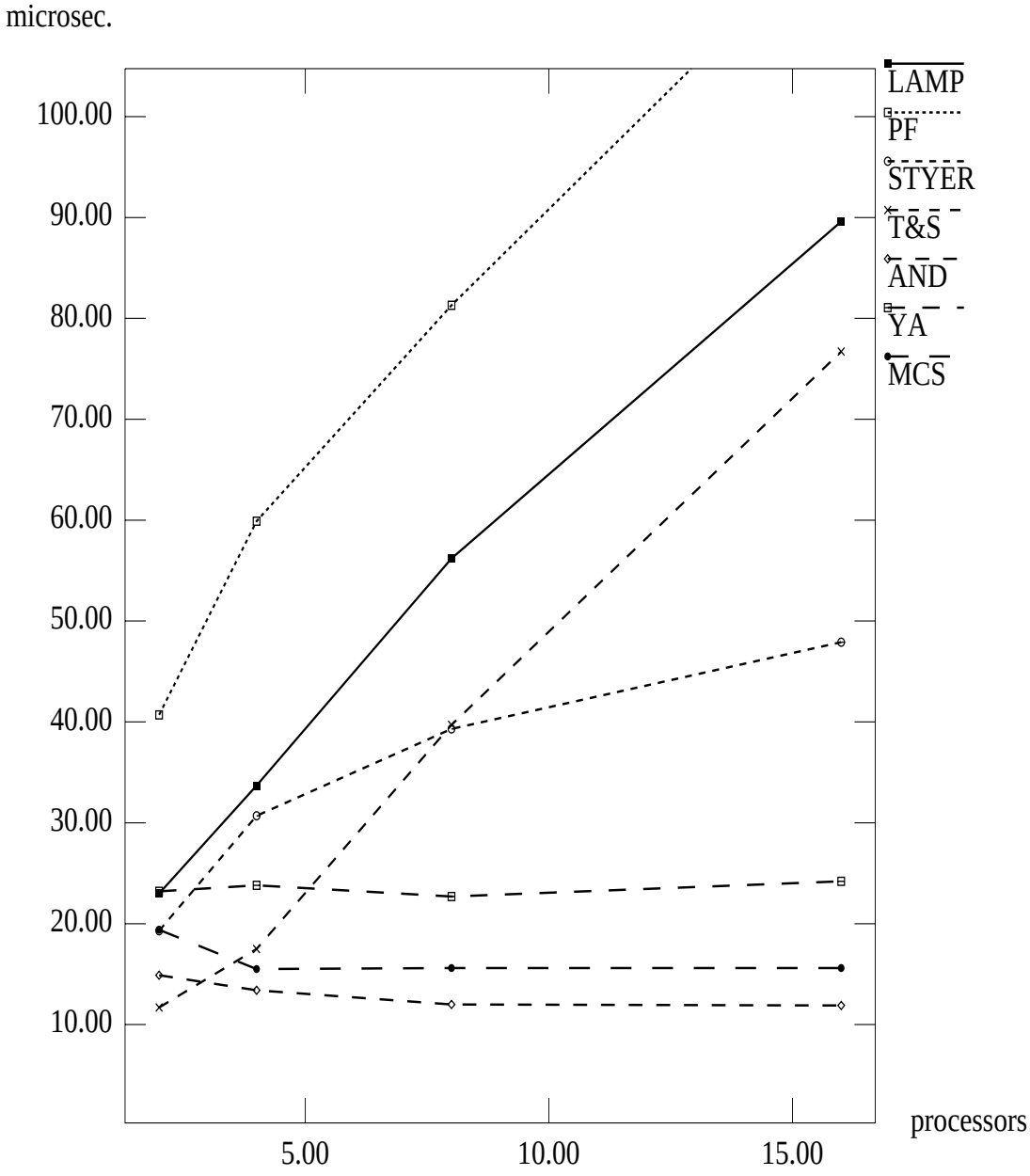

Figure 2.7: Performance results on the Symmetry.

Dependence on coherent caching for efficient synchronization [9, 26] is questionable, as many caching schemes do not cache shared writable data. Our solution neither requires a coherent cache for efficient implementation nor any strong primitives. An efficient implementation of our algorithm requires only that each processor has some part of shared memory that is locally accessible, and that read and write operations are atomic. We consider these to be minimal hardware requirements for efficient synchronization. It is worth noting that, without fetch-and-add and compare-and-swap primitives, T. Anderson's algorithm and Mellor-Crummey and Scott's algorithm are not starvation-free.

2.7 Discussion

We have presented a scalable mutual exclusion algorithm for shared memory multiprocessors that does not require any hardware support other than atomic read and write operations. Our algorithm has better worst-case time complexity than any previously published mutual exclusion algorithm based on read/write atomicity, requiring $O(\log N)$ remote operations under any amount of contention. We have also presented an extension of our algorithm for fast mutual exclusion in the absence of contention that generates $O(1)$ remote operations in the presence of contention and $O(N)$ in the absence of contention.

In the time complexity calculations given in this chapter, the distinction between remote and local operations is based upon a static assignment of shared variables to processes. Other definitions, which incorporate specific architectural details of systems, are also possible. For example, for programs intended for machines with coherent caching, it might be appropriate to consider a read of

a shared variable x by a process p to be local if x has not been written by another process since p's most recent access of x. However, because of the many parameters that go into defining a cache-coherence protocol, such definitions can be problematic. The next chapter provides a means to calculate the time complexity of concurrent programs for cache-coherent models.

A natural approach to measuring the time complexity of concurrent programs would be to simply count the total number of operations executed. However, a straightforward application of such an approach does not provide any insight into the behavior of mutual exclusion algorithms under heavy contention. In particular, in any algorithm in which processes busy-wait, the number of operations needed for one process to get to its critical section is unbounded. In order to serve as a measure of time complexity, a measure should be both intuitive and easy to compute. In sequential programming, the usual measure of time complexity, which is obtained by simply counting operations, satisfies these criteria. By contrast, there has been much disagreement on how time complexity should be measured in concurrent programs, and a complexity measure satisfying these criteria has yet to be adopted. We believe that an appropriate time complexity measure for concurrent algorithms is one based on the number of remote memory references. As seen in this chapter, such a measure can be used to make meaningful distinctions concerning the scalability of concurrent programs.

Chapter 3

Time/Contention Trade-offs for Multiprocessor Synchronization

3.1 Introduction

In this chapter, we consider bounds on time for mutual exclusion, a subject that has received scant attention in the literature. Past work on the complexity of mutual exclusion has almost exclusively focused on space requirements [17]; the limited work on time bounds that has been done has focused on partially synchronous models [43].

The lack of prior work on time bounds for mutual exclusion within asynchronous models is probably due to difficulties associated with measuring the time spent within busy-waiting constructs. In fact, because of such difficulties, there has been scarcely little work of any kind on time bounds for asynchronous concurrent programming problems for which busy-waiting is inherent. One of the primary contributions of this chapter is to show that it *is* possible to establish meaningful time bounds for such problems.

In Chapter 2, we proposed a time measure for concurrent programs that dis-

tinguishes between local and remote accesses of shared memory [60]. Under our measure, the time complexity of a concurrent program is measured by counting only remote accesses of shared variables; local accesses are ignored.

We present several lower-bound results for mutual exclusion that are based on the time complexity measure proposed in Chapter 2. Our results establish trade-offs between time complexity and write- and access-contention for solutions to the mutual exclusion problem. The *write-contention* (*access-contention*) of a concurrent program is the number of processes that may be simultaneously enabled to write (access) the same shared variable. Limiting access-contention is an important consideration when designing algorithms for problems, such as mutual exclusion and shared counting, that must cope well with high competition among processes [9, 28, 29, 53]. Performance problems associated with high access-contention can be partially alleviated by employing coherent caching techniques to reduce concurrent reads of the same memory location. However, even when such techniques are employed, limiting write-contention is still an important concern.

We show that, for any N-process mutual exclusion algorithm, if write-contention is w, and if each atomic operation accesses at most v remote variables, then there exists an execution involving only one process in which that process executes $\Omega(\log_{vw} N)$ remote operations for entry into its critical section. We further show that, among these operations, $\Omega(\sqrt{\log_{vw} N})$ distinct remote variables are accessed. For algorithms with access-contention c, we show that the latter bound can be improved to $\Omega(\log_{vc} N)$.

These results have a number of important implications. For example, because the first access of any variable causes a cache miss, the latter two bounds imply

that a time/contention trade-off exists even if coherent caching techniques are employed. Also, because the execution that establishes these bounds involves only one process, it follows that fast mutual exclusion algorithms require arbitrarily high write-contention in the worst case. These bounds apply not only to the mutual exclusion problem, but also to a class of decision problems that includes the leader-election problem.

In most shared-memory multiprocessors, an atomic operation may access only a constant number of remote variables. In fact, most commonly-available synchronization primitives access only one remote variable; examples include read, write, test-and-set, fetch-and-store, compare-and-swap, and fetch-and-add. If v is taken to be a constant, then our results imply that, for any N-process mutual exclusion algorithm with write-contention w, some process executes $\Omega(\log_w N)$ remote operations in the absence of competition for entry into its critical section. Further, among these remote operations, $\Omega(\sqrt{\log_w N})$ distinct remote variables are accessed. For algorithms with access-contention c, the latter bound is improved to $\Omega(\log_c N)$. It can be shown that the first and last of these bounds are asymptotically tight.

In the mutual exclusion algorithm depicted in Figure 2.3, only reads and writes are used, i.e., $v = 1$ holds. Note that this algorithm has access-contention (and hence write-contention) two. Thus, this algorithm gives us an upper bound matching the first (last) bound mentioned in the previous paragraph, for the class of algorithms with constant write-contention (access-contention). In Section 3.4, we present an algorithm that provides us upper bounds matching these lower bounds for arbitrary contention.

Related work includes previous research by Dwork et al. given in [23], where

it is shown that solving mutual exclusion with access-contention c requires $\Omega((\log_2 N)/c)$ memory references. Our work extends that of Dwork et al. in several directions. First, the implications concerning fast mutual exclusion and cache coherence noted above do not follow from their work. Second, we consider programs in which atomic operations may access multiple shared variables, whereas they only consider reads, writes, and read-modify-writes. Third, in our main result, we restrict only write-contention, and if v is a constant, then we obtain a tight bound of $\Omega(\log_w N)$, which exceeds the bound established by them. Finally, and most importantly, Dwork et al. make no distinction between local and remote shared memory accesses. Because busy-waiting is required for mutual exclusion in general, an unbounded number of memory accesses (local or remote) are required in the worst case. It is our belief that time complexity results that do not distinguish between local and remote accesses of shared memory are of questionable value as a measure of performance of mutual exclusion algorithms under contention.

The rest of the chapter is organized as follows. In Section 3.2, we present our model of shared memory systems. In Section 3.3, we define a simplified version of the mutual exclusion problem called the "minimal" mutual exclusion problem. The above-mentioned time bounds are then established in Sections 3.4 and 3.5. We end the chapter with some discussion in Section 3.6.

3.2 Shared-Memory Systems

Our model of a shared-memory system is similar to that given by Merritt and Taubenfeld in [46]. A *system* $S = (C, P, V)$ consists of a set of computations C,

a set of processes $P = \{1, 2, \ldots, N\}$, and a set of variables V. A *computation* is a finite sequence of events.

An *event* is denoted $[R, W, i]$, where $R = \{(x_j, u_j) | 1 \leq j \leq m\}$ for some m, $W = \{(y_k, v_k) | 1 \leq k \leq n\}$ for some n, and $i \in P$; this notation represents reading value u_j from variable x_j, for $1 \leq j \leq m$, and writing value v_k to variable y_k, for $1 \leq k \leq n$. Each variable in R (W) is assumed to be distinct. We say that this event *accesses* each such x_j and y_k. We use $R.var$ to denote the set of variables x_j such that $(x_j, u_j) \in R$ for some u_j, and $W.var$ to denote the set of variables y_k such that $(y_k, v_k) \in W$ for some v_k.

Each variable is *local* to at most one process and is *remote* to all other processes. (Note that we allow variables that are remote to *all* processes.) An *initial value*, denoted x_{init}, is associated with each variable x. An event is *local* if it does not access any remote variable, and is *remote* otherwise.

We use $\langle e, \ldots \rangle$ to denote a computation that begins with the event e, and $\langle\rangle$ to denote the empty computation. We define the *length* of computation H, denoted $|H|$, as the number of events in H. $H \circ G$ denotes the computation obtained by concatenating computations H and G. If G is a subsequence of H, then $H - G$ is the computation obtained by removing all events in G from H. The value of variable x at the end of computation H, denoted $value(x, H)$, is the last value that is written to x in H (or x_{init} if x is not written in H). More formally, we define $value(x, H)$ recursively as follows.

$value(x, H) \equiv$

if $H = \langle\rangle$ **then** x_{init}

else if $H = G \circ \langle [R, W, i] \rangle$ **then**

if $(x, v) \in W$ **then** v

else $value(x, G)$ **fi fi fi**

Similarly, we define the last event to write variable x in H, denoted $writer(x, H)$, as follows. If x is not written by any event in H, then we let $writer(x, H) = \perp$.

$writer(x, H) \equiv$
  if $H = \langle\rangle$ **then** $\perp$
  else if $H = G \circ \langle[R, W, i]\rangle$ **then**
    if $(x, v) \in W$ **then** $[R, W, i]$
    else $writer(x, G)$ **fi fi fi**

An *extension* of computation H is a computation of which H is a prefix. For a computation H and a set of processes Y, H_Y denotes the subsequence of H that contains all events in H of processes in Y.

Computations H and G are *equivalent* with respect to a set of processes Y, denoted $H[Y]G$, iff $H_Y = G_Y$. Note that $[Y]$ is an equivalence relation. We now present our model of shared-memory systems.

Definition: A *shared-memory system* $S = (C, P, V)$ is a system that satisfies the following properties.

- (P1) If $H \in C$ and G is a prefix of H, then $G \in C$.
- (P2) If $H \circ \langle[R, W, i]\rangle \in C$, $G \in C$, $G[Y]H$, and $i \in Y$, and if for all $x \in R.var$, $value(x, G) = value(x, H)$ holds, then $G \circ \langle[R, W, i]\rangle \in C$.
- (P3) If $H \circ \langle[R, W, i]\rangle \in C$, $G \in C$, $G[Y]H$, and $i \in Y$, then $G \circ \langle[R', W', i]\rangle \in C$ for some R' and W' such that $R'.var = R.var$ and $W'.var = W.var$.

- (P4) For any $H \in C$, $H \circ \langle [R, W, i] \rangle \in C$ only if for all $(x, v) \in R$, $v = value(x, H)$ holds. □

For simplicity, we call a remote event a *remote read* if it reads a remote variable, and a *remote write* if it writes remote variables. Note that a remote event can be both a remote read and a remote write.

Consider a shared-memory system $S = (C, P, V)$. A computation H is a Y-computation iff either $H = \langle\rangle$ and $Y \subseteq P$, or Y is the minimal subset of P such that $H = H_Y$ holds. For simplicity, we abbreviate the preceding definitions when applied to a singleton set of processes. For example, if $Y = \{i\}$, then we use H_i to mean $H_{\{i\}}$, i-computation to mean $\{i\}$-computation, and $[i]$ to mean $[\{i\}]$.

In the following sections, we establish time bounds involving various notions of contention. Consider a shared-memory system $S = (C, P, V)$. The strictest notion of contention is static in nature. In particular, consider a variable x in V. A process i in P is a *reader* (*writer*) of x iff there is an event of i that reads (writes) x in some computation in C. We say that x is a *k-reader* (*k-writer*) *variable* iff there are k readers (writers) of x. The other two notions of contention that we employ are dynamic in nature. For $H \in C$ and $x \in V$, let $overwriters(x, H) \equiv \{i \mid H \circ \langle [R, W, i] \rangle \in C \text{ where } x \in W.var\}$. Then, the *write-contention* of S is $\mathbf{max}_{x \in V, H \in C}(|overwriters(x, H)|)$. Similarly, let $contenders(x, H) \equiv \{i \mid H \circ \langle [R, W, i] \rangle \in C \text{ where } x \in (R.var \cup W.var)\}$. Then, the *access-contention* of S is $\mathbf{max}_{x \in V, H \in C}(|contenders(x, H)|)$. These notions of contention bound the number of processes that may simultaneously write (access) the same memory location.

3.3 Minimal Mutual Exclusion

Our main results concerning the mutual exclusion problem are based on a simplified version of the problem, which we call the "minimal mutual exclusion problem".

Minimal Mutual Exclusion Problem: We define the minimal mutual exclusion problem for a shared-memory system $S = (C, P, V)$ as follows. Each process $i \in P$ has a local variable $i.dine$ that ranges over $\{think, hungry, eat\}$. Variable $i.dine$ is initially *think* and is accessed only by the following events:

$$\begin{aligned} Think_i &\equiv [\{\}, \{(i.dine, think)\}, i] \\ Hungry_i &\equiv [\{\}, \{(i.dine, hungry)\}, i] \\ Eat_i &\equiv [\{\}, \{(i.dine, eat)\}, i] \end{aligned}$$

The allowable transitions of $i.dine$ are as follows: for any $H \in C$, $H \circ \langle Think_i \rangle \in C$ iff $value(i.dine, H) = eat$; $H \circ \langle Hungry_i \rangle \in C$ iff $value(i.dine, H) = think$; and if $H \circ \langle Eat_i \rangle \in C$, then $value(i.dine, H) = hungry$. System S solves the minimal mutual exclusion problem iff the following requirements are satisfied.

- *Exclusion*: For any $H \in C$ and processes $i \neq j$, $value(i.dine, H) = eat \Rightarrow value(j.dine, H) \neq eat$.
- *Progress*: For any $H \in C$ and process $i \in P$, if H is an i-computation, then either H contains Eat_i, or there exists an i-computation G such that $H \circ G \circ \langle Eat_i \rangle \in C$. □

Note that the Progress requirement above is much weaker than that usually specified for the mutual exclusion problem. (This, of course, strengthens our

impossibility proofs.) Note also that any solution to the leader election problem easily solves the minimal mutual exclusion problem. Thus, our time bounds apply not only to the mutual exclusion problem, but also to the leader election problem, and any other decision problem that can be used to directly solve leader election.[1]

Before presenting our main results, we give bounds for the case of statically-defined contention. In this theorem and those that follow, we assume that S is a shared-memory system and that $i \in P$.

Theorem 3.1: For any $S = (C, P, V)$ that solves the minimal mutual exclusion problem, if each event accesses at most v remote variables, and if either all variables in V are k-reader variables, or all variables in V are k-writer variables, then there exists an i-computation in C that contains $\Omega(N/vk)$ remote events but no Eat_i event.

Proof: Suppose that all variables in V are k-reader variables. (A similar argument applies if all variables are k-writer variables.) By the Progress requirement of the minimal mutual exclusion problem, there exists an i-computation $H(i) \cdot Eat_i$ in C for each $i \in P$ such that $H(i)$ does not contain Eat_i. Let $C' = \{H(i) \mid i \in P\}$.

It can be shown that for each i and j such that $i \neq j$, $H(i)$ contains a write of a variable that is read in $H(j)$. (Otherwise, we could show that $H(i) \circ H(j) \circ Eat_i \circ Eat_j$ is a computation in C, violating the Exclusion requirement.) Select

[1]For example, the *ranking* problem. In this problem, each process is assigned a "rank" between 1 and N. The process that obtains a rank of 1 can be defined to be the "leader".

one such variable for each pair (i, j) where $i \neq j$. Let V' be the set of the variables selected.

Because each variable is a k-reader variable, $H(i)$ contains writes of at least $\lceil (N-1)/k \rceil$ variables in V'. If there exists $i \in P$ such that $\lceil (N-1)/2k \rceil$ such variables are remote to i, then the theorem easily follows. So, assume that each process $i \in P$ has at least $\lceil (N-1)/2k \rceil$ such variables, denoted as L_i, as local variables.

Observe that $L_i \subseteq V'$ and, because the variables in L_i are local to i, $L_i \cap L_j = \{\}$ holds for any $i \neq j$. By the construction of V', for each $x \in L_i$, there exists $H(j)$ in C' that contains a remote event reading x, where $j \neq i$. Thus, there exists a set of remote events in C' that collectively read at least $\lceil (N-1)/2k \rceil$ variables in L_i (remotely). Thus, there exists a set of remote events in C' that collectively read at least $\lceil N(N-1)/2k \rceil$ variables in V' (remotely). If each event accesses at most v remote variables, then by the pigeon-hole principle, there exists an i-computation in C' that contains at least $\lceil (N-1)/2vk \rceil$ remote events. $\Box$

For any N-process system S that satisfies the conditions of Theorem 3.1, some process i executes $\Omega(N/vk)$ remote events in the absence of competition. If we remove process i from system S, we obtain a system that satisfies the conditions of the theorem with N replaced by $N-1$. Thus, there is a process $j \neq i$ in system S that executes $\Omega((N-1)/vk)$ remote events in the absence of competition. Continuing in this manner, at least half the processes in S execute at least $\Omega(N/2vk)$ remote events in the absence of competition. Thus, we have the following corollary.

Corollary 3.1: For any system S satisfying the conditions of Theorem 3.1, there exist $\Omega(N)$ processes i in P for which the conclusion of the theorem holds.

□

Similar corollaries apply to the theorems in the following sections.

In [6], a mutual exclusion algorithm requiring $O(N)$ remote memory references per critical section acquisition is given that employs only single-reader, single-writer variables. Thus, if v and k are taken to be positive constants, then the bound of Theorem 3.1 is asymptotically tight. In the remainder of the paper, we consider more interesting bounds based on dynamic notions of contention.

3.4 Main Result: Bounding Remote Events

In this section, we show that for any system with write-contention w, if an event may access at most v remote variables, then $\Omega(\log_{vw} N)$ remote events are required in the absence of competition to solve the minimal mutual exclusion problem.

This bound has important consequences for distributed shared-memory multiprocessing systems. On such systems, remote events require a traversal of a global interconnection network and hence are more expensive than local events. Thus, for such machines, the lower bound of Theorem 3.3 below not only gives the inherent time complexity of the problem, it also bounds the communication complexity measured in terms of global traffic.

We begin by presenting several lemmas that are needed to prove the main theorem. The first lemma directly follows from the definitions of $value(x, H)$

and $writer(x, H)$.

Lemma 3.1. $writer(x, H) = writer(x, G) \Rightarrow value(x, H) = value(x, G)$. □

Theorem 3.3 is proved by considering a class of computations, as defined by a set of conditions. Each of these conditions refers to an arbitrary computation H in this class. The first condition is as follows.

- (C1) For events $[R, U, i]$ and $[T, W, j]$ in H, if $(R.var \cap W.var) \neq \{\}$ holds and $[T, W, j]$ precedes[2] $[R, U, i]$ in H, then $i = j$. Informally, no process reads a variable that is accessed by a preceding write of another process in H.

We will use this condition and those that follow to inductively construct longer and longer computations. Condition (C1) eliminates "information flow" between processes in the computations so constructed.

The next lemma gives us a means for projecting a computation onto a set of processes so that the resulting projection is itself a computation.

Lemma 3.2: For any $S = (C, P, V)$, if $G \circ H$ is a computation in C satisfying (C1), then for any $Y \subseteq P$, $G \circ H_Y \in C$.

Proof: We prove that $G \circ H_Y \in C$ by induction on the length of H_Y.

Induction Base. Because $G \circ H \in C$ holds, by (P1), $G \in C$ holds.

[2] Although our definition of an event allows multiple instances of the same event, we assume that such instances are distinguishable from each other. (For simplicity, we do not extend our notion of an event to include an additional identifier for distinguishability.)

Induction Hypothesis. Suppose that Lemma 3.2 holds for H_Y if $|H_Y| = m$.

Induction Step. We now consider H_Y of length $m+1$. Let $H_Y = \langle e_0, e_1, \ldots, e_{m-1}, e_m \rangle$. Let $H = H' \circ \langle e_m \rangle \circ H''$.

By (P1), $G \circ H' \in C$. Observe that $G \circ H'_Y = G \circ \langle e_0, e_1, \ldots, e_{m-1} \rangle$. Hence, by the induction hypothesis, $G \circ H'_Y \in C$. Next, we prove $G \circ H_Y \in C$ by considering two cases. Let $e_m = [R, W, i]$ for some $i \in Y$.

Because $G \circ H' \circ \langle e_m \rangle$ is a prefix of $G \circ H$, by (P1), $G \circ H' \circ \langle e_m \rangle \in C$. Note that $G \circ H'$ $[Y]$ $G \circ H'_Y$. Thus, to prove that $G \circ H_Y \in C$, it suffices to prove that, for any $x \in R.var$, $value(x, G \circ H') = value(x, G \circ H'_Y)$. In particular, if the latter holds, then (P2) implies that $G \circ H_Y = G \circ H'_Y \circ \langle e_m \rangle \in C$ also holds. If $R = \{\}$, then this remaining proof obligation is vacuous, so in the remainder of the proof, assume that $R \neq \{\}$.

We consider two cases according to whether x is written in $G \circ H'$. If $writer(x, G \circ H') = \perp$, then $writer(x, G \circ H'_Y) = \perp$, and by Lemma 3.1, $value(x, G \circ H') = value(x, G \circ H'_Y)$. If $writer(x, G \circ H') = [L, U, j]$, then because $G \circ H$ satisfies (C1), $j = i$. It follows that $writer(x, G \circ H'_Y) = [L, U, j]$, and by Lemma 3.1, $value(x, G \circ H') = value(x, G \circ H'_Y)$. This concludes the proof of Lemma 3.2. □

Before presenting the remaining lemmas, we state the remaining three conditions that serve to characterize the class of computations considered in the main theorem. Recall that in these conditions, H denotes an arbitrary computation from the class to be considered.

The first of these conditions refers to "active" processes. If $H = \langle\rangle$ or $H_i \neq \langle\rangle$,

then process i is *active* in H; otherwise i is *inactive* in H. The notion of an active process will arise in subsequent inductive proofs. Initially, all processes are active; in a non-null computation, only those processes that have taken steps are active.

- (C2) For any event $[R, W, i]$ in H, if $x \in (R.var \cup W.var)$, and if x is local to a process j that is active in H, then $i = j$. Informally, no local variable of an active process is accessed by other processes in H.

- (C3) For any events $[R, W, i]$ and $[T, U, j]$ in H, if $(W.var \cap U.var) \neq \{\}$, then $i = j$. Informally, each variable is written by at most one process in H.

- (C4) For any prefix G of H, $value(i.dine, G) \neq eat$. Informally, no process eats in H.

By (C2), "information flow" between processes can only occur through remote events in the computations we inductively construct. Condition (C3) makes it easier for us to make an active process inactive, i.e., remove its events from a given computation. In particular, because each variable is written by at most one process, if a process is made inactive, then the variables it writes simply take on their initial values. Condition (C4) arises because we intend to compute the time complexity required for a process to eat for the first time.

The next two lemmas give us means for extending a computation. We will usually use these lemmas to extend a computation by appending local events.

Lemma 3.3: Consider $S = (C, P, V)$. Let F, G, and H be computations such that for some $i \in P$, F is an i-computation, no event in F accesses a variable

that is written by processes other than i in either G or H, $H \in C$, and $G[i]H$. If $G \circ F \in C$, then $H \circ F \in C$.

Proof: As in the statement of the lemma, assume the following: (i) F is an i-computation; (ii) no event in F accesses a variable that is written by processes other than i in either G or H; (iii) $H \in C$; (iv) $G[i]H$; and (v) $G \circ F \in C$. We prove that $H \circ F \in C$ by induction on the length of F.

Induction Base. If $|F| = 0$, then $H \circ F = H$. By assumption (iii), $H \in C$ holds.

Induction Hypothesis. Suppose that Lemma 3.3 holds when $|F| = m$.

Induction Step. We now consider F of length $m+1$. Let $F = \langle e_0, e_1, \ldots, e_m \rangle$. We use (P2) to prove that $H \circ \langle e_0, e_1, \ldots, e_m \rangle \in C$. By assumption (v),

$$G \circ \langle e_0, e_1, \ldots, e_{m-1} \rangle \circ \langle e_m \rangle \in C \quad . \tag{3.1}$$

By (P1), (3.1) implies that $G \circ \langle e_0, e_1, \ldots, e_{m-1} \rangle \in C$ holds. Thus, by the induction hypothesis, we have the following.

$$H \circ \langle e_0, e_1, \ldots, e_{m-1} \rangle \in C \tag{3.2}$$

By assumption (iv), $G[i]H$ holds, so the following holds.

$$G \circ \langle e_0, e_1, \ldots e_{m-1} \rangle \; [i] \; H \circ \langle e_0, e_1, \ldots e_{m-1} \rangle \tag{3.3}$$

Let $e_m = [R, W, i]$. By assumption (ii), $[R, W, i]$ does not access a variable that is written by processes other than i in either G or H. Thus, each x in $R.var$ is not written by other processes in either G or H. Thus, $G[i]H$ implies that $writer(x, G) = writer(x, H)$, which implies that $writer(x, G \circ \langle e_0, e_1, \ldots e_{m-1} \rangle) = writer(x, H \circ \langle e_0, e_1, \ldots e_{m-1} \rangle)$. By Lemma 3.1, this implies that the following holds.

$$value(x, G \circ \langle e_0, e_1, \ldots e_{m-1} \rangle) = value(x, H \circ \langle e_0, e_1, \ldots e_{m-1} \rangle) \tag{3.4}$$

Thus, by (3.1), (3.2), (3.3), (3.4), and (P2), we have $H \circ \langle e_0, e_1, \ldots, e_m \rangle \in C$. $\Box$

Lemma 3.4: Consider $S = (C, P, V)$ and $Q \subseteq P$, where every process in Q is active in H. Without loss of generality, assume that the processes are numbered so that $Q = \{1, 2, \ldots, |Q|\}$. Let H and $L(j)$, $1 \leq j \leq |Q|$, be computations satisfying the following conditions: $L(j)$ is a j-computation; $H \circ L(j) \in C$; and no event in $L(j)$ accesses any variable that is accessed by other processes in $H \circ L(1) \circ L(2) \circ \cdots \circ L(|Q|)$. Then, $H \circ L(1) \circ L(2) \circ \cdots \circ L(|Q|) \in C$.

Proof: As in the statement of the lemma, we have the following: (i) $L(j)$ is a j-computation; (ii) $H \circ L(j) \in C$; and (iii) no event in $L(j)$ accesses any variable that is accessed by other processes in $H \circ L(1) \circ L(2) \circ \cdots \circ L(|Q|)$. We prove that $H \circ L(1) \circ L(2) \circ \cdots \circ L(|Q|) \in C$ by induction on $|Q|$.

Induction Base. By (P1) and assumption (ii), $H \in C$.

Induction Hypothesis. Assume that $H \circ L(1) \circ L(2) \circ \cdots \circ L(j-1) \in C$, where $1 \leq j \leq |Q|$.

Induction Step. We use Lemma 3.3 to prove that $H \circ L(1) \circ L(2) \circ \cdots \circ L(j) \in C$. By assumption (i),

$$H \ [j] \ H \circ L(1) \circ L(2) \circ \cdots \circ L(j-1) \quad . \tag{3.5}$$

By (3.5), the induction hypothesis, and assumptions (i), (ii), and (ii), Lemma 3.3 implies that $H \circ L(1) \circ L(2) \circ \cdots \circ L(j) \in C$ holds. $\Box$

According to the next lemma, if n processes are competing for entry into their critical sections, and if each of these n processes has no knowledge of the

others, then at least $n-1$ of the processes has at least one more remote event to execute. To formally capture the latter, consider a system $S = (C, P, V)$ that solves the minimal mutual exclusion problem and let $i \in P$ and $H \in C$. We say that i *has a remote event after* H iff there exists an i-computation M such that M does not contain Eat_i, M has a remote event, and $H \circ M \in C$.

Lemma 3.5: Suppose that $S = (C, P, V)$ solves the minimal mutual exclusion problem. Let $Y \subseteq P$ be a set of n processes, and let H be a Y-computation in C satisfying (C1), (C2), and (C4). Then, at least $n-1$ processes in Y have a remote event after H.

Proof: Assume to the contrary that $\{i, j\} \subseteq Y$ have no remote event after H. Because H satisfies (C1), by Lemma 3.2, $H_i \in C$. Also, because H satisfies (C4), H_i satisfies (C4). Hence, because S satisfies the Progress requirement, there exists an i-computation G such that $H_i \circ G \circ \langle Eat_i \rangle \in C$, and G does not contain Eat_i. Similarly, there exists a j-computation G' such that $H_j \circ G' \circ \langle Eat_j \rangle \in C$, and G' does not contain Eat_j. We consider three cases.

Case 1. G contains a remote event. Let $G = F \circ \langle [R, W, i], \ldots \rangle$, where $[R, W, i]$ is the first remote event in G. We prove that i has a remote event after H, which is a contradiction to our assumption. In particular, we use (P3) to prove that $H \circ F \circ \langle [R', W', i] \rangle \in C$, where $R'.var = R.var$ and $W'.var = W.var$. Because $H_i \circ G \circ \langle Eat_i \rangle \in C$ holds, by (P1), we have the following.

$$H_i \circ F \circ \langle [R, W, i] \rangle \in C \tag{3.6}$$

We now use Lemma 3.3 to prove that $H \circ F \in C$. The following assertions are

used in applying Lemma 3.3.

$$H \in C \tag{3.7}$$

$$H_i \ [i] \ H \tag{3.8}$$

$$H_i \circ F \in C \tag{3.9}$$

(3.7) holds by the definition of H, (3.8) holds by the definition of $[i]$, and (3.9) follows from (3.6) and (P1). Observe that F is an i-computation consisting of local events. Thus, because i is active in H and H_i, and because both H and H_i satisfy (C2), no event in F accesses a variable that is written by processes other than i in either H or H_i. Hence, by (3.7), (3.8), (3.9), and Lemma 3.3, the following holds.

$$H \circ F \in C \tag{3.10}$$

Observe that (3.8) implies that the following holds.

$$H \circ F \ [i] \ H_i \circ F \tag{3.11}$$

By (3.6), (3.10), (3.11), and (P3), $H \circ F \circ \langle [R', W', i] \rangle \in C$, where $R'.var = R.var$ and $W'.var = W.var$. Because $H \circ F \circ \langle [R', W', i] \rangle \in C$, i has a remote event after H, which is a contradiction.

Case 2. G' contains a remote event. We can prove that j has a remote event after H. The proof is similar to that of Case 1, and hence is omitted.

Case 3. G and G' do not contain any remote event. We prove that S does not solve the minimal mutual exclusion problem.

We first use Lemma 3.3 to prove that $H \circ G \circ \langle Eat_i \rangle \in C$ holds. By assumption, we have the following.

$$H_i \circ G \circ \langle Eat_i \rangle \in C \tag{3.12}$$

Observe that $G \circ \langle Eat_i \rangle$ is an i-computation consisting of local events. Thus, because i is active in H and H_i, and because H and H_i both satisfy (C2), no event in $G \circ \langle Eat_i \rangle$ accesses a variable that is written by processes other than i in either H or H_i. Hence, by (3.7), (3.8), (3.12), and Lemma 3.3, $H \circ G \circ \langle Eat_i \rangle \in C$. Similarly, $H \circ G' \circ \langle Eat_j \rangle \in C$.

Let $F = H \circ G \circ \langle Eat_i \rangle \circ G' \circ \langle Eat_j \rangle$. It is straightforward to use Lemma 3.4 to prove that $F \in C$. (Let $L(1) = G \circ \langle Eat_i \rangle$ and let $L(2) = G' \circ \langle Eat_j \rangle$.) Note that $value(i.dine, F) = eat \ \wedge \ value(j.dine, F) = eat$ holds, which implies that S does not solve the minimal mutual exclusion problem. □

The next theorem by Turán [58] will be used in subsequent lemmas.

Theorem 3.2 (Turán): Let $G = \langle V, E \rangle$ be an undirected multigraph,[3] where V is a set of vertices and E is a set of edges. If the average degree is d, then there exists an independent set[4] with at least $\lceil |V|/(d+1) \rceil$ vertices. □

Our next lemma provides the induction step that leads to the lower bound in Theorem 3.3.

Lemma 3.6: Let $S = (C, P, V)$ be a shared-memory system with write-contention w that solves the minimal mutual exclusion problem. Let $Y \subseteq P$ be a set of n processes, and let H be a Y-computation in C satisfying (C1), (C2), (C3), and

[3] A *multigraph* is a graph in which multiple edges are allowed between any two vertices. For brevity, we will henceforth use "graph" to mean an undirected multigraph.

[4] An *independent set* of a graph $G = \langle V, E \rangle$ is a subset $V' \subseteq V$ of vertices such that no edge in E is incident to two vertices in V'.

(C4) such that each process in Y executes r remote events in H. Suppose that each event accesses at most v remote variables. Then, there exist $Z \subseteq Y$, where $|Z| = \lceil (n-1)/(2v+1)^2 vw \rceil$, and a Z-computation G in C satisfying (C1), (C2), (C3), and (C4) such that each process in Z executes $r+1$ remote events in G.

Proof: The proof strategy is as follows. We show that there exists $Z \subseteq Y$ that can execute another remote event without violating any of the conditions (C1) through (C4). We "eliminate" processes not in Z, i.e., ones that may violate some condition. Finally, we construct a Z-computation G that satisfies (C1), (C2), (C3), and (C4).

Lemma 3.5 implies that there exists $Y1 \subseteq Y$, where $|Y1| \geq n-1$, such that the following holds: for any $i \in Y1$, there exists an i-computation $B(i)$ such that $H \circ B(i) \in C$, $B(i)$ does not contain Eat_i, and $B(i)$ has at least one remote event. For $i \in Y1$, let $B(i) = L(i) \circ \langle [R_i, W_i, i], \ldots \rangle$ where $[R_i, W_i, i]$ is the first remote event in $B(i)$. Note that, by (P1), the following holds.

$$H \circ L(i) \circ \langle [R_i, W_i, i] \rangle \in C \tag{3.13}$$

We construct $Y2$, a subset of $Y1$, as follows. First, select a process $i \in Y1$. Let $X = \{x \mid x \in W_i.var \text{ and } x \text{ is remote to } i\}$, i.e., X is the set of remote variables written by the event $[R_i, W_i, i]$. By assumption, $|X| \leq v$. Let $Q_X = \{j \mid j \in Y1 \ \wedge \ j \neq i \ \wedge \ (W_j.var \cap X) \neq \{\}\}$, i.e., Q_X includes those processes other than i that write variables in X. Because write-contention is w, it is straightforward to use Lemma 3.4 to show that $|Q_X| \leq v(w-1)$. Delete i and all processes in Q_X from $Y1$, and add i to $Y2$. Repeat the above procedure until $Y1$ becomes empty. It follows, by construction, that

$$|Y2| \geq \lceil (n-1)/vw \rceil \quad . \tag{3.14}$$

Now, we identify any possible "information flow" between the events $\{[R_i, W_i, i] \mid i \in Y2\}$ and the events of processes in $Y2$ in H. Recall that $\{[R_i, W_i, i] \mid i \in Y2\}$ contains events that can be applied after H. We construct a graph $\langle Y2, E\rangle$ as follows. We do not distinguish a vertex representing p from the process p when this does not cause any confusion. Informally, an edge joining two processes represents possible information flow between the two processes. Our proof strategy is to prohibit information flow between active processes. Suppose that $x \in R_p.var \cup W_p.var$ and x is remote to p. Without loss of generality, we assume x is local to q for some $q \neq p$. Note that q may or may not be a member of $Y2$. We construct E by the following rules.

- (R1): If $q \in Y2$, then introduce an edge (p, q).
- (R2): If there is process $w \in Y2$ that writes to x in H, where $w \neq p \wedge w \neq q$, then introduce an edge (p, w). Note that, because H satisfies (C2), $q \notin Y2$ holds.

Consider the event $[R_i, W_i, i]$, where $i \in Y2$. Because (R1) and (R2) are exclusive, at most one edge is introduced for each remote variable this event accesses. Therefore, because each event accesses at most v remote variables, at most v edges are introduced by this event in total. It follows that the average degree in $\langle Y2, E\rangle$ is at most $2v$. By Theorem 3.2 and (3.14), this implies that there exists a subgraph $\langle Y3, \{\}\rangle$ of $\langle Y2, E\rangle$, where

$$|Y3| \geq \lceil (n-1)/(2v+1)vw \rceil \quad . \tag{3.15}$$

Without loss of generality, assume the processes are numbered so that $Y3 = \{1, 2, \ldots, |Y3|\}$. Consider the following computation.

$$H' \;=\; H_{Y3} \circ L(1) \circ L(2) \circ \cdots \circ L(|Y3|) \circ \langle [R_1, W_1, 1], [R_2, W_2, 2], \ldots,$$

$[R_{|Y3|}, W_{|Y3|}, |Y3|]\rangle$

We will use H' to construct the computation G mentioned at the beginning of the proof. In order to motivate the construction of G, we first prove that H' satisfies conditions (C2) through (C4). We consider each of these conditions as a separate case. In these cases, we make use of the fact that, because H satisfies (C2) through (C4), H_{Y3} also satisfies (C2) through (C4).

Condition $(C4)$. By construction, $L(i)$ does not contain Eat_i, and $[R_i, W_i, i] \neq Eat_i$. Hence, H' satisfies (C4). □

Condition $(C3)$. H_{Y3} satisfies (C2) and (C3), and each $L(i)$ consists only of local events, so $H_{Y3} \circ L(1) \circ L(2) \circ \cdots \circ L(|Y3|)$ satisfies (C3). Hence, to complete the proof that H' satisfies (C3), it suffices to prove that for each distinct i and j in $Y3$, $[R_i, W_i, i]$ does not write a variable that is written by $[R_j, W_j, j]$ or by any event of process j in H_{Y3} or $L(j)$.

By (R1), $[R_i, W_i, i]$ does not access a variable that is local to process j. Hence, $[R_i, W_i, i]$ does not write a variable that is locally written by process j in H_{Y3} or any variable that is written by j in $L(j)$. By (R2), $[R_i, W_i, i]$ does not access a variable that is remotely written by j in H. Hence, $[R_i, W_i, i]$ does not write a variable that is remotely written by j in H_{Y3}. By the definition of $Y3$ (specifically, the construction of $Y2$), the remote variables written by $[R_i, W_i, i]$ and $[R_j, W_j, j]$ are distinct. Hence, $[R_i, W_i, i]$ does not write a variable that is written by $[R_j, W_j, j]$. Hence, we conclude that H' satisfies (C3). □

Condition $(C2)$. No $L(i)$ accesses a remote variable, and hence, $H_{Y3} \circ L(1) \circ L(2) \circ \cdots \circ L(|Y3|)$ satisfies (C2). By (R1), no $[R_i, W_i, i]$ accesses a variable that is local to another process in $Y3$. Hence, H' satisfies (C2). □

The above reasoning leaves only condition (C1). We now show that H' may violate this condition. By (R1) and (R2), for each $j \neq i$, $[R_i, W_i, i]$ does not read a variable that is written by any event of process j in H_{Y3} or $L(j)$. Note, however, that $[R_i, W_i, i]$ may read a variable that is written by $[R_j, W_j, j]$. Such conflicts are the only way that H' may violate (C1). We now apply another graph argument in order to eliminate such conflicts among the events $\{[R_i, W_i, i] \mid i \in Y3\}$. Suppose that $x \in R_p.var$ and x is remote to p. Then, we construct a graph $\langle Y3, E' \rangle$, where the edges in E' are defined according to the following rule.

- (R3): If there is process $w \neq p$ such that $x \in W_w.var$ and $w \in Y3$, then introduce an edge (p, w).

Because H' satisfies (C3), p introduces at most one edge for each remote variable it reads. Because each event reads at most v remote variables, p introduces at most v edges in total. Thus, by Theorem 3.2 and (3.15), there exists a subgraph $\langle Z, \{\} \rangle$ of $\langle Y3, E' \rangle$, where

$$|Z| \geq \lceil (n-1)/(2v+1)^2 vw \rceil \quad . \tag{3.16}$$

The set Z represents the subset of the original n processes in Y that can execute another remote event without violating any of the conditions (C1) through (C4). We show this below.

Without loss of generality, assume the processes are numbered so that $Z = \{1, 2, \ldots, |Z|\}$. The computation G we seek is defined as follows.

$$G = H_Z \circ L(1) \circ L(2) \circ \cdots \circ L(|Z|) \circ \langle [R_1, W_1, 1], [R_2, W_2, 2], \ldots, [R_{|Z|}, W_{|Z|}, |Z|] \rangle$$

Observe that, because H' satisfies (C2) through (C4), G also satisfies (C2) through (C4). We now show that G satisfies (C1).

Condition $(C1)$. Because H satisfies (C1) and (C2), H_Z satisfies (C1) and (C2). Hence, because each $L(i)$ consists only of local events, $H_Z \circ L(1) \circ L(2) \circ \cdots \circ L(|Z|)$ satisfies (C1). Let p be any process in Z. To complete the proof that G satisfies (C1), it suffices to prove that no variable x in $R_p.var$ is written in G by a process other than p.

We first show that x is not written by processes other than p in $H_Z \circ L(1) \circ L(2) \circ \cdots \circ L(|Z|)$. By (R1) and (R2), $[R_p, W_p, p]$ does not access a variable that is written in H by other processes in Z. This implies that x is not written by processes other than p in H_Z. (R1) implies that $[R_p, W_p, p]$ does not access a variable that is local to another process in Z. Because each $L(i)$ consists of only local events, this implies that x is not written by processes other than p in $H_Z \circ L(1) \circ L(2) \circ \cdots \circ L(|Z|)$. Furthermore, by (R3), x is not written by $[R_j, W_j, j]$, where $j \neq p$. Hence, we conclude that G satisfies (C1). □

To complete the proof of the lemma, we need to show that G is actually a computation in C. This is established in the following claim.

Claim 3.1. $G \in C$.

Proof: The proof is by induction on the subsequence $\langle [R_1, W_1, 1], \ldots, [R_{|Z|}, W_{|Z|}, |Z|] \rangle$.

Induction Base. We use Lemmas 3.2, 3.3, and 3.4 to establish the base case. Because H satisfies (C1), by Lemma 3.2, $H_Z \in C$. Consider $j \in Z$. By (3.13) and (P1), $H \cdot L(j) \in C$. Because $j \in Z$, $H[j]H_Z$. Because H and H_Z satisfy (C2), and because $L(j)$ consists only of local events, no event in $L(j)$ accesses any variable accessed by processes other than j in H or H_Z. By Lemma 3.3, this implies

that $H_Z \cdot L(j) \in C$.

As above, because G satisfies condition (C2), no event in $L(j)$ accesses any variable accessed by another process in $H_Z \circ L(1) \circ L(2) \circ \cdots \circ L(|Z|)$. By Lemma 3.4, it follows that $H_Z \circ L(1) \circ L(2) \circ \cdots \circ L(|Z|) \in C$.

Induction Hypothesis. Assume that

$$H_Z \circ L(1) \circ L(2) \circ \cdots \circ L(|Z|) \circ \langle [R_1, W_1, 1], [R_2, W_2, 2], \ldots, [R_{j-1}, W_{j-1}, j-1] \rangle \in C \quad . \tag{3.17}$$

Induction Step. We use (P2) to prove that $H_Z \circ L(1) \circ L(2) \circ \cdots \circ L(|Z|) \circ \langle [R_1, W_1, 1], [R_2, W_2, 2], \ldots, [R_j, W_j, j] \rangle \in C$. Because $j \in Z$, the following holds.

$$H_Z \circ L(1) \circ L(2) \circ \cdots \circ L(|Z|) \circ \langle [R_1, W_1, 1], [R_2, W_2, 2], \ldots, [R_{j-1}, W_{j-1}, j-1] \rangle \; [j] \; H \circ L(j) \tag{3.18}$$

Consider x in $R_j.var$. Because G satisfies (C1), x is not written by a process other than j in $H_Z \circ L(1) \circ L(2) \circ \cdots \circ L(|Z|) \circ \langle [R_1, W_1, 1], [R_2, W_2, 2], \ldots, [R_{j-1}, W_{j-1}, j-1] \rangle$. Hence, we have the following.

$$\begin{aligned} (\forall x : x \in R_j.var \;\; :: \;\; & value(x,\; H_Z \circ L(1) \circ L(2) \circ \cdots \circ L(|Z|) \circ \\ & \langle [R_1, W_1, 1], [R_2, W_2, 2], \ldots, [R_{j-1}, W_{j-1}, j-1] \rangle) \\ = \;\; & value(x,\; H \circ L(j))) \end{aligned} \tag{3.19}$$

By (3.13), (3.17), (3.18), (3.19), and (P2), we conclude that $H_Z \circ L(1) \circ L(2) \circ \cdots \circ L(|Z|) \circ \langle [R_1, W_1, 1], [R_2, W_2, 2], \ldots, [R_j, W_j, j] \rangle \in C$.

□

By construction, each process in Z executes $r+1$ remote events in G. As shown above, G satisfies conditions (C1) through (C4). Hence, by (3.16) and Claim 3.1, the lemma follows. □

We now present our first main result. According to this result, there exists a fundamental trade-off between write-contention and time-complexity in solutions to the mutual exclusion problem. This result also shows a trade-off between the degree of atomicity and time-complexity.

Theorem 3.3: For any $S = (C, P, V)$ with write-contention $w > 1$ that solves the minimal mutual exclusion problem, if each event accesses at most v remote variables, then there exists an i-computation in C that contains $\Omega(\log_{vw} N)$ remote events but no Eat_i event.

Proof: $\langle\rangle$ is a P-computation and satisfies (C1), (C2), (C3), and (C4). By repeatedly applying Lemma 3.6, this implies that there exists a computation F in C that satisfies (C1) and (C4) and that contains $\Omega(\log_{((2v+1)^2vw)} N) = \Omega(\log_{vw} N)$ remote events of some process i in P. By Lemma 3.2, $F_i \in C$ holds, from which the theorem follows. □

Corollary 3.2: For any system S satisfying the conditions of Theorem 3.3, there exist $\Omega(N)$ processes i in P for which the conclusion of the theorem holds.

□

If v is taken to be a positive constant, then it is possible to show that the

bound of Theorem 3.3 is asymptotically tight for any value of w. In particular, an algorithm by Mellor-Crummey and Scott given in [45] solves the mutual exclusion problem for w processes, in $O(1)$ time, with access-contention (and hence write-contention) w. By applying this solution within a balanced w-ary tree with N leaves, it is possible to obtain an N-process $\Theta(\log_w N)$ mutual exclusion algorithm with access-contention w.

Note that Mellor-Crummey and Scott's algorithm uses load-and-store and compare-and-swap. Even with weaker atomic operations, logarithmic behavior can be achieved. In particular, an N-process $\Theta(\log_2 N)$ mutual exclusion algorithm based on read/write atomicity has been given in Figure 2.3. This algorithm has access-contention (and hence write-contention) two.

3.5 Bounds for Cache-Coherent Multiprocessors

On cache-coherent shared-memory multiprocessors, the number of remote memory references may be reduced: if a process repeatedly accesses the same remote variable, then the first access may create a copy of the variable in a local cache line, with further accesses being handled locally. In this section, we count the number of distinct remote variables a process must access to solve the minimal mutual exclusion problem. A lower bound on such a count not only implies a lower bound on the number of cache misses a process causes, but also implies that these cache misses will incur global traffic.

We prove two lower bounds. First, in Theorem 3.4 below, we show that if the conditions of Theorem 3.3 are strengthened so that at most c processes

can concurrently access (read *or* write) any variable, then some process accesses $\Omega(\log_{vc} N)$ distinct remote variables before eating. Second, in Theorem 3.5 below, we show that with the conditions of Theorem 3.3 unchanged, i.e., write-contention is w, then some process accesses $\Omega(\sqrt{\log_{vw} N}\ )$ distinct remote variables before eating. Before establishing the first of these results, we introduce some additional definitions.

Definition: Consider a remote event e of a process p in a computation H. Let X be the remote variables accessed by e. If e is the first event by p in H that accesses some variable in X, then we say that e is an *expanding event* in H. If e is a read (write) event, and if e is the first event by p in H that reads (writes) some variable in X, then we say that e is an *expanding read (write) event* in H. If e is neither an expanding read nor an expanding write, then we say that e is a *nonexpanding event* in H. □

An expanding event can be an expanding read, or an expanding write, or both. Note, however, that an expanding read (write) is not necessarily an expanding event. We count the number of expanding events in order to determine the number of distinct remote variables accessed. Observe that if a process executes r expanding events, then it accesses at least r distinct remote variables.

Because the first result of this section is based on a restriction on all concurrent accesses (rather than only concurrent writes) of the same variable, it is necessary to replace condition (C3) by the following.

- (C5) For any events $[R, W, i]$ and $[T, U, j]$ in H, if $((R.var \cup W.var) \cap (T.var \cup U.var)) \neq \{\}$, then $i = j$. Informally, each variable is accessed by

at most one process in H.

Our next lemma provides the induction step that leads to the lower bound in Theorem 3.4.

Lemma 3.7: Let $S = (C, P, V)$ be a shared-memory system with access-contention c that solves the minimal mutual exclusion problem. Let $Y \subseteq P$ be a set of n processes, and let H be a Y-computation in C satisfying (C2), (C4), and (C5) such that each process in Y executes r expanding remote events in H. Suppose that each event accesses at most v remote variables. Then, there exist $Z \subseteq Y$, where $|Z| = \lceil (n-1)/(2v+1)vc \rceil$, and a Z-computation G in C satisfying (C2), (C4), and (C5) such that each process in Z executes $r+1$ expanding remote events in G.

Proof: The proof strategy is as follows. We show that there exists $Z \subseteq Y$ that can execute another remote event without violating any of the conditions (C2), (C4), or (C5). We eliminate processes not in Z, i.e., ones that may violate some condition. Finally, we construct a Z-computation G that satisfies (C2), (C4), and (C5).

Because H satisfies (C5), it is possible to prove a result similar to Lemma 3.5 showing that there exists $Y1 \subseteq Y$, where $|Y1| \geq n-1$, such that the following holds: for any $i \in Y1$, there exists an i-computation $B(i)$ such that $H \circ B(i) \in C$, $B(i)$ does not contain Eat_i, and $B(i)$ has at least one expanding remote event. (If there are two processes that do not have an expanding remote event after H, then the Exclusion requirement can be violated; note that (C5) implies that these processes do not access any common variable in their entry sections.) For

$i \in Y1$, let $B(i) = F(i) \circ \langle [R_i, W_i, i], \ldots \rangle$ where $[R_i, W_i, i]$ is the first expanding remote event in $B(i)$.

We construct $Y2$, a subset of $Y1$, as follows. First, select a process $i \in Y1$. Let $X = \{x \mid x \in R_i.var \cup W_i.var$ and x is remote to $i\}$, i.e., X is the set of remote variables accessed by the event $[R_i, W_i, i]$. By assumption, $|X| \leq v$. Let $Q_X = \{j \mid j \in Y1 \wedge j \neq i \wedge (R_j.var \cup W_j.var) \cap X \neq \{\}\}$, i.e., Q_X includes those processes other than i that access variables in X. Because access-contention is c, it is straightforward to use Lemma 3.4 to show that $|Q_X| \leq v(c-1)$. Delete i and all processes in Q_X from $Y1$, and add i to $Y2$. Repeat the above procedure until $Y1$ is empty. By construction,

$$|Y2| \geq \lceil (n-1)/vc \rceil \quad . \tag{3.20}$$

Observe that if $i \in Y2$, $j \in Y2$, and $i \neq j$ hold, then $[R_i, W_i, i]$ and $[R_j, W_j, j]$ do not access a common variable. Thus, there is no information flow among $\{[R_i, W_i, i] \mid i \in Y2\}$. Now, we identify any possible information flow between $\{[R_i, W_i, i] \mid i \in Y2\}$ and the events in H of processes in $Y2$. Recall that $\{[R_i, W_i, i] \mid i \in Y2\}$ contains events that can be applied after H.

Suppose that $x \in R_p.var \cup W_p.var$ and x is remote to p. Without loss of generality, we assume x is local to q for some $q \neq p$. Note that q may or may not be a member of $Y2$. We construct E by the following rules.

- (R1): If $q \in Y2$, then introduce an edge (p, q).
- (R2): If there is process $w \in Y2$ that accesses x in H, where $w \neq p \wedge w \neq q$, then introduce an edge (p, w). Note that, because H satisfies (C2), $q \notin Y2$ holds.

Because (R1) and (R2) are exclusive, at most one edge is introduced for each remote variable an event accesses. Because each event accesses at most v remote variables, at most v edges are introduced for each remote event. We eliminate all edges by applying Theorem 3.2. The number of vertices is reduced by a factor of $1/(2v+1)$. These remaining vertices represent the subset of processes selected from the original n processes in Y. We use Z to denote this subset of Y. Note that, for any $i \in Z$, by Rule (R1), $[R_i, W_i, i]$ does not access any variable that is local to another process in Z, and by Rule (R2), it does not access a variable that is accessed in H by other processes in Z.

Without loss of generality, assume the processes are numbered so that $Z = \{1, 2, \ldots, |Z|\}$. By (3.20), we have $|Z| \geq \lceil (n-1)/(2v+1)vc \rceil$. The computation G we seek is defined as follows.

$$G = H_Z \circ F(1) \circ F(2) \circ \cdots \circ F(|Z|) \circ \langle [R_1, W_1, 1], [R_2, W_2, 2], \ldots, [R_{|Z|}, W_{|Z|}, |Z|] \rangle$$

Because H satisfies (C5), H also satisfies (C1). Thus, by Lemma 3.2, $H_Z \in C$. It is straightforward to use this fact to prove that $G \in C$.

By construction, each process in Z executes $r+1$ expanding remote events in G. To complete the proof of Lemma 3.7, it suffices to prove that G satisfies (C2), (C4), and (C5). We consider each of these conditions as a separate case. In these cases, we make use of the fact that, because H satisfies (C2), (C4), and (C5), H_Z also satisfies (C2), (C4), and (C5).

Condition $(C2)$. Because H_Z satisfies (C2), and because no $F(i)$ contains an expanding remote event, $H_Z \circ F(1) \circ F(2) \circ \cdots \circ F(|Z|)$ satisfies (C2). By (R1), no $[R_i, W_i, i]$ accesses a variable that is local to another process in Z. Hence, G satisfies (C2).

Condition $(C4)$. By construction, $F(i)$ does not contain Eat_i, and $[R_i, W_i, i] \neq$

Eat_i. Hence, G satisfies (C4).

Condition (C5). H_Z satisfies (C2) and (C5), and each $F(i)$ does not contain an expanding remote event, so $H_Z \circ F(1) \circ F(2) \circ \cdots \circ F(|Z|)$ satisfies (C5). Hence, to complete the proof that G satisfies (C5), it suffices to prove that for each distinct i and j in Z, $[R_i, W_i, i]$ does not access a variable that is accessed by $[R_j, W_j, j]$ or by any event of process j in H_Z or $F(j)$.

Because $F(j)$ contains no expanding remote event, any variable accessed by process j in $F(j)$ is either local to j or accessed remotely by j in H. By (R1), $[R_i, W_i, i]$ does not access a variable that is local to process j. By (R2), $[R_i, W_i, i]$ does not access a variable that is remotely accessed by j in H. Hence, $[R_i, W_i, i]$ does not access a variable that is remotely accessed by j in H_Z. By the definition of Z (specifically, the construction of $Y2$), the remote variables accessed by $[R_i, W_i, i]$ and $[R_j, W_j, j]$ are distinct. Hence, $[R_i, W_i, i]$ does not access a variable that is accessed by $[R_j, W_j, j]$. Hence, we conclude that G satisfies (C5).

This concludes the proof of Lemma 3.7. □

Theorem 3.4: For any $S = (C, P, V)$ with access-contention $c > 1$ that solves the minimal mutual exclusion problem, if each event accesses at most v remote variables, then there exists an i-computation in C containing no Eat_i event in which $\Omega(\log_{vc} N)$ distinct remote variables are accessed. □

Proof: $\langle\rangle$ is a P-computation and satisfies (C2), (C4), and (C5). By repeatedly applying Lemma 3.7, this implies that there exists a computation F in C that satisfies (C4) and (C5) (and hence C(1)) and that contains $\Omega(\log_{vc} N)$ expanding

remote events of some process i in P. By Lemma 3.2, $F_i \in C$ holds, from which the theorem follows. □

Corollary 3.3: For any system S satisfying the conditions of Theorem 3.4, there exist $\Omega(N)$ processes i in P for which the conclusion of the theorem holds. □

The tree-based algorithms mentioned after Corollary 3.2 have time complexity $\Theta(\log_c N)$, i.e, the bound of Theorem 3.4 is asymptotically tight for any value of c if v is taken to be a positive constant.

In the remainder of this section, we prove a lower bound on the number of distinct remote variable accesses required for solving the minimal mutual exclusion problem with write-contention w. Before proving this result, we define the notion of a "critical" remote event. Such events are used in the next theorem to count the number of distinct remote variables a process must access in its entry section. After showing that some process must execute $\Omega(\log_{vw} N)$ critical remote events before entering its critical section, we investigate how this bound is related to the number of distinct remote variables accessed.

Definition: Consider a computation H that contains a nonexpanding event e by process i. Let X denote the remote variables accessed by e. Let $S \equiv \{f \mid$ for some $x \in X$, f is the last event by i in H that accesses x before $e\}$. Observe that $|S| \leq |X|$. The first event of S in H is called the *predecessor* of e in H. Note that any suffix of H that contains the predecessor of e contains events by i (before e) that collectively access all variables in X. □

Definition: Consider a remote event e of a process i in a computation H. Event e is a *critical* event in H iff one of the following holds: e is an expanding write in H; e is an expanding read in H; e is a nonexpanding event and there is an expanding write by i between e and its predecessor in H. □

The next lemma is a variation of Lemma 3.5 that deals with critical remote events. Suppose that $S = (C, P, V)$ solves the minimal mutual exclusion problem and let $i \in P$ and $H \in C$. Corresponding to the definition prior to Lemma 3.5, we say that i *has a critical remote event after* H iff the following holds: there exists a remote event e of process i, and an i-computation L consisting of local events, each differing from Eat_i, such that $H \circ L \circ e \in C$ holds, where e is critical in $H \circ L \circ e$.

Lemma 3.8: Suppose that $S = (C, P, V)$ solves the minimal mutual exclusion problem. Let $Z \subseteq P$ be a set of n processes, and let H be a Z-computation in C satisfying (C1), (C2), (C3), and (C4). Then, there exists a Z-computation H' in C satisfying (C1), (C2), (C3), and (C4) such that H' contains all events contained in H and at least $n-1$ processes in Z have a critical remote event after H'.

Proof: Lemma 3.5 implies that at least $n-1$ of the processes in Z have a remote event after H. If all $n-1$ of these remote events are critical after H, then the conclusion of the lemma holds. So, assume that one of these events is noncritical after H. Then, there exists a process p in Z and a computation

$$H \circ L \circ \langle e \rangle \in C \quad , \tag{3.21}$$

where L is a p-computation consisting of only local events, and e is a noncritical remote event of p in $H \circ L \circ \langle e \rangle$. Because e is noncritical, we have

$$H \circ L \circ \langle e \rangle \;=\; X \circ \langle f \rangle \circ Y \circ L \circ \langle e \rangle \quad , \tag{3.22}$$

where f is the predecessor of e in $H \circ L \circ \langle e \rangle$, and Y contains no expanding write by p.

Let $G \;\equiv\; X \circ \langle f \rangle \circ Y_p \circ L \circ \langle e \rangle \circ (Y - Y_p)$. Observe that G is a Z-computation. We will first show that $G \in C$ holds in a series of steps, and then show that G satisfies (C1) through (C4). Observe that G contains all events contained in H and more remote events than H. By the Progress requirement, this implies that we can apply this argument only a finite number of times, i.e., if we repeatedly apply Lemma 3.5 and construct a new computation in the manner in which G is constructed, then we eventually obtain a computation H' such that applying Lemma 3.5 yields $n-1$ processes in Z, each of which has a *critical* remote event after H'. By our construction, H' is a computation in C, satisfies (C1) through (C4), and contains all events contained in H.

To begin the construction of G, note that, because $H \in C$, (3.22) implies $H = X \circ \langle f \rangle \circ Y \in C$. Furthermore, by assumption, H satisfies (C1). Hence, by Lemma 3.2, we have the following.

$$X \circ \langle f \rangle \circ Y_p \in C \tag{3.23}$$

We now apply Lemma 3.3 to prove that $X \circ \langle f \rangle \circ Y_p \circ L \in C$ holds. In applying Lemma 3.3, we use the following assertions.

$$X \circ \langle f \rangle \circ Y \;[p]\; X \circ \langle f \rangle \circ Y_p \tag{3.24}$$

$$X \circ \langle f \rangle \circ Y \circ L \in C \tag{3.25}$$

(3.24) holds by definition, and (3.25) follows from (3.21), (3.22), and (P1).

Because H satisfies (C2), by (3.22), both $X \circ \langle f \rangle \circ Y$ and $X \circ \langle f \rangle \circ Y_p$ also satisfy (C2). Also, recall that L is a p-computation consisting of local events and that p is active in H. Thus, no event in L accesses a variable that is written by processes other than p in either $X \circ \langle f \rangle \circ Y$ or $X \circ \langle f \rangle \circ Y_p$. Hence, by (3.23), (3.24), (3.25), and Lemma 3.3, the following holds.

$$X \circ \langle f \rangle \circ Y_p \circ L \in C \tag{3.26}$$

The next step in the proof is to use (P2) to establish that $X \circ \langle f \rangle \circ Y_p \circ L \circ \langle e \rangle$ is in C, where e is as defined at the beginning of the proof. Let $e = [R_p, W_p, p]$. The following assertion follows from (3.22).

$$X \circ \langle f \rangle \circ Y_p \circ L \ [p] \ H \circ L \tag{3.27}$$

Because $H \circ L \circ \langle e \rangle$ satisfies (C1), for all $x \in R_p.var$, the following holds.

$$value(x,\ X \circ \langle f \rangle \circ Y_p \circ L) = value(x,\ H \circ L) \tag{3.28}$$

By (3.21), (3.26), (3.27), (3.28), and (P2), it follows that

$$X \circ \langle f \rangle \circ Y_p \circ L \circ \langle e \rangle \in C \quad . \tag{3.29}$$

We now show that G is in C by establishing the following claim.

Claim 3.2. $X \circ \langle f \rangle \circ Y_p \circ L \circ \langle e \rangle \circ (Y - Y_p) \in C$.

Proof: Let $(Y - Y_p) = \langle e_0, e_1, \ldots, e_m \rangle$. The proof is by induction on $|Y - Y_p|$.

Induction Base. By (3.29), $X \circ \langle f \rangle \circ Y_p \circ L \circ \langle e \rangle \in C$ holds.

Induction Hypothesis. Suppose that $X \circ \langle f \rangle \circ Y_p \circ L \circ \langle e \rangle \circ \langle e_0, e_1, \ldots,$

$e_{m-1}\rangle \in C$ holds.

Induction Step. We prove that $X \circ \langle f \rangle \circ Y_p \circ L \circ \langle e \rangle \circ \langle e_0, e_1, \ldots, e_m \rangle \in C$ holds. Without loss of generality, assume that $Y = Q \circ \langle e_m \rangle \circ T$. Then, by (P1), (3.25) implies that the following holds.

$$X \circ \langle f \rangle \circ Q \circ \langle e_m \rangle \in C \tag{3.30}$$

Let $e_m = [R, W, i]$ for some $i \neq p$. Because $i \neq p$, the following holds.

$$X \circ \langle f \rangle \circ Y_p \circ L \circ \langle e \rangle \circ \langle e_0, e_1, \ldots, e_{m-1} \rangle \ [i] \ X \circ \langle f \rangle \circ Q \tag{3.31}$$

Let $x \in R.var$. We now show that x is not written by any event in Y_p, L, or $\langle e \rangle$. Suppose that x is written by e or by an event in Y_p. e is noncritical and hence is not an expanding write. Also, Y_p does not contain any expanding write by p. Thus, by (3.22), x is also written by p in $X \circ \langle f \rangle$. Because $i \neq p$, this implies that H does not satisfy (C1), which is a contradiction.

Now, suppose that x is written by an event in L. Recall that L consists only of local events of p. Thus, event $e_m = [R, W, i]$, which is in H, reads a local variable of process $p \neq i$. Because p is active in H, this implies that H does not satisfy (C2), which is a contradiction. Thus, we conclude that x is not written by any event in Y_p, L, or $\langle e \rangle$. This implies that, for each x in $R.var$, $writer(x,\ X \circ \langle f \rangle \circ Y_p \circ L \circ \langle e \rangle \circ \langle e_0, e_1, \ldots, e_{m-1} \rangle) = writer(x,\ X \circ \langle f \rangle \circ Q)$ holds. By Lemma 3.1, this implies that the following holds.

$$value(x,\ X \circ \langle f \rangle \circ Y_p \circ L \circ \langle e \rangle \circ \langle e_0, e_1, \ldots, e_{m-1} \rangle) = value(x,\ X \circ \langle f \rangle \circ Q) \tag{3.32}$$

By the induction hypothesis, (3.30), (3.31), (3.32), and (P2), $X \circ \langle f \rangle \circ Y_p \circ L \circ \langle e \rangle \circ \langle e_0, e_1, \ldots, e_m \rangle \in C$. □

Having shown that G is in C, we now show that G satisfies (C1) through (C4). Observe that the events in $L \circ \langle e \rangle$ are the only events in G that are not in H. L consists only of local events of process p, none of which are Eat_p. Also, e, being a noncritical remote event, does not access any remote variable that p does not access in H. Hence, because H satisfies (C2) through (C4), it follows that G also satisfies (C2) through (C4).

As for (C1), our proof obligation is to show that no event in G reads a variable previously written by another process. Because H satisfies (C1), by (3.22), no event in $X \circ \langle f \rangle \circ Y_p$ reads a variable previously written by another process.

Now, consider events in $L \circ \langle e \rangle \circ (Y - Y_p)$. Observe that L consists only of local events of p, p is active in H, and H satisfies (C2). Hence, no event in L reads a variable that is previously written by another process in G.

If e reads a variable that is previously written by another process in G, then that variable is written in X, because $\langle f \rangle \circ Y_p \circ L$ consists of events by p. If e reads a variable that is written by another process in X, then, by the definition of a predecessor, there exists an event in $\langle f \rangle \circ Y_p$ that accesses that same variable. However, this implies that H violates (C1) or (C3), which is a contradiction.

Finally, because H satisfies (C1), no event in $Y - Y_p$ reads a variable written by another process in $X \circ \langle f \rangle \circ Y_p$. By the reasoning at the end of the proof of Claim 3.2, no event in $Y - Y_p$ reads a variable that is written by p in $Y_p \circ L \circ \langle e \rangle$. We conclude that G satisfies (C1).

We have shown that if some process in Z has a next remote event after H that is noncritical, then there exists a Z-computation in C satisfying (C1), (C2),

(C3), and (C4) that contains more remote events than H. As noted previously, if this argument could be applied repeatedly, then it would be possible to construct a computation in C that violates the Progress requirement. This proves the lemma. □

The next lemma is a stronger version of Lemma 3.6 in which only critical remote events are counted rather than all remote events.

Lemma 3.9: Let $S = (C, P, V)$ be a shared-memory system with write-contention w that solves the minimal mutual exclusion problem. Let $Y \subseteq P$ be a set of n processes, and let H be a Y-computation in C satisfying (C1), (C2), (C3), and (C4) such that each process in Y executes r critical remote events in H. Suppose that each event accesses at most v remote variables. Then, there exist $Z \subseteq Y$, where $|Z| = \lceil (n-1)/(2v+1)^2 vw \rceil$, and a Z-computation G in C satisfying (C1), (C2), (C3), and (C4) such that each process in Z executes $r+1$ critical remote events in G.

Proof: Lemma 3.8 implies that there exists $Y1 \subseteq Y$, where $|Y1| \geq n-1$, such that the following holds: for any $i \in Y1$, there exists an i-computation $L(i)$ consisting of local events, such that $H \circ L(i) \circ [R_i, W_i, i] \in C$, where $[R_i, W_i, i]$ is a critical remote event in $H \circ L(i) \circ [R_i, W_i, i]$. The rest of the proof is identical to that of Lemma 3.6. □

According to the following theorem, among the $\Omega(\log_{vw} N)$ remote events mentioned in Theorem 3.3, $\Omega(\sqrt{\log_{vw} N})$ distinct remote variables are accessed.

Theorem 3.5: For any $S = (C, P, V)$ with write-contention $w > 1$ that solves the minimal mutual exclusion problem, if each event accesses at most v remote variables, then there exists an i-computation in C containing no Eat_i event in which $\Omega(\sqrt{\log_{vw} N}\ )$ distinct remote variables are accessed.

Proof: $\langle\rangle$ is a P-computation and satisfies (C1), (C2), (C3), and (C4). By repeatedly applying Lemma 3.9, this implies that there exists a computation F in C that satisfies (C1) and (C4) and that contains $\Omega(\log_{vw} N)$ critical remote events of some process i in P. By Lemma 3.2, $F_i \in C$. Let W denote the number of expanding writes in F_i, let R denote the number of expanding reads in F_i, and let E denote the number of nonexpanding critical remote events in F_i. Then, because F_i contains $\Omega(\log_{vw} N)$ critical remote events,

$$(W + R + E) \geq c \cdot \log_{vw} N \tag{3.33}$$

holds for some positive constant c. Let D denote the number of distinct remote variables accessed in F_i. Observe that D is at least as big as W and R. Also, D is at least as big as the number of distinct remote variables accessed by events in E. The following claim provides an upper bound on the number of events in E.

> **Claim 3.3.** There are at most D nonexpanding critical events between two successive expanding writes in F_i.
>
> **Proof:** Let x and y denote two successive expanding writes in F_i, and let $F_i = X \circ \langle x \rangle \circ Y \circ \langle y \rangle \circ Z$. By assumption, Y does not contain an expanding write. Let $e_0, e_1, \ldots, e_m$ denote the nonexpanding critical events in Y. By the definition of a critical event, their predecessors

in F_i appear in X. We claim that each e_j, where $1 \leq j \leq m$, accesses a remote variable that is not accessed in $e_0, \ldots, e_{j-1}$. Otherwise, the predecessor of e_j in F_i is not an event in X, which is a contradiction. Because e_0 accesses at least one remote variable, $e_0, e_1, \ldots, e_m$ access at least $m+1$ distinct remote variables. Thus, $m < D$ holds, which proves the claim. □

By Claim 3.3, at most D nonexpanding critical events may occur between an expanding write and the next expanding write (if any). In addition, by the definition of a critical event, no nonexpanding critical remote events may exist before the first expanding write. Thus, we have at most D nonexpanding critical remote events per expanding write, i.e., $E \leq DW$. Because $D \geq W$ and $D \geq R$ hold, this implies that

$$D \geq \mathbf{max}(W, R, E/W) \quad . \tag{3.34}$$

We now show that $D \geq m \cdot \sqrt{\log_{vw} N}$ for some positive constant m. Assume, to the contrary, that $D < m \cdot \sqrt{\log_{vw} N}$. Then, by (3.34), we have $W < m \cdot \sqrt{\log_{vw} N}$ and $R < m \cdot \sqrt{\log_{vw} N}$. By (3.33), this implies that

$$\frac{E}{W} > \frac{c \cdot \log_{vw} N - 2m \cdot \sqrt{\log_{vw} N}}{m \cdot \sqrt{\log_{vw} N}} \quad .$$

By (3.34), this inequality implies that $D \geq s \cdot \sqrt{\log_{vw} N}$ for some positive constant s. □

Corollary 3.4: For any system S satisfying the conditions of Theorem 3.5, there exist $\Omega(N)$ processes i in P for which the conclusion of the theorem holds.

□

3.6 Discussion

The time bounds proved in this chapter establish that trade-offs exist between time complexity and write- and access-contention for solutions to the minimal mutual exclusion problem. These time bounds also show that similar trade-offs exist between time complexity and atomicity. Because any algorithm that solves the leader election or mutual exclusion problems also solves the minimal mutual exclusion problem, these trade-offs apply to these problems as well. Our results imply that synchronization in shared-memory multiprocessors has some inherent cost involving the global interconnect, either in terms of a larger amount of global traffic, or in terms of higher contention.

One may be interested in determining the effect of contention on space requirements. It is quite easy to show that solving the minimal mutual exclusion problem with write-contention w requires at least N/w variables. In particular, it can be shown that every process writes a variable before eating. So, consider the computation in which every process is enabled to perform its first write. Because write-contention is w, the total number of variables enabled to be written is $\Omega(N/w)$. It can be shown that this bound is tight; it is possible to obtain a deadlock-free solution to mutual exclusion with write-contention w by arranging test-and-set variables in a balanced w-ary tree with $\lceil N/w \rceil$ leaves.

Chapter 4

Hardware Support for Local Spin Synchronization

4.1 Introduction

A concurrent program consists of a collection of sequential programs called *processes*, which communicate by accessing shared data structures called *objects*. Associated with each object is a set of *operations*; such operations provide the only means for accessing the object. Coarse-grained atomic operations can be arbitrarily powerful and therefore are convenient to use when designing concurrent programs. However, a program with atomic operations that are overly complex cannot be readily translated into machine instructions and hence cannot be considered suitable for execution. In view of this, concurrent programs are often developed in a top-down fashion: under this approach, a program is first developed using coarse-grained objects, and then each coarse-grained object is implemented by fine-grained objects.

In this chapter, we consider the latter problem, i.e., that of implementing one kind of object in terms of another. Our specific goal is to determine the

extent to which such implementations can be achieved without busy-waiting on remote variables. As seen in Chapter 2, such busy-waiting should be avoided if good scalability is needed. Our focus in this chapter is on the distributed shared memory multiprocessors; we investigate implementations of shared objects without global busy-waiting on such machines. Recall that on such machines each shared variable is local to one processor and is remote to all others.

Objects in the most general form allow conditional operations, i.e., operations with enabling conditions that involve shared variables. An example of such an object is the semaphore object that allows the P primitive, which consists of an assignment "$X := X - 1$", where X is shared, that may be executed only when the enabling condition "$X > 0$" holds. We represent conditional operations by means of the syntax "**await** $B \rightarrow S$", where B is a boolean expression over program variables and S is a multiple-assignment. This operation can be executed only when its enabling expression B is true. It is atomically executed (when enabled) by performing its assignment S. We abbreviate such an operation as "**await** B" if its assignment is null, and as "S" if its enabling expression is identically true. Observe that variables are read if they appear in the right hand side of assignment S, and that they are written if they appear in the left hand side of S. Also variables are used to specify when such assignments can be executed, if they appear in the enabling condition B.

In this chapter, we determine if there exist fine-grained shared objects from which other objects with arbitrary conditional operations of the form "**await** $B \rightarrow S$" can be implemented without global busy-waiting. We show that very simple fine-grained objects suffice, particularly single-reader, single-writer variables.

Recent work on wait-free synchronization has dealt with the implementation of objects that are only read or written. The seminal work on this subject is Lamport's paper on interprocess communication [36]; other representative papers include [1, 4, 5, 8, 12, 16, 18, 31, 33, 42, 48, 51, 54, 57, 59]. In work on wait-free synchronization, the central problem is that of implementing one class of objects from another class of objects without any waiting. Because waiting in any form is precluded in such implementations, both classes of objects are clearly restricted to allow only operations that may read or write shared variables.

In a recent paper [6], Anderson showed that any object that allow only operations of the form "S" can be implemented from single-reader, single-writer variables, without busy-waiting on remote variables. In this chapter, we extend past work on object implementations by considering operations with enabling conditions.

In the rest of this chapter, we say that a class C of objects is *implementable* from another class D of objects iff any operation of C can be implemented by using operations of D without global busy-waiting.

The key result of this chapter is as follows:

> Any object that allows operations of the form "**await** $B \rightarrow S$" is implementable by using simpler objects that allow only operations of the form "$X := y$" and "$y := X$", where y is a private boolean variable and X is a shared, single-reader, single-writer boolean variable.[1]

This result establishes that on distributed shared memory machines, any concurrent program can be refined in practice to one in which only local spins

[1] An *m-reader, n-writer variable* can be read by m processes and can be written by n processes.

are employed, and that only very simple primitives are required for local-spin synchronization.

The rest of this chapter is organized as follows. In Section 4.2, we define what it means to implement an object of one class by using objects of another class. The result mentioned above are established in Sections 4.3 and 4.4. We discuss the implications of our results in Section 4.5.

4.2 Implementations

As stated in the introduction, we consider a class C of objects is implementable from another class D of objects iff any operation of C can be implemented without global busy-waiting by using operations of D. In this section, we define the notion of an implementation formally. Except for modifications to handle liveness conditions, our notion of implementation is similar to that given by Herlihy in [27], which is based on the I/O automata of Lynch and Tuttle [44]. The following description is adopted from Herlihy [27].

4.2.1 I/O automata

We model concurrent programs using a simplified form of I/O automata [44]. I/O automata provide a convenient way for describing what it means for one object to implement another.

An *I/O automaton* A is a nondeterministic automaton with the following components:

- A set of *states*, including a distinguished nonempty set of initial states.

- A set of *actions*, which are partitioned into sets of *internal* and *external* actions. External actions are divided into *input* and *output* actions.

- A *transition relation*, which is a set of triples (s, e, t), with states s, t and action e. Such a triple, called a *step*, indicates that an automaton in state s can transit to state t by executing the action e.

If (s, e, t) is a step, then we say that e is *enabled* at s. Otherwise, we say that e is *disabled* at s. A *history* of an automaton A is a sequence $s_0 \stackrel{e_0}{\rightarrow} s_1 \stackrel{e_1}{\rightarrow} \cdots$, where each (s_i, e_i, s_{i+1}) is a step of A and s_0 is an initial state.

We can compose a set of I/O automata $\{A_1, \ldots, A_n\}$ to get a new I/O automaton A. A state of the composed automaton A is the Cartesian product of states of component automata A_i's, and an initial state is defined analogously. The set of input actions of A is obtained from the union of sets of input actions of each A_i, by eliminating input actions that belong to output actions of any component automata. The output actions and internal actions of A are simply all output actions and all internal actions of each A_i respectively. The transition relation of A is the set of triples (s, e, t) such that, for every component automaton A_i, either e is an action of A_i and the projection of the triple onto A_i is a step of A_i, or e is not an action of A_i and the projection of s and t onto A_i yields identical states for A_i. The subhistory of H consisting of actions of A_i is denoted by $H|A_i$, where H is a history of a composite automaton A.

4.2.2 Concurrent Programs

A *concurrent program* consists of a set of processes and a set of objects. In the remainder of this section, we use P to denote a process, X an object, *op* an

operation of X, and *res* a result. P is an I/O automaton with output actions *invoke*(P, *op*, X) and input actions *respond*(P, *res*, X). X has input actions *invoke*(P, *op*, X), and output actions *respond*(P, *res*, X). These actions are referred to as *invocations* and *responses*. An invocation and a response *match* if their process and object names are the same. If an invocation is not followed by a matching response, then it is said to be a *pending* invocation.

A *concurrent program* $\{P_1, \ldots, P_n; A_1, \ldots, A_m\}$ is an I/O automaton composed from processes $\{P_1, \ldots, P_n\}$ and objects $\{A_1, \ldots, A_m\}$ by identifying *invoke* actions of processes and objects with corresponding *respond* actions of objects and processes respectively. If a history has an invocation as its first action and if it alternates matching invocations and responses, it is called a *sequential* history.

Each history H induces a partial order on its operations: an operation p precedes another operation q in this ordering, denoted $p \prec_H q$ if the response for p precedes the invocation for q. Observe that if H is sequential, then $\prec_H$ is a total order. A concurrent program $\{P_1, \ldots, P_n; A_1, \ldots, A_m\}$ is *linearizable* if, for each history H, there is a sequential history S such that $H|P_i = S|P_i$ for any P_i and $\prec_H$ is a subset of $\prec_S$. In other words, each operation "appears" to take effect instantaneously at some point between its invocation and its response. Henceforth, every object we consider is assumed to be linearizable.

4.2.3 Implementations

An *implementation* of an object A is a concurrent program $\{F_1, \ldots, F_n; R\}$ obtained by composing an object R with processes F_i's called *front-ends*. Front-ends communicate only by applying operations to R. Input actions of R are composed with matching output actions of each F_i, and input actions of each F_i

with matching output actions of R. These composed actions are internal actions of A. The external actions of A are all the external actions of each F_i.

A history α of an automaton A is *fair* if the following conditions hold for each action e of A:

- If α is finite, then e is disabled in the final state of α.
- If α is infinite, then either α contains an infinite number of executions of e, or α contains an infinite number of states in which e is disabled.

Note this definition corresponds to weak fairness of every action.[2] Unless otherwise noted, we henceforth assume that all histories are fair.

Let I_j be an implementation of A_j. Following Lynch and Tuttle [44] and Herlihy [27], we say that I_j is *correct*, if for every fair history H of every system $\{P_1, \ldots, P_n; A_1, \ldots, I_j, \ldots, A_m\}$, there exists a fair history H' of $\{P_1, \ldots, P_n; A_1, \ldots, A_j, \ldots, A_m\}$, such that $H|\{P_1, \ldots, P_n\} = H'|\{P_1, \ldots, P_n\}$.

A *local spinning implementation* is a correct implementation that in every history of the implementation, no invocation of P_i is pending across an infinite number of steps of F_i that access variables remote to P_i.

4.2.4 Reasoning about Programs

For brevity, we represent concurrent programs using pseudocode rather than as I/O automata. It is straightforward to translate such a program into a collection of I/O automata. We briefly explain the correspondence between such programs and I/O automata.

[2] Alternatively, we could have defined weak fairness for every process. Our result would still hold if we used such a notion of fairness.

A program in pseudocode is specified using labeled **await** statements; it corresponds to processes in I/O automata. The *variables* used in the program represent objects; a state of an object corresponds to an assignment of values to such variables that are used to represent that object. A statement, in the program for process P, that accesses variables that represent an object X, specifies actions $invoke(P, op, X)$ and $respond(P, res, X)$. Each process of a concurrent program has a special private variable called its *program counter*: the statement with label k in process p may be executed only when the value of the program counter of p equals k. The statements and program counters suggest a transition relation in I/O automata.

To facilitate the presentation, we assume that shared variables appear only in **await** statements. For an example of the syntax we employ for programs, see Figure 4.3. Note that the fairness requirement of Section 4.2.3 implies that each continuously enabled statement is eventually executed. Unless otherwise noted, we henceforth assume that all histories are fair.

4.2.5 Example: A Semaphore Lock

A lock program that uses a semaphore object X is depicted in 4.1. The program may be considered as a representation of the I/O automata with a set of states $\{p, q, r, s\}$, where p is the initial state. The set of actions is

$\{invoke(u,$ "**await** $X > 0 \rightarrow X := X - 1"$, $X)$,

$respond(u,$ "**await** $X > 0 \rightarrow X := X - 1"$, $X)$,

$invoke(u,$ "$X := X + 1"$, $X)$,

$respond(u,$ "$X := X + 1"$, $X)\}$.

The *invoke* actions are the input actions of X and output actions of u, and

shared var $X : 0..1$
initially $X = 1$

$/*$ X is a shared object with operations
"**await** $X > 0 \rightarrow X := X - 1$" and "$X := X + 1$" $*/$

process u
while *true* **do**
0: **await** $X > 0 \rightarrow X := X - 1$;
1: Critical Section;
2: $X := X + 1$
od

Figure 4.1: A semaphore lock program.

respond actions are the output actions of X and input actions of u. And the transition relation is

$\{(p,\ invoke(u,\ \text{"}\mathbf{await}\ X > 0\ \rightarrow\ X := X - 1\text{"},\ X),\ q),$

$(q,\ respond(u,\ \text{"}\mathbf{await}\ X > 0\ \rightarrow\ X := X - 1\text{"},\ X),\ r),$

$(r,\ invoke(u,\ \text{"}X := X + 1\text{"},\ X),\ s),$

$(s,\ respond(u,\ \text{"}X := X + 1\text{"},\ X),\ p)\}.$

Suppose that there is another process v that executes the same program as u. The infinite history in which v is blocked at its first statement while u executes infinite number of statements is a fair history, because actions of v are disabled infinitely often.

4.2.6 Implementations by Critical Sections

Suppose that object C is implemented by a set of objects D. If C is an object in program P, and if program Q is obtained from P by substituting D for C, then

we refer to P as the *implemented program*, and Q as the *implementation*. One way to ensure linearizable implementation is to use critical sections. Because most of the implementations discussed in this chapter are based upon critical sections, we discuss this approach in more detail in this section.

In the usual definition of the mutual exclusion problem, there are no constraints on *when* critical sections may be executed, other than mutual exclusion and starvation-freedom requirements defined in Section 1.2. As a result, the mutual exclusion problem is not a very useful paradigm when implementing **await** statements. To see this, consider the statement "**await** $B \rightarrow S$". Not only does this statement specify that S must be executed atomically, it also gives an enabling condition B that must be true prior to each such execution. Thus, if S is to be implemented as a critical section, then, in addition to mutual exclusion and starvation-freedom, the following condition must hold: the critical section for S can be executed only when B is true. The usual definition of the mutual exclusion problem does not take this requirement into account.

With this discussion in mind, we now present the conditions required of an implementation (when using critical sections). Assume that the **await** statements of the implemented program are denoted "**await** $B_k \rightarrow S_k$", where $k \geq 0$. Then, an implementation is obtained by replacing each "**await** $B_k \rightarrow S_k$" by a program fragment P_k of the following form.

Entry Section;

Critical Section;

Exit Section

The entry and exit sections are not allowed to modify any variable of the implemented program. Further, the critical section is required to have the effect of

atomically performing the assignment S_k when executed in isolation. Informally, an implementation is correct iff the following requirements hold for each k.

- *Boundedness*: P_k is free of unbounded **do-od** loops that may generate an unbounded number of remote memory references.
- *Exclusion*: For each $j \neq k$, if S_j and S_k have variables in common, then the critical section of P_j cannot be executed concurrently with that of P_k.
- *Synchrony*: When the critical section of P_k first becomes enabled, B_k holds.
- *Progress*: The critical and exit sections of P_k eventually terminate, and if B_k holds continuously, then the entry section of P_k also eventually terminates.

As shown later, these requirements can be defined formally using invariants and leads-to assertions.

4.3 Results

In this section, we present the main result of the chapter. We first consider a number of lemmas that are needed in order to establish Theorem 4.1.

Lemma 4.1: Any shared object is implementable by shared objects that only have operations of the form "**await** B" or "S". □

We establish this lemma in Section 4.4 by considering a variant of the mutual exclusion problem called the conditional mutual exclusion problem. In the conditional mutual exclusion problem, there is a predicate associated with each

process that must be true when that process executes its critical section. This problem is motivated by our desire to implement operations of the form "**await** $B \rightarrow S$" by a bounded number of invocations of operations of the form "**await** B" and "S". Our solution to this problem shows that it is possible to implement any operation that combines both waiting and assignment in terms of operations that do not. The next two lemmas show that we can simplify operations of the form "**await** B" and "S", respectively.

Lemma 4.2: Any shared object that only has operations of the form "**await** B" or "S" is implementable by shared objects that only have operations of the form "S".

Proof: We use $B_1, \ldots, B_N$ to denote the enabling predicates of operations of the form "**await** B". The implementation is obtained by replacing each operation of the form "**await** B_k" by an operation of the form "**await** X_k", where X_k is a "fresh" shared boolean variable; X_k is initially true iff predicate B_k is initially true. Each operation of the form "S" that may possibly modify B_k is modified to also atomically assign $X_k := B_k$. This ensures that $X_k = B_k$ is kept invariant for each k. Note that we have not introduced an unbounded number of invocations.

Observe that we may implement "**await** X_k" by a busy-waiting loop "**while** $\neg X_k$ **do od**", and that each X_k is a single-reader, multi-writer variable. Now we can implement X_k in a wait-free manner using single-reader, single-writer boolean variables [1, 4, 5, 8, 12, 16, 18, 31, 33, 36, 42, 48, 51, 54, 57, 59]. All these variables are made local to the waiting process that invokes "**await** B".

Although, we now have a process that may invoke an unbounded number of operations (for evaluating $\neg X_k$ repeatedly), note that this process accesses only local variables in the busy-waiting loop. Thus, Lemma 4.2 holds. □

Our final lemma shows that we can implement "S" by single-reader, single-writer variables.

Lemma 4.3: Any object whose operation is of the form "S" is implementable by single-reader, single-writer variables.

Proof: Our proof obligation is to show that any operation of the form "S" can be implemented by using operations of the form "$X := y$" and "$y := X$", where y is a private boolean variable and X is a shared, single-reader, single-writer, boolean variable, without busy-waiting on remote variables.

Anderson has shown in [6] that the mutual exclusion problem can be solved without global busy-waiting using only single-reader, single-writer, boolean variables. In this solution, global busy-waiting is avoided and shared variables are accessed only within statements of the form "$X := y$" and "$y := X$", where y is private and X is shared. Let ENTRY and EXIT denote the entry and exit sections of such a solution. Then, we can implement each **await** statement of the form "S" as follows.

ENTRY;

S;

EXIT

By the properties of the mutual exclusion problem, S cannot be executed con-

currently with any other statement that may modify the variables appearing in S. It is therefore straightforward to implement S in terms of single-reader, single-writer boolean variables.

Unfortunately, there is a potential problem with our implementation as it stands now. Suppose, for example, that the implemented program contains an assignment of the form "$A,\ B := \mathit{true},\ \mathit{false}$". Consider the program fragment in the implementation that replaces this assignment. Suppose that this program fragment happens to be executed when $\neg A \ \wedge \ B$ holds, and that in the critical section of this program fragment, A is assigned before B. Observe that A and B both hold in the interval between these two assignments. Thus, it is possible for a single process p to execute two consecutive **await** statements "**await** A" and "**await** B" in this interval. Such an execution corresponds to a linearization in which "**await** A" occurs after the program fragment for "$A,\ B := \mathit{true},\ \mathit{false}$" and "**await** B" occurs before. However, because "**await** A" and "**await** B" are supposed to be executed consecutively by p, such an execution is not linearizable. This scenario can be prevented by replacing the statement "**await** B" by a program fragment of the following form.

ENTRY;

EXIT;

await B

With this modification, ENTRY and EXIT are executed by p between the statements "**await** A" and "**await** B". Thus, these two statements both cannot be executed in the interval between the assignments to A and B as described above. By introducing additional ENTRY and EXIT sections in this manner, it is possible to obtain a correct implementation without introducing global busy-waiting.

□

The preceding three lemmas establish the following theorem, which implies that any object can be "reduced", using local-spin techniques, to one whose operations is as fine-grained as possible.

Theorem 4.1: Any object can be implemented by single-reader, single-writer variables. □

4.4 Conditional Mutual Exclusion

In this section, we define the conditional mutual exclusion problem. We then present a program that solves this problem in which only **await** statements of the form "**await** B" and "S" are used. Our solution to this problem is used in the proof of Lemma 4.1 in Section 4.3. In the conditional mutual exclusion problem, there are N processes, each of which has the following structure.

do *true* $\rightarrow$

Noncritical Section;

Entry Section;

Critical Section;

Exit Section

od

Associated with each process i is an enabling condition $B[i]$ that must be true when that process enters its critical section. An enabling predicate's value can be changed only by a process in its critical section. It is assumed that each

process begins execution in its noncritical section. It is further assumed that each critical section execution terminates. By contrast, a process is allowed to halt in its noncritical section. No variable appearing in any entry or exit section may be referred to in any noncritical section. Also, with the exception of enabling predicates, no such variable may be referred to in any critical section. Let $ES(i)$ ($CS(i)$) be a predicate that is true iff the value of process i's program counter equals a label of a statement appearing in its entry section (critical section). Let $BCS(i)$ be a predicate that is true iff the value of process i's program counter equals the label of the first statement in its critical section. (For simplicity, we assume that this statement is executed once per critical section execution.) Then, the requirements that must be satisfied by a program that solves this problem are as follows.

- *Exclusion*: $(\forall i, j : i \neq j :: CS(i) \;\Rightarrow\; \neg CS(j))$ is an invariant. Informally, at most one process can execute its critical section at a time.

- *Synchrony*: $(\forall i :: BCS(i) \;\Rightarrow\; B[i])$ is an invariant. Informally, when a process first enters its critical section, its enabling predicate is true.

- *Progress*: $(\forall i :: ES(i) \;\mapsto\; CS(i) \;\vee\; \neg B[i])$ holds. Informally, if a process is in its entry section and its enabling predicate continuously holds, then that process eventually executes its critical section.

We also require that each process in its exit section eventually enters its noncritical section; this requirement holds trivially for the solution considered in this chapter, so we will not consider it further. Observe that the conditional mutual exclusion problem reduces to the mutual exclusion problem when each process's enabling predicate is always identically true.

```
process i
do true →
    Noncritical Section;
    ENTRY;
    do ¬B[i] → EXIT; ENTRY od;
    Critical Section;
    EXIT
od
```

Figure 4.2: Using mutual exclusion to solve conditional mutual exclusion.

If global busy-waiting is allowed, then it is straightforward to use a solution to the mutual exclusion problem to obtain a program that solves the conditional mutual exclusion problem. In particular, consider the program given in Figure 4.2, which is taken from [10]; in this program, ENTRY and EXIT denote entry and exit sections from an N-process solution to the mutual exclusion problem. In order to execute its critical section, process i repeatedly executes ENTRY and EXIT, checking $B[i]$ in between. Its critical section is entered only if $B[i]$ is true; otherwise, EXIT and ENTRY are executed again. Note that when process i has executed ENTRY but not EXIT, it is effectively within its "mutual exclusion critical section".

A program that solves the conditional mutual exclusion problem without global busy-waiting is given in Figure 4.3. This program uses the doubly linked list that is implemented by *Pred* and *Suc*; *Pred*[0] points to the tail of the list and *Suc*[0] points to the head of the list. The program also uses the queue that is implicitly implemented by *Count*, *Head*, and *ticket*.

Loosely speaking, this program works as follows. When process i wants to enter its critical section, it first enters the queue by executing statement 1, and

then waits until it becomes the first process in the queue ($Head = ticket$). Then, if its enabling predicate $B[i]$ holds, it can enter its critical section. Otherwise, process i executes statement 4 to insert its process id to the tail of the doubly linked list ($Pred[0]$), and executes statement 5 to remove itself from the (implicit) queue. Then process i waits until "notified" by another process that i is enabled to enter its critical section ($Turn = i$). In this case, process i executes statement 8 to delete its id from the doubly linked list and executes its critical section.

After executing the critical section, process i traverses the linked list to see if there is an enabled process. If j is the first enabled process in the list, then process i informs process j that j may enter its critical section. If there is no enabled process in the list, process i increments *Head* to allow the next process in the queue, if any, to proceed. Note that processes in the list are given "priorities" over processes in the queue.

Observe that when multiple processes contend for the critical section, they are blocked at statement 2 or at statement 6. Also observe that a process, after executing its critical section, "wakes up" at most one process by executing either statement 12 or statement 15. From this fact, it can be shown that the following invariant holds, which implies that the Exclusion requirement is satisfied.

invariant $(\mathbf{N}i :: i@\{8\}) \leq 1$

When process i enters its critical section, it does so either by executing statement 3 when $B[i]$ holds, or by getting a "notification" at statement 6 from another process that executes statement 12. Such a notification occurs only if $B[i]$ holds. Thus, the following invariant holds, which implies that the Synchrony requirement holds.

invariant $(\forall i :: i@\{8\} \;\Rightarrow\; B[i])$

As explained above, processes are ordered in the list and the queue. Because such an order is statically kept in the program, it is straightforward to show that the following assertion holds, which implies that the Progress requirement holds.

$(\forall i :: i@\{1..7\} \ \mapsto \ i@\{8\} \ \vee \ \neg B[i])$

4.5 Discussion

Anderson's results [6] imply that objects with unconditional operations are implementable by single-reader, single-writer variables, in distributed shared memory machines and cache-coherent machines. Our results show that objects with conditional operations are also implementable by single-reader, single-writer variables, in distributed shared memory machines. In other words, any conditional operation can be implemented by using only simple reads and writes to such variables, and local spinning. Our results also show that, from a *computational* standpoint, operations that combine both waiting and assignment, such as the P semaphore primitive, are *not* fundamental.

```
shared var   Pred, Suc : array[0..N] of 0..N;
             B         : array[1..N] of boolean
             Count, Head, Turn : 0..N
initially    (∀ j :: Pred[j] = 0 ∧ Suc[j] = 0) ∧
             Count = 0 ∧ Head = 0 ∧ Turn = 0

process  i          { i ranges over 1..N }

private var more : boolean;
            scan, ticket : 0..N

do true →
     0: Noncritical Section;
     1: ticket, Count := Count, (Count + 1) mod (N + 1);
     2: await (Head = ticket) ;
     3: if ¬B[i] then
     4:     Pred[i], Suc[i], Pred[0], Suc[Pred[0]] := Pred[0], 0, i, i;
     5:     Head := (Head + 1) mod (N + 1);
     6:     await (Turn = i);
     7:     Suc[Pred[i]], Pred[Suc[i]], Turn := Suc[i], Pred[i], 0
        fi;
     8: Critical Section;
     9: more, scan := true, Suc[0];
    10: while more ∧ (scan ≠ 0) do
    11:     if B[scan] then
    12:         Turn, more := scan, false
            else
    13:         scan := Suc[scan]
            fi
        od;
    14: if more then
    15:     Head := (Head + 1) mod N
        fi
od
```

Figure 4.3: Program for conditional mutual exclusion.

Chapter 5

Concluding Remarks

5.1 Summary of Results

In this thesis, we have presented several results concerning scalable synchronization in shared-memory multiprocessing systems.

In Chapter 2, we proposed a time complexity measure that captures the communication overhead of shared-memory concurrent programs. Under our proposed measure, the time complexity of a concurrent program is measured by the number of remote memory references induced by the program. Our performance studies show that this measure is useful as a metric of the scalability of concurrent programs.

We presented a scalable N-process mutual exclusion algorithm based on read/write atomicity that has $O(\log N)$ time complexity. Its time complexity is better than that of any previous solutions to the mutual exclusion problem based on read and write instructions. We also presented an extension of our algorithm in which only $O(1)$ memory references are required to achieve mutual exclusion in the absence of contention. Our performance studies indicate

that these algorithms exhibit scalable performance under heavy contention. In fact, our mutual exclusion algorithm out-performs all prior algorithms based on read/write atomicity, and its performance under heavy contention rivals that of the fastest queue-based spin locks that employ strong primitives such as compare-and-swap or fetch-and-add.

In Chapter 3, we obtained several trade-offs between the contention and time complexity in synchronization algorithms. We showed that, if at most v remote variables can be accessed atomically, any solution to the N-process minimal mutual exclusion problem with write-contention w has $\Omega(\log_{vw} N)$ time complexity. We further showed that such a solution must access $\Omega(\sqrt{\log_{vw} N})$ distinct remote variables. For algorithms with access-contention c, we improved the latter bound to $\Omega(\log_{vc} N)$. As any solution to the mutual exclusion or the leader election problem also solves the minimal mutual exclusion problem, our trade-offs hold for these problems as well.

In most shared-memory multiprocessors, an atomic operation may access only a constant number of remote variables. In this case, the first and the last of our bounds are asymptotically tight. These results also show that our N-process $\Theta(\log_2 N)$ mutual exclusion algorithm based on read/write atomicity is optimal.

In Chapter 4, we showed that local-spin techniques are applicable to a wide class of synchronization problems with read/write atomicity. In particular, we showed that any shared object, no matter how complicated, can be implemented from single-reader, single-writer variables without global busy-waiting on distributed shared memory multiprocessors.

5.2 Future Research

For wait-free algorithms, Herlihy has characterized synchronization primitives by consensus number [27]. Such a characterization is not applicable when waiting is introduced. One way of determining the power of synchronization primitives in this case is to compare the time complexity of mutual exclusion using such primitives. For instance, it is possible to solve the mutual exclusion problem with $O(1)$ time complexity using fetch-and-store or fetch-and-add, while the best-known upper bound for read/write algorithms is $O(\log N)$ as achieved in Chapter 2. If a lower-bound result could be proved showing that this gap is fundamental, then this would establish that reads and writes are weaker than read-modify-writes from a performance standpoint. This would provide contrasting evidence to Herlihy's hierarchy, from which it follows that reads and writes are weaker than read-modify-writes from a *resiliency* standpoint. It is interesting to note that there exist read/write mutual exclusion algorithms with write-contention N that have $O(1)$ time complexity in the absence of competition [3, 37, 60]. Thus, establishing the above-mentioned lower bound for read/write algorithms will require proof techniques that differ from those given in Chapter 2.

We do not know whether the bound of Theorem 3.5 is tight. We conjecture that this bound can be improved to $\Omega(\log_{vw} N)$, which has a matching algorithm when v is taken to be a constant [60].

It is our belief that the most important contribution of Chapter 3 is to show that meaningful time bounds can be established for concurrent programming problems for which busy-waiting is inherent. We hope that our work will spark new work on time complexity results for such problems.

In Chapter 4, we have primarily limited our attention to determining the

possibility of object implementations without global busy-waiting in distributed shared memory multiprocessors. Other issues, such as performance, are yet to be considered. In all of our implementations, statements are implemented by using mutual exclusion. This is partly due to the fact that in our main result, namely the implementation of statements of the form "**await** $B \rightarrow S$", no restrictions are placed upon the variables appearing in B or S: such a statement could conceivably reference every shared variable of a program! Without such restrictions, an implementation must ensure that only one such statement is executed at a time. By imposing restrictions on variable access, it should be possible to implement **await** statements with greater parallelism. The development of such implementations is an important avenue for further research. Extending the results of Chapter 4 in order to apply to cache-coherent multiprocessors would also be interesting.

Bibliography

[1] Y. Afek, H. Attiya, D. Dolev, E. Gafni, M. Merritt, and N. Shavit, "Atomic Snapshots of Shared Memory", *Proceedings of the Ninth Annual Symposium on Principles of Distributed Computing*, 1990, pp. 1-14.

[2] A. Agarwal and M. Cherian, "Adaptive Backoff Synchronization Techniques", *Proceedings of the 16th International Symposium on Computer Architecture*, May, 1989, pp. 396-406.

[3] R. Alur and G. Taubenfeld, "Results about Fast Mutual Exclusion", *Proceedings of the Thirteenth IEEE Real-Time Systems Symposium*, December, 1992, pp. 12-21.

[4] J. Anderson, "Multi-Writer Composite Registers", *Distributed Computing*, Vol. 7, 1994, pp. 175-195.

[5] J. Anderson, "Composite Registers", *Distributed Computing*, Vol. 6, 1993, pp. 141-154.

[6] J. Anderson, "A Fine-Grained Solution to the Mutual Exclusion Problem", *Acta Informatica*, Vol. 30, No. 3, 1993, pp. 249-265.

[7] J. Anderson and M. Gouda, "A Criterion for Atomicity", *Formal Aspects of Computing: The International Journal of Formal Methods*, Vol. 4, No. 3, May, 1992, pp. 273-298.

[8] J. Anderson and B. Grošelj, "Beyond Atomic Registers: Bounded Wait-Free Implementations of Nontrivial Objects", *Science of Computer Programming*, Vol. 19, No. 3, December, 1992, pp. 197-237.

[9] T. Anderson, "The Performance of Spin Lock Alternatives for Shared-Memory Multiprocessors", *IEEE Transactions on Parallel and Distributed Systems*, Vol. 1, No. 1, January, 1990, pp. 6-16.

[10] G. Andrews, *Concurrent Programming: Principles and Practice*, The Benjamin/Cummings Publishing Company, Inc., Redwood City, California, 1991.

[11] J. Archibald and J. Baer, "An Economical Solution to the Cache Coherence Problem", *Proceedings of the 12th International Symposium on Computer Architecture*, June, 1985, pp. 355-362.

[12] J. Aspens and M. Herlihy, "Wait-Free Data Structures in the Asynchronous PRAM Model", *Proceedings of the Second Annual ACM Symposium on Parallel Architectures and Algorithms*, July, 1990.

[13] BBN Advanced Computers, *Inside the TC2000 Computer*, February, 1990.

[14] G. Bell, "Ultracomputers: A Teraflop Before Its Time", *Communications of the ACM*, Vol. 35, No. 8, 1992, pp. 26-47.

[15] P. Bitar and A. Despain, "Multiprocessor Cache Synchronization Issues, Innovations, Evolution", *Proceedings of the 13th International Symposium on Computer Architecture*, June, 1986, pp. 424-433.

[16] B. Bloom, "Constructing Two-Writer Atomic Registers", *IEEE Transactions on Computers*, Vol. 37, No. 12, December, 1988, pp. 1506-1514.

[17] J. Burns and N. Lynch, "Bounds on Shared Memory for Mutual Exclusion", *Information and Computation*, Vol. 107, 1993, pp. 171-184.

[18] J. Burns and G. Peterson, "Constructing Multi-Reader Atomic Values from Non-Atomic Values", *Proceedings of the Sixth Annual Symposium on Principles of Distributed Computing*, 1987, pp. 222-231.

[19] M. Censier and P. Feautier, "A New Solution to Coherence Problems in Multicache Systems", *IEEE Transactions on Computers*, Vol. 27, No. 12, December, 1978, pp. 1112-1118.

[20] K. Chandy and J. Misra, *Parallel Program Design: A Foundation*, Addison-Wesley, 1988.

[21] E. Dijkstra, "Solution of a Problem in Concurrent Programming Control", *Communications of the ACM*, Vol. 8, No. 9, 1965, pp. 569.

[22] E. Dijkstra, *A Discipline of Programming*, Prentice-Hall, Englewood Cliffs, New Jersey, 1976.

[23] C. Dwork, M. Herlihy, and O. Waarts, "Contention in Shared Memory Algorithms", *Proceedings of the 25th ACM Symposium on Theory of Computing*, May, 1993, pp. 174-183.

[24] A. Glew and W. Hwu, "A Feature Taxonomy and Survey of Synchronization Primitive Implementations", Technical Report UILU-ENG-91-2211, Center for Reliable and High-Performance Computing, University of Illinois at Urbana-Champaign, February, 1991.

[25] J. Goodman, M. Vernon, and P. Woest, "Efficient Synchronization Primitives for Large-Scale Cache-Coherent Multiprocessors", *Proceedings of the Third International Conference on Architectural Support for Programming Languages and Operating Systems*, 1989, pp. 64-75.

[26] G. Graunke and S. Thakkar, "Synchronization algorithms for shared-memory multiprocessors", *IEEE Computer*, Vol. 23, June, 1990, pp. 60-69.

[27] M. Herlihy, "Wait-Free Synchronization", *ACM Transactions on Programming Languages and Systems*, Vol. 13, No. 1, 1991, pp. 124-149.

[28] M. Herlihy, B. Lim, and N. Shavit, "Low Contention Load Balancing on Large-Scale Multiprocessors", *Proceedings of Symposium on Parallel Algorithms and Architectures '92*, 1992, pp. 219-227.

[29] M. Herlihy, N. Shavit, and O. Waarts, "Low Contention Linearizable Counting", *Proceedings of the 32nd IEEE Symposium on Foundations of Computer Science*, October, 1991, pp. 526-535.

[30] M. Herlihy and J. Wing, "Linearizability: A Correctness Condition for Concurrent Objects", *ACM Transactions on Programming Languages and Systems*, Vol. 12, No. 3, 1990, pp. 463-492.

[31] A. Israeli and M. Li, "Bounded time-stamps", *Proceedings of the 28th IEEE Symposium on Foundations of Computer Science*, 1987, pp. 371-382.

[32] J. Kessels, "Arbitration Without Common Modifiable Variables", *Acta Informatica*, Vol. 17, 1982, pp. 135-141.

[33] L. Kirousis, E. Kranakis, and P. Vitanyi, "Atomic Multireader Register", *Proceedings of the Second International Workshop on Distributed Computing*, Lecture Notes in Computer Science 312, pp. 278-296, Springer Verlag, 1987.

[34] D. Knuth, "Additional Comments on a Problem in Concurrent Programming Control", *Communications of the ACM*, Vol. 9, No. 5, 1966, pp. 321-322.

[35] Kendall Square Research, *Technical Summary*, Walthan, Massachusetts, 1992.

[36] L. Lamport, "On Interprocess Communication, Parts I and II", *Distributed Computing*, Vol. 1, 1986, pp. 77-101.

[37] L. Lamport, "A Fast Mutual Exclusion Algorithm", *ACM Transactions on Computer Systems*, Vol. 5, No. 1, February, 1987, pp. 1-11.

[38] L. Lamport, "*win* and *sin*: Predicate Transformers for Concurrency", *ACM Transactions on Programming Languages and Systems*, Vol. 12, No. 3, 1990, pp. 396-428.

[39] C. Lee and U. Ramachandran, "Synchronization with Multiprocessor Caches", *Proceedings of the 17th International Symposium on Computer Architecture*, May, 1990, pp. 27-37.

[40] D. Lenowski et. al., "The Directory-Based Cache Coherence Protocol for the DASH Multiprocessor", *Proceedings of the 17th International Symposium on Computer Architecture*, May, 1990, pp. 148-159.

[41] D. Lenowski et. al., "The Stanford DASH Multiprocessor", *IEEE Computer*, Vol. 25, No. 3, 1992, pp. 63-79.

[42] M. Li, J. Tromp, and P. Vitanyi, "How to Construct Wait-Free Variables", *Proceedings of International Colloquium on Automata, Languages, and Programming*, Lecture Notes in Computer Science 372, pp. 488-505, Springer Verlag, 1989.

[43] N. Lynch and N. Shavit, "Timing-Based Mutual Exclusion", *Proceedings of the Thirteenth IEEE Real-Time Systems Symposium*, December, 1992, pp. 2-11.

[44] N. Lynch and M. Tuttle, "An Introduction to Input/Output Automata", Technical Report MIT/LCS/TM-373, MIT Laboratory for Computer Science, November, 1988.

[45] J. Mellor-Crummey and M. Scott, "Algorithms for Scalable Synchronization on Shared-Memory Multiprocessors", *ACM Transactions on Computer Systems*, Vol. 9, No. 1, February, 1991, pp. 21-65.

[46] M. Merritt and G. Taubenfeld, "Knowledge in Shared Memory Systems", *Proceedings of the Tenth ACM Symposium on Principles of Distributed Computing*, August, 1991, pp. 189-200.

[47] M. Michael and M. Scott, "Fast Mutual Exclusion, Even With Contention", Technical Report, University of Rochester, June, 1993.

[48] R. Newman-Wolfe, "A Protocol for Wait-Free, Atomic, Multi-Reader Shared Variables, *Proceedings of the Sixth Annual Symposium on Principles of Distributed Computing*, 1987, pp. 232-248.

[49] S. Owicki and D. Gries, "An Axiomatic Proof Technique for Parallel Programs I", *Acta Informatica*, Vol. 6, 1976, pp. 319-340.

[50] G. Peterson, "Myths About the Mutual Exclusion Problem", *Information Processing Letters*, Vol. 12, No. 3, June, 1981, pp. 115-116.

[51] G. Peterson and J. Burns, "Concurrent Reading While Writing II: The Multi-Writer Case", *Proceedings of the 28th Annual Symposium on Foundations of Computer Science*, 1987.

[52] G. Peterson and M. Fischer, "Economical Solutions for the Critical Section Problem in a Distributed System", *Proceedings of the 9th ACM Symposium on Theory of Computing*, May, 1977, pp. 91-97.

[53] G. Pfister and A. Norton, "Hot Spot Contention and Combining in Multistage Interconnection Networks", *IEEE Transactions on Computers*, Vol. C-34, No. 11, November, 1985, pp. 943-948.

[54] A. Singh, J. Anderson, and M. Gouda, "The Elusive Atomic Register, Revisited", *Proceedings of the Sixth Annual Symposium on Principles of Distributed Computing*, 1987, pp. 206-221. To appear in *Journal of the ACM*.

[55] Sequent Computer Systems, *Sequent Technical Summary*, 1987.

[56] E. Styer, "Improving Fast Mutual Exclusion", *Proceedings of the Eleventh Annual ACM Symposium on Principles of Distributed Computing*, 1992, pp. 159-168.

[57] J. Tromp, "How to Construct an Atomic Variable", *Proceedings of the Third International Workshop on Distributed Algorithms*, Lecture Notes in Computer Science 392, pp. 292-302, Springer Verlag, 1989.

[58] P. Turán, "On an extremal problem in graph theory"(in Hungarian), *Mat. Fiz. Lapok*, Vol. 48, 1941, pp. 436-452.

[59] P. Vitanyi and B. Awerbuch, "Atomic Shared Register Access by Asynchronous Hardware", *Proceedings of the 27th IEEE Symposium on the Foundations of Computer Science*, 1986, pp. 233-243.

[60] J.-H. Yang and J. Anderson, "Fast, Scalable Synchronization with Minimal Hardware Support", *Proceedings of the Twelfth ACM Symposium on Principles of Distributed Computing*, August, 1993, pp. 171-182.

[61] J.-H. Yang and J. Anderson, "Time Bounds for Mutual Exclusion and Related Problems", *Proceedings of the 26th Annual ACM Symposium on Theory of Computing*, May, 1994, pp. 224-233.