Using Local-Spin k-Exclusion Algorithms to Improve Wait-Free Object Implementations*

James H. Anderson
Department of Computer Science
The University of North Carolina
Chapel Hill, NC 27599

Mark Moir
Department of Computer Science
The University of Pittsburgh
Pittsburgh, PA 15260

November 1995
Revised November 1996, February 1997

Abstract

We present the first shared-memory algorithms for k-exclusion in which all process blocking is achieved through the use of "local-spin" busy waiting. Such algorithms are designed to reduce interconnect traffic, which is important for good performance. Our k-exclusion algorithms are starvation-free, and are designed to be fast in the absence of contention, and to exhibit scalable performance as contention rises. In contrast, all previous starvation-free k-exclusion algorithms require unrealistic operations or generate excessive interconnect traffic under contention. We also show that efficient, starvation-free k-exclusion algorithms can be used to reduce the time and space overhead associated with existing wait-free shared object implementations, while still providing some resilience to delays and failures. The resulting "hybrid" object implementations combine the advantages of local-spin spin locks, which perform well in the absence of process delays (caused, for example, by preemptions), and wait-free algorithms, which effectively tolerate such delays. We present performance results that confirm that this technique can improve the performance of existing wait-free shared object implementations. These results also show that lock-based implementations can be susceptible to severe performance degradation under multiprogramming, while our hybrid implementations are not.

1 Introduction

The *k-exclusion problem* was posed by Fischer et al. [14] as a generalization of the well-known mutual exclusion problem [12]. In the k-exclusion problem, the objective is to design a set of $N > k$ processes, each of which has a "critical section" of code. Each process can enter its critical section repeatedly, and at most k processes may be in their critical sections at any time. In this paper, we present several efficient, starvation-free k-exclusion algorithms for cache-coherent and distributed shared memory multiprocessors.

Our initial motivation for studying the k-exclusion problem arose from the design of a technique for improving the performance of existing wait-free shared object implementations, while simultaneously reducing their space requirements. This technique, which is described in detail later, is based on k-exclusion. For our purposes, a k-exclusion algorithm must satisfy two properties in addition to those mentioned above. The

*Work supported, in part, by NSF grants CCR 9216421 and CCR 9510156, and by a Young Investigator Award from the U.S. Army Research Office, grant number DAAH04-95-1-0323. In addition, the first author was supported by an Alfred P. Sloan Research Fellowship, and the second author was supported by a UNC Alumni Fellowship. A preliminary version of this work appeared in the *Proceedings of the 13th Annual ACM Symposium on Principles of Distributed Computing*, August 1994 [4].

reasons for these requirements are explained later. First, progress must be guaranteed in the face of up to $k-1$ undetectable process halting failures. k-exclusion algorithms that satisfy this property are said to be *starvation-free*. The second requirement is that, before a process p enters its critical section, p is assigned an identifier (or "name") from $\{0 \ldots k-1\}$ that is not assigned to any other process until p leaves its critical section. Following the terminology of Burns and Peterson [9], we call this variant of the k-exclusion problem the *k-assignment* problem.[1]

In this paper, we study starvation-free k-assignment algorithms for shared-memory multiprocessors. Although the k-assignment problem may seem to be much harder than the k-exclusion problem, we show that, on most shared-memory multiprocessors, any k-exclusion algorithm can easily be extended to solve the k-assignment problem. Therefore, this paper is almost entirely devoted to starvation-free algorithms for k-exclusion. Our conversion of k-exclusion algorithms to k-assignment algorithms involves the use of a simple *long-lived renaming* algorithm [21, 22] that allows each process to acquire a name before entering its critical section, and to release that name upon exiting its critical section. The renaming algorithm we present is based on test-and-set, and has time complexity that is directly proportional to the number of processes that concurrently hold or request names.

Table 1 compares the k-exclusion algorithms presented in this paper to previously known ones. This table gives the time complexity (see below) of each listed algorithm, both under contention and in the absence of contention. Throughout this paper, we use the term *contention* to refer to the number of processes that are outside their critical sections. Because we later use our k-exclusion algorithms to implement shared objects, this usage is consistent with the notion of contention for the implemented object (i.e., the number of processes that concurrently access the object). Table 1 also specifies the set of instructions used by each algorithm, and whether or not the algorithm is starvation-free. Time complexity figures in the table specify the number of remote accesses of shared memory required per critical section acquisition.[2] An access is *remote* if it requires a traversal of the global interconnect between processors and shared memory, and *local* otherwise. Observe that all previously published algorithms have very high time complexity under contention, and that most have high time complexity even in the absence of contention.[3] In addition, the algorithms of [14] and [15] assume the existence of large mutually exclusive critical sections that are executed atomically, despite the fact that processes may fail, and the algorithms of [16, 23][4] are not starvation-free. (The algorithm of [16] ensures that, provided there are no process failures, any contending process eventually enters its critical section. However, this algorithm does *not* satisfy the starvation-freedom property considered in this paper, because a *single* process failure can prevent other processes from reaching their critical sections. The algorithm of [23] allows individual processes to starve even if no failures occur.)

Our decision to distinguish between local and remote accesses of shared memory is motivated by recent work on local-spin spin locks [3, 7, 17, 20, 25, 26]. In such locks, the impact of the processor-to-memory bottleneck is minimized by ensuring that processes busy wait only on locally-accessible shared variables. In practice, a shared variable can be made locally-accessible by storing it in a local cache line or in a local partition of distributed shared memory. Performance studies presented in [7, 17, 20, 25, 26] show that minimizing remote memory accesses is important for scalable performance in the design of synchronization algorithms. In particular, studies presented in [20] show that, on both classes of machines, mutual exclusion algorithms that ensure that all busy-wait loops involve only locally-accessible shared variables perform better than similar algorithms that do not.

The k-exclusion algorithms we present employ only local spins for process blocking. The first few algo-

[1] The k-assignment problem was originally posed by Attiya et al. in [8]; they called it "slotted k-exclusion".

[2] This importance of minimizing remote references in busy-waiting algorithms was recognized by T. Anderson [7], by Graunke and Thakkar [17], and by Mellor-Crummey and Scott [20]; the time complexity measure we use to formalize this notion was proposed by Anderson and Yang [6, 26].

[3] We should point out that the designers of previous k-exclusion algorithms were not concerned with minimizing remote memory references. Also, in some cases, they concentrated on providing "fairness" to processes, which our algorithms do not. (See [1] for details.)

[4] The algorithm presented in [23] actually solves mutual exclusion, and not k-exclusion; it is an elementary exercise to modify this algorithm so that it solves k-exclusion [19].

Reference	Time Complexity		Instructions Used	Starvation Free?
	With Contention	Without		
Fischer et al. [14]	∞	$\Theta(1)$	Large Critical Sections	Yes
Fischer et al. [15]	∞	$\Theta(1)$	Large Critical Sections	Yes
Dolev et al. [13]	∞	$\Theta(N^2)$	Safe Bits	Yes
Afek et al. [1]	∞	$\Theta(N)$	Atomic Read and Write	Yes
Peterson [23] (CC)	$\Theta(N^3 - Nk^2)$	$\Theta(N)$	Atomic Read and Write	No
Peterson [23] (DSM)	∞	$\Theta(N^2 - Nk)$	Atomic Read and Write	No
Burns and Peterson [9]	∞	$\Theta(N)$	Atomic Read and Write	Yes
Gottlieb et al. [16]	∞	$\Theta(1)$	Fetch-and-Add	No
CC Algs: Thm. 3	$\Theta(k \log(N/k))$	$\Theta(1)$	Fetch-and-Add, Test-and-Set	Yes
Thm. 4	$\Theta(c)$	$\Theta(1)$	Fetch-and-Add, Test-and-Set	Yes
DSM Algs: Thm. 7	$\Theta(k \log(N/k))$	$\Theta(1)$	Fetch-and-Add, Test-and-Set	Yes
Thm. 8	$\Theta(c)$	$\Theta(1)$	Fetch-and-Add, Test-and-Set	Yes
Thm. 11	$\Theta(k \log(N/k))$	$\Theta(1)$	Above and Compare-and-Swap	Yes
Thm. 12	$\Theta(c)$	$\Theta(1)$	Above and Compare-and-Swap	Yes

Table 1: A comparison of N-process k-exclusion algorithms for shared-memory systems. Time complexity is measured in terms of the number of remote memory references. Therefore, algorithms that do not spin locally have unbounded time complexity. In the first column of time complexity figures, c is the level of contention. The compare-and-swap-based algorithms of Theorems 11 and Theorem 12 improve upon the algorithms of Theorems 7 and 8 by having lower space complexity.

rithms we present are designed for implementation on shared-memory multiprocessors that provide coherent caches. In these algorithms, spins are local only if there is an underlying cache-coherence mechanism. The remaining algorithms in this paper do not require cache coherence. Hence, they can be implemented on distributed shared-memory machines that do not have coherent caches. For both classes of machines, we present algorithms that have $O(k \log(N/k))$ time complexity under contention and algorithms that have time complexity that is directly proportional to contention (see Table 1). As shown in Table 1, all of these algorithms have constant time complexity in the absence of contention, and are based on commonly-available synchronization primitives.

In the latter part of the paper, we present our k-assignment-based technique for improving the performance of existing wait-free shared object implementations. This technique is based on "hybrid" shared object implementations that incorporate both wait-free and lock-based techniques. The hybrid implementations we consider are $(k-1)$-*resilient*, which means that they can withstand undetectable halting failures of up to $k-1$ processes. As explained in Section 5, wait-free objects are a special case of this definition: an N-process object implementation is *wait-free* iff it is $(N-1)$-resilient.

The $(k-1)$-resilient shared object implementations we consider are obtained by encasing a wait-free, k-process object implementation within a k-assignment "wrapper". This wrapper permits only k processes to access the wait-free implementation concurrently, and assigns these processes unique names to use within that implementation. This approach allows $k-1$ process halting failures to be tolerated. From the object designer's standpoint, k is a parameter that can be set on a per-object basis in order to optimize performance. For example, for an object that is expected to be concurrently accessed by a small number of processes, a small value of k will minimize overhead, while still ensuring that processes rarely have to wait to access the object. On the other hand, for an object that is likely to be accessed concurrently by a large number of processes, k should be set higher in order to reduce the likelihood of waiting. Resilient objects constructed in the manner described above combine the advantages of local-spin spin locks, which are scalable in the absence

of multiprogramming, and wait-free algorithms, which effectively tolerate delays (caused, for example, by preemptions).

To demonstrate the utility of the technique described above, we used Herlihy's wait-free universal construction [18] to implement a priority queue, and then used some of our k-assignment algorithms to improve its performance and reduce its space requirements. We conducted performance experiments using these implementations on a cache-coherent multiprocessor and on a distributed shared memory multiprocessor. These experiments confirm that our k-assignment algorithms are fast enough to allow our approach to achieve a significant performance gain. Our performance experiments also demonstrate the advantages of resilient object implementations over lock-based ones. In particular, they show that resilient implementations effectively tolerate the relatively long delays due to preemption under multiprogramming, while a state-of-the-art mutual exclusion algorithm [20] does not. To our knowledge, the experiments we present are the first to demonstrate the advantages that resilient object implementations can have over lock-based implementations in systems that are multiprogrammed. (All previously published performance evaluations of resilient objects that we know of assume a one-process-per-processor model of computation [5, 18].)

The remainder of this paper is organized as follows. In Section 2, we present definitions and notation that will be used in the rest of the paper. In Section 3, we dispense with the k-assignment problem as discussed above by showing that a simple renaming algorithm can be combined with any k-exclusion algorithm to solve k-assignment. We then present k-exclusion algorithms for cache-coherent and distributed shared-memory machines in Sections 4.1 and 4.2, respectively. In Section 5, we present our k-assignment-based technique for improving the performance of wait-free shared object implementations. Finally, we present performance results in Section 6, and concluding remarks in Section 7. Some of the longer proofs appear in an appendix.

2 Preliminaries

Our programming notation should be self-explanatory; as an example of this notation, see Figure 3. With the exception of critical sections, remainder sections, and *Acquire* and *Release* procedures (see Figure 3), we assume that each labeled statement is atomic. Our model of program execution is similar to most others found in the literature, so we only sketch the important details of it here. A program's semantics is defined by a set of histories. A *history* of a program is a sequence $t_0 \xrightarrow{s_0} t_1 \xrightarrow{s_1} \cdots$, where t_0 is an initial state and $t_i \xrightarrow{s_i} t_{i+1}$ denotes that state t_{i+1} is reached from state t_i via the execution of statement s_i. We prove a program correct by proving that its correctness conditions hold in all histories that start from a state that satisfies the initial conditions of the program.

When reasoning about programs, we define safety properties using invariant and unless assertions and progress properties using leads-to assertions [10]. A state assertion[5] is an *invariant* iff it holds in each state of every history. For state assertions B and C, B *unless* C holds iff for each pair of consecutive states in each history, if $B \wedge \neg C$ holds in the first state, then $B \vee C$ holds in the second state. B *leads-to* C iff in each history $t_0 \xrightarrow{s_0} t_1 \xrightarrow{s_1} \cdots$, for each state t_i in which B holds, there is a state t_j in which C holds, where $j \geq i$.

A program that solves the *k-exclusion* problem consists of $N > k$ processes, which we assume are numbered from 0 to $N-1$. Each process begins execution in a *remainder section*, and cycles through its remainder section, an *entry section*, a *critical section*, and an *exit section*. Unless stated otherwise, we assume that no variable (other than program counters) appearing in any entry or exit section is modified in any remainder or critical section.

A program that solves the k-exclusion problem must be able to cope with process halting failures. A process p is *faulty* in a history $t_0 \xrightarrow{s_0} t_1 \xrightarrow{s_1} \cdots$ iff for some $i \geq 0$, process p is outside of its remainder section at state t_i, and $t_i \xrightarrow{s_i} t_{i+1} \xrightarrow{s_{i+1}} \cdots$ includes no statement executions of process p. Informally, a nonfaulty process can only halt in its remainder section. Note that this implies that a nonfaulty process cannot halt in its critical section.

[5] A *state assertion* is a first-order assertion over program variables that does not involve temporal operators.

We now state the key safety and progress requirements for the k-exclusion problem. Let $ES(p)$ ($CS(p)$) be a state assertion that is true iff the value of process p's program counter equals a label of a statement appearing in its entry section (critical section). Then, a program solves the k-exclusion problem iff it satisfies the following properties.

- *k-Exclusion*: $|\{p : 0 \leq p < N :: CS(p)\}| \leq k$ is an invariant. Informally, at most k processes can execute their critical sections at the same time.

- *Starvation-Freedom*: For each process p, $ES(p)$ *leads-to* $CS(p)$ in each history in which p is nonfaulty and at most $k-1$ processes are faulty.[6] Informally, if a nonfaulty process is in its entry section, then that process eventually executes its critical section, provided that fewer than k processes have failed. (We also require that each process in its exit section eventually enters its remainder section; this requirement holds trivially in all of our k-exclusion algorithms, so we do not consider it further.)

The *k-assignment* problem extends the k-exclusion problem by requiring each process p to have a private variable $p.name$ ranging over $\{0, \ldots, k-1\}$. If distinct processes p and q are in their critical sections, then it is required that $p.name \neq q.name$. In other words, we require that $(\forall p, q : p \neq q :: (CS(p) \wedge CS(q)) \Rightarrow (p.name \neq q.name))$ is an invariant.

As mentioned previously, we focus on cache-coherent and distributed shared-memory machines, and measure time complexity by counting "remote" references of shared memory. On distributed shared-memory machines, each shared variable is local to one processor, and remote to all others. Thus, the distinction between local and remote memory references is straightforward. On cache-coherent machines, making this distinction is more problematic. The main difficulty is in determining how many cache misses a busy-waiting loop generates. In our cache-coherent algorithms, all busy waiting is by means of simple loops of the form "**while** $Q = p$ **do od**", where Q is a shared variable and p is the process identifier of the spinning process. We assume that such a loop generates at most two remote memory references. In particular, we assume that the first read of Q generates a remote memory reference that causes a copy of Q to migrate to p's local cache. Subsequent reads before Q is written are therefore local. When another process modifies Q, the cache entry is invalidated, so a subsequent read of Q generates a second remote memory reference. In our cache-coherent algorithms, each process modifies Q only by assigning its own process identifier to Q, so the loop terminates after this second remote read. These assumptions correspond to an idealized write-invalidate cache-coherence protocol, where a cached copy of Q is invalidated only by writes to Q. In fact, other invalidations are possible, for example because of preemption. However, these other invalidations should be rare relative to the frequency of reads that are satisfied by the cache in a tight busy-wait loop. Our time complexity model should therefore give a good indication of actual performance.

The time complexity figures given in this paper specify the worst-case number of remote memory references required for any process to enter and then exit its critical section. Some of the time complexity figures we give depend on the level of contention in a program. For any state of any history, we define *contention* at that state to be the number of processes outside their remainder sections at that state. We say that *contention is at most c in a history* iff contention is at most c at each state of that history. We say that a program that solves the k-exclusion or k-assignment problem has *time complexity R if contention is at most c* iff each matching entry and exit section of any process of that program together generate at most R remote memory references in any history for which contention is at most c. When we refer to the time complexity of a program in the absence of contention, we mean time complexity if contention is exactly one.

Our algorithms employ fetch-and-add, compare-and-swap, and test-and-set instructions. Let X be a shared variable, and let d and e be private variables or constants. Then, the operation $fetch\_and\_add(X, d)$ atomically adds d to X, and returns the original value of X. The operation $compare\_and\_swap(X, d, e)$ has the effect of the following atomic code sequence: "**if** $X = d$ **then** $X := e$; **return** *true* **else return** *false* **fi**". If B is a shared boolean variable, then the operation $test\_and\_set(B)$ atomically assigns "$B :=$ *true*"

[6] It is assumed that every nonfailing process that enters the critical section eventually leaves the critical section.

```
shared variable
    X : array[0..k − 1] of boolean
initially
    (∀i : 0 ≤ i < k :: ¬X[i])

private variable
    name : 0..k − 1

while true do
0:  Remainder Section;
1:  Acquire(N,k);                                   /* Entry section for (N,k)-exclusion */
2:  name := 0;
3:  while test_and_set(X[name]) do name := name + 1 od;   /* Set first clear bit ... */
4:  Critical Section using name name;                    /* ... to get a name */
5:  X[name] := false fi;                            /* Release name by resetting bit found */
6:  Release(N,k)                                    /* Exit section for (N,k)-exclusion */
od
```

Figure 1: Algorithm for k-assignment using test-and-set for renaming.

and returns the original value of B. (Some authors define the return value of this operation to be *true* if the operation successfully changes B from *false* to *true*, and *false* otherwise.)

Before concluding this section, we state several notational conventions that will be used in the remainder of the paper.

Notational Conventions: We use $p.s$ to denote the statement or sequence of statements with label s in process p. If s labels a single statement, then $p@s$ holds iff that statement is enabled for execution. If s labels a sequence of statements (remainder or critical section, *Acquire* or *Release* procedure) then $p@s$ holds iff some statement within that sequence is enabled for execution. We use $p@S$ as shorthand for $(\exists s : s \in S :: p@s)$. We use $p.var$ to represent p's private variable *var*. For brevity, we refer to k-exclusion for N processes as (N,k)-*exclusion*; similarly for (N,k)-*assignment*. Unless stated otherwise, we assume that $k > 0$, $N > k$, and that p, q, and r range over $0..N-1$. □

3 k-Assignment

As explained in the introduction, the k-assignment problem can be solved by combining a solution to the long-lived renaming problem [21, 22] with a program that solves the k-exclusion problem. In the long-lived renaming problem, processes repeatedly acquire and release unique names from a fixed name space.

Figure 1 depicts a program that solves the k-assignment problem in the manner described above. The entry and exit sections of the k-exclusion algorithm being used are denoted by *Acquire*(N,k) and *Release*(N,k), respectively. The renaming mechanism employs a sequence of test-and-set bits, one per name. In order to obtain a name, a process tests each bit in order, until a test-and-set succeeds (line 3). The bit $X[j]$ is associated with name j, where $0 \leq j < k$. A process that has obtained name j releases it by simply clearing $X[j]$ (line 5).

The renaming algorithm employed in Figure 1 is presented in more detail and proved correct in [22]. As the correctness proof of [22] shows, if a process is about to test-and-set $X[i]$, then $\neg X[j]$ holds for some j where $i \leq j < k$. Thus, if a process has unsuccessfully tested bits $X[0]$ through $X[k-2]$, then $\neg X[k-1]$ holds, so the kth test-and-set will succeed. Note that this renaming algorithm has time complexity $k+1$ under contention (k remote references to acquire a name and 1 to release it). Also, if contention is at most c, then it has time complexity $c+1$. Because each process has a private *name* variable, the renaming algorithm requires $O(N)$ space. Thus, we have the following lemma.

```
shared variable
    X : (k − N)..k;
    Q : queue of 0..N − 1
initially
    X = k ∧ Q = null

process p                                                    /* 0 ≤ p < N */
while true do
0:  Remainder Section;
1:  ⟨ if fetch_and_add(X, −1) ≤ 0 then                       /* If no critical section slots are available... */
        Enqueue(p, Q) ⟩;                                     /* ... then get into queue ... */
2:      while Element(p, Q) do /* null */ od                /* ... and busy wait until released */
    fi;
3:  Critical Section;
4:  ⟨ Dequeue(Q);                                            /* Remove first process from Q */
    fetch_and_add(X, 1) ⟩                                    /* Increase counter of available slots again */
od
```

Figure 2: (N, k)-exclusion using atomic queue procedures.

Lemma 1: Suppose that $Acquire(N, k)$ and $Release(N, k)$ can be implemented with time complexity B under contention, and C if contention is at most c, and with space complexity $O(D)$. Then, (N, k)-assignment can be implemented with time complexity $B + k + 1$ under contention, and $C + c + 1$ if contention is at most c, and with space complexity $O(D + N)$. □

4 k-Exclusion Algorithms

In Section 4.1, we present several fast k-exclusion algorithms for cache-coherent machines, and in Section 4.2, we present algorithms for distributed shared-memory machines. First, however, we explain the key insight on which all of our k-exclusion algorithms are based.

On first thought, it may seem that the k-exclusion problem could be efficiently solved by simply modifying a queue-based spin lock [7, 17, 20] so that a process waits in the queue only if k other processes are already in their critical sections. Before giving our first algorithm, we explain why this simple approach is problematic. Consider the simple (unrealistic) queue-based (N, k)-exclusion algorithm in Figure 2. The shared variable X in this algorithm counts the number of processes that may safely enter the critical section. X is initially k. When $X \leq 0$, a process trying to enter the critical section waits in the queue Q. $Enqueue(p, Q)$ and $Dequeue(p, Q)$ are the normal queue operations, and $Element(p, Q)$ is a function that returns true iff p is in Q. Multi-line atomic statements are enclosed in angle brackets.

Aside from the multi-line atomic statements, there are two difficulties involved with implementing this algorithm. First, the queue operations typically require several atomic steps if implemented using only simple primitives. Such an implementation is complicated by the possibility that a process may fail after having only partially executed a queue operation. Second, a queue imposes a linear order on the waiting processes. If a process in the queue fails, then other processes in the queue are blocked.

Note, however, that both problems disappear when $N = k + 1$, because, in this case, at most one process ever waits in the queue. This insight is the basis of the algorithms we present. Specifically, we concentrate on solving $(k + 1, k)$-exclusion, and then inductively apply such a solution to solve (N, k)-exclusion. In the following section we present a $(k + 1, k)$-exclusion for cache-coherent machines, and show how it can be inductively applied to solve (N, k)-exclusion.

```
shared variable
    X : −1..k;                                          /* Counter of available slots */
    Q : 0..N − 1                                        /* Spin location */
initially
    X = k

process p                                               /* 0 ≤ p < N */
while true do
0:  Remainder Section;
1:  Acquire(N, k + 1);                                  /* Entry section of (N, k + 1)-exclusion */
2:  if fetch_and_add(X, −1) = 0 then                    /* No slots available */
3:      Q := p;                                         /* Initialize spin location */
4:      if X < 0 then                                   /* Still no slots available - must wait */
5:          while Q = p do /* null */ od               /* Busy-wait until released */
    fi  fi;
6:  Critical Section;
7:  fetch_and_add(X, 1);                                /* Release a slot */
8:  Q := p;                                             /* Release waiting process (if any) */
9:  Release(N, k + 1)                                   /* Exit section of (N, k + 1)-exclusion */
od
```

Figure 3: (N, k)-exclusion on a cache-coherent machine

4.1 Algorithms for Cache-Coherent Machines

Our $(k+1, k)$-exclusion algorithm for cache-coherent machines is shown in Figure 3. In this algorithm, the idea of having one process in the queue is approximated by using a shared variable Q to store the identifier of the process that is "in the queue". A process can perform the dual functions of enqueueing itself and dequeueing the previously-queued process by simply assigning its own process identifier to Q. The variable X in Figure 3 is being used in the same way as in the queue-based algorithm of Figure 2. In Figure 3, the two procedures *Acquire* and *Release* are inductively assumed to implement $(N, k+1)$-exclusion. That is, we assume the following properties, where the latter two are required to hold only if process p is nonfaulty and at most $k-1$ processes are faulty.

invariant $|\{q :: q@\{2..8\}\}| \leq k+1$ (I1)

$p@1$ *leads-to* $p@2$ (L1)

$p@9$ *leads-to* $p@0$ (L2)

It is assumed that the variables used by $Acquire(N, k+1)$ and $Release(N, k+1)$ are distinct from those in the remainder of the algorithm. Note that if $N = k+1$, then $Acquire(N, k+1)$ and $Release(N, k+1)$ are trivially implemented by skip statements. We later use this as the basis of an induction to show that (N, k)-exclusion can be implemented efficiently.

The algorithm shown in Figure 3 is proved correct by establishing the following properties.

- *k-Exclusion*: **invariant** $|\{p :: p@6\}| \leq k$
- *Starvation-Freedom*: If process p is nonfaulty and at most $k-1$ processes are faulty, then $p@1$ *leads-to* $p@6$. (Given (L2), Starvation-Freedom for the exit section is trivial.)

Several properties are presented below in order to prove k-Exclusion and Starvation-Freedom. The first two of these properties are straightforward to prove directly from the program text, and are therefore stated without proof.

invariant $X = k - |\{p :: p@\{3..7\}\}|$ (I2)

invariant $X < 0 \Rightarrow (\exists p :: p@3 \ \vee \ (p@\{4,5\} \ \wedge \ Q = p))$ (I3)

The invariant given next establishes the k-Exclusion property.

invariant $|\{p :: p@6\}| \leq k$ (I4)

Proof: If $X \geq 0$, then by (I2), $|\{p :: p@\{3..7\}\}| \leq k$ holds, so (I4) holds. If $X < 0$, then by (I3), $(\exists p :: p@\{3,4,5\})$ holds, so by (I1), (I4) holds. □

The following simple unless property, which follows immediately from the program text, is used in the proof of Starvation-Freedom.

$p@5 \ \wedge \ Q \neq p \ \ \textit{unless} \ \ p@6$ (U1)

Starvation-Freedom: If process p is nonfaulty and at most $k-1$ processes are faulty, then $p@1$ *leads-to* $p@6$.

Proof: By (L1) and (L2), the only risk to Starvation-Freedom is that a nonfaulty process p is blocked forever at statement $p.5$. Process p only reaches $p.5$ by executing $p.4$ when $X < 0$ holds. By (I2), this implies that $|\{p :: p@\{3..7\}\}| > k$ holds when $p.4$ is executed. By the assumption that at most $k-1$ processes are faulty, this implies that there is a nonfaulty process $q \neq p$ such that $q@\{3..7\}$ holds when $p.4$ is executed.

If $p@5 \ \wedge \ Q = p$ holds after $p.4$ is executed, then process q is not blocked at $q.5$ because $q \neq p$. If $p@5 \ \wedge \ Q = p$ continues to hold, then q, being nonfaulty, eventually executes $q.8$ and establishes $Q \neq p$. Thus, $p@5 \ \wedge \ Q \neq p$ holds at some state after $p.4$ is executed. Therefore, by (U1), $p@6$ eventually holds, because p is nonfaulty. This concludes the proof of Starvation-Freedom. □

For $N = k+1$, $Acquire(N, k+1)$ and $Release(N, k+1)$ can be trivially implemented by skip statements. Thus, by the above properties, the algorithm shown in Figure 3 can be used to implement $(k+1, k)$-exclusion with time complexity 7 (recall that the loop at statement 5 is assumed to generate at most two remote memory references). Using this result, we can inductively solve (N, k)-exclusion. The inductive algorithm consists of $N-k$ nested "levels", where each level corresponds to an instance of the algorithm of Figure 3. The outermost level solves $(N, N-1)$-exclusion, the next level solves $(N-1, N-2)$-exclusion, and so on. For this and other inductive solutions considered in this section to be correct, we must insist that different instances of the algorithm of Figure 3 use distinct Q and X variables. (This point may seem a little obvious, but we violate it later in Section 4.4.)

Since each level has time complexity 7, the algorithm described in the previous paragraph has time complexity $7(N-k)$. Each level requires constant space, so the algorithm's space complexity is $O(N)$. These observations give us the following result. (In this theorem and those that follow, we only list atomic operations other than reads and writes.)

Theorem 1: Using fetch-and-add, (N, k)-exclusion can be implemented on a cache-coherent machine with time complexity $7(N-k)$ and space complexity $O(N)$. □

On the surface, this algorithm is similar to Peterson's mutual exclusion [23] and k-exclusion (posed as an exercise in [19]) algorithms, in which processes go through a series of levels, each of which stops one process. A key difference between these algorithms and ours is that we have used the fetch-and-add instruction to ensure that each level generates a constant number of remote references, and, in the absence of waiting, each level is completed in constant time. Furthermore, our algorithms tolerate up to $k-1$ process failures, while in Peterson's algorithm, a *single* process failure can prevent other processes from reaching their critical sections.

The inductive algorithm described above requires $O(N)$ remote memory references, which is a significant disadvantage. Note, however, that Theorem 1 implies that $(2k, k)$-exclusion can be implemented with time complexity $7k$. We can use such an algorithm inductively as a "building block" to obtain a more efficient implementation of (N, k)-exclusion. Specifically, we can achieve logarithmic time complexity by arranging

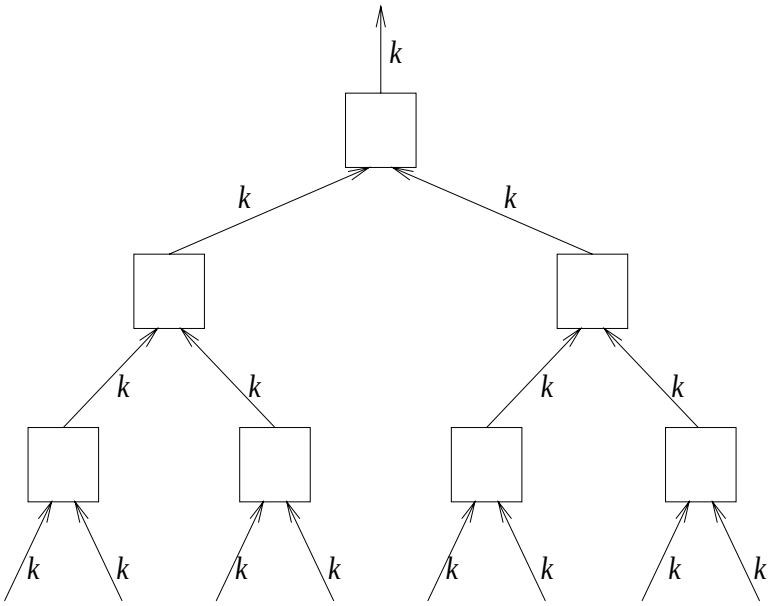

Figure 4: Implementing $Acquire(8k, k)$ in a tree. $Release(N, k)$ is implemented analogously. Each arrow represents a set of processes. Solid boxes represent $Acquire(2k, k)$.

these building blocks in a tree that halves the number of process at each level, until only k remain.[7] Figure 4 depicts this approach for $8k$ processes. This algorithm is shown in Figure 5. In this figure, it is inductively assumed that *Acquire_left* and *Release_left* correctly implement $(\lceil N/2 \rceil, k)$-exclusion, and *Acquire_right* and *Release_right* correctly implement $(\lfloor N/2 \rfloor, k)$-exclusion. In addition, *Acquire_middle* and *Release_middle* are assumed to correctly implement $(2k, k)$-exclusion using the algorithm given in Figure 3 with $N = 2k$.

Before continuing, we should point out a key property of our $(k + 1, k)$-exclusion algorithm that allows it to be efficiently used in inductive applications as described above. Specifically, this algorithm does not require a process in its entry section to know the identity of any other process in advance. To see this, note that in the proof of the algorithm of Figure 3, it does not matter how $Acquire(N, k+1)$ and $Release(N, k+1)$ are actually implemented — they could be implemented using the tree algorithm of the previous paragraph, or any other algorithm that is a correct $(N, k + 1)$-exclusion algorithm. The correctness of the algorithm of Figure 3 also does not depend on which set of up to $k+1$ processes actually make it past $Acquire(N, k+1)$ to execute the code given for $(k+1, k)$-exclusion. Without this property, it might be difficult to efficiently apply such an algorithm inductively as done here. In particular, $O(N)$ loops might be required to detect the identity of competing processes at each instance of the inductively-applied algorithm, resulting in performance that does not scale.

The algorithm in Figure 5 can be proved correct by an induction on the tree depth, using the fact that our $(2k, k)$-exclusion algorithm is correct. The depth of the tree is $\lceil \log_2(N/k) \rceil$, and the time complexity of accessing each $(2k, k)$-exclusion building block is $7k$. The algorithm uses $2\lceil N/k \rceil - 1$ $(2k, k)$-exclusion building blocks in total, and by Theorem 1, each build block requires $O(k)$ space. Thus, the algorithm shown in Figure 5 yields the following result.

Theorem 2: Using fetch-and-add, (N, k)-exclusion can be implemented on a cache-coherent machine with time complexity $7k\lceil \log_2(N/k) \rceil$ and with space complexity $O(N)$. □

[7] Tree-based algorithms for mutual exclusion have been presented previously by Yang and Anderson [26], and by Choy and Singh [11]. However, these algorithms are not easily transformed into k-exclusion algorithm by "chopping off" the top of the tree because this approach would prevent certain sets of k processes from reaching the critical section concurrently.

```
process p                                                    /* 0 ≤ p < N */
while true do
0:   Remainder Section;
1:   if p < ⌈N/2⌉ then                                       /* Half access left subtree */
2:        Acquire_left(⌈N/2⌉, k)
     else                                                    /* Half access right subtree */
3:        Acquire_right(⌊N/2⌋, k)
     fi;
4:   Acquire_middle(2k, k);                                  /* All access root of tree */
5:   Critical Section;
6:   Release_middle(2k, k);
7:   if p < ⌈N/2⌉ then                                       /* Half access left subtree */
8:        Release_left(⌈N/2⌉, k)
     else                                                    /* Half access right subtree */
9:        Release_right(⌊N/2⌋, k)
     fi;
od
```

Figure 5: (N, k)-exclusion in a tree.

The tree approach offers a significant improvement over the approach used in Theorem 1. However, we would like to further reduce the number of remote memory references performed when contention is low. This can be achieved by adding a "fast path", as shown in Figure 7. A test-and-set instruction is used to select one process that directly executes $Acquire(k+1, k)$. The remaining $N-1$ processes must first execute $Acquire(N-1, k)$, thereby ensuring that at most $k+1$ processes concurrently access the innermost $(k+1, k)$-exclusion algorithm. This approach is depicted in Figure 6, in which the dotted box represents $Acquire(N-1, k)$. Using this algorithm, if contention is one, then the test-and-set of the single competing process succeeds. Hence, that process executes only the innermost $Acquire(k+1, k)$ and $Release(k+1, k)$. Observe that this process performs at most 9 remote memory references: 2 are required to set and clear the test-and-set bit, and at most 7 are required for the $(k+1, k)$-exclusion.

The performance under contention for the algorithm shown in Figure 7 is determined by the implementation of $(N-1, k)$-exclusion — the "slow path". One alternative is to use a tree approach like the one illustrated in Figure 4. In this case, a process performs at most $7k\lceil \log_2(N/k) \rceil + 8$ remote memory references: 1 is required for an unsuccessful test-and-set, at most 7 are required for the innermost $(k+1, k)$-exclusion, and at most $7k\lceil \log_2(N/k) \rceil$ are required for the $(N-1, k)$-exclusion tree. Space complexity is dominated by the space required for the $(N-1, k)$-exclusion tree, which by Theorem 2 is $O(N)$. Thus, we have the following result.

Theorem 3: Using fetch-and-add and test-and-set, (N, k)-exclusion can be implemented on a cache-coherent machine with time complexity 9 in the absence of contention, and $7k\lceil \log_2(N/k) \rceil + 8$ under contention, and with space complexity $O(N)$. □

A second alternative is to implement $(N-1, k)$-exclusion inductively using the algorithm given in Figure 7, as depicted inside the dotted box in Figure 6. This results in performance that degrades gracefully with increasing contention, rather than performance that drops suddenly when contention rises. In particular, if contention is at most c, then a process accesses at most c instances of $(k+1, k)$-exclusion, each of which generates at most 7 remote memory references. A process that accesses c instances of $(k+1, k)$-exclusion also performs $c-1$ unsuccessful test-and-set operations and one successful test-and-set (if $c < N$), and clears the bit it successfully sets. This gives a total of $8c+1$ remote memory references. Note that this approach uses $N-k$ instances of $(k+1, k)$-exclusion, and $N-k+1$ test-and-set bits. By Theorem 1, each instance of $(k+1, k)$-exclusion requires $O(1)$ space. Thus, we have the following.

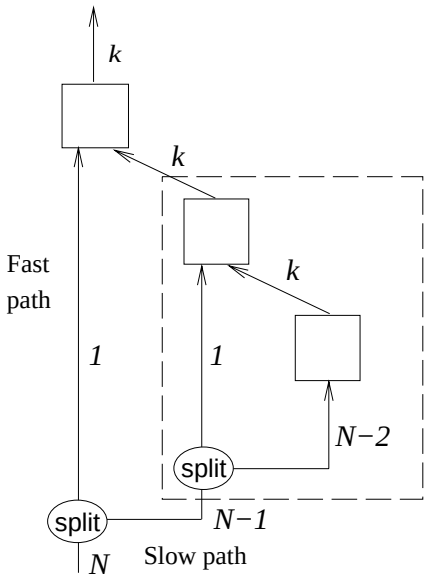

Figure 6: Implementing $Acquire(N, k)$ using fast paths. Each arrow represents a set of processes. The split is implemented using test-and-set, and causes a process that executes alone to take the fast path. Solid boxes represent $Acquire(k+1, k)$. The dotted box represents $Acquire(N-1, k)$. One approach for implementing $Acquire(N-1, k)$ using nested fast paths is depicted for $N = k+3$ (so $N-2 = k+1$).

```
shared variable
    X : boolean;                                   /* Test for fast path */
initially
    X = false

process p                                          /* 0 ≤ p < N */
private variable
    slow : boolean                                 /* Path taken */

while true do
0:  Remainder Section;
1:  slow := test_and_set(X);                       /* Try to get fast path */
2:  if slow then                                   /* Take slow path */
        Acquire(N − 1, k)                          /* Slow path */
    fi;
3:  Acquire(k + 1, k);                             /* Fast path */
4:  Critical Section;
5:  Release(k + 1, k);
6:  if slow then                                   /* Check if slow path was taken */
        Release(N − 1, k)
    else
7:      X := false                                 /* Release fast path */
    fi
od
```

Figure 7: (N, k)-exclusion with a "fast path".

Theorem 4: Using fetch-and-add and test-and-set, (N, k)-exclusion can be implemented on a cache-coherent machine with time complexity $8c + 1$ if contention is at most c, and with space complexity $O(N)$. □

The following corollaries regarding k-assignment follow from Theorems 3 and 4 and Lemma 1.

Corollary 1: Using fetch-and-add and test-and-set, (N, k)-assignment can be implemented on a cache coherent machine with time complexity 11 in the absence of contention, and $7k\lceil \log_2(N/k) \rceil + k + 8$ under contention, and with space complexity $O(N)$. □

Corollary 2: Using fetch-and-add and test-and-set, (N, k)-assignment can be implemented on a cache-coherent machine with time complexity $9c + 2$ if contention is at most c, and with space complexity $O(N)$. □

4.2 Algorithms for Distributed Shared-Memory Machines

In the previous section, we showed that k-exclusion can be efficiently implemented on cache-coherent machines. Such implementations are efficient because when a process waits on a variable, that variable migrates to a local cache line. A distributed shared-memory machine without cache-coherence does not provide this luxury. On such a machine, each variable is local to only one processor, so for good scalability, different processes must wait on different variables. This makes k-exclusion more difficult to implement efficiently. Nevertheless, in this section, we show that (N, k)-exclusion can be implemented efficiently on distributed shared-memory machines without coherent caches. This is achieved by designing algorithms in which all busy waiting is performed on locally-accessible shared variables that are statically allocated to processes. The algorithms presented here are much simpler and more efficient than the distributed shared-memory algorithms we presented in the preliminary version of this paper [4].

Our approach here is the same as that of Section 4.1. In particular, we inductively reduce the problem of implementing (N, k)-exclusion to that of implementing $(k + 1, k)$-exclusion. We present two algorithms for $(k + 1, k)$-exclusion, both of which have constant time complexity. The two algorithms differ in space complexity. The first algorithm uses fetch-and-add and test-and-set. Although this algorithm is efficient, it can lead to high space complexity when used inductively. In particular, each process needs a distinct spin location for each instance of $(n + 1, n)$-exclusion, where $k \leq n < N$, in an inductive application. The second algorithm we present improves on this by being structured so that in inductive applications, each process uses only a constant number of spin locations across all instances of $(n + 1, n)$-exclusion. This reduction in spin locations comes at the expense of using a third primitive, namely compare-and-swap. Both algorithms are based on the same intuition as that of the algorithm for cache-coherent machines in Figure 3. Specifically, we seek to implement a queue of size one to hold any blocked processes. However, the need to rely only on statically allocated local spin locations complicates matters slightly. The resulting correctness proofs are not hard, but are slightly more tedious than that given in the previous section, so we defer their presentation to an appendix.

4.3 First Algorithm

Our first (N, k)-exclusion algorithm for distributed shared-memory machines is given in Figure 8. As before, we assume that the *Acquire* and *Release* procedures correctly implement $(N, k + 1)$-exclusion, and that they use variables distinct from those in the remainder of the algorithm. Instead of all processes waiting on one spin location Q, each process p now has its own local spin location, $P[p]$.

The main difference between this algorithm and the one in Figure 3 is that spin locations are now separate from the queue Q. In the cache-coherent algorithm of Figure 3, a process can enqueue itself onto the queue, dequeue the previously-queued process, and end any spinning by the previously-queued process in one step by simply assigning its own process identifier to Q. In the algorithm shown in Figure 8, it takes several steps

```
shared variable
    X : −1..k;
    Q : 0..N − 1;
    P : array[0..N − 1] of boolean                          /* P[p] is local to process p */
initially
    X = k ∧ Q = 0 ∧ ¬P[0] ∧ (∀i : 1 ≤ i < N :: P[i])

process p                                                   /* 0 ≤ p < N */
private variable
    v : 0..N − 1

while true do
0:  Remainder Section;
1:  Acquire(N, k + 1);                                      /* Entry section of (N, k + 1)-exclusion */
2:  if fetch_and_add(X, −1) = 0 then                        /* No slots available */
3:      v := Q;                                             /* Get current spin location */
4:      if ¬test_and_set(P[v]) then                         /* Release currently spinning process */
5:          Q := p;                                         /* Become waiting process */
6:          P[p] := false;                                  /* Initialize spin location */
7:          if X < 0 then                                   /* Still no slots available - must wait */
8:              while ¬P[p] do /* null */ od               /* Wait until released */
    fi fi fi;
9:  Critical Section;
10: fetch_and_add(X, 1);                                    /* Release a slot */
11: v := Q;                                                 /* Get current spin location */
12: if ¬test_and_set(P[v]) then                             /* Release currently spinning process */
13:     Q := p;                                             /* Pretend to become waiting process */
14:     P[p] := false                                       /* Initialize spin location */
    fi;
15: Release(N, k + 1)                                       /* Exit section of (N, k + 1)-exclusion */
od
```

Figure 8: (N, k)-exclusion for distributed shared-memory machines.

to accomplish all of this. Thus, we could potentially have a situation in which two or more processes both concurrently attempt to enqueue themselves onto Q.

Let us examine this possibility in more detail. When $k + 1$ processes have successfully executed the *Acquire* procedure, it is required that at least one of these processes wait, so that k-Exclusion is not violated. Thus, when a process q releases a process p from its spin loop, process q should itself start waiting. Since it takes several steps for q to accomplish this, it is possible that before q releases p, another process r detects that p is in the queue, releases p, and starts waiting.[8] If q does not detect this, then q might start waiting too. If the $k - 1$ remaining processes are faulty, then q and r might wait forever, violating Starvation-Freedom. Thus, we need a mechanism to allow process q to detect that process p has already been released. The test-and-set instruction in statement 4 of Figure 8 serves this purpose. Specifically, this statement ensures that at most one of q and r will release p from its spinning. This is the essential difference between the algorithm of Figure 8 and that of Figure 3.

With $Acquire(N, k+1)$ and $Release(N, k+1)$ replaced by skip statements, the algorithm in Figure 8 solves $(k + 1, k)$-exclusion on distributed shared-memory machines with time complexity 9 (recall that $P[p]$ is local

[8] To see that it is possible for this many processes to concurrently execute statements 3 through 8, consider the following. From a state in which one process is at statement 3, and k processes are in their critical sections, let one of the latter leave its critical section, and then attempt to enter it again. If no other process takes a step, then that process must reach statement 3. Continuing in this manner, we could reach a state in which $k + 1$ processes are executing statements 3 through 8.

to process p). As before, in inductive applications of this algorithm, we require that different instances of the algorithm employ distinct shared variables. Because of the spin locations, $P[0]$, ..., $P[N-1]$, each instance requires $O(N)$ space (as compared to $O(1)$ for the algorithm of Figure 3). Thus, we have the following counterpart to Theorem 1.

Theorem 5: Using fetch-and-add and test-and-set, (N,k)-exclusion can be implemented on a distributed shared-memory machine with time complexity $9(N-k)$ and with space complexity $O(N^2)$. □

The tree-based approach of Figure 4 can be applied here as well, yielding the following counterpart to Theorem 2.

Theorem 6: Using fetch-and-add and test-and-set, (N,k)-exclusion can be implemented on a distributed shared-memory machine with time complexity $9k\lceil \log_2(N/k) \rceil$ and with space complexity $O(N^2)$. □

The two fast-path approaches described in Section 4.1 can be also be used. Hence, we have the following counterparts to Theorems 3 and 4.

Theorem 7: Using fetch-and-add and test-and-set, (N,k)-exclusion can be implemented on a distributed shared-memory machine with time complexity 11 in the absence of contention, and $9k\lceil \log_2(N/k) \rceil + 10$ under contention, and with space complexity $O(N^2)$. □

Theorem 8: Using fetch-and-add and test-and-set, (N,k)-exclusion can be implemented on a distributed shared-memory machine with time complexity $10c+1$ if contention is at most c, and with space complexity $O(N^2)$. □

The time complexity calculations in the above two theorems are carried out in the same way as was done prior to Theorems 3 and 4, but using 9 as the time complexity of $(k+1,k)$-exclusion instead of 7.

The following corollaries regarding k-assignment follow from Theorems 7 and 8 and Lemma 1.

Corollary 3: Using fetch-and-add and test-and-set, (N,k)-assignment can be implemented on a distributed shared-memory machine with time complexity 13 in the absence of contention, and $9k\lceil \log_2(N/k) \rceil + k + 10$ under contention, and with space complexity $O(N^2)$. □

Corollary 4: Using fetch-and-add and test-and-set, (N,k)-assignment can be implemented on a distributed shared-memory with time complexity $11c+2$ if contention is at most c, with space complexity $O(N^2)$. □

4.4 Second Algorithm

Our first $(k+1,k)$-exclusion algorithm for distributed shared-memory machines has the disadvantage that in inductive applications each process needs a separate spin location for each instance of the algorithm. This can result in space complexity that is somewhat high. Our second $(k+1,k)$-exclusion algorithm for distributed shared-memory machines remedies this problem by allowing each process to use the same two spin locations across all instances of the algorithm in inductive applications. This algorithm is shown in Figure 9. The two spin locations of each process p are denoted $R[p]$ and $P[p]$.

Before examining the code in Figure 9 in detail, let us first consider the pitfalls involved in using a constant number of spin locations per process in inductive applications. Specifically, consider an inductive application of the algorithm in Figure 9, and suppose that each process uses the same R and P variables in all instances of this algorithm (as in our previous algorithms, we assume that other shared variables are not used in different instances). Then, to make sure that such an inductive application is correct, in our correctness proof for the algorithm of Figure 9, we need to allow for the possibility that $R[p]$ and $P[p]$ may be modified in the remainder section, *Acquire*$(N,k+1)$ procedure, critical section, or *Release*$(N,k+1)$ procedure of p or some other process. This is because these sections and procedures may in fact contain other instances of the algorithm in Figure 9 that may modify these variables.

```
shared constant
    IN: an integer value
                /* In inductive applications, different (k+1,k)-exclusion instances use distinct IN numbers */

type
    Spintype = record flag: boolean; instance: integer end

shared variable
    X: −1..k;
    Z: boolean;
    Q: 0..N − 1;
    R, P: array[0..N − 1] of Spintype                    /* R[p] and P[p] are local to process p */
        /* In inductive applications, each (k+1,k)-exclusion instance uses distinct X, Z, and Q variables, ... */
                                  /* ... but R[p] and P[p] are shared across all (k+1,k)-exclusion instances */
initially
    X = k ∧ Z = false ∧ Q = 0

process p                                                                          /* 0 ≤ p < N */
private variable
    v: 0..N − 1

while true do
0:  Remainder Section;
1:  Acquire(N, k+1);                                  /* Entry section of (N,k+1)-exclusion */
2:  if fetch_and_add(X, −1) = 0 then                                  /* No slots available */
3:      if ¬test_and_set(Z) then
4:          v := Q;                                                   /* Get current spin location */
5:          compare_and_swap(P[v], (false, IN), (true, IN));          /* Release currently spinning process */
6:          Q := p;                                                   /* Become waiting process */
7:          P[p] := (false, IN);                                      /* Initialize "first" spin location */
8:          Z := false;
9:          R[p] := (false, IN);                                      /* Initialize "second" spin location */
10:         if X < 0 then                                             /* Still no slots available - must wait */
11:             while P[p] = (false, IN) ∧
12:                   R[p] = (false, IN) do /* null */ od          /* Wait until released */
    fi fi fi;
13: Critical Section;
14: fetch_and_add(X, 1);                                              /* Release a slot */
15: v := Q;                                                           /* Get current spin location */
16: compare_and_swap(R[v], (false, IN), (true, IN));                  /* Release currently spinning process */
17: Release(N, k+1)                                   /* Exit section of (N,k+1)-exclusion */
od
```

Figure 9: Space-efficient (N, k)-exclusion for distributed shared-memory machines.

It should be clear from the preceding paragraph that, in proving the algorithm of Figure 9 correct, we need to allow for the possibility that $R[p]$ and $P[p]$ may be modified by some statement in the remainder section, $Acquire(N, k+1)$ procedure, critical section, or $Release(N, k+1)$ procedure of p or some other process. However, note that in inductive applications of this algorithm, these variables may be modified in these sections and procedures only by executing code like that in Figure 9. If we assign distinct *instance numbers* — the *IN* constant in Figure 9 — to different instances of this code, then the modifications to spin locations in one instance will not adversely interfere with another instance. In the following lemma, we use Hoare triples to enumerate the kinds of interferences that can occur.

Lemma 2: For any statement s in the remainder or critical section, or $Acquire(N, k+1)$ or $Release(N, k+1)$ procedure of any process q, the following properties hold. (In each of these properties, b is universally quantified over $\{false, true\}$.)

$$\{q \neq p \ \wedge \ P[p] = (b, IN)\} \ q.s \ \{P[p] = (b, IN)\} \tag{S1}$$
$$\{q \neq p \ \wedge \ R[p] = (b, IN)\} \ q.s \ \{R[p] = (b, IN)\} \tag{S2}$$
$$\{q = p \ \wedge \ P[p] = (b, IN)\} \ q.s \ \{P[p] = (b, IN) \ \vee \ P[p].instance \neq IN\} \tag{S3}$$
$$\{q = p \ \wedge \ R[p] = (b, IN)\} \ q.s \ \{R[p] = (b, IN) \ \vee \ R[p].instance \neq IN\} \tag{S4}$$
$$\{P[p].instance \neq IN\} \ q.s \ \{P[p].instance \neq IN\} \tag{S5}$$
$$\{R[p].instance \neq IN\} \ q.s \ \{R[p].instance \neq IN\} \tag{S6}$$

Proof: Let statement s be as defined in the lemma. To see that property (S1) holds, note that $q.s$ can change $P[p]$ only by means of a compare-and-swap instruction in an instance other than *IN*. Hence, as different instances of the algorithm in Figure 9 have different instance numbers, s cannot falsify $P[p] = (b, IN)$ in this case, because if it attempted to do so, then its compare-and-swap would fail. Property (S2) holds for similar reasons. To see that property (S3) holds, note that, because different instances of the algorithm use different instance numbers, if $p.s$ modifies $P[p]$, then it establishes $P[p].instance \neq IN$. Property (S4) holds from similar reasons. Finally, to see that property (S5) holds, note that process p only establishes $P[p].instance = IN$ by executing statements within the instance of the algorithm with instance number *IN*. Also, no process $q \neq p$ can change the *instance* field of $P[p]$. Property (S6) holds for similar reasons. □

The discussion above explains much of the insight that underlies the algorithm of Figure 9. The remaining details are as follows. As in the algorithm of the previous subsection, we need to ensure that we do not end up enqueueing two processes at the same time. This is prevented by the test-and-set of Z at statement 5. This test-and-set ensures that at most one process at a time executes the code in statements 4 through 8. Note, however, that a process could enter this region of code and then fail before executing its compare-and-swap at statement 5, i.e., before freeing the currently-spinning process. To get around this potential problem, we use a second spin location that processes update in their exit sections. Thus, if a nonfailed process p is spinning at statements 11 and 12, and if another process executes a successful test-and-set at statement 3 but fails before executing its compare-and-swap at statement 5, then we can show that there is a nonfailed process that will eventually free p by executing its compare-and-swap at statement 16. This completes our informal description of this algorithm.

With $Acquire(N, k+1)$ and $Release(N, k+1)$ replaced by skip statements, the algorithm in Figure 9 solves $(k+1, k)$-exclusion on distributed shared-memory machines with time complexity 10 (recall the $P[p]$ and $R[p]$ are both local to process p). As discussed above, in inductive applications of this algorithm, each process uses the same two spin locations across all instances of $(n+1, n)$-exclusion, where $k \leq n < N$. This gives a total of $O(N)$ space for spin locations. In addition, constant additional space is required for each instance. Thus, if an inductive application uses M instances of $(n+1, n)$-exclusion, then its space complexity is $O(N+M)$. With these observations in mind, we have the following counterparts to Theorems 5 through 8, and Corollaries 3 and 4 respectively.

Theorem 9: Using fetch-and-add, compare-and-swap, and test-and-set, (N, k)-exclusion can be implemented on a distributed shared-memory machine with time complexity $10(N-k)$ and with space complexity

$O(N)$. □

Theorem 10: Using fetch-and-add, compare-and-swap, and test-and-set, (N, k)-exclusion can be implemented on a distributed shared-memory machine with time complexity $10k\lceil \log_2(N/k) \rceil$ and with space complexity $O(N)$. □

Theorem 11: Using fetch-and-add, compare-and-swap, and test-and-set, (N, k)-exclusion can be implemented on a distributed shared-memory machine with time complexity 12 in the absence of contention, and $10k\lceil \log_2(N/k) \rceil + 11$ under contention, and with space complexity $O(N)$. □

Theorem 12: Using fetch-and-add, compare-and-swap, and test-and-set, (N, k)-exclusion can be implemented on a distributed shared-memory machine with time complexity $11c + 1$ if contention is at most c, and with space complexity $O(N)$. □

Corollary 5: Using fetch-and-add, compare-and-swap, and test-and-set, (N, k)-assignment can be implemented on a distributed shared-memory machine with time complexity 14 in the absence of contention, and $10k\lceil \log_2(N/k) \rceil + k + 11$ under contention, and with space complexity $O(N)$. □

Corollary 6: Using fetch-and-add, compare-and-swap, and test-and-set, (N, k)-assignment can be implemented on a distributed shared-memory with time complexity $12c + 2$ if contention is at most c, with space complexity $O(N)$. □

5 Using k-Assignment to Improve Wait-Free Object Implementations

In this section, we describe a technique for using k-assignment to improve both the performance and the space requirements of wait-free shared object implementations.

A *shared object* is a data structure, along with associated operations, that is shared by a collection of processes. The conventional approach to implementing shared objects involves encapsulating operations within mutually exclusive critical sections. However, the use of mutually exclusive critical sections in multiprogrammed systems presents a problem: if a process is preempted while accessing a shared object within a critical section, then other processes are subsequently prevented from accessing that object until the preempted process is resumed. To address this problem, other researchers have suggested using a modified kernel interface that releases any critical section acquired by a preempted process [24]. However, operating-system-based solutions such as this entail added scheduling overhead and limit portability.

Such operating system support can be avoided if the (user-level) code that processes execute to access objects is tolerant of delays. Algorithms that are delay-tolerant in this way are called "resilient". In formal terms, an implementation of an object is *t-resilient* iff any process can complete any operation in a finite number of its own steps, provided at most t other processes fail undetectably by halting. Because a t-resilient object is able to withstand up to t "permanent" halting failures, it can effectively tolerate up to t simultaneously delayed processes.

Resilient object implementations have received much attention in the distributed algorithms community. Of particular interest to this community are object implementations that are completely tolerant of delays, i.e., N-process object implementations that are $(N-1)$-resilient. Such implementations are called *wait-free*. To see why, note that in an N-process wait-free object implementation, if $N-1$ processes experience delays, then the remaining process can complete any object access in a finite number of its own steps. This precludes the use of idle-waiting constructs or busy-waiting loops.

Because wait-free algorithms preclude all waiting dependencies among processes, they are well suited to implementing shared objects in multiprogrammed systems. Unfortunately, the high level of resiliency ensured by wait-free objects comes at a price: most wait-free algorithms have time complexity that is at

least proportional to N, the total number of processes that *potentially* access the object; hence, performance does not scale gracefully as the number of processes increases. In most wait-free algorithms, such poor scalability is largely the result of having to tolerate $N-1$ process failures.[9]

Wait-free algorithms are designed to tolerate the delay of any process when *all* processes simultaneously access an object. However, in a well-designed application, one would seldom expect all processes to compete simultaneously for a single object. In short, wait-freedom links resiliency to worst-case contention, and this can be overkill in practice. From a performance standpoint, linking resiliency to *expected* levels of contention may be preferable. However, doing so requires an approach for efficiently implementing shared objects that tolerate fewer than $N-1$ failures, while incurring less overhead. This can be accomplished by combining a wait-free algorithm for $k < N$ processes with one of our k-assignment algorithms. In particular, we can implement a $(k-1)$-resilient shared object by encasing a wait-free, k-process implementation of that object within a k-assignment "wrapper". This wrapper permits only k processes to access the wait-free implementation concurrently, and assigns these processes unique names from a range of size k to use within that implementation. This approach allows $k-1$ process failures to be tolerated (recall that our k-assignment algorithms can tolerate $k-1$ process failures). Hence, if contention is at most k, then such an implementation is effectively wait-free.

In the following section we present the results of performance experiments that show that our technique can improve the performance of wait-free object implementations.

6 Performance Results

In the following two subsections, we present results from performance experiments that compare $(k-1)$-resilient objects, implemented using some of our k-assignment algorithms, as discussed above, with wait-free and spin-lock-based object implementations. The first of these subsections contains results of experiments conducted on a Sequent Symmetry — a cache-coherent multiprocessor, while the second contains results from experiments conducted on a BBN GP1000 — a distributed shared-memory multiprocessor without a cache-coherence mechanism. These experiments show that, for both classes of machines,

- our k-assignment-based technique does indeed improve the performance of the wait-free implementation; and
- the ability of the wait-free and $(k-1)$-resilient implementations to tolerate delays can be a significant advantage over lock-based implementations in multiprogrammed systems.

These results also verify that our k-exclusion algorithms are fast enough to be useful for the technique described in the previous section.

All of our experiments have the same structure, so before we present any specific details, we give a brief overview. On both machines, we implemented a shared priority queue[10] using the local-spin queue lock of Mellor-Crummey and Scott [20], and Herlihy's universal wait-free object construction [18]. To test the performance of our algorithms we also used the "inductive fast path" method of Figure 7 with each level implemented using the algorithm in Figure 3 on the Sequent Symmetry, and in Figure 8 on the BBN GP1000. In each experiment, a fixed number of priority queue operations are performed (50,000 on the Sequent Symmetry, and 20,000 on the BBN GP1000). The number of participating processes is varied, and the priority queue operations are equally divided among these processes. Previous experiments involving scalable synchronization constructs have assumed that each process runs on a dedicated processor [7, 17, 20, 25]. However, in practice it can be desirable to run more than one process on each processor. In our experiments, we consider scenarios in which processes share processors by multiprogramming. In order to test

[9] Exceptions include objects, such as snapshot objects [2], in which operations return state information that is $\Omega(N)$ in size. By definition, such objects will not scale well, even at lower levels of resiliency.

[10] We used the same priority queue code as that presented in [18]

the performance of each method under varying levels of multiprogramming, we fix the number of processors (to 10 on the Sequent Symmetry and to 40 on the BBN GP1000) [11] and vary the number of processes.

In each performance graph presented, we plot the total time taken to complete the operations using the various object implementations being compared. In the case of our approach, we show $(k-1)$-resilient implementations for varying values of k. An object implementation scales well if its total time does not increase much as the number of participating processes increases, i.e., if the curve plotted for that implementation is relatively flat.

6.1 Cache-Coherent Multiprocessors

The results presented in this section are taken from experiments run on a Sequent Symmetry multiprocessor. The Sequent Symmetry is a shared-memory multiprocessor whose processor and memory nodes are interconnected via a shared bus. A processor node consists of an Intel 80386 and a 64 Kbyte, two-way set-associative cache. Cache coherence is maintained by a snoopy protocol. The Symmetry provides an atomic fetch-and-store instruction. For these experiments, we simulated a simple, round-robin multiprogramming environment through the use of a dedicated processor to act as a scheduler. (Scheduling on the Sequent Symmetry is priority-based and is therefore not particularly representative of general multiprogramming environments.) Processes are preempted and rescheduled using Unix signals, and a preempted process waits using the Unix `sigpause()` system call. We simulated synchronization primitives that are not directly provided by using short critical sections. In order to closely simulate these primitives, they and the scheduler were designed so that a process would not be preempted while executing in one of these critical sections.

Figure 10 compares the performance of various implementations of a shared priority queue. First, observe that Mellor-Crummey and Scott's algorithm suffers severely under multiprogramming. This is because processes in this algorithm wait in a queue for access to the critical section. If a process in the queue is delayed due to preemption, then all the processes behind it in the queue are also delayed. Furthermore, while these processes are waiting for the delayed process, they are likely to be preempted themselves, exacerbating the problem even further.

Figure 10 also shows the performance of Herlihy's wait-free algorithm in this setting. Two curves are shown for Herlihy's algorithm — one for a 100-process implementation and one for a 200-process implementation. (Note that these are not the actual number of processes participating in the experiments, but the maximum number of processes the implementation can accommodate.) It is interesting to note that the resiliency provided by Herlihy's algorithm allows it to outperform Mellor-Crummey and Scott's algorithm by a significant margin in both cases. However, as mentioned above, many wait-free shared object implementations, including Herlihy's, do not scale well as N — the total number of processes for which the object is implemented — increases. This is because they have time complexity that is at least proportional to N. This is demonstrated in the case of Herlihy's algorithm by the fact that the 100-process implementation outperforms the 200-process implementation, despite the fact that the same number of processes perform the same number of operations in each case. As discussed below, objects implemented using our algorithms scale better because they do not have time complexity that depends on N.

We now turn to the performance of objects implemented using the k-exclusion algorithm presented in Figure 3. Observe that, under multiprogramming, our algorithm performs significantly better than Mellor-Crummey and Scott's for all values of k shown, and also better than Herlihy's algorithm in most cases. Also observe that, as the level of multiprogramming increases, the performance of our approach is better for larger choices of k, that is, when a higher level of resilience is provided. However, when the highest level of resilience, i.e., wait-freedom, is used, performance is worse. This demonstrates the utility of algorithms such as ours that allow the level of resilience of a shared object implementation to be set according to system parameters.

It is interesting to note that when $k = 1$ — that is, when our algorithm is reduced to a mutual exclusion

[11] In each case, this was the maximum number of processes we could reliably and repeatedly acquire for our experiments.

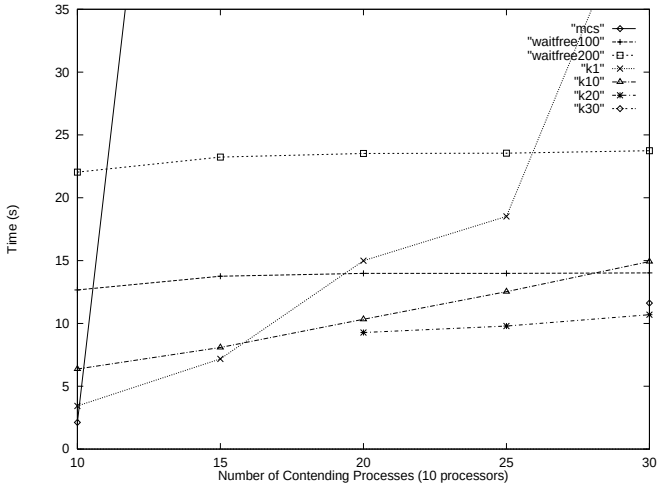

Figure 10: Performance Experiments on the Sequent Symmetry.

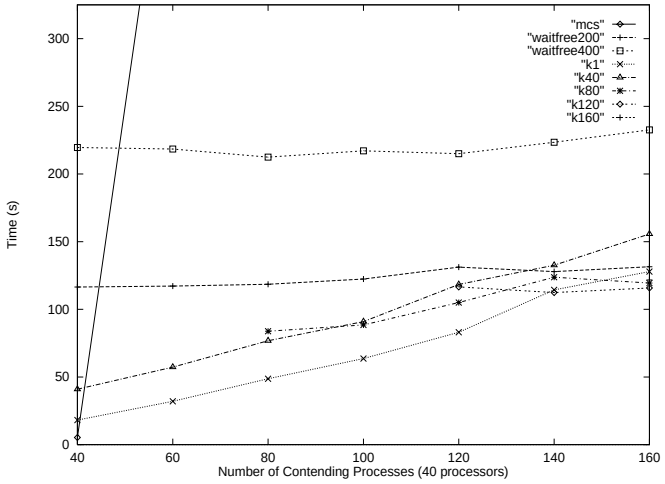

Figure 11: Performance Experiments on the BBN GP1000.

algorithm — it significantly outperforms Mellor-Crummey and Scott's mutual exclusion algorithm, and in some cases outperforms implementations (both Herlihy's and ours) that provide higher levels of resiliency. Two factors contribute to this good performance. First, our k-exclusion algorithm does not enforce a strict FIFO order on waiting processes. Thus, the bad behavior described above for Mellor-Crummey and Scott's algorithm is not exhibited by ours. Second, our k-exclusion algorithm does not need to provide any resiliency when $k = 1$, which makes it simpler than Herlihy's algorithm and simpler than our algorithm for higher values of k. Despite these advantages, however, our 1-exclusion algorithm can be seen to suffer under higher levels of multiprogramming. This is because, as more and more waiting processes are preempted, the likelihood of "chains" of waiting processes increases. Our algorithm outperforms Mellor-Crummey and Scott's because their algorithm is queue-based, which means these chains are much more likely to form, and once they have formed, are almost certain to remain. In contrast, our algorithm avoids these chains forming, and also allows the chains to disband when the preempted processes resume execution. This is achieved by allowing processes to enter the critical section despite the preemption of other waiting processes.

6.2 Distributed Shared-Memory Multiprocessors

The results presented in this section are taken from experiments run on a BBN GP1000 multiprocessor. The BBN GP1000 is a shared-memory multiprocessor whose processor and memory nodes are interconnected using a BBN Butterfly Switch. Each processor node consists of a Motorola 68020 with no cache. Processes are assigned to processors on a round-robin basis upon creation, and do not subsequently move to other processors. Each node schedules its processes on a round-robin basis, with a quantum of 100ms. The GP1000 provides an atomic "clear-then-add" instruction, which can be used to directly implement test-and-set and fetch-and-add instructions. We simulated synchronization primitives that are not directly provided by using short critical sections. Unlike our experiments on the Sequent Symmetry, processes on the BBN GP1000 are multiprogrammed under kernel control. As a result, it is possible for a process to be preempted within the critical section of a simulated synchronization primitive. However, simulated primitives are used to a similar extent by all algorithms tested, so this problem does not unfairly penalize any of the algorithms.

We performed experiments on the BBN GP1000 similar to those described in the previous subsection. In this case, we used 40 processors, and varied the number of processes between 40 and 160. As mentioned above, we used the k-exclusion algorithm presented in Figure 8 to test the performance of our approach to shared object implementation. The results are presented in Figure 11.

Figure 11 shows trends for the relative performance of the three implementations that are similar to those shown for the Sequent Symmetry. Again, Mellor-Crummey and Scott's algorithm suffers severely under multiprogramming and Herlihy's algorithm performs much better. As before, two curves are shown for Herlihy's algorithm — this time one curve is for a 200-process implementation and one for a 400-process implementation. As in the Sequent Symmetry experiments, the performance of Herlihy's algorithm does not scale well as N increases. When $k = 1$, our algorithm is reduced to a mutual exclusion algorithm and outperforms all other approaches for most data points shown. However, as the multiprogramming level increases, the performance of this approach degrades, and by choosing k appropriately, the k-exclusion-based approach can be configured to perform better. The reason that the $k = 1$ case performs better on the BBN GP1000 than it did on the Sequent Symmetry relative to the other approaches is the increased relative cost of copying on the BBN GP1000. In order to tolerate delays, Herlihy's algorithm performs each operation on a copy of the object and later attempts to install this copy as the new state of the object. The need to make copies of the object adversely affects the performance of Herlihy's algorithm and therefore of our algorithms when $k > 1$. However, the increased cost of copying does not degrade performance when $k = 1$ because no copying is necessary in this case. This concludes the description of our performance experiments.

7 Concluding Remarks

We have presented several shared-memory algorithms for k-exclusion and k-assignment in which all process blocking is achieved through the use of "local-spin" busy waiting. These algorithms are designed to minimize interconnect traffic associated with busy waiting on cache-coherent and distributed shared-memory multiprocessors. The algorithms we have presented are starvation-free, are based on commonly-available synchronization primitives, and exhibit scalable performance. In contrast, all prior starvation-free k-exclusion algorithms either require unrealistic atomic operations, or have unacceptably high time complexity. To our knowledge, the algorithms of this paper are the first local-spin synchronization algorithms to tolerate process failures.

To illustrate the utility of our algorithms, we have also presented a technique for improving the performance and the space requirements of wait-free shared object implementations. This technique uses a starvation-free k-assignment algorithm to restrict access to a k-process, wait-free implementation, thereby producing a $(k-1)$-resilient implementation of that object. To test viability of this technique, we conducted performance studies on cache-coherent and distributed shared-memory multiprocessors that compare the performance of a priority queue implemented using a wait-free construction, using the same wait-free

construction combined with our k-exclusion algorithms, and using a well-known spinlock algorithm. While these experiments are not very extensive (for example, we did not test other wait-free constructions, spinlock algorithms, or objects, and we did not conduct experiments to determine whether these implementations scale for use in systems with very large numbers of processors), they do allow us to draw two useful conclusions. First, in multiprogrammed systems, our "hybrid" object implementations do indeed perform better than the wait-free implementations on which they are based. Thus, efficient k-exclusion algorithms (either ours or others) *can* be used to improve the performance of wait-free constructions. Second, our experiments demonstrate the advantages of tolerating delays under multiprogramming.

It would be interesting to try to improve upon our results by developing k-exclusion algorithms for which performance under contention is completely independent of N. Ideally, we would like for such algorithms to have performance that approaches that of the fastest spin-lock algorithms when k approaches 1. We leave this as a topic for further research.

Acknowledgement: We are grateful to the anonymous referees for their helpful comments, and to Phil McKinley and Chuck Severance of Michigan State University for their assistance with the use of their BBN GP1000 multiprocessor. We also acknowledge Argonne National Laboratories for providing us with access to their Sequent Symmetry.

References

[1] A. Afek, D. Dolev, E. Gafni, M. Merritt, and N. Shavit, "A Bounded, First-In, First-Enabled Solution to the ℓ-Exclusion Problem", *ACM Transactions on Programming Languages and Systems*, 16(3), 1994, pp. 939-953.

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

[3] J. Anderson, "A Fine-Grained Solution to the Mutual Exclusion Problem", *Acta Informatica*, 30(3), 1993, pp. 249-265.

[4] J. Anderson and M. Moir, "Using k-Exclusion to Implement Resilient, Scalable Shared Objects", *Proceedings of the 13th Annual ACM Symposium on Principles of Distributed Computing*, ACM, New York, 1994, pp. 141-150.

[5] J. Anderson and M. Moir, "Universal Constructions for Large Objects", *Proceedings of the Ninth International Workshop on Distributed Algorithms*, Lecture Notes in Computer Science 972, Springer-Verlag, 1995, pp. 168-182.

[6] J. Anderson and J.-H. Yang, "Time/Contention Tradeoffs for Multiprocessor Synchronization", *Information and Computation*, 124(1), 1996, pp. 68-84.

[7] T. Anderson, "The Performance of Spin Lock Alternatives for Shared-Memory Multiprocessors", *IEEE Transactions on Parallel and Distributed Systems*, 1(1), 1990, pp. 6-16.

[8] H. Attiya, A. Bar-Noy, D. Dolev, D. Peleg, and R. Reischuk, "Renaming in an Asynchronous Environment", *Journal of the ACM* 37(3), 1990, pp. 524-548.

[9] J. Burns and G. Peterson, "The Ambiguity of Choosing", *Proceedings of the Eighth Annual ACM Symposium on Principles of Distributed Computing*, ACM, New York, 1989, pp. 145-157.

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

[11] M. Choy and A. Singh, "Adaptive Solutions to the Mutual Exclusion Problem", *Distributed Computing*, 8(1), 1994, pp. 1-17.

[12] E. Dijkstra, "Solution of a Problem in Concurrent Programming Control", *Communications of the ACM*, 8(9), 1965, p. 569.

[13] D. Dolev, E. Gafni, and N. Shavit, "Towards a Non-atomic Era: l-Exclusion as a Test Case", *Proceedings of the 20th ACM Symposium on Theory of Computing*, 1988, pp. 78-92.

[14] M. Fischer, N. Lynch, J. Burns, and A. Borodin, "Resource Allocation with Immunity to Process Failure", *Proceedings of the 20th Annual IEEE Symposium on Foundations of Computer Science*, 1979, pp. 234-254.

[15] M. Fischer, N. Lynch, J. Burns, and A. Borodin, "Distributed FIFO Allocation of Identical Resources Using Small Shared Space", *ACM Transactions on Programming Languages and Systems*, 11(1), 1989, pp. 90-114.

[16] A. Gottlieb, B. Lubachevsky, and L. Rudoph, "Basic Techniques for the Efficient Coordination of Very Large Numbers of Cooperating Sequential Processors", *ACM Transactions on Programming Languages and Systems*, 5(2), 1983, pp. 164-189.

[17] G. Graunke and S. Thakkar, "Synchronization Algorithms for Shared-Memory Multiprocessors", *IEEE Computer*, 23, 1990, pp. 60-69.

[18] M. Herlihy, "A Methodology for Implementing Highly Concurrent Data Objects", *ACM Transactions on Programming Languages and Systems*, 15(5), 1993, pp. 745-770.

[19] N. Lynch, "*Distributed Algorithms*", Morgan Kaufmann, San Mateo, Calif., 1996.

[20] J. Mellor-Crummey and M. Scott, "Algorithms for Scalable Synchronization on Shared-Memory Multiprocessors", *ACM Transactions on Computer Systems*, 9(1), 1991, pp. 21-65.

[21] M. Moir and J. Anderson, "Fast, Long-Lived Renaming", *Proceedings of the Eighth International Workshop on Distributed Algorithms*, Lecture Notes in Computer Science 857, Springer-Verlag, 1994, pp. 141-155.

[22] M. Moir and J. Anderson, "Wait-Free Algorithms for Fast, Long-Lived Renaming", *Science of Computer Programming* 25, 1995, pp. 1-39.

[23] G. Peterson, "Myths about the Mutual Exclusion Problem", *Information Processing Letters*, 12(3), 1981, pp. 115-116.

[24] R. Wisniewski, L. Kontothanassis, and M. Scott, "Scalable Spin Locks for Multiprogrammed Systems", Technical Report 454, Computer Science Department, University of Rochester, 1993.

[25] J.-H. Yang and J. Anderson, "Fast, Scalable Synchronization with Minimal Hardware Support", *Proceedings of the 12th Annual ACM Symposium on Principles of Distributed Computing*, ACM, New York, 1993, pp. 171-182.

[26] J.-H. Yang and J. Anderson, "A Fast, Scalable Mutual Exclusion Algorithm", *Distributed Computing*, 9, 1995, pp. 51-60.

Appendix: Proofs for Distributed Shared-Memory Algorithms

In this appendix, we provide formal correctness proofs for the two algorithms for $(k+1, k)$-exclusion presented in Section 4.2. Each algorithm is considered in a separate subsection.

First Algorithm

In this subsection, we prove that the algorithm of Figure 8 is correct. As in Section 4.1, we assume the following properties, where the latter two are required to hold only if process p is nonfaulty and at most $k-1$ processes are faulty.

invariant $|\{q :: q@\{2..14\}\}| \leq k+1$ (I5)

$p@1 \;\; \textit{leads-to} \;\; p@2$ (L3)

$p@15 \;\; \textit{leads-to} \;\; p@0$ (L4)

Before proving that k-Exclusion and Starvation-Freedom hold for the program of Figure 8, we first establish some useful properties. The first of these properties is straightforward and is stated without proof.

invariant $X = k - |\{p :: p@\{3..10\}\}|$ (I6)

We now prove that the following three assertions taken together are an invariant. This, of course, implies that each of these assertions taken individually is an invariant.

$$(\neg P[q] \Rightarrow Q = q) \;\wedge\; ((\exists p :: p@\{5,6,13,14\}) = (\forall i :: P[i])) \tag{I7}$$

$$|\{q :: q@\{5,6,13,14\}\}| \leq 1 \tag{I8}$$

$$p@\{6,14\} \;\vee\; (p@\{7,8\} \;\wedge\; \neg P[p]) \Rightarrow Q = p \tag{I9}$$

invariant (I7) $\wedge$ (I8) $\wedge$ (I9)

Proof: It is straightforward to check that each of (I7), (I8), and (I9) is initially true. In the remainder of the proof, we show that no statement execution falsifies any of these assertions if executed when *all* three assertions hold.

The first conjunct of (I7) could potentially be falsified by any statement that falsifies $P[q]$ or that modifies Q. The statements to check are $q.6$, $q.14$, $p.5$, and $p.13$, where p is any process. By (I9), $Q = q$ holds before (and hence after) the execution of $q.6$ or $q.14$. By the second conjunct of (I7), $P[q]$ is true before (and hence after) the execution of $p.5$ or $p.13$.

Now, consider the second conjunct of (I7). Observe that the assertion $p@\{5,6,13,14\}$ is established only by $p.4$ or $p.12$, and only if $P[p.v]$ is false. By the first conjunct of (I7), this is the only component of P that is false when $p.4$ or $p.12$ is executed. Therefore, after the execution of one of these statements, all components of P are true. $p@\{5,6,13,14\}$ is only falsified by $p.6$ or $p.14$, both of which falsify $(\forall i :: P[i])$. By (I8), both also falsify $(\exists p :: p@\{5,6,13,14\})$. Note that no statements other than $p.4$, $p.6$, $p.12$, and $p.14$ establish or falsify $(\forall i :: P[i])$, and that we have already shown that these statements do not falsify (I7).

(I8) is proved using the second conjunct of (I7). In particular, if $p@\{5,6,13,14\}$ holds for some process p, then $(\forall i :: P[i])$ holds. Hence, $q@\{5,6,13,14\}$ cannot be established for another process q.

The antecedent of (I9) is only established by $p.5$ or $p.13$, each of which also establishes the consequent. The consequent of (I9) is only falsified by $q.5$ or $q.13$, where $q \neq p$. By (I8), $q@\{5,13\}$ implies that $p@\{6,14\}$ is false. By the second conjunct of (I7), $q@\{5,13\}$ also implies that $p@\{7,8\} \wedge \neg P[p]$ is false. Thus, if either $q.5$ or $q.13$ is enabled, then the antecedent of (I9) is false. □

invariant $X < 0 \Rightarrow (\exists r :: r@3 \;\vee\; (r@4 \;\wedge\; r.v = Q) \;\vee\; r@\{5,6\} \;\vee\; (r@\{7,8\} \;\wedge\; \neg P[r]))$ (I10)

Proof: The antecedent of (I10) is initially false, and is established only by $p.2$, which also establishes $p@3$. In the remainder of the proof, we check those statements that may falsify a disjunct of the consequent if executed when the antecedent holds. We show that if such a disjunct is falsified, then another disjunct holds.

$p@3$ can only be falsified by $p.3$, which also establishes $p@4 \ \wedge \ p.v = Q$.

$p@4 \ \wedge \ p.v = Q$ can be falsified only by $p.4$ or $q.5$ or $q.13$, where $q \neq p$. If $p.4$ is executed when $(\forall i :: P[i])$ holds, then by the second conjunct of (I7), $(\exists q :: q\{5, 6, 13, 14\})$ holds. If $X < 0$ holds, then, by (I5) and (I6), $(\exists q :: q\{5, 6\})$ holds, which implies that $p.4$ does not falsify (I10). If $p.4$ is executed when $(\forall i :: P[i])$ does not hold, then by the first conjunct of (I7), $P[p.v]$ is false (since $p.v = Q$). Hence, $p.4$ establishes $p@5$. Finally, $q.5$ establishes $q@6$, and if $X < 0$, then (I5) and (I6) imply that $q.13$ is not enabled.

$p@\{5, 6\}$ is falsified only by $p.6$, which establishes $p@\{7, 8\} \ \wedge \ \neg P[p]$.

If $X < 0$, then $p@\{7, 8\} \ \wedge \ \neg P[p]$ can only be falsified by $q.4$ or $q.12$ for some $q \neq p$. However, if $q.4$ establishes $P[p]$, then it also establishes $q@5$. Also, because $X < 0$, (I5) and (I6) imply that $q.12$ is not enabled. □

The following invariant establishes the k-Exclusion property for the program of Figure 8.

invariant $|\{p :: p@9\}| \leq k$ (I11)

Proof: If $X \geq 0$, then by (I6), $|\{p :: p@\{3..10\}\}| \leq k$ holds, so (I11) holds. If $X < 0$, then by (I10), $(\exists p :: p@\{3..8\})$ holds, so by (I5), (I11) holds. □

The following unless property, which follows directly from the program text, is used in the proof of Starvation-Freedom.

$p@8 \ \wedge \ P[p] \ \ \textit{unless} \ \ p@9$ (U2)

Starvation-Freedom: If process p is nonfaulty and at most $k - 1$ processes are faulty, then $p@1$ *leads-to* $p@9$.

Proof: By (L3) and (L4), the only risk to Starvation-Freedom is that a nonfaulty process p is blocked forever at $p.8$. Process p only reaches $p.8$ by executing $p.7$ when $X < 0$ holds. By (I6), this implies that $|\{p :: p@\{3..10\}\}| > k$ holds when $p.7$ is executed. By the assumption that at most $k-1$ processes are faulty, this implies that there is a nonfaulty process $q \neq p$ such that $q@\{3..10\}$ holds when $p.7$ is executed.

We now show that $p@8 \ \wedge \ P[p]$ holds at some state after $p.7$ is executed. Assume, to the contrary, that $p@8 \ \wedge \ \neg P[p]$ holds continually after $p.7$ is executed. Note that, by the first conjunct of (I7), this implies that process q is not blocked at $q.8$ because $q \neq p$. Because $p@8 \ \wedge \ \neg P[p]$ continues to hold, by (I9), $p@8 \ \wedge \ \neg P[p] \ \wedge \ Q = p$ continues to hold. Hence, because process q is nonfaulty and does not become blocked at $q.8$, $q.11$ is eventually executed when $p@8 \ \wedge \ \neg P[p] \ \wedge \ Q = p$ holds, establishing $q@12 \ \wedge \ q.v = Q \ \wedge \ \neg P[p] \ \wedge \ Q = p$. Statement $q.12$ is then eventually executed, establishing $P[p]$. Thus, $p@8 \ \wedge \ P[p]$ holds at some state after $p.7$ is executed.

To conclude the proof, observe that once $p@8 \ \wedge \ P[p]$ holds, by (U2), $p@9$ eventually holds, because p is nonfaulty. Thus, Starvation-Freedom holds for the program of Figure 8. □

Second Algorithm

In this subsection, we prove that the algorithm of Figure 9 is correct. We begin by repeating the following lemma, which was proved in Section 4.4.

Lemma 2: For any statement s in the remainder or critical section, or $Acquire(N, k+1)$ or $Release(N, k+1)$ procedure of any process q, the following properties hold, where b ranges over $\{false, true\}$.

$\{q \neq p \ \wedge \ P[p] = (b, IN)\} \ q.s \ \{P[p] = (b, IN)\}$ (S1)

$\{q \neq p \ \wedge \ R[p] = (b, IN)\} \ q.s \ \{R[p] = (b, IN)\}$ (S2)

$\{q = p \ \wedge \ P[p] = (b, IN)\}\ q.s\ \{P[p] = (b, IN) \ \vee \ P[p].instance \neq IN\}$ (S3)
$\{q = p \ \wedge \ R[p] = (b, IN)\}\ q.s\ \{R[p] = (b, IN) \ \vee \ R[p].instance \neq IN\}$ (S4)
$\{P[p].instance \neq IN\}\ q.s\ \{P[p].instance \neq IN\}$ (S5)
$\{R[p].instance \neq IN\}\ q.s\ \{R[p].instance \neq IN\}$ (S6)
□

As before, we assume the following properties concerning $Acquire(N, k+1)$ and $Release(N, k+1)$, where the latter two are required to hold only if process p is nonfaulty and at most $k-1$ processes are faulty.

invariant $|\{q :: q@\{2..16\}\}| \leq k+1$ (I12)
$p@1$ *leads-to* $p@2$ (L5)
$p@17$ *leads-to* $p@0$ (L6)

Now that we have stated the assumptions that we require of the remainder and critical sections and $Acquire(N, k+1)$ and $Release(N, k+1)$ procedures, we proceed to prove that k-Exclusion and Starvation-Freedom hold for the program of Figure 9. We first prove a number of useful intermediate properties. The first three of these properties are straightforward and are stated without proof.

invariant $X = k - |\{p :: p@\{3..14\}\}|$ (I13)

invariant $Z = (\exists q :: q@\{4..8\})$ (I14)

invariant $|\{q :: q@\{4..8\}\}| \leq 1$ (I15)

Note that (I14) and (I15) must be proved together as a conjunction.

invariant $p@\{5, 6\} \Rightarrow p.v = Q$ (I16)

Proof: (I16) clearly holds initially. The antecedent of (I16) is established only by $p.4$, which also establishes the consequent. The consequent can be falsified while the antecedent holds only by $q.6$, where $q \neq p$. However, if the antecedent holds, then by (I15), statement $q.6$ is not enabled. □

invariant $p@6 \Rightarrow P[p.v] \neq (false, IN)$ (I17)

Proof: (I17) clearly holds initially. The antecedent is established only by statement $p.5$. If $p.5$ is executed when $P[v] = (false, IN)$ holds, then it establishes $P[v] = (true, IN)$. If $p.5$ is executed when $P[v] \neq (false, IN)$, then it does not modify $P[v]$. In either case, $P[v] \neq (false, IN)$ holds after the execution of $p.5$.

Now, consider statements that may falsify the consequent $P[p.v] \neq (false, IN)$. Note that this expression is equivalent to $(P[p.v] = (true, IN)) \vee (P[p.v].instance \neq IN)$. Let $r = p.v$. By (S1) and (S5), it follows that the consequent is not falsified by any statement within the remainder or critical section or $Acquire(N, k+1)$ or $Release(N, k+1)$ procedure of any process $q \neq r$. Furthermore, by (S3), if process r modifies $P[r]$ in one of these sections or procedures when $p.v = r$ holds, then it establishes $P[p.v].instance \neq IN$. The remaining statements to consider are $p.4$ and $p.15$, which may modify $p.v$, $r.7$, which may modify $P[p.v]$ (recall that $r = p.v$), and $q.5$, where $q.v = p.v$, which may also modify $P[p.v]$. However, the antecedent of (I17) is false after the execution of $p.4$ or $p.15$. By (I15), it is also false after the execution of $r.7$ or $q.5$, where $q \neq p$. If $q = p$, then $P[v] \neq (false, IN)$ holds after the execution of $q.5$, as explained in the previous paragraph. □

invariant $p@\{7, 8\} \Rightarrow Q = p$ (I18)

Proof: (I18) clearly holds initially. The antecedent is established only by statement $p.6$, which also establishes the consequent. The consequent can only be falsified by statement $q.6$, where $q \neq p$. However, by (I15), the antecedent of (I18) is false after the execution of $q.6$. □

invariant $p@8 \Rightarrow P[p] = (false, IN)$ (I19)

Proof: (I19) clearly holds initially. The antecedent is established only by $p.7$, which also establishes the consequent. If the antecedent holds, then by (S1), the consequent could potentially be falsified only by $q.5$, where $q \neq p$. However, by (I15), $q.5$ is not enabled when the antecedent holds. □

invariant $p@\{9..12\} \Rightarrow ((\neg Z \ \vee \ (\exists r :: r@\{4,5\})) \ \wedge \ Q = p) \ \vee \ P[p] \neq (false, IN)$ (I20)

Proof: (I20) clearly holds initially. The antecedent of (I20) is established only by statement $p.8$. By (I18), $p.8$ also establishes $\neg Z \ \wedge \ Q = p$. In the remainder of the proof, we consider the two disjuncts of the consequent. We show that if one disjunct is falsified while the antecedent holds, then the other disjunct holds.

We first dispose of the second disjunct, i.e., $P[p] \neq (false, IN)$. This expression is equivalent to $(P[p] = (true, IN)) \ \vee \ (P[p].instance \neq IN)$. By (S1) and (S5), this expression is not falsified by any statement within the remainder or critical section or $Acquire(N, k+1)$ or $Release(N, k+1)$ procedure of any process $q \neq p$. If it is falsified by process p in one of these sections or procedures, then the antecedent $p@\{9..12\}$ is false. The remaining statement to consider is $p.7$. However, the antecedent of (I20) is false after the execution of $p.7$.

We now consider the first disjunct, i.e., $(\neg Z \ \vee \ (\exists r :: r@\{4,5\})) \ \wedge \ Q = p$. This expression could potentially be falsified by any statement that establishes Z or $Q \neq p$, or that falsifies $(\exists r :: r@\{4,5\})$. We consider such statements in turn.

Z is established only by statement $q.3$ for some process q. However, if this statement is executed when $\neg Z \ \wedge \ Q = p$ holds, then it establishes $(\exists r :: r@\{4,5\}) \ \wedge \ Q = p$.

$Q \neq p$ is established only by statement $q.6$, where $q \neq p$. However, by (I16) and (I17), if $q@6 \ \wedge \ Q = p$ holds, then $P[p] \neq (false, IN)$ also holds, and $q.6$ does not falsify this expression.

Finally, $(\exists r :: r@\{4,5\})$ is falsified only by statement $r.5$. However, if this statement is executed when $(\exists r :: r@\{4,5\}) \ \wedge \ Q = p \ \wedge \ P[p] = (false, IN)$ holds, then by (I16), it establishes $P[p] = (true, IN)$. □

invariant $p@\{11,12\} \ \wedge \ P[p] = (false, IN) \Rightarrow Q = p$ (I21)

Proof: Follows directly from (I20). □

invariant $X < 0 \Rightarrow (\exists r :: r@\{3..8\} \ \vee \ (r@9 \ \wedge \ P[r] = (false, IN)) \ \vee$
$(r@\{10..12\} \ \wedge \ P[r] = (false, IN) \ \wedge \ R[r] = (false, IN)))$ (I22)

Proof: The antecedent of (I22) is initially false, and is established only by $p.2$. However, if $p.2$ establishes $X < 0$, then it also establishes $p@3$. In the remainder of the proof, we check those statements that may falsify a disjunct of the consequent. We show that if such a disjunct is falsified, then another disjunct holds or the antecedent is false.

The first disjunct, $r@\{3..8\}$, can only be falsified by statements $r.3$ and $r.8$. However, $r.3$ falsifies $r@\{3..8\}$ only if executed when $Z = true$, which by (I14), implies that $(\exists q :: q@\{4..8\})$ holds. Also, by (I19), $r.8$ establishes $r@9 \ \wedge \ P[r] = (false, IN)$.

By (S1), the second disjunct, $r@9 \ \wedge \ P[r] = (false, IN)$, can only be falsified by statements $r.9$ and $q.5$, where q is any process. (Note that process r could potentially falsify $P[r] = (false, IN)$ in its remainder or critical sections or $Acquire(N, k+1)$ or $Release(N, k+1)$ procedures, but in this case $r@9$ is false.) However, if $r.9$ is executed when $r@9 \ \wedge \ P[r] = (false, IN)$ holds, then it establishes $r@\{10..12\} \ \wedge \ P[r] = (false, IN) \ \wedge \ R[r] = (false, IN)$. Also, $q@\{3..8\}$ holds after the execution of $q.5$.

Finally, consider the third disjunct, i.e., $r@\{10..12\} \ \wedge \ P[r] = (false, IN) \ \wedge \ R[r] = (false, IN)$. By (S1) and (S2), this disjunct can only be falsified by $r.10$, $q.5$, and $q.16$, where q is any process. (As before, process r could potentially falsify $P[r] = (false, IN)$ or $R[r] = (false, IN)$ in its remainder or critical sections or $Acquire(N, k+1)$ or $Release(N, k+1)$ procedures, but in this case $r@\{10..12\}$ is false.) However, $r.10$ does not falsify $r@\{10..12\}$ while the antecedent holds. Also, $q@\{3..8\}$ holds after the execution of $q.5$, and by (I12) and (I13), $q.16$ is not enabled while the antecedent holds. □

The following invariant establishes the k-Exclusion property for the program of Figure 9.

invariant $|\{p :: p@13\}| \leq k$ (I23)

Proof: If $X \geq 0$, then by (I13), $|\{p :: p@\{3..9\}\}| \leq k$ holds, so (I23) holds. If $X < 0$, then by (I22), $(\exists p :: p@\{3..12\})$ holds, so by (I12), (I23) holds. □

The following unless property is used in the proof of Starvation-Freedom.

$p@\{11, 12\} \ \wedge \ (P[p] \neq (\mathit{false}, \mathit{IN}) \ \vee \ R[p] \neq (\mathit{false}, \mathit{IN})) \quad \mathit{unless} \quad p@13$ (U3)

Proof: By (S1), (S2), (S5), (S6), and the program text, neither $P[p] \neq (\mathit{false}, \mathit{IN})$ nor $R[p] \neq (\mathit{false}, \mathit{IN})$ can be falsified while $p@\{11, 12\}$ holds. □

Starvation-Freedom: If process p is nonfaulty and at most $k - 1$ processes are faulty, then $p@1$ *leads-to* $p@13$.

Proof: By (L5) and (L6), the only risk to Starvation-Freedom is that a nonfaulty process p is blocked forever at $p.11$ and $p.12$. Process p only reaches $p.11$ by executing $p.10$ when $X < 0$ holds. By (I13), this implies that $|\{p :: p@\{3..14\}\}| > k$ holds when $p.11$ is executed. By the assumption that at most $k - 1$ processes are faulty, this implies that there is a nonfaulty process $q \neq p$ such that $q@\{3..14\}$ holds when $p.10$ is executed.

We now show that $p@\{11, 12\} \ \wedge \ (P[p] \neq (\mathit{false}, \mathit{IN}) \ \vee \ R[p] \neq (\mathit{false}, \mathit{IN}))$ holds at some state after $p.10$ is executed. Assume, to the contrary, that $p@\{11, 12\} \ \wedge \ P[p] = (\mathit{false}, \mathit{IN}) \ \wedge \ R[p] = (\mathit{false}, \mathit{IN})$ holds continually after $p.10$ is executed. Note that, by (I21), this implies that process q is not blocked at $q.11$ and $q.12$ because $q \neq p$. Because $p@\{11, 12\} \ \wedge \ P[p] = (\mathit{false}, \mathit{IN}) \ \wedge \ R[p] = (\mathit{false}, \mathit{IN})$ continues to hold, by (I21), $p@\{11, 12\} \ \wedge \ P[p] = (\mathit{false}, \mathit{IN}) \ \wedge \ R[p] = (\mathit{false}, \mathit{IN}) \ \wedge \ Q = p$ continues to hold. Hence, because process q is nonfaulty and does not become blocked at $q.11$ and $q.12$, statement $q.15$ is eventually executed when $p@\{11, 12\} \wedge P[p] = (\mathit{false}, \mathit{IN}) \wedge R[p] = (\mathit{false}, \mathit{IN}) \wedge Q = p$ holds, establishing $q@16 \ \wedge \ q.v = Q \ \wedge \ R[p] = (\mathit{false}, \mathit{IN}) \ \wedge \ Q = p$. Statement $q.16$ is then eventually executed, establishing $R[p] = (\mathit{true}, \mathit{IN})$. Thus, $p@\{11, 12\} \ \wedge \ (P[p] \neq (\mathit{false}, \mathit{IN}) \ \vee \ R[p] \neq (\mathit{false}, \mathit{IN}))$ holds at some state after $p.10$ is executed.

To conclude the proof, note that once $p@\{11, 12\} \ \wedge \ (P[p] \neq (\mathit{false}, \mathit{IN}) \ \vee \ R[p] \neq (\mathit{false}, \mathit{IN}))$ holds, by (U3), $p@13$ eventually holds, because p is nonfaulty. Thus, Starvation-Freedom holds for the program of Figure 9. □