A Lock-Free Approach to Object Sharing in Real-Time Systems

by

Srikanth Ramamurthy

A dissertation submitted to the faculty of the University of North Carolina at Chapel Hill in partial fulfillment of the requirements for the degree of Doctor of Philosophy in the Department of Computer Science.

Chapel Hill

1997

Approved by:

Prof. James Anderson, Adviser

Prof. Prasun Dewan, Reader

Prof. Kevin Jeffay, Reader

©1997
Srikanth Ramamurthy
ALL RIGHTS RESERVED

Srikanth Ramamurthy. A Lock-Free Approach to Object Sharing in Real-Time Systems
(Under the direction of Professor James H. Anderson.)

Abstract

This work aims to establish the viability of lock-free object sharing in uniprocessor real-time systems. Naive usage of conventional lock-based object-sharing schemes in real-time systems leads to unbounded *priority inversion*. A priority inversion occurs when a task is blocked by a lower-priority task that is inside a critical section. Mechanisms that bound priority inversion usually entail kernel overhead that is sometimes excessive.

We propose that lock-free objects offer an attractive alternative to lock-based schemes because they eliminate priority inversion and its associated problems. On the surface, lock-free objects may seem to be unsuitable for hard real-time systems because accesses to such objects are not guaranteed to complete in bounded time. Nonetheless, we present scheduling conditions that demonstrate the applicability of lock-free objects in hard real-time systems. Our scheduling conditions are applicable to schemes such as rate-monotonic scheduling and earliest-deadline-first scheduling.

Previously known lock-free constructions are targeted towards asynchronous systems; such constructions require hardware support for strong synchronization primitives such as compare-and-swap. We show that constructions for uniprocessor real-time systems can be significantly simplified — and the need for strong primitives eliminated — by exploiting certain characteristics of real-time scheduling schemes.

Under lock-based schemes, a task can perform operations on many shared objects simultaneously via nested critical sections. For example, using nested critical sections, a task can atomically dequeue an element from one shared queue and enqueue that element in another shared queue. In order to achieve the level of functionality provided by nested critical sections, we provide a lock-free framework that is based on a multi-word compare-and-swap primitive and that supports multi-object accesses — the lock-free counterpart to nested critical sections. Because multi-word primitives are not provided in hardware, they have to be implemented in software. We provide a time-optimal implementation of the multi-word compare-and-swap primitive.

Finally, we present a formal comparison of the various object-sharing schemes based on scheduling conditions, followed by results from a set of simulation experiments that we conducted. Also, as empirical proof of the viability of lock-free objects in practical systems, we present results from a set of experiments conducted on a desktop videoconferencing system.

Acknowledgements

I would like to thank my adviser Jim Anderson for educating me and advising me over the years with much enthusiasm and patience. I would also like to thank him for supporting me for several years.

I would also like to thank my committee: Rance Cleaveland, Prasun Dewan, Rich Gerber, Kevin Jeffay, and Don Stanat. I thank them for their willingness to find time to attend the myriad pre-proposal meetings, oral examinations, proposals, and one-on-one meetings. Special thanks to Rance Cleaveland for driving long distances to attend the various meetings, and to Rich Gerber for enduring many long meetings over the phone and providing very insighful comments on my work. I am also grateful to Kevin Jeffay for being extremely supportive and encouraging of my work.

My work has also benefited significantly from many discussions with the following friends and colleagues: Steve Goddard, Rohit Jain, Mark Moir, M. Paramasivam, Tom White, and Jason Wilson. I would like to thank Dave Bennett, Peter Nee, Mark Parris, and Don Stone for their help in all the experimental work.

I would also like to thank many of my friends in Chapel Hill for their support and camaraderie; graduate school would have been boring and weary without them. I am thankful to my parents for encouraging me to strive for the best always and for their support. I am also thankful to Ranga and Malathi for their love and support. This dissertation would not have been possible were it not for the support of my wife Bhuva. She has tolerated a lot over the past few of years with much cheer.

Contents

List of Figures viii

List of Tables x

1 Introduction 1
- 1.1 Lock-Based Object Sharing in Hard Real-Time Systems . . . 3
- 1.2 Lock-Free Objects . . . 6
- 1.3 Background on Real-Time Systems . . . 11
  - 1.3.1 Real-Time Scheduling Schemes . . . 12
- 1.4 Our Contributions . . . 15
  - 1.4.1 Schedulability of Lock-Free Task Sets . . . 15
  - 1.4.2 Eliminating Hardware Support for Strong Primitives . . . 16
  - 1.4.3 A General Framework for Lock-Free Accesses . . . 17
  - 1.4.4 Experimental Results . . . 18
- 1.5 Organization of the Dissertation . . . 19

2 Background and Related Work 20
- 2.1 Scheduling Conditions . . . 20
  - 2.1.1 Static-Priority Scheduling Conditions . . . 21
  - 2.1.2 Dynamic-Priority Scheduling Conditions . . . 23
- 2.2 Lock-Based Object Sharing . . . 26
  - 2.2.1 Static-Priority Scheduling Conditions . . . 32
  - 2.2.2 Dynamic-Priority Conditions . . . 33
- 2.3 Accounting for System Overhead Costs . . . 35
- 2.4 Lock-Free Object Implementations . . . 38
  - 2.4.1 Linearizability . . . 39
  - 2.4.2 The Consensus Hierarchy . . . 41
  - 2.4.3 Universal Constructions of Lock-Free Objects . . . 43
  - 2.4.4 Specific Objects . . . 49

3 Scheduling Conditions 51
- 3.1 Assumptions and Definitions . . . 52
- 3.2 Preliminary Lemmas . . . 58
- 3.3 Static-Priority Scheduling Conditions . . . 63

3.4 Dynamic-Priority Scheduling Conditions . . . 69
3.5 Accounting for Different Retry-Loop Costs . . . 74
3.5.1 Additional Notation and Definitions . . . 75
3.5.2 Bounding Interference Cost . . . 77
3.5.3 Static-Priority Scheduling Schemes . . . 78
3.5.4 Dynamic-Priority Scheduling Schemes . . . 84
3.6 Static-Priority Scheduling Conditions . . . 85
3.7 Dynamic-Priority Scheduling Conditions . . . 88

4 Support for Strong Primitives 90
4.1 The Real-Time Task Model . . . 91
4.2 Definitions and Notation . . . 93
4.3 Universality of Load and Store Instructions . . . 96
4.4 Implementing CAS using Loads and Stores . . . 101
4.4.1 Correctness Proof . . . 106
4.5 Implementing CAS using Move, Load, and Store Instructions . . . 127
4.5.1 Correctness Proof . . . 130
4.6 Implementing Multi-Word Primitives . . . 138
4.6.1 A Wait-Free Implementation of MWCAS . . . 138
4.6.2 Correctness Proof . . . 147

5 A Transactional Framework for Implementing Lock-Free Objects 161
5.1 Lock-Free Transactions . . . 162
5.1.1 Correctness Proof . . . 169

6 A Comparative Study of Object-Sharing Schemes 178
6.1 Formal Comparison . . . 179
6.1.1 Static-Priority Scheduling . . . 179
6.1.2 Dynamic-Priority Scheduling . . . 181
6.1.3 Wait-Free Objects . . . 183
6.2 Simulation Results . . . 185
6.3 Timing Measurements . . . 198
6.4 Experiments on a Videoconferencing System . . . 205
6.4.1 Experimental Setup . . . 206
6.4.2 Static-Priority Scheduling . . . 208
6.4.3 Dynamic-Priority Scheduling . . . 212

7 Conclusions 215
7.1 Summary . . . 215
7.2 Conclusions and Future Work . . . 218

Bibliography 222

List of Figures

1.1 The priority inversion problem. 4
1.2 Lock-free queue implementation. 7
1.3 Common real-time scheduling schemes. 13

2.1 Implementation of a monitor task using using *rendezvous* primitive. 26
2.2 Illustrated example depicting schemes that bound priority inversion 28
2.3 Lock-free queue implementation. 40
2.4 Example interleavings of lock-free enqueue and dequeue operations. 40
2.5 Definitions of common instructions 43
2.6 Implementation of Herlihy's small object constructions. 45
2.7 Illustration depicting Anderson and Moir's large object construction. 47

3.1 Illustration of the task sets defined in Examples 3.1 and 3.2. 56
3.2 Pseudo-code to calculate f_i^v values. 80

4.1 Operations interleavings on an asynchronous system and on a real-time system 92
4.2 Incorrect solution to consensus using loads and stores. 96
4.3 The enabled late-write problem. 97
4.4 Consensus using loads and stores. 98
4.5 Proof of Lemma 4.1. 99
4.6 Implementation of CAS using loads and stores. 102
4.7 Illustrated example depicting the implementation of CAS using load and store instructions. 105
4.8 Proof of Lemma 4.5. 112
4.9 Subcase 2.1 in the Proof of Lemma 4.14. 122
4.10 Subcase 2.2 in the Proof of Lemma 4.14. 123
4.11 Implementation of CAS/Read using move. 128
4.12 Proof of Lemma 4.20. 133
4.13 Wait-free implementation of MWCAS. 140
4.14 Illustrated example to explain the MWCAS implementation in Section 4.6 . 144
4.15 Example of a Read operation by Task T_r. 145
4.16 Some subtleties of the MWCAS implementation in Section 4.6. 146
4.17 Proof of Lemma 4.29. 153

5.1 An example transaction. 163
5.2 Implementation of the *MEM* array for lock-free transactions (depicted for $B = 5$). 164
5.3 Lock-free transaction implementation. 165
5.4 Proof of Lemma 5.8. 176

6.1 BU and BCU curves for r/w ratio = 0.25 and cost ratio = 0.50 189
6.2 BU and BCU curves for r/w ratio = 0.25 and cost ratio = 1.00 190
6.3 BU and BCU curves for r/w ratio = 0.25 and cost ratio = 2.00 191
6.4 BU and BCU curves for r/w ratio = 0.50 and cost ratio = 0.50 192
6.5 BU and BCU curves for r/w ratio = 0.50 and cost ratio = 1.00 193
6.6 BU and BCU curves for r/w ratio = 0.50 and cost ratio = 2.00 194
6.7 BU and BCU curves for r/w ratio = 0.75 and cost ratio = 0.50 195
6.8 BU and BCU curves for r/w ratio = 0.75 and cost ratio = 1.00 196
6.9 BU and BCU curves for r/w ratio = 0.75 and cost ratio = 2.00 197
6.10 Tasks and shared queues in the videoconferencing system. 207

List of Tables

2.1 Herlihy's consensus-number hierarchy. 42

6.1 Worst-case execution times of various implementations of strong primitives. 201
6.2 Worst-case cost of operations on commonly used data structures. 202
6.3 Task characteristics in the videoconferencing system. 209
6.4 Interrupt handler execution times and periods. 210

Chapter 1

Introduction

A real-time computer system is required to provide timely responses to external events occurring in its operating environment. The performance of such a system is directly related to its ability to adhere to timing constraints placed by these external events and the strictness of these constraints. Based on the nature of these constraints, real-time systems can be broadly classified into *hard* and *soft* real-time systems. Hard real-time systems are required to meet every timing constraint without fail. The performance of hard real-time systems must be predictable in a deterministic sense. In contrast, temporal correctness requirements in soft real-time systems are less stringent — i.e., failure to meet every timing constraint does not affect the correctness of the system.

An example of a hard real-time system is a robot-arm controller in an automated factory. In such a system, external sensors periodically provide the controlling computer with information on the position, velocity, orientation, etc., of the different sections of the arm. Based on this information, the controlling computer determines whether a signal must

be sent to an actuator to change the velocity or direction of a section of the arm. If the computer does not send the signal to the actuator in a timely manner, significant material damage may be caused. For example, suppose that the external sensors indicate that a collision with another robot arm is imminent within one second. Clearly, the controlling computer must process the information from the sensors and ensure that the arm stops moving within one second, failing which both arms could collide. Thus, timing constraints in hard real-time systems must *never* be violated.

On the other hand, consider a videoconferencing system in which audio and video samples are received from a communications network. Suppose that the video monitor refreshes the screen every 33 milliseconds. To ensure good playback quality, each video (audio) sample must be processed and displayed (played back) within 33 milliseconds. However, unlike the case of the robot arm, failure to process a video sample within 33 milliseconds will not result in material damage; it will only result in inferior playback quality. Therefore, in order to achieve reasonable playback quality, we only need to ensure that the video and audio samples are processed and played back within 33 milliseconds "most of the time".

In this dissertation, we focus our attention on uniprocessor hard real-time systems. Such systems are obviously important in their own right. In addition, even if the environment can tolerate the non-hard-real-time behavior of a system, there is an advantage to treating the system as if it were a hard real-time system: guarantees of adherence to performance constraints can be provided, thereby ensuring that the system exhibits predictable behavior.

1.1 Lock-Based Object Sharing in Hard Real-Time Systems

Typically, a real-time[1] system invokes a *task* — i.e., a sequential program — in response to each external (or internal) event. *External* events are repeatedly generated by sensors monitoring the operating environment, while *internal* events are triggered by alarms generated by internal timers. Because a real-time system is required to handle multiple independent events, tasks associated with different events are usually multiprogrammed on a single processor. Shared objects are of interest in such systems because they provide a framework for intertask communication and task synchronization.

A *shared object* is a data structure (e.g., a queue) that can be accessed or modified by means of a fixed set of operations (e.g., enqueue and dequeue). Each operation has a fixed set of input parameters and produces a set of output values. Because uncontrolled accesses to shared objects can result in corruption of data, shared objects are typically embedded inside critical sections and are accessed using a mutual exclusion protocol (e.g., semaphores, monitors, etc.). Using such protocols, a task first acquires a lock associated with a shared object, accesses the object, and then releases the lock. Tasks contending for that object must wait for the lock to be released. A key advantage of lock-based protocols is that they are easy to use and provide a framework for implementing arbitrary shared objects.

Lock-based object sharing in real-time systems is complicated by the fact that such systems require predictable upper bounds on object access times. Naive usage of lock-based objects in such systems can result in unbounded priority inversion. A *priority inversion*

[1] Henceforth, we use the terms "real-time system" and "hard real-time system" interchangeably.

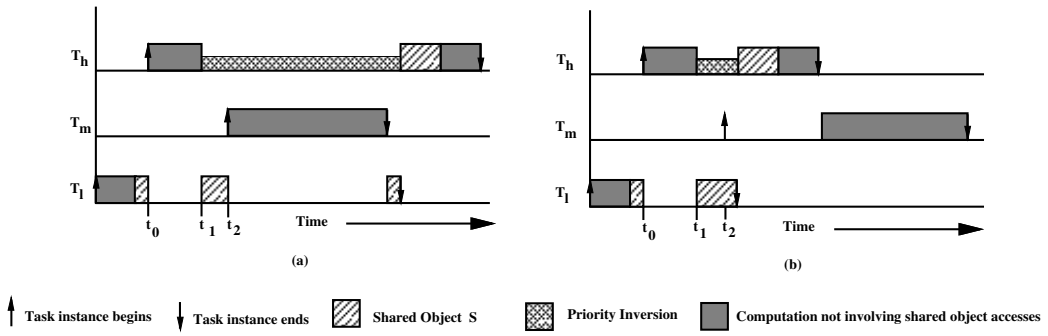


Figure 1.1: (a) Priority inversion. (b) The priority inheritance protocol.

exists when a given task must wait on a task of lower priority to release a critical section, as illustrated in Figure 1.1(a). In this figure, at time t_0, task T_l is preempted by a higher-priority task T_h while accessing a lock-based shared object S. Because task T_l holds the lock for object S, T_h cannot access S at time t_1. Hence, T_h relinquishes the processor to task T_l, which resumes its object access. However, before T_l can complete its object access, it is preempted by task T_m which has priority lower than T_h's priority but higher than T_l's priority. Thus, T_h is prevented from accessing S for a lengthy interval of time because T_l must wait for T_m to complete before it can release the lock on S. Priority inversion is said to occur for the duration of time in which T_h waits for T_l to release the lock on S.

Unless priority inversions are carefully controlled in a hard real-time system, it may be difficult or impossible to ensure that task deadlines are always met. For this reason, substantial effort has been devoted to the problem of bounding the duration of priority inversions in real-time systems either by using kernel support [19, 26, 44, 69, 71] or by using scheduling techniques [29, 65, 83]. Because techniques based on the former approach are very relevant to this dissertation, they are briefly described below. (All of these techniques are described in detail in Chapter 2.)

Typically, when priority inversion is controlled using kernel support, the kernel provides a registration mechanism by which tasks identify the objects that they access. Using this information, the kernel dynamically adjusts task priorities to ensure that a task within a critical section executes at a priority that is sufficiently high to bound the duration of any priority inversion. An example of a mechanism that uses kernel support to bound the duration of priority inversion is the priority inheritance protocol (PIP) [69, 71]. Figure 1.1(b) illustrates the working of the PIP on the task set depicted in Figure 1.1(a). In Figure 1.1(b), task T_h is blocked from accessing S at time t_1. However, the PIP ensures that T_l's priority is raised to that of task T_h at that time, thereby precluding task T_m from preempting T_l at time t_2. Thus, the duration of time for which T_h is blocked by T_l is no longer than the time taken to access object S — a significant improvement from the situation in Figure 1.1(a).

Although the PIP and similar schemes provide a general framework for real-time synchronization, they suffer from several drawbacks. First, they entail additional operating system overhead that sometimes can be excessive, particularly if object accesses are of short duration. In such situations, overhead associated with modifying task priorities may constitute a significant fraction of object access costs. Secondly, using some of these schemes complicates run-time mode change protocols. *Mode changes* in real-time systems are characterized by a change in task parameters, or by the addition or removal of tasks from a system. For example, a change in the sampling rate of a tracking task in a radar system might require that task to execute at a faster rate; this constitutes a mode change. Mode change protocols must maintain the semantic requirements of a task set and ensure that

deadlines are not missed when tasks are added or deleted. Such protocols are further complicated when tasks access lock-based shared objects under some PIP-like schemes. This is because, in order to guarantee bounded priority inversion and freedom from deadlock during mode changes, these mode-change protocols must determine a specific order in which tasks can be added or removed from the system [72]. Furthermore, in some lock-based schemes, certain operating system tables must be modified when mode changes occur [72].

1.2 Lock-Free Objects

In this dissertation, we consider an alternative approach to interprocess communication in real-time systems. In particular, we show that *lock-free* shared objects [22, 37, 52, 62] — i.e., objects that are not critical-section-based — are a viable alternative to lock-based schemes in such systems. Such objects are typically implemented using retry loops that are potentially unbounded. For example, the enqueue operation depicted in Figure 1.2 is implemented without using any locks and involves executing an unbounded repeat-until loop. Formally, a shared object implementation is *lock-free* if, for every operation by each task T, *some* operation is guaranteed to complete after a finite number of steps of task T, even if other tasks are delayed. Lock-free objects offer an attractive alternative to lock-based protocols because they eliminate priority inversion.

In this dissertation, we also consider an important special class of lock-free objects, namely wait-free objects. Wait-free objects are required to satisfy a strong form of lock-freedom that precludes all waiting dependencies among tasks, including potentially unbounded loops. More precisely, individual wait-free operations are required to be starvation-

```
type Qtype = record data: valtype; next: pointer to Qtype end
shared var Head, Tail: pointer to Qtype
private var old, new: pointer to Qtype; ret: boolean
                    addr: pointer to pointer to Qtype

procedure Enqueue(input: valtype)
   *new := (input, NULL);
   repeat old := Tail
      if Tail = NULL then addr := &Head else addr := &(old->next) fi
   until CAS2(&Tail, addr, old, NULL, new, new)


procedure Dequeue() returns *Qtype
   repeat old := Head;
          if old = NULL then return NULL fi;
          new := old->next;
          if old = Tail then ret := CAS2(&Head, &Tail, old, old, NULL, NULL)
          else ret := CAS2(&Head, &(old->next), old, new, new, NULL)
          fi
   until ret;
   return(old)
```

Figure 1.2: Lock-free queue implementation.

free. In contrast, lock-free objects guarantee only system-wide progress: if several tasks concurrently access such an object, then *some* task will complete its operation. Formally, an implementation is *wait-free* if, for every operation by each task T, *that* operation is guaranteed to complete after a finite number of steps of task T, even if other tasks are delayed. Note that the requirements of lock-free and wait-free objects preclude the use of locking: if some task is delayed for a long time while holding a lock to an object, then *no* other task can access that object.

As mentioned above, lock-free objects are typically implemented like the shared queue implementation depicted in Figure 1.2. In the lock-free implementation in Figure 1.2, an item is enqueued using a two-word compare-and-swap (CAS2) instruction[2] to atomically update a shared tail pointer and the "next" pointer of the last item in the queue. The CAS2

[2]The first two parameters of CAS2 specify addresses of two shared variables, the next two parameters are values to which these variables are compared, and the last two parameters are new values to assign to the variables if both comparisons succeed.

instruction is attempted repeatedly until it succeeds. Dequeue is implemented similarly. Note that the queue is never "locked" by any task.

Superficially, lock-free objects may seem to be unsuitable for hard real-time applications because concurrent lock-free accesses can interfere with one another; in fact, repeated interference can delay a task's completion indefinitely. Wait-free objects seem to be more appropriate for such systems because they have bounded access times. Unfortunately, satisfying the stronger progress guarantees required of wait-free objects comes at the expense of algorithmic overhead that cannot be justified for many shared objects.

Another potential disadvantage of lock-free objects is that their implementation typically requires strong primitives such as compare-and-swap (CAS). Strong primitives are provided by processors such as Intel's Pentium and Motorola's 68030, 68040, and PowerPC processors. Unfortunately, processors that are often used in embedded systems such as Motorola's 68HC11 or 68HC812 do not support strong primitives. If such primitives are not supported by the underlying hardware, then they must be "simulated", and the most obvious way to do so is by using lock-based protocols. However, this brings us back to problems associated with locking — problems that lock-free objects are designed to eliminate.

From a software engineering standpoint, lock-free objects have the reputation of being difficult to design and to verify as correct. As a result, even the design of simple objects such as linked lists requires a lot of creativity. To rectify this situation, many researchers have developed universal constructions that allow easy implementation of lock-free objects. A *universal construction* is used to automatically generate lock-free implementations of arbitrary objects from their sequential implementations. Universal constructions

were proposed first by Herlihy [36], and improved later by others [2, 7, 20, 37, 42]. To implement a lock-free object using a universal construction, a programmer first writes code for a sequential implementation of that object. This code is then embedded within a retry loop that is automatically generated by the universal construction. A lock-free implementation that is based on a universal construction is guaranteed to be correct if the sequential implementation of that object is correct.

One shortcoming of the universal constructions just cited is that they only allow a task to access one shared object at a time. In contrast, when tasks use a lock-based framework, they may access multiple objects simultaneously via nested critical sections. For example, using nested critical sections, a task can atomically dequeue an element from one shared queue and enqueue that element in another shared queue. To achieve similar functionality in a lock-free system, universal constructions have been proposed that allow a task to perform operations on multiple objects simultaneously [8, 73]. Unfortunately, these implementations entail high algorithmic overhead, and are therefore too expensive to be competitive with nested critical sections in real-time systems. To match the generality and flexibility provided by nested critical sections, an efficient framework for lock-free object sharing is required that allows simultaneous accesses to multiple objects.

To summarize, while lock-free objects are attractive from a real-time perspective because they eliminate priority inversion, their applicability to real-time systems has been limited because of an apparent lack of predictability, lack of hardware support, and lack of a general lock-free framework that is competitive with nested critical sections. In this dissertation, we show that these problems can be solved and that lock-free objects are a

viable alternative to lock-based object sharing in real-time systems. The main thesis to be supported by this dissertation is that

> *priority inversion is not fundamental to object sharing in uniprocessor hard real-time systems, and efficient and predictable use of lock-free objects in such systems can be achieved with little kernel and hardware support.*

All of the work presented in this dissertation supports the above thesis directly. Our contributions are as follows. First, for each of the task scheduling schemes that we consider, we derive expressions to predict whether a set of hard real-time tasks that access lock-free objects will meet their deadlines. Second, we present several original lock-free implementations for real-time systems that significantly improve on the time and space overhead costs associated with previously known implementations. In particular, we give software-based implementations of single- and multi-word compare-and-swap (MWCAS) primitives that facilitate the use of lock-free objects in real-time systems that lack adequate hardware support for strong primitives. Third, we provide a lock-free framework that supports accesses to multiple lock-free objects simultaneously. Such a lock-free framework provides functionality similar to that of nested critical sections. Finally, we present results from a set of experiments that compare the performance of lock-free objects with that of lock-based objects.

Most of the work presented in this dissertation assumes some basic knowledge of real-time systems. Therefore, before describing our contributions in more detail, we define some basic terminology and concepts pertaining to uniprocessor hard real-time systems and present some well-known schemes for scheduling tasks in such systems.

1.3 Background on Real-Time Systems

A real-time system can be abstractly visualized as a collection of tasks. A *task* is a sequential program that is invoked repeatedly. A single task invocation is called a *job*. The time at which a job arrives for execution is called its *release time*. The *deadline* of a job is an instant in real time by which that job must complete execution. The *relative deadline* of a task is the elapsed time between the release time of a job of that task and the deadline of that job. In this dissertation, we assume that the relative deadline of a task is at most the length of its task period.

A real-time task can be characterized as periodic, sporadic, or aperiodic, based on the job releases of that task. A task is *periodic* if and only if the interval between job releases is constant. The *period* of such a task is the length of the interval between successive job releases. A task is said to be *sporadic* if there exists a minimum (but not a maximum) separation between successive job releases. The release times of jobs of an *aperiodic* task are completely unrelated. A set of real-time tasks is said to be *synchronous* if the release time of the first job of all tasks coincide; otherwise, the task set is said to be *asynchronous*. A set of tasks is said to be *independent* if the tasks do not access any shared objects, and if there are no precedence constraints among the tasks. Tasks in a real-time system are multiprogrammed on a processor by means of a *scheduling scheme*.

A task set is said to be *schedulable* under a given scheduling scheme iff all jobs will meet their deadlines when scheduled under that scheme. A task set is said to be *feasible* iff there exists some scheduling scheme under which all jobs will meet their deadlines. A scheduling scheme is called *optimal* if it can schedule any feasible task set. Throughout this

dissertation, we assume that time is discrete, i.e., all release times and periods are integers. A task is said to *execute at time* t if it executes during the interval $[t, t+1)$.

1.3.1 Real-Time Scheduling Schemes

In this subsection, we consider some well-known priority-based preemptive scheduling schemes for uniprocessor real-time systems that are relevant to the work in this dissertation. A scheme is priority-based if, at every instant, the job scheduled for execution has the highest priority of all non-idle jobs. A scheme is said to be *preemptive* if a job can be preempted by another job during its execution.

Priority-based, preemptive scheduling schemes can be classified into *static-priority* and *dynamic-priority* schemes. Under static-priority scheduling, each task is assigned a distinct priority that does not change over time. Examples of such schemes include *rate monotonic* (RM) scheduling [60] and *deadline monotonic* (DM) scheduling [58]. Under *dynamic-priority* schemes, the priority of a task can vary over time. Examples of such schemes include earliest-deadline-first (EDF) scheduling [60] and least-laxity-first (LLF) scheduling [65]. In this dissertation, we only consider RM, DM, and EDF scheduling; these scheduling schemes are described using illustrated examples.

DM scheduling: Under the DM scheme, tasks with smaller relative deadlines are assigned higher priorities. The period of a task scheduled under this scheme is assumed to be at least as large as that task's relative deadline. When a set of tasks is synchronous and independent, the DM scheme is an optimal static-priority scheduling scheme [58], i.e., any task set that is schedulable under any other static-priority scheme is schedulable under the

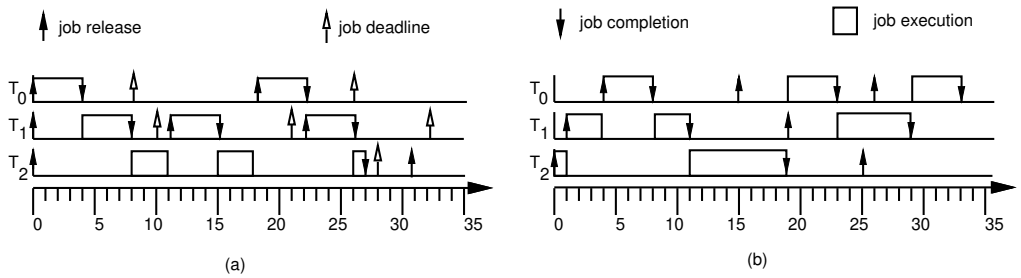


Figure 1.3: (a) DM scheduling (b) EDF scheduling.

DM scheme. The following example illustrates the working of the DM scheme.

Example 1.1: *Consider a set of synchronous periodic tasks* $\{T_0, T_1, T_2\}$ *scheduled under the DM scheme. Let the relative deadlines/periods/computation times of tasks* T_0, T_1, *and* T_2 *be* 8/18/4 *units,* 10/11/4 *units, and* 28/31/7 *units, respectively. All tasks are released simultaneously at time 0.*

The execution of the above task set is illustrated in Figure 1.3(a). *In this figure shaded up-arrows represent job releases, unshaded up-arrows represent job deadlines, and down-arrows represent job completions.*

Under the DM scheme, T_0, T_1, *and* T_2 *are assigned highest priority, intermediate priority, and low priority, respectively. This is because* T_0*'s relative deadline is smaller than that of* T_1, *which has a smaller relative deadline than* T_2. *Note that any job of* T_1 *can preempt any job of* T_2 *because all jobs of a given task have the same priority.* □

RM scheduling: RM scheduling constitutes a special case of DM scheduling in which the relative deadline of a task is equal to the period of that task. As in the DM scheme, tasks with smaller relative deadlines (periods) are assigned higher priorities.

EDF scheduling: The EDF scheme is a dynamic-priority scheme in which, at every instant, the job with the closest deadline is scheduled for execution. Under this scheme task periods need not be related to task deadlines, i.e., a task's period may be larger, smaller, or equal to its relative deadline. When tasks are independent, the EDF scheme is an optimal scheduling scheme [60] because it can successfully schedule any task set as long as the processor is not overloaded. The following example illustrate the working of the EDF scheme.

Example 1.2: *Consider a set of asynchronous periodic tasks* $\{T_0, T_1, T_2\}$ *scheduled under the EDF scheme. The release times/periods/computation times of tasks* T_0, T_1, *and* T_2 *are* 4/11/4 *units,* 1/18/6 *units,* 0/25/9 *units, respectively. The relative deadline of every task is equal to its period.*

The execution of the above task set is illustrated in Figure 1.3(b). *In this figure up-arrows represent both job releases and job deadlines, and down-arrows represent job completions.*

As illustrated in Figure 1.3(b), *the job executing on the processor at any instant has the closest deadline of all jobs that are available for execution at that instant. For example, at time 19,* T_2*'s executes on the processor because it has a deadline at time 25 and because the deadlines of jobs of tasks* T_0 *and* T_1 *are after time 25. Observe that, unlike the DM scheme, every job of task* T_1 *cannot necessarily preempt any job of task* T_2. *In Figure* 1.3(b), T_1*'s second job cannot preempt* T_2*'s job because the latter has an earlier deadline.* □

1.4 Our Contributions

In this section, we provide descriptions of the contributions of this dissertation.

1.4.1 Schedulability of Lock-Free Task Sets

From a real-time perspective, lock-free objects are useful because they eliminate priority inversion and deadlock with no underlying operating system support for object sharing. However, it is not apparent that such objects can be employed if tasks must adhere to strict timing constraints. In particular, repeated executions of lock-free retry loops can cause a lock-free operation to take an arbitrarily long time to complete. Nonetheless, we show that lock-free retry loops are bounded when tasks are scheduled under the RM, DM, or EDF schemes. For each of these schemes, we derive *scheduling conditions* — based on the worst-case resource requirements of tasks — that predict whether a set of tasks that access lock-free objects will meet their deadlines when scheduled under that scheme.

The derivation of our scheduling conditions hinges on determining a bound on the total wasted computation due to failed retry-loop executions. First, we derive such a bound based on the assumption that retry-loop costs[3] of different shared objects are uniform. (This assumption is equivalent to assuming that the cost of every retry loop equals that of the largest retry loop in the system.) Based on this bound, we derive scheduling conditions for RM, DM, and EDF scheduling. The bound that we estimate can be inaccurate if retry-loop costs of different lock-free objects can vary widely. Later, we relax the assumption that all retry-loop costs are uniform, and derive a much tighter bound on wasted computation using linear programming. Finally, we derive scheduling conditions for each of the RM, DM, and

[3]The cost of executing one iteration of a lock-free retry loop.

EDF schemes based on this tighter bound.

1.4.2 Eliminating Hardware Support for Strong Primitives

A possible criticism of lock-free objects is that they require hardware support for strong synchronization primitives such as CAS2. The fact that lock-free objects are typically implemented using strong synchronization primitives is no accident. In his seminal work [36], Herlihy showed that strong primitives are, in general, necessary for implementing lock-free objects. Herlihy's results are based upon a categorization of objects by "consensus number". An object has *consensus number* N if it can be used to solve N-process consensus, but not $(N+1)$-process consensus, in a lock-free manner. In the *consensus problem*, a set of N asynchronous processes, each with a given input value, must agree on a common output value equaling some process's input. Herlihy showed that an object with consensus number N is universal in a system of N processes. An object is *universal in a system of N processes* if it can be used to implement *any* object in a lock-free (or wait-free) manner in that system. These results give rise to a "hierarchy" of objects: each object is placed at the level of its consensus number and, in a system of N processes, it is impossible to construct a lock-free (or wait-free) implementation of a shared object at level N using any object from a lower level. Herlihy's results are significant because they imply that primitives with unbounded consensus numbers, such as CAS and CAS2, are necessary for implementing general-purpose lock-free objects, i.e., objects that may be accessed by an unbounded number of tasks. Thus, Herlihy's hierarchy implies that lock-free objects cannot be implemented in processors that do not support sufficiently strong primitives.

Nevertheless, we show that Herlihy's hierarchy collapses for uniprocessor real-

time systems because load and store primitives have unbounded consensus number in such systems. We also show that lock-free objects tailored for real-time systems are more efficient than objects designed for general asynchronous systems. Our results are based on a real-time task model that exploits certain characteristics of task interleavings that arise when priority-based schedulers are used. We develop this task model in Chapter 4 and then use this task model to simplify lock-free object implementations for real-time systems that use priority-based scheduling. The main results in Chapter 4 are as follows. First, we show that N-task consensus can be solved using only load and store instructions under the real-time task model. Second, we provide two CAS implementations based on load/store instructions and the memory-to-memory move instruction, respectively. Finally, we provide a wait-free MWCAS implementation that uses single-word CAS instructions.

1.4.3 A General Framework for Lock-Free Accesses

A key advantage of lock-based schemes is that they provide an easy-to-use framework for updating multiple shared objects via nested critical sections. For example, in order to transfer the contents of one shared queue to another, a task acquires locks for these two objects in a specific order, performs the transfer, and then releases the locks. In contrast, universal lock-free constructions that allow updates to multiple objects [7, 73] — the lock-free counterpart to nested critical sections — are not competitive with nested critical sections in real-time systems. Another advantage of using lock-based schemes is that, with little effort, any sequential object implementation can be converted to its concurrent version. In contrast, there are no known techniques for easily deriving object-specific lock-free implementations of an object from its sequential implementation. Hence, lock-free objects

are hard to design correctly.

In Chapter 5, we consider the problem of efficiently implementing multi-object lock-free operations and transactions, the lock-free counterpart to nested critical sections. Towards this end, we present a universal construction that supports lock-free multi-object operations, and that is based on the MWCAS primitive that we develop in Chapter 4. Our implementation is more efficient than previously known universal constructions that support multi-object operations because it is specifically tailored for real-time systems. As explained in Chapter 5, our universal construction can also be used to implement memory-resident real-time databases.

1.4.4 Experimental Results

We present experiments that demonstrate the effectiveness of the techniques described in Chapters 3, 4, and 5. Details of the experiments and the corresponding results are outlined in Chapter 6. First, we present a formal comparison of the various object-sharing schemes based on the scheduling conditions derived in Chapter 3. Second, we present detailed simulation studies that compare the performance of lock-free, wait-free, and lock-based schemes for various task sets and object access characteristics. Third, we provide a basis for comparing object-specific implementations under lock-based and lock-free schemes by providing typical access costs of common objects such as queues, stacks, linked lists, and read/write buffers, as measured from an actual implementation. Finally, we describe experiments we performed on a real-time videoconferencing system; these experiments provide empirical evidence demonstrating the utility of lock-free objects in practical systems. Our experiments involve comparing the performance of three versions of the videoconferencing

system. The first version uses lock-based schemes to implement shared queues accessed by tasks in the system; the second and third versions of the system employ lock-free and wait-free shared queues, respectively. We compare the performance of the different versions under different scheduling schemes and tabulate the results.

1.5 Organization of the Dissertation

The rest of this dissertation is organized as follows. A context for our work, along with related background material, is presented in Chapter 2. Chapter 3 describes scheduling conditions for hard real-time task sets that access lock-free objects. In Chapter 4, we present several algorithms and techniques to implement strong primitives in systems that lack adequate hardware support. An object-sharing framework is presented in Chapter 5 that supports operations on many arbitrary lock-free objects simultaneously. Experimental results that compare and contrast the behavior of lock-free and lock-based schemes are presented in Chapter 6. Finally, conclusions and a discussion of future directions for this research appear in Chapter 7. (This dissertation includes work that is based on previously published work [9, 11, 12, 13, 14, 70].)

Chapter 2

Background and Related Work

In this chapter, we provide background material on real-time systems and lock-free objects. First, we present scheduling conditions for various real-time scheduling schemes. Then, we describe various lock-based object-sharing schemes used in conventional real-time systems. Finally, we present background material on lock-free objects. In particular, we explain the relevance of consensus numbers and Herlihy's consensus hierarchy to lock-free implementations, and then provide a description of the correctness requirements of lock-free objects. We also present descriptions of universal constructions by Herlihy [36, 37] and by Anderson and Moir [7, 8] because they are relevant to this dissertation. This chapter concludes with a brief discussion of previous work on object-specific lock-free implementations.

2.1 Scheduling Conditions

Scheduling disciplines in real-time systems can be broadly classified into *preemptive* and *non-preemptive* schemes, depending on whether tasks can be preempted during their

execution. Under non-preemptive schemes, system calls to lock or unlock an object are not necessary because the scheduler provides mutual exclusion for free. Hence, object sharing is not an issue when non-preemptive schemes are used. For this reason, we consider only preemptive scheduling schemes in this dissertation. Specifically, we consider the RM, DM, and EDF schemes because (*i*) they are commonly used in practical real-time systems, and (*ii*) they are known to be optimal static- and dynamic-priority schemes for scheduling independent, synchronous task sets on uniprocessor real-time systems. A static-priority (dynamic-priority) scheme is said to be *optimal* iff it can successfully schedule any task set that is schedulable under any other static-priority (dynamic-priority) scheme. As mentioned earlier, *scheduling conditions* provide *a priori* guarantees on the schedulability of real-time task sets — hard real-time systems require such *a priori* guarantees. The rest of this subsection deals with scheduling conditions for the RM, DM, and EDF scheduling schemes.

2.1.1 Static-Priority Scheduling Conditions

In [60], Liu and Layland developed a sufficiency condition for determining the schedulability of a synchronous, periodic, and independent task set scheduled under the RM scheme. They showed that, under the RM scheme, schedulability is related to the achievable worst-case processor utilization. The derivation of their condition is based on the notion of a critical instant. A *critical instant* for a task is defined to be an instant at which a request for that task will have the longest response time. The reasoning behind their critical instant argument is as follows: if a task's job can meet its deadline when released at its critical instant, then every job of that task will meet its deadline. Liu and Layland showed that the critical instant of every task occurs at its first job release, when

it is released along with all higher-priority tasks. They also proved that a set of N periodic tasks is schedulable under the RM scheme if cumulative processor utilization of the tasks is at most $N * (2^{1/N} - 1)$, which converges to 0.69 as N tends to infinity. Hence, a periodic task set is schedulable if the worst-case cumulative processor utilization is at most 69%. Although Liu and Layland's scheduling condition is appealing because of its simplicity, it is not very accurate and can lead to under-utilization of the processor.

Later, Lehoczky, Sha, and Ding showed that the maximum achievable utilization under the RM scheme is close to 88% in practice [57], thereby showing that the condition developed in [60] is not sufficiently accurate. Based on a critical instant argument, Lehoczky et al. also developed an exact schedulability test for synchronous, independent, periodic task sets [57]. According to this test, such a task set is schedulable iff the following condition holds for every task T_i in the task set, where $0 \leq i < N$.

$$(\exists t : 0 < t \leq p_i : \sum_{j=0}^{i} \left\lceil \frac{t}{p_j} \right\rceil \cdot c_j \leq t) \tag{2.1}$$

Expression (2.1) is obtained by determining the worst-case cumulative demand placed by a set of tasks over an interval. In this expression, the worst-case computational requirement of task T_j and its task period are denoted by c_j and p_j, respectively. The summation on the left-hand side of the inequality denotes the cumulative demand placed on the processor by T_i and higher-priority tasks in the interval $[0, t]$. (The term $\lceil t/p_j \rceil$ denotes the exact number of job releases of T_j in that interval; each released job demands c_j units of processing.) The right-hand side of the above inequality denotes the maximum processor time available in the interval $[0, t]$.

As mentioned previously, the RM scheme requires that the relative deadline of

every task in the system be equal to its period. From a practical standpoint, this requirement is too restrictive because a task may have a relative deadline that is different from its period. For example, in order to ensure that sensor values are propagated to other tasks as soon as possible, a tracking task in a radar system may execute every 100 milliseconds but have a relative deadline of 30 milliseconds.

Leung and Whitehead [58] showed that the DM scheme is an optimal static priority scheme when each task's relative deadline is at most the length of its period. However, they did not provide scheduling conditions for the DM scheme. Later, in [18], Audsley et al. developed a necessary and sufficient scheduling condition for periodic and synchronous task sets scheduled under the DM scheme. The scheduling condition in [18] is similar to (2.1) with one difference: t ranges from 0 to l_i instead of p_i, where l_i is the relative deadline of task T_i. When task sets are asynchronous, the conditions for RM and DM scheduling are no longer both necessary and sufficient. This is because conditions that are necessary *and* sufficient for synchronous task sets to be schedulable are only sufficient for asynchronous task sets [21, 58].

2.1.2 Dynamic-Priority Scheduling Conditions

Under dynamic-priority schemes, a task's priority can change over time. Such schemes allow higher achievable processor utilization than static-priority scheme, but they entail higher run-time overhead compared to static-priority schemes. Of the existing dynamic-priority schemes, EDF and least-laxity-first (LLF) scheduling are the most well known [60, 65]. Under the LLF scheme, at every instant, the job with the smallest laxity is given highest priority. The *laxity* of a job J at time t is given by $t - d - c$, where d and

c denote J's deadline and its unfulfilled computation requirement, respectively. Theoretically, the LLF and the EDF scheduling schemes exhibit similar schedulability because their scheduling conditions are identical. However, when LLF scheduling is used in practice, significant run-time overhead is incurred due to excessive context switching. In contrast, under the EDF scheme, relatively few context switches occur at run-time. Hence, EDF scheduling is preferred to LLF scheduling in practical uniprocessor real-time systems. For this reason, EDF scheduling is the only dynamic-priority scheme considered in this dissertation.

Liu and Layland were also the first to derive scheduling conditions for independent, periodic, synchronous task sets scheduled under the EDF scheme [60]. They showed that a set of N periodic tasks is schedulable iff the cumulative processor utilization of all tasks in the system is at most one. Formally, their scheduling conditions is as follows.

$$\sum_{j=0}^{N-1} \frac{c_j}{p_j} \leq 1 \tag{2.2}$$

In the derivation of (2.2), the relative deadline of every task T_i is assumed to be equal to its task period. As mentioned earlier, this assumption is too restrictive because it does not hold for many task sets in practice. We now consider conditions that determine the schedulability of an asynchronous task set under the EDF scheme, when tasks' relative deadlines do not equal their periods.

Because EDF is an optimal scheduling scheme [27], the problem of determining the schedulability of a task set under the EDF scheme is equivalent to the problem of determining the feasibility of that task set. (Recall that an optimal scheduling scheme can schedule any feasible task set.) The feasibility problem was studied by Baruah, Howell, and Rosier [21], who developed necessary and sufficient conditions for determining the feasibility

of an asynchronous set of periodic, independent tasks. They proved that (2.2) is a necessary (but not sufficient) condition for scheduling tasks under the EDF scheme, when task periods are different from their deadlines. We now state the sufficient condition developed in [21] for determining the feasibility of a set of N periodic tasks. The condition below is also a sufficient condition for schedulability of that task set under the EDF scheme.

$$(\forall t : 0 \leq t : \sum_{j=0}^{N-1} c_j \cdot max(0, \left\lfloor \frac{t - s_j - l_j}{p_j} \right\rfloor) \leq t) \tag{2.3}$$

In (2.3), s_i and l_i denote task T_i's first job release point and its relative deadline, respectively, where $s_i \geq 0$. (Other terms in (2.3) are defined in the glossary.) In the above expression, the left-hand side of the inequality denotes the maximum possible demand due to job releases of all tasks in the interval $[0, t]$, and the right-hand side denotes the available processor time in that interval. As stated above, the expression cannot be evaluated because t is unbounded. Fortunately, it is shown in [21] that the above expression only needs to be checked for values of t that are less than or equal to $2 \cdot L + max_{1 \leq i \leq N}(s_i) + max_{1 \leq i \leq N}(l_i)$, where L is the least common multiple (LCM) of the task periods. Thus, in the worst-case, determining the feasibility of a task set is exponential in the task periods. However, if an upper bound on the processor utilization is known, then Baruah et al. showed that the values of t for which the above expression must be evaluated lies in a much smaller range. Specifically, if processor utilization U lies in the interval $(0, 1)$, then t in the above condition ranges from 0 to M, where $M = \frac{U}{1-U} max_{1 \leq i \leq N}(p_i - l_i)$.

```
Task Object_Monitor
begin
    rendezvous(any task T_i);
    ⟨ Access shared object ⟩
    rendezvous(T_i)
end
```

Figure 2.1: Implementation of a monitor task using using *rendezvous* primitive.

2.2 Lock-Based Object Sharing

The problem of task synchronization and interprocess communication in hard real-time systems was first studied by Al Mok. In his dissertation [65], Mok proved a number of fundamental results that highlighted the complexity of lock-based real-time object sharing. In particular, he proved that the problem of deciding whether a schedule exists for a set of periodic tasks that only use semaphores to enforce mutual exclusion is NP-hard in the strong sense. As a first step towards developing a practical real-time object-sharing scheme, Mok prescribed the *deterministic rendezvous* approach and the *kernelized monitor* approach, both of which are outlined below.

Under the deterministic rendezvous model, tasks communicate with one another using ADA-like rendezvous primitives[1] and are scheduled using the EDF scheme. Shared objects under this model are implemented via special monitor tasks as illustrated in Figure 2.1. To access a shared object, a task T_i executes the following code sequence "*rendezvous*(*Object_Monitor*); *rendezvous*(*Object_Monitor*)". To enforce mutual exclusion, the run-time scheduler modifies task deadlines using information from a precomputed database

[1] Using such primitives, task T_i executes *rendezvous*(T_j) in order to synchronize with task T_j, and vice-versa. If T_i executes that primitive before T_j, then it waits until T_j executes *rendezvous*(T_i).

— this ensures that a task is never preempted while executing within a monitor. However, this approach is not very flexible because the database has to be recomputed every time the task set is modified. Furthermore, this technique suffers two major shortcomings: first, it places restrictions on the length of the task periods of communicating tasks; second, the kernel does not differentiate between different critical sections, i.e., a single logical shared object is assumed.

Under the kernelized monitor model, the operating system enforces mutual exclusion by allocating processor time to tasks in time quantums that are larger than the longest critical section. At time t, the run-time system maintains a database of "forbidden regions"[2] in the interval $[kL, (k+1)L]$, where L is the LCM of the task periods and k is some integer such that $kL \leq t \leq (k+1)L$. This database is recycled at run-time every L units of time. This database is used by the scheduler to make scheduling decisions at run-time. In particular, the scheduler allocates a time quantum to the task with the earliest deadline at time t iff t does not lie within a forbidden region. However, this technique works well only if the critical sections are small and the LCM of task periods is not very large. The main shortcoming of these techniques is that they lack the flexibility to provide a general framework for object sharing in real-time systems.

Lampson and Redell were the first to recognize priority inversion as a problem associated with using lock-based objects in priority-based systems [55]. A *priority inversion* occurs when a task is delayed by a lower-priority task that is inside a critical section. This problem is illustrated in Figure 2.2(a). In Figure 2.2, up-arrows and down-arrows indicate task invocations and task completions, respectively. Tasks are indexed in the order of

[2]Tasks cannot access shared resources during these forbidden regions.

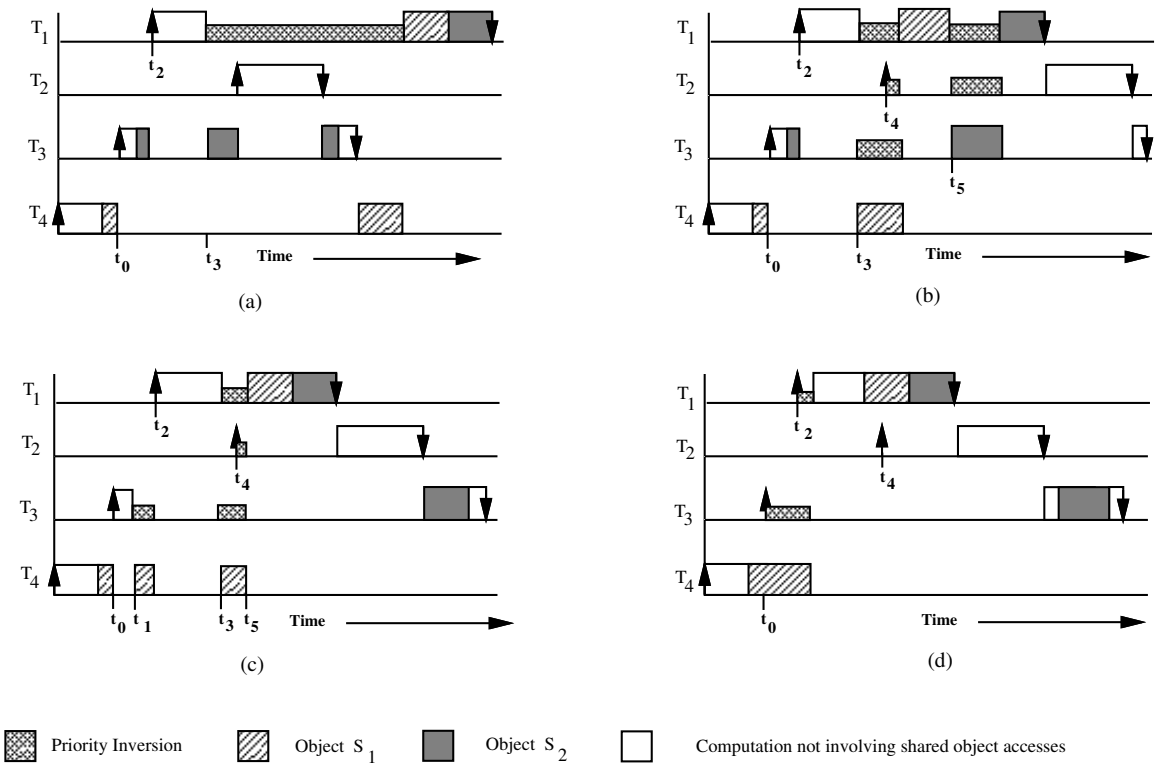


Figure 2.2: Illustrated example depicting (a) priority inversion, (b) the priority inheritance protocol, (c) the priority ceiling protocol, and (d) the stack resource policy.

decreasing priorities; tasks T_1 and T_4 have the highest and lowest priorities, respectively. Tasks T_3 and T_4 access S_2 and S_1, respectively; task T_1 accesses both objects; T_2 accesses neither.

In Figure 2.2(a), after they acquire locks to objects S_2 and S_1, T_4 and T_3 are preempted at times t_0 and t_2, respectively. At time t_3, T_1 is blocked from accessing object S_1; it is forced to relinquish the processor to T_3. However, T_1 is delayed from accessing S_1 for a lengthy duration because T_4's object access is delayed until jobs of tasks T_2 and T_3 complete execution. Thus, task T_1 can potentially miss its deadline due to a lower-priority task's object access. Clearly, priority inversion should be avoided, or its ill-effects should be minimized.

Priority inversion manifests itself in the form of blocking factors in the scheduling conditions for priority-based schemes. The *blocking factor* associated with task T_i represents the longest duration for which priority inversion can occur during the execution of T_i or higher-priority tasks. The performance of lock-based schemes for object sharing is measured in terms of their ability to minimize blocking factors. The rest of this section deals with mechanisms for reducing blocking factors in lock-based schemes.

Lampson and Redell were the first to solve the priority inversion problem using kernel support by associating with each resource R the priority of the highest-priority task that may lock that resource [55]. The priority of any task that accesses resource R is elevated to the priority associated with that resource. However, Lampson and Redell's work was targeted for the Mesa programming language; they did not provide a general solution to the priority inversion problem. Later, Sha, Rajkumar, and Lehoczky considered this problem from a broader perspective of accessing lock-based objects in general priority-based systems [71]. Based on ideas presented in [55], they developed several mechanisms for reducing priority inversion. Two such mechanisms were developed by them: the *priority inheritance protocol* (PIP) and the *priority-ceiling-protocol* (PCP) [71, 69]. Under the PIP, if a task T is blocked by a lower-priority task T' that is accessing some object, then T' executes at T's priority level for the remainder of its object access, i.e., T' inherits T's priority. Figure 2.2(b) illustrates the working of the PIP. As in Figure 2.2(a), tasks T_3 and T_4 are preempted during their respective object accesses at times t_0 and t_2, respectively. At time t_3, task T_1 is prevented from accessing object S_1; it is forced to relinquish the processor to T_4 which holds the lock to S_1 at that time. Before T_4 resumes execution, the

kernel raises its priority to be equal to that of T_1. We say that T_4 inherits T_1's priority at time t_2. Observe that the priority inheritance mechanism prevents T_2 from preempting T_4 at time t_4. Later, at time t_5, T_1 is blocked for the second time by task T_3, which inherits T_1's priority and completes its object access before relinquishing the processor to T_1. Although the PIP eliminates unbounded priority inversion, it can still suffer large blocking factors resulting from *multiple blocking*. Multiple blocking occurs when a task is blocked multiple times during an invocation. In the example in Figure 2.2(b), T_1 is blocked before each object access.

In order to eliminate multiple blocking, Sha et al. enhanced the basic priority inheritance mechanism with the notion of a *priority ceiling* to create the PCP. The priority ceiling of a shared object is the priority of the highest-priority job that may access that object. Under the PCP, a task acquires a lock to a shared object iff its priority is greater than the maximum priority ceiling of any locked shared object, as illustrated in Figure 2.2(c). In this example, the priority ceiling of both shared objects equals T_1's priority. At time t_1, T_3 is blocked from accessing S_2 because its priority is smaller than the ceiling of object S_1, which is locked by T_4. The PCP ensures that T_4 inherits T_3's priority before it resumes execution at time t_1. Later, before T_4 can complete its object access, it is preempted by T_1 at time t_2. However, because T_1's priority is not greater than priority ceiling of object S_1, it is forced to relinquish the processor to T_4 at time t_3. Task T_4 then inherits T_1's priority and successfully completes its object access.

Note that, unlike the PIP, each job is blocked at most once under the PCP. In particular, the duration for which a job of task T is blocked is bounded by the longest critical

section that belongs to some lower-priority task that accesses an object also accessed by T. However, as evident from this example, the PCP can result in additional context switches. In Figure 2.2(c), extra context switches occur at times t_0, t_1, t_2, and t_3. If context switches are expensive, the PCP can result in loss of predictability because the currently-known scheduling analysis [71] for the PCP ignores the effect of these extra context switches. Furthermore, these additional context switches can significantly increase blocking factors.

The *stack resource policy* (SRP) [19] eliminates additional context switches by delaying the execution of a task until all required resources become available. Unlike the PIP/PCP, the SRP differentiates between priority levels and *preemption levels*. The notion of preemption levels is based on the relative deadlines of tasks; it is independent of priority levels. In particular, if a task T has a closer relative deadline than task T', then T's preemption level is higher than that of T'. Preemption levels are derived from the fact that, in many common real-time scheduling schemes, a task T cannot preempt another task T' if its relative deadline is at least that of T'. Examples of such schemes that allow preemption levels are RM, DM, and EDF scheduling. Because preemption levels are independent of the underlying scheduling scheme, the SRP can be used for both static- and dynamic-priority schemes; in contrast, the PCP and PIP can only be used for static-priority schemes.

The SRP introduces the notion of *preemption ceilings*, akin to the notion of a priority ceiling in the PCP. The SRP ensures that a task T begins execution only when all higher-priority tasks are idle and when no object is locked by any task with a preemption level higher than that of T. In Figure 2.2(d), task indices are inversely related to their preemption levels; T_1 and T_4 are at the highest and lowest preemption levels, respectively.

The preemption ceilings of S_1 and S_2 are equal to T_1's preemption level. In this example, task T_3 does not start executing at t_0 because its preemption level is not greater than S_1's preemption ceiling; T_1 does not start executing at time t_2 for the same reason. By delaying a job's execution, the SRP ensures that the job never blocks during its execution. The SRP successfully addresses many of the problems associated with the PCP: first, it minimizes additional context switching overhead; second, it supports general semaphores.

In [44], Jeffay developed another protocol for lock-based object sharing under the EDF scheme, namely the *earliest-deadline-first with dynamic-deadline-modification* (EDF/DDM) protocol. The working of the EDF/DDM scheme is not illustrated here because it imitates the behavior of the SRP under EDF scheduling. Under the EDF/DDM scheme, a deadline ceiling is associated with every shared object S. The *deadline ceiling* of an object is equal the smallest relative deadline of any task that accesses S. Before accessing an object S, task T executes a *acquire_resource* system call that modifies T's deadline so that T cannot be preempted during its access by any other task that accesses S. Upon completing its access T performs a *release_resource* system call that restores its original deadline.

2.2.1 Static-Priority Scheduling Conditions

When static-priority schemes are used, scheduling conditions for task sets that access lock-based objects are similar to those for independent tasks, with one difference: conditions of the former type have an additional blocking factor which gives the maximum duration of time for which a task can be blocked by lower-priority tasks. Sha, Rajkumar, and Lehoczky [71] developed a sufficient condition for determining the schedulability of a

set of periodic tasks that access lock-based objects and that are scheduled under the RM scheme. According to their condition, a task set is schedulable under the RM scheme if the following condition holds for every task T_i.

$$(\exists t : 0 < t \leq p_i : B_i + \sum_{j=0}^{i} \left\lceil \frac{t}{p_j} \right\rceil \cdot c_j \leq t) \tag{2.4}$$

Observe that the only difference between the above condition and Condition (2.1) for independent tasks is the blocking factor B_i on the left-hand side of the inequality. The term B_i denotes the maximum amount of execution time claimed by lower-priority tasks during the execution of a job of task T_i. For any given task set, the B_i term is usually much larger for PIP than it is for PCP and SRP. This is because of the multiple blocking problem explained earlier. In contrast, B_i is at most the length of one critical section, under the PCP and the SRP. The scheduling condition for lock-based task sets scheduled under the DM scheme is similar to (2.4) with one difference: t ranges up to l_i rather than p_i.

2.2.2 Dynamic-Priority Conditions

The least predictable and least efficient mechanism for object sharing under dynamic-priority schemes is the *dynamic priority ceiling protocol* (DPCP) developed by Chen and Lin [26]. The DPCP is a straightforward adaptation of the PCP for dynamic-priority scheduling. One major difference between these schemes is that the DPCP modifies priority ceilings of objects at run-time. Chen and Lin developed the following scheduling condition for the DPCP [26].

$$\sum_{j=0}^{N-1} \frac{c_j + B_j}{p_j} \leq 1 \tag{2.5}$$

In the above equation, B_j represents the maximum amount of blocking that can

experienced by a job of task T_j. The DPCP suffers from a major shortcoming: maintaining dynamic priority ceilings entail significant overhead at run-time. Furthermore, the large blocking factors in Condition 2.5 result in poor predicted schedulability. These problems were later addressed and solved in the SRP developed by Baker [19] and the EDF/DDM protocol developed by Jeffay[43, 44]. These schemes perform much better than DPCP because they result in optimal blocking factors. Although the implementation of the SRP and EDF/DDM are identical, the scheduling conditions developed for them are different.

Baker developed the following sufficient condition for determining the schedulability of a task set under the EDF scheme, when tasks' relative deadlines are not equal to their periods [19].

$$(\forall k : 0 \leq k \leq N-1 : \frac{B_k}{l_k} + \sum_{j=0}^{k} \frac{c_j}{l_j} \leq 1) \tag{2.6}$$

The term B_k/l_k is akin to the blocking factor in (2.4). It denotes the fraction of processor utilization wasted due a task at a lower-preemption level executing at the level of task T_k. In [19], Baker also developed a scheduling condition for a set of tasks that have relative deadlines equal to their periods. This condition is similar to (2.6), with one difference: the l_k and l_j terms are replaced by p_k and p_j, respectively. Baker's condition is also applicable to the DPCP, and is a significant improvement over Chen and Lin's condition [26].

In [44], Jeffay developed an exact schedulability test for asynchronous, sporadic tasks scheduled under the EDF/NPD scheme and that access objects using the EDF/DDM scheme. (As explained earlier, if tasks are periodic, then the condition in [44] is sufficient but not necessary.) According to this test, such a task set is schedulable iff the following

conditions hold.

$$\sum_{j=0}^{N-1} \frac{c_j}{p_j} \leq 1 \tag{2.7}$$

$$(\forall i, t : 0 \neq i \leq N-1 \ \wedge \ x_i \neq 0 \ \wedge \ p_{x_i} < t < p_i : c_i + \sum_{j=0}^{i-1} \left\lfloor \frac{t-1+p_j-l_j}{p_j} \right\rfloor c_j \leq t) \tag{2.8}$$

The terms x_i and p_{x_i} denote the index of the shared object that has the smallest deadline ceiling of any object accessed by T_i and the deadline ceiling of that object, respectively. (x_i equals zero if T_i does not access any shared object.) Condition (2.7) states that the processor utilization is at most one. The left-hand side of the inequality in Condition (2.8) denotes the maximum total demand, in any interval of length t, due to tasks with relative deadline less than or equal to T_i's relative deadline. The right-hand side of the inequality denotes the available processor time in that interval. The above conditions are also applicable to the DPCP and the SRP schemes, and are significantly more accurate than Conditions (2.5) and (2.6).

2.3 Accounting for System Overhead Costs

In the experiments described in Chapter 6, we determine the schedulability of a set of tasks in a videoconferencing system. The tasks in the system are scheduled under the EDF/NPD scheme or the DM scheme. However, to determine the schedulability of that task set, we do not use Conditions (2.7) and (2.8) or Condition (2.4) directly, because these conditions do not account for system overhead costs such as context switch costs and interrupt handling overhead costs. We now discuss previous work on incorporating system overhead costs into scheduling conditions for periodic task sets.

Context Switching Overhead Costs: In [48], Katcher, Arakawa, and Strosnider developed a RM scheduling condition that accounted for context-switching overhead costs. They enhanced Condition 2.4 by including an additional term to account for context switching overhead costs. In [48], Katcher et al. derived the following expression for $CS_i(t)$ the total additional demand due to context switches in T_i or higher-priority in an interval of length t.

$$CS_i(t) \leq \sum_{j=0}^{i} \left\lceil \frac{t}{p_j} \right\rceil (C_{save} + C_{rest}) \tag{2.9}$$

In (2.9), the worst-case time to save the context and to restore the context of any task is denoted by C_{save} and C_{rest}, respectively. The reasoning behind this expression is as follows. In any interval of length t, $\lceil t/p_j \rceil$ jobs of T_j are released, and the each release of a job of T_j entails $C_{save} + C_{rest}$ units of context switching cost in the worst case: C_{save} units to save the context of the currently executing task and C_{rest} units to restore the context a lower-priority job when T_j's job completes. Note that this expression is pessimistic because a lower-priority job need not be executing when a job of task T_j is released; the processor may be idle or a higher-priority job may be executing. The expressions in [48] were derived only for the RM scheme, but they also hold for the DM and EDF schemes.

Interrupt Handling Overhead Costs: The problem of accounting for interrupt handling overhead costs was first addressed by Jeffay and Stone in [46]. In their work, Jeffay and Stone focus on a system in which interrupts arrive periodically and in which a set of periodic tasks are scheduled under the EDF scheme. They considered a system consisting of a set of M interrupts $\{I_0, \ldots, I_{M-1}\}$ and N tasks $\{T_0, \ldots, T_{N-1}\}$, and derived the following expression that places a bound on $IH(t)$ the total demand due to handling interrupts in

any interval of length t.

$$IH(t) \leq \sum_{j=0}^{M-1} \left\lceil \frac{t}{v_j} \right\rceil e_j \tag{2.10}$$

In (2.10), e_j and v_j denote the worst-case execution cost and the period of interrupt I_j, respectively. The reasoning behind (2.10) is as follows: in any interval of length t, there are at most $\lceil t/v_j \rceil$ interrupts of type I_j, each of which requires e_j units of computation. Although, the work in [46] considers only the EDF scheme, it is also applicable to static-priority schemes.

Under the RM scheme, a scheduling condition that accounts for interrupt-handling and context-switching overhead costs can be achieved by including the terms $IH(t)$ and $CS_i(t)$ in the left-hand side of the inequality in Condition (2.4). (A similar condition for the DM scheme looks identical that for the RM scheme with one small difference: t ranges from 0 to l_i instead of 0 to p_i.)

In [77], Stone adapted the conditions in [46] for EDF/NPD scheduling to account for context switching and interrupt handling costs, when tasks access tasks under the EDF/DDM scheme; these conditions are stated below.

$$\sum_{j=0}^{M-1} \frac{e_j}{v_j} + \sum_{j=0}^{N-1} \frac{c_j + C_{save} + C_{rest}}{p_j} \leq 1 \tag{2.11}$$

$$(\forall i, t : 0 \neq i \leq N-1 \ \wedge \ x_i \neq 0 \ \wedge \ p_{x_i} < t < p_i : CS_i(t) + IH(t) + c_i + \sum_{j=0}^{i-1} \left\lfloor \frac{t-1+p_j-l_j}{p_j} \right\rfloor c_j \leq t) \tag{2.12}$$

2.4 Lock-Free Object Implementations

Lock-free shared objects have been proposed as viable alternatives to lock-based objects in general asynchronous systems by various researchers [1, 3, 4, 36, 37, 49, 54, 67, 68, 74]. In this dissertation, we use the term "lock-free" to refer to object implementations based on an unbounded retry loop structure like that depicted in Figure 1.2.[3] Some lock-free implementations do not adhere to this characterization. For example, there exists an important special class of lock-free implementations known as *wait-free* implementations [36, 37, 67, 54] in which operations must satisfy a strong form of lock-freedom that precludes all waiting dependencies among tasks, including potentially unbounded retry loops.

Formally, a shared object implementation is *lock-free* iff the following holds: if several objects access a shared object concurrently and a subset of the tasks are delayed,[4] an operation by some non-delayed task is guaranteed to complete in a bounded number of its own steps. Researchers have also proposed the use of wait-free objects, which constitute a special case of lock-free objects. Wait-free objects are required to satisfy a strong form of lock-freedom that precludes all waiting dependencies among tasks, including potentially unbounded loops. An implementation is said to be *wait-free* iff it is lock-free and every task accessing that object is guaranteed to complete its execution in a bounded number of its own steps. Note that the above definitions preclude the use of mutual exclusion in the implementation of lock-free objects, because if a task is permanently delayed within the critical section, then *no* task completes its operation. Also, note that the above definitions imply that individual wait-free operations are required to be starvation-free. In contrast,

[3] Some authors use the term "nonblocking" to refer to such implementations.

[4] In a uniprocessor system, such delays are typically caused when a task is preempted during its operation.

lock-free objects guarantee only system-wide progress: if several tasks concurrently access such an object, then *some* access will eventually complete.

In the remainder of this section, we introduce linearizability — the correctness condition used to verify lock-free implementations — and describe previous research on such implementations.

2.4.1 Linearizability

An operation on lock-free objects typically requires many atomic statements that are executed over an interval of time. Because tasks can invoke concurrent operations, two or more of these intervals may overlap. This gives rise to a partial order over invocations: if one invocation completes before another invocation starts, then the former precedes the latter in the partial order, and if two invocations overlap, then they are not ordered. In order for a shared object implementation to be useful, each operation should "appear" to the invoking tasks to take effect instantaneously at some point during its execution. The formal correctness condition used to ensure this is *linearizability* [39]. Linearizability requires that the partial order that arises from any series of invocations on an object can be extended to a total order in such a way that the values returned by the invocations in the total order are consistent with the sequential semantics of the implemented object. In particular, when an object implementation is linearizable, operations performed on that object can always be serialized in some order depending on task interleavings at run-time.

Consider a lock-free shared queue based on the implementation given in Figure 2.3. (Other than the line numbers, the implementation in Figure 2.3 is identical to that in Figure 1.2.) The lock-free enqueue operation in this implementation linearizes to the CAS2

```
type Qtype = record data: valtype; next: pointer to Qtype end
shared var Head, Tail: pointer to Qtype
private var old, new: pointer to Qtype; addr: pointer to pointer to Qtype

procedure Enqueue(input: valtype)
1:  *new := (input, NULL);
    repeat
2:    old := Tail
3:    if Tail = NULL then addr := &Head
4:    else addr := &(old->next) fi
5:  until CAS2(&Tail, addr, old, NULL, new, new)

    procedure Dequeue() returns *Qtype
      repeat
6:      old := Head;
7:      if old = NULL then return NULL fi;
        new := old->next;
8:      if old = Tail then
9:        addr := &Tail;
10:       ret := CAS2(&Head, addr, old, old, NULL, NULL)
        else
11:       addr := &(old->next);
12:       ret := CAS2(&Head, addr, old, new, new, NULL)
        fi
      until ret;
13:   return(old)
```

Figure 2.3: Lock-free queue implementation.

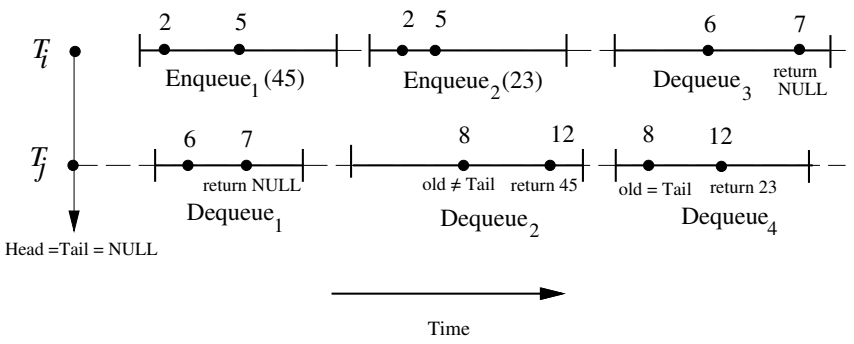


Figure 2.4: Example interleavings of lock-free enqueue and dequeue operations.

primitive (line 5), i.e., the enqueue operation takes effect iff the CAS2 operation is successfully executed. The dequeue operation linearizes to one of three statements, depending on the number of elements in the queue when the current head of the queue is read in line 6. The dequeue operation linearizes to: (*i*) line 6, if the queue is empty; (*ii*) line 10, if the queue contains one element; (*iii*) line 12, if the queue contains more than one element.

Suppose that two tasks T_i and T_j access a common shared queue that is im-

plemented as shown in Figure 2.3. Figure 2.4 depicts some possible interleavings of the operation steps of tasks T_i and T_j. In the figure, only statement numbers relevant to this example are shown and the queue is assumed to be empty initially. In the figure, operations Enqueue_1 and Dequeue_1 overlap and their steps are interleaved. These operations linearize to statements 5 and 6, respectively. However, T_i executes statement 5 *after* T_j executes statement 6. Hence, the operations appear as if they are executed in the order "Dequeue_1; $\text{Enqueue}_1(45)$". Observe that the return values of these operations correspond to the return values one would expect if the operations are sequentially executed in the same order. Similarly, the linearization statement of operation $\text{Enqueue}_2(23)$ (Dequeue_4) is executed before that of Dequeue_2 (Dequeue_3). The operations in the example appear as if they are executed in the same order in which their linearization statements are executed, i.e., the operations produce return values as if they were executed in the following serial order: "Dequeue_1; $\text{Enqueue}_1(45)$; $\text{Enqueue}_2(23)$; Dequeue_2; Dequeue_4; Dequeue_3".

2.4.2 The Consensus Hierarchy

In the *consensus problem*, a set of N asynchronous processes, each with a given input value, must agree on a common output value equaling some process's input. Loui and Abu-Amara [61] showed that the N-task consensus problem cannot be solved in a wait-free manner for $N > 1$ in an asynchronous system using load and store instructions.

Herlihy later extended these results to other primitives by classifying each primitive according to its consensus number [35, 36]. The *consensus number* of a primitive is the maximum number of tasks for which a wait-free (or lock-free) consensus algorithm exists that relies only on that primitive, and load/store instructions. Herlihy showed that for each

Consensus Number	Object
1	read/write registers
2	test&set, swap, fetch&add, queue
⋮	⋮
$2n-2$	n-register assignment
⋮	⋮
∞	memory-to-memory move and swap, compare&swap, load-linked/store-conditional

Table 2.1: Herlihy's consensus-number hierarchy.

$N \geq 1$, there exists a a primitive with consensus number N. More importantly, he also showed that an object with consensus number N is universal in a system of N tasks. A primitive is *universal in a system of N tasks* if it can be used to implement *any* object[5] in a wait-free (or lock-free) manner in that system. These results give rise to a "hierarchy" of objects: each object is placed at the level of its consensus number and, in a system of N tasks, it is impossible to construct a wait-free implementation of a shared object at level N using any object from a lower level. Herlihy's hierarchy is shown in Figure 2.1. Herlihy's hierarchy implies that any object can be implemented in a wait-free manner for any number of tasks in a system that supports instructions like CAS, CAS2, or LL/SC. (Formal definitions of these primitives are given in Figure 2.5) It also implies that, in general, lock-free objects cannot be implemented using only load and store instructions, even when they are accessed only by two tasks.

[5]Note that there is a subtle difference between the terms "object" and "primitive". Load and CAS primitives can be used to access and modify a shared variable, whereas a CAS object is a shared object that implements a shared variable that can be modified using a CAS operation provided by that implementation. Conceptually, a shared variable that is accessed (modified) using a load (CAS) primitive can be considered as a CAS object.

```
CAS(X, v, w)          ≡ if X = v then X := w; return true else return false fi
CAS2(X, Y, v, w, x, y) ≡ if X = v ∧ Y = w then X := x; Y := y; return true
                         else return false fi
MWCAS(N, X, v, w)     ≡ for i := 0 to N − 1 do
                             if X[i] ≠ v[i] then return false fi
                         od;
                         for i := 0 to N − 1 do X[i] := w[i] od;
                         return false
LL(X)                 ≡ valid_X[p] := true; return X
SC(X, v)              ≡ if valid_X[p] then
                             X := v;
                             for i := 0 to N − 1 do valid_X[i] := false od;
                             return true
                         else
                             return false
                         fi
```

Figure 2.5: Equivalent atomic code fragments for common instructions used. Fragments for LL and SC are for task T_p. $valid_X$ is a shared array of booleans associated with variable X. i is a private variable of task T_p. N is the total number of tasks. The semantics of SC are undefined if task T_p has not previously executed a LL instruction.

2.4.3 Universal Constructions of Lock-Free Objects

In recent years, several groups of researchers have presented methods for automatically "transforming" sequential object implementations into lock-free ones [20, 35, 36, 37, 73]. These methods are called *universal constructions*. A universal construction relieves the object designer of the need to reason about concurrency, thereby greatly simplifying the task of providing a correct lock-free implementation for a particular shared object. We now outline two universal constructions that are relevant to the work in this dissertation: one by Herlihy [37], and another by Anderson and Moir [8].

Herlihy's Construction

In [37], Herlihy presents two universal constructions for "small" and "large" objects. Herlihy's lock-free construction for small objects maintains $N + 1$ copies of the implemented object — a "current" copy, and a "working" copy for each task. As shown in Figure 2.6, a shared pointer records which copy contains the current value of the object. In order to perform an operation, a task T_i reads the pointer to identify the current copy, and then copies the contents of the current copy to T_i's local copy. Then, T_i performs the desired operation by executing its sequential code on the local copy. Finally, T_i attempts to modify the shared pointer so that it points to T_i's local copy, thereby making T_i's copy current.

In Figure 2.6, tasks T_i and T_j are both attempting to perform an operation. If T_i and T_j read the same object copy, then both perform their operations on the same object value. If T_i subsequently modifies the shared pointer to point to T_i's local copy, and then T_j modifies the pointer to point to T_j's local copy, then the effect of T_i's operation will be lost. In order to avoid this condition, the pointer is read and written with two special instructions, load-linked (LL) and store-conditional (SC). The LL instruction simply returns the value of the pointer. The SC instruction, when executed by task T_i, writes a new value to the pointer only if no other SC instruction has successfully modified the pointer since T_i's last LL on the pointer. Otherwise, the SC does not modify the pointer, and returns false. Thus, in the scenario described above, T_j's SC would return false, thereby informing T_j that its operation had had no effect. In that case, T_j can restart the entire operation. Note that T_j restarts its operation only if some other task successfully completes an operation.

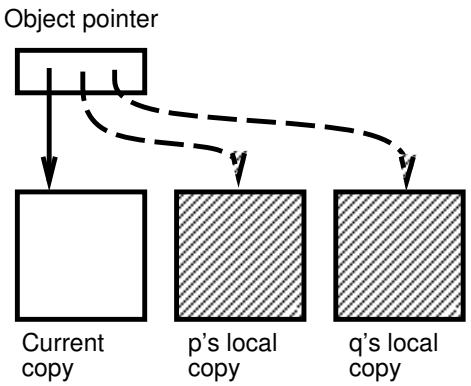


Figure 2.6: Implementation of Herlihy's small object constructions.

Thus, the implementation is lock-free.

Objects that are "small" can be efficiently implemented using the above technique. However, if the object in question is "large", then the overhead associated with making a private copy of the object could be very high. To overcome this problem, Herlihy proposes that a large object be broken into "fragments" linked together by pointers in order to reduce the copying overhead. Herlihy's lock-free implementation suffers from two shortcomings. First, the programmer has to determine the required fragmentation based on the semantics of the object. The programmer also has to explicitly determine how the copying is done. The burden of proving the correctness of an implementation is placed on the programmer. Secondly, this approach reduces copying overhead only for certain objects such as heaps, but not for other objects such as queues. To see why this is so, consider a queue that is implemented as a linked list of blocks. An enqueue operation on such a queue must copy the whole queue because, in order to link in a new block, the "next" pointer of the last block must be modified, which in turn necessitates modifying the next-to-last block, and so

on. As described below, these drawbacks are addressed in Anderson and Moir's universal construction for large objects.

Large-Object Construction

Anderson and Moir's large-object construction [7] fragments a large object into blocks; this is similar to Herlihy's approach for large object constructions. The construction in [7] differs from Herlihy's in two aspects: it is array-based rather than pointer-based, and it uses long-LL and long-SC primitives[6] instead of single-word LL/SC primitives. long-LL and long-SC primitives are not provided by any machine; they are implemented using single-word LL/SC primitives. As illustrated in Figure 2.7, a large object is viewed as a long array *MEM* that is fragmented into S blocks — where each block consists of B words — accessible by a bank of pointers *BANK*. (In this example S equals 5.) In order to modify the object, a task T_i first makes a private copy of the bank of pointers using the long-LL routine. Then, task T_i accesses and modifies elements of the array *MEM* using *Read* and *Write* library routines, which are described below. After modifying the object, task p uses a long-SC primitive to install its private copy of the bank of pointers as the new bank of pointers.

The *Read* routine takes an "address" (this is actually an index into *MEM*) as input and returns the contents of that location. The parameters input to the *Write* routine are an address (i.e., an index into *MEM*) and a new value. The *Write* procedure is described using an example illustrated in Figure 2.7. In this example, in order to write into a location in block 1, task T_i makes a local copy of block 1 and then modifies its local copy. Task

[6]Unlike single-word LL/SC primitives, these primitives operate on "large" shared variables containing multiple words.

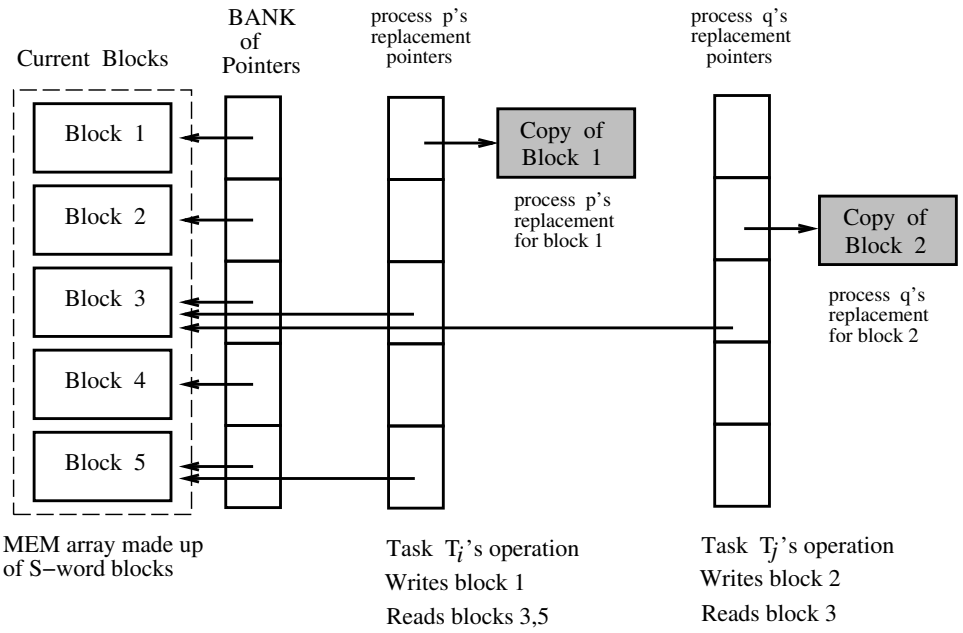


Figure 2.7: Implementation of the *MEM* array for large object constructions.

T_i also modifies the pointer to block 1 in its bank of replacement pointers such that it points to T_i's local copy. Task T_j writes into block 2 in a similar manner. The *Read* and *Write* library routines hide from the programmer the implementation details associated with copying blocks and other associated book-keeping.

Although Anderson and Moir's construction permits a great deal of transparency, it suffers from certain shortcomings. First, a task must copy (replace) the whole bank of pointers even if it accesses (modifies) only a small number of blocks. Hence, two operations can interfere with each other even if they access different sets of blocks. In the previous example, the operations of tasks T_i and T_j can interfere with each other even though they access different blocks.

Another drawback of Anderson and Moir's construction concerns multi-object ac-

cesses, i.e., operations that update multiple objects simultaneously. Multi-object constructions are the lock-free counterpart of nested critical sections. In conventional lock-based systems, operations on multiple objects can be performed via nested critical sections. For example, two critical sections might be nested in such a system to transfer the contents of one shared buffer to another. To achieve a high degree of concurrency, any implementation that supports multi-object operations must allow operations on different objects to proceed concurrently. However, if a multi-object construction is based on Anderson and Moir's large object construction, then operations of two tasks can interfere with one another even if they access objects located in different blocks of memory. Hence, their construction is unsuitable for multi-object operations because it severely restricts concurrency. To rectify this problem, Anderson and Moir developed a multi-object construction that enables a high degree of concurrency [8].

Multi-Object Constructions

The multi-object construction presented by Anderson and Moir in [8] is a generalization of Herlihy's lock-free construction described previously. As in Herlihy's construction, a task T_i loads the pointer to each object that it accesses and then makes a local copy of that object. Then, task T_i applies its multi-object operation on its local object copies. Finally, T_i tries to "install" new versions of all objects that it modifies using a multiword SC (MWSC) primitive.[7] Task T_i repeatedly performs the above steps until its MWSC is successful. Note that if task T_i succeeds its multi-object operation, then each object S

[7] The MWSC primitive extends the semantics of single-word SC primitive to multiple words. In particular, the MWSC primitive atomically modifies multiple words, unlike the Long-SC primitive which atomically modifies a single "long" word.

accessed by T_i's operation was not modified by another task's operation, after T_i loads the pointer to object S and before T_i's subsequent MWSC operation.

Anderson and Moir's multi-object construction suffers from two main drawbacks. First, because their construction is based on Herlihy's lock-free construction, copying overhead can be excessive if large objects are accessed. Second, implementing the MWSC primitive[8] used in their construction entails high algorithmic overhead. In particular, the worst-case running-time of the MWSC implementation presented in [8] is $O(N^3M)$, where N is the number of tasks in the system and M is the number of implemented shared words. Thus, although their construction exhibits flexibility similar to that of nested critical sections, it is not competitive with nested critical sections in uniprocessor real-time systems.

2.4.4 Specific Objects

Many researchers have presented lock-free and wait-free implementations for specific shared objects. Such implementations can potentially take advantage of the semantics of the object under consideration to improve performance. However, most implementations of specific objects have required considerable creative and intellectual effort, highlighting the need for universal constructions. Some specific object implementations are listed below.

Many researchers have studied implementations of various kinds of wait-free shared objects using only read/write registers. These implementations include constructions of complex registers from simpler registers [23, 24, 49, 50, 54, 59, 66, 67, 68, 75]; atomic snapshots that allow multiple variables to be read atomically; [1, 3, 17], algorithms for maintaining timestamps [25, 28]; and mechanisms for implementing any object whose oper-

[8]The MWSC primitive must be implemented in software because it is not supported by any real machine.

ations satisfy certain algebraic requirements [5, 16]. For example, a construction is given in [5] that implements any object such that, for each pair of operations on the object, either the two operations commute with each other, or one overwrites the other (i.e., the effects of executing both operations is the same as executing just one of them).

Other researchers have considered wait-free and lock-free implementations using instructions that are stronger than simple reads and writes. Implementations of various types of queues have been presented by Lamport [53], by Herlihy and Wing [38], by Israeli and Rappoport [41], by Wing and Gong [81, 82], and by Michael and Scott [64]. Anderson and Woll [15] and Lanin and Shasha [56] present implementations for various set operations. Valois presents lock-free implementations for various data structures, including queues, lists, trees, and dictionaries [78, 79, 80]. Finally, Massalin and Pu have implemented an entire operating system using lock-free data structures such as lists, queues, and stacks [63].

Chapter 3

Scheduling Conditions

In this chapter, we derive conditions to predict the schedulability of task sets that access lock-free objects. The scheduling conditions we derive are essential for enabling the use of lock-free objects in hard real-time systems. Our conditions are obtained by modifying scheduling conditions for independent tasks to account for the overhead of operation interferences. Specifically, we derive scheduling conditions for the RM, DM, EDF, and EDF/NPD schemes. For the sake of clarity, when we refer to EDF scheduling in this chapter, we mean EDF scheduling with the restriction that each task's relative deadline equals its period. We will use the term EDF/NPD (EDF with nonequal deadlines and periods) to refer to EDF scheduling in which each task's relative deadline is at most its period.

This chapter is organized as follows. In Section 3.1, we present key assumptions that underlie our task model. We also present notation used in deriving our scheduling conditions. Then, in Section 3.2, we derive certain key lemmas used in the proofs our conditions. Scheduling conditions for static-priority and dynamic-priority schemes are pre-

sented in Sections 3.3 and 3.4, respectively. These conditions can be used without any knowledge of a task's object accesses, but are based on the assumption that all retry-loop costs are relatively uniform. In Section 3.5, we present a general approach based on integer linear programming to bound the cost of operation interferences over an interval of time. Based on the results in Section 3.5, we derive accurate scheduling conditions for static-priority and dynamic-priority schemes in Sections 3.6 and 3.7, respectively. We now present definitions and notation used in this chapter. (Definitions of terms such as period, relative deadline etc., are defined earlier in Section 1.3.)

3.1 Assumptions and Definitions

We implicitly assume that tasks share a set of objects implemented using lock-free algorithms. Note that there is no need to explicitly include such objects in our model, because operations on lock-free objects are implemented by task-level code sequences. For simplicity, we assume that jobs can be preempted at arbitrary points during their execution, and ignore system overhead costs like context switching costs, interrupt handling costs, etc.

We say that a job is *interfered with* (or experiences an *interference*) if it executes a lock-free retry loop that does not successfully complete. Unless specified otherwise, we assume that the *deadline* of a job of a task is the end of the corresponding period of that task, i.e., the task's relative deadline equals its period. A task set is *schedulable* if and only if all jobs of all tasks meet their deadlines. The following is a list of symbols used in deriving our scheduling conditions.

- N - The number of tasks in the system. We use i and j as task indices. Unless stated

otherwise, we assume that i and j are universally quantified over $\{0, \ldots, N-1\}$.

- T_i - The i^{th} task in the system.
- p_i - The period of task T_i.
- l_i - The relative deadline of task T_i. Unless specified otherwise, we assume that $l_i \leq p_i$.
- $r_i(k)$ - The release time of the k^{th} job of T_i, where $r_i(k) = r_i(1) + (k-1) \cdot p_i$. We use k as a job index. Unless stated otherwise, we assume that k is universally quantified with range $k \geq 1$.
- $J_{i,k}$ - The k^{th} job of task T_i.
- $e_i(k)$ - The completion time of job $J_{i,k}$.
- $d_i(k)$ - The deadline of job $J_{i,k}$.
- c_i - The worst-case computational cost (execution time) of task T_i when it is the only task executing on the processor, i.e., when there is no contention for the processor or for shared objects.
- s - The worst-case computational cost of executing one iteration of any lock-free retry-loop in the system.

In this chapter, we derive conditions only for some common scheduling schemes. However, our approach for bounding the cost of interferences is applicable to any scheduling scheme satisfying the following axioms.

Axiom 3.1: If a job of task T_i can preempt a job of task T_j, then no job of T_j preempts any job of T_i. □

Axiom 3.2: The priority of a job does not change while accessing a shared object. □

Axiom 3.3: Different jobs of the same task cannot preempt one another. □

These axioms hold for the RM, DM, EDF, and EDF/NPD schemes, and for variations of these schemes in which tasks consist of multiple phases with separate execution priorities. We make explicit the preemption order between any two tasks — as specified by Axiom 3.1 — by indexing the tasks such that, if a job of task T_i can preempt a job of task T_j, then $i < j$. For the RM, DM, EDF, and EDF/NPD schemes, arranging tasks in the order of increasing deadlines results in task indices compatible with this indexing scheme. Under RM scheduling, we assume that if two tasks have the same period, then the task with the smaller task index has higher priority. Under EDF scheduling, we assume that if two jobs have the same deadline, then the job with the earlier release time has higher priority; if two such jobs are released at the same time, then the one with the smaller index has higher priority.

In Sections 3.3 and 3.4, we assume that s units of execution time is required for one loop iteration of any lock-free object, i.e., the lock-free retry-loop cost is the same of all objects. We obtain conditions for schedulability by determining the worst-case unfulfilled demand of each task. The *unfulfilled demand* of task T_i at time t is the remaining computation time of T_i's job executing at that time. The unfulfilled demand of T_i decreases by one from time t to time $t+1$ if a job of T_i executes at time t. When a job of task T_i is released,

T_i's unfulfilled demand increases by c_i. Task T_i's unfulfilled demand can also increase due to interferences experienced by its jobs. Such increases are characterized by the following *interference assumptions*. Note that Assumption IA2 below holds only for the results in Sections 3.3 and 3.4. This assumption is relaxed later in Section 3.5.

IA1 *A job J experiences an interference at time t if and only if, for some $t' \leq t$, (i) J executes at time $t'-1$, (ii) J is preempted at time t' by some higher-priority job, (iii) only higher-priority jobs execute in the interval $[t', t]$, (iv) no higher-priority job that accesses an object in common with J is released in $[t', t)$, and (v) at least one such job is released at time t. (This implies that job J can be interfered with at most once during any interval when it is preempted.)*

IA2 *Each interference experienced by a job $J_{i,k}$ increases T_i's unfulfilled demand by s.*

IA1 is pessimistic because the preempted job J may not, in fact, be accessing any shared object when preempted. IA2 is pessimistic because the cost of executing one iteration of the retry loop of any object is assumed to be s units. If retry loop costs are not the same, IA2 essentially requires that the unfulfilled demand of a task T_i be increased by the cost of the largest retry loop, for each interference in jobs of T_i.

In the proofs of our scheduling conditions, we also use the notion of task "demand", which is related to the notion of unfulfilled demand. The *demand* placed by a task T_i on the processor in an interval $[t, t']$ is the amount of processing time required by jobs of T_i in that interval [45]. In particular, task T_i's demand in $[t, t']$ includes T_i's unfulfilled demand at time t, c_i time units for each job release of T_i in $(t, t']$, and s time units for each interference occurring within $(t, t']$ in jobs of task T_i. A task is said to be *inactive* at time t if it places

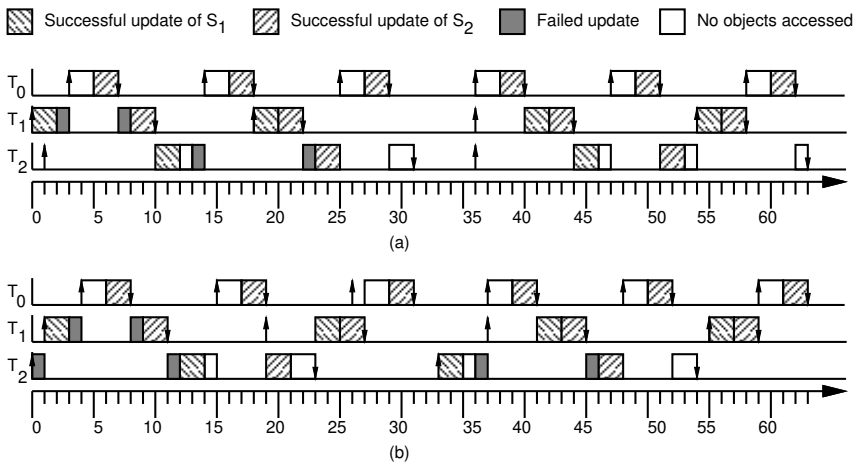


Figure 3.1: Illustration of the task sets defined in Examples 3.1 and 3.2.

no demand on the processor at that time. The following examples illustrate some of the subtleties of our task model.

Example 3.1: *Let a task T_i be given by the tuple $(r_i(1), c_i, p_i)$. Consider the following set of periodic tasks scheduled under the RM scheme.*

$$T_0 = (3, 4, 11) \qquad T_1 = (0, 4, 18) \qquad T_2 = (1, 7, 35)$$

Assume that object S_1 is accessed by T_1 and T_2, and S_2 is accessed by T_0, T_1, and T_2. Tasks T_0 and T_1 access S_1 and S_2 in that order. Also, assume that $s = 2$.

The execution of the above task set is illustrated in Figure 3.1(a). In this figure, up-arrows represent job releases, down-arrows represent job completions, and shaded regions represent shared object accesses. Because the deadline of a job of a task corresponds to the release of the next job of that task, up-arrows also denote job deadlines.

We see that job $J_{2,1}$ experiences an interference at time 14, when job $J_{0,2}$ is released. This is because job $J_{0,2}$ is the earliest job that accesses an object in common with

$J_{2,1}$ and that is released in the closed interval between $J_{2,1}$'s preemption at time 14 and its subsequent resumption at time 22. Thus, $J_{2,1}$'s operation on S_2 that begins at time 13 fails. This operation is retried at time 23, when it is successfully completed. Observe that $J_{2,1}$ experiences only one interference in the interval $[14, 22]$, even though two jobs ($J_{0,2}$ and $J_{1,2}$) are released in that interval that can potentially interfere with $J_{2,1}$'s object access. By IA1, we assume that $J_{1,4}$ interferes with $J_{2,2}$ at time 54 (because $J_{1,4}$ preempts $J_{2,2}$), even though $J_{2,2}$ is not actually accessing a shared object at that point. Thus, the upper bound on interference costs in the analysis presented later is rather pessimistic.

Task T_2's unfulfilled demand increases by 7 units at time 1 due to a job release and increases by 2 units at time 14 due to an interference by $J_{0,2}$. Also, task T_2's unfulfilled demand decreases by one unit at every instant in the interval $[11, 14]$ because it executes on the processor. The demand placed by T_2 on the processor in the interval $[5, 37]$ is 16 units; 7 units due to T_2's unfulfilled demand at time 5, 2 units due to an interference in $J_{2,1}$ at time 14, and 7 units due the release of $J_{2,2}$ at time 36. □

Example 3.2: *Consider the following set of periodic tasks scheduled under the EDF scheme.*

$$T_0 = (4, 4, 11) \qquad T_1 = (1, 4, 18) \qquad T_2 = (0, 7, 33)$$

Assume that object S_1 is accessed by T_1 and T_2, and S_2 is accessed by T_0 and T_1. Task T_1 accesses S_1 and S_2 in that order; T_2 accesses S_1 twice. As in the previous example, we assume that $s = 2$.

The execution of the above task set is illustrated in Figure 3.1(b). We see that job $J_{2,1}$ experiences an interference at time 4 while accessing S_1, when job $J_{0,1}$ is released. $J_{0,1}$ has an earlier deadline than $J_{2,1}$ and is the first job that accesses an object in common

with $J_{2,1}$ and that is released in the closed interval between $J_{2,1}$'s preemption at time 1 and its subsequent resumption at time 11. Observe that $J_{1,1}$ also experiences an interference at time 4 while accessing object S_2 due to the release of $J_{0,1}$. The total demand placed in the interval $[0, 25]$ by jobs with deadlines at or before 25 is 10 units; 4 and 6 units due to tasks T_0 and T_1, respectively. Of the 6 units of T_1's demand, 2 units are due to an interference in T_1 at time 4. Note that we do not include the demand due to jobs $J_{0,2}$, $J_{1,2}$ and $J_{2,1}$ because their deadlines are after time 25. □

3.2 Preliminary Lemmas

Before we present our scheduling conditions, we prove several lemmas used in the proofs of these conditions. In [60], it is shown that for independent tasks (i.e., tasks that do not share objects), the longest response time of a task occurs at a *critical instant* of time, at which jobs of that task and all higher-priority tasks are released. However, this is not necessarily the case if tasks share lock-free objects, as illustrated in Example 3.1. In this example, the longest response time of task T_2 does not occur when its job is released along with higher-priority jobs at time 36. The job released at time 1 has a longer response time.

Instead of defining the critical instant, we introduce the notion of a "busy point". The *busy point* $b_i(k)$ of job $J_{i,k}$ is the latest point in time at or before $r_i(k)$ when jobs that have priority at least that of $J_{i,k}$ are either inactive or have a job release. For example, in Figure 3.1, the busy point of $J_{2,1}$ occurs at time 0, i.e., $b_2(1) = 0$. Because each task is either inactive or releases a job at time 0, $b_i(k)$ is well-defined for any i and k. Our scheduling conditions are obtained by inductively counting interferences over intervals of time. A busy

point provides a convenient instant at which to start such an inductive argument, because tasks that are inactive or that have just released a job have experienced no interferences.

Lemma 3.1: *Consider any $t \in [b_i(k), r_i(k+1))$ at which $J_{i,k}$ has positive unfulfilled demand. Let v be the priority of $J_{i,k}$. In the interval $[b_i(k), t]$, the number of interferences in jobs with priority at least v is bounded by the number of instants in the interval $(b_i(k), t]$ at which some job with priority greater than v is released.*

Proof: To simplify the proof, we prove a slightly stronger statement: the number of interferences in the interval $[b_i(k), t]$ in jobs with priority at least v is bounded by the difference between (i) the number of instants in the interval $(b_i(k), t]$ at which some job with priority greater than v is released and (ii) the number of preempted jobs at time t that have not been interfered with[1] and that have priority at least v. The proof is by induction on t.

Basis: We show that the lemma holds at $b_i(k)$, $J_{i,k}$'s busy point. There can be no interferences in the interval $[b_i(k), b_i(k)]$ in jobs with priority at least v because $J_{i,k}$ and higher-priority jobs are either inactive or have a job release at $b_i(k)$. For the same reason, there are no preempted jobs at $b_i(k)$ with priority at least v. Clearly, there are zero instants in the interval $(b_i(k), b_i(k)]$ at which some job is released that has priority greater than v. Hence, the basis of the induction holds.

Induction Step: Assume that the above lemma holds at time $t - 1 \geq b_i(k)$. Let J be some job executing at time $t - 1$. (Such a job J must exist by the definition of $b_i(k)$.) Suppose that there are f interferences in the interval $[b_i(k), t - 1]$ in jobs with priority at least v,

[1] Note that jobs that have been released but have not yet executed cannot be interfered with.

and that there are w instants in the interval $(b_i(k), t-1]$ at which some job with priority greater than v is released. Also, suppose that there are x preempted jobs at time $t-1$ that have priority at least v and that have not been interfered with. Our inductive hypothesis can be formally written as $f \leq w - x$. We now consider two cases.

Case 1: If no job is released at time t that has priority greater than v, then by IA1, no interference can occur at that time in any job with priority at least v. Hence, there are f interferences in $[b_i(k), t]$ and w instants in $(b_i(k), t]$ at which some job is released that has priority greater than $J_{i,k}$'s priority. Also, the number of preempted jobs at time t is either $x-1$ or x, depending on whether J completes at time t or not. In either case, the lemma holds at time t because our inductive hypothesis implies both $f \leq w - (x-1)$ and $f \leq w - x$.

Case 2: If $y > 0$ jobs are released at time t that have priority greater than v, then there are $w+1$ instants in the interval $(b_i(k), t]$ at which some job is released that has priority greater than v. Suppose that some number q of the x preempted jobs incur an interference at time t due to some newly released job that accesses a common object. We consider three subcases.

J has higher priority than all newly released jobs. It follows from IA1 that none of the jobs released at time t can interfere with J. Thus, there are $f+q$ interferences in $[b_i(k), t]$ and $x-q$ preempted jobs that have not been interfered with. (The y jobs released at time t cannot be interfered with because they have not started execution yet.) The lemma holds at time t because our inductive hypothesis implies $f + q \leq (w+1) - (x-q)$.

J is preempted at time t but none of the newly released jobs accesses an object in common with J. By IA1, none of the newly released jobs can interfere with J. Therefore, the number of interferences in $[b_i(k), t]$ is $f + q$. The number of preempted jobs that have not been interfered with is $x - q + 1$ (including J). The lemma holds at time t because our inductive hypothesis implies $f + q \leq (w + 1) - (x - q + 1)$.

J is preempted at time t and some newly released job accesses an object in common with J. It follows from IA1 that J is interfered with at time t. Hence, the number of interferences in $[b_i(k), t]$ is $f + q + 1$. The number of preempted jobs that have not been interfered with is $x - q$. Again, our inductive hypothesis implies $f + q + 1 \leq (w + 1) - (x - q)$.

□

Lemma 3.2: *Consider any $t \in [b_i(k), r_i(k + 1))$. Under the DM scheme, the number of interferences in T_i and higher-priority tasks in the interval $[b_i(k), t]$ is at most*

$$\sum_{j=0}^{i-1} \left\lceil \frac{t - b_i(k)}{p_j} \right\rceil.$$

Proof: From Lemma 3.1, it follows that the number of interferences in jobs with priority at least that of $J_{i,k}$ in the interval $[b_i(k), t]$ is bounded by the number of instants in $(b_i(k), t]$ at which some job is released that has priority greater than that of $J_{i,k}$. Under the DM scheme, only jobs of tasks T_0 through T_{i-1} have priority greater than $J_{i,k}$, and the number of jobs of task T_j released in the interval $(b_i(k), t]$ is at most $\lceil (t - b_i(k))/p_j \rceil$. Therefore, the number of interferences in T_i and higher-priority tasks in the interval $[b_i(k), t]$ is bounded by $\sum_{j=0}^{i-1} \left\lceil \frac{t - b_i(k)}{p_j} \right\rceil$. □

Lemma 3.3: *Under the DM scheme, if, at time $d_i(k) - 1$, T_i has positive unfulfilled demand*

and the total unfulfilled demand of T_i and higher-priority tasks is greater than one, then, for any t in the interval $[b_i(k), d_i(k))$, the difference between (i) the total demand placed on the processor by T_i and higher-priority tasks in the interval $[b_i(k), t]$, and (ii) the available processor time in that interval, is greater than one.

Proof: The proof can be established by contradiction. To this end, suppose that there exists a $t \in [b_i(k), d_i(k))$ such that the difference between the total demand placed by tasks T_1 through T_i in the interval $[b_i(k), t]$ and the available processor time in that interval is at most one. It follows that the total unfulfilled demand of tasks T_0 through T_i at time t equals zero or one. Because the total unfulfilled demand of T_i and higher-priority tasks is greater than one at time $d_i(k) - 1$, $t \neq d_i(k) - 1$ holds. Also, either tasks T_0 through T_i are inactive at time t, or one of tasks T_0 through T_i has unit unfulfilled demand and is the highest-priority job executing on the processor. In both cases, it follows that T_i and higher-priority tasks are either inactive or have a job release at time $t + 1$. To complete the proof, we show that this leads to a contradiction.

By the definition of a busy point, $b_i(k)$ is the latest time at or before $r_i(k)$ at which T_i and higher-priority tasks are either inactive or have a job release. Thus, it follows that $t + 1$ cannot lie in the interval $(b_i(k), r_i(k)]$. Also, as explained earlier, $t \neq d_i(k) - 1$, i.e., $t + 1 \neq d_i(k)$. Hence, $t + 1$ lies in the interval $(r_i(k), d_i(k))$. T_i clearly cannot have a job release in the interval $[t + 1, d_i(k))$ because $t + 1 > r_i(k)$ and $d_i(k) \leq r_i(k + 1)$. Thus, T_i is inactive — and hence has no unfulfilled demand — throughout the interval $[t + 1, d_i(k))$, contradicting our assumption that T_i has positive unfulfilled demand at time $d_i(k) - 1$. □

Lemma 3.4: *Under the EDF/NPD scheme, the number of interferences in jobs with a*

deadline at or before $d_i(k)$ *in the interval* $[b_i(k), d_i(k))$ *is at most*

$$\sum_{j=0}^{N-1} \left\lfloor \frac{d_i(k)-b_i(k)-2+p_j-l_j}{p_j} \right\rfloor.$$

Proof: It follows from Lemma 3.1 that the number of interferences in $[b_i(k), d_i(k))$ is bounded by the number of instants in $(b_i(k), d_i(k))$ at which some job is released that has priority greater than $J_{i,k}$'s priority, i.e., the released job's deadline is before $d_i(k)$. Under the EDF scheme, the number of jobs of T_j released in the interval $(b_i(k), d_i(k))$ that have a deadline before $d_i(k)$ is at most $\left\lfloor \frac{(d_i(k)-1)-(b_i(k)+1)+(p_j-l_j)}{p_j} \right\rfloor$. Therefore, the number of interferences in jobs with deadlines at or before $d_i(k)$ in the interval $[b_i(k), d_i(k))$ is bounded by $\sum_{j=0}^{N-1} \left\lfloor \frac{d_i(k)-b_i(k)-2+(p_j-l_j)}{p_j} \right\rfloor$. □

Corollary 3.1: *Under the EDF scheme, the number of interferences in jobs with a deadline at or before* $r_i(k+1)$ *in the interval* $[b_i(k), r_i(k+1))$ *is at most*

$$\sum_{j=0}^{N-1} \left\lfloor \frac{r_i(k+1)-b_i(k)-2}{p_j} \right\rfloor.$$

Proof: Follows from Lemma 3.4, by substituting p_i for l_i and $r_i(k+1)$ for $d_i(k)$. (Recall that, under the EDF scheme, $p_i = l_i$ and $d_i(k) = r_i(k+1)$ for any task T_i and any $k \geq 1$.)

□

3.3 Static-Priority Scheduling Conditions

In this section, we give separate necessary and sufficient conditions for the schedulability of a set of periodic tasks that share lock-free objects under the DM scheme. These conditions assume that priority is assigned by the DM scheme [58], in which tasks with

smaller relative deadlines have higher priorities. We also briefly consider the RM scheme, which is special case of DM scheduling.

The following theorem gives a necessary scheduling condition for the DM scheme. The left-hand side of the quantified expression given below gives the minimum demand — which arises when there are no interferences — placed on the processor by T_i and higher-priority tasks in the interval $[0, t]$, where $0 < t \leq p_i$. The right-hand side gives the available processor time in that interval.

Theorem 3.1: (*Necessity under DM*) *If a set of periodic tasks that share lock-free objects is schedulable under the DM scheme, then the following condition holds for every task* T_i.

$$\langle \exists t : 0 < t \leq l_i : \sum_{j=0}^{i} \left\lfloor \frac{t}{p_j} \right\rfloor \cdot c_j \leq t \rangle$$

Proof: Consider some task T_i belonging to a set of tasks schedulable under the DM scheme. Let t_m be any time such that $t_m > r_j(1)$, for any task T_j. Suppose that $J_{i,k}$ is some job released after t_m. Because the task set is schedulable, $J_{i,k}$ completes within l_i instants after its release, i.e., $e_i(k) - r_i(k) \leq l_i$. Because T_i executes on the processor at time $e_i(k) - 1$, T_i has unit unfulfilled demand and all higher-priority tasks are inactive — and hence, have no job releases — at that time. It follows that the difference between (i) the total demand $D_i(r_i(k), e_i(k) - 1)$ due to T_i and higher-priority tasks in the interval $[r_i(k), e_i(k) - 1]$, and (ii) the available processor time in that interval, is at most one. This observation can be formally stated as follows.

$$D_i(r_i(k), e_i(k) - 1) - (e_i(k) - r_i(k) - 1) \leq 1 \tag{3.1}$$

The previous expression can also be expressed as follows.

$$D_i(r_i(k), e_i(k) - 1) \leq e_i(k) - r_i(k) \tag{3.2}$$

We now derive a least upper bound on $D_i(r_i(k), e_i(k) - 1)$. The term $D_i(r_i(k), e_i(k) - 1)$ is comprised of two components: (i) $D_i'(r_i(k))$, the unfulfilled demand at time $r_i(k)$ due to jobs of T_i and higher-priority tasks released before $r_i(k)$, and (ii) the demand due to jobs of those tasks released during $[r_i(k), e_i(k) - 1]$. The minimum number of jobs of T_j released in the interval $[r_i(k), e_i(k) - 1]$ is given by $\lfloor (e_i(k) - r_i(k))/p_j \rfloor$, and each job requires c_j units of computation. Hence, we have

$$D_i(r_i(k), e_i(k) - 1) \geq D_i'(r_i(k)) + \sum_{j=0}^{i} \left\lfloor \frac{e_i(k) - r_i(k)}{p_j} \right\rfloor c_j.$$

Because $D_i'(r_i(k)) \geq 0$, the previous expression implies

$$D_i(r_i(k), e_i(k) - 1) \geq \sum_{j=0}^{i} \left\lfloor \frac{e_i(k) - r_i(k)}{p_j} \right\rfloor c_j.$$

The previous inequality, along with (3.2), implies the following.

$$\sum_{j=0}^{i} \left\lfloor \frac{e_i(k) - r_i(k)}{p_j} \right\rfloor c_j \leq e_i(k) - r_i(k)$$

Now, $r_i(k) < e_i(k) \leq r_i(k) + l_i$, i.e., $0 < e_i(k) - r_i(k) \leq l_i$ holds. Replacing $e_i(k) - r_i(k)$ with t in the above expression, where $0 < t \leq l_i$, we have the following.

$$\sum_{j=0}^{i} \left\lfloor \frac{t}{p_j} \right\rfloor c_j \leq t$$

□

Theorem 3.1 also is also a necessary condition for the RM scheme because the DM scheme reduces to the RM scheme when $l_i = p_i$. The resulting necessary condition for the

RM scheme differs slightly from that in [57] because we allow tasks to release their first jobs at arbitrary times. In particular, the floor functions within the summation expression in Theorem 3.1 are replaced by ceiling functions.

The next theorem gives a sufficient scheduling condition for the DM scheme. The left-hand side of the quantified expression given below gives the maximum demand placed by T_i and higher-priority tasks in the interval $[0, t)$. The first summation represents the demand placed on the processor by T_i and higher-priority tasks, not including the demand due to interferences. The second summation represents the total additional demand placed on the processor due to interferences in T_i and higher-priority tasks. The right-hand side of the expression is the available processor time in $[0, t)$. Observe that this condition can be applied without knowledge of which tasks access which objects.

Theorem 3.2: (*Sufficiency under DM*) *A set of periodic tasks that share lock-free objects is schedulable under the DM scheme if the following condition holds for every task* T_i.

$$\langle \exists t : 0 < t \leq l_i : \sum_{j=0}^{i} \left\lceil \frac{t}{p_j} \right\rceil \cdot c_j + \sum_{j=0}^{i-1} \left\lceil \frac{t-1}{p_j} \right\rceil \cdot s \leq t \rangle$$

Proof: We prove that if a task set is not schedulable, then the negation of the above expression holds. Assume that the given task set is not schedulable. Let $J_{i,k}$ be the first job to miss its deadline. (If several jobs simultaneously miss their deadline along with $J_{i,k}$, then let $J_{i,k}$ be the one with highest priority, i.e., smallest task index.) Consider any t in the interval $[b_i(k), d_i(k))$. We begin by deriving a bound on $D(b_i(k), t)$, the total demand placed on the processor by T_i and higher-priority tasks in the interval $[b_i(k), t]$. $D(b_i(k), t)$ is comprised of the demand placed by job releases and the extra demand placed

by interferences. Recall that at the busy point $b_i(k)$, T_i and all higher-priority tasks are either inactive or have a job release. Each job release of some task T_j introduces a demand of c_j on the processor, and there are at most $\lceil (t - b_i(k) + 1)/p_j \rceil$ job releases of that task in the interval $[b_i(k), t]$. Therefore, the total demand placed on the processor due to job releases of T_i and higher-priority tasks is at most $\sum_{j=0}^{i} \lceil (t - b_i(k) + 1)/p_j \rceil c_j$.

By Lemma 3.2, the number of interferences in jobs of T_i and higher-priority tasks in the interval $[b_i(k), t]$ is bounded by $\sum_{j=0}^{i-1} \lceil (t - b_i(k))/p_j \rceil$. By IA2, each interference introduces s units of additional demand on the processor. Therefore, the total additional demand due to interferences in jobs of T_i and higher-priority tasks is at most $\sum_{j=0}^{i-1} \lceil (t - b_i(k))/p_j \rceil s$. Therefore, we have

$$D(b_i(k), t) \leq \sum_{j=0}^{i} \left\lceil \frac{t - b_i(k) + 1}{p_j} \right\rceil c_j + \sum_{j=0}^{i-1} \left\lceil \frac{t - b_i(k)}{p_j} \right\rceil s.$$

Job $J_{i,k}$ will miss its deadline if and only if, at time $d_i(k) - 1$, T_i has positive unfulfilled demand and the total unfulfilled demand of T_i and higher-priority tasks is greater than one. By Lemma 3.3, it follows that the difference between the total demand due to T_i and higher-priority tasks in the interval $[b_i(k), t]$ and the available processor time in that interval is greater than one. Hence, we have the following.

$$D(b_i(k), t) - (t - b_i(k)) > 1$$

Using the bound on $D(b_i(k), t)$ derived above, the previous expression implies the following.

$$\sum_{j=0}^{i} \left\lceil \frac{t - b_i(k) + 1}{p_j} \right\rceil c_j + \sum_{j=0}^{i-1} \left\lceil \frac{t - b_i(k)}{p_j} \right\rceil s > t - b_i(k) + 1$$

The above expression holds for all t in the interval $[b_i(k), d_i(k))$. Because this expression is independent of the end points (it is a function of the length of the interval), we can replace $t - b_i(k)$ with t', where $t' = t - b_i(k)$ and $t' \in [0, d_i(k) - b_i(k))$. Hence, we have the following.

$$\sum_{j=0}^{i} \left\lceil \frac{t'+1}{p_j} \right\rceil c_j + \sum_{j=0}^{i-1} \left\lceil \frac{t'}{p_j} \right\rceil s > t' + 1$$

Now, replace t' with t in the above expression, where $t = t' + 1$ and $t \in (0, d_i(k) - b_i(k)]$. Then, the following holds for all $t \in (0, d_i(k) - b_i(k)]$.

$$\sum_{j=0}^{i} \left\lceil \frac{t}{p_j} \right\rceil c_j + \sum_{j=0}^{i-1} \left\lceil \frac{t-1}{p_j} \right\rceil s > t$$

By definition, $b_i(k) \leq r_i(k)$. Therefore, the interval $(0, d_i(k) - r_i(k)]$ is completely contained in $(0, d_i(k) - b_i(k)]$. Also, by definition, $d_i(k) - r_i(k) = l_i$. Therefore, the expression above holds for all t in $(0, l_i]$. □

Because RM scheduling is a special case of DM scheduling, the following corollaries follow from Theorems 3.1 and 3.2.

Corollary 3.2: (*Necessity under RM*) *If a set of periodic tasks that share lock-free objects is schedulable under the RM scheme, then the following condition holds for every task* T_i.

$$\langle \exists t : 0 < t \leq p_i : \sum_{j=0}^{i} \left\lfloor \frac{t}{p_j} \right\rfloor \cdot c_j \leq t \rangle$$ □

Corollary 3.3: (*Sufficiency under RM*) *A set of periodic tasks that share lock-free objects is schedulable under the RM scheme if the following condition holds for every task* T_i.

$$\langle \exists t : 0 < t \leq p_i : \sum_{j=0}^{i} \left\lceil \frac{t}{p_j} \right\rceil \cdot c_j + \sum_{j=0}^{i-1} \left\lceil \frac{t-1}{p_j} \right\rceil \cdot s \leq t \rangle$$ □

In the videoconferencing system described in Chapter 6, tasks are sporadic rather than periodic. However, the scheduling conditions that we derive also apply if tasks are sporadic.

3.4 Dynamic-Priority Scheduling Conditions

In this section, we give separate necessary and sufficient conditions for the schedulability of a set of periodic tasks that share lock-free objects under the EDF scheme. The following theorem gives a necessary scheduling condition for the EDF/NPD scheme. According to this theorem, a task set is schedulable only if processor utilization is at most one. This condition also holds when tasks periods are equal to their relative deadlines, and is identical to the condition in [60].

Theorem 3.3: (*Necessity under EDF/NPD*) *If set of periodic tasks that share lock-free objects is schedulable under the EDF/NPD scheme, then*

$$\sum_{i=0}^{N-1} \frac{c_i}{p_i} \leq 1.$$

Proof: Consider a set of tasks that is schedulable under the EDF/NPD scheme. Let t_m be the earliest instant such that $t_m \geq r_j(1)$ for any task T_j. Consider any job $J_{i,k}$ such that $r_i(k) \geq t_m$. We begin by deriving a lower bound on $D(0, d_i(k))$, the total demand due to all jobs with deadline at or before time $d_i(k)$, in the interval $[0, d_i(k)]$.

By assumption, we have $d_i(k) > t_m \geq r_j(1)$. Hence, there are $\lfloor (d_i(k) - r_j(1) + p_j - l_j)/p_j \rfloor$ jobs of task T_j released in the interval $[0, d_i(k)]$ that have a deadline at or before $d_i(k)$, and each job requires c_j units of computation. It follows that the total demand placed

by tasks T_1 through T_N is given by $\sum_{j=0}^{N-1} \lfloor (d_i(k) - r_j(1) + p_j - l_j)/p_j \rfloor c_j$. Therefore, we the following.

$$D(0, d_i(k)) = \sum_{j=0}^{N-1} \left\lfloor \frac{d_i(k) - r_j(1) + p_j - l_j}{p_j} \right\rfloor c_j$$

Because no job misses its deadline at time $d_i(k)$, it follows that the total demand placed by tasks T_1 through T_N in the interval $[0, d_i(k)]$ is at most the available processor time in that interval, i.e., $D(0, d_i(k)) \leq d_i(k)$. Therefore, we have the following.

$$\sum_{j=0}^{N-1} \left\lfloor \frac{d_i(k) - r_j(1) + p_j - l_j}{p_j} \right\rfloor c_j \leq d_i(k)$$

Because $\lfloor (d_i(k) - r_j(1) + p_j - l_j)/p_j \rfloor$ is at least $((d_i(k) - r_j(1) + p_j - l_j)/p_j) - 1$, we have

$$\sum_{j=0}^{N-1} \left(\frac{d_i(k) - r_j(1) + p_j - l_j}{p_j} - 1 \right) c_j \leq d_i(k).$$

Rearranging the terms in the above expression yields the following.

$$\left(\sum_{j=0}^{N-1} \frac{c_j}{p_j} - 1 \right) d_i(k) \leq \sum_{j=0}^{N-1} \frac{(r_j(1) + l_j) \cdot c_j}{p_j}$$

The right-hand side of the above inequality is a positive constant, but $d_i(k)$ can take arbitrarily large values. (Recall that $J_{i,k}$ is any job released after t_m.) Hence, it follows that the term $(\sum_{j=0}^{N-1} \frac{c_j}{p_j} - 1)$ on the left-hand side is at most zero. Hence, we have $\sum_{j=0}^{N-1} \frac{c_j}{p_j} \leq 1$. □

The following theorem gives a sufficient condition for schedulability under the EDF/NPD scheme. Like the DM and RM sufficiency conditions of the previous section,

conditions for EDF and EDF/NPD schemes can be applied without knowledge of which tasks access which objects.

Theorem 3.4: (*Sufficiency under EDF/NPD*) *A set of periodic tasks that share lock-free objects is schedulable under the EDF scheme if the following condition holds.*

$$\langle \forall t : t \geq 0 : \sum_{j=0}^{N-1} \left\lfloor \frac{t + p_j - l_j}{p_j} \right\rfloor \cdot c_j + \sum_{j=0}^{N-1} \left\lfloor \frac{t - 2 + p_j - l_j}{p_j} \right\rfloor \cdot s \leq t \rangle$$

Proof: We prove that if a task set is not schedulable then the following expression holds for some $t \geq 0$.

$$\sum_{j=0}^{N-1} \left\lfloor \frac{t + p_j - l_j}{p_j} \right\rfloor \cdot c_j + \sum_{j=0}^{N-1} \left\lfloor \frac{t - 2 + p_j - l_j}{p_j} \right\rfloor \cdot s > t$$

Assume that the given task set is not schedulable. Let $J_{i,k}$ be the first job to miss its deadline. (If several jobs simultaneously miss their deadline along with $J_{i,k}$, then let $J_{i,k}$ be the one with lowest priority.) We begin by deriving a bound on $D(b_i(k), d_i(k) - 1)$, the total demand placed on the processor by $J_{i,k}$ and higher-priority jobs, i.e., jobs with deadlines at or before $d_i(k)$, in the interval $[b_i(k), d_i(k))$. $D(b_i(k), d_i(k) - 1)$ is comprised of the demand placed by job releases and the extra demand placed by interferences. Recall that at $J_{i,k}$'s busy point, all jobs of equal or higher priority either are inactive or have a job release. Each job of some task T_j can place a demand of c_j on the processor, and there are at most $\lfloor (d_i(k) - b_i(k) + p_j - l_j)/p_j \rfloor$ job releases of that task in the interval $[b_i(k), d_i(k))$ that have a deadline at or before $d_i(k)$. Therefore, the total demand placed on the processor due to such jobs is at most

$$\sum_{j=0}^{N-1} \left\lfloor \frac{d_i(k) - b_i(k) + p_j - l_j}{p_j} \right\rfloor \cdot c_j.$$

By Lemma 3.4, the total number of interferences in jobs with deadlines at or before $d_i(k)$ in the interval $[b_i(k), d_i(k))$ is bounded by the term $\sum_{j=0}^{N-1} \lfloor (d_i(k) - b_i(k) - 2 + p_j - l_j)/p_j \rfloor$. By IA2, each interference requires s units of additional demand. Therefore, the total additional demand due to interferences is at most

$$\sum_{j=0}^{N-1} \left\lfloor \frac{d_i(k) - b_i(k) - 2 + p_j - l_j}{p_j} \right\rfloor \cdot s.$$

Job $J_{i,k}$ will miss its deadline if and only if the difference between the total demand due to tasks with a deadline at or before $d_i(k)$ in the interval $[b_i(k), d_i(k))$ and the available processor time in that interval is greater than one. Therefore, we have

$$\sum_{j=0}^{N-1} \left\lfloor \frac{d_i(k) - b_i(k) + p_j - l_j}{p_j} \right\rfloor \cdot c_j + \sum_{j=0}^{N-1} \left\lfloor \frac{d_i(k) - b_i(k) - 2 + p_j - l_j}{p_j} \right\rfloor \cdot s - (d_i(k) - b_i(k) - 1) > 1,$$

which can be rewritten as

$$\sum_{j=0}^{N-1} \left\lfloor \frac{d_i(k) - b_i(k) + p_j - l_j}{p_j} \right\rfloor \cdot c_j + \sum_{j=0}^{N-1} \left\lfloor \frac{d_i(k) - b_i(k) - 2 + p_j - l_j}{p_j} \right\rfloor \cdot s > d_i(k) - b_i(k).$$

Replacing $(d_i(k) - b_i(k))$ by t yields the following expression, where $0 \leq t < d_i(k) - b_i(k)$; this completes the proof.

$$\sum_{j=0}^{N-1} \left\lfloor \frac{t + p_j - l_j}{p_j} \right\rfloor \cdot c_j + \sum_{j=0}^{N-1} \left\lfloor \frac{t - 2 + p_j - l_j}{p_j} \right\rfloor \cdot s > t \qquad \square$$

As formulated above, the expression in Theorem 3.4 cannot be verified because the range of t is unbounded. However, there is an implicit bound on t. For example, if all tasks are released at time 0, then we only need to consider values less than or equal to L the least common multiple of the task periods. On the other hand, if task T_i is released at

time v_i, where $v_i \geq 0$ and $v_i \neq v_j$ for some i and j, then we need to consider values of t less than or equal to $2 \cdot L + max_{0 \leq i < N}(v_i) + max_{0 \leq i < N}(l_i)$.

In [21], Baruah, Howell, and Rosier t showed that the range of t can be further reduced if these exists an upper bound on processor utilization. Specifically, Baruah et. al. show that if there exists an upper bound U on processor utilization, then values of t range over the interval $(0, \frac{U}{1-U} max_{0 \leq i < N}(p_i - l_i)]$. Fortunately, in practical systems such as the videoconferencing system considered in Chapter 6, there exists an implicit upper bound on U. In such systems, system overhead costs due to context switching, interrupt processing, etc., consume a non-negligible of processor utilization.

We now prove a sufficient condition for EDF scheduling for the special case when $p_j = l_j$ for any task T_j. The following condition states that a task set is schedulable if processor utilization is at most one, when interferences are taken into account.

Theorem 3.5: (*Sufficiency under EDF*) *A set of periodic tasks that share lock-free objects is schedulable under the EDF scheme if the following condition holds.*

$$\sum_{j=0}^{N-1} \frac{c_j + s}{p_j} \leq 1$$

Proof: We prove that if a task set is not schedulable then $\sum_{j=0}^{N-1}(c_j + s)/p_j > 1$ holds. Assume that the given task set is not schedulable. Because the EDF scheme is a special case of the EDF/NPD scheme, it follows from Theorem 3.4 that the following expression holds for some $t \geq 0$.

$$\sum_{j=0}^{N-1} \left\lfloor \frac{t + p_j - l_j}{p_j} \right\rfloor \cdot c_j + \sum_{j=0}^{N-1} \left\lfloor \frac{t - 2 + p_j - l_j}{p_j} \right\rfloor \cdot s > t$$

Because $p_j = l_j$ for all T_j, we have the following.

$$\sum_{j=0}^{N-1} \left\lfloor \frac{t}{p_j} \right\rfloor \cdot c_j + \sum_{j=0}^{N-1} \left\lfloor \frac{t-2}{p_j} \right\rfloor \cdot s > t$$

The above expression implies

$$\sum_{j=0}^{N-1} \frac{t}{p_j} \cdot c_j + \sum_{j=0}^{N-1} \frac{t}{p_j} \cdot s > t.$$

Cancelling t from both sides of the inequality yields the following, thus completing the proof.

$$\sum_{j=0}^{N-1} \frac{c_j + s}{p_j} > 1 \qquad \square$$

The conditions derived so far assume that the cost of executing all retry loops is uniform. The above conditions allow us to check the schedulability without knowledge of the tasks' object accesses. However, when retry-loop costs can vary widely, the above conditions are not very accurate, i.e., for many task sets, they are liable to provide incorrect predictions on task schedulability. To improve the accuracy of these conditions, we now derive scheduling conditions to account for the actual costs of the various retry loops while deriving the scheduling conditions.

3.5 Accounting for Different Retry-Loop Costs

The task model described previously is inadequate when different retry-loop costs are considered because it does not incorporate knowledge about the different object access costs of each task. We now present a task model that is more suitable for the purpose of accounting for different retry-loop costs.

3.5.1 Additional Notation and Definitions

We assume that each job is composed of distinct nonoverlapping computational fragments or *phases*. Each phase is either a *computation phase* or an *object-access phase*. Shared objects are not accessed during a computation phase. An object-access phase consists of exactly one retry loop in which one or more objects are accessed. The cost of an object-access phase is equal to the cost of its associated retry loop.

The following is a list of symbols that will be used repeatedly in deriving our scheduling conditions. These symbols are using in conjunction with some of the notation described earlier in Section 3.1.

- $w(i)$ - The number of phases in a job of task T_i. The phases are numbered from 1 to $w(i)$. We use u and v to denote phases.

- c_i^v - The worst-case computational cost of the v^{th} phase of task T_i's job, where $1 \leq v \leq w(i)$, assuming no contention for the processor or shared objects. We denote total cost over all phases by $c_i = \sum_{v=1}^{w(i)} c_i^v$.

- $m_j^{i,v}(t)$ - The worst-case number of interferences in T_i's v^{th} phase due to T_j in an interval of length t.

- f_i^v - An upper bound on the number of interferences of the retry loop in the v^{th} phase of T_i during a single execution of that phase.

As in Sections 3.3 and 3.4, we obtain scheduling conditions by determining the worst-case demand of each task, and by accounting for the cost of operation interferences. As in our previous model, T_j's unfulfilled demand at time t decreases by one if it executes on

the processor at that, and it increases by c_j if it has a job release at that time. Interference assumption IA1 still holds under the new task model. However, IA2 does not hold because we do not assume that all retry loop costs are uniform. Hence, Assumption IA2 is redefined as follows for the rest of this chapter.

IA2 *Each interference experienced by the v^{th} phase of job $J_{i,k}$ increases T_i's unfulfilled demand by $s_j^{i,v}$.*

IA2 holds because if the v^{th} phase of a job of T_i is interfered with, then $s_j^{i,v}$ units of additional demand is placed on the processor, because another execution of the retry-loop iteration in T_i's v^{th} phase is required. Note that IA2 does not distinguish between computation phases and object-access phases. This is because we appropriately define $s_j^{i,v}$ to account for the fact that computational phases cannot be interfered with. Formally, $s_j^{i,v}$ is defined as follows.

Definition 3.1: Let T_i and T_j be two distinct tasks, where T_i has at least v phases. Let z_j denote the set of objects modified by T_j, and a_i^v denote the set of objects accessed in the v^{th} phase of T_i. Then,

$$s_j^{i,v} = \begin{cases} c_i^v & \text{if } j < i \ \wedge \ a_i^v \cap z_j \neq \emptyset \\ 0 & \text{otherwise.} \end{cases}$$ □

If phase v of task T_i is an object-access phase, then the cost of an interference in v equals c_i^v — the cost of the retry loop associated with this phase — if T_j can interfere

with the operation in the v^{th} phase of T_i, where $j < i$. Hence, if phase v of task T_i is a computation phase, then the cost of an interference in v is zero.

3.5.2 Bounding Interference Cost

Before deriving scheduling conditions for the various schemes, we outline a general approach to bound the total additional demand due to interferences over an interval $\mathcal{I}$. To this end, we define an expression that gives the *exact* worst-case cost of interferences in tasks T_0 through T_i in any interval of length t.

Definition 3.2: The total cost of interferences in jobs of tasks T_0 through T_i in any interval of length t, denoted $E_i(t)$, is defined as follows.

$$E_i(t) \equiv \sum_{j=0}^{i} \sum_{v=1}^{w(j)} \sum_{l=0}^{j-1} m_l^{j,v}(t) s_l^{j,v}.$$

□

The term $m_l^{j,v}(t)$ in the above expression denotes the worst-case number of interferences caused in T_j's v^{th} phase by jobs of T_l in an interval of length t. The term $s_l^{j,v}$ represents the amount of additional demand required if T_l interferes once with T_j's v^{th} phase. The expression within the leftmost summation denotes the total cost of interferences in a task T_j over all phases of all jobs of T_j in an interval of length t.

Expression $E_i(t)$ accurately reflects the worst-case additional demand placed on the processor in an interval $\mathcal{I}$ of length t due to interferences in tasks T_0 through T_i. Of course, to evaluate this expression, we first must determine values for the $m_l^{j,v}(t)$ terms. Unfortunately, in order to do so, we potentially have to examine an exponential number

of possible task interleavings in the interval $\mathcal{I}$. Instead of exactly computing $E_i(t)$, our approach is to obtain a bound on $E_i(t)$ that is as tight as possible. We do this by viewing $E_i(t)$ as an expression to be maximized. The $m_l^{j,v}(t)$ terms are the "variables" in this expression. These variables are subject to certain constraints. We obtain a bound for $E_i(t)$ by using integer linear programming to determine a maximum value of $E_i(t)$ subject to these constraints. We now explain how appropriate constraints on the $m_l^{j,v}(t)$ variables are obtained.

3.5.3 Static-Priority Scheduling Schemes

In this explanation, we focus on the DM scheme. Later, we explain how similar constraints can be obtained for other schemes. We impose three sets of constraints on the $m_j^{i,v}(t)$ variables. All of these constraints are straightforward. However, the third constraint involves terms (f_i^v) that are not completely straightforward to calculate. Most of the rest of this subsection is devoted to explaining how these terms are computed. For a set of tasks scheduled under the DM scheme, and an interval of length t, the three sets of constraints are as follows.

Constraint Set 1:

$$(\forall i, j : j < i :: \sum_{v=1}^{w(i)} m_j^{i,v}(t) \leq \left\lceil \frac{t+1}{p_j} \right\rceil).$$

Constraint Set 2:

$$(\forall i :: \sum_{j=0}^{i} \sum_{v=1}^{w(j)} \sum_{l=0}^{j-1} m_l^{j,v}(t) \leq \sum_{j=0}^{i-1} \left\lceil \frac{t+1}{p_j} \right\rceil).$$

Constraint Set 3:

$$(\forall i, v :: \sum_{j=0}^{i-1} m_j^{i,v}(t) \leq \left\lceil \frac{t+1}{p_i} \right\rceil f_i^v).$$

The first set of constraints follows because the number of interferences in jobs of T_i due to T_j in an interval $\mathcal{I}$ of length t is bounded by the maximum number of jobs of T_j that can be released in $\mathcal{I}$. The second set of constraints follows from Lemma 3.1, which states that the total number of interferences in jobs of tasks T_i and higher-priority tasks in an interval $\mathcal{I}$ of length t is bounded by the maximum number of jobs of tasks T_0 through T_{i-1} released in $\mathcal{I}$. In the third set of constraints, the term f_i^v is an upper bound on the number of interferences of the retry loop in the v^{th} phase of T_i during a single execution of that phase. The details of calculating f_i^v are described later. The reasoning behind this set of constraints is as follows. If at most f_i^v interferences can occur in the v^{th} phase of a job of T_i, and if there are n jobs of T_i released in an interval $\mathcal{I}$, then at most nf_i^v interferences can occur in the v^{th} phase of T_i in $\mathcal{I}$.

We use an inductive approach to calculate f_i^v for any i and v. This inductive approach is expressed in pseudo-code in Figure 3.2. The *compute_retries* procedure in this figure computes all f_i^v values. This procedure begins by setting f_0^v to zero for all phases of T_0 (line 1). This is because, by Axiom 3.1 and our ordering on tasks, operations of T_0 can never be interfered with. We then calculate the f values for tasks T_1 through T_{N-1}, respectively. If the v^{th} phase of task T_i is a computation phase, then f_i^v is set to zero (line 5) because a computation phase cannot be interfered with. Lines 6 through 10 are executed if the v^{th} phase of task T_i is an object-access phase. In this case, we first calculate a bound R_1 on the maximum time it takes to execute phase v, given that at most k interferences of phase v can occur (line 7). We then calculate a bound R_2 on the maximum time it takes to execute phase v, given that at most $k+1$ interferences of phase v can occur (line 8). The

```
    procedure compute_retries()
1:  for v := 1 to w(0) do f_0^v := 0 od;                    /* Task T_0 cannot be interfered with */
2:  for i := 1 to N − 1 do
3:      for v := 1 to w(i) do                                /* Consider each phase v of task T_i */
4:          if T_i's v^th phase is a computational phase then
5:              f_i^v := 0                                   /* Computational phase cannot be interfered with */
            else                                             /* Phase v is an object-access phase */
6:              for k := 0 to ∞ do
7:                  R_1 := (min t :: c_i^v + Σ_{j=0}^{i−1} ⌈(t − 1)/p_j⌉c_j + int_cost(i, v, k, t − 1) ≤ t);
8:                  R_2 := (min t :: c_i^v + Σ_{j=0}^{i−1} ⌈(t − 1)/p_j⌉c_j + int_cost(i, v, k + 1, t − 1) ≤ t);
9:                  if R_2 ≥ p_i then f_i^v := ∞; break fi;          /* Period exceeded, give up */
10:                 if R_2 = R_1 then f_i^v := k; break fi           /* At most k interferences can occur */
                od
            fi
        od
    od
```

/* Procedure *int_cost* computes bound on **i**nterference **c**osts in T_i and higher-priority tasks in an interval of length t during the v^{th} phase of T_i in which T_i is interfered with at most k times */

procedure $int_cost(i, v, k, t)$ **returns integer**

return the maximum value of
$\sum_{j=0}^{i-1} m_j^{i,v}(t) s_j^{i,v} + \sum_{j=0}^{i-1} \sum_{u=1}^{w(j)} \sum_{l=0}^{j-1} m_l^{j,u}(t) s_l^{j,u}$
subject to the following constraints:

(a) $\sum_{j=0}^{i-1} m_j^{i,v}(t) \leq k$

(b) $(\forall j : 0 \leq j < i :: m_j^{i,v}(t) \leq \left\lceil \frac{t+1}{p_j} \right\rceil)$

(c) $(\forall j, l : j < l < i :: \sum_{u=1}^{w(l)} m_j^{l,u}(t) \leq \left\lceil \frac{t+1}{p_j} \right\rceil)$

(d) $(\forall l : l < i :: \sum_{j=0}^{l} \sum_{v=1}^{w(j)} \sum_{l'=0}^{j-1} m_{l'}^{j,v}(t) \leq \sum_{j=0}^{l-1} \left\lceil \frac{t+1}{p_j} \right\rceil)$

(e) $(\forall l, u : 0 \leq l < i :: \sum_{j=0}^{l-1} m_j^{l,u}(t) \leq \left\lceil \frac{t+1}{p_l} \right\rceil f_l^u)$

Figure 3.2: Pseudo-code to calculate f_i^v values.

manner in which R_1 and R_2 are determined is described below. If R_2 exceeds the period of task T_i, then we have failed to find a constraint that can be imposed on the number of interferences in phase v (line 9). If R_1 equals R_2, then phase v of T_i can experience at most k interferences (line 10).

We now explain the manner in which R_1 is determined; R_2 is calculated in a similar manner. R_1 is assigned a value t that is an upper bound on the length of an interval that includes $n \leq k+1$ iterations of phase v of task T_i; the interval begins with the first statement execution in the first iteration of phase v, and ends with the last statement execution of the n^{th} execution of phase v. In line 7, the first component in the left-hand side of the inequality denotes the portion of time in the interval that is taken to execute the last iteration of phase v. The second component denotes the time spent executing jobs of higher-priority tasks, excluding interferences in those tasks. (In the interval of length t in question, there can be no higher-priority job releases at the first point in the interval, and any such job released at the $(t+1)^{st}$ point in the interval executes after the interval. This is why $t-1$ appears in this expression.) The third component denotes an upper bound on the additional time spent executing additional iterations of loops that have been interfered with in T_i's v^{th} phase and in higher-priority tasks.

The third component is calculated by invoking *int_cost*(i, v, k, t), which determines an upper bound on the interference cost in tasks T_0 through T_{i-1} and the v^{th} phase of task T_i in an interval of length t in which T_i is interfered with at most k times. Determining an exact bound is difficult, so we use integer linear programming within *int_cost* to obtain an upper bound. The constraints for *int_cost*(i, v, k, t) only use f values of tasks T_0 through

T_{i-1}, so there is no circularity.

The maximum value of the expression given in *int_cost* is determined subject to five constraint sets, labeled (a) through (e). The set of constraints labeled (a) follows from our definition of the interval to be determined. For example, for R_1, the interval ends with the completion of the n^{th} iteration of phase v of task T_i, where $n \leq k+1$. The set of constraints labeled (b) follows from the fact that the number of times a higher-priority task T_j can interfere with T_i's v^{th} phase in an interval is bounded by the number of jobs of T_j released in that interval. The rest of the constraint sets are similar to Constraint Sets 1 through 3 given earlier. We now show that the bounds returned by *compute_retries* are correct by proving the following lemma.

Lemma 3.5: The value returned for f_i^v by *compute_retries* is an upper bound on the number of times the v^{th} phase of T_i can be interfered with in a single job of T_i.

Proof: We consider the execution of the loop at lines 6 through 10 in Figure 3.2 for a given i and v. If f_i^v is assigned a value at line 9, then the lemma is clearly true for i and v. In the remainder of the proof, we assume that f_i^v is assigned a value in line 10. (Showing that the loop eventually terminates and assigns a value to f_i^v is straightforward.) Suppose that the value m is assigned to f_i^v in line 10, and let R_1^m and R_2^m denote the values computed in lines 7 and 8, respectively, during the final loop iteration. By line 10, these two values are equal. Also, from the discussion in Section 3.5.2, it is fairly easy to see that R_1^m (and hence R_2^m) is an upper bound on the time it takes to complete $n \leq m+1$ iterations of phase v in T_i.

We now prove that there cannot be a job of T_i in which $h > m$ interferences occur

in T_i's v^{th} phase. Suppose, to the contrary, that there exists such a job. Suppose the v^{th} phase in this job begins execution at time t_0, and that the n^{th} loop iteration of T_i's v^{th} phase completes at time t_n, where $1 \leq n \leq h+1$. Now, consider the $(m+1)^{st}$ loop iteration in $[t_0, t_{h+1}]$. Because R_1^m is an upper bound on the worst-case time taken to execute $m+1$ loop iterations, we have $R_1^m \geq t_{m+1} - t_0$. Furthermore, because $h > m$, the $(m+1)^{st}$ iteration in $[t_0, t_{m+1}]$ fails, i.e., phase v of T_i is interfered with $m+1$ times in $[t_0, t_{m+1}]$. This implies that phase v of T_i is interfered with at least $m+1$ times in $[t_0, t_0 + R_1^m]$. We now show that there can be at most m interferences of phase v of T_i in *any* interval of length R_1^m, which will give the desired contradiction.

Let W be the set of higher-priority tasks that can interfere with the v^{th} phase of T_i. By Definition 3.1, $s_j^{i,v}$ equals c_i^v if $j \in W$, and $s_j^{i,v}$ equals zero if $j \notin W$. With this notation, we can rewrite the first term of the expression to maximize in *int_cost* as $c_i^v \sum_{j \in W} m_j^{i,v}(t)$. Constraint (a) in *int_cost* can be rewritten as $\sum_{j \in W} m_j^{i,v}(t) \leq k$. Also, constraint (b) implies $\sum_{j \in W} m_j^{i,v}(t) \leq \sum_{j \in W} \lceil (t+1)/p_j \rceil$. Hence, the maximum value of $c_i^v \cdot \sum_{j \in W} m_j^{i,v}(t)$ is given by $c_i^v \cdot min(k, \sum_{j \in W} \lceil (t+1)/p_j \rceil)$.

We now make three important observations. First, in the expression maximized by *int_cost*, the first summation term is contingent on constraints (a) and (b) only, and the second summation term depends on constraints (c) through (e) only. Second, the expression maximized by *int_cost*(i, v, k, $t-1$) is identical to that maximized by *int_cost*(i, v, $k+1$, $t-1$). Third, the constraints for these calls are the same, except for constraint (a).

Now, consider the procedure calls *int_cost*(i, v, m, $t-1$) and *int_cost*(i, v, $m+1$, $t-1$) in lines 7 and 8 when R_1^m and R_2^m, respectively, are determined. Based on the

observations made in the previous paragraph and the fact that the values returned by these two calls are the same, $c_i^v \cdot min(m, \sum_{j \in W} \lceil R_1^m / p_j \rceil)$ equals $c_i^v \cdot min(m+1, \sum_{j \in W} \lceil R_2^m / p_j \rceil)$. Hence, $\sum_{j \in W} \lceil R_1^m / p_j \rceil = \sum_{j \in W} \lceil R_2^m / p_j \rceil < m+1$. This implies that the number of higher-priority jobs in any interval of length R_1^m that can interfere with T_i's v^{th} phase is less than $m+1$, a contradiction. □

The constraints considered so far apply not only to DM scheduling, but to any static-priority scheduling scheme. We now present similar constraints can be derived for the EDF and EDF/NPD schemes.

3.5.4 Dynamic-Priority Scheduling Schemes

The constraint sets for the EDF/NPD scheme is very similar to those derived for the DM scheme. By comparing the constraints below to those for static-priority schemes, we see that the constraint sets differ only in the terms on the right-hand side of each constraint. This is because the number of jobs with higher priority than $J_{i,k}$ released in any interval of length t is different for these two schemes. The following constraint sets hold for the EDF/NPD scheme.

Constraint Set 4:

$$(\forall i, j : j < i :: \sum_{v=1}^{w(i)} m_j^{i,v}(t) \leq \left\lfloor \frac{t + p_j - l_j}{p_j} \right\rfloor).$$

Constraint Set 5:

$$(\forall i :: \sum_{j=0}^{i} \sum_{v=1}^{w(j)} \sum_{l=0}^{j-1} m_l^{j,v}(t) \leq \sum_{j=0}^{i-1} \left\lfloor \frac{t + p_j - l_j}{p_j} \right\rfloor).$$

Constraint Set 6:

$$(\forall i, v :: \sum_{j=0}^{i-1} m_j^{i,v}(t) \leq \left\lfloor \frac{t + p_i - l_i}{p_i} \right\rfloor f_i^v).$$

The first set of constraints follows because the number of interferences in jobs of T_i due to T_j in an interval $\mathcal{I}$ of length t is bounded by the maximum number of jobs of T_j released in $\mathcal{I}$ with deadlines at or before the end of that interval. The second set of constraints follows from Lemma 3.1. Hence, the total number of interferences in jobs of tasks with deadline at or before the end of an interval $\mathcal{I}$ of length t is bounded by the maximum number of jobs of released in $\mathcal{I}$ with deadlines before the end of that interval. The reasoning behind the third set of constraints is similar to that for Constraint Set 3. The *compute_retries* and *int_cost* procedures for static-priority schemes can be directly used to derive f_i^v values under EDF/NPD. However, we must use ceilings instead of floors in these procedures because the actual task deadlines are not considered when bounding interferences in its phases.

3.6 Static-Priority Scheduling Conditions

In this section, we derive scheduling conditions for static- and dynamic-priority schemes based on the results derived in the previous section. Specifically, we present sufficient scheduling conditions for the RM, DM, EDF, and EDF/NPD schemes. The necessary conditions for these schemes are identical to those derived in Sections 3.3 and 3.4. In each of these conditions, we give an expression that represents the maximum demand in an interval $\mathcal{I}$ of length t, and require that total demand over $\mathcal{I}$ is less than or equal to the available processor time in $\mathcal{I}$. The expression for demand consists of two components: the first represents demand due to job releases, and the second represents demand due to interferences. Recall that $E_i(t)$ is the actual worst-case cost of interferences in jobs of tasks T_0 through

T_i in any interval of length t. We let $E_i'(t)$ denote a bound on $E_i(t)$ that is determined as described in the previous subsection. The scheduling condition for the DM scheme is as follows.

Theorem 3.6: Under the DM scheme, a set of tasks is schedulable if the following holds for every task T_i.

$$(\exists t : 0 < t \leq l_i :: \sum_{j=0}^{i} \left\lceil \frac{t}{p_j} \right\rceil c_j + E_i'(t-1) \leq t)$$

Proof: We prove that if a task set is not schedulable, then the negation of the above expression holds. Let the k^{th} job of some task T_i be the first to miss its deadline. Consider any t in the interval $[b_i(k), d_i(k))$. We begin by deriving a bound on $D(b_i(k), t)$, the total demand placed on the processor by T_i and higher-priority tasks in the interval $[b_i(k), t]$. $D(b_i(k), t)$ is comprised of the demand placed by job releases and the extra demand placed by interferences. Recall that at the busy point $b_i(k)$, T_i and all higher-priority tasks are either inactive or have a job release. Each job release of some task T_j introduces a demand of c_j on the processor, and there are at most $\lceil (t - b_i(k) + 1)/p_j \rceil$ job releases of that task in the interval $[b_i(k), t]$. Therefore, the total demand placed on the processor due to job releases of T_i and higher-priority tasks is at most $\sum_{j=0}^{i} \lceil (t - b_i(k) + 1)/p_j \rceil c_j$.

The total cost of interferences in jobs of T_i and higher-priority tasks in the interval $[b_i(k), t]$ is given by $E_i(t)$. As mentioned earlier, the term $E_i'(t)$ denotes a bound on $E_i(t)$. Therefore, the total additional demand due to interferences in jobs of T_i and higher-priority tasks is at most $E_i'(t)$. Therefore, we have

$$D(b_i(k), t) \leq \sum_{j=0}^{i} \left\lceil \frac{t - b_i(k) + 1}{p_j} \right\rceil c_j + E_i(t - b_i(k)).$$

Job $J_{i,k}$ will miss its deadline if and only if, at time $d_i(k) - 1$, T_i has positive unfulfilled demand and the total unfulfilled demand of T_i and higher-priority tasks is greater than one. By Lemma 3.3, it follows that the difference between the total demand due to T_i and higher-priority tasks in the interval $[b_i(k), t]$ and the available processor time in that interval is greater than one. Hence, we have the following.

$$D(b_i(k), t) - (t - b_i(k)) > 1$$

Using the bound on $D(b_i(k), t)$ derived above, the previous expression implies the following.

$$\sum_{j=0}^{i} \left\lceil \frac{t - b_i(k) + 1}{p_j} \right\rceil c_j + E_i(t - b_i(k)) > t - b_i(k) + 1$$

The above expression holds for all t in the interval $[b_i(k), d_i(k))$. Because this expression is independent of the end points (it is a function of the length of the interval), we can replace $t - b_i(k)$ with t', where $t' = t - b_i(k)$ and $t' \in [0, d_i(k) - b_i(k))$. Hence, we have the following.

$$\sum_{j=0}^{i} \left\lceil \frac{t' + 1}{p_j} \right\rceil c_j + E_i(t') > t' + 1$$

Now, replace t' with t in the above expression, where $t = t' + 1$ and $t \in (0, d_i(k) - b_i(k)]$. Also, $E_i'(t') \geq E_i(t')$. Therefore, the following holds for all $t \in (0, d_i(k) - b_i(k)]$.

$$\sum_{j=0}^{i} \left\lceil \frac{t}{p_j} \right\rceil c_j + E_i'(t - 1) > t$$

By definition, $b_i(k) \leq r_i(k)$. Therefore, the interval $(0, d_i(k) - r_i(k)]$ is completely contained in $(0, d_i(k) - b_i(k)]$. Also, by definition, $d_i(k) - r_i(k) = l_i$. Therefore, the expression above holds for all t in $(0, l_i]$. □

Because RM scheduling is a special case of DM scheduling, the above scheduling condition also holds for the RM scheme.

3.7 Dynamic-Priority Scheduling Conditions

We now derive a sufficient condition for schedulability of task set scheduled under the EDF/NPD scheme. In the expression below, the first and second terms on the left-hand side of the inequality denote the demand due to jobs of all tasks released during an interval of length t and the total interference cost in all tasks during that interval.

Theorem 3.7: Under the EDF/NPD scheme, a set of tasks is schedulable if the following holds.

$$(\forall t :: \sum_{j=0}^{N-1} \left\lfloor \frac{t+p_j-l_j}{p_j} \right\rfloor c_j + E'_{N-1}(t-1) \leq t)$$

Proof: We prove that if a task set is not schedulable then $\sum_{j=0}^{N-1} \left\lfloor \frac{t+p_j-l_j}{p_j} \right\rfloor \cdot c_j + E'_{N-1}(t-1) > t$ holds for some $t \geq 0$. Assume that the given task set is not schedulable. Let $J_{i,k}$ be the first job to miss its deadline. (If several jobs simultaneously miss their deadline along with $J_{i,k}$, then let $J_{i,k}$ be the one with lowest priority.) We begin by deriving a bound on $D(b_i(k), d_i(k) - 1)$, the total demand placed on the processor by $J_{i,k}$ and higher-priority jobs, i.e., jobs with deadlines at or before $d_i(k)$, in the interval $[b_i(k), d_i(k))$. $D(b_i(k), d_i(k) - 1)$ is comprised of the demand placed by job releases and the extra demand placed by interferences. Recall that at $J_{i,k}$'s busy point, all jobs of equal or higher priority either are inactive or have a job release. Each job of some task T_j can place a demand of c_j on the

processor, and there are at most $\lfloor (d_i(k) - b_i(k) + p_j - l_j)/p_j \rfloor$ job releases of that task in the interval $[b_i(k), d_i(k))$ that have a deadline at or before $d_i(k)$. Therefore, the total demand placed on the processor due to such jobs is at most

$$\sum_{j=0}^{N-1} \left\lfloor \frac{d_i(k) - b_i(k) + p_j - l_j}{p_j} \right\rfloor \cdot c_j.$$

The exact total additional demand due to interferences in all jobs with deadlines at or before $d_i(k)$ in the interval $[b_i(k), d_i(k))$ is given by $E_{N-1}(d_i(k) - b_i(k) - 1)$. Therefore, the total additional demand due to interferences is at most $E'_{N-1}(d_i(k) - b_i(k) - 1)$.

Job $J_{i,k}$ will miss its deadline if and only if the difference between the total demand due to tasks with a deadline at or before $d_i(k)$ in the interval $[b_i(k), d_i(k))$ and the available processor time in that interval is greater than one. Therefore, we have

$$\sum_{j=0}^{N-1} \left\lfloor \frac{d_i(k) - b_i(k) + p_j - l_j}{p_j} \right\rfloor \cdot c_j + E'_{N-1}(d_i(k) - b_i(k) - 1) - (d_i(k) - b_i(k) - 1) > 1,$$

which can be rewritten as

$$\sum_{j=0}^{N-1} \left\lfloor \frac{d_i(k) - b_i(k) + p_j - l_j}{p_j} \right\rfloor \cdot c_j + E'_{N-1}(d_i(k) - b_i(k) - 1) > d_i(k) - b_i(k).$$

Replacing $(d_i(k) - b_i(k))$ by t yields the following expression, where $0 \leq t < d_i(k) - b_i(k)$, thus completing the proof.

$$\sum_{j=0}^{N-1} \left\lfloor \frac{t + p_j - l_j}{p_j} \right\rfloor \cdot c_j + E'_{N-1}(t - 1) > t \qquad \square$$

As formulated above, the expression in Theorem 3.4 cannot be verified because the value of t is unbounded. Nonetheless, as explained in Section 3.4, t is bounded in practice.

Chapter 4

Support for Strong Primitives

As explained in Chapter 2, Herlihy's hierarchy implies that hardware support for strong primitives[1] to implement lock-free objects in general asynchronous systems. Nevertheless, we present several wait-free implementations of strong primitives from weaker primitives, for uniprocessor real-time systems. Our results are based on the fact that priority-based scheduling schemes used in hard real-time systems give rise to only a subset of the task interleavings possible in general asynchronous systems. In particular, we show that, in uniprocessor real-time systems, the consensus problem can be solved for any number of tasks using only load and store instructions. Our results imply that Herlihy's hierarchy collapses in uniprocessor real-time systems, and that sophisticated hardware support is not required to implement lock-free objects in such systems. In fact, using the universal constructions described in [36, 42], it is possible to implement any lock-free object using only consensus protocols. However, we eschew such an approach to implementing lock-free objects because practical implementations of such objects are usually based on

[1] Primitives with unbounded consensus number.

strong primitives such as CAS. Towards this end, we present wait-free implementations of the single-word and multi-word[2] CAS (MWCAS) primitives.

This chapter is organized as follows. First, we present our real-time task model, followed by a description of notation used in the correctness proofs of our implementations. Then, we present our solution to the N-task consensus problem using only load and store instructions. We also present two implementations of single-word CAS: the first is based on load and store instructions, and the second uses a *memory-to-memory move* (move) instruction in addition to load and store instructions. Finally, we present an implementation of the MWCAS primitive that uses load, store, and single-word CAS instructions.

4.1 The Real-Time Task Model

The basis for the results presented in this chapter is the realization that the use of priority-based schedulers in hard real-time systems precludes the occurrence of certain task interleavings. For example, if a task T_i accesses an object in the time interval $[t, t']$, and if another task T_j on the same processor accesses that object in the interval $[u, u']$, then it is not possible to have $t < u < t' < u'$, because the higher-priority task must finish its operation before relinquishing the processor. Requiring an object implementation to correctly deal with this interleaving is pointless because it cannot arise in practice. The distinction between traditional asynchronous systems, to which Herlihy's work is directed, and hard real-time systems is illustrated in Figure 4.1. Operations of different tasks in a traditional asynchronous system can arbitrarily overlap one another. In contrast, in a real-

[2]Multi-word CAS extends the semantics of single-word CAS to multiple words.

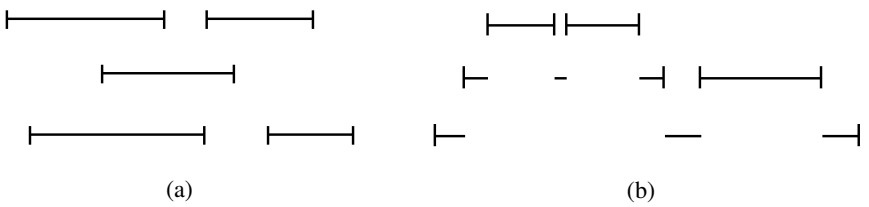


Figure 4.1: (a) Interleaved operations in a asynchronous system. (b) Interleaved operations in a uniprocessor real-time system. Line segments denote operations on shared objects with time running from left to right. Each level corresponds to operations by a different task.

time system, operations of higher-priority tasks are completely contained within operations of lower-priority tasks.

The real-time task model we assume is characterized by two key requirements: (i) on a given processor, a task T_i may preempt another task T_j only if T_i has higher priority than T_j; (ii) a task's priority can change over time, but not during any object access. Requirement (i) is fundamental to all priority-driven scheduling policies. When tasks are scheduled under such policies, a high-priority task can arbitrarily preempt any low-priority task executing on the same processor. Requirement (ii) holds for most common scheduling policies, including RM, DM, and EDF scheduling and variations of these in which tasks are broken into phases that are allowed to have distinct priorities [33]. The only common scheduling policy that we know of that violates requirement (ii) is LLF scheduling [65]. Under LLF scheduling, the priority of a task invocation *can* change during its execution.

Observe that requirements (i) and (ii) expressly preclude the use of locking within a processor. In particular, if a high-priority task T_i attempts to lock an object that is already locked by a low-priority task T_j, then to ensure progress, T_j must be allowed to resume execution. To ensure that requirement (i) is met, T_j's priority must be raised to

exceed T_i's before T_j is resumed. However, raising T_j's priority during its object access violates requirement (ii). An object implementation in this task model is *lock-free* (*wait-free*) iff, for any set of tasks with overlapping operation executions, some (each) task in that set completes its operation in a bounded number of its own steps.

4.2 Definitions and Notation

We begin by describing our model of program execution. We assume that shared objects are accessed or modified via a fixed set of operations. We assume that operations on shared objects are implemented as procedure calls, and that such operations are invoked by a fixed collection of tasks. We model the execution of these tasks by state transitions. A *state* is a mapping that assigns a value to all shared memory locations and to all private task variables (including the program counter for each task). A *history* is a totally-ordered sequence of states, with each pair of consecutive states separated by a task *step* that causes the computation to go from the first state to the second. Thus, in the history $s_0 \xrightarrow{q.1} s_1 \xrightarrow{p.1} s_2 \xrightarrow{p.2}$ ..., the initial state is s_0; the execution of task T_q's first statement causes the computation to go from state s_0 to state s_1; the execution of task T_p's first statement causes the computation to go from state s_1 to state s_2, and so on. We assume that each labeled statement within a procedure is atomic, i.e., causes a single transition between a pair of states when executed. When we refer to an interval of states [t,u], we are referring to a subsequence in the execution history of a task set that starts at state t and ends at state u. We say that an operation *executes within* the interval $[t, u]$ iff the first statement execution of that operation occurs after state t and the last statement execution of that

operation occurs before state u.

The correctness condition we use for our lock-free and wait-free implementations is linearizability [39]. An implementation of an object is *linearizable* if, in every history h, the partial order over the operation invocations in h can be extended to a total order such that the sequence of operations in the total order is consistent with the sequential semantics of the implemented operations. In our linearizability proofs, we show that this total order exists by defining, for each operation invocation, a unique point in time, called its *linearization point* (different invocations of the same operation can have different linearization points). We say that an invocation is *linearized to* its linearization point. We also define a "current" value of the implemented object. We then show that, at the linearization point of each invocation, the value of the implemented object before and after that point is consistent with the semantics of the implemented operation, and that the invocation returns the same value as the sequential operation would if executed atomically at that linearization point.

We abstract away from the task scheduler by defining an *environment* that determines task priorities and enables tasks to execute on the processor. The behavior of the environment is contingent on the underlying scheduling scheme. In this chapter, we consider only systems in which the environment has the following properties: (*i*) it does not modify task variables or program counters; (*ii*) it does not modify the priority of a task during a shared object access; (*iii*) it assigns unique priorities to the different tasks. Note that this implies that if a given task begins executing one of the numbered statements in our procedures (e.g., see Figure 4.2), then no lower-priority task may execute any statement until that procedure invocation completes. In the proofs of the implementations that

follow, we will only concern ourselves with statement executions that arise from invoking our procedures, i.e., we abstract away from the other activities of the tasks invoking these procedures.

The space complexity of our algorithms is measured in terms of the total space required, which includes the space required for shared variables and for private variables. The time complexity of our lock-free algorithms is measured in terms of *contention-free time complexity*, which is the time taken to perform an operation in the absence of contention. (Lock-free objects have unbounded execution time in the presence of contention, but are required to terminate in the absence of contention.)

Notation. Unless stated otherwise, q and r are assumed to be universally quantified over task identifiers, and t, u, and v are assumed to be states. If s is a statement label, then the predicate $q@s$ holds iff the statement with label s is the next statement to be executed by task T_q. We use $q@S$, where S is a set of statement labels, as shorthand for $(\exists s : s \in S :: q@s)$. Unless stated otherwise, labeled program fragments are assumed to be atomic. Each such fragment accesses at most one shared variable. We use $q.s$ to denote the statement with label s of task T_q, and $q.v$ to denote T_q's local variable v. If a task T_p is not executing one of our procedures, we consider it to be executing within a *remainder section*, which is denoted by statement $p.0$. We use the term $en(p)$ to indicate that some statement of task T_p is enabled for execution, i.e., T_p is the highest-priority task executing on the processor. We require that if a given task has an enabled statement at a state, then no lower-priority task has an enabled statement at that state.

For any state assertions B and C, B *unless* C denotes an *unless* property. Formally,

```
shared var Final : valtype ∪ ⊥
initially Final = ⊥
procedure incorrect-decide(val : valtype) returns valtype
1 : if Final = ⊥ then
2 :     Final := val
    fi;
3 : return Final
```

Figure 4.2: Incorrect solution to consensus using loads and stores.

B *unless* C holds iff for each statement, if $B \wedge \neg C$ holds before that statement is executed, then $B \vee C$ holds after (intuitively, if B holds, then it must continue to hold unless C eventually holds).

4.3 Universality of Load and Store Instructions

In this section, we show that load and store instructions are universal in a uniprocessor system by solving the consensus problem for an arbitrary number of tasks using only these instructions. The main difficulty to be faced in our solution to the consensus problem is the "enabled late-write problem". We explain this problem by considering an incorrect solution to the consensus problem.

In Figure 4.2, procedure *incorrect-decide* gives an incorrect solution to the consensus problem. Procedure *incorrect-decide* uses a variable *Final*, in which the decision value is stored. Each task that does not detect another task's input in *Final* (line 1) writes its own value into *Final* (line 2). The procedure returns the value in *Final* (line 3). The main problem with this solution is that two tasks may reach different decisions. This problem arises from the fact that once a task T_p is enabled to modify *Final* at $p.2$, it is committed to do so even if it is preempted by a higher-priority task T_q. Consider Figure 4.3. In this figure,

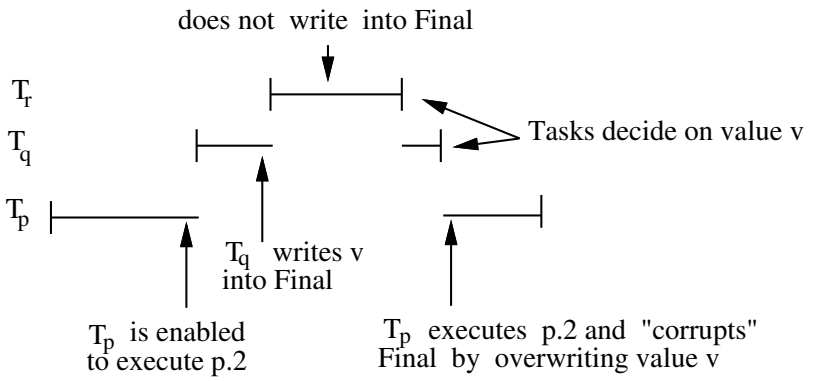


Figure 4.3: The enabled late-write problem.

T_q and T_r complete their entire operations and decide on some value v, before relinquishing the processor to T_p. When T_p resumes execution, it chooses a value different from v because its enabled late-write overwrites the value in *Final*.

In solving the consensus problem, we overcome the late-write problem by means of a two-step procedure depicted in Figure 4.4. In this figure, procedure *decide* uses two shared variables, *Propose* and *Final*. In the first step of the procedure, a task "proposes" its value in *Propose*, if no other task has already written into *Propose*. In the second step, the task copies the value in *Propose* to *Final*. This two-step procedure solves the late-write problem by ensuring that if task T_p is committed to writing a variable *Propose*, then before it does so, the first (and correct) value written to *Propose* has already been copied to another variable *Final*, which T_p cannot subsequently modify. In Figure 4.4, procedure *decide* solves consensus on a real-time uniprocessor using loads and stores. Each task that does not detect the input of another task in *Propose* or *Final* writes its own value into *Propose*. Having ensured that some value has been proposed (lines 1 to 3), a task copies the proposed value to *Final*, if necessary (lines 4 to 6); it is easy to see that no task returns

```
shared var Final, Propose : valtype ∪ ⊥
private var v : valtype
initially Final = ⊥ ∧ Propose = ⊥
procedure decide(val : valtype) returns valtype
1 : if Final = ⊥ then
2 :     if Propose = ⊥ then
3 :         Propose := val
        fi;
4 :     if Final = ⊥ then
5 :         v := Propose;
6 :         Final := v
        fi
    fi;
7 : return Final
```

Figure 4.4: Consensus using loads and stores.

before some task's input value is written into *Final*, and that all tasks return a value read from *Final*. The following lemma implies that all tasks return the same value, and therefore yields the theorem below.

Lemma 4.1: *The first value written into Propose is the only value written into Final.*

Proof: Let T_p be the first task to write its value $p.val$ into *Propose*. Recall that no higher-priority task is executing when $p.3$ is executed. Thus, if task T_q executes between $p.3$ and $p.7$, then $q.1$ occurs after $p.3$, which implies that T_q does not modify *Propose*. Therefore, $Propose = p.val$ holds continuously between $p.3$ and $p.7$. (Refer to Figure 4.5.)

Now, consider how some task T_q can modify *Final* after $p.3$ (note that T_q could be T_p). If T_q starts execution after $p.7$, then $q.1$ reads $Final \neq \perp$, so T_q does not write *Final*. If T_q starts execution before $p.3$ is executed, and $q \neq p$, then by the assumption that $p.3$ writes *Propose* first, $q@\{1..3\}$ holds when $p.3$ is executed. This implies that T_q has lower priority than T_p, and therefore does not execute again until after T_p completes execution. Thus, the test at $q.4$ fails and T_q does not write *Final*. Therefore, if task T_q writes *Final*

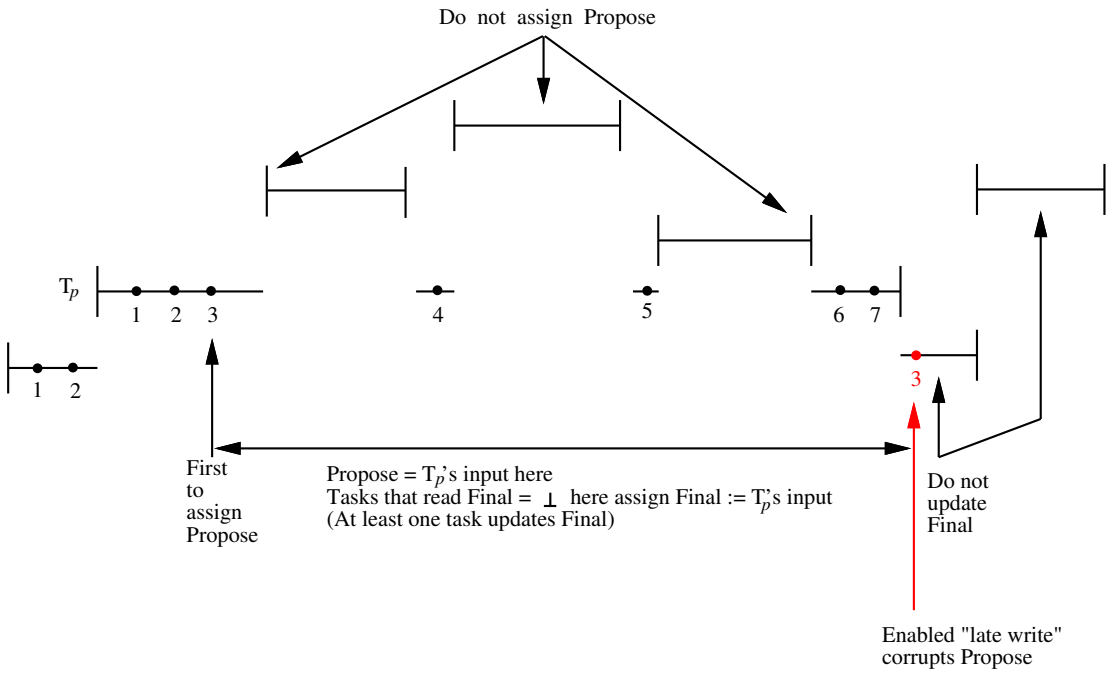


Figure 4.5: Proof of Lemma 4.1.

after $p.3$, then either $p = q$ or T_q starts execution between $p.3$ and $p.7$. The latter implies that T_q has higher priority than T_p, which implies that T_q completes execution before $p.7$. In either case, because $Propose = p.val$ holds continuously between $p.3$ and $p.7$, $q.5$ establishes $q.v = p.val$ and $q.6$ establishes $p.val = Final$. □

Theorem 4.1: *Consensus can be implemented with constant time and space using loads and stores on a real-time uniprocessor system.*

Proof: By Lemma 4.1, exactly one value is written into *Final* and all tasks decide on the value in *Final*. Hence, it follows that all tasks decide on the same value. □

Using consensus objects, any shared object can be implemented in a wait-free

(and hence lock-free) manner [36, 42]. However, such implementations usually entail high overhead; practical wait-free and lock-free implementations are typically based on primitives such as CAS and LL/SC. To enable the use of such practical implementations in real-time uniprocessor systems, we present two implementations of a shared object that supports Read and CAS operations.[3] (LL/SC can be implemented using Read and CAS in constant time [8].) These implementations use load/store and move instructions, respectively.

The key problem to be faced in our implementations is the problem of enabled late writes described in Section 4.3. Each of our implementations employ a different technique to solve this problem. In the CAS implementation described in Section 4.4, an enabled late-write instruction can potentially modify the current value of the shared object. The late-write problem is handled by replicating the value of the shared object in $2N - 1$ variables, where N denotes the number of tasks accessing the object. The "current" value of the object is given by the value found in a majority of the $2N - 1$ variables. A task T_q ensures that the same value is written to all of $2N - 1$ shared variables before relinquishing the processor to lower-priority tasks. Each lower-priority task can have at most one pending late-write operation and can corrupt one of the $2N - 1$ variables, potentially. Thus, even if all $N - 1$ of the lower-priority tasks subsequently modify one of these variables, the value written by T_q is still in a majority of the $2N - 1$ variables.

The CAS implementation presented in Section 4.5 uses move, load, and store instructions. This implementation uses *Propose* and *Final* variables like in our consensus algorithm described previously. The value of the implemented object is contained in shared

[3]Henceforth, we use the term "Read" to denote a read operation on any implemented shared object; this distinguishes such read operations from reads of shared variables.

variable *Final*. To change the value of the shared object, a task first proposes a value in *Propose* and uses a move instruction to copy *Propose* to *Final*. An enabled late-move operation of a low-priority task can potentially overwrite the value of the object by copying *Propose* to *Final*. To ensure that enabled late moves of low-priority tasks do not overwrite *Final*, each task ensures that the same value is stored in *Final* and *Propose* before relinquishing the processor.

4.4 Implementing CAS using Loads and Stores

Our load/store implementation of Read and CAS for N tasks is given in Figure 4.6,[4] where $N \geq 2$. Values of the implemented object are stored in the *Val* array, which contains one element $Val[p]$ for each task T_p. The component of *Val* that holds the current value of the object is determined from the array *Buf*, which contains $2N-1$ task identifiers. Component $Rv[r]$ of array *Rv* is used by T_r to detect if its CAS operation was successful. Similarly, component $Pm[r]$ of array *Pm* is used by T_r to detect whether its CAS or Read operation has been preempted and interfered with. The shared variable V is used by tasks to communicate return values of Read operations invoked by lower-priority tasks.

The following informal definition states that the current "value" CV of the shared object is determined by the task identifier that is stored in a majority of the locations $Buf[1]$ through $Buf[2N-1]$. (A formal definition of CV is given later.)

$$CV \equiv (Val[p] :: p \textit{ is a majority in } Buf)$$

[4]For simplicity, the shared object is not explicitly passed as a parameter to the Read and CAS procedures shown in this figure.

```
shared var
    Val : array [0..N − 1] of valtype;        /* If r is majority identifier in Buf, Val[r] is current value */
    Buf : array [1..2N − 1] of 0..N − 1;      /* Array of task identifiers used to determine current value */
    Rv : array [0..N − 1] of valtype ∪ ⊥;     /* Indicates success/failure of a CAS operation */
    Pm : array [0..N − 1] of boolean;         /* Flag Pm[p] is set when
                                                 a higher-priority task modifies CV during T_p's operation */
    V : valtype          /* If Buf changes during a Read operation m, then V contains return value of m */
initially (∀k :: Buf[k] = 0) ∧ Val[0] = initial value
private var                                   /* For Task T_p, where 0 ≤ p ≤ N − 1 */
    i : integer; w, maj : 0..N − 1; current : valtype; count : array [0..N − 1] of 0..2N − 1

procedure Read() returns valtype
1 :  Pm[p], count := false, (0, ..., 0);
2 :  for i := 1 to 2N − 1 do
3 :  w, count[w] := Buf[i], count[w] + 1
     od;
4 :  maj := select q : (∀r :: count[q] ≥ count[r]);
5 :  current := Val[maj];
6 :  if Pm[p] then
7 :      current := V
     fi;
8 :  return current

procedure CAS(old, new) returns boolean
9 :  Pm[p] := false;
10 : Rv[p], count := ⊥, (0, ..., 0);
11 : for i := 1 to 2N − 1 do
12 :     w := Buf[i]; count[w] := count[w] + 1
     od;
13 : maj := select q : (∀r :: count[q] ≥ count[r]);
14 : current := Val[maj];

/* CAS continued.... */
15 : if ¬Pm[p] ∧ current = old then
16       if old = new then return true fi;
17 :     if maj ≠ p then Rv[maj], i := current, 1;
18 :         while ¬Pm[p] ∧ i ≤ 2N − 1 do
19 :             Buf[i], i := maj, i + 1
             od
         fi;
20 :     Val[p], i := new, 1;
21 :     while ¬Pm[p] ∧ i ≤ 2N − 1 do
22 :         Buf[i], i := p, i + 1
         od;
23 :     if i > 2N − 1 ∨ Rv[p] = new then
24 :         V := new;
25 :         for i := 0 to N − 1 do
26 :             Pm[i] := true
             od;
27 :         return true
         fi
     fi;
28 : return false
```

Figure 4.6: Implementation of CAS using loads and stores.

In order to perform a CAS operation, a task T_p first attempts to determine CV. This is achieved by clearing T_p's "preempted" flag $Pm[p]$ (line 9 in Figure 4.6), reading the entire Buf array, and counting the task identifiers read (line 11-13). Then, T_p attempts to find a majority maj among the identifiers read. (Because T_p does not read Buf atomically, it is possible that T_p does not find a majority, but in any case, T_p chooses *some* valid task identifier.) T_p then reads $Val[maj]$ (line 14). It can be shown that if $Pm[p]$ is set between $p.9$ and $p.15$, then the value of CV changed in this interval (see Lemma 4.3). In this case, $p.old \neq CV$ holds either before or after the change of CV, so T_p's CAS can fail at that point. Also, if $Pm[p]$ is *not* set between $p.9$ and $p.15$, then it can be shown that no task modifies Buf in this interval (see Lemma 4.2). In this case, T_p correctly determines the majority maj and therefore correctly determines CV at line 14. If $p.current$ — the value determined for CV — differs from $p.old$ then T_p fails immediately (line 15). Also, if $p.current = p.old$ and $p.old = p.new$, then T_p can succeed (line 16), because its successful CAS would not modify the implemented object. This leaves CAS operations that determine that $CV = p.old$, and that must attempt to change CV from $p.old$ to $p.new$. Such operations can succeed by first writing $Val[p] := p.new$ (line 20), and by then establishing T_p as the majority in Buf (lines 21 and 22).

A Read operation determines CV (lines 1 to 5) the same way as a CAS operation does (lines 9 to 14). If Buf changes while the Read operation executes lines 2 and 3, then it can be shown (see Lemma 4.17) that the value read from V (line 7) is the value of CV at some point during the Read.

Several subtle difficulties arise in this implementation. First, it is important to

ensure that the majority is not changed by "late" writes to *Buf*. This could potentially occur if a task is preempted while writing *Buf*, and continues writing an "old" value when it resumes running. This possibility is avoided by ensuring that, when a new majority q is established, q is written into all $2N-1$ locations of *Buf* before the next majority is established, and by also ensuring that each of at most $N-1$ lower-priority tasks can write only one "late" value. (Observe that this ensures that at least N elements of *Buf* still contain q.) The latter is achieved by having successful CAS operations set the *Pm* flags of all tasks (lines 25 and 26), and having each task check its *Pm* flag before proceeding with each write (lines 18 and 21). Thus, each task can perform at most one "late" write before detecting the overlapping operation and failing. In order to ensure that all $2N-1$ locations of *Buf* are written whenever a new majority is established (the task that established the new majority may have been preempted before finishing its writes to *Buf*), each task writes the majority read to all elements of *Buf* (lines 18 and 19) before attempting to change the majority to its own identifier (lines 21 and 22).

As explained above, if T_p detects that $Pm[p]$ is true before establishing itself as a majority, then T_p can fail, because it can be shown that (see Lemma 4.7) an overlapping CAS changed the value of *CV*. However, it is possible for a task T_p to succeed in making its own identifier the majority, and to be subsequently preempted and therefore fail to detect that it achieved a majority. In this case, the fact that T_p achieved a majority is "communicated" to T_p via $Rv[p]$ (line 17). Then, when T_p executes line 23, it detects that it did achieve a majority, and hence that it succeeded.

Example: Figure 4.7 depicts the effects of three CAS operations m_0, m_1, and m_2 by tasks

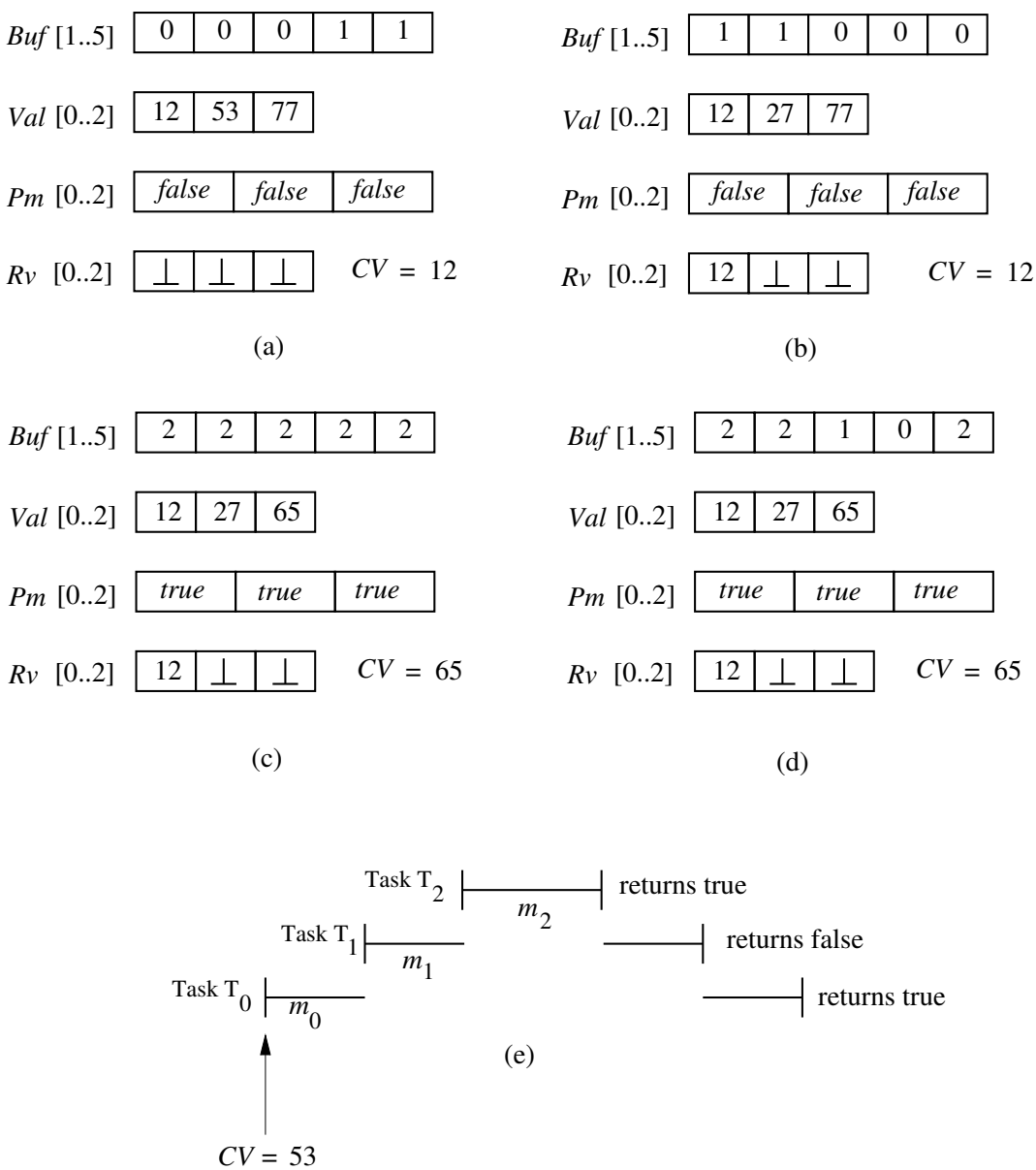


Figure 4.7: Tasks invoke CAS operations m_0, m_1, and m_2 with old/new values 53/12, 12/27, and 12/65, respectively. The contents of the relevant variables are shown (a) at the beginning of T_1's operation; (b) at the beginning of T_2's operation; (c) at the end of T_2's operation; (d) at the end of T_0's operation. The corresponding operation interleaving is shown in (e).

T_0, T_1, and T_2, respectively. The old/new values of the m_0, m_1, and m_2 are 53/12, 12/27, and 12/65, respectively. The value of CV equals 53 when m_0 begins execution. Inset (a) shows the contents of the various variables just before m_1 begins. At this time, CV equals 12 and T_0 is enabled to write zero into $Buf[4]$. (Note that m_0 has established a majority in Buf by this point.) During m_1, task T_1 detects zero as the majority task identifier in Buf (lines 11 and 12 in Figure 4.6). Then, m_1 informs T_0 that m_0 succeeded by assigning m_0's new value (12) to $Rv[0]$ (line 17). Task T_1 then assigns zero to all components of Buf (lines 18 and 19). However, before T_1 can establish a new majority identifier in Buf (lines 21 and 22), it is preempted by T_2. (T_1 is enabled to write into $Buf[3]$ at the preemption point.) Insets (b) and (c) show the relevant variables at the beginning and at the end of operation m_2, respectively. T_2 successfully changes CV by writing its new value (65) into $Val[2]$ and by writing its task identifier (2) into all components of Buf. Before returning, T_2 writes *true* into all components of Pm to inform lower-priority tasks that their operations have been interfered with. Inset (d) shows the relevant variables at the termination of m_0. Observe that CV does not change when the enabled late writes of T_0 and T_1 modify $Buf[4]$ and $Buf[3]$, respectively. Also, as expected, when m_1 (m_0) resumes execution, T_1 fails (T_0 succeeds) the test at line 23 and returns false (true). □

4.4.1 Correctness Proof

Before we present some properties required of our implementation, we first define a few terms used in the correctness proof of our CAS implementation.

Definition 4.1: $NUM(p) \equiv |\{n \ : \ 1 \leq n \leq 2N-1 \ :: \ Buf[n] = p\}|$. □

Definition 4.2: $MAJ(p) \equiv (\forall q \ : \ q \neq p \ :: \ NUM(p) > NUM(q))$. □

Definition 4.3: $CV \equiv (Val[p] :: MAJ(p))$ □

According to Definition 4.1, $NUM(p)$ denotes the number of locations in Buf that contain task identifier T_p. According to Definition 4.2, $MAJ(p)$ is a boolean expression that evaluates to true if a majority of the locations in Buf contain task identifier T_p; it evaluates to false otherwise. Definition 4.3 formally defines the current value of the implemented object.

The correctness proof of our CAS implementation proceeds as follows. We first state the properties required of our implementation. Then, we establish several lemmas required to prove the stated properties.

Property 4.1: A CAS operation invoked by task T_r returns *true* from line 27 iff one of the following holds.

(*i*) $\neg Pm[r] \ \wedge \ CV = r.old \ \wedge \ r.maj = r$ holds immediately before line 20 is executed, and $CV = r.new$ holds immediately after.

(*ii*) $\neg Pm[r] \ \wedge \ CV = r.old \ \wedge \ r.i = N$ holds immediately before line 22 is executed and $CV = r.new$ holds immediately after. □

Property 4.2: A CAS operation invoked by task T_r returns from line $r.16$ only if $CV = r.old \ \wedge \ r.old = r.new$ holds immediately before $r.15$ is executed. □

Property 4.1 implies that each successful CAS operation that changes CV lin-

earizes either to line 20 or to line 22. Property 4.2 implies that each successful CAS operation that does not change CV linearizes to line 15.

Property 4.3: A CAS operation invoked by a task T_r returns from line 28 only if $CV \neq r.old$ holds at some point during that operation. □

Property 4.3 states that if a CAS fails, then the operation linearizes to some state in which the operation's old value does not match the current value of the implemented object.

Property 4.4: The value returned by a Read operation equals the value of CV at some point during that operation. □

Property 4.4 states that a Read operation can be linearized to some state during that operation at which the value of the implemented shared object equals the return value. Before proving the above properties, we first state and prove following lemmas.

Lemma 4.2: *If* $r@\{2..6, 10..24\} \wedge \neg Pm[r]$ *holds when a statement of* T_r *is enabled for execution, then no other task modifies Buf or Val during* T_r*'s operation.*

Proof: Suppose that $r@\{2..6, 10..24\} \wedge \neg Pm[r]$ holds at some state u during an operation m invoked by T_r, when a statement of T_r is enabled for execution. Let t be the state immediately following the execution of $r.1$ or $r.9$ by m. The proof is by contradiction. Assume that some higher-priority task invokes a CAS operation that modifies *Buf* and *Val* during the interval $[t, u]$. (Note that Read operations do not modify *Buf* or *Val*.) Let m' be a CAS operation that modifies *Buf* and *Val* and that is invoked by T_q the highest-priority

task to execute during the interval $[t, u]$. It follows from the real-time task model that T_q's operation is completely contained within $[t, u]$ and that no other task modifies *Buf* or *Val* during T_q's operation. Also, no other task executes lines 25 and 26 during m'. (If some task executes lines 25 and 26 during m', then that task also modifies *Val*.) Therefore, $\neg Pm[q]$ is not falsified by another task after the execution of $q.9$ until T_q executes $q.26$ when $q.i = q$. From the program (lines 18-22), it follows T_q successfully updates all locations of *Buf*. Hence, T_q succeeds the test at $q.23$ and writes *true* into $Pm[r]$ before it returns from m'. Also, the execution of any statement other than $r.9$ ($r.1$) cannot establish $\neg Pm[r]$ after m' completes and before state u. Therefore, $Pm[r]$ holds at state u, a contradiction. □

Lemma 4.3: *If* $r@\{2..6, 10..24\} \ \wedge \ Pm[r]$ *holds when a statement of* T_r *is enabled for execution, then some other task modifies Buf and Val during* T_r*'s operation.*

Proof: Suppose that $r@\{2..6, 10..24\} \ \wedge \ Pm[r]$ holds at some state u during an operation m invoked by T_r, when a statement of T_r is enabled for execution. Let t be the state immediately following the execution of $r.1$ or $r.9$ by m. Then, T_r is preempted by some higher-priority task that executes line 26 during the interval $[t, u]$. Consider T_q the highest-priority task to execute line 26 that preempts T_r during $[t, u]$, i.e., $\neg Pm[q]$ holds throughout T_q's operation. From the program text (lines 18-22), it follows that T_q successfully updates $Val[q]$ and *Buf*. □

Lemma 4.4: $\neg Pm[r] \ \wedge \ r@\{21..22\} \Rightarrow \ (r.maj = r \ \wedge \ NUM(r) = 2N - 1) \ \vee \ (r.maj \neq r \ \wedge \ NUM(r) = r.i - 1 \ \wedge \ NUM(r.maj) = 2N - r.i)$.

Proof: Suppose that $\neg Pm[r] \ \wedge \ r@\{21, 22\}$ holds at state u. Let t be the state immediately

following the execution of $r.9$. By Lemma 4.2, no other task modifies *Buf* or *Val* during $[t, u]$, which, by Lemma 4.3 implies that $\neg Pm[r]$ holds throughout $[t, u]$. From the program code, we see that T_r establishes $NUM(r.maj) = 2N - 1$ by writing into all components of *Buf* at $r.18$ and .19 before it executes $r.20$. We need to consider the following two cases.

Case: 1 If $r.maj = r$ holds before the execution of $r.20$, then $r.20$ establishes $\neg Pm[r] \wedge r@\{21..22\} \wedge NUM(r.maj) = 2N-1 \wedge NUM(r) = 2N-1 \wedge r.i = 1$. Hence, $NUM(r.maj) = 2N-1 \wedge NUM(r) = 2N-1$ holds before T_r executes $r.21$ for the first time and the execution of $r.21$ and $r.22$ does not affect $NUM(r)$. Because no other task modifies *Buf* until state u, this implies that $NUM(r) = 2N - 1$ holds at state u.

Case: 2 If $r.maj \neq r$ holds before the execution of $r.20$, then $r.20$ establishes $\neg Pm[r] \wedge r@\{21..22\} \wedge NUM(r.maj) = 2N - 1 \wedge NUM(r) = 0 \wedge r.i = 1$. Hence, $NUM(r) = r.i - 1 \wedge NUM(r.maj) = 2N - r.i$ holds before T_r executes $r.21$ for the first time. After each loop iteration, both $NUM(r)$ and $r.i$ increase by one and $NUM(r.maj)$ decreases by one. Because no other task modifies *Buf* until state u, this implies that $NUM(r) = r.i - 1 \wedge NUM(r.maj) = 2N - r.i$ holds at state u. □

Lemma 4.5: *In every state, there exists some T_q such that $MAJ(q)$ holds.*

Proof: Initially, the lemma holds because $MAJ(0)$ holds. We now inductively show that if there is a unique majority in *Buf* in all states until some state v and the execution of a statement by T_r results in a transition from state v to state v', then there exists a unique majority in state v'. We only need to consider statements of T_r that modify *Buf*, namely $r.19$ and $r.22$.

Let t be the state immediately following the execution of $r.9$. Observe that $r.19$ and $r.22$ are executed only if $\neg Pm[r]$ held previously at $r.18$ and $r.21$, respectively. If $\neg Pm[r]$ holds at $r.18$ or $r.21$, then, by Lemma 4.2, no other task modifies *Buf* while T_r reads it at $r.11$ and $r.12$. Therefore, The majority task identifier read by T_r from *Buf* at $r.11$ and $r.12$ is assigned to $r.maj$ at $r.13$, i.e., $MAJ(r.maj)$ holds after $r.13$. (Because a majority exists in *Buf* until state v, there is a unique majority in *Buf* after $r.13$.) We now consider statements $r.19$ and $r.22$.

Statement $r.19$: Suppose that T_r executes $r.18$ and $r.19$ at states u and v, respectively, and that $r.i = k$ holds at state u and state v, where $1 \leq k \leq 2N - 1$. Statement $r.19$ destroys the majority in *Buf* only if, for some q, $MAJ(q) \wedge r.maj \neq q$ holds at state v. However, T_r executes $r.19$ at state v only if $\neg Pm[r]$ holds at state u. By Lemma 4.2, no other task modifies *Buf* during $[t, u]$. This, when combined with the fact that $MAJ(r.maj)$ holds when $r.13$ is executed, implies that $MAJ(r.maj)$ holds at state u. Because $MAJ(q) \wedge r.maj \neq q$ holds at state v, it follows that *Buf* is modified by some other task during $[u, v]$. Let m' be the last CAS operation to execute statement 9 during $[u, v]$ that modifies *Buf* and that is not preempted by any other task that modifies *Buf*. Let $T_{q'}$ be the task that invokes m'. (Refer to Figure 4.8.)

Because no other task modifies *Buf* or *Val* during m', it follows from Lemma 4.3 that $\neg Pm[q']$ is not falsified by some other task during m'. From the program code (lines 20-26), it can be seen that $T_{q'}$ writes q' into all components of *Buf* and ensures that $(\forall p : Pm[p])$ holds before returning from m'. By our assumption about $T_{q'}$, tasks that execute after m', and before state v, have lower priority than $T_{q'}$ and have already executed

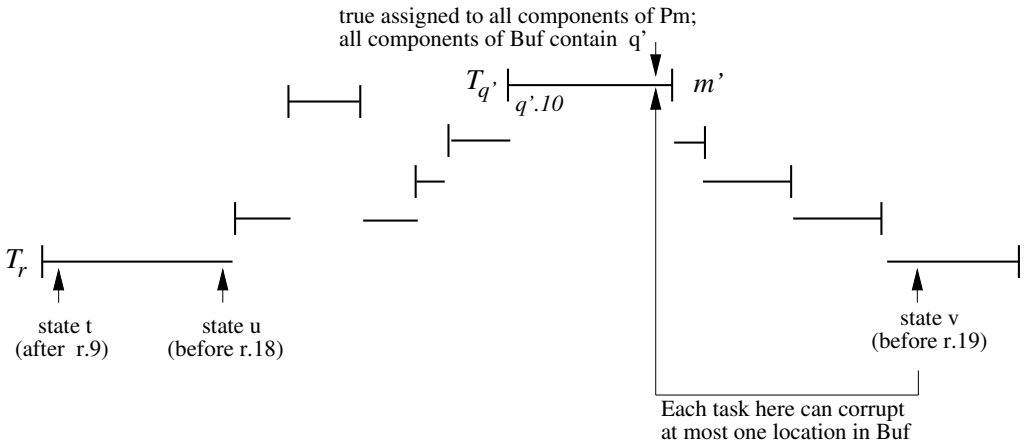


Figure 4.8: Proof of Lemma 4.5.

statement 9. Other than T_r, there are at most $N-2$ such tasks, and each such task can modify at most one component of *Buf* before failing the test at line 18 or 21. Hence, at most $N-2$ components of *Buf* contain values different from q' at state v, i.e., at least $N+1$ components of *Buf* contain q' at state v. Therefore, the execution of $r.19$ at state v will not falsify $MAJ(q')$.

Statement $r.22$: Suppose that $r.22$ is executed at state v when $r.i = k$ holds, where $1 \leq k \leq 2N-1$. We consider two possibilities.

$\neg Pm[r]$ *holds at state* v: If $r.maj = r \ \wedge \ r.i = k$ holds at state v, then, by Lemma 4.4, the execution of $r.22$ establishes $\neg Pm[r] \ \wedge \ r@21 \ \wedge \ r.i = k+1 \ \wedge \ NUM(r) = 2N-1$, which implies that $MAJ(r)$ holds at state v'. On the other hand, if $r.maj \neq r \ \wedge \ r.i = k$ holds at state v, then, by Lemma 4.4, $Pm[r] \wedge r@21 \wedge r.i = k+1 \wedge NUM(r) = k \wedge NUM(r.maj) = 2N-k-1$ holds at state v'. This implies that either $MAJ(r.maj)$ or $MAJ(r)$ holds at state v' depending on whether $1 \leq k < N$ or $N \leq k \leq 2N-1$.

$Pm[r]$ holds at state v: Suppose that T_r executes $r.21$ at state u. Observe that T_r executes $r.22$ at state v only if $\neg Pm[r]$ holds at state u. By Lemmas 4.2 and 4.3, it follows that *Buf* is modified by some other task during $[u, v]$. By following an argument similar to the one in the proof of statement $r.19$, we can show that the execution of $r.22$ does not destroy the majority in *Buf*. □

For the rest of the proofs in this section, we assume that a unique majority always exists in *Buf* — we do not explicitly quote Lemma 4.5 to establish this fact.

Lemma 4.6: *Task T_r cannot change the majority in Buf by executing $r.19$.*

Proof: Similar to the proof of statement $r.19$ in Lemma 4.5. □

Lemma 4.7: *If $r@\{2..6, 10..24\} \wedge Pm[r]$ holds, then some task T_q returns from line 27 during T_r's operation.*

Proof: Suppose that $r@\{2..6, 10..24\} \wedge Pm[r]$ holds during a CAS or Read operation m invoked by task T_r. Then, some higher-priority task executes line 26 during m and returns from line 27. □

Lemma 4.8: $r@\{10..24\} \wedge Pm[r] \Rightarrow \neg MAJ(r)$.

Proof: Suppose that T_r invokes a CAS operation m. Let t be the state immediately following the execution of $r.9$ by m, and let v be some state during m at which $r@\{10..24\} \wedge Pm[r]$ holds. By Lemma 4.3, some higher-priority task updates *Buf* in $[t, v]$. Using an argument similar to the one in the proof of statement $r.19$ of Lemma 4.5, we can show

that a CAS operation of some higher-priority task $T_{r'}$ updates all components of *Buf* and establishes $(\forall p : Pm[p])$ before it returns, and that tasks with priority between T_r and $T_{r'}$ can write a value different from r' in at most $N-2$ locations before state v. Task T_r can also write a value different from r' into at most one component of *Buf* when it resumes execution. This implies that $MAJ(r')$ holds at state v. □

Lemma 4.9: $MAJ(r) \ \wedge \ r@\{21..22\} \ \wedge \ \neg Pm[r]$ *unless* $Rv[r] = Val[r] \ \vee \ r.i > 2N-1$.

Proof: Let $A \equiv MAJ(r) \ \wedge \ r@\{21..22\} \ \wedge \ \neg Pm[r]$. Our proof obligation is to show that, for any enabled statement s, the following Hoare triple holds.

$$\{A \ \wedge \ \neg(Rv[r] = Val[r] \ \vee \ r.i > 2N-1)\} \ s \ \{A \ \vee \ (Rv[r] = Val[r] \ \vee \ r.i > 2N-1)\} \quad (4.1)$$

Among statements of task T_r, we only need to consider $r.21$ and $r.22$ because they can falsify A. Statement $r.21$ falsifies A by establishing $r@23$ only if $r.i > 2N-1$ holds before the execution of $r.21$. Statement $r.22$ cannot falsify $MAJ(r)$ because T_r assigns r into *Buf*. Therefore, (4.1) holds if s is a statement of T_r.

Among statements of some higher-priority task T_q, we only need to consider $q.19$, $q.22$, and $q.26$. (4.1) holds if s is $q.19$ because, by Lemma 4.6, $q.19$ cannot falsify $MAJ(r)$. However, there is a danger that $q.22$ can falsify $MAJ(r)$, or that $q.26$ can falsify $\neg Pm[r]$. (Note that the real-time task model implies that T_q's operation is completely contained within T_r's operation.) To complete the proof, we show that if s is $q.22$ or $q.26$, then the precondition in (4.1) does not hold.

Suppose that $q.22$ ($q.26$) is executed at state u. Let t be the last state before u at which a statement of T_r is executed. Suppose that $A \ \wedge \ \neg(Rv[r] = Val[r] \ \vee \ r.i > 2N-1)$

holds at state t. If $Pm[r]$ holds at state u, then, by Lemma 4.8, $MAJ(r)$ does not hold at state u. On the other hand, if $\neg Pm[r]$ holds at state u, then no other task assigns *true* to $Pm[q]$ during the interval $[t, u]$. To see why this is so, assume that $\neg Pm[r]$ holds at state u and that some higher-priority task updates $Pm[q]$ during $[t, u]$. Then, by Lemma 4.7, some higher-priority task $T_{q'}$ returns from line 27 during T_q's operation before state u. From the program text (lines 25 and 26), it follows that $(\forall p : Pm[p])$ holds when $T_{q'}$ returns from its operation. This implies that $Pm[r]$ holds at state u, a contradiction. Therefore, no other task writes *true* into $Pm[q]$ during $[t, u]$. Having established this, we now consider statements $q.22$ and $q.26$.

Statement $q.22$: Suppose that $q.22$ is executed at state u when $\neg Pm[r]$ holds. Because no other task writes *true* into $Pm[q]$ during $[t, u]$, and because T_q does not write into $Pm[q]$ until the execution of $q.26$, it follows that $\neg Pm[r] \ \wedge \ \neg Pm[q]$ holds at state u. By Lemma 4.2, no other task modifies *Buf* or *Val* during T_q's execution until state u. Because $MAJ(r)$ holds at state u, this implies that $MAJ(r)$ holds throughout T_q's operation. It follows that T_q detects r as the majority in *Buf* at $q.11$ and $q.12$. Then, $q.13$ establishes $r = q.maj$, $q.14$ establishes $Val[r] = q.current$, and $q.17$ establishes $Val[r] = Rv[r]$. Furthermore, $Val[r]$ is not modified by any task after $q.17$ and before state u. Therefore, $Rv[r] = Val[r]$ holds at state u.

Statement $q.26$: Suppose that statement $q.26$ is executed at state u, when $\neg Pm[r]$ holds. Because no higher-priority task assigns *true* to $Pm[q]$ in the interval $[t, u]$, it follows that $\neg Pm[q]$ holds during T_q's operation at least until $q.25$ is executed. From the program code (lines 21 and 22), it follows that T_q assigns q to all $2N - 1$ components of *Buf*. Because T_r

executes no statements in $[t, u]$, this implies that $MAJ(r)$ does not hold at state u. □

In our implementation, the value of CV does during the execution of an operation by task T_r as long as $\neg Pm[r]$ holds when T_r is enabled to execute. This is formalized by the following two lemmas. The $en(r)$ terms in the expressions in these lemmas are required because, even if $\neg Pm[r] \wedge r@\{16..20\}$ holds, these expressions do not hold in the interval between the time some higher-priority task T_q preempts T_r and changes CV and the time $Pm[r]$ is established by some higher-priority task.

Lemma 4.10: $en(r) \wedge \neg Pm[r] \wedge r@\{16..20\} \Rightarrow r.current = CV \wedge CV = Val[r.maj]$.

Proof: Initially, the above lemma holds because $r@0$ holds. The antecedant in the above lemma can be established by statement $r.15$ of task T_r, or by the environment[5], which can establish $en(r)$. Suppose that $en(r) \wedge \neg Pm[r] \wedge r@15$ holds at some state u during a CAS operation m of task T_r. Let t be the state immediately after the execution of $r.9$ by m, and let q be the majority identifier in Buf at state t, i.e., $CV = Val[q] \wedge MAJ(q)$ holds at state t. By Lemma 4.2, no other task modifies Buf or Val during $[t, u]$. Hence, the following holds throughout $[t, u]$.

$$CV = Val[q] \wedge MAJ(q) \tag{4.2}$$

By (4.2), and by examining lines 11 through 15 of the program text, we see that $r.15$ establishes $r.maj = q \wedge r.current = Val[r.maj]$. Along with (4.2), this implies that the execution of $r.15$ establishes the lemma.

If $en(r)$ becomes true at some state u when $\neg Pm[r] \wedge r@\{16..20\}$ holds, then

[5]Refer to Section 4.2.

the consequent holds. This is because, by Lemma 4.2, no other task modifies *Buf* or *Val* during m until state u, and, we have already shown that the execution of $r.15$ establishes the consequent.

We now consider statements that can potentially falsify the above lemma. Statements of any task T_q, where $q \neq r$, cannot falsify the lemma because, by the real-time task model, no statement of any other task T_q is enabled when $en(r)$ holds. Among the statements of T_r, only $r.20$ can falsify the lemma by changing CV. However, the execution of $r.20$ falsifies the antecedant by establishing $r@21$. □

Lemma 4.11: $en(r) \wedge \neg Pm[r] \wedge r@\{21..22\} \wedge r.maj \neq r \wedge r.i \leq N \Rightarrow r.current = CV \wedge CV = Val[r.maj]$.

Proof: Let $A \equiv en(r) \wedge \neg Pm[r] \wedge r@\{21..22\} \wedge r.maj \neq r \wedge r.i \leq N$ and let $B \equiv r.current = CV \wedge CV = Val[r.maj]$. Initially, the above lemma holds because $r@0$ holds. A can be established by statement $r.20$ of T_r, or by the environment, which can establish $en(r)$. Observe that A is established only if $r.20$ is executed when $en(r) \wedge \neg Pm[r] \wedge r@20 \wedge r.maj \neq r$ holds. By Lemma 4.10, this implies that B holds immediately before the execution of $r.20$. B also holds immediately after $r.20$ because $r.20$ does not change $Val[r.maj]$ when $r.maj \neq r$. Hence, $r.20$ establishes the above lemma.

If $en(r)$ becomes true at some state t when $\neg Pm[r] \wedge r@\{21..22\} \wedge r.maj \neq r \wedge r.i \leq N$ holds, then the consequent holds. This is because, by Lemma 4.2, no other task modifies *Buf* or *Val* during m until state t, and we have already shown that $r.20$ establishes the consequent.

We now consider statements that can potentially falsify the above lemma. Statements of any task T_q, where $q \neq r$, cannot falsify the lemma because, by the real-time task model, no statement of some other task T_q is enabled when $en(r)$ holds. Among the statements of T_r, only statement $r.22$ can potentially falsify B by changing CV.

Suppose that $r.22$ is executed at state u when $A \wedge B$ holds. Observe that $r.22$ establishes $MAJ(r)$ — and changes the value of CV — only if $r.maj \neq r \wedge NUM(r) = N-1$ holds at state u. By Lemma 4.4, A implies that $NUM(r) = r.i - 1$ holds at state u. Combining the above facts, it follows that $r.i = N$ holds at state u. Hence, the execution of $r.22$ at state u falsifies A by establishing $r.i > N$. □

Lemma 4.12: *A CAS operation m invoked by task T_r changes the value of CV iff (i) m executes $r.20$ when $\neg Pm[r] \wedge r.maj = r$ holds or (ii) m executes $r.22$ when $\neg Pm[r] \wedge r.maj \neq r \wedge r.i = N$ holds.*

Proof: Task T_r can change CV either by changing the current majority in Buf or by changing $Val[r]$ when r is the majority in Buf. From the program, it follows that only statements $r.19$, $r.20$, and $r.22$ can potentially change CV. However, it follows from Lemma 4.6 that the execution of $r.19$ cannot modify CV. Hence, we only need to consider statements $r.20$ and $r.22$.

Statement $r.20$: Suppose that $r.20$ is executed at state u. Note that the value of CV can be changed by $r.20$ iff $r@20 \wedge MAJ(r)$ holds at state u. By Lemma 4.8, it follows that $\neg Pm[r]$ holds at state u. By Lemma 4.2, this implies that no other task has modified Buf during T_r's operation. Also, by Lemma 4.6, $r.19$ cannot modify the majority in Buf. From these

facts, and from the fact that $MAJ(r)$ holds at state u, it follows that r is the majority task identifier in *Buf* when T_r reads *Buf* at $r.11$ and $r.12$, and that $r.13$ establishes $r.maj = r$. Therefore, $\neg Pm[r] \ \wedge \ r.maj = r$ holds at state u.

Statement $r.22$: Suppose that $r.22$ is executed at state u. Let v be the state immediately following u and let t be the state immediately following the execution of $r.9$. Observe that CV can be changed by $r.22$ iff $r@22 \ \wedge \ MAJ(q) \ \wedge \ q \neq r \ \wedge \ NUM(r) = N - 1$ holds at state u. Because the execution of $r.22$ establishes $MAJ(r)$ at state v, it follows from Lemma 4.8 that $\neg Pm[r]$ holds at state v. By Lemma 4.2, this implies that no other task modifies *Buf* during the interval of states $[t, v]$, which, by Lemma 4.3, implies that $\neg Pm[r]$ holds throughout $[t, v]$. Also, by Lemma 4.6, $r.19$ does not destroy the majority in *Buf*. From these facts, and from the fact that $MAJ(q)$ holds at state u, it follows that q is the majority task identifier in *Buf* when T_r reads *Buf* at $r.11$ and $r.12$, and that $r.13$ establishes $r.maj = q$. Therefore, $r@22 \ \wedge \ \neg Pm[r] \ \wedge \ r.maj \neq r$ holds at state u. Along with Lemma 4.4 and the fact that $NUM(r) = N - 1$ holds at state u, this implies that $r.i = N$ holds at state u as well. □

Lemma 4.13: *Suppose that a CAS operation m invoked by T_r changes CV. Then, m changes CV from $r.old$ to $r.new$, and returns true from line* 27.

Proof: By Lemma 4.12, if T_r invokes a CAS operation m that modifies CV then, either (i) T_r executes $r.20$ when $r@20 \ \wedge \ \neg Pm[r] \ \wedge \ r.maj = r$ holds, or (ii) T_r executes $r.22$ when $r@22 \ \wedge \ \neg Pm[r] \ \wedge \ r.maj \neq r \ \wedge \ r.i = N$ holds. Suppose that (i) or (ii) happens at some state t. Observe that T_r executes $r.20$ or $r.22$ only if T_r succeeds the test at line 15, i.e.,

$r.current = r.old$ holds at state t. Along with Lemma 4.10, this implies that $CV = r.old$ holds at state t. To complete the proof we need to show that m changes CV to $r.new$ and returns true.

Suppose that (i) happens at state t. From Lemma 4.10 and Definition 4.3, the following holds at state t.

$$\neg Pm[r] \ \wedge \ CV = Val[r.maj] \ \wedge \ r.maj = r \ \wedge \ MAJ(r) \tag{4.3}$$

(4.3) implies that the execution of $r.20$ establishes $MAJ(r) \ \wedge \ r@\{21..22\} \ \wedge \ \neg Pm[r] \ \wedge \ Val[r] = r.new$, which implies that $CV = r.new$. Along with Lemma 4.9, this also implies that T_r succeeds the test at line 23 and returns from line 27.

Suppose next that (ii) happens at some state t. From the program text, the execution of $r.20$ establishes $Val[r] = r.new$, which holds until m completes. Because the execution of $r.22$ at state t changes CV by establishing $MAJ(r)$, it follows that the execution of $r.22$ at state t changes CV to $r.new$ and establishes $MAJ(r) \ \wedge \ r@\{21..22\} \ \wedge \ \neg Pm[r]$. From Lemma 4.9 and from the fact that $Val[r] = r.new$ holds at state t until the completion of m, it follows that T_r succeeds the test at line 23 and returns from line 27. □

Lemma 4.14: *If task T_r invokes a CAS operation that returns from line 27, then T_r modifies CV from $r.old$ to $r.new$ during its operation.*

Proof: Suppose that T_r invokes a CAS operation m that returns from line 27. Our proof obligation is to show that either condition (i) or (ii) in Lemma 4.12 is satisfied. (By Lemma 4.13, if condition (i) or (ii) in Lemma 4.12 is satisfied, then T_r changes CV from $r.old$ to $r.new$.) Let t (v) be the state immediately following (preceding) the execution of $r.9$ $(r.23)$.

If T_r returns from line $r.27$, then $r@23 \ \wedge \ (r.i > 2N-1 \ \vee \ Rv[r] = r.new)$ holds at state v and $r.old \neq r.new$ holds during m. We need to consider the following two cases.

Case 1: Suppose that $r.i > 2N-1$ holds at state v. This implies that T_r updates all components of *Buf* in the loop at lines 21 and 22. From the program text, it follows that $\neg Pm[r]$ holds at all states following t, at least until some state at which $r@21 \wedge r.i = 2N-1$ holds. Hence, if $r.maj = r$ holds at state v, then $\neg Pm[r] \ \wedge \ r.maj = r$ holds immediately before the execution of of $r.20$, and condition (i) in Lemma 4.12 holds. On the other hand, if $r.maj \neq r$ holds at state v, then $r@22 \ \wedge \ \neg Pm[r] \ \wedge \ r.maj \neq r \ \wedge \ r.i = N$ holds at some state when $r.22$ is executed, and condition (ii) of Lemma 4.12 holds.

Case 2: Suppose that $r@23 \ \wedge \ r.i \leq 2N-1 \ \wedge \ Rv[r] = r.new$ holds at state v. Let u be the state immediately following the execution of $r.15$. From the program code (line 15), it follows that $\neg Pm[r] \ \wedge \ r@16 \ \wedge \ r.old = r.current$ holds at state u. By Lemma 4.10 and Definition 4.3, the following also holds at state u.

$$CV = r.current \ \wedge \ CV = Val[r.maj] \ \wedge \ r.old = r.current \ \wedge \ MAJ(r.maj) \qquad (4.4)$$

Because $Rv[r] = r.new$ holds at state v, it follows that, after the execution of $r.10$, some higher-priority task assigns $r.new$ to $Rv[r]$. (Observe that T_r cannot update $Rv[r]$ at $r.17$.) Let m' be the first CAS operation to update $Rv[r]$ during the execution of m. Because m' executes during m, it follows from the real-time task model that m' is invoked by a higher-priority task and that m' is completely contained within m. Let T_q be the task that invokes m'. Observe that the execution of $q.17$ updates $Rv[r]$ only if $q.13$ establishes $q.maj = r$. This implies that T_q detects r as the majority task identifier in *Buf* at $q.11$ and

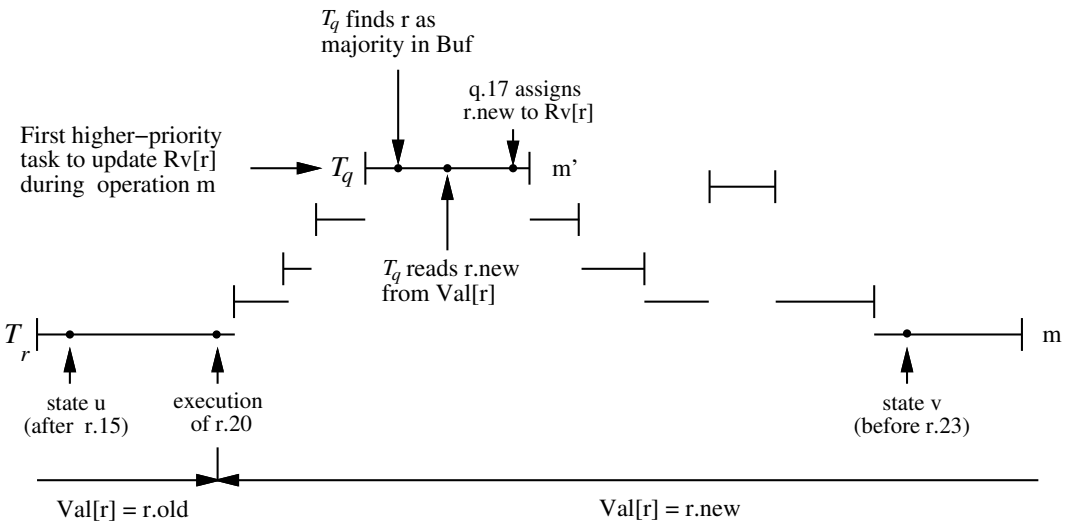


Figure 4.9: Subcase 2.1 in the Proof of Lemma 4.14.

$q.12$. Also, T_q executes $q.17$ only if $\neg Pm[q]$ holds at $q.15$. By Lemma 4.2, this implies that no other task modifies *Buf* or *Val* between the execution of $q.9$ and $q.15$. From the above facts, it follows that $MAJ(r)$ holds before $q.15$. We now consider the following subcases.

Subcase 2.1: Suppose that $r.maj = r$ holds at state u. By (4.4), $Val[r] = r.old$ holds at state u until the execution of $r.20$, which establishes $Val[r] = r.new$. (Refer to Figure 4.9.) We now show that m' is executed after the execution of $r.20$ by m.

Suppose that T_q executes m' before $r.20$ is executed by m. Then, $q.14$ establishes $q.current = r.old$ and $q.17$ establishes $Rv[r] = r.old$. Furthermore, $Pm[r]$ is established by T_q or some higher-priority task before m' completes. (If $\neg Pm[q]$ holds between $q.10$ and $q.24$, then T_q establishes $Pm[r]$ at $q.26$. Otherwise, if $Pm[q]$ is established between $q.10$ and $q.24$, then, by Lemma 4.7, some other higher-priority task returns from line 27, and hence establishes $Pm[r]$.) From the program text (lines 21 and 22), this implies that T_r cannot

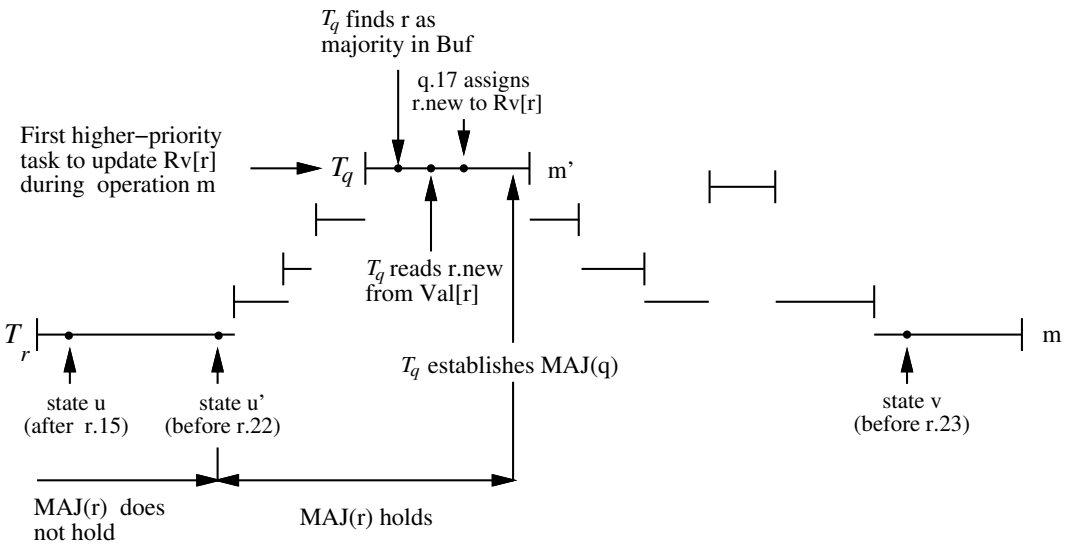


Figure 4.10: Subcase 2.2 in the Proof of Lemma 4.14.

establish $MAJ(r)$ before state v and after m' completes. This implies that any higher-priority task that preempts T_r after $r.20$ will not detect r as a majority in Buf and hence will not update $Rv[r]$. This implies that $Rv[r] = r.old$ holds at state v, a contradiction. Therefore, T_q executes m' *after* the execution of $r.20$ by m.

By (4.4), $MAJ(r)$ holds at state u. As shown previously, $MAJ(r)$ holds before $q.15$. Hence, it follows $MAJ(r)$ is not falsified between state u and $q.15$. (If some other task's operation falsifies $MAJ(r)$ between state u and $q.15$, then that task detects r as a majority in Buf and hence updates $Rv[r]$ before m', which contradicts our assumption about m'.) Therefore, $MAJ(r)$ holds immediately before the execution of $r.20$ by m. By Lemma 4.8, this implies that $\neg Pm[r]$ holds before the execution of $r.20$. Also, because $r.maj = r$ holds at state u, it also holds before the execution of $r.20$. Therefore, $\neg Pm[r] \wedge r.maj = r$ holds when $r.20$ is executed, and condition (i) of Lemma 4.12 holds.

Subcase 2.2: Suppose that $r.maj \neq r$ holds at state u. Along with (4.4) and Definition 4.3, this implies that $MAJ(r)$ does not hold at state u. By examining lines 15 through 22, we see that if some higher-priority task $T_{q'}$ preempts T_r, after state u and before T_r establishes $MAJ(r)$, then (i) $T_{q'}$ does not write to $Rv[r]$, and (ii) $T_{q'}$ assigns either $q'.maj$ or q' to Buf, where $q'.maj \neq r \ \wedge \ q' \neq r$. This implies $MAJ(r)$ can be established during m only by task T_r in the loop at lines 21 and 22 (by Lemma 4.6, the loop at lines 18 and 19 cannot establish $MAJ(r)$). Because T_q detects r as a majority in Buf at $q.11$ and $q.12$, it follows that T_q preempts T_r *after* T_r establishes a majority in Buf at $r.21$ and $r.22$. (Refer to Figure 4.10.)

Let u' be the state at which the execution of $r.22$ establishes $MAJ(r)$, and let u'' be the state immediately following u'. Because $MAJ(r)$ does not hold at state u' but holds at state u'', it follows that $NUM(r) = N$ $(NUM(r) = N - 1)$ holds at state u'' (state u'). Because $r@21 \ \wedge \ MAJ(r)$ holds at state u'', it follows from Lemma 4.8 that $\neg Pm[r]$ holds at state u'' — $\neg Pm[r]$ holds at state u' as well because $r.22$ does not modify $Pm[r]$. Hence, $r@22 \ \wedge \ \neg Pm[r] \ \wedge \ NUM(r) = N - 1 \ \wedge \ r.maj \neq r$ holds at state u'. By Lemma 4.4, this implies that $\neg Pm[r] \ \wedge \ r.maj \neq r \ \wedge \ r.i = N$ holds at state u'. Hence, condition (ii) of Lemma 4.12 holds. □

Lemma 4.15: *If T_r executes $r.15$ when $r.current = r.old \ \wedge \ r.old = r.new \ \wedge \ \neg Pm[r]$ holds, then $CV = r.old$ also holds, and T_r returns from line* 16.

Proof: Suppose that T_r invokes a CAS operation m. Also, suppose that T_r executes $r.15$ when $r.current = r.old \ \wedge \ r.old = r.new \ \wedge \ \neg Pm[r]$ holds. The execution of $r.15$ establishes $r@16 \ \wedge \ \neg Pm[r] \ \wedge \ r.old = r.new \ \wedge \ r.current = r.old$. By Lemma 4.10,

$CV = r.current \ \wedge \ CV = Val[r.maj]$ holds after $r.15$. Hence, $CV = r.old$ holds after $r.15$ — $CV = r.old$ also holds before $r.15$ because $r.15$ does not change CV. Clearly, T_r returns from $r.16$. □

Lemma 4.16: *If T_r invokes a CAS operation m that returns from $r.28$, then m does not change CV and $CV \neq r.old$ holds at some state during m.*

Proof: Suppose that T_r invokes a CAS operation m that returns from $r.28$. By Lemma 4.13, if m modifies CV, then it returns from $r.27$. Hence, m does not change CV. We now show that $CV \neq r.old$ holds at some state during m. Task T_r can fail m by failing the test either at $r.15$ or at $r.23$. Consider the following three cases.

Case 1: Suppose that $r.15$ is executed when $\neg Pm[r] \ \wedge \ r.current \neq r.old$ holds. The execution of $r.15$ establishes $r@16 \ \wedge \ \neg Pm[r] \ \wedge \ r.current \neq r.old$. Along with Lemma 4.10, this implies that $CV \neq r.old$ holds in the state following the execution of $r.15$.

Case 2: Suppose that $r.15$ is executed when $Pm[r]$ holds. By Lemma 4.7, some higher-priority task T_q returns from $q.27$ during T_r's operation. By Lemma 4.14, T_q modifies CV during its operation from $q.old$ to $q.new$. In this case, $CV \neq r.old$ holds in the state preceding or the state following T_q's linearization step.

Case 3: Suppose that $r.23$ is executed when $r.i \leq 2N-1 \ \wedge \ Rv[r] \neq r.new$ holds. Because T_r does not write into all $2N-1$ locations of *Buf* at lines 21 and 22, it follows that $Pm[r]$ is established during or before the loop at line 21. By Lemma 4.7, some higher-priority task T_q returns from $q.27$ during T_r's operation. As in Case 2, $CV \neq r.old$ holds in the state

preceding or in the state following T_q's linearization step. □

Lemma 4.17: *If T_r invokes a Read operation, then, at some state during T_r's operation, CV equals the value returned by the operation.*

Proof: Suppose that T_r invokes a Read operation m. If $\neg Pm[r]$ holds at $r.6$, then by Lemma 4.2, no other task modifies *Buf* or *Val* during T_r's operation, and T_r assigns to $r.maj$ the majority task identifier in *Buf* at $r.4$, and assigns CV to $r.current$ at $r.5$. On the other hand, if $Pm[r]$ holds at $r.6$, then m returns V, and it follows from Lemma 4.7 that one or more higher-priority tasks return from line 27 during m — these tasks also update V. Of these tasks, let T_q be the last task to update V before the execution of $r.6$ by m. From the real-time task model, it follows that T_q's operation is completely contained within T_r's operation. Also, by Lemma 4.14, T_q changes CV to $q.new$ and establishes $q.new = V$ at $q.24$. This implies that T_r returns the value of CV at the state immediately following T_q's linearization step. □

Theorem 4.2: *Read and CAS can be implemented on a real-time uniprocessor system with $O(N)$ time and $O(N^2)$ space complexity.*

Proof: Properties 4.1 through 4.4 follow directly from the above lemmas. Specifically, Property 4.1 follows from Lemmas 4.12, 4.13, and 4.14; Property 4.2 from Lemma 4.15; Property 4.3 from Lemma 4.16; and Property 4.4 follows from Lemma 4.17. This proves that the program in Figure 4.6 correctly implements a shared object that supports CAS and Read. Clearly, our implementation has $O(N)$ time complexity. The implementation has $O(N^2)$ space complexity because each task requires $O(N)$ private variables. □

Corollary 4.1: *On a uniprocessor real-time system, any object can be implemented in a lock-free manner with time complexity $O(N)$ using only load and store instructions.*

Proof: The proof follows directly from the following observations: (*i*) any shared object can be implemented in a lock-free manner by using a constant number of LL and SC instructions [37]; (*ii*) an object that supports LL and SC operations can be implemented using only CAS, load, and store instructions with worst-case time complexity $O(1)$ [7]; and, (*iii*) by Theorem 4.2, in any uniprocessor hard real-time system consisting of N tasks, an object that supports CAS and Read operations can be implemented using only load and store instructions with $O(N)$ worst-case time complexity. □

4.5 Implementing CAS using Move, Load, and Store Instructions

We now present an implementation of a shared object that supports Read and CAS operations. This implementation uses the move instruction in addition to loads and stores. The move instruction is widely available on most systems, e.g., uniprocessor systems based on Intel's 80x86 and Pentium line of processors support the move instruction.

In our implementation, a task modifies (reads) the value of the implemented object by invoking the CAS (Read) procedure shown in Figure 4.11.[6] Shared variable $Rv[r]$ is used by T_r to detect whether a CAS operation by it was successful. The shared variable *Run* is used by a task T_r to determine whether a CAS operation by it has been interfered with.

[6]For simplicity, the shared object is not explicitly passed as parameter to the CAS and Read operations.

```
type objtype = record val : valtype ; tid : 0..N − 1 end        /* These fields are packed into one word */
shared var Rv : array [0..N − 1] of valtype ∪ ⊥;     /* Indicates success/failure of a CAS operation */
           Final : objtype;                                    /* Stores the current value of the object */
           Propose : objtype;                                  /* Used to propose a new value for the object */
           Run : 0..N − 1    /* Used to determine whether a CAS operation has been interfered with */

private var oldf : objtype                                     /* For task T_p, where 0 ≤ p ≤ N − 1 */
procedure CAS(old, new : valtype) returns boolean

1 :  if old = new then return Final.val = old fi;
2 :  if Final.val ≠ old then return false fi;
3 :  Run := p;
4 :  Rv[p] := ⊥;
5 :  oldf := Final;
6 :  Rv[oldf.tid] := oldf.val;
7 :  if Final.val = old then
8 :      Propose := (new, p);
9 :      if Run = p then
10 :         Final := Propose;                                 /* Move the proposed value to Final */
11 :         if Run = p then return true fi;
12 :         if Rv[p] = new then return true fi
         fi;
13 :     Propose := Final              /* Move Final to Propose to handle enabled late-move problem */
     fi;
14 : return false

procedure Read() returns valtype

15 : return Final.val
```

Figure 4.11: Implementation of CAS/Read using move.

The words that may be accessed by the CAS and Read procedures are assumed to be of type *objtype*. A word of this type consists of two fields: a *val* field, which contains the value of the implemented object, and a *tid* field that stores the identifier of the task that wrote that value. We use the variables *Final* and *Propose* in a manner similar to the consensus object implementation in Section 4.3. Specifically, a task that invokes a CAS operation first proposes a value in *Propose* and finalizes the value by copying *Propose* to *Final*. However, our CAS object implementation is more complicated than the consensus object implementation because a CAS object can be assigned values many times. In contrast, a consensus object can be assigned a value only once.

The Read operation is simply implemented by a read of *Final.val*. If $old = new$ (line 1), then a CAS(old, new) operation by task T_p succeeds or fails immediately, depending on the value of *Final.val*. Also, if T_p detects that *Final.val* $\neq$ *old*, its CAS can fail immediately (lines 2 and 7). Otherwise, T_p writes its new value into *Propose* (line 8) and, if it does not detect an overlapping successful CAS operation (line 9), it attempts to move its new value from *Propose* into *Final* (line 10), thereby succeeding. It is possible for a task T_p to successfully moves its new value to *Final* and then get preempted before executing $p.11$. In this case, T_p cannot detect that it succeeded. To solve this problem, before a task q modifies *Final*, it first "informs" the task that previously modified *Final* that it succeeded (lines 5 and 6). The preempted task T_p can therefore detect by reading $Rv[p]$ (line 12) that it succeeded. To ensure that a low-priority task does not modify *Final* "late", thereby "corrupting" the value written by a previous CAS operation, each CAS operation ensures that, if it modifies *Propose* or *Final*, then *Propose* = *Final* holds before it relinquishes the

processor (line 13). This ensures that the "late" move operation has no effect on *Final*.

4.5.1 Correctness Proof

We first define some terms and state the properties required of our CAS implementation. Then, we prove several lemmas used to prove the stated properties.

Definition 4.4: The *current value* of the shared object is given by *Final.val*.

Property 4.5: The value returned by a Read operation invoked by a task T_r equals the value of the shared object immediately before $r.15$ is executed. □

Property 4.6: Suppose that T_r invokes a CAS operation that returns true from $r.1$. Then, $Final.val = r.old \ \wedge \ r.old = r.new$ holds immediately before $r.1$ is executed, and $Final.val = r.new$ holds immediately after. □

Property 4.7: Suppose that T_r invokes a CAS operation that returns true from $r.11$ or $r.12$. Then, $Final.val = r.old \ \wedge \ Propose = (r.new, r)$ holds immediately before $r.10$ is executed, and $Final.val = r.new$ holds immediately after. □

Property 4.8: Suppose that T_r invokes a CAS operation that returns false. Then, $Final.val \neq r.old$ holds at some state during T_r's operation. □

Property 4.5 states that a read operation linearizes to $r.15$. Property 4.6 states that a successful CAS operation linearizes to line 1 if it does not modify the value of the shared object. Property 4.7 states that a successful CAS operation linearizes to $r.10$ if it changes the value of the object. Property 4.8 states that if a CAS operation returns

false, then the old value does not match the value of the object at some point during the operation. It is easy to see that Properties 4.5 and 4.6 hold. We now prove the following lemmas in order to establish the remaining properties.

Lemma 4.18: *If* $r@\{4..13\} \wedge Run = r$ *holds when a statement of* T_r *is enabled to execute, then no other task has modified Final or Propose since* T_r *executed* $r.3$.

Proof: Suppose that T_r invokes a CAS operation m. Let t be the state immediately after the execution of $r.3$ by m. Let u be some subsequent state during m when $r@\{4..13\} \wedge Run = r$ holds and T_r is enabled to execute. The proof is by contradiction. Suppose that some other task T_q invokes a CAS operation m' that modifies *Final* or *Propose* in the interval of states $[t, u]$. Then, T_q has higher priority than T_r, and from the real-time task model, m' is completely contained within m. Before T_q modifies *Final* or *Propose*, it first establishes $Run \neq r$ by executing $q.3$. Now, T_r cannot establish $Run = r$ after m' because $r.3$ is executed *before* state t. Also, higher-priority tasks cannot establish $Run = r$. Therefore, $Run \neq r$ holds at state u, a contradiction. □

Lemma 4.19: *If* $r@\{4..13\} \wedge Run \neq r$ *holds when* T_r *is enabled to execute, then some task changes Final.val and writes to Propose after* T_r *executes* $r.3$.

Proof: Suppose that T_r invokes a CAS operation m. Let t be the state immediately after the execution of $r.3$. Let u be some subsequent state during m when $r@\{4..13\} \wedge Run \neq r$ holds and T_r is enabled to execute. Observe that $Run \neq r$ holds at state t only if some other task modifies *Run* in the interval of states $[t, u]$. Consider the highest-priority task T_q that invokes a CAS operation m' that updates *Run* after T_r's preemption and before its

subsequent resumption. Observe that T_q updates *Run* only if $q.old \neq q.new$ holds during m'. Also, observe that T_q updates *Run* only if $q@2 \ \wedge \ Final.val = q.old$ holds before $q.2$. Since no other task modifies *Run* during m', it follows that no other task modifies *Final* or *Propose* during m'. (If a task updates *Final* or *Propose*, then it also modifies *Run*.) Therefore, T_q succeeds the test at $q.7$, assigns $(q.new, q)$ to *Propose* at $q.8$, succeeds the test at $q.9$, and changes *Final.val* from $q.old$ to $q.new$ at $q.10$. □

Lemma 4.20: *If* $Run \neq r$ *holds immediately before* $r.10$ *is executed during a CAS operation by* T_r*, then Final is not modified by the execution of* $r.10$.

Proof: Suppose that T_r invokes a CAS operation m and that, during m, $r.10$ is executed at state u when $r@10 \ \wedge \ Run \neq r$ holds. By Lemma 4.19, during T_r's operation, some higher-priority task invokes a CAS operation that modifies *Final.val* and writes to *Propose*. Consider the last CAS operation m' to execute line 3 before state u. Let T_q be the task that invokes m'. (Refer to Figure 4.12.) Observe that T_q executes $q.3$ only if $Final.val = q.old$ holds when T_q executes $q.2$. Because no other task modifies *Propose* or *Final* during m' (otherwise, T_q is not the last task to execute line 3 before state u), it follows from Lemma 4.19 that $Run = q$ holds throughout T_q's execution. These facts imply that T_q succeeds the tests at $q.7$, $q.9$, and $q.11$, and establishes $Run = q \ \wedge \ Final = Propose$ before it returns from m'.

Let A denote the expression $Run = q \ \wedge \ Final = Propose$. Let t be the state immediately after m' completes. We have that A holds at state t. We now prove that A holds at state u. The proof is by induction over the priority levels, i.e., we show that if A holds when a task resumes execution during $[t, u]$, then it also holds when that task

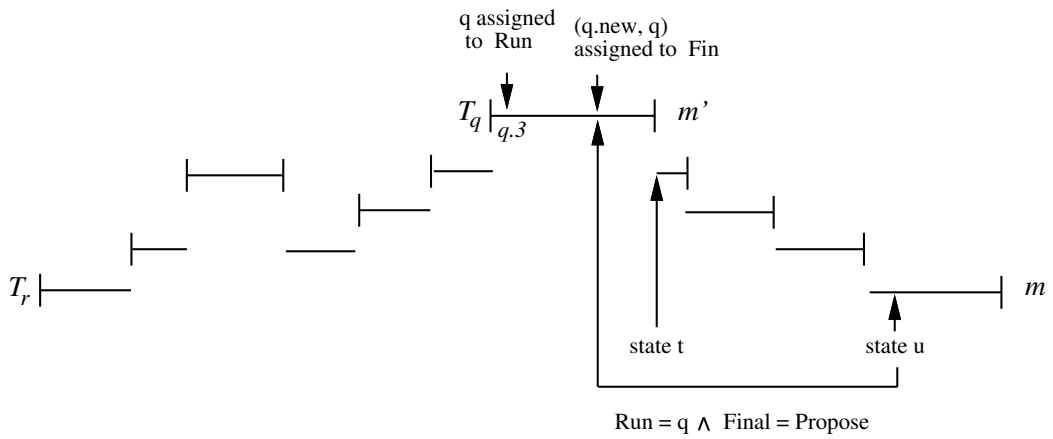


Figure 4.12: Proof of Lemma 4.20.

completes. Consider a task T_p that executes during $[t, u]$. By our induction assumption, A holds when T_p resumes execution. Also, because T_q is the last task to execute $q.3$ before state u, it follows that $p@\{4..14\}$ holds when T_p resumes, and that T_p is not preempted during $[t, u]$ by any other task that modifies *Final* or *Propose* (if some other task preempts one of these tasks and modifies *Final* or *Propose*, then T_q is not the last task to execute line 3). Task T_p can potentially falsify A only by changing *Final* or *Propose*. However, if T_p executes $p.8$ after it resumes, then it fails the test at $p.9$, and T_p executes $p.13$, which ensures that A holds when it returns from $p.14$. On the other hand, if T_p executes $p.10$ when it resumes, then it does not modify *Final*. This ensures that A holds before T_p returns from $p.11$, $p.12$ or $p.14$. Thus, A holds when T_p returns from its CAS operation. This concludes the proof of our induction step. Therefore, A holds at state u, which implies that $r.10$ does not modify *Final*. □

Lemma 4.21: $r@11 \wedge Final = (r.new, r)$ *unless* $r@0 \vee (r@\{11, 12\} \wedge Rv[r] = r.new)$.

Proof: Let $A \equiv r@11 \ \wedge \ Final = (r.new, r)$ and let $B \equiv r@0 \ \vee \ (r@\{11, 12\} \ \wedge \ Rv[r] = r.new)$. (Recall that $r@0$ denotes the execution of T_r within the remainder section; see Section 4.2.) Our proof obligation is to show that, for any enabled statement s, the following Hoare triple holds.

$$\{A \ \wedge \ \neg B\} \ s \ \{A \ \vee \ B\} \tag{4.5}$$

Only statement $r.11$ of T_r and statement $q.10$ of some higher-priority task T_q can potentially falsify (4.5). We now consider these two statements.

Statement $r.11$: Suppose that $r.11$ is executed when $r@11 \ \wedge \ Final = (r.new, r)$ holds. From the program text (lines 1 and 7), we see that $r.11$ is executed only if $r.old \neq r.new \ \wedge \ Final.val = r.old$ holds at $r.7$. From these facts, it follows that *Final* was modified by the execution of $r.10$. (Other tasks cannot assign r to *Final.tid*.) By Lemma 4.20, this implies that $Run = r$ holds immediately before the execution of $r.10$, and hence after. We need to consider two cases.

Case 1: If $Run = r$ holds immediately before T_r executes $r.11$, then the execution of $r.11$ establishes $r@0$.

Case 2: If $Run \neq r$ holds before the execution of $r.11$, then one or more higher-priority tasks modify *Run* after the execution of $r.10$. Of these tasks, consider T_q, the first task to execute line 10 after the execution of $r.10$. Because the execution of $q.10$ modifies *Final*, it follows from Lemma 4.20, that $Run = q$ holds before the execution of $q.10$. Also, because no other task executes line 10 between $r.10$ and $q.10$, it follows that T_q assigns $(r.new, r)$ to *oldf* at $q.5$. Hence, the execution of $q.6$ establishes $Rv[r] = r.new$ which holds until the

execution of $q.10$, which establishes $Final.tid \neq r$. Higher-priority tasks that execute after the execution of $q.10$ will not detect r in $Final.tid$ at line 5, and hence will not modify $Rv[r]$ at line 6. Therefore, $Rv[r] = r.new$ holds before the execution of $r.11$ and hence after. It follows that (4.5) holds if s is $r.11$.

Statement $q.10$: Suppose that the execution of $q.10$ falsifies $Final = (r.new, r)$. Because $r@11$ holds when $q.10$ is executed, T_q is a higher priority that preempts T_r. Also, because $Final = (r.new, r)$ holds immediately before $q.10$ is executed, it follows that $q.5$ establishes $q.oldf = (r.new, r)$ at $q.5$, and that $q.6$ establishes $Rv[r] = r.new$. Therefore, $Rv[r] = r.new$ holds immediately before $q.10$ is executed and hence after. It follows that (4.5) holds if s is $q.10$. □

Lemma 4.22: *If T_r invokes a CAS operation m that changes Final.val, then $Final.val = r.old \wedge Propose = (r.new, r)$ holds before the execution of $r.10$ by m, and m returns true from $r.11$ or $r.12$.*

Proof: Task T_r can change $Final.val$ only by executing $r.10$. Because T_r's operation modifies $Final$, it follows from Lemma 4.20 that T_r executes $r.10$ when $Run = r$ holds. By Lemma 4.18, this implies that no other task modifies $Final$ or $Propose$ between $r.3$ and $r.10$. Because the execution of $r.7$ establishes $Final.val = r.old$ and the execution of $r.8$ establishes $Propose = (r.new, r)$, it follows from the above facts that $Final.val = r.old \wedge Propose = (r.new, r)$ holds before $r.10$ is executed. Hence, the execution of $r.10$ establishes $r@11 \wedge Final = (r.new, r)$. From Lemma 4.21, it follows that T_r returns from $r.11$ or $r.12$. □

Lemma 4.23: *If T_r invokes a CAS operation m that returns true from $r.11$ or $r.12$, then m executes $r.10$ when $Final.val = r.old \ \wedge \ Propose = (r.new, r)$ holds.*

Proof: Suppose that T_r invokes a CAS operation m that returns true from $r.11$ or $r.12$. Observe that T_r executes $r.11$ or $r.12$ only if (i) $r.old \neq r.new$ holds during m and (ii) $Run = r$ holds immediately before the execution of $r.9$. By Lemma 4.18, (ii) implies that no other task modifies *Final* or *Propose* between $r.3$ and $r.9$. From the program text (line 7) and from the above facts, the following holds between the execution of $r.3$ and $r.9$.

$$r@\{4..9\} \ \wedge \ Run = r \ \wedge \ Final.val = r.old \tag{4.6}$$

We now consider the statements from which m returns.

Statement $r.11$: Suppose that T_r executes $r.11$ and returns from m. Observe that T_r returns from $r.11$ only if $r@11 \ \wedge \ Run = r$ holds before the execution of $r.11$, which implies that no other task modifies *Run* between the executions of $r.3$ and $r.11$. By Lemma 4.18, this implies that no other task modifies *Final* or *Propose* after $r.3$ is executed and before $r.11$ is executed. This fact, along with (4.6) and the fact that $r.8$ establishes $Propose = (r.new, r)$, implies that $Final.val = r.old \ \wedge \ Propose = (r.new, r)$ holds immediately before $r.10$.

Statement $r.12$: Suppose that T_r executes $r.12$ and returns from m. Observe that T_r returns from $r.12$ only if $Run \neq r \ \wedge \ Rv[r] = r.new$ holds before the execution of $r.12$. Therefore, T_r or some higher-priority task reads $(r.new, r)$ from *Final* at line 5 and assigns $r.new$ to $Rv[r]$ at line 6. However, T_r cannot assign $r.new$ to $Rv[r]$ because it follows from (4.6) that $r.5$ reads $r.old$ from $Final.val$ and that $r.6$ can only assign $r.old$ to $Rv[r]$. Therefore, some higher-priority task preempts T_r and assigns $r.new$ into $Rv[r]$. This implies that

$Final = (r.new, r)$ is established during m. However, higher-priority tasks cannot write r into *Final.tid*, and, by the real-time task model, lower-priority tasks do not take a step during m. Therefore, T_r assigns $(r.new, r)$ to *Final* when it executes $r.10$. This implies that $Propose = (r.new, r)$ holds immediately before $r.10$ is executed and that no other task updates *Propose* between $r.8$ and $r.10$. Because a task can modify *Final* only updating *Propose*, this implies that no other task modifies *Final* between $r.8$ and $r.10$. Along with (4.6), this implies that $Final.val = r.old$ holds before the execution of $r.10$. Therefore, $Final.val = r.old \ \wedge \ Propose = (r.new, r)$ holds immediately before $r.10$ is executed. □

Lemma 4.24: *If T_r invokes a CAS operation that returns false, then $Final.val \neq r.old$ holds at some state during that operation.*

Proof: It is easy to see that $Final.val \neq r.old$ holds if T_r returns from $r.1$, $r.2$, or from $r.14$ after failing the test at $r.7$. If T_r returns from $r.14$ after failing the test at $r.9$ or the tests at $r.11$ and $r.12$, then $Run \neq r$ holds during T_r's operation. By Lemma 4.19 and from the real-time task model, it follows that a CAS operation invoked by some higher-priority task T_q executes within T_r's operation and changes *Final.val*. It follows that $Final.val \neq r.old$ holds in the state preceding or following the linearization step of T_q's CAS operation. □

Theorem 4.3: *Read and CAS can be implemented on a real-time uniprocessor system with constant time and $O(N)$ space complexity using move, load, and store instructions.*

Proof: Property 4.5 holds trivially because *Final.val* contains the implemented value. It is easy to see that Property 4.6 holds. Property 4.7 follows from Lemmas 4.22 and 4.23; Property 4.8 from Lemma 4.24. It is also easy to see that the time and space complexity

of the CAS implementation are $O(1)$ and $O(N)$, respectively. □

4.6 Implementing Multi-Word Primitives

In this section, we implement MWCAS, which extends the semantics of CAS to multiple words. MWCAS is a useful primitive for two reasons. First, it simplifies the implementation of many lock-free objects, e.g., queues are easy to implement with MWCAS, but hard to implement with single-word primitives. Second, it can be used to implement multi-object operations and transactions. For example, an operation that atomically dequeues an item off one queue and enqueues that item onto another could be implemented by combining the body of the retry loops in the *enqueue* and *dequeue* procedures in Figure 1.2, and by using MWCAS at the bottom of the loop to make a pointer changes. In Chapter 5, we further illustrate the utility of MWCAS by using it to implement lock-free transactions.

4.6.1 A Wait-Free Implementation of MWCAS

Figure 4.13 depicts our implementation of MWCAS and an associated Read primitive. The implementation requires a CAS instruction. Requiring CAS is not a severe limitation because hardware on the Motorola 680x0 line of processors and on the Intel Pentium support the CAS instruction. A shared object that supports Read and CAS operations can also be easily implemented using LL and SC— the LL/SC instructions are currently supported by most processors, including Motorola's PowerPC and DEC's Alpha. Furthermore, in a system consisting of N tasks, CAS can be implemented in $O(N)$ time using load and store instructions, as described in Section 4.4, or in constant time using the move

instruction, as described in Section 4.5.

In our implementation, a task performs a MWCAS operation on a collection of words by invoking the MWCAS procedure. This procedure takes as input an integer parameter indicating the number of words to be accessed, an array containing the addresses of the words to be accessed, and arrays containing old and new values, respectively, for these words. We assume that each MWCAS operation accesses at most B words. A task performs a Read operation by invoking the Read procedure, which takes as input the address of the word to be read. The words that may be accessed by the MWCAS and Read procedures are assumed to be of type *wordtype*. A word of this type consists of four fields: a *val* field, which contains an application-dependent value, and *count* ($\lceil \log B \rceil$ bits), *valid* (one bit), and *tid* ($\lceil \log N \rceil$ bits) fields, which are used in the implementation. In most applications, the *val* field contains an object pointer, and perhaps a small amount of control information. For example, in the implementation of lock-free transactions presented in Chapter 5, the *val* field consists of a pointer to a region of shared memory and a "version counter". Assuming a 32-bit word, these fields can be defined to allow transactions by over one thousand tasks, on several thousand objects.

We present below a detailed description of the MWCAS and Read procedures. We begin with an overview of the MWCAS procedure. We follow this by an example that illustrates the key ideas. After this, we present a brief overview of the Read procedure. We then conclude by considering several subtleties of the implementation that are not addressed in our initial overview.

A MWCAS operation by task T_r is executed in three phases. In the first phase

```
type                                          /* Assume N tasks, each MWCAS accesses at most B words */
    wordtype = record val: valtype; count: 0..B-1; valid: boolean; tid: 0..N-1 end;   /* All of these fields
            are stored in one word; the val field is application dependent; the valid field should be initially true */
    addrlisttype = array[0..B-1] of pointer to wordtype;          /* Addresses to perform MWCAS on */
    vallisttype = array[0..B-1] of valtype                        /* List of old and new values for MWCAS */

shared var
    Status: array[0..N-1] of integer initially 0;
                                  /* Status of task's latest MWCAS: 0 if pending, 1 if invalid, 2 if valid */
    Save: array[0..N-1, 0..B-1] of valtype
                                  /* Used to temporarily save value from a word during a MWCAS on that word */

private var                                                       /* For task p, where 0 ≤ p < N */
    init, assn: array[0..B-1] of wordtype;          /* Values initially read and assigned to words by MWCAS */
    i,j: 0..B+1; retval: boolean; word: wordtype; val: valtype

procedure MWCAS(numwds: 0..B; addr: addrlisttype;
            old, new: vallisttype) returns boolean
1:  Status[p] := 0;
2:  i := 0;
3:  while i < numwds ∧ Status[p] = 0 do
4:    init[i] := *addr[i];
5:    if init[i].valid ∨ Status[init[i].tid] = 2 then
6:      val := init[i].val
      else
7:      val := Save[init[i].tid, init[i].count]
      fi;
8:    Save[p,i] := val;
9:    if old[i] ≠ val then
10:     Status[p] := 1
      else
11:     assn[i] := (new[i], i, false, p);
12:     if ¬CAS(addr[i], init[i], assn[i]) then
13:       Status[p] := 1
        fi;
14:     i := i + 1
      fi
    od;
/* MWCAS continued */
15: retval := CAS(&Status[p], 0, 2);
16: for j := 0 to i-1 do
17:   if old[j] ≠ new[j] ∧ retval then
18:     CAS(addr[j], assn[j], (new[j], 0, true, p));
19:     if ¬init[j].valid then CAS(&Status[init[j].tid], 0, 1)
        fi
20:   else if ¬CAS(addr[j], assn[j], init[j]) then
21:     if ¬init[j].valid then CAS(&Status[init[j].tid], 0, 1)
        fi
      fi
    od;
22: return(retval)

procedure Read(addr: pointer to wordtype)
                                                  returns valtype
23: word := *addr;
24: if word.valid ∨ Status[word.tid] = 2 then
25:   return(word.val)
    else
26:   return(Save[word.tid, word.count])
    fi
```

Figure 4.13: Wait-free implementation of MWCAS.

(lines 1 through 14), the k^{th} word that is accessed by T_r — call it w — is updated so that its *val* field contains the desired new value, the *count* field contains the value k, the *valid* field is false, and the *tid* field contains the value r (see lines 11 and 12). In addition, the old value of w is saved in the shared variable $Save[r, k]$ (line 8). The *tid* and *count* fields of w are used by other tasks to retrieve the old value from the *Save* array. The *tid* field is also used as an index into the *Status* array, the role of which is described below.

To understand the "effect" the first phase has on the words that are accessed, it is necessary to understand how each word's "current value" is defined. This notion is formalized in the following definition.

Definition 4.5: Let w denote a shared variable of type *wordtype* that is accessible by a MWCAS or Read operation. At any state, the *current value* of word w is given by the following expression.

$$Val(w) = \begin{cases} w.val & \text{if } w.valid \ \vee \ Status[w.tid] = 2 \\ Save[w.tid, \ w.count] & \text{otherwise} \end{cases}$$

□

The shared variable $Status[r]$ — which affects $Val(w)$ when $w.tid = r$ — gives the "status" of task T_r's latest MWCAS operation. $Status[r]$ is initialized to 0 when such an operation begins (line 1). If the operation is interfered with by other MWCAS operations, or if the current value of some word accessed by the operation differs from the old value specified for that word, then $Status[r]$ is assigned the value 1 (lines 10, 13, 19, and 21). A value of 2 in $Status[r]$ indicates that task T_r's latest MWCAS operation has succeeded.

With Definition 4.5 in mind, the "effect" of the first phase of a MWCAS operation

can now be understood. This phase does not change the current value of any word that is accessed. However, if this phase is "successful" — i.e., the operation does not get interfered with, nor does it find that the current value of some word differs from the old value specified for that word — then at the end of the first phase, the proposed new value for each word is contained within the *val* field of that word.

The second phase of a MWCAS operation consists of only one statement: the CAS at line 15. This CAS attempts to both validate and commit the operation by changing the value of $Status[r]$ from 0 to 2. $Status[r] = 0$ will hold when the CAS at line 15 is performed iff the first phase was "successful". By Definition 4.5, this CAS, if successful, atomically changes the current value of each word accessed to the desired new value.

A complication arises from the way in which $Status[r]$ is used. If a new MWCAS operation is attempted by task T_r, then changing $Status[r]$ to 0 at line 1 might have the undesired effect of changing the current value of words previously accessed by T_r. The third phase of the MWCAS (lines 16 through 22) ensures that reinitializing $Status[r]$ during the next MWCAS does not cause this problem. In this phase, each word w that is accessed by the MWCAS operation of T_r is "cleaned up" so that, upon completion of that operation, $w.tid \neq r \ \vee \ w.valid$ holds; this implies that the current value of word w does not depend on $Status[r]$.

Lines 19 and 21 are executed to invalidate any pending lower-priority MWCAS operation that has been interfered with. Note that such a pending operation exists for word w if task T_r detects that $w.valid$ is false. Line 19 is executed by task T_r only if its own operation has succeeded in changing the value of some word. Line 21 is executed by task T_r

only if its attempt to "clean up" a word fails. This failure signifies that that word has been modified by a higher-priority task during T_r's execution, so it is appropriate to invalidate any pending lower-priority operation that accesses that word.

Example. Figure 4.14 depicts the effects of a MWCAS operation m by task T_4 on three words x, y, and z, with old/new values 12/5, 22/10, and 8/17, respectively. Inset (a) shows the contents of various variables just before m begins. Note that the current value of each word matches the desired old value. Inset (b) shows relevant variables after the first phase of m has completed, assuming no interferences by higher-priority tasks. The current value of each word is unchanged. Note that changing the value of *Status*[4] from 0 to 2 in inset (b) would have the effect of atomically changing the current value of each of x, y, and z to the desired new value. Inset (c) shows relevant variables at the termination of m, assuming no interferences by higher-priority tasks. The current value of each word is now the desired new value, and all *valid* fields are *true* (so the value of *Status*[4] is no longer relevant). Before returning, task T_4 updates *Status*[3] (line 19 of Figure 4.13) to indicate that task T_3 (which must be of lower priority) has been interfered with. Inset (d) shows relevant variables at the termination of m, assuming an interference on word z by task T_9 (which must be of higher-priority) with new value 56. *Status*[4] is now 1, indicating the failure of task T_4's operation. *Status*[3] is left unchanged in this case. Observe that task T_4 has successfully restored the original values of words x and y. Insets (e) and (f) show the operation interleavings corresponding to insets (c) and (d), respectively. □

Having dispensed with the MWCAS procedure, the Read procedure can be readily explained. If the Read procedure is invoked with the address of word w as input, then it

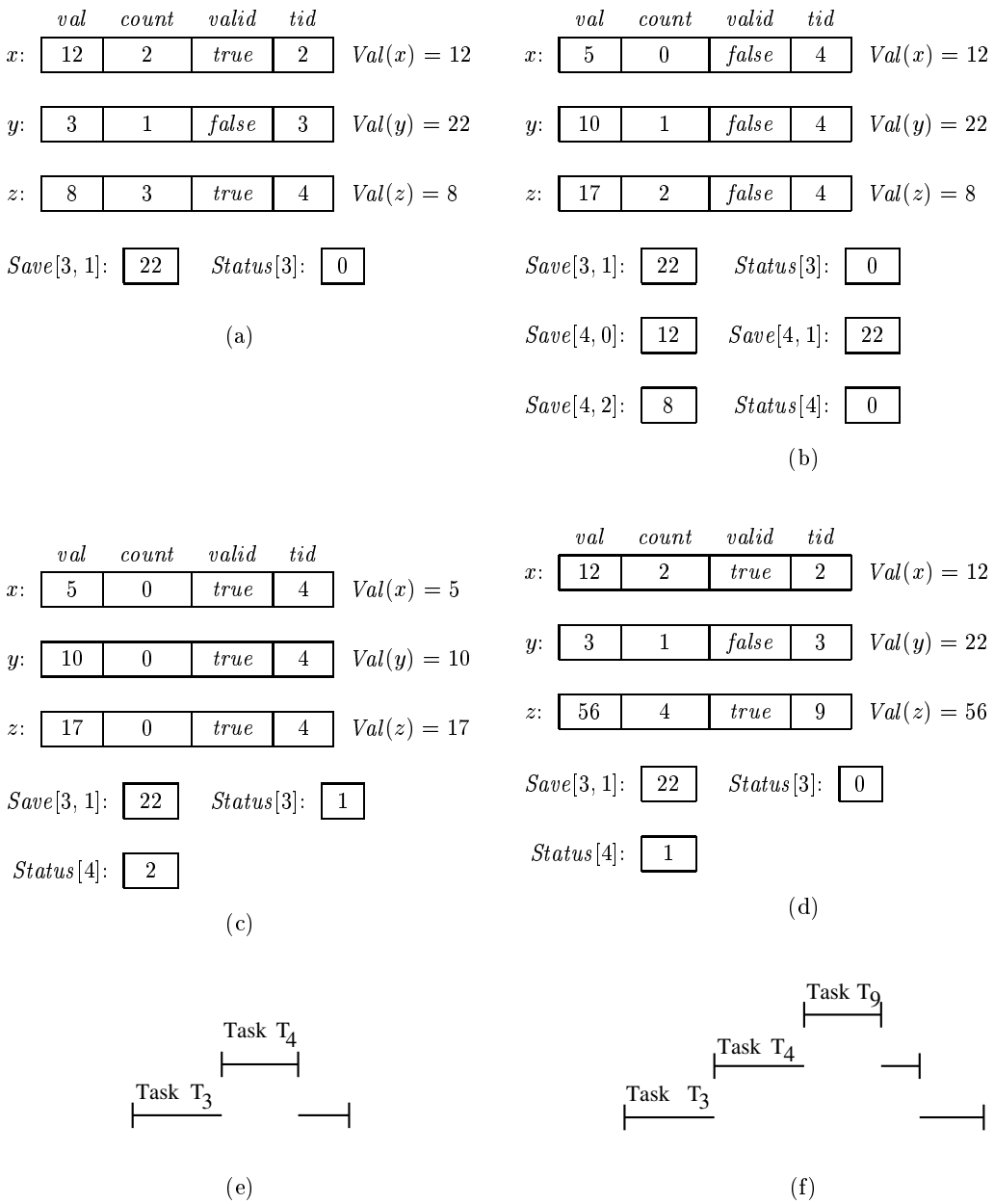


Figure 4.14: Task T_4 performs a MWCAS operation on words x, y, and z, with old/new values 12/5, 22/10, and 8/17, respectively. The contents of relevant shared variables are shown (a) at the beginning of the operation; (b) after the loop in lines 3..17; (c) at the end of the operation, assuming success; and (d) at the end of the operation, assuming failure on word z. The operation interleaving that results in (c) is shown in (e) (T_4 preempts T_3). The operation interleaving that results in (d) is shown in (f) (T_4 preempts T_3, and T_9 preempts T_4).

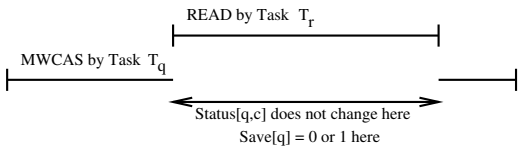


Figure 4.15: Example of a Read operation by Task T_r.

simply computes the current value of w as given in Definition 4.5. Note that the current value of each word accessed by the MWCAS procedure is computed within that procedure in the same way as done in the Read procedure (see lines 4 through 9).

Although the above description conveys the basic idea of the implementation, there are some subtleties that we have not yet addressed. We now attempt to explain some of these subtleties.

One such subtlety concerns the Read procedure. If this procedure is invoked to read word w, and if line 23 is executed when $w.tid = q \ \wedge \ w.count = c$ holds, then the value of $Val(w)$ could potentially be determined incorrectly if the values of $Status[q]$ or $Save[q, c]$ were to change during the execution of the Read procedure. However, $Status[q]$ and $Save[q, c]$ affect the value of $Val(w)$ only if $w.valid$ is false when line 23 is executed. As explained above, any MWCAS operation of task T_q that accesses word w "cleans up" in its third phase, thereby ensuring that $w.tid \neq q \ \vee \ w.valid$ holds upon termination of that operation. Thus, if $w.tid = q \ \wedge \ \neg w.valid$ holds when line 23 of the Read procedure is executed by task T_r, then it must be the case that T_r has preempted T_q as illustrated in Figure 4.15. Because T_q has been preempted, the value of $Save[q, c]$ cannot change during the execution of the Read procedure. Also, it must be the case that $Status[q] \neq 2$ holds

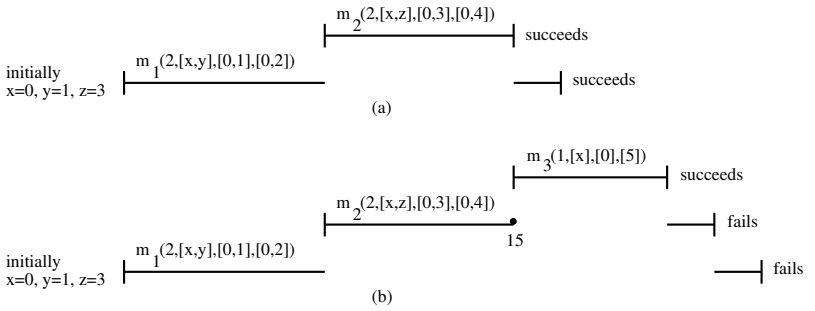


Figure 4.16: (a) Two overlapping MWCAS operations m_1 and m_2. Parameters are shown as (*number of words*, [*list of words accessed*], [*list of old values*], [*list of new values*]). The only potential conflict is on word x, which neither m_1 nor m_2 changes, so both succeed. (b) m_1 and m_2 are overlapped by a third MWCAS operation, m_3, which does change x; m_3 preempts m_2 just after m_2 executes the CAS at line 15. m_3 causes m_2 to fail by updating m_2's *Status* variable; m_2 passes the failure along to m_3 by updating m_3's *Status* variable.

during the execution of this procedure (the value of $Status[q]$ could potentially be changed by a higher-priority task from 0 to 1, but this does not affect the value of $Val(w)$).

A final subtlety that we consider involves MWCAS operations in which some word is accessed but not modified. Such an access should not interfere with accesses of that word by other tasks. To see how this is accomplished in our implementation, consider the situation in Figure 4.16(a). The only word in common between operations m_1 and m_2 is x. Neither operation changes the value of x, so both succeed. In particular, note that m_2 restores the value of x (line 20) before completing. Thus, it appears to m_1 that no other task has updated x. In contrast, consider the situation in Figure 4.16(b). In this situation, m_1 and m_2 are preempted by a third operation m_3 that does modify x. In this case, m_3 causes m_2 to fail by updating m_2's *Status* variable (line 19). m_2 in turn passes the failure along to m_1 by updating m_1's *Status* variable (line 21).

4.6.2 Correctness Proof

Before presenting the correctness proof of our MWCAS implementation, we present some definitions that are used in the proof. Our implementation requires the following of the address-list parameter passed to any invocation of the MWCAS procedure by task T_r.

Requirement 4.1: $(\forall k, l : 0 \leq k, l < r.numwds \ \wedge \ k \neq l :: r.addr[k] \neq r.addr[l])$. □

Requirement 4.1 states that there must be no duplications in the list of addresses passed as inputs to any MWCAS operation. We now state the properties we intend to prove for the implementation. There are four such properties, the first of which is as follows.

Definition 4.6: A MWCAS operation of task T_r *accesses* word w[7] iff $r@12 \ \wedge \ r.i = k \ \wedge \ r.addr[k] = address(w)$ holds at some state during the execution of that operation. □

Definition 4.7: A MWCAS operation of task T_r *modifies* word w iff $r@12 \ \wedge \ r.i = k \ \wedge \ r.addr[k] = address(w) \ \wedge \ r.old[k] \neq r.new[k]$ holds at some state during the execution of that operation. □

Note that a MWCAS operation that modifies word w may in fact fail, in which case that operation actually leaves w unchanged.

Property 4.9: Suppose that task T_r executes statement $r.23$ at state t as a result of an invocation of the Read procedure with input parameter $r.addr = address(w)$. Let v be the value returned by this procedure invocation. Then, $Val(w) = v$ holds at state t. □

[7] Henceforth, we use the term w and $address(w)$ to denote a word of type *wordtype* and the address of that word, respectively.

Property 4.10: Task T_r can change the value of $Val(w)$ only by executing statement $r.15$. □

Property 4.11: Suppose that $r.retval$ is assigned the value *true* by $r.15$. Then, $(\forall k : 0 \leq k < r.numwds :: Val(*r.addr[k]) = r.old[k])$ holds immediately before $r.15$ is executed and $(\forall k : 0 \leq k < r.numwds :: Val(*r.addr[k]) = r.new[k])$ holds immediately after. □

Property 4.9 implies that a Read operation on word w can be linearized to the state at which w is read. Properties 4.10 and 4.11 imply that each successful MWCAS operation can be linearized to the state at which the CAS at line 15 is performed, and that each failed MWCAS operation does not change the value of any of the implemented words.

Property 4.12: Suppose that task T_r executes statement $r.15$ while executing a MWCAS operation. Let t (u) be the state immediately before (after) the first (last) statement execution of that operation. If $r.retval$ is assigned the value *false* by $r.15$, then for some word w, where $r.addr[k] = address(w)$, $0 \leq k < r.numwds$, there exists a state in $[t, u]$ at which $Val(w) \neq r.old[k]$. □

Property 4.12 deals with failed MWCAS operations. If operation m fails, then it can be linearized to a state such that, for some word accessed by m, the current value of that word differs from the old value specified for that word as an input parameter to m. Before proving the properties stated above, we first state and prove a number of lemmas. The first of these lemmas is as follows.

Lemma 4.25: $w.tid = q \wedge \neg w.valid \Rightarrow (\exists k :: w.count = k \wedge ((q@14 \wedge q.i =$

$k \ \wedge \ *addr[k] = w) \ \vee \ (q@\{3..14 \ \wedge \ q.i > k\})) \ \vee \ q@\{15..22\}.$

Proof: If a task T_q performs a MWCAS operation that accesses word w, then statements $q.18$ and $q.20$ ensure that once $q.22$ has been executed, $w.tid \neq q \ \vee \ w.valid$ holds. $w.tid = q \ \wedge \ \neg w.valid$ can be subsequently established only by $q.12$, which also establishes the consequent. □

The next lemma formalizes the claims made previously in the discussion of the example that uses Figure 4.15. This lemma shows that the value of $Val(r.word)$ remain unchanged while T_r computes a return value in the Read procedure. This lemma is used below to prove Property 4.9.

Lemma 4.26: $r@\{24..26\} \ \wedge \ \neg r.word.valid \ \wedge \ r.word.tid = q \ \wedge \ r.word.count = c \ \wedge \ Save[q, c] = b \ \wedge \ Status[q] \neq 2 \ \ unless \ \ \neg r@\{24..26\}.$

Proof: Let $B = \neg r.word.valid \ \wedge \ r.word.tid = q \ \wedge \ r.word.count = c \ \wedge \ Save[q, c] = b \ \wedge \ Status[q] \neq 2$. Our proof obligation is to show that, for any enabled statement s,

$$\{r@\{24..26\} \ \wedge \ B\} \ s \ \{\neg r@\{24..26\} \ \vee \ B\}. \tag{4.7}$$

Statements $r.24$, $r.25$, and $r.26$ do not update any of the variables appearing in B, so (4.7) clearly holds if s is a statement of task T_r. (4.7) also clearly holds if s is not a statement of task T_q (the only variable in B that could be updated by such a statement is $Status[q]$, but a task other than T_q cannot falsify $Status[q] \neq 2$). On the other hand, if s *is* a statement of task T_q, then there is a danger that s modifies either $Status[q]$ or $Save[q, c]$. However, by Lemma 4.25, $r.23$ can establish $r@\{24..26\} \ \wedge \ \neg r.word.valid \ \wedge \ r.word.pid = q$

only if executed when $q@\{3..22\}$ holds, which implies that T_r is of higher priority than T_q. Because T_r has higher priority than T_q, if $r@\{24..26\}$ holds, then T_q has no currently-enabled statement. Thus, s is not a statement of T_q. □

We now prove Property 4.9, which is restated below.

Property 4.9: Suppose that task T_r executes statement $r.23$ at state t as a result of an invocation of the Read procedure with input parameter $r.addr = address(w)$. Let v be the value returned by this procedure invocation. Then, $Val(w) = v$ holds at state t.

Proof: Let r, t, and w be as defined in the statement of the lemma. Let u be the state immediately following the execution of $r.23$. Let v be the value of $Val(w)$ at state u. We need to consider two cases.

Case 1: Suppose that $w.valid$ holds at state u. By Definition 4.5, this implies that $Val(w) = w.val$ holds at state u. From the program text (lines 23-25), and from the fact that the procedure returns v, it follows that $w.val = v$ holds at state u. Because $r.23$ does not modify w, this implies that $Val(w) = v$ holds at state t.

Case 2: Suppose that $\neg w.valid \ \wedge \ Status[w.pid] = 2$ holds at state u. Let q be the value of $w.pid$ at state u. By Lemma 4.25, $q@\{3..22\}$ holds at state u, which implies that T_q is a lower-priority task. If $Status[q] = 2$ holds before the execution of $r.24$, then $Status[q] = 2$ holds at state t. This is because only T_q can establish $Status[q] = 2$ and because T_q cannot take steps after the execution of $r.23$ until T_r completes its operation. By Definition 4.5, and from the fact that the value v is returned by Read, this implies that $w.val = v$ holds

at state u, and hence at state t also.

On the other hand, if $Status[q] \neq 2$ holds before the execution of $r.24$, then by Lemma 4.26, $Val(r.word) = v$ holds at state u (the lemma implies that the value of $Val(r.word)$ is stable while the return value of the Read procedure is being determined). By the program text, $r.23$ establishes $r.word = w$. Thus, $Val(w) = v$ holds at state u — and hence also at state t. □

Lemma 4.27: $((r@\{3..22\} \wedge r.i > k) \vee (r@\{5..22\} \wedge r.i = k)) \wedge \neg r.init[k].valid \wedge r.init[k].tid = q \wedge r.init[k].count = c \wedge Save[q, c] = b \wedge Status[q] \neq 2$ *unless* $\neg r@\{3..22\}$.

Proof: Similar to the proof of Lemma 4.26. □

Corollary 4.2: $((r@\{3..22\} \wedge r.i > k) \vee (r@\{5..22\} \wedge r.i = k)) \wedge Val(r.init[k]) = v$ *unless* $\neg r@\{3..22\}$. □

According to the next lemma, if the CAS at line 12 succeeds when executed by a task T_r, then T_r has a "correct" old value for the word with address $r.addr[i]$.

Lemma 4.28: $r@12 \wedge r.i = k \wedge *r.addr[k] = r.init[k] \Rightarrow Val(*r.addr[k]) = r.old[k]$.

Proof: If statement $r.4$ is executed when $r@4 \wedge r.i = k$ holds, then it establishes

$$r@5 \wedge r.i = k \wedge Val(r.init[k]) = v \tag{4.8}$$

for some value v. By Corollary 4.2, $Val(r.init[k])$ has the same value at the state prior to the execution of $r.12$ as it does at the state following the execution of $r.4$. Thus,

$$r@12 \wedge r.i = k \wedge *r.addr[k] = r.init[k] \Rightarrow Val(*r.addr[k]) = v. \tag{4.9}$$

By the program text (lines 4 through 7), Lemma 4.27, the definition of v, and Definition 4.5,

$$r@12 \ \wedge \ r.i = k \ \Rightarrow \ r.val = v \tag{4.10}$$

By the program text (line 9),

$$r@12 \ \wedge \ r.i = k \ \Rightarrow \ r.old[k] = r.val. \tag{4.11}$$

Combining (4.9), (4.10), and (4.11), we have $r@12 \ \wedge \ r.i = k \ \wedge \ *r.addr[k] = r.init[k] \ \Rightarrow \ Val(*r.addr[k]) = r.old[k]$. □

Lemma 4.29: Let $[t, u]$ be an interval of states during the execution of a MWCAS operation by task T_r. If T_r performs no CAS on w (lines 12, 18, and 20) in the interval $[t, u]$, and if no successful MWCAS operation that modifies w executes within $[t, u]$, then w has the same value at both states t and u.

Proof: Let $[t, u]$ be an interval of states during the execution of a MWCAS operation m by task T_r. Assume that T_r performs no CAS on w in $[t, u]$, and that no successful MWCAS operation that modifies w executes within $[t, u]$. Given these assumptions, we show that w has the same value at both states t and u.

Observe that w can be modified in $[t, u]$ only if some task executes a CAS on w at line 12, 18, or 20. By assumption, the only task that may do so is a task other than T_r that executes a MWCAS operation that accesses w within $[t, u]$ or that fails its attempt to modify w within $[t, u]$. By our priority-based task model, such operations "overlap" m as illustrated in Figure 4.17.

We now prove by induction on priority level that each overlapping MWCAS oper-

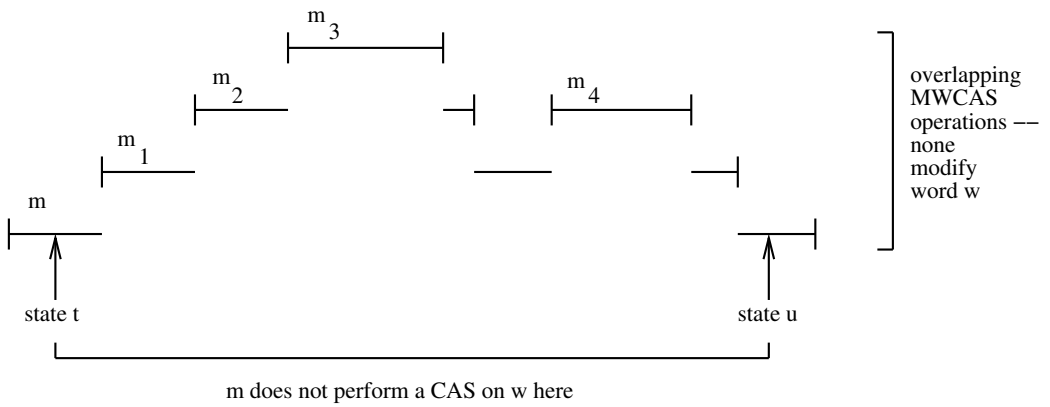


Figure 4.17: Proof of Lemma 4.29.

ation restores the value of w to that which existed before the execution of that operation. Consider an overlapping MWCAS m' that accesses w or that fails its attempt to modify w, and assume that all higher-priority MWCAS operations that access w successfully restore the value of w. Because m' accesses w or fails to modify w, m' executes the CAS at line 20 on word w. Because all MWCAS operations that overlap m' either do not access w or successfully restore the value of w, this CAS succeeds, restoring the value of w. It follows from this argument that w has the same value at both states t and u. □

We now prove Property 4.10, which is repeated below.

Property 4.10: Task T_r can change the value of $Val(w)$ only by executing statement $r.15$.

Proof: Task T_r could potentially change the value of $Val(w)$ by executing any of the statements $r.1$, $r.8$, $r.10$, $r.12$, $r.13$, $r.15$, $r.18$, $r.19$, $r.20$, and $r.21$. These are the statements that can change either w itself or a component of the *Save* or *Status* arrays. We consider

each statement other than $r.15$ below.

Statement $r.1$: By the contrapositive of Lemma 4.25, $w.tid \neq r \ \vee \ w.valid$ holds prior to the execution of $r.1$. Thus, this statement does not change the value of $Val(w)$.

Statement $r.8$: By the contrapositive of Lemma 4.25, $w.tid \neq r \ \vee \ w.valid \ \vee \ w.count \neq r.i$ holds prior to the execution of $r.8$. Because the execution of $r.8$ changes $Val(w)$ only if $\neg w.valid \ \wedge \ w.tid = r$ holds prior to the execution of $r.8$, it follows that $r.8$ does not change the value of $Val(w)$.

Statement $r.10$: $Status[r] = 0$ is established by statement $r.1$, and $Status[r] = 2$ can only be subsequently established by statement $r.15$. Thus, $Status[r] \neq 2$ holds before $r.10$ is executed. It follows that $r.10$ cannot change the value of $Val(w)$.

Statement $r.12$: Let u (u') be the state immediately preceding (following) the execution $r.12$ when $r.i = k$ and $r.addr[k] = address(w)$. The CAS of statement $r.12$ could potentially change the value of $Val(w)$ only if it succeeds. Suppose this is the case, i.e., $r@12 \ \wedge \ r.i = k \ \wedge \ w = r.init[k]$ holds at state u. By Lemma 4.28, this implies that $Val(w) = r.old[k]$ holds at u.

We now show that $Val(w) = r.old[k]$ holds at state u'. First, observe that $r.8$ establishes $Save[r, k] = r.val$ when $r.i = k$, and that $r.9$ ensures that $r.val = r.old[k]$ holds at u' (see line 11). Combining these two facts, we have that $Save[r, k] = r.old[k]$ holds at u'. Because $r.12$ succeeds when executed at u, we also know that $\neg w.valid \ \wedge \ w.tid = r \ \wedge \ w.count = k$ holds at u'. In addition, observe that $r.1$ establishes $Status[r] = 0$ and $r.15$ is the only statement that can establish $Status[r] = 2$. Hence, $Status[r] \neq 2$ holds at

state u'. Putting all of this together, we can assert that the following expression holds at state u'.

$$\neg w.valid \ \wedge \ w.tid = r \ \wedge \ w.count = k \ \wedge \ Save[r,k] = r.old[k] \ \wedge \ Status[r] \neq 2$$

By Definition 4.5, this implies that $Val(w) = r.old[k]$ holds at state u'. Thus $Val(w)$ has the same value at both states u and u'.

Statement $r.13$: Similar to the proof for statement $r.10$.

Statement $r.18$: Suppose $r.18$ is executed at state u. Note that the value of $Val(w)$ could potentially be changed by $r.18$ only if $r@18 \ \wedge \ r.j = k \ \wedge \ r.addr[k] = address(w) \ \wedge \ w = r.assn[k]$ holds at u. Also, observe that statement $r.18$ is executed only if statement $r.15$ previously established $Status[r] = 2$. No higher-priority task can falsify $Status[r] = 2$ before state u by executing line 19 or 21. Thus, we have the following at state u.

$$r.j = k \ \wedge \ w = r.assn[k] \ \wedge \ Status[r] = 2$$

Because of the value assigned to $r.assn[k]$ by $r.11$ (line 11), this implies that $Val(w) = r.new[k]$ holds at u. By examining statement $r.18$, it is clear that $Val(w) = r.new[k]$ also holds at the state following u.

Statement $r.19$: The value of $Val(w)$ could potentially be changed by $r.19$ only if $r@19 \ \wedge \ r.j = k \ \wedge \ r.init[k].tid = q \ \wedge \ w.tid = q \ \wedge \ \neg w.valid \ \wedge \ Status[q].tid = 0$ holds. However, in this case, $r.19$ establishes $Status[q].tid \neq 2$. This implies that the value of $Val(w)$ is not changed by the execution of $r.19$.

Statement $r.20$: Suppose that $r.20$ is executed at state u. Note that the value of $Val(w)$ could

potentially be changed by $r.20$ only if $r@20 \ \wedge \ r.j = k \ \wedge \ r.addr[k] = address(w) \ \wedge \ w = r.assn[k]$ holds at u. Also, observe that statements $r.8$ and $r.9$ ensure that $Save[r,k] = r.old[k]$ holds at state u (note that if $r.old[k] \neq r.val$ holds when $r.9$ is executed, then $r.i$ is not incremented by $r.14$ and the loop at lines 3 through 14 terminates). Furthermore, note that statement $r.20$ is executed only if $r.15$ did not previously establish $Status[r] = 2$ — which implies that $Status[r] \neq 2$ holds at u — or if $r.old[k] = r.new[k]$ holds at u. Putting all of this together, we conclude that the following holds at state u.

$$r.j = k \ \wedge \ w = r.assn[k] \ \wedge \ Save[r,k] = r.old[k] \ \wedge$$
$$(Status[r] \neq 2 \ \vee \ (Status[r] = 2 \ \wedge \ r.old[k] = r.new[k]))$$

This implies that $Val(w) = r.old[k]$ holds at u. By examining lines 4 through 9, and from Lemma 4.27, it follows that $Val(r.init[k])$ is stable between $r.4$ and $r.9$. Hence, $Val(r.init[k]) = r.old[k]$ holds at state u, and statement $r.22$ does not change the value of $Val(w)$.

Statement $r.21$: Similar to the proof for statement $r.19$. □

Next, we prove Property 4.11, which is repeated below.

Property 4.11: Suppose that $r.retval$ is assigned the value *true* by $r.15$. Then, $(\forall k : 0 \leq k < r.numwds :: Val(*r.addr[k]) = r.old[k])$ holds immediately before $r.15$ is executed and $(\forall k : 0 \leq k < r.numwds :: Val(*r.addr[k]) = r.new[k])$ holds immediately after.

Proof: Consider a MWCAS operation m of task T_r. Let u be the state immediately prior to the execution of $r.15$ by m. Suppose that $r.retval$ is assigned the value *true* by $r.15$, i.e.,

$Status[r] = 0$ holds at u. Consider k, where $0 \leq k < r.numwds$, and let $w = *r.addr[k]$. Our proof obligation is to show that $Val(w) = r.old[k]$ holds at u and that the execution of $r.15$ at state u establishes $Val(w) = r.new$.

To this end, let t be the state immediately following the execution of the CAS at $r.12$ by m when $r.i = k \ \wedge \ r.addr[k] = address(w)$ holds. Since $Status[r] = 0$ holds at state u, the CAS at $r.12$ succeeds when $r.i = k \ \wedge \ w = r.init[k]$ holds (otherwise, $Status[r] \neq 0$ is established by $r.13$). By Property 4.10, the execution of $r.12$ by m does not change $Val(w)$. By Lemma 4.28, this implies that $Val(w) = r.old[k]$ holds at state t.

By Lemma 4.29, this implies that either $Val(w) = r.old[k]$ holds at state u, or some successful MWCAS operation that modifies w executes within the interval $[t, u]$. However, in the latter case, because $w.tid = r$ at state t, the first such MWCAS to read w at line 4 would establish $Status[r] = 1$ at line 19. This would imply that $Status[r] \neq 0$ at state u, which is a contradiction. We therefore conclude that $Val(w) = r.old[k]$ holds at state u.

Because the CAS at $r.12$ succeeds, the execution of $r.12$ establishes $w.val = r.new \ \wedge \ w.tid = r$ at state t. As explained above, no other task modifies w in the interval $[t, u]$. Hence, it follows that $w.val = r.new \ \wedge \ w.tid = r$. The execution of $r.15$ at state u establishes $Val(w) = r.new$ by establishing $Status[r] = 2$. This establishes our proof obligation. □

Finally, we prove Property 4.12, which is restated below.

Property 4.12: Suppose that task T_r executes statement $r.15$ while executing a MWCAS operation. Let t (u) be the state immediately before (after) the first (last) statement

execution of that operation. If $r.retval$ is assigned the value *false* by $r.15$, then for some word w, where $r.addr[k] = address(w)$, $0 \leq k < r.numwds$, there exists a state in $[t, u]$ at which $Val(w) \neq r.old[k]$ holds .

Proof: Consider a MWCAS operation m by task T_r. Let t be the state immediately following the execution of statement $r.1$ by m, and let u be the state immediately prior to the execution of statement $r.18$ by m. Suppose that the CAS of line 15 executed by m fails, i.e., $Status[r] \neq 0$ at state u. Observe that $Status[r] = 0$ at state t. Thus, T_r or some higher-priority task assigns the value 1 to $Status[r]$ between states t and u by executing line 10, 13, 19, or 21.

Suppose that $Status[r]$ is assigned the value 1 by T_r at line 10 when $r.i = k$ holds, where $k \leq r.numwds$. By examining lines 4 though 9 of the program, we see that the execution of $r.4$ during the k^{th} loop iteration establishes $Val(*r.addr[k]) = Val(r.init[k])$. By Lemma 4.27, it follows that $Val(r.init[k])$ does not change between $r.4$ and $r.9$. Combining the above facts, it follows that $r.old[k] \neq Val(*r.addr[k])$ holds in the state following the execution of $r.4$ by m when $r.i = k$ holds.

Suppose next that $Status[r]$ is assigned the value 1 by T_r at line 13. Then, the CAS at line 12 fails for some word w when $r.i = k \ \wedge \ w \neq r.init[k] \ \wedge \ w.tid = q$ holds, where $address(w) = r.addr[k]$. Because T_r previously established $w = r.init[k]$ at line 4 and because $w.tid = q$ holds at line 12, it follows that w was modified by task T_q during T_r's operation. Task T_q has higher priority than T_r because lower-priority tasks cannot modify w during T_r's execution. By the contrapositive of Lemma 4.25, it follows that $w.valid$ holds at $r.12$. This implies that T_q succeeded the CAS at $q.18$. From the program text (line 17),

it follows that if the execution of $q.18$ modifies w, then $q.j = l \ \wedge \ q.retval \ \wedge \ q.old[l] \neq q.new[l] \ \wedge \ addr[l] = address(w)$ holds before the execution of $q.18$. This implies that T_q's MWCAS operation successfully executes the CAS at $q.15$. By Property 4.11, T_q changes $Val(w)$. It follows that $Val(w) \neq r.old$ holds in the state preceding or the state following the execution of $q.15$.

Suppose next that $Status[r]$ is assigned 1 by T_r or some higher-priority task T_q at line 19. We consider two cases.

Case 1: Suppose T_r modifies $Status[r]$ by executing $r.19$. Observe that the execution of $r.19$ by m modifies $Status[r]$ only if $r.j = k \ \wedge \ r.addr[k] = address(w) \ \wedge \ \neg r.init[k].valid \ \wedge \ r.init[k].tid = r$ holds prior the execution of $r.19$ by m. This implies that $r.i = k \ \wedge \ r.addr[k] = address(w) \ \wedge \ \neg w.valid \ \wedge \ w.tid = r$ holds prior to the execution of $r.4$ by m when $r.i = k$ holds. By Requirement 4.1, it follows that $w.tid = r$ was established by some previous MWCAS operation m' invoked by T_r. By Lemma 4.25, this implies that $w.valid \ \wedge \ r.addr[k] = address(w)$ holds prior to the execution of $r.4$ by m, a contradiction.

Case 2: Suppose that $Status[r]$ is assigned 1 by some higher-priority task T_q that executes $q.19$ when $q.j = k \ \wedge \ address(w) = q.addr[k]$ holds. Observe that task T_q executes $q.19$ only if $q.retval \ \wedge \ q.old[k] \neq q.new[k]$ holds before the execution of $q.19$. From Property 4.11 and from the program, it follows that T_q's MWCAS successfully changes $Val(w)$ from $q.old[k]$ to $q.new[k]$ by executing $q.15$. Therefore, $Val(w) \neq r.old$ holds in the state preceding or the state following the execution of $q.15$.

Finally, suppose that $Status[r]$ is assigned the value 1 by some higher-priority

task T_q at line 21. (We can show that $Status[r]$ is not assigned the value 1 by T_r using the argument outlined in Case 1 above.) Suppose that $q.21$ is executed at state v when $q.j = k \ \wedge \ q.addr[k] = address(w) \ \wedge \ r.init[k].tid = r \ \wedge \ \neg r.init[k].valid$ holds. From the program text (line 16), it follows that $q.i > q.j$ holds at state v. Hence, $q.i > k$ holds at state v. This implies that $w = q.assn[k]$ is established by the execution of $q.12$ when $r.i = k$ holds (observe that if T_q fails the CAS at $q.12$, then $q.i$ is not incremented by $q.14$ and the loop at lines 3 through 14 terminates when $q.i = k$). However, T_q executes $q.21$ only if $w \neq q.assn[k]$ holds at $q.20$. By the contrapositive of Lemma 4.29, some successful MWCAS operation m' modifies the value of w after the execution of $q.12$ by m when $q.i = k$ holds and before the execution of $q.20$ by m. By Property 4.10, m' changes $Val(w)$ by executing line 15. Hence, $Val(w) \neq r.old$ holds in the state preceding or the state following the execution of line 15 by m'. □

The following theorem follows from the program code and from the fact that our MWCAS implementation satisfies the required properties.

Theorem 4.4: A Read operation and a W-word MWCAS operation can be implemented in a wait-free manner from CAS with $O(1)$ and $O(W)$ time complexity, respectively, on a real-time uniprocessor system. □

Chapter 5

A Transactional Framework for Implementing Lock-Free Objects

In this chapter, we present a framework for implementing lock-free transactions and multi-object operations — the lock-free counterpart to nested critical sections — on memory-resident data. The framework that we present is based on universal lock-free constructions by Anderson and Moir for implementing large objects and for implementing multi-object operations in asynchronous systems [7, 8]. The behavior of transactions implemented under our transactional framework is very similar to the behavior of transactions implemented under conventional optimistic concurrency control (OCC) schemes [51].

One aspect in which our implementation differs from conventional OCC implementations is that we use a strong synchronization primitive at the user level to validate and commit transactions — transactions can be preempted during their validate and commit phases. In contrast, transactions in conventional OCC schemes cannot be preempted dur-

ing their validate and commit phases [40] — such transactions entail a blocking factor due to the validate and commit phases of lower-priority transactions. Our transactional framework differs from conventional real-time database systems in another aspect. In conventional memory-resident database systems, major functional components are implemented as separate modules (e.g., transaction manager, lock manager, etc.), each of which consists of one or more processes. Transactions interact with these modules by invoking special calls (e.g., *Begin_Transaction*, *End_Transaction*, *Read*, *Write*, *Commit*, *Abort*). Although structuring a system in this manner is attractive from a software engineering standpoint, such an arrangement potentially can result in significant interprocess communication overhead. In contrast, transactions in our implementation are invoked by a collection of prioritized tasks executing on the processor. Transactions of a task access shared data by invoking user-level routines.

5.1 Lock-Free Transactions

In this section, we present a detailed description of our transaction implementation. Our implementation consists of three procedures, *Tr_Read*, *Tr_Write*, and *Tr_Exec*, which are given in Figure 5.2. These procedures support the "illusion" of a contiguous shared array *MEM* of memory words. In reality, data is not stored in contiguous locations of memory, but is composed of a number of blocks. The user writes transaction code in a sequential manner using the *Tr_Read* (*Tr_Write*) procedure to read words from (write words to) the *MEM* array. The *Tr_Exec* procedure takes this user-supplied transaction code as input and executes it within the body of a lock-free retry loop. The input transaction is validated and

```
constant Tail = 0; Head = 1; maxsize = 10
local variable newtail: integer
procedure enqueue(item: integer returns (SUCCESS, FULL)
    Tr_Write(Tr_Read(Tail),item);
    newtail = (Tr_Read(Tail) + 1) mod maxsize;
    if (newtail == Tr_Read(Head)) then
      return FULL
    fi;
    Tr_Write(Tail, newtail);
    return SUCCESS
```

Figure 5.1: An example transaction.

committed in *Tr_Exec* using a MWCAS operation. Because we assume that the MWCAS operation is implemented as described in Section 4.6, the MWCAS primitive is used in conjunction with the Read operation described in that section. Also, we assume that Read and MWCAS operations are atomically executed because, as shown in the correctness proof of the MWCAS/Read implementation in Section 4.6, these operations linearize to a distinct step.

When a transaction τ of task T_p accesses a word in the implemented array of memory words, say $MEM[k]$, the block containing the k^{th} word is identified. If τ writes into $MEM[k]$, then T_p must replace the corresponding block. The details of identifying blocks and replacing modified blocks are hidden from the programmer by means of the *Tr_Read* and *Tr_Write* routines, which perform all necessary address translation and bookkeeping. These routines are called within the programmer's transaction code in order to read or write a word of the *MEM* array. Thus, instead of writing "$MEM[1] := MEM[10]$", the programmer would write "*Tr_Write*(1, *Tr_Read*(10))". Figure 5.1 shows a simple example transaction, which enqueues an item onto a shared queue. This transaction is executed by a task by calling *Tr_Exec*(*enqueue*).

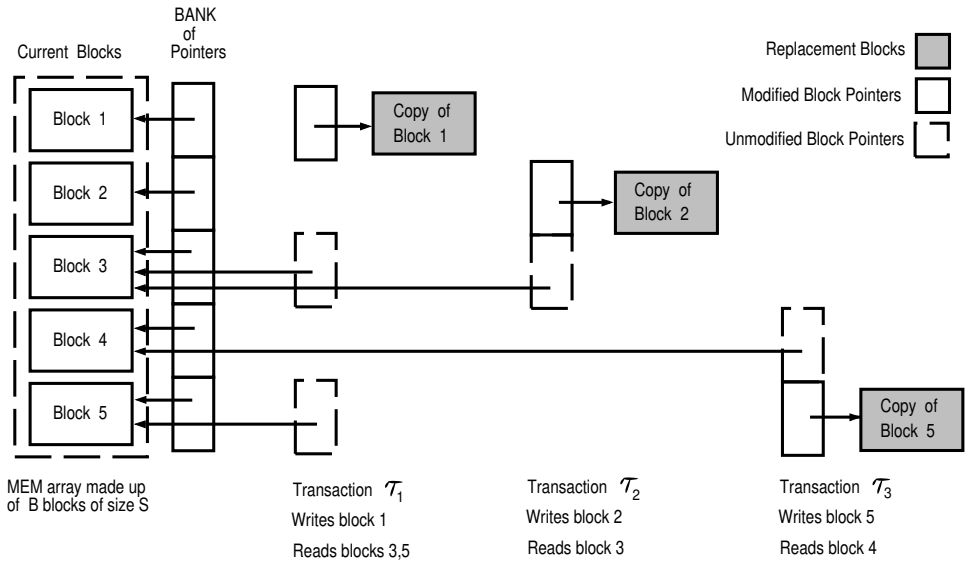


Figure 5.2: Implementation of the *MEM* array for lock-free transactions (depicted for $B = 5$).

The implemented array *MEM* is partitioned into B blocks of size S. (We assume a constant block size here for simplicity.) Figure 5.2 depicts this arrangement for $B = 5$. The first block contains array locations 0 through $S - 1$, the second contains locations S through $2S - 1$, and so on. A bank of pointers — one for each block — is used to point to the blocks that make up the array. (These are really array indices, not pointers.) In order to modify the contents of *MEM*, a task makes a copy of each block to be modified, and then attempts to atomically replace the old blocks with their modified copies using the MWCAS primitive.

In Figure 5.2, *BANK* is a B-word shared array. Each element of *BANK* contains a pointer to a block of size S and a version number (see below) for that pointer. The B blocks pointed to by *BANK* constitute the current version of the *MEM* array. We assume that an upper bound C is known on the number of blocks modified by any transaction. Because a

```
constant  N = Number of tasks in the system
          B = Number of blocks that constitute array MEM
          S = Block size in words
          C = Maximum number of blocks modified by a transaction

type
    blktype = array [0..S − 1] of memwdtype;
    valtype = record blid: 0..B + NC − 1; ver: 0..V − 1 end;
    wdtype = record val: valtype; count: 0..B − 1; valid: boolean; pid: 0..N − 1 end
                    /* The count, valid, and pid fields are used by the MWCAS/Read procedures */

shared var
    BANK: array [0..B − 1] of wdtype;                    /* Bank of pointers to array blocks */
    BLK: array [0..B + NC − 1] of blktype                /* Array and copy blocks */

initially
(∀k : 0 ≤ k < B :: BANK[k] = ((NC + k, 0), 0, 0, true, 0) ∧ BLK[NC + k] = initial value of k^th block)

private var
    copy: array [0..C − 1] of 0..B + NC − 1;             /* Indices for copy block of task T_p */
    curr: array [0..B − 1] of valtype;                   /* Task T_p's current view of the MEM array */
    addrlist: array [0..B − 1] of pointer to wdtype;     /* Addresses for MWCAS */
    blklist: array [0..B − 1] of 0..B − 1;               /* List of blocks that have been accessed */
    oldval,newval: array [0..B − 1] of valtype;          /* Old and new values for MWCAS */
    dirty: array [0..B − 1] of 0..2;                     /* 0 if block not accessed, 1 if read, 2 if modified */
    dcnt: 0..C − 1; done: boolean; i, j,numblks,blk: 0..B; tmp: 0..B + NC − 1;
    env: jmp_buf                                         /* Used by setjmp and longjmp system calls */

initially (∀k : 0 ≤ k < C :: copy[k] = pC + k) ∧ (∀k : 0 ≤ k < B :: dirty[k] = 0)

procedure Tr_Read(memwd: 0..BS − 1)
                                        returns memwdtype
1:  blk := memwd div S;
2:  if dirty[blk] = 0 then
3:    dirty[blk] := 1;
4:    curr[blk] := Read(&BANK[blk]);
5:    addrlist[numblks] := &BANK[blk];
6:    blklist[numblks] := blk;
7:    oldval[numblks] := curr[blk];
8:    numblks := numblks + 1
    fi;
9:  v := BLK[curr[blk].blid][memwd mod S];
10: if Read(&BANK[blk]) = curr[blk] then
      return v
11: else longjmp(env, 1)
    fi

procedure Tr_Write(memwd: 0..BS − 1;
                   value: memwdtype)
12: blk := memwd div S;
13: if dirty[blk] = 0 then
14:   curr[blk] := Read(&BANK[blk]);
15:   addrlist[numblks] := &BANK[blk];
16:   blklist[numblks] := blk;
17:   oldval[numblks] := curr[blk];
18:   numblks := numblks + 1
    fi;
19: if dirty[blk] ≠ 2 then
20:   dirty[blk] := 2;
21:   memcpy(BLK[copy[dcnt]],
                BLK[curr[blk].blid],
                sizeof(blktype));
22:   curr[blk].blid := copy[dcnt];
23:   dcnt := dcnt + 1
24:   if Read(&BANK[blk]) ≠ curr[blk] then
25:     longjmp(env, 1)
      fi
    fi;
26: tmp := curr[blk].blid;
27: BLK[tmp][memwd mod S] := value
```

Figure 5.3: Lock-free transaction implementation.

```
procedure Tr_Exec(tr: function_ptr)
28:   done := false;
29:   while ¬done do
30:     dcnt, numblks := 0, 0;
31:     if setjmp(env) ≠ 1 then
32:       *tr();
33:       for j := 0 to numblks − 1 do
34:         i := blklist[j];
35:         newval[j] := curr[i];
36:         if dirty[i] = 2 then
37:           newval[j].ver := newval[j].ver + 1 mod V
            fi
          od;
38:       done := MWCAS(numblks, addrlist, oldval, newval)
        fi;
39:     i := 0;
40:     for j := 0 to numblks − 1 do
41:       if done ∧ dirty[blklist[j]] = 2 then
42:         copy[i] := oldval[blklist[j]].blid;
            i := i + 1
          fi;
43:       dirty[blklist[j]] := 0
        od
      od
```

Figure 5.3: (continued) Lock-free transaction implementation.

task's transaction copies a block before modifying it, C "copy" blocks are required per task. Therefore, a total of $B + NC$ blocks are used. These blocks are stored in the array BLK. Initially, blocks $BLK[NC]$ to $BLK[NC + B - 1]$ are the blocks of the MEM array, and $BLK[pC]$ to $BLK[(p+1)C-1]$ are task T_p's copy blocks. However, the roles of these blocks are not fixed. If T_p successfully completes a transaction, then T_p reclaims the replaced blocks as copy blocks (lines 39-43). Thus, some of T_p's copy blocks become part of the current array, and vice versa.

As mentioned above, user-supplied transaction code accesses the MEM array in a sequential manner using the *Tr_Read* and *Tr_Write* procedures. The *Tr_Read* procedure first computes the index of the block containing the accessed word (line 1). If the block

has not yet been read by this transaction, then it is marked as having been read (line 3), and is recorded in the transaction's *curr* array (line 4). This array gives the transaction's "current view" of *MEM*. The block index is also recorded in an array *blklist* (line 6), which is used later in reclaiming copy blocks when the transaction successfully completes. In addition, the address and old value of the block pointer are saved in arrays (lines 5 and 7) that are later used as parameters to the MWCAS procedure. The new value of the block pointer is determined later, prior to invoking MWCAS (lines 33-37). The *Tr_Read* procedure completes by retrieving a value from the appropriate offset within the block that is accessed (line 9), and by performing a consistency check (lines 10 and 11), the purpose of which we describe below. The *Tr_Write* procedure is similar to the *Tr_Read* procedure, except that, when a block is first modified, it is recorded as having been modified (line 20), and a local copy of the block is made (line 21).

The *ver* counter associated with each block pointer in *BANK* records the current version number of the corresponding block. If a transaction successfully replaces a modified block, then it increments that block's version number. In contrast, if a block is read but not modified, then its block pointer and version number are not changed. This ensures that read-only transactions do not interfere with each other. A transaction can determine whether the i^{th} block has been changed by comparing the version number that it last read from $BANK[i]$ to the current version number of $BANK[i]$. The *ver* counter is assumed to be large enough so that it cannot cycle around during the execution of any transaction. Specifically, in a system with N periodic tasks that invoke transactions, the version counter of a block will not cycle during the execution of a transaction if the size of the counter

is larger than P_{max}/P_{min}, where P_{max} and P_{min} are the longest period and the shortest period, respectively, of all tasks that access the *MEM* array.

Before concluding this description, one subtlety that we have glossed over must be mentioned. If the *BANK* variable is modified by a transaction of task T_q during the execution of a transaction of some lower-priority task T_p, then T_p may read inconsistent values from the *MEM* array. Because its MWCAS operation will subsequently fail, T_p will not be able to install corrupted data. However, there is a risk that T_p's sequential operation might cause an error, such as a division by zero or a range error. This problem is solved by ensuring that, if the version number of one of the blocks accessed by a transaction changes during that transaction, then control is returned from the *Tr_Read* or *Tr_Write* procedure to line 31 in *Tr_Exec* using Unix-like *longjmp* calls. In this event, relevant data structures are reinitialized (lines 40-43) and the transaction is retried. Transactions can take advantage of this mechanism by re-reading previously accessed blocks in order to fail early in the event that such a block has been modified by another transaction.

In our implementation, read-only transactions do not interfere with one another, nor do transactions that modify disjoint sets of blocks. This is illustrated in Figure 2.7, which depicts three concurrent transactions τ_1, τ_2, and τ_3. Transactions τ_1 and τ_2 do not interfere with each other because neither of them modifies a block accessed by both. However, τ_3 can potentially interfere with τ_1 because τ_3 modifies block 5, which is read by transaction τ_1.

The transaction implementation in Figure 5.2 can be optimized in several ways. For example, the code can be modified to support different block sizes. This would help

to avoid false-sharing and fragmentation problems when using the *MEM* array to store a mixed collection of objects of different sizes. Also, small objects such as queues can be incorporated into the implementation without using a copy-based solution. The *Tr_Read* and *Tr_Write* procedures can also be optimized by ensuring that the overhead associated with calling the *Tr_Read* and the *Tr_Write* procedures is incurred only when a block is accessed for the first time during a transaction; subsequent accesses to that block can be executed as macros. Also, if every transaction accesses or modifies only a few blocks, then the MWCAS operation can be implemented as a short kernel call; interrupts are disabled during the execution of the call.

5.1.1 Correctness Proof

We now state and prove the properties required for the correctness of our implementation. We begin by defining a few terms.

Definition 5.1: A transaction τ *accesses* block k iff τ invokes a *Tr_Read* operation that reads a word in the block pointed to by $BANK[k]$. □

Definition 5.2: A transaction τ *modifies* block k iff τ invokes a *Tr_Write* operation that writes a word in the block pointed to by $BANK[k]$. □

Definition 5.3: An *update* of a transaction τ constitutes the execution of one iteration of the lock-free retry-loop at lines 29-43. □

Definition 5.4: An update of a transaction τ is *successful* iff the MWCAS operation at line 38 is successfully executed during that update; the update *fails* otherwise. □

According to Definitions 5.3 and 5.4, a transaction τ can be viewed as a sequence of zero or more failed updates followed by one successful update. The correctness of our implementation is contingent on the following requirement.

Requirement 5.1: *During the execution of any transaction, the version number of each block pointer in the BANK array changes no more than V times, where V is the size of the ver field in the BANK array.* □

We now state and prove some simple properties of our implementation.

Lemma 5.1: *At any given state, each task has a distinct set of copy blocks that are different from other task's copy blocks and from the blocks that constitute the MEM array.*

Proof: Initially the above lemma holds because the copy blocks of task T_r are different from the blocks constituting the *MEM* array and from other tasks' copy blocks. The above lemma can be falsified only if a transaction τ of some task T_r executes $r.38$ successfully and then reclaims a block at $r.42$ that is either one of the blocks currently constituting the *MEM* array or a copy block of some other task T_q. However, if the lemma holds before the execution of $r.38$ by τ, then it also holds after T_r reclaims its copy blocks at $r.42$. To see why this is so, observe that, for any block b_i modified by τ, $curr[b_i]$ points to one of T_r's own copy block. Along with the program text (lines 33-37) and our assumption that the lemma holds until the execution of $r.38$, this implies that the successful execution of $r.38$ by τ ensures that (i) the blocks placed by τ in *MEM* are different from other blocks in *MEM* and from other tasks' copy blocks; (ii) the blocks displaced from the *MEM* array are no longer part of the *MEM* array or parts of another task's copy blocks. These facts

imply that blocks reclaimed by τ (lines 40-43) are different from all other tasks' copy blocks. Therefore, the above lemma holds after T_r reclaims its copy blocks in the loop at line 40. □

In the following proofs, we do not explicitly quote Lemma 5.1; we assume that it always holds.

Because a task only modifies its own copy blocks or its private variables when it executes statements other than line 38, it is easy to see that the following lemma holds.

Lemma 5.2: *A transaction modifies the contents of the MEM array only by executing the MWCAS operation at line 38.* □

Observe that a failed update either does not execute line 38 (if a *longjmp* call is invoked during a *Tr_Read* or *Tr_Write* procedure call) or fails the MWCAS at line 38. In either case, the *MEM* array is not modified by the transaction, and we have the following.

Lemma 5.3: *A failed update of a transaction τ does not modify the MEM array.* □

Lemma 5.4: *If a transaction invoked by some task T_r accesses some block b_i for the first time during a successful update by calling the Tr_Read procedure when $r.numblks = l'$, then the following holds immediately after the execution of r.10 during that Tr_Read procedure call.*

$$r.numblks \geq l' \ \wedge \ r.blk = b_i \ \wedge \ r.curr[b_i] = r.oldval[l'] \ \wedge$$
$$r.oldval[l'] = *r.addrlist[l'] \ \wedge \ *r.addrlist[l'] = BANK[b_i] \ \wedge \ r.dirty[b_i] = 1 \quad (5.1)$$

Proof: Suppose that a task T_r invokes a transaction τ that accesses block b_i. Let $r.numblks = l'$ hold when τ calls procedure *Tr_Read* to access block b_i for the first time during a successful update m. Observe that τ accesses block b_i during that procedure call only if $r.memwd$ **div** $S = b_i$ holds during that call. Also, observe that $r.dirty[b_i] = 0$ holds before the execution $r.29$ during an update. (For any $0 \leq l \leq B-1$, $r.dirty[l] = 0$ holds initially. It is also established by $r.43$ before T_r completes the execution of an update.) Also, from our assumption that m is a successful update, it follows that the T_r succeeds the test at $r.10$. Therefore, $r.curr[r.blk] = BANK[r.blk]$ holds before the execution of $r.10$ by τ. From this fact, and by examining lines 3-8 of the program code, we see that (5.1) holds immediately before the execution of $r.10$, and hence immediately after. □

Lemma 5.5: *If a transaction invoked by some task T_r modifies some block b_i for the first time during a successful update by calling the Tr_Write procedure when $r.numblks = l'$, then the following holds immediately after the execution of $r.24$ during that Tr_Write procedure call.*

$$r.numblks \geq l' \ \wedge \ r.blk = b_i \ \wedge \ r.curr[b_i] \neq r.oldval[l'] \ \wedge$$
$$r.oldval[l'] = *r.addrlist[l'] \ \wedge \ *r.addrlist[l'] = BANK[b_i] \ \wedge \ r.dirty[b_i] = 2 \quad (5.2)$$

Proof: Similar to the proof of Lemma 5.4. □

Lemma 5.6: *Suppose that a transaction invoked by task T_r accesses block b_i for the first time during a successful update by calling procedure Tr_Read when $r.numblks = l'$ holds and that the procedure call establishes* (5.1)*. Then,* (5.1) *can only be falsified by the execution of line* 38 *during a successful update by T_r or some higher-priority task that modifies block b_i.*

Proof: Let r, b_i, and l' be as defined in the lemma. Suppose that task T_r invokes a transaction τ and that (5.1) is established during the execution of a successful update m by τ. Observe that (5.1) cannot be falsified during that update by subsequent *Tr_Read* calls by that access block b_i. (This is because τ will fail the test at $r.2$, and hence not modify any of the variables in (5.1).) It follows that after τ establishes (5.1), the expression can be falsified only by changing the value of $BANK[b_i]$. However, by Lemma 5.2, $BANK[b_i]$ can only be changed by the execution of line 38, and, by the real-time task model, lower-priority tasks cannot take steps during T_r's execution. These facts imply that (5.1) can only be falsified by the execution of line 38 by T_r or some higher-priority task. To complete the proof, we show that the execution of line 38 by a higher-priority transaction τ' can falsify (5.1) only if τ' modifies block b_i.

Suppose that a higher-priority transaction τ' executes after τ accesses or modifies block b_i for the first time during m and before τ executes $r.38$ during m, and that τ' does not modify block b_i. Let T_q be the task that invokes τ'. We consider the following two possibilities.

Case 1: If τ' does not access or modify block b_i, then $BANK[b_i]$ — and hence (5.1) — is not affected by the execution of $q.38$ by τ'.

Case 2: If τ' executes a failed update, then τ' does not modify $BANK[b_i]$ and hence cannot falsify (5.1). If τ' executes a successful update that accesses but does not modify block b_i, then we have the following. Consider the execution of the loop at lines 33-37 by τ' when $q.j = l \wedge q.blklist[l] = b_i$ holds, where $l \leq q.numblks$. Because the execution of the *Tr_Read* operation on a block b_i cannot establish $q.dirty[b_i] = 2$, it follows that $q.35$ establishes

$q.newval[l] = q.curr[b_i]$ and that τ' fails the test at $q.36$ during the l^{th} iteration of the loop. By Lemma 5.4, and the fact that $q.oldval[l] = *q.addrlist[l]$ holds before the execution of $q.38$ (because τ' executes a successful update), it follows that $q.oldval[l] = q.newval[l]$ holds before the execution of $q.38$. Therefore, the execution of $q.38$ does not modify $BANK[b_i]$, and hence does not falsify (5.1). □

Lemma 5.7: *Suppose that a transaction invoked by task T_r modifies block b_i for the first time during a successful update by calling procedure Tr_Write when $r.numblks = l'$ holds and that the procedure call establishes* (5.2)*. Then,* (5.2) *can only be falsified by the execution of line* 38 *during a successful update by T_r or some higher-priority task that modifies block b_i.*

Proof: Similar to the proof of Lemma 5.6. □

Lemma 5.8: *Suppose that task T_r invokes a transaction τ that accesses or modifies blocks $b_1, \ldots, b_k$ in the MEM array. Then, τ succeeds the MWCAS operation at $r.38$ iff no other transaction modifies any block b_i, where $1 \leq i \leq k$, after τ accesses or modifies b_i and before τ executes $r.38$.*

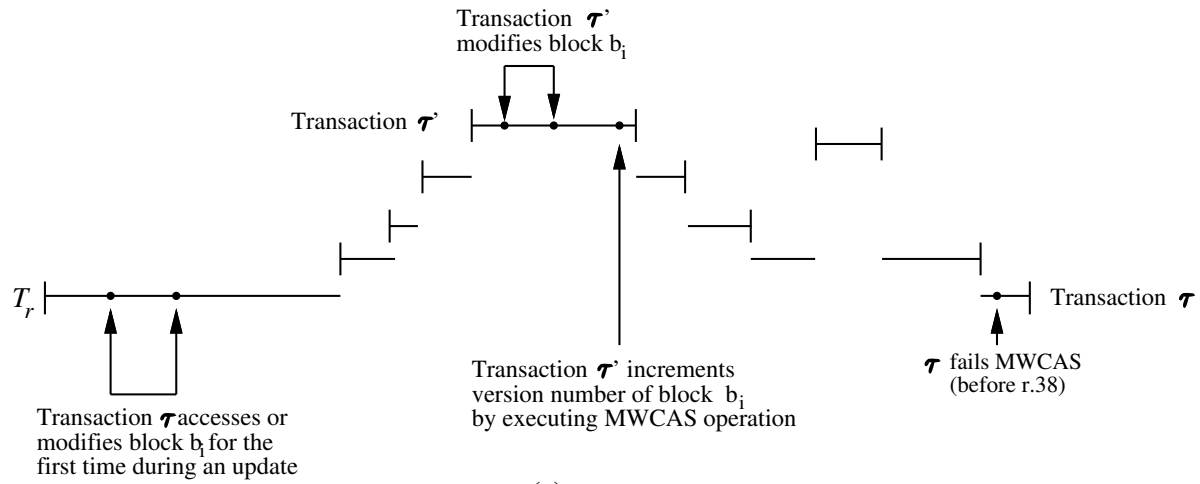
Proof: Suppose that T_r invokes a transaction τ that accesses or modifies blocks b_1,...,b_k in the *MEM* array. Observe that the *Tr_Read* and *Tr_Write* procedures store the address of any block pointer accessed by τ exactly once in the *addrlist* array. (This ensures that the address list parameter passed to the MWCAS procedure (line 38) satisfies Requirement 4.1 in Chapter 4, which states that no address in the address list parameter may be duplicated.)

First, we show that if τ successfully executes $r.38$ then no other transaction modifies any of the blocks accessed or modified by τ. If, before the execution of $r.38$ by τ, some

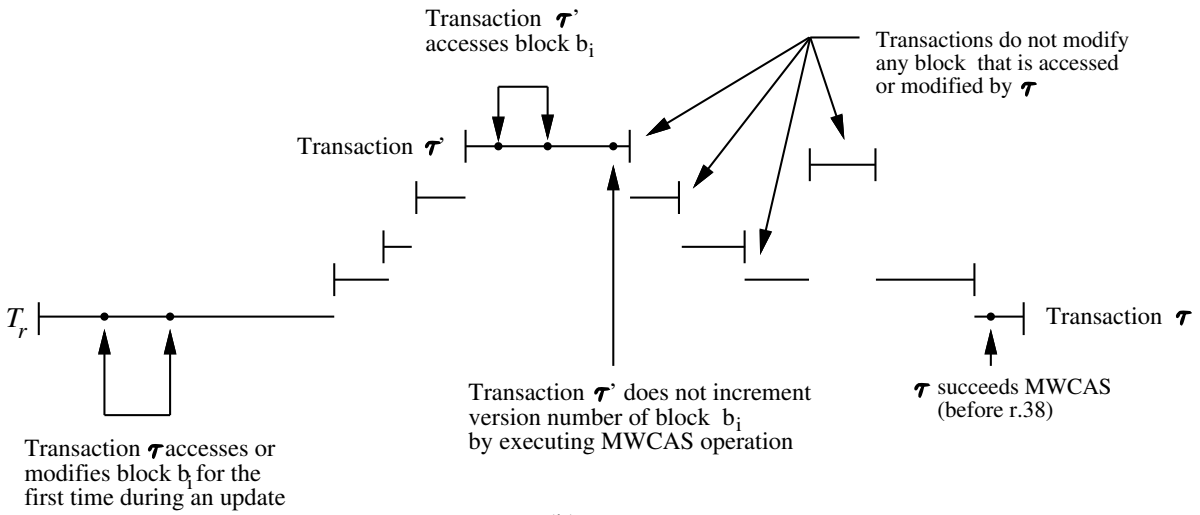
transaction τ' of a higher-priority task T_q modifies some block b_i that is also accessed or modified by τ, then τ' increments the *ver* field of $BANK[b_i]$ before it completes, ensuring that τ fails the MWCAS at $r.38$ when it resumes execution. (Refer to Figure 5.4(a).) Thus, if τ succeeds the MWCAS at $r.38$, then no other transaction modifies any block accessed (modified) by τ, after τ reads from (writes to) that block for the first time.

To complete the proof, we need to show that if no other transaction modifies any block accessed or modified by τ, after τ accesses or modifies that block for the first time during an update and before τ executes $r.38$ during that update, then τ succeeds the MWCAS operation at $r.38$. Consider any block b_i accessed or modified during the execution of an update m by transaction τ. By Lemmas 5.4 and 5.5, either (5.1) or (5.2) is established when τ accesses or modifies block b_i for the first time during m. By Lemmas 5.6 and 5.7, and the fact that higher-priority tasks do not modify block b_i, this implies that (5.1) or (5.2) holds for block b_i before the execution of $r.38$. Thus, all old values match when τ executes the MWCAS operation at $r.38$, and τ succeeds the MWCAS operation at $r.38$. □

The space complexity of the shared variables in the algorithm in Figure 5.2 is $O(B + NCS)$, and the space complexity of the private variables for each of the N tasks is $O(B+C)$. Thus the overall space complexity of the algorithm is $O(NB+NCS)$. As shown in Chapter 4, the time complexity of executing one MWCAS operation that accesses W words is $O(W)$. Because each operation is assumed to modify at most C blocks, line 21 is executed at most C times, where each execution of line 21 requires $O(S)$ running time. Furthermore, a transaction can potentially access all B blocks and perform a MWCAS operation on B words. From these observations, and from Lemma 5.8, we have the following theorem.



(a)



(b)

Figure 5.4: Proof of Lemma 5.8.

Theorem 5.1: In a uniprocessor real-time system, lock-free transactions can be implemented with $O(NB + NCS)$ space complexity and $O(CS + B)$ contention-free time complexity, where N denotes the number of tasks that access the *MEM* array, B denotes the size of the *BANK* array, C denotes the maximum number of blocks modified by any transaction, and S denotes the block size in the *MEM* array. □

Chapter 6

A Comparative Study of Object-Sharing Schemes

In this chapter, we compare the performance of lock-free objects to that of conventional lock-based schemes and wait-free schemes. First, we provide a theoretical comparison of lock-free and lock-based object sharing based on scheduling conditions. This comparison is only somewhat accurate because it is based on the assumption that access costs of all objects are identical. Then, we present a set of experiments that compare the different object-sharing schemes by simulating the execution of randomly-generated sets of periodic tasks that access shared objects. This is followed by a presentation of typical worst-case execution times of operations on various lock-free and lock-based objects measured from actual implementations. Finally, we present a set of experiments that evaluate the performance of the different object sharing schemes in an actual application — a desktop videoconferencing system.

6.1 Formal Comparison

The formal comparison presented in this section is based upon the scheduling conditions presented in Sections 3.3 and 3.4, and scheduling conditions for lock-based schemes found in the literature [44, 69]. In deriving the scheduling conditions in Sections 3.3 and 3.4, we assume that the execution of the retry-loops of the different objects in the system are identical. We also assume that tasks do not invoke multi-object operations. In this section, in order to more easily compare the object sharing schemes, we assume that all accesses to lock-based objects require r units of time, and that tasks do not perform any nested object calls. Thus, the computation time c_i of a task T_i can be written as $c_i = u_i + m_i \cdot t_{acc}$, where u_i is the computation time not involving accesses to shared objects, m_i is the number of shared object accesses by T_i, and t_{acc} is the maximum computation time for any object access, i.e., s for lock-free objects and r for lock-based objects. (Recall that c_i is the computation time of T_i when it is the only task executing on the processor, i.e., it does not including blocking terms associated with priority inversions in the lock-based case or interference costs in the lock-free case.)

6.1.1 Static-Priority Scheduling

We begin by comparing the overhead of lock-free object sharing under RM scheduling with the overhead of the lock-based priority ceiling protocol (PCP) [69]. When tasks synchronize by locking, a higher-priority job can be blocked by a lower-priority job that accesses a common object; the maximum blocking time is called the *blocking factor*. Under the PCP, the worst-case blocking time equals the time required to execute the longest crit-

ical section. Since we do not consider nested critical sections, the blocking factor equals r, the time to execute a single critical section. We denote the schedulability condition for periodic tasks using the PCP by the predicate *sched_PCP*, which on the basis of the analysis in [69], is defined as follows.

$$sched_PCP \equiv \langle \forall i \ \exists t : 0 < t \leq p_i : r + \sum_{j=1}^{i} \left\lceil \frac{t}{p_j} \right\rceil (u_j + m_j \cdot r) \leq t \rangle$$

In the above equation, the first term on the left-hand side represents the blocking factor. In the second term, $u_j + m_j \cdot r$ represents the computation time of task T_j. The expression on the left-hand side represents the maximum demand due to T_i and higher-priority tasks in a interval of length t.

We now derive conditions under which lock-free objects are guaranteed to perform at least as well as lock-based objects under the PCP. Consider the following derivation.

$$\langle \forall j : j \leq i : (m_j + 1) \cdot s \leq m_j \cdot r \rangle \ \wedge \ sched_PCP$$

$$\{\text{Substituting } (m_j + 1) \cdot s \text{ for } m_j \cdot r \text{ in } sched_PCP\}$$

$$\Rightarrow \ \langle \forall i \ \exists t : 0 < t \leq p_i : \sum_{j=1}^{i} \left\lceil \frac{t}{p_j} \right\rceil (u_j + m_j \cdot s) + \sum_{j=1}^{i} \left\lceil \frac{t}{p_j} \right\rceil \cdot s + r \leq t \rangle \qquad (6.1)$$

$$\Rightarrow \ \langle \forall i \ \exists t : 0 < t \leq p_i : \sum_{j=1}^{i} \left\lceil \frac{t}{p_j} \right\rceil (u_j + m_j \cdot s) + \sum_{j=1}^{i-1} \left\lceil \frac{t-1}{p_j} \right\rceil \cdot s \leq t \rangle \qquad (6.2)$$

Because $c_j = u_j + m_j \cdot s$ in the lock-free case, the last expression in this derivation is equivalent to the scheduling condition of Theorem 3.2. Note that $s \leq \frac{r}{2}$ implies that $\langle \forall j : j \leq i : (m_j + 1) \cdot s \leq m_j \cdot r \rangle$ because, for positive m_j, $\frac{1}{2} \leq \frac{m_j}{m_j+1} < 1$. Thus, if the time taken to execute one iteration of a lock-free retry loop is less than half the time it takes to access a lock-based object under the PCP, then any task set that is schedulable under the PCP is also schedulable when using lock-free objects. This also implies that there are

certain task sets that are schedulable when lock-free objects are used, but not under the PCP.

In deriving (6.2) from (6.1) in the comparison above, we dropped the term r, effectively ignoring the effect of blocking under the PCP. If blocking times are considerable, then lock-free objects would be more competitive than this comparison indicates. It should also be noted that our scheduling analysis is very pessimistic. In reality, a preempted task need not be accessing a shared object, and hence may not necessarily have an interference as we have assumed.

6.1.2 Dynamic-Priority Scheduling

We now compare the overhead of lock-free objects with the dynamic deadline modification (DDM) scheme under EDF scheduling (EDF/DDM) [44], which is a lock-based protocol for dynamic-priority schemes. Under this scheme, tasks are divided into one or more phases. During each phase, a task accesses at most one shared resource. Before a task T_i accesses a shared object S_m, its deadline is modified to the deadline of some task T_j that accesses S_m and that has the smallest deadline of all tasks that access S_m. Upon completing the shared object access, T_i's deadline is restored to its original value. In our comparison, we assume that phases in which some shared object is accessed are r units in length. Under the EDF/DDM scheme, r includes the cost of a system call to modify the task deadline before accessing an object, the cost of performing the shared-object operation, and the cost of a system call to restore the task deadline after an access. Based on the analysis of [44], a sufficient condition for the schedulability of a set of periodic tasks under the EDF/DDM scheme, *sched_DDM*, can be defined as follows.

$$sched_DDM \equiv (\sum_{j=1}^{N} \frac{u_j + m_j \cdot r}{p_j} \leq 1) \wedge$$

$$\langle \forall i, t : P_i < t < p_i : r + \sum_{j=1}^{i-1} \left\lfloor \frac{t-1}{p_j} \right\rfloor \cdot (u_j + m_j \cdot r) \leq t \rangle$$

In the first conjunct of the above equation, the expression in the left-hand side of the inequality represents the total processor utilization due to tasks in the system. The term $u_j + m_j \cdot r$ represents the computation time of T_j. In the second conjunct, the term P_i is defined as the minimum p_j such that T_j shares a common object with T_i. The first term on the left-hand side of the inequality in the second conjunct represents the maximum time T_i can block tasks with smaller periods, and the second term represents the total demand on the processor due to tasks with smaller periods in an interval of length $t-1$. The expression on the left-hand side of the inequality represents the maximum demand that can be placed on the processor during an interval of length t.

We now derive conditions under which lock-free objects are guaranteed to perform at least as well as objects implemented using the DDM scheme. Consider the following derivation.

$$\langle \forall j : (m_j + 1) \cdot s \leq m_j \cdot r \rangle \ \wedge \ sched_DDM \tag{6.3}$$

$$\{\text{By the definition of } sched_DDM\}$$

$$\Rightarrow \quad \langle \forall j : (m_j + 1) \cdot s \leq m_j \cdot r \rangle \ \wedge \ \sum_{j=1}^{N} \frac{u_j + m_j \cdot r}{p_j} \leq 1 \tag{6.4}$$

$$\{\text{Substituting } (m_j + 1) \cdot s \text{ for } m_j \cdot r\}$$

$$\Rightarrow \quad \sum_{j=1}^{N} \frac{u_j + (m_j + 1) \cdot s}{p_j} \leq 1$$

Because $c_j = u_j + m_j \cdot s$ in the lock-free case, the last expression in this derivation is equivalent to the scheduling condition of Theorem 3.5. As noted previously, $s \leq \frac{r}{2}$ implies

$\langle \forall j : (m_j + 1) \cdot s \leq m_j \cdot r \rangle$. Thus, as with the PCP, if the time taken to execute one iteration of a lock-free retry loop is less than half the time it takes to access an object using the DDM scheme, then any task that is schedulable under the EDF/DDM scheme is also schedulable under EDF scheduling using lock-free objects. As mentioned previously, s is likely to be smaller than r for many objects.

In deriving (6.4) from (6.3) in the above comparison, we dropped the second conjunct in *sched_DDM*, effectively ignoring the effect of blocking under the EDF/DDM scheme. If blocking times are considerable, then lock-free objects would perform better than as indicated above.

6.1.3 Wait-Free Objects

Wait-free shared objects differ from lock-free objects in that wait-free objects are required to guarantee that individual tasks are free from starvation. Most wait-free universal constructions ensure termination by requiring each task to "help" every other task to complete any pending object access [36, 37]. To see how this works, consider the lock-free universal construction of Herlihy [37], which is described in Subsection 2.4.3. This construction does not guarantee termination because the *store-conditional* operation of each retry loop iteration may fail. Herlihy extends this construction to be wait-free by requiring each task to "announce" any pending operation by recording it in a shared array. Using this information, each task is able to "help" other tasks with pending operations by performing their operations in addition to its own. If a task is repeatedly unsuccessful in modifying the shared object pointer, then it is eventually helped by another task — in fact, after at most two retry loop iterations.

Note, however, that on a uniprocessor, lower-priority tasks cannot help higher-priority tasks because a higher-priority task does not release the processor until its demand has been fulfilled. Thus, each task only helps lower-priority tasks. Hence, the greater the task priority, the larger the access time. In some sense, the problem of priority inversion still exists, because a medium-priority task will have to wait while a high-priority task helps a low-priority task. On the other hand, when lock-free objects are used, the time to complete an object access decreases with increasing priority. For these reasons, some task sets that are schedulable when using lock-free objects will not be schedulable when using wait-free objects. This is true of the task set evaluated in Section 6.4.

In order to more formally compare lock-free and wait-free objects, let us assume that objects are implemented using Herlihy's universal constructions. First, note that tasks that share wait-free objects can be viewed as independent tasks, i.e., the scheduling conditions derived in [57] and [60] apply. These conditions are the same as those given in Theorems 3.2 and 3.5, respectively, when $s = 0$. The computational cost c_j in this case equals $u_j + m_j \cdot t_{wf}$, where t_{wf} is the worst-case access time of a wait-free object, which occurs when *all* lower-priority tasks with pending operations are helped. If s is the time taken for the retry loop in Herlihy's less-complicated lock-free construction, then we would expect $t_{wf} = c \cdot s$, for some $c \gg 1$. Observe that if $c \geq 2$, then the c_j term in the wait-free case is greater than or equal to $u_j + (m_j + 1) \cdot s$. Note also that $u_j + (m_j + 1) \cdot s$ is at least as large as the terms corresponding to T_j in Theorems 3.2 and 3.5. Thus, if a task set is schedulable using Herlihy's wait-free universal construction, then it is also schedulable using Herlihy's lock-free universal construction.

The conclusion drawn above that lock-free implementations always perform better than wait-free ones may not apply if wait-free objects are implemented using techniques other than a Herlihy-like helping scheme. In fact, Anderson, Ramamurthy, and Jain have shown recently that it is possible to dramatically reduce helping overhead in wait-free implementations for priority-based real-time systems [10]. For such implementations, it may indeed be the case that wait-free implementations of some objects are superior to their lock-free counterparts, although the extent to which this is true is currently unknown.

6.2 Simulation Results

In this section, we present results from simulation experiments conducted to compare lock-free, wait-free, and lock-based object implementations under the RM scheme. These experiments involved randomly generated task sets consisting of 10 tasks and 5 shared objects, obtained by varying four parameters: r/w ratio, cost ratio, conflicts, and nesting level. The *r/w ratio* parameter specifies the fraction of all operations that are read-only. This parameter is of interest because, as explained in Chapter 5, read-only operations do not interfere with each other in lock-free implementations. The *cost ratio* parameter specifies the ratio of the cost of a lock-free (wait-free) object access to that of a lock-based access; e.g., the retry-loop cost of a lock-free object is (on average) twice as expensive as a lock-based access if the cost ratio parameter is 2. The cost of a lock-based operation includes the cost of acquiring and releasing a lock; for lock-based objects, an implementation based on the stack resource policy was assumed [19]. If the *conflicts* parameter is k, then at least one object is accessed by k tasks, and no object is accessed by more than k

tasks. In our experiments, tasks were modeled as a sequence of three phases, of which only the second is an object-access phase. The *nesting level* parameter specifies the number of objects accessed in this phase, and ranges from 1 to 3 (the word "nesting" refers to nested locks in lock-based implementations).

In order to bound the simulation lengths, task periods were randomly selected from a predetermined set of 36 periods. For the set of periods considered, the LCM of the periods was 134,534,400 time units, and the minimum and maximum periods were 8,448 and 1,747,200 time units, respectively. Computation phase costs ranged between 1 and 500 time units, and were randomly generated subject to the constraint that overall utilization is at most one. In all experiments, context switch times were ignored.

Lock-based object access costs were randomly generated assuming a normal distribution with mean and standard deviation of 128 and 20 time units, respectively. The overall cost of each object-access phase depends on both the nesting level and the object access costs. In our experiments, nesting levels 1, 2, and 3 were selected with probability 0.6, 0.25, and 0.15, respectively. We selected this distribution based on our belief that multi-object accesses are less frequent than single-object accesses in practice.

To compare the performance of the different schemes, we calculated the breakdown (computation) utilization of each task set that was generated. The *breakdown utilization* (BU) of a task set is obtained by scaling the cost of task phases, and is defined to be the maximum utilization at which the task set is still schedulable. The total utilization of all computation phases of all tasks at the breakdown point is called the *breakdown computation utilization* (BCU).

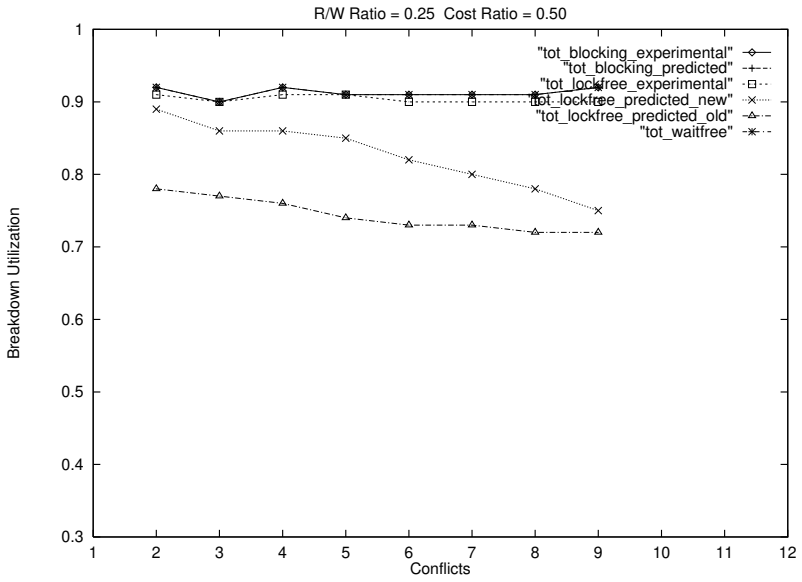
Typical BU and BCU curves resulting from our experiments are shown in Figure 6.1. The r/w ratio and cost ratio in the graphs in Figure 6.1 equals 0.25 and 0.50, respectively. (Corresponding graphs for various r/w ratio and cost ratio values are given in Figures 6.2 through 6.9.) Each curve in these figures was obtained from 4,000 generated task sets. Experimental BU (BCU) values for lock-based and lock-free schemes are given by "blocking_experimental" and "lockfree_experimental", respectively. These values were obtained for each generated task set by checking schedulability in a brute force manner, i.e., by checking to the LCM of the task periods. Predicted BU (BCU) values for lock-based objects are given by "blocking_predicted". Values for this case were obtained by using the scheduling condition given in [69]. BU (BCU) values predicted by the scheduling conditions presented in Section 3.5 and in Section 3.3 are given by "lockfree_predicted_new" and "lockfree_predicted_old", respectively. Observe that the RM scheduling condition presented in Section 3.5 is much tighter than the one presented in Section 3.3. Also, the new condition results in better predictions when there are fewer conflicts and when most operations are read-only. (Refer to the graphs in Figures 6.7 through 6.9.) BU (BCU) values for wait-free objects are given by "waitfree"; these values were obtained by using the RM scheduling condition in [57]. Experimental BCU values are not tabulated for this case because the RM condition in [57] is necessary and sufficient.

Our simulations indicate that only the cost ratio parameter significantly affects relative performance. In examining the effects of various cost ratios, it is best to focus on BCU values. This is because the BU curves include overhead associated with object accesses, and because the experimental BU curves do not show much variation. (A high BU

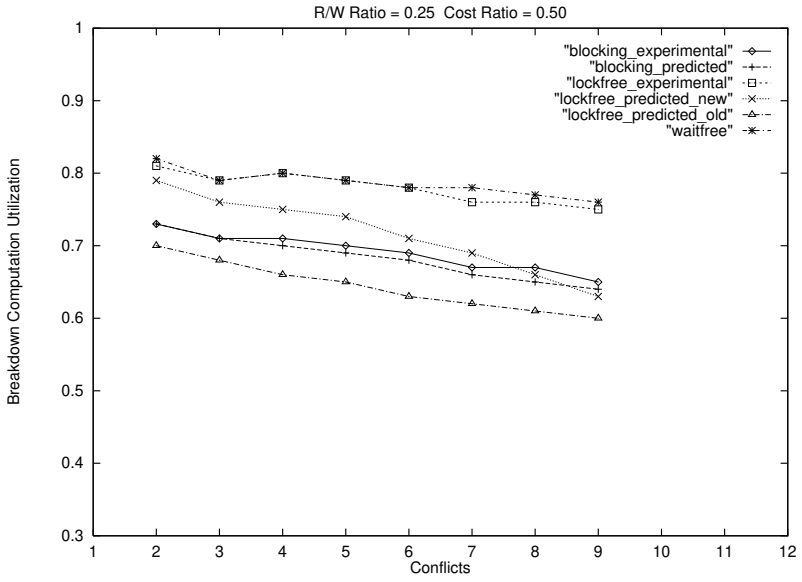
curve can be misleading, because much of the utilization accounted for in the BU values may be due to object sharing overhead; an inefficient object sharing scheme may give rise to high BU values solely because of this overhead.) The BCU curves in Figures 6.4 through 6.6 indicate that when the r/w ratio parameter equals 0.5, lock-free objects perform better than lock-based schemes when the cost ratio is less than one (Figure 6.4(b)), slightly worse than lock-based schemes when the cost ratio equals one (Figure 6.5(b)), and worse than lock-based schemes when the cost ratio is greater than one (Figure 6.6(b)).

The main conclusion to be drawn from these experiments is that, when lock-free loop costs are (on average) less than corresponding lock-based access costs, lock-free implementations are likely to perform better. As indicated by the figures in Table 6.2, lock-free implementations of common objects like queues, stacks, and linked lists are likely to be more efficient than lock-based implementations. On the other hand, lock-based implementations of more complex objects like balanced trees are likely to be more efficient than lock-free ones. Wait-free implementations perform better than their lock-free counterparts in all situations when access costs are identical. However, in practice, wait-free operation costs are typically much higher than corresponding lock-free costs, due to the additional algorithmic overhead required to ensure wait-freedom.

Although our results indicate that the r/w ratio parameter is not very significant, in practice, a high r/w ratio will result in a low cost ratio for lock-free objects. This is because, for many objects, read-only operations do not require copying and are therefore less expensive than read-write operations. Thus, lock-free implementations may be preferable if most operations are read-only. In our experiments, we did not account for the fact that

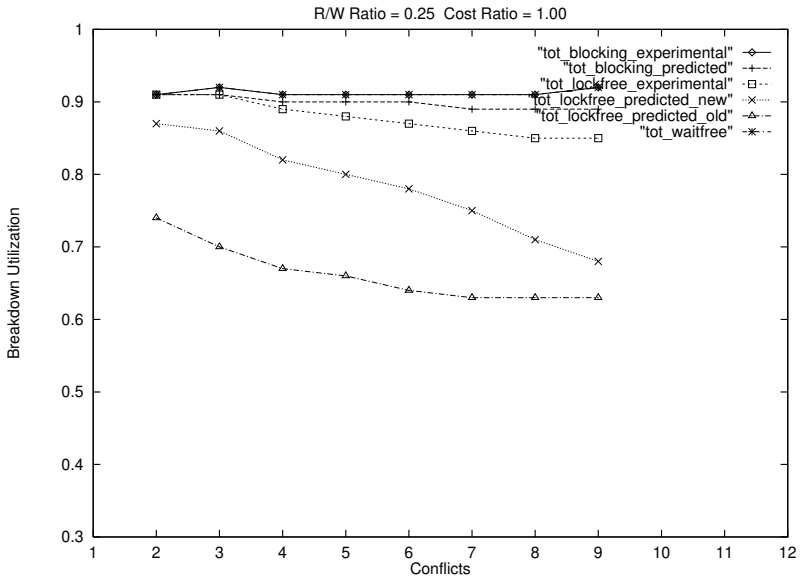


(a)

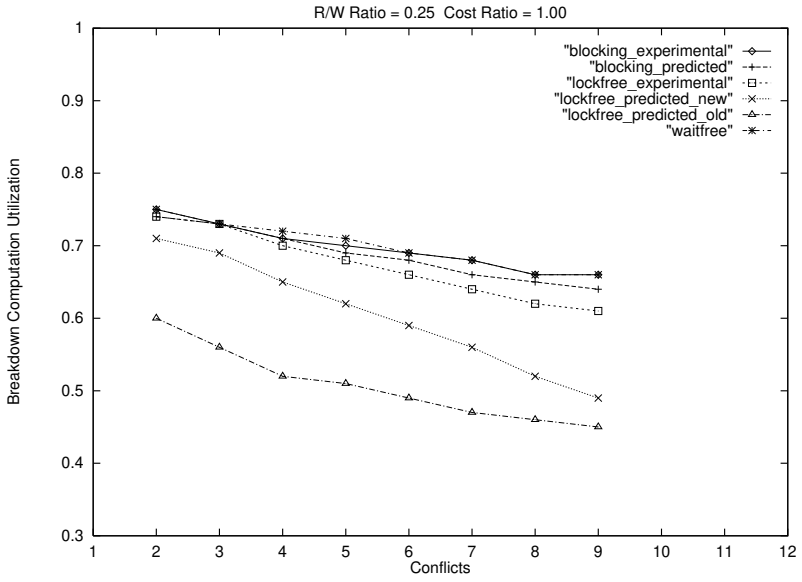


(b)

Figure 6.1: The r/w ratio and cost ratio parameters are 0.25 and 0.50, respectively, for these figures. (a) Breakdown utilizations. (b) Corresponding breakdown computation utilizations.

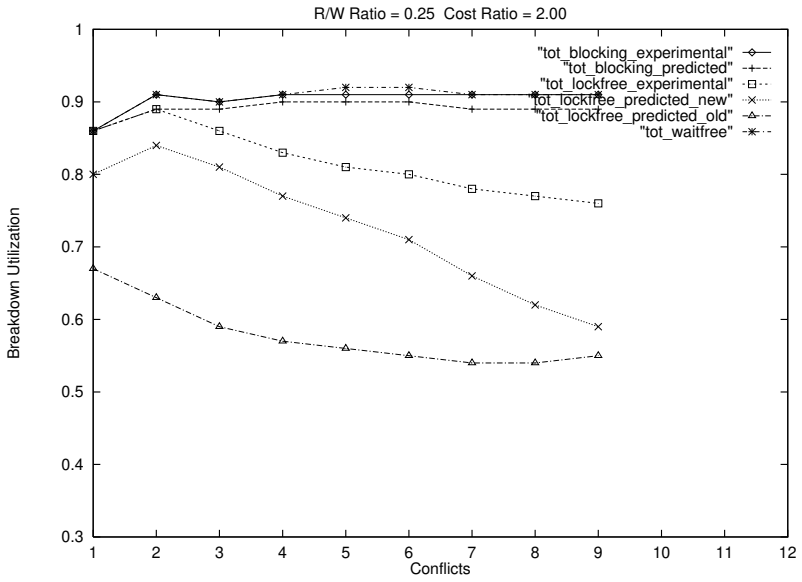


(a)

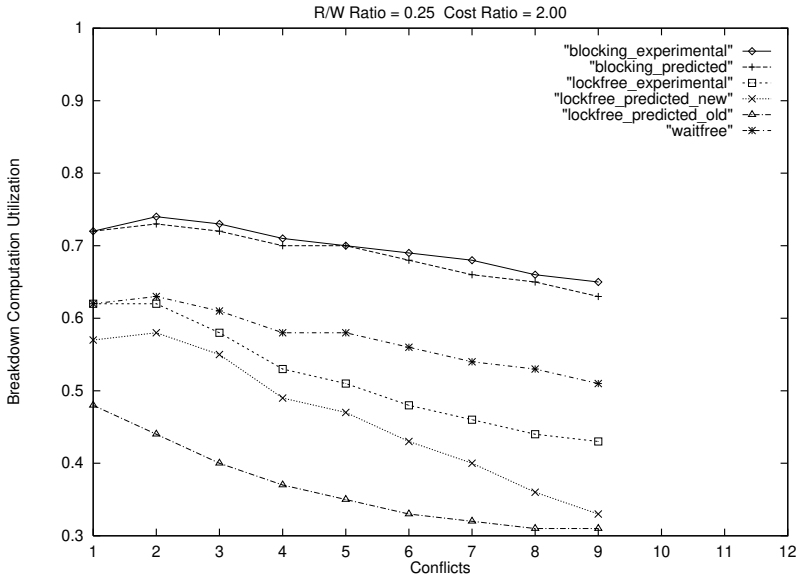


(b)

Figure 6.2: The r/w ratio and cost ratio parameters are 0.25 and 1.00, respectively, for these figures. (a) Breakdown utilizations. (b) Corresponding breakdown computation utilizations.

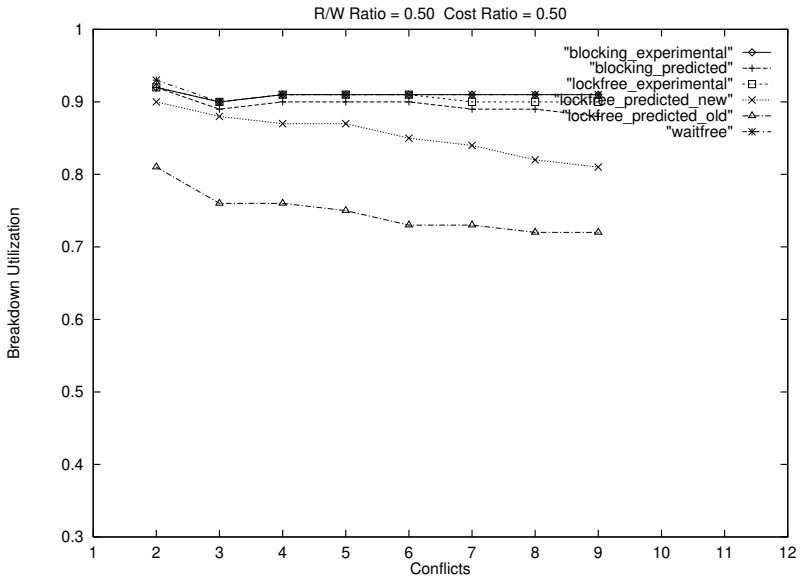


(a)

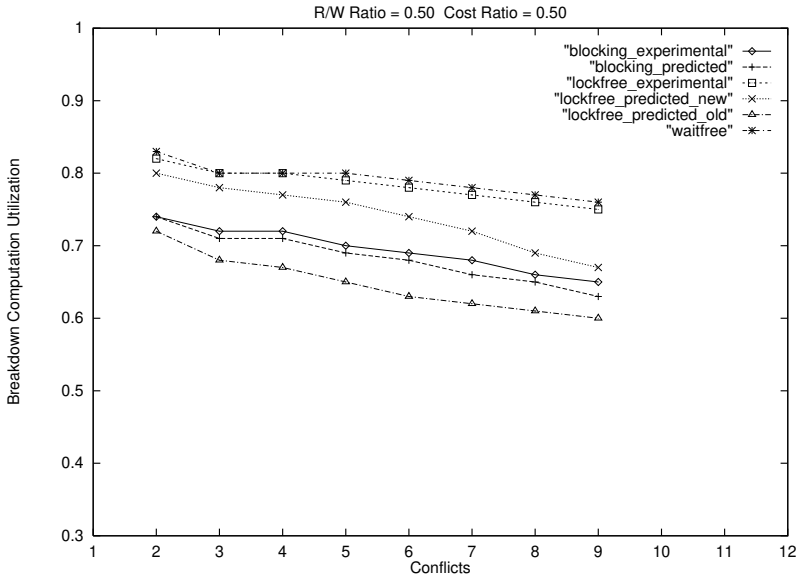


(b)

Figure 6.3: The r/w ratio and cost ratio parameters are 0.25 and 2.00, respectively, for these figures. (a) Breakdown utilizations. (b) Corresponding breakdown computation utilizations.

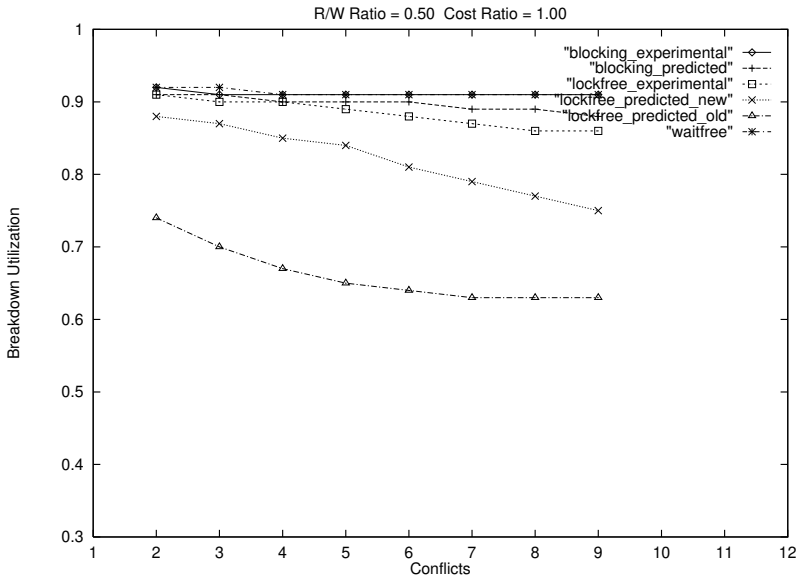


(a)

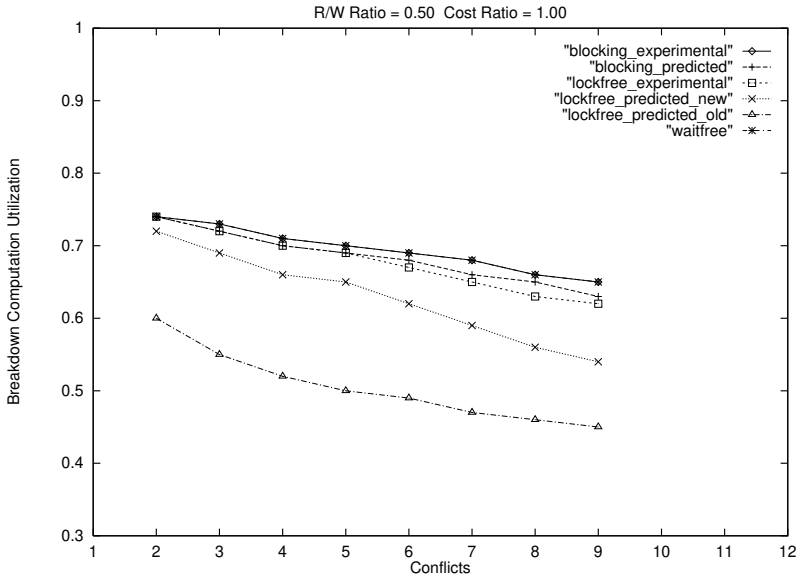


(b)

Figure 6.4: The r/w ratio and cost ratio parameters are 0.50 and 0.50, respectively, for these figures. (a) Breakdown utilizations. (b) Corresponding breakdown computation utilizations.

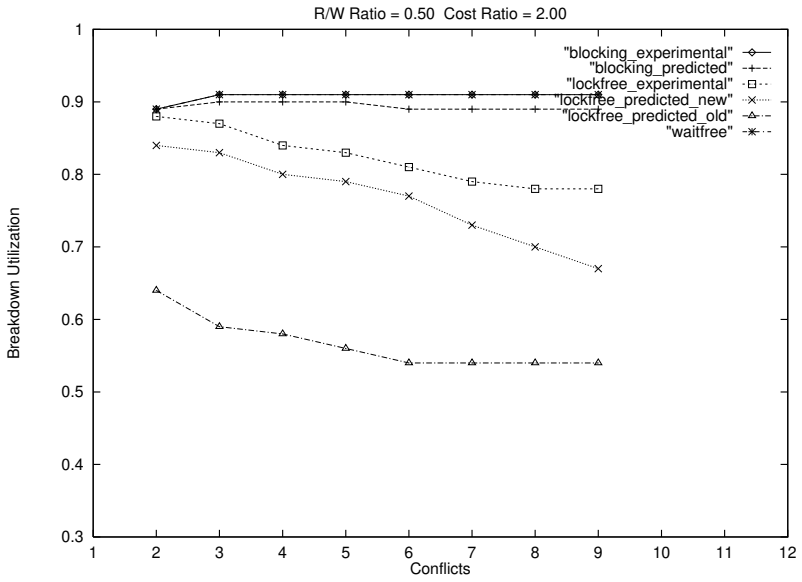


(a)

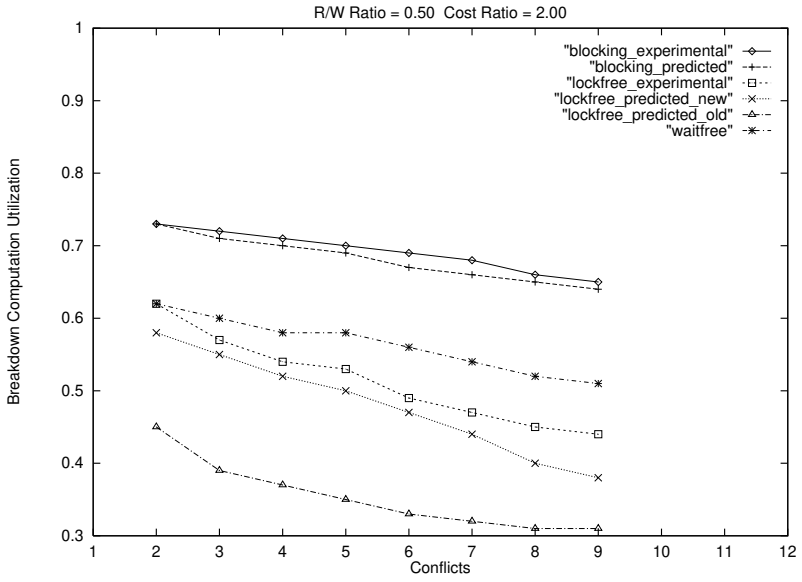


(b)

Figure 6.5: The r/w ratio and cost ratio parameters are 0.50 and 1.00, respectively, for these figures. (a) Breakdown utilizations. (b) Corresponding breakdown computation utilizations.

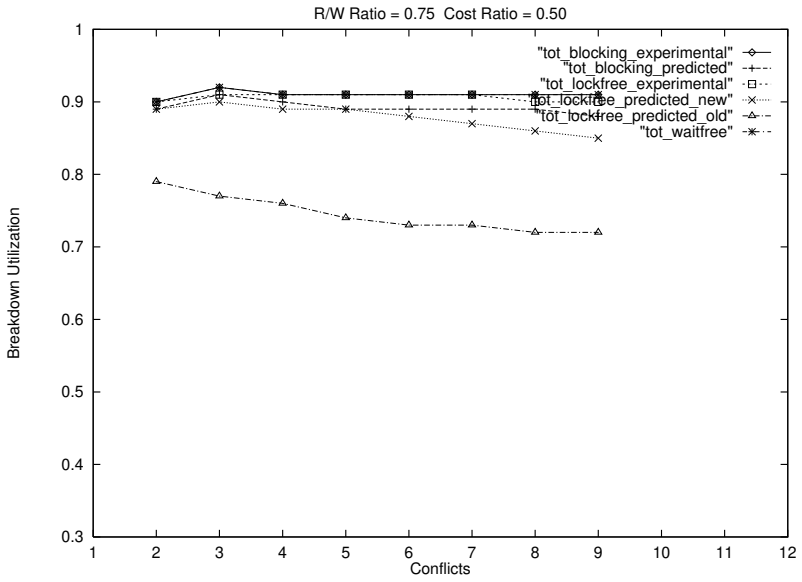


(a)

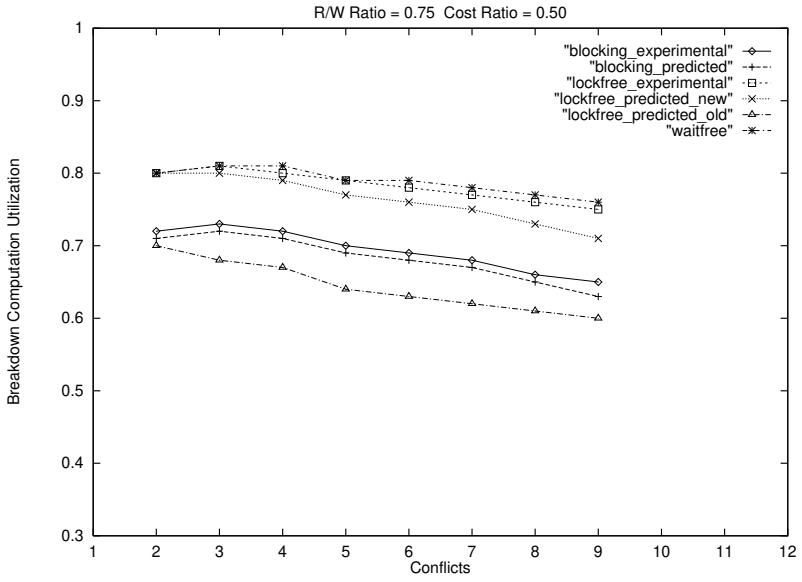


(b)

Figure 6.6: The r/w ratio and cost ratio parameters are 0.50 and 2.00, respectively, for these figures. (a) Breakdown utilizations. (b) Corresponding breakdown computation utilizations.

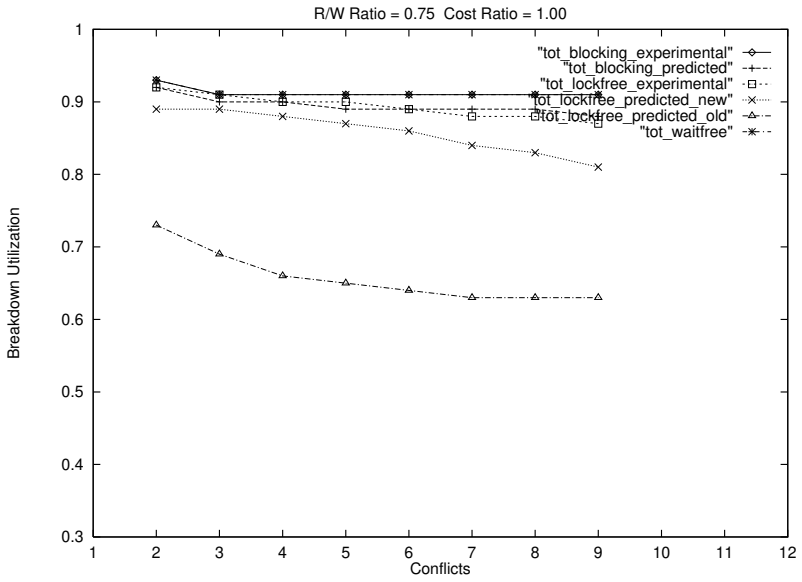


(a)

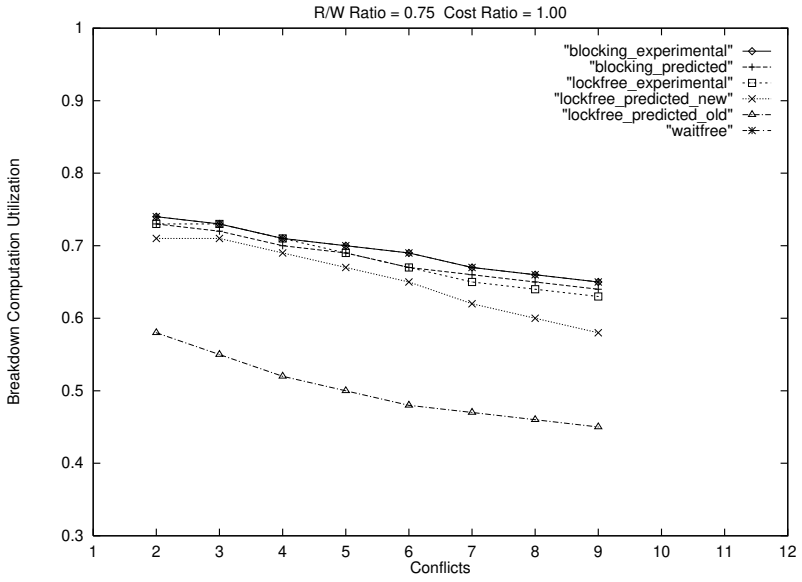


(b)

Figure 6.7: The r/w ratio and cost ratio parameters are 0.75 and 0.50, respectively, for these figures. (a) Breakdown utilizations. (b) Corresponding breakdown computation utilizations.

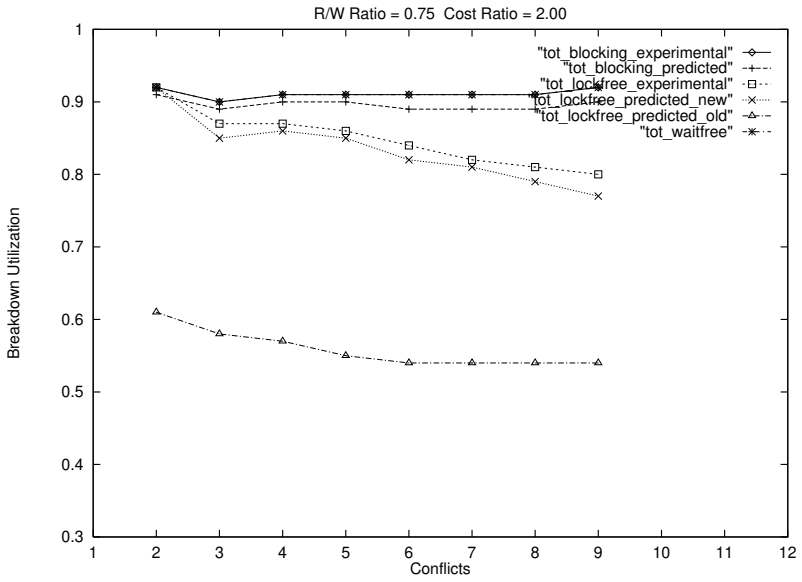


(a)

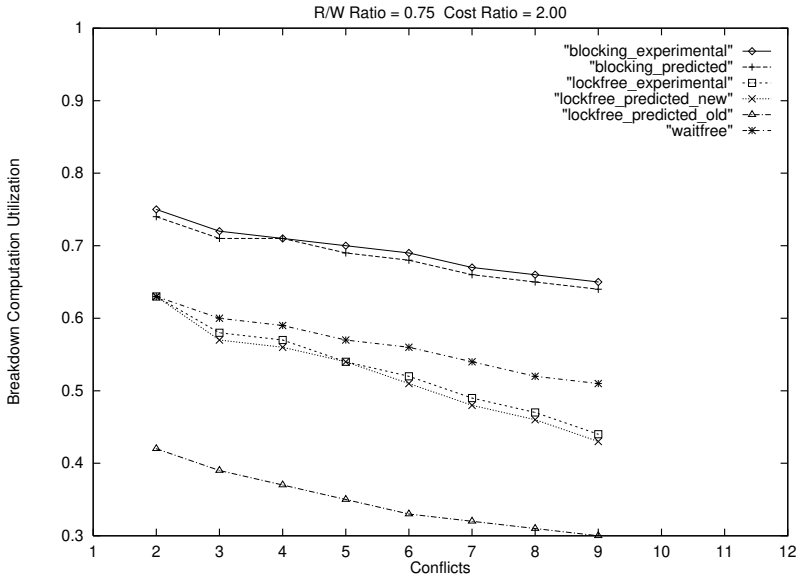


(b)

Figure 6.8: The r/w ratio and cost ratio parameters are 0.75 and 1.00, respectively, for these figures. (a) Breakdown utilizations. (b) Corresponding breakdown computation utilizations.



(a)



(b)

Figure 6.9: The r/w ratio and cost ratio parameters are 0.75 and 2.00, respectively, for these figures. (a) Breakdown utilizations. (b) Corresponding breakdown computation utilizations.

read-only operations are usually more efficient than updates in lock-free implementations. In addition, the larger conflicts parameter values considered (which resulted in less accurate predications for lock-free) may not be reflective of what one would find in practice. In fact, as explained in the following section, in many practical real-time systems, write-write conflicts cannot occur between concurrently executing tasks [31], implying that the conflicts parameter is likely to be small. Furthermore, because most shared objects in such systems are single-writer multi-reader read/write buffers and because most of the operations in such a system are likely to be read-only, the average cost of a lock-free access is likely to be smaller than the average cost of a lock-based access. (As indicated by the timing measurements in the following section, this is true even if objects are implemented under our transactional framework.)

6.3 Timing Measurements

Since the above comparison between lock-free objects and the PCP rests on r and s values, it is instructive to take a closer look at what these values are actually comprised of in practice. s is the cost of one lock-free retry loop. For most simple data structures like read/write buffers, queues, linked lists, and stacks, s is often simply the cost of a simple, straight-line code sequence. For more complex data structures like skew heaps [76] — a data structure similar to a balanced heap — and balanced trees, a lock-free implementation would be more complicated, and corresponding s values might be relatively high. In some applications, s might also include the time taken to copy the implemented object if a universal construction were being used, or the time taken to simulate a synchronization

primitive such as CAS2 if such primitives were not provided in hardware. Under the PCP, r includes the cost of a system call to modify the calling task's priority before accessing an object, the cost of actually performing the shared-object operation, and the cost of a system call to restore the calling task's priority after an access.

What are typical values of s and r? A performance comparison of various lock-free objects is given by Massalin in [62]. Massalin reports that, given hardware support for primitives like compare-and-swap, s varies from 1.3 microseconds for a counter to 3.3 microseconds for a circular queue. In the absence of hardware support, such primitives can be simulated by a trap, adding an additional 4.2 microseconds. Massalin's conclusions are based on experiments run on a 25 MHz, one-wait-state memory, cold-cache 68030 CPU. In contrast, lock-based implementations fared much worse in a recent performance comparison of commercial real-time operating systems run on a 25 MHz, zero-wait-state memory 80386 CPU [30]. In this comparison, the implementation of semaphores on LynxOS took 154.4 microseconds to lock and unlock a semaphore in the worst case. The corresponding figure for POSIX mutex-style semaphores was 243.6 microseconds. These figures cannot be regarded as definitive because s values can vary widely depending on the implemented object. (However, they do give some indication as to the added overhead when operating-system-based locking mechanisms are used.) To provide a better comparison of these schemes, we provide s and r values for many common objects measured from actual object implementations. The objects were implemented under the YARTOS kernel developed at UNC [47, 77], on a 66-MHz, 80486-based IBM PC. To ensure a fair comparison, the lock and unlock calls in the YARTOS kernel were optimized as much as possible.

Because the 80486 does not support strong primitives other than the memory-to-memory move instruction, we simulated CAS, CAS2, and MWCAS in software. Because the cost of lock-free access is directly affected by the cost of implementing strong primitives, we compared the cost of different strong-primitive implementations to determine the most efficient scheme of implementing strong primitives. The schemes that we considered implemented strong primitives either as short non-preemptible code fragments or as user-level procedure calls using the algorithms presented in Chapter 4.

Typical execution times of various implementations of CAS, CAS2, and MWCAS primitives are given in Table 6.1. In the table, strong primitive implementations denoted by "Disable Interrupts" were implemented as non-preemptible code fragments by disabling interrupts while executing the primitive. Implementations based on the algorithms in Chapter 4 are identified by their section numbers. The figures in Table 6.1 indicate that primitives implemented as short non-interruptible code fragments are the least expensive. Of the remaining software-based implementations, the CAS implementations based on the *move* primitives is the most efficient. These figures also indicate that, if the number of tasks that access a shared object is three or less, then the load/store implementation CAS in Section 4.4 is practical. (Recall that the running time of the implementation in Section 4.4 is linear in the number of tasks accessing that word.) Also, MWCAS implementations that are implemented as non-interruptible code fragments significantly outperform the software-based implementation present in Section 4.6. Based on the above results, we implemented all strong primitives in our experiments as non-preemptible code fragments.

It should be noted that the above timing measurements do not imply that prim-

Primitive	Implementation	WCET (in μseconds)
CAS	Disable Interrupts	Read/CAS
2 tasks		1/3
CAS	Section 4.4	Read/CAS
2 tasks		14/23
3 tasks		16/28
4 tasks		17/31
5 tasks		19/34
CAS	Section 4.5	Read/CAS
		1.5/5
CAS2	Disable Interrupts	Read/CAS
2 tasks		1/5
MWCAS	Disable Interrupts	Read/CAS
1 word		1/4
3 words		1/6
5 words		1/8.5
7 words		1/10
9 words		1/12
10 words		1/13
20 words		1/24.5
MWCAS	Section 4.6	Read/MWCAS
1 word		3/17.5
3 words		3/35
5 words		3/52
7 words		3/66
9 words		3/81
10 words		3/88
20 words		3/169

Table 6.1: Worst-case execution times (WCET) of various implementations of strong primitives. Times are given in μseconds.

Shared Object	Worst-Case Execution Time (WCET)					
	Lock-Based (r)		Lock-Free (s)			
			Object Specific		Transaction-based	
Read/Write Buffer	Update	Read	Update	Read	Update	Read
	37	35	5	4	24	22.5
Stack	Push/Pop	Peek	Push/Pop	Peek	Push/Pop	Peek
	36/36	35	7/7	4	24/23.5	22
Queue	Enq./Deq.	Peek	Enq./Deq.	Peek	Enq./Deq.	Peek
	37/37	35	7/7	5	22.5/19	16
Skew Heap	Enq./Deq.	Peek	Enq./Deq.	Peek	Enq./Deq.	Peek
5 nodes	54.5/48	35	N/A	N/A	56/51	24
9 nodes	62/49.5	35	N/A	N/A	73/68	24
17 nodes	69/55	35	N/A	N/A	95/85.5	24
33 nodes	76/58	35	N/A	N/A	121.5/104	24
65 nodes	83/60.5	35	N/A	N/A	134/126	24
Linked List	Ins./Del.	Search	Ins./Del.	Search	Ins./Del.	Search
10 nodes	60/53.5	51	31/23.5	19	57/52	33.5
20 nodes	70.5/61	60	44.5/40	30	69.5/67	48
30 nodes	77/70.5	64.5	59.5/52	43	84/81	58
40 nodes	93/79	72	74/64.5	53	96/95	68
50 nodes	97/87	80.5	78/85.5	74	110/105	75.5

Table 6.2: Worst-case cost of operations on commonly used data structures. Times are given in μseconds.

itives implemented as non-preemptible code fragments are always less expensive than software-based implementations. For example, if instructions to disable and enable interrupts in a system are privileged instructions, i.e., they cannot be executed by user-level code, then, in order to disable interrupts, a user task must execute a kernel call to switch from user mode to kernel mode — and crossing the user/kernel boundary can sometimes be expensive. In such situations, implementations of primitives based on the algorithms in Chapter 4 may be more appropriate.

In Table 6.2, we present typical s and r values for three different implementa-

tions of read/write buffers, stacks, queues, skew heaps, and linked lists. Under each object implementation scheme, figures in the column on the left are execution times for update operations on the object, whereas figures in the column on the right indicate the execution times for read-only operations. The lock-based versions of these data structures were implemented under the EDF/DDM scheme. For objects other than skew heaps,[1] the object-specific lock-free implementations were based on implementations presented elsewhere. In particular, the read/write buffer, stack, and queue implementations are from [62], and the linked list implementations are from [32]. Transaction-based lock-free object implementations are based on the transactional framework described in Chapter 5. However, unlike the implementation described in Chapter 5, the implementation used in our experiments did not employ the MWCAS implementation described in Section 4.6. Instead, for efficiency reasons, we implemented the MWCAS primitive as a non-interruptible code fragment. Also, the size of the blocks in our implementation of the transactional framework is 32 bytes, and no transaction performed a MWCAS operation on more than 8 words.

The figures in Table 6.2 indicate that object-specific lock-free implementations of read/write buffers, stacks, queues, and linked lists significantly outperform their lock-based counterparts. This is due to the fact that, when implementations of these objects are lock-based, the total cost of executing a lock/unlock sequence is much greater than the time taken to perform an operation on the shared object. In contrast, the cost of executing a strong primitive is approximately equal to the time taken to perform a operation on the lock-free implementations of these objects.

[1] Because all known implementations of skew heaps are based on universal constructions, we did not measure execution times for object-specific implementations of skew heaps.

Of the implementations based on our transactional framework, read/write buffers, queues, and stacks are less expensive than their lock-based counterparts — however, they are more expensive than their object-specific lock-free counterparts. The figures in Table 6.2 indicate that lock-free implementations of data structures such as skew heaps and balanced trees are likely to be more expensive than their lock-based counterparts. These figures also indicate that read-only operations under either lock-free implementations are less expensive than their lock-based counterparts. Read-only operations are less expensive under our transactional framework because a significant part of the overhead entailed during a write operation is associated with making a local copy of blocks modified by an operation.

It should be noted that the above execution times and the formal analysis in Section 6.1 do not necessarily imply that implementations based on lock-free transactions will always perform worse than lock-based scheme. This is because of two main reasons. First, although our transactional framework is an important step in making lock-free object sharing viable in real-time systems, much more research needs to be directed towards a more efficient transactional framework. Second, for a given task set, the choice of one object sharing scheme over another depends entirely on the schedulability of the task set under these schemes. For example, in many practical real-time systems, write-write conflicts cannot occur between concurrently executing tasks [31]. In such systems, most shared objects are single-writer multi-reader read/write buffers, and most of the operations in such a system are likely to be read-only. Because read-only operations do not interfere with each other under the transactional framework, the worst-case number of interferences in such a system is likely to be low. Our transactional framework is likely to outperform lock-based

schemes in such situations because, as shown in Table 6.2, read-only operations under our transactional framework are more efficient than their lock-based counterparts. Furthermore, objects implemented under our transactional framework can be optimized for certain task sets. For example, if most of the objects in a system are single-writer objects and if the writer task has higher-priority than all the reader tasks, then the *Tr_Write* operation for the writer task can be optimized to eliminate copying overhead entirely. Also, future work on developing efficient lock-free constructions is necessary to further improve the performance of lock-free objects.

6.4 Experiments on a Videoconferencing System

In this section, we provide empirical evidence that lock-free objects are superior to their lock-based counterparts. This evidence comes from a set of experimental comparisons performed using a real-time desktop videoconferencing system implemented at UNC [47]. We modified this system to support lock-free shared objects implemented under both DM and EDF scheduling, semaphores implemented using the PCP under DM scheduling, and semaphores implemented under EDF/DDM scheduling. We also considered wait-free shared objects implemented under both DM and EDF scheduling. The formal analysis for each synchronization scheme was applied to determine whether it was theoretically possible to ensure that no deadlines would be missed. We then executed the system using each synchronization scheme under a variety of loading conditions, and compared the actual performance to that predicted by the formal analysis. In all cases, the formal analysis predicted the actual behavior of the system. Moreover, our lock-free synchronization schemes frequently

led to higher levels of sustainable system utilization than was possible with lock-based synchronization. Also, our experiments confirm that, for the objects considered, lock-free implementations are superior to wait-free implementations based on Herlihy-like helping schemes, for real-time computing on uniprocessors. The following subsection describes the videoconferencing system in more detail.

6.4.1 Experimental Setup

The videoconferencing system considered in our investigations acquires analog audio and video samples on a workstation and then digitizes, compresses, and transmits the samples over a local-area network to a second workstation where they are decompressed and displayed. Here we consider only the portion of the system responsible for the acquisition, compression, and network transmission of media samples by the sending workstation.

Abstractly, the tasks on the sending workstation are organized as a software pipeline. Communication between stages is realized through a queue of media samples that is shared using a simple producer/consumer protocol. These queues must support a *get_length* operation in addition to *enqueue* and *dequeue*, which slightly complicates their implementation. Queues of shared media samples exist between the digitizing task and the compression task and between the compression task and the network transmission tasks. The real-time constraints on the operation of the pipeline require media samples to flow through the pipeline in a predictable manner. These media samples arrive sporadically and are manipulated by a set of sporadic tasks. Each task must process arriving media samples before a prespecified deadline that does not coincide with that task's period, and no media samples may be lost due to buffer overflows.

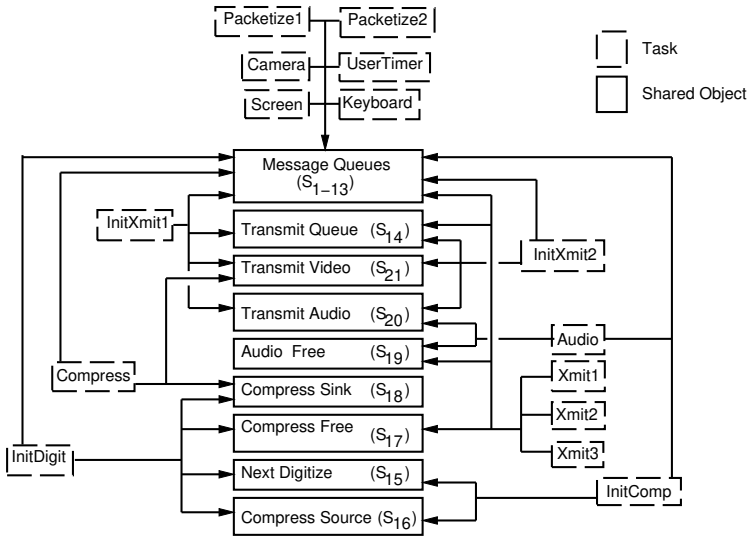


Figure 6.10: Tasks and shared queues in the videoconferencing system.

A comprehensive view of the tasks (dashed boxes) and shared queues (solid boxes) on the sending workstation is given in Figure 6.10. In this figure, an arrow is directed from each task to each of the shared objects it accesses. The message queues ($S_1 - S_{13}$) are used for inter-task communication. For our purposes, it suffices to consider the tasks in Figure 6.10 to be an abstract set of tasks — details regarding the function of each task, and how the tasks interact are not important to us. For a more detailed description of this system, we refer the interested reader to [77].

We evaluated the performance of the system when the shared queues were implemented using lock-free algorithms, wait-free algorithms, and lock-based techniques. We implemented lock-free queues by using the shared queue implementation given in [62] (modified to support the *get_length* operation), and wait-free queues by using the wait-free universal construction given in [37]. Massalin's queue implementation requires CAS (needed for the dequeue operation) and CAS2 (needed for the enqueue operation), and Herlihy's

construction requires *load-linked* and *store-conditional*. We implemented these primitives by short kernel calls; interrupts were disabled for the duration of these calls.

We found that the videoconferencing task set was not schedulable, in any experiment, when the shared queues were implemented using Herlihy's wait-free universal construction. This is due to the high overhead of helping, as discussed in Section 6.1.3. In contrast, our lock-free implementations required very little overhead, with interferences occurring only rarely. For example, in ten executions of the system, only 363 interferences occurred in 415,229 enqueue operations. We also found that multiple interferences of a single operation *never* occurred. In the following two subsections, we discuss results of experiments that were conducted to compare lock-free and lock-based schemes under static- and dynamic-priority scheduling.

6.4.2 Static-Priority Scheduling

In this subsection, we discuss the results of experiments that compare the overhead of lock-free objects to lock-based objects implemented using the PCP. In both cases scheduling was performed using the DM scheduling algorithm [58].

Qualitatively, when queue synchronization was achieved using semaphores, approximately seven media samples were lost in the pipeline every second due to buffer overflow. In contrast, no media samples were lost when lock-free objects were used.

This result is predicted by the formal analysis of the system, which we now present. The model we consider consists of a set of $N = 15$ sporadic tasks, $M = 21$ shared objects, and $Q = 12$ periodic and sporadic interrupt handlers. The i^{th} task in the system is given by the tuple $\langle c_i, p_i, l_i, a_i \rangle$, where c_i and p_i have the usual meanings, l_i is the relative deadline

Table 6.3: Task characteristics. Times are given in μseconds.

Task Name	T_i	Cost [DM] c_i		Cost [EDF] c_i		Period p_i	Deadline l_i	WC Response†	
		PCP	LF	DDM	LF			t_i^*	t_i^{**}
`InitXmit1`	T_1	579	459	687	649	33333	6705	4743	4623
`Xmit1`	T_2	147	147	147	147	45603	6705	4890	4807
`Xmit2`	T_3	147	147	147	147	45603	6705	5037	4991
`Xmit3`	T_4	147	147	147	147	45603	6705	5184	5175
`Compress`	T_5	602	528	669	624	9573	8000	5786	5740
`Camera`	T_6	396	396	416	416	15746	15000	6182	6173
`Audio`	T_7	1017	953	1024	966	15746	15000	7199	7163
`InitDigit`	T_8	1110	1046	1137	1096	31492	15000	8309	8246
`InitComp`	T_9	1332	746	1069	640	31492	15000	10243	9029
`InitXmit2`	T_{10}	710	604	821	982	33333	19850	11287	10235
`Packetize1`	T_{11}	8315	8315	8315	8315	40842	33333	22651	21943
`Packetize2`	T_{12}	8315	8315	8315	8315	40842	33333	N/A	30860
`UserTimer`	T_{13}	126	122	102	137	54538	54538	37872	31385
`Keyboard`	T_{14}	580	549	637	589	490853	490853	39054	37065
`Screen`	T_{15}	142	71	148	78	1963379	1963379	39196	37173

† Worst-case (WC) response times apply to DM scheduling.

of T_i, and a_i is the set of shared objects accessed by T_i. We assume that tasks are indexed in the order of nondecreasing deadlines. The i^{th} interrupt handler is given by the tuple $\langle e_i, v_i \rangle$, where e_i is the execution time of the handler and v_i is the minimum time between interrupts. Interrupt handlers are executed in a first-come-first-served manner and always have priority over application tasks. The periods, relative deadlines, and the execution times of the tasks in our formal model are shown in Table 6.3. The periods and execution times of the interrupt handlers are shown in Table 6.4.

The formal model of the experimental system can be analyzed by using the scheduling condition given in Theorem 3.2 when lock-free objects are used, and that given in [57] when lock-based objects are used. Note, however, that these conditions do not consider the

Table 6.4: Interrupt handler execution times and periods. Times are given in μseconds.

I_i	I_1	I_2	I_3	I_{4-5}	I_{6-7}	I_{8-9}	I_{10-12}
e_i	254	333	333	183	389	389	389
v_i	54925	16666	10493	15492	45666	42603	47666

cost of handling interrupts, and hence cannot be used directly. Fortunately, this problem can be overcome by using techniques derived in [46]. The idea is to derive an expression that bounds the demand due to interrupt handlers in any given interval, and to then account for this demand in the scheduling conditions of Theorem 3.2 and [57].

Informally, we account for the cost of interrupt handlers as follows (see [46] for a more formal version of this argument). First, we define the term $F(t)$ to be the cost of handling interrupts over an interval of length t. In order to derive a bound on $F(t)$, consider I_i, the i^{th} interrupt in the system, and consider an interval $[t_0, t_0 + t)$ of length t, where $t_0 \geq 0$. I_i occurs at most $\lceil t/v_i \rceil$ times in that interval, and requires e_i units of processor time for every occurrence. Hence, the total demand placed on the processor by I_i in the interval is at most $\lceil t/v_i \rceil \cdot e_i$. It then follows that the total demand due to all the interrupt handlers, $F(t)$, is bounded by the summation on the right-hand side of the following inequality.

$$F(t) \leq \sum_{j=1}^{Q} \left\lceil \frac{t}{v_j} \right\rceil \cdot e_j \tag{6.5}$$

Using (6.5), we can obtain a schedulability condition when the tasks synchronize using lock-based objects and the PCP. This involves modifying the condition presented in [57] to account for the demand placed by interrupt handlers, as given by (6.5). The resulting

condition is as follows.

$$\langle \forall i \ \exists t : 0 < t \leq l_i : \ r + \sum_{j=1}^{i} \left\lceil \frac{t}{p_j} \right\rceil \cdot c_j + \sum_{j=1}^{Q} \left\lceil \frac{t}{v_j} \right\rceil \cdot e_j \leq t \rangle \tag{6.6}$$

The first term in (6.6) gives the worst-case blocking time in the system, and the second term gives the demand placed by T_i and higher-priority tasks on the processor. The third term gives the maximum demand placed by all interrupt handlers in the same interval.

In our system, r equals 151. In Table 6.3, t_i^* gives a value of t in the interval $(0, l_i]$ that satisfies (6.6). The analysis shows that the task `Packetize2` is not schedulable. This task copies compressed media sample buffers to the network adapter. When `Packetize2` does not meet its deadline, the sender drops (never transmits) some of the media samples. This analysis explains why some media samples were lost when the system was run using lock-based objects and the PCP.

We now consider the system when the tasks synchronize using lock-free objects. A schedulability condition for this case is obtained by modifying the condition of Theorem 3.2 to account for the demand placed by interrupt handlers, as given by (6.5). The resulting condition is as follows.

$$\langle \forall i \ \exists t : 0 < t \leq l_i : \ \sum_{j=1}^{i} \left\lceil \frac{t}{p_j} \right\rceil \cdot c_j + \sum_{j=1}^{i-1} \left\lceil \frac{t-1}{p_j} \right\rceil \cdot s + \sum_{j=1}^{Q} \left\lceil \frac{t}{v_j} \right\rceil \cdot e_j \leq t \rangle \tag{6.7}$$

In the above equation, the first term gives the demand placed on the processor due to T_i and higher-priority tasks. The second term gives the additional demand due to interferences, and the third term gives the maximum demand placed on the processor by interrupt handlers in the same interval. In our system, s equals 37. (Observe that s is less than $r/2$ in our system.) In Table 6.3, t_i^{**} gives a value of t in the interval $(0, l_i]$ that

satisfies (6.7). It can be seen that all tasks are schedulable when lock-free objects are used. This is confirmed by the fact that no media samples are lost during the execution of the system.

6.4.3 Dynamic-Priority Scheduling

In this subsection, we discuss the results of experiments that compare the overhead of lock-free objects under the EDF scheme to lock-based objects under the EDF/DDM scheme. Our experiments showed that the task set of Tables 6.3 and 6.4 is schedulable under both schemes. This is result is predicted by the formal analysis of the system, which we now present.

Our analysis of the EDF/DDM scheme is based upon the following scheduling condition, which is proved in [77].

$$
\begin{aligned}
&(\textstyle\sum_{j=1}^{N} c_j/p_j + \sum_{j=1}^{Q} e_j/v_j \leq 1) \wedge \\
&\langle \forall t : p_1 \leq t \leq B_{DDM} : \textstyle\sum_{j=1}^{N} \left\lfloor \frac{t-l_j+p_j}{p_j} \right\rfloor \cdot c_j + \sum_{j=1}^{Q} \left\lceil \frac{t}{v_j} \right\rceil \cdot e_j \leq t \rangle \wedge \\
&\langle \forall i, t : p_1 < t < p_i : r + \textstyle\sum_{j=1}^{i-1} \left\lfloor \frac{t-1-l_j+p_j}{p_j} \right\rfloor \cdot c_j + \sum_{j=1}^{Q} \left\lceil \frac{t}{v_j} \right\rceil \cdot e_j \leq t \rangle
\end{aligned}
$$

In the second conjunct above, $B_{DDM} \equiv (\sum_{j=1}^{N} c_j + \sum_{j=1}^{Q} e_j)/(1 - \sum_{j=1}^{N} c_j/p_j + \sum_{j=1}^{Q} e_j/v_j)$. The first and third conjuncts above correspond to the two conjuncts of *sched_DDM* given in Section 6.1.2. However, these conjuncts have been modified to account for the overhead of interrupt handlers, and to reflect the fact that in the videoconferencing system deadlines and job releases do not necessarily coincide. In the left-hand side of the inequality in the second conjunct, the first and second summation terms give the maximum demand due to the tasks and interrupt handlers, respectively, in an interval of length t. The

right-hand side of the inequality gives the available processor time in that interval. It can be shown that this scheduling condition holds for the abstract task set defined in Tables 6.3 and 6.4.

Our analysis of the system when lock-free objects are used is based upon the scheduling condition below. This condition is based upon the conditions given in Theorems 3.5 and 3.4 and the techniques given in [46] for accounting for the overhead of interrupt handlers.

$$(\sum_{j=1}^{N}(c_j + s)/p_j + \sum_{j=1}^{Q} e_j/v_j \leq 1) \ \wedge$$

$$\langle \forall t : t \in [p_1, B_{LF}] : \sum_{j=1}^{N} \left(\left\lfloor \frac{t - l_j + p_j}{p_j} \right\rfloor \cdot c_j + \left\lfloor \frac{t - 1 - l_j + p_j}{p_j} \right\rfloor \cdot s \right) + \sum_{j=1}^{Q} \left\lceil \frac{t}{v_j} \right\rceil \cdot e_j \leq t \rangle$$

In this expression, $B_{LF} \equiv (\sum_{j=1}^{N}(c_j + s) + \sum_{j=1}^{Q} e_j)/(1 - \sum_{j=1}^{N}(c_j + s)/p_j + \sum_{j=1}^{Q} e_j/v_j)$. The first conjunct above is the condition of Theorem 3.5 augmented to include utilization due to interrupt handlers. The second conjunct follows from Theorem 3.4 and results of [46]. The three summation terms in this conjunct give the maximum demand due to the tasks, interferences, and interrupt handlers, respectively, in an interval of length t. The right-hand side of the stated inequality gives the available processor time in that interval. It can be shown that this scheduling condition holds for the abstract task set defined in Tables 6.3 and 6.4.

In order to more precisely compare lock-free objects with objects implemented under the EDF/DDM scheme, we introduced a dummy task T_{16}, given by the tuple $\langle c_{16}$, 2342664, 2342664, $\{S_{17-21}\}\rangle$, to increase the processor utilization of the system. This dummy task consists of a bounded loop. During each loop iteration, the task performs some busy work and accesses some shared objects. The demand on the processor was

varied by modifying the number of loop iterations executed by the dummy task.

Our experiments showed that processor utilization was higher under the EDF/DDM scheme for all task loads. Under the EDF/DDM scheme, tasks started to miss deadlines when the dummy task performed approximately 3500 loop iterations. The processor utilization corresponding to this load was close to 99.4%. For the same load, the processor utilization was only 94% when lock-free objects were used. Processor utilization is higher under EDF/DDM for the same load due to the overhead of modifying task deadlines for each shared object access. This confirms the prediction of Section 6.1.2 that lock-free objects often require less overhead than object implemented under the EDF/DDM scheme. In our experiments, when lock-free objects were used, tasks started missing deadlines when processor utilization was about 99.1%.

Chapter 7

Conclusions

In this chapter, we first give a brief summary of the results presented in this dissertation, and discuss some of the lessons learned from this research.

7.1 Summary

In Chapter 3, we presented two sets of scheduling conditions for the DM and the EDF scheduling schemes. These scheduling conditions are an essential step towards utilizing lock-free objects in hard real-time systems. The first set of conditions that we developed was based on the assumption that retry-loop costs for all lock-free objects are relatively uniform. These conditions do not perform well if the retry-loop costs of the objects in the system can vary widely. We removed this restricting assumption in the second set of conditions presented in Section 3.5. The scheduling conditions developed in this section are based on integer linear programming, and are therefore more expensive to evaluate than the first set of conditions. However, as indicated by the experimental results in Chapter 6,

the second set of conditions are much more accurate than the first set of conditions.

The constructions presented in Chapter 4 exploit the fact that scheduling in real-time systems is usually priority-based. Specifically, we presented a solution to the consensus problem, for a uniprocessor real-time system, that uses only load and store instructions. Our consensus protocol implies that Herlihy's hierarchy collapses in uniprocessor hard real-time systems. In this chapter, we also presented two software-based implementations of CAS. The first implementation uses only load and store instructions, and the second uses the memory to memory move instruction. This chapter concluded with an implementation of MWCAS primitive that uses single-word CAS primitives. The running time of our MWCAS implementation is optimal — the running time is $O(W)$ for a MWCAS on W words.

In Chapter 5, we presented a framework for implementing lock-free transactions and lock-free multi-object operations. This framework allows the programmer to implement lock-free objects without having to prove the correctness of those implementations. However, the programmer is required to use the *Tr_Read* (*Tr_Write*) procedures to read from (write to) shared memory. When an object is implemented in this fashion, the resulting code closely resembles the sequential code for the object. Under this lock-free framework, operations that access different blocks of the array *MEM* can execute concurrently, without fear of interferences — these operations may even be on the same object. For example, an *enqueue* operation on a shared queue and a *dequeue* operation on that queue can execute concurrently because the head and the tail of the queue can be located in different blocks. Also, read-only operations on objects implemented under our lock-free framework do not interfere with each other. Objects implemented under our lock-free framework are likely to

perform well in systems where objects are accessed by a single writer and multiple readers, and where most of the operations are read-only.

In Chapter 6, we presented experimental results that validate the claims made earlier on in the dissertation. Specifically, we presented a formal comparison of lock-free and lock-based objects which indicates that the performance of these schemes hinge on the values of s and r, where s denotes the cost of executing a lock-free retry-loop once and r denotes the cost of a lock-based access — including the time to lock and unlock the object. Specifically, if $s \leq r/2$, then any task that is schedulable under lock-based protocols is also schedulable under lock-free schemes. Then, we presented simulation results that compare the performance of lock-free and lock-based objects. The results of this simulation indicate that choosing a scheme for implementing an object must be based on the average object access cost. If most of the object accesses are read-only, then it is likely that the average object-access cost will be smaller when lock-free schemes are used. We then presented s and r values for many different objects. The execution times that we presented indicate that $s \leq r/2$ holds for simple objects such as read/write buffers, stacks, queues, and linked lists, but not for more complicated objects such as skew heaps and balanced trees. These timing measurements indicate that objects such as queues, stacks and read/write buffers should always be implemented using object-specific lock-free implementations. To demonstrate the utility of lock-free objects in practical applications, we also presented results of some experiments on a videoconferencing system developed at UNC. These experiments indicate that lock-free queue implementations are superior to their lock-based counterparts.

7.2 Conclusions and Future Work

In the course of our work, we learned several lessons about implementing shared objects in uniprocessor hard real-time systems.

- Hardware support for implementing strong synchronization primitives is not essential (but preferable) in uniprocessor hard real-time systems. In such situations, implementing strong primitives as short non-interruptible code fragments is the most efficient method of implementing primitives such as CAS, CAS2, and MWCAS, if the cost of disabling/enabling interrupts is very small. Otherwise, implementations based on the algorithms in Chapter 4 are more appropriate. If the move instruction is supported by the underlying architecture, then an object that supports Read and CAS operations can be implemented very efficiently. It remains to be seen whether a MWCAS implementation based on the move primitive is inexpensive enough to be practical in uniprocessor hard real-time systems.

- Wait-free implementations that are based on Herlihy's universal constructions are not competitive with lock-free schemes. In order to be more competitive with lock-free schemes, the helping overhead has to be significantly reduced. In fact, on uniprocessor systems, it is possible to implement wait-free objects that are competitive with lock-based and lock-free schemes using a technique called *incremental helping* [10]. This technique exploits the real-time task model to ensure that a task helps at most one lower-priority task complete its operation before it performs its own operation. The scheduling conditions for task sets that employ this technique is very similar to the conditions for conventional lock-based schemes. Each task entails a "helping factor"

(the amount of time spent helping lower-priority tasks complete their operation) akin to blocking factors encountered in conventional lock-based systems.

- The work presented in this dissertation focuses only on object implementations in uniprocessor real-time systems. Although lock-free objects perform well in such systems, they likely to perform poorly in multiprocessor systems because, in such a system, a lock-free operation on one processor can interfere with an operation on another processor — accurately determining the worst-case wasted computation in this case is very difficult. Lock-based schemes also perform very poorly in multiprocessor systems because the blocking factors entailed by a task can be significant [69]. As demonstrated in [6], wait-free object implementations that use the cyclic helping technique, in conjunction with incremental helping, can significantly reduce helping overhead on a multiprocessor real-time system. Under the *cyclic-helping scheme*, processors in a system are thought of as if they were part of a logical ring. Tasks executing on these processors are helped through the use of a "help counter", which cycles around the ring. To advance the help counter from processor R to the next processor on the ring, a process must first help the currently announced process on processor R. In order to perform an operation, a process does the following: it first repeatedly advances the help counter until any pending announced (lower-priority) operation on its own processor has been completed; it then announces its own operation; it then repeatedly advances the help counter until is own operation has been completed.

  Using techniques such as cyclic-helping and incremental-helping, the amount of helping overhead entailed by a task is proportional to the number of processors in the

system. In contrast helping overhead under conventional wait-free schemes is proportional to the number of tasks in the system. Another important issue that warrants further research is the development of an efficient transactional framework for object sharing in multiprocessor systems.

- A CAS operation on an object implemented using the algorithm presented in Section 4.4 requires execution time proportional to the number of tasks accessing that object. It remains to be seen whether CAS can be implemented in constant-time using only load and store instructions. Also, it remains to be seen whether information about a task's priority can be used to simplify the constructions presented in Chapter 4. Another research issue to be tackled is the development of a MWCAS implementation based on the move primitive. Such a primitive is likely to be more efficient than the MWCAS implementation presented in Section 4.6.

- Most of the focus in Chapter 5 is on designing shared objects using our transactional framework. Extensive work is required to determine the extent to which our transactional framework can be adapted in order to implement real-time databases. Specifically, techniques are required for performing logging and recovery in databases implemented using our transactional framework. Furthermore, by adapting our transactional framework to use information about transaction priorities, the amount of wasted computation due to retries can be reduced, potentially — such techniques are likely to result in a significant reduction of missed transaction deadlines [34].

- Although we have developed some simple rules for choosing an object sharing scheme, the right choice of a scheme for implementing an object in an application is entirely

dependent on the task set of the application. Further research is required on the issue of determining appropriate object-sharing schemes for a given application.

- The conclusions drawn from the experimental results presented in Section 6.4 cannot be extended to all real-time systems in general because all shared objects in the experimental videoconferencing system were queues. To better evaluate these schemes, applications other than videoconferencing must be considered that employ both simple objects, like queues, stacks, etc., and more complicated objects, such as skew heaps and balanced trees.

Bibliography

[1] Y. Afek, H. Attiya, D.Dolev, E. Gafni, M. Merritt, and N. Shavit. Atomic snapshots of shared memory. In *Proceedings of the Ninth Annual ACM Symposium on Principles of Distributed Computing*, pages 1–14. ACM, August 1990.

[2] Y. Afek, D. Dauber, and D. Touitou. Wait-free made fast (extended abstract). In *Proceedings of the 27th Annual ACM Symposium on Theory of Computing*, pages 538–547. ACM, 1995.

[3] J. Anderson. Composite registers. In *Proceedings of the Ninth Annual ACM Symposium on Principles of Distributed Computing*, pages 15–30. ACM, August 1990.

[4] J. Anderson and B. Grošelj. Beyond atomic registers: Bounded wait-free implementations of nontrivial objects. *Science of Computer Programming*, 19(3):197–237, December 1992.

[5] J. Anderson and M. Moir. Towards a necessary and sufficient condition for wait-free synchronization. In *Proceedings of the Seventh International Workshop on Distributed Algorithms*, pages 39–53, September 1993.

[6] J. H. Anderson, R. Jain, and S. Ramamurthy. Wait-free object-sharing schemes for real-time uniprocessors and multiprocessors. In *Proceedings of the Eighteenth IEEE Real-Time Systems Symposium (to appear)*, 1997.

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

[8] J. H. Anderson and M. Moir. Universal constructions for multi-object operations. In *Proceedings of the 14th Annual ACM Symposium on Principles of Distributed Computing*, pages 184–193. ACM, August 1995.

[9] J. H. Anderson and S. Ramamurthy. A framework for implementing objects and scheduling tasks in lock-free real-time systems. In *Proceedings of the 17th IEEE Real-Time Systems Symposium*, pages 92–105. IEEE, December 1996.

[10] J. H. Anderson, S. Ramamurthy, and R. Jain. Implementing wait-free objects in priority-based systems. In *Proceedings of the 16th Annual ACM Symposium on Principles of Distributed Computing*, pages 229–238. ACM, August 1997.

[11] J. H. Anderson, S. Ramamurthy, and K. Jeffay. Real-time computing with lock-free objects. Technical Report TR95-021, Department of Computer Science, University of North Carolina, Chapel Hill, North Carolina, 1995.

[12] J. H. Anderson, S. Ramamurthy, and K. Jeffay. Real-time computing with lock-free objects. *ACM Transactions on Comp. Sys.*, 15(6):388–395, May 1997.

[13] J. H. Anderson, S. Ramamurthy, M. Moir, and K. Jeffay. Lock-free transactions for real-time systems. In *Proceedings of the First International Workshop on Real-Time Databases*, pages 107–114, March 1996.

[14] J. H. Anderson, S. Ramamurthy, M. Moir, and K. Jeffay. Lock-free transactions for real-time systems. In *Real-Time Databases: Issues and Applications*. Kluwer Academic Publishers, Amsterdam, 1997.

[15] R. Anderson and H. Woll. Wait-free parallel algorithms for the union-find problem. In *Proceedings of the 23rd Annual ACM Symposium on Theory of Computing*, pages 370–380. ACM, August 1991.

[16] J. Aspnes and M. Herlihy. Wait-free data structures in the asynchronous pram model. In *Proceedings of the Second Annual ACM Symposium on Parallel Architectures and Algorithms*, pages 340–349, June 1990.

[17] H. Attiya and O. Rachman. Atomic snapshots in $o(n \log n)$ operations. In *Proceedings of the Twelveth Annual ACM Symposium on Principles of Distributed Computing*, pages 29–40. ACM, August 1993.

[18] N. C. Audsley, A. Burns, M. F. Richardson, and A. J. Wellings. Hard real-time scheduling: The deadline monotonic approach. In *Proceedings of the Eighth Workshop on Real-Time Operating Systems and Software*, pages 133–138, May 1991.

[19] T. P. Baker. Stack-based scheduling of real-time processes. *Real-Time Systems*, 3(1):67–99, March 1991.

[20] G. Barnes. A method for implementing lock-free shared data structures. In *Proceedings of the Fifth Annual ACM Symposium on Parallel Architectures and Algorithms*, pages 261–270. ACM, 1993.

[21] S. K. Baruah, R. R. Howell, and L. E. Rosier. Feasibility problems for recurring tasks on one processor. *Theoretical Computer Science*, 118:3–20, 1993.

[22] B. Bershad. Practical considerations for non-blocking concurrent objects. In *Proceedings of the Thirteenth International Conference on Distributed Computing Systems*, pages 264–274, May 1993.

[23] B. Bloom. Constructing two-writer atomic registers. *IEEE Trans. on Computer Systems*, 37(12):1506–1514, December 1988.

[24] J. Burns and G. Peterson. Constructing multi-reader atomic values from non-atomic values. In *Proceedings of the Eighth Annual ACM Symposium on Principles of Distributed Computing*, pages 222–231, August 1987.

[25] C.Dwork and O. Waarts. Simple and efficient bounded and concurrent timestamping or bounded concurrent timestamp systems are comprehensible! In *Proceedings of the 24th Annual ACM Symposium on Theory of Computing*, pages 655–666. ACM, April 1992.

[26] M. I. Chen and K. J. Lin. Dynamic priority ceiling: A concurrency control protocol for real time systems. *Real-Time Systems*, 2(1):325–346, 1990.

[27] M. Dertouzos. Control robotics: The procedural control of physical processors. In *Proceedings of the IFIP Congress*, pages 807–813, 1974.

[28] D. Dolev and N.Shavit. Bounded concurrent timestamp systems are constructible! In *Proceedings of the 21st Annual ACM Symposium on Theory of Computing*, pages 78–92. ACM, April 1989.

[29] S. R. Faulk and D. L. Parnas. On synchronization in hard real-time systems. *Comm. of the ACM*, 31(3):275–287, 1988.

[30] B. O. Gallmeister and C. Lanier. Early experience with posix 1003.4 and posix 1003.4a. In *Proceedings of the Twelveth IEEE Real-Time Systems Symposium*, pages 190–198. IEEE, December 1991.

[31] M. H. Graham. How to get serializability for real-time transactions without having to pay for it. In *Proceedings of the Fourteenth IEEE Real-Time Systems Symposium*, pages 56–65, 1993.

[32] M. Greenwald and D. Cheriton. The synergy between non-blocking synchronization and operating system structure. In *Proceedings of the USENIX Association Second Symposium on Operating Systems Design and Implementation*, pages 123–136, 1996.

[33] M. G. Harbour, M. H. Klein, and J. P. Lehoczky. Fixed priority scheduling of periodic tasks with varying execution priority. In *Proceedings of the Twelveth IEEE Symposium on Real-Time Systems*, pages 116–128. IEEE, December 1991.

[34] J. Haritsa, M. Carey, and M. Livny. On being optimistic about real-time constraints. In *Proceedings of the Ninth ACM Symposium on Principles of Database Systems*, pages 331–343, 1990.

[35] M. Herlihy. Impossibility and universality results for wait-free synchronization. In *Proceedings of the Seventh Annual ACM Symposium on Principles of Distributed Computing*, pages 276–290. ACM, August 1988.

[36] M. Herlihy. Wait-free synchronization. *ACM Trans. on Programm. Lang. Syst.*, 13(1):124–149, 1991.

[37] M. Herlihy. A methodology for implementing highly concurrent data objects. *ACM Trans. on Programm. Lang. Syst.*, 15(5):745–770, 1993.

[38] M. Herlihy and J. Wing. Axioms for concurrent objects. In *Proceedings of the Fourteenth ACM Symposium on Principles of Programming Languages*, pages 13–26. ACM, 1987.

[39] M. Herlihy and Jeanette Wing. Linearizability: A correctness condition for concurrent objects. *ACM Trans. on Programm. Lang. Syst.*, 12(3):463–492, 1990.

[40] J. Huang, J. Stankovic, K. Ramamritham, and D. Towsley. Experimental evaluation of real-time optimistic concurrency control schemes. In *Proceedings of the Seventeenth International Conference on Very Large Databases*, pages 35–46, 1991.

[41] A. Israeli and L. Rappaport. Efficient wait-free implementation of a concurrent priority queue. In *Proceedings of the Seventh International Workshop on Distributed Algorithms*, pages 1–16, October 1993.

[42] P. Jayanti and S. Toueg. Some results on the impossibility, universality, and decidability of consensus. In *Proceedings of the Sixth International Workshop on Distributed Algorithms*, pages 69–84. Springer Verlag, November 1992.

[43] K. Jeffay. *The Real-Time Producer/Consumer Paradigm: Towards Verifiable Real-Time Computations.* PhD thesis, University of Washington, Seattle, WA, 1989.

[44] K. Jeffay. Scheduling sporadic tasks with shared resources in hard real-time systems. In *Proceedings of the Thirteenth IEEE Symposium on Real-Time Systems*, pages 89–98. IEEE, December 1992.

[45] K. Jeffay, D. F. Stanat, and C. U. Martel. On non-preemptive scheduling of periodic and sporadic tasks. In *Proceedings of the Twelveth IEEE Symposium on Real-Time Systems*, pages 129–139. IEEE, December 1991.

[46] K. Jeffay and D. Stone. Accounting for interrupt handling costs in dynamic priority task systems. In *Proceedings of the Fourteenth IEEE Symposium on Real-Time Systems*, pages 212–221. IEEE, December 1993.

[47] K. Jeffay, D. Stone, and F. D. Smith. Kernel support for live digital audio and video. *Computer Communications*, 15(6):388–395, July 1992.

[48] Dan I. Katcher, H. Arakawa, and J. K. Strosnider. Engineering and analysis of fixed-priority schedulers. *IEEE Trans. on Software Engineering*, 19(9):920–934, September 1993.

[49] L. Kirousis, E. Kranakis, and P. Vitanyi. Atomic multireader register. In *Proceedings of the Second International Workshop on Distributed Algorithms*, pages 278–296, October 1987.

[50] L. Kirousis, P. Spirakis, and P. Tsigas. Reading many variables in one atomic operation: Solutions with linear or sublinear complexity. In *Proceedings of the Fifth International Workshop on Distributed Algorithms*, pages 229–241, September 1991.

[51] H. Kung and J. Robinson. On optimistic methods for concurrency control. *ACM Trans. on Database Systems*, 6(2):213–226, December 1981.

[52] L. Lamport. Concurrent reading and writing. *Commun. ACM*, 20(11):806–811, November 1977.

[53] L. Lamport. Specifying concurrent program modules. *ACM Transactions on Programming Languages and Systems*, 5(2):190–222, 1983.

[54] L. Lamport. On interprocess communication, parts 1 and 2. *Distributed Computing*, 1:77–101, 1986.

[55] B. W. Lampson and D. D. Redell. Experiences with processes and monitors in mesa. *Communications of the ACM*, 23(2):105–117, 1980.

[56] V. Lanin and D. Shasha. Concurrent set manipulation without locking. In *Proceedings of the Seventh Annual ACM Symposium on Principles of Database Systems*, pages 211–220. ACM, 1988.

[57] J. P. Lehoczky, L. Sha, and Y. Ding. The rate monotonic scheduling algorithm: Exact characterization and average case behavior. In *Proceedings of the Tenth IEEE Symposium on Real-Time Systems*, pages 166–171. IEEE, December 1989.

[58] J. Y. T. Leung and J. Whitehead. On the complexity of fixed-priority scheduling of periodic, real-time tasks. *Performance Evaluation*, 2(4):237–250, 1982.

[59] M. Li, J. Tromp, and P. Vitanyi. How to construct wait-free variables. In *Proceedings of International Colloquium on Automata, Languages, and Programming*, pages 288–505, 1989.

[60] C. Liu and J. Layland. Scheduling algorithms for multiprogramming in a hard real–time environment. *Journal of the ACM*, 30:46–61, January 1973.

[61] M. Loui and H. Abu-Amara. Memory requirements for agreement among unreliable asynchronous processes. *Advances in Computing Research*, 4:163–183, 1987.

[62] H. Massalin. *Synthesis: An Efficient Implementation of Fundamental Operating System Services.* PhD thesis, Columbia University, New York, New York, 1992.

[63] H. Massalin and C. Pu. A lock-free multiprocessor os kernel. Technical Report CUCS-005-91, Columbia University, New York, New York, 1991.

[64] M. Michael and M. Scott. Simple, fast, and practical non-blocking and blocking concurrent queue algorithms. In *Proceedings of the Fifteenth Annual ACM Symposium on Principles of Distributed Computing*, pages 267–276. ACM, May 1996.

[65] A. K. Mok. *Fundamental Design Problems of Distributed Systems for Hard Real-Time Environments.* PhD thesis, Massachusetts Institute of Technology, Cambridge, Mass., 1983.

[66] R. Newman-Wolfe. A protocol for wait-free, atomic, multi-reader shared variables. In *Proceedings of the Sixth Annual Symposium on Principles of Distributed Computing*, pages 232–248, 1987.

[67] G. L. Peterson. Concurrent reading while writing. *ACM Trans. on Programm. Lang. Syst.*, 5(1):46–55, 1983.

[68] G. L. Peterson and J. Burns. Concurrent reading while writing ii: The multi-writer case. In *Proceedings of the 28th Annual ACM Symposium on Foundation of Computer Science.* ACM, 1987.

[69] R. Rajkumar. *Synchronization In Real-Time Systems – A Priority Inheritance Approach.* Kluwer Academic Publishers, Boston, 1991.

[70] S. Ramamurthy, M. Moir, and J. H. Anderson. Real-time object sharing with minimal support. In *Proceedings of the Fifteenth Annual ACM Symposium on Principles of Distributed Computing*, pages 233–242. ACM, May 1996.

[71] L. Sha, R. Rajkumar, and J. P. Lehoczky. Priority inheritance protocols: An approach to real-time system synchronization. *IEEE Transactions on Computers*, 39(9):1175–1185, 1990.

[72] L. Sha, R. Rajkumar, J. P. Lehoczky, and K. Ramamritham. Mode change protocols for priority-driven preemptive scheduling. *Real-Time Systems Journal*, 1(1):243–264, 1989.

[73] N. Shavit and D. Touitou. Software transactional memory. In *Proceedings of the Fourteenth Annual ACM Symposium on Principles of Distributed Computing*, pages 204–213. ACM, August 1995.

[74] A. Singh, J. Anderson, and M. Gouda. The elusive atomic register, revisited. In *Proceedings of the Sixth Annual ACM Symposium on Principles of Distributed Computing*, pages 206–221. ACM, August 1987.

[75] A. Singh, J. Anderson, and M. Gouda. The elusive atomic register. *Journal of the ACM*, 41(2):311–339, 1994.

[76] D. D. Sleator and R. E. Tarjan. Self adjusting binary trees. In *Proceedings of the Fifteenth ACM Symposium on Theory of Computing*, pages 52–59, 1983.

[77] D. L. Stone. *Managing the Effect of Delay Jitter on the Display of Live Continuous Media*. PhD thesis, University of North Carolina, Chapel Hill, North Carolina, 1995.

[78] J. Valois. Implementing lock-free queues. In *Proceedings of the Seventh International Conference on Parallel and Distributed Systems*, pages 64–69, 1994.

[79] J. Valois. *Lock-Free Data Structures*. PhD thesis, Renesselaer Polytechnic Institute, Troy, New York, 1995.

[80] J. Valois. Lock-free linked lists using compare-and-swap. In *Proceedings of the Fourteenth Annual ACM Symposium on Principles of Distributed Computing*, pages 214–222. ACM, August 1995.

[81] J. Wing and C. Gong. A library of concurrent objects and their proofs of correctness. Technical Report CMU-CS-90-151, Carnegie Mellon University, Pittsburg, PA, 1990.

[82] J. Wing and C. Gong. Testing and verifying concurrent objects. *Journal of Parallel and Distributed Computing*, 17(2):164–182, December 1993.

[83] J. Xu and D. L. Parnas. Scheduling processes with release times, deadlines, precedence, and exclusion relations. *IEEE Trans. on Software Engineering*, 16(3):360–369, 1990.