An Improved Lower Bound for the Time Complexity of Mutual Exclusion*

James H. Anderson and Yong-Jik Kim
Department of Computer Science
University of North Carolina at Chapel Hill

January 2002, Revised September 2002

Abstract

We establish a lower bound of $\Omega(\log N/\log\log N)$ remote memory references for N-process mutual exclusion algorithms based on reads, writes, or comparison primitives such as *test-and-set* and *compare-and-swap*. Our bound improves an earlier lower bound of $\Omega(\log\log N/\log\log\log N)$ established by Cypher. Our lower bound is of importance for two reasons. First, it *almost* matches the $\Theta(\log N)$ time complexity of the best known algorithms based on reads, writes, or comparison primitives. Second, our lower bound suggests that it is likely that, from an asymptotic standpoint, comparison primitives are no better than reads and writes when implementing local-spin mutual exclusion algorithms. Thus, comparison primitives may not be the best choice to provide in hardware if one is interested in scalable synchronization.

Keywords: Atomicity, local spinning, lower bounds, mutual exclusion, shared memory, time complexity

1 Introduction

Mutual exclusion algorithms [11] are used to resolve conflicting accesses to shared resources by asynchronous, concurrent processes. The problem of designing such an algorithm is widely regarded as one of the preeminent "classic" problems in concurrent programming. In the mutual exclusion problem, a process accesses the resource to be managed by executing a "critical section" of code. Activities not involving the resource occur within a corresponding "noncritical section." Before and after executing its critical section, a process executes two other code fragments, called "entry" and "exit" sections, respectively. A process may halt within its noncritical section but not within its critical section. The objective is to design the entry and exit sections so that the following requirements hold.[1]

- **Exclusion:** At most one process executes its critical section at any time.
- **Livelock-freedom:** If some process is in its entry section, then some process eventually enters its critical section. In addition, any process in its exit section eventually enters its noncritical section.

Often, livelock-freedom is replaced by the following stronger property.

- **Starvation-freedom:** If some process is in its entry (respectively, exit) section, then *that* process eventually enters its critical (respectively, noncritical) section.

*Work supported by NSF grants CCR 9732916, CCR 9972211, CCR 9988327, ITR 0082866, and CCR 0208289.

[1] Some authors use the term "deadlock-freedom" to denote the progress property termed livelock-freedom here. We prefer the latter term, because deadlock-freedom has been inconsistently used in the literature as both a safety property and a progress property. Also, some authors prefer "lockout-freedom" to starvation-freedom.

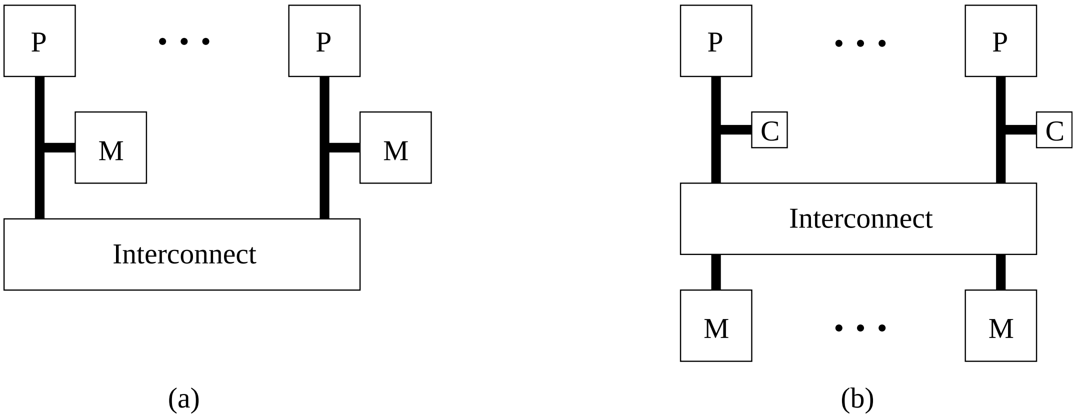

Figure 1: **(a)** DSM machine. Each variable is local to a specific process. **(b)** CC machine. No variable is local to any process. In both insets, 'P' denotes a processor, 'C' a cache, and 'M' a memory module.

Recent work on mutual exclusion has focused on the design of "scalable" algorithms that minimize contention for the processors-to-memory interconnection network through the use of *local spinning*. A mutual exclusion algorithm is *scalable* if its performance degrades only slightly as the number of contending processes increases. In this paper, we present a new time-complexity lower bound that pertains to local-spin mutual exclusion algorithms. (Under the time-complexity measure considered in this paper, all non-local-spin algorithms have unbounded time complexity.) Before describing our main contribution, we first present a brief overview of the concept of local spinning and relevant research on local-spin algorithms.

In local-spin mutual exclusion algorithms, good scalability is achieved by requiring all busy-waiting loops to be read-only loops in which only locally-accessible shared variables are accessed that do not access the processors-to-memory interconnection network. Two architectural paradigms have been considered in the literature that allow shared variables to be locally accessed: distributed shared-memory (DSM) machines and cache-coherent (CC) machines. Both are illustrated in Fig. 1. In a DSM machine, each processor has its own memory module that can be accessed without accessing the global interconnect. On such a machine, a shared variable is statically stored in a processor's memory module, and is local to that processor only. In a CC machine, each processor has a private cache, and some hardware protocol is used to enforce cache consistency. On such a machine, a shared variable is copied into a processor's local cache when accessed on that processor. (In the CC machine depicted in Fig. 1(b), each memory module is remote to all processors. However, in our proof, we allow for the possibility that some processors may have local memory modules.)

In this paper, we exclusively use the *RMR* (*remote-memory-reference*) *time complexity* measure. As its name suggests, only remote memory references that cause an interconnect traversal are counted under this measure. We will assess the RMR time complexity of an algorithm by counting the total number of remote memory references required by one process to enter and then exit its critical section once. An algorithm may have different RMR time complexities under the CC and DSM models because the notion of a remote memory reference differs under these two models. In the CC model, we assume that, once a spin variable has been cached, it remains cached until it is either updated or invalidated as a result of being modified by another process on a different processor. (Effectively, we are assuming an idealized cache of infinite size: a cached variable may be updated or invalidated but it is never replaced by another variable because of associativity or capacity limitations.)

Early work on local-spin mutual algorithms focused on the design of queue-based locks that are implemented using read-modify-write instructions such as *fetch-and-add* and *fetch-and-store*. Several such algorithms were proposed in which only $O(1)$ remote memory references are required for a process to enter and exit its critical section: the most well-known such algorithms are T. Anderson's algorithm [6], which uses *fetch-and-add*, Graunke and Thakkar's algorithm [12], which uses *fetch-and-store*, and Mellor-Crummey and Scott's algorithm, which uses *compare-and-swap* and *fetch-and-store* [14]. In each of these algorithms, blocked processes wait by local spinning within a "spin queue." When a process exits its critical section, it updates the spin variable of the process that follows it in the queue (if any) to allow that process to enter its critical section. A process enters the spin queue (if it must wait) by using a read-modify-write instruction to update a shared tail pointer; by using a read-modify-write instruction, the tail pointer can be updated and its old value returned in one atomic operation. Performance studies presented in several papers [6, 12, 14, 20] have shown that these queue-based algorithms scale well as contention increases.

In later work, Yang and Anderson initiated a study of time complexity bounds for local-spin synchronization algorithms, under the RMR time complexity measure [5, 20]. One of their main contributions was an N-process mutual exclusion algorithm with $\Theta(\log N)$ time complexity that uses only read and write operations [20]. In addition, they established trade-offs between time complexity and write- and access-contention for mutual exclusion algorithms [5]. The *write-contention* (*access-contention*) of a concurrent program is the number of processes that may be simultaneously enabled to write (access by reading and/or writing) the same shared variable. Yang and Anderson showed that, for any N-process mutual exclusion algorithm, if write-contention is w, and if at most v remote variables can be accessed by a single atomic operation, then there exists a single-process execution in which that process executes $\Omega(\log_{vw} N)$ remote operations for entry into its critical section. They further showed that, among these operations, $\Omega(\sqrt{\log_{vw} N})$ distinct remote variables are accessed; for algorithms with access-contention c, they showed that $\Omega(\log_{vc} N)$ distinct remote variables are accessed. In CC machines, the first access of a remote variable must cause a cache miss. Thus, by counting the number of distinct remote variables accessed, a lower bound for CC machines is obtained. Therefore, the last two of these bounds imply that $\Omega(\sqrt{\log_{vw} N})$ (or $\Omega(\log_{vc} N)$) remote operations are required in both DSM and CC machines. These results imply that a trade-off between contention and time complexity exists even if coherent caching techniques are employed.

Yang and Anderson's lower bounds are meaningful only for algorithms with limited write- or access-contention. In particular, they left open the question of whether $\Omega(\log N)$ is a lower bound for mutual exclusion under read/write atomicity with arbitrary write- and access-contention; such a bound would imply that their $\Theta(\log N)$ algorithm is time-optimal.

Cypher was the first to attempt to address this open question [10]. He established a lower bound of $\Omega(\log\log N/\log\log\log N)$, which is applicable to both DSM and CC systems. Surprisingly, his bound is applicable not only to algorithms based on reads and writes, but also to algorithms that use comparison primitives such as *test-and-set* and *compare-and-swap*.[2] Given that primitives such as *fetch-and-add* and *fetch-and-store* can be used to implement mutual exclusion in $O(1)$ time, this result pointed to an unexpected weakness of *compare-and-swap*, which is still widely regarded as being one of the most useful of all synchronization primitives to provide in hardware.

In this paper, we show that Cypher's lower bound can be improved to $\Omega(\log N/\log\log N)$. Thus, we have *almost* succeeded in establishing the optimality of Yang and Anderson's $\Theta(\log N)$ algorithm. Our result is stronger than Cypher's in that we allow atomic operations that may access many local shared variables (*i.e.*, variables that are statically associated to a process). Like Cypher's result, ours is applicable to algorithms that use comparison primitives (on remote variables), and is applicable to most DSM and CC systems. The only exception is a system with write-update caches[3] in which comparison primitives are supported, and with hardware capable of executing failed comparison primitives on cached remote variables without interconnection network traffic. We call such a system an LFCU ("Local Failed Comparison with write-Update") system. For LFCU systems, we show that $O(1)$ time complexity is possible using only *test-and-set* (the simplest of all comparison primitives).[4] While this result seemingly suggests that comparison primitives offer some advantages over reads and writes, it is worth noting that write-update protocols are almost never implemented in practice [16, page 721].

[2] A *comparison primitive* conditionally updates a shared variable after first testing that its value meets some condition.

[3] In a system with *write-update* caches, when a processor writes to a variable v that is also cached on other processors, a message is sent to these processors so that they can *update* v's value and maintain cache consistency.

[4] Although Cypher established a lower bound of $\Omega(\log\log N/\log\log\log N)$ for CC machines *with* comparison primitives, his cache-coherence model does not encompass write-update caches. According to his model, if a process p writes a variable v, then the first read of v by any other process q after p's write causes network traffic. Thus, if many processes read the value of v written by p, then each such read counts as a cache miss.

Our lower bound suggests that it is likely that, for non-LFCU systems, comparison primitives are no better from an asymptotic standpoint than reads and writes when implementing local-spin mutual exclusion algorithms. Moreover, the time complexity gap that exists in such systems between comparison primitives and primitives such as *fetch-and-add* and *fetch-and-store* is actually quite wide. Thus, comparison primitives may not be the best choice to provide in hardware if one is interested in scalable synchronization.

The rest of the paper is organized as follows. In Sec. 2, we present our model of shared-memory systems. The key ideas of our lower bound proof are then sketched in Sec. 3. In Sec. 4, the proof is presented in detail. The $O(1)$ algorithm for LFCU systems mentioned above is then presented in Sec. 5. Concluding remarks appear in Sec. 6.

2 Definitions

In the following subsections, we define our model of a shared-memory system (Sec. 2.1), state the properties required of a mutual exclusion algorithm implemented within this model (Sec. 2.2), and present a categorization of events that allows us to accurately deduce the network traffic generated by an algorithm in a system with coherent caches (Sec. 2.3).

2.1 Shared-memory Systems

Our model of a shared-memory system is similar to that used by Anderson and Yang [5].

A *shared-memory system* $\mathcal{S} = (C, P, V)$ consists of a set of computations C, a set of processes P, and a set of variables V. A *computation* is a finite sequence of events. To complete the definition of a shared-memory system, we must define the notion of an "event" and state the requirements to which events and computations are subject. This is done in the remainder of this subsection. As needed terms are defined, various notational conventions are also introduced that will be used in the rest of the paper.

Informally, an *event* is a particular execution of an atomic statement of some process that involves reading and/or writing of one or more variables. Each variable is *local* to at most one process and is *remote* to all other processes. (Note that we allow variables that are remote to *all* processes; thus, our model applies to both DSM and CC systems.) The locality relationship is static, *i.e.*, it does not change during a computation. A local variable may be shared; that is, a process may access local variables of other processes. An *initial value* is associated with each variable. An event is *local* if it does not access any remote variables, and is *remote* otherwise.

Events, informally considered. Below, formal definitions pertaining to events are given; here, we present an informal discussion to motivate these definitions. An event is executed by a particular process, and may access at most one variable that is remote to that process (by reading, writing, or executing a comparison primitive), plus any number of local (shared) variables.[5] Thus, we allow arbitrarily powerful operations on local variables. As usually defined, a *comparison primitive* is an atomic operation on a shared variable v expressible using the following pseudo-code.

$compare\_and\_fg(v, old, new)$
  $temp := v$;
  if $v = old$ **then** $v := f(old, new)$ **fi**;
  return $g(temp, old, new)$

For example, *compare-and-swap* can be defined by defining $f(old,\ new) = new$ and $g(temp,\ old,\ new) = (temp = old)$. We call an execution of such a primitive a *comparison event*. As we shall see, our formal

[5]We do not distinguish between private and shared variables in our model. In an actual algorithm, some variables local to a process might be private and others shared.

definition of a comparison event, which is given later in this section, generalizes the functionality encompassed by the pseudo-code above by allowing arbitrarily many local shared variables to be accessed.

As an example, assume that variables a, b, and c are local to process p and variables x and y are remote to p. Then, the following atomic statements by p are allowed in our model:

statement $s1$:	$a := a + 1;\ b := c + 1;$
statement $s2$:	$a := x;$
statement $s3$:	$y := a + b;$
statement $s4$:	$\mathit{compare\text{-}and\text{-}swap}(x, 0, b)$

For example, if every variable has an initial value of 0, and if these four statements are executed in order, then we will have the following four events:

$e1$:	p reads 0 from a, writes 1 to a, reads 0 from c, and writes 1 to b;	/* local event */
$e2$:	p reads 0 from x and writes 0 to a;	/* remote read from x */
$e3$:	p reads 0 from a, reads 1 from b, and writes 1 to y;	/* remote write to y */
$e4$:	p reads 0 from x, reads 1 from b, and writes 1 to x	/* comparison primitive execution on x */

On the other hand, the following atomic statements by p are not allowed in our model, because $s5$ accesses two remote variables at once, and $s6$ and $s7$ cannot be expressed as a comparison primitive.

statement $s5$:	$x := y;$	/* accesses two remote variables */
statement $s6$:	$a := x;\ x := 1;$	/* *fetch-and-store* (*swap*) on a remote variable */
statement $s7$:	$x := x + b$	/* *fetch-and-add* on a remote variable */

Describing each event as in the preceding examples is inconvenient, ambiguous, and prone to error. For example, if statement $s7$ is executed when $x = 0 \wedge b = 1$ holds, then the resulting event can be described in the same way as $e4$ is. (Thus, $e4$ is allowed as an execution of $s4$, yet disallowed as an execution of $s7$.) In order to systematically represent the class of allowed events, we need a more refined formalism.

Definitions pertaining to events. An *event* e is denoted $[\mathsf{Op}, R, W, p]$, where $p \in P$ (the set of processes). We call Op the *operation* of event e, denoted $op(e)$. Op can be one of the following: $\perp$, $\mathsf{read}(v)$, $\mathsf{write}(v)$, or $\mathsf{compare}(v, \alpha)$, where v is a variable in V and α is a value from the value domain of v. (Informally, e can be a local event, a remote read, a remote write, or an execution of a comparison primitive. The precise definition of these terms is given below.)

The sets R and W consist of pairs (v, α), where $v \in V$. This notation represents an event of process p that reads the value α from variable v for each element $(v, \alpha) \in R$, and writes the value α to variable v for each element $(v, \alpha) \in W$. Each variable in R is assumed to be distinct; the same is true for W. We define $Rvar(e)$, the set of variables read by e, to be $\{v\colon (v, \alpha) \in R\}$, and $Wvar(e)$, the set of variables written by e, to be $\{v\colon (v, \alpha) \in W\}$. We also define $var(e)$, the set of all variables accessed by e, to be $Rvar(e) \cup Wvar(e)$. We say that an event e *writes* (respectively, *reads*) a variable v if $v \in Wvar(e)$ (respectively, $v \in Rvar(e)$) holds, and that it *accesses* any variable that it writes or reads. We also say that a computation H *contains a write* (respectively, *read*) of v if H contains some event that writes (respectively, reads) v. Finally, we say that process p is the *owner* of $e = [\mathsf{Op}, R, W, p]$, denoted $owner(e) = p$. For brevity, we sometimes use e_p to denote an event owned by process p.

Our lower bound is dependent on the Atomicity Property stated below. This assumption requires each remote event to be an atomic read operation, an atomic write operation, or a comparison-primitive execution.

Atomicity Property: Each event $e = [\mathsf{Op}, R, W, p]$ must satisfy one of the conditions below.

- If $\mathsf{Op} = \perp$, then e does not access any remote variables. (That is, all variables in $var(e)$ are local to p.) In this case, we call e a *local event*.

- If $\mathsf{Op} = \mathsf{read}(v)$, then e reads exactly one remote variable, which must be v, and does not write any remote variable. (That is, $(v, \alpha) \in R$ holds for some α, v is not in $Wvar(e)$, and all other variables [if any] in $var(e)$ are local to p.) In this case, e is called a *remote read event.*

- If $\mathsf{Op} = \mathsf{write}(v)$, then e writes exactly one remote variable, which must be v, and does not read any remote variable. (That is, $(v, \alpha) \in W$ holds for some α, v is not in $Rvar(e)$, and all other variables [if any] in $var(e)$ are local to p.) In this case, e is called a *remote write event.*

- If $\mathsf{Op} = \mathsf{compare}(v, \alpha)$, then e reads exactly one remote variable, which must be v. We say that e is a *comparison event* in this case. Comparison events must be either successful or unsuccessful.

  - e is a *successful comparison event* if the following hold: $(v, \alpha) \in R$ (*i.e.*, e reads the value α from v), and $(v, \beta) \in W$ for some $\beta \neq \alpha$ (*i.e.*, e writes to v a value *different* from α).
  - e is an *unsuccessful comparison event* if e does not write v, *i.e.*, $v \notin Wvar(e)$ holds.

  In either case, e does not write any other remote variable. □

Our notion of an unsuccessful comparison event includes both comparison-primitive invocations that fail (*i.e.*, $v \neq old$ in the pseudo-code given for *compare_and_fg* above) and also those that do not fail but leave the remote variable that is accessed unchanged (*i.e.*, $v = old \ \wedge \ v = f(old, new)$). In the latter case, we simply assume that the remote variable v is not written. We categorize both cases as unsuccessful comparison events because this allows us to simplify certain cases in the proofs in Sec. 4. (On the other hand, we allow a remote write event on v to preserve the value of v, *i.e.*, to write the same value as v had before the event.)

Note that the Atomicity Property allows arbitrarily powerful operations on local (shared) variables. For example, if variable v, ranging over $\{0, \ldots, 10\}$, is remote to process p, and arrays $a[1..10]$ and $b[1..10]$ are local to p, then an execution of the following statement is a valid event by p with operation $\mathsf{compare}(v, 0)$.

$$
\begin{array}{l}
\textbf{if } v = 0 \textbf{ then} \\
\qquad v := \left(\sum_{j=1}^{10} a[j]\right) \textbf{ mod } 11; \\
\qquad \textbf{for } j := 1 \textbf{ to } 10 \textbf{ do } a[j] := b[j] \textbf{ od} \\
\textbf{else} \\
\qquad \textbf{for } j := 1 \textbf{ to } v \textbf{ do } b[j] := a[j] + v \textbf{ od} \\
\textbf{fi}
\end{array}
$$

In this case, $Wvar(e)$ is $\{v, a[1..10]\}$ if e reads $v = 0$ and writes a nonzero value (*i.e.*, e is a successful comparison event), $\{a[1..10]\}$ if e reads and writes $v = 0$, and $\{b[1..v]\}$ if e reads a value between 1 and 10 from v.

It is important to note that, saying that an event e_p writes (reads) a variable v is *not* equivalent to saying that e_p is a remote write (read) operation on v; e_p may also write (read) v if v is local to process p, or if p is a comparison event that accesses v.

We say that two events $e = [\mathsf{Op}, R, W, p]$ and $f = [\mathsf{Op}', R', W', q]$ are *congruent*, denoted $e \sim f$, if and only if the following conditions are met.

- $p = q$;
- $\mathsf{Op} = \mathsf{Op}'$, where equality means that both operations are the same *with the same arguments* (v and/or α).

Informally, two events are congruent if they execute the same operation on the same remote variable. For read and write events, the values read or written may be different. For comparison events, the values read or written (if successful) may be different, but the parameter α must be the same. (It is possible that a successful comparison operation is congruent to an unsuccessful one.) Note that e and f may access different *local* variables.

Definitions pertaining to computations. The definitions given until now have mostly focused on events. We now present requirements and definitions pertaining to computations.

The value of variable v at the end of computation H, denoted $value(v, H)$, is the last value written to v in H (or the initial value of v if v is not written in H). The last event to write to v in H is denoted $last\_writer\_event(v, H)$,[6] and its owner is denoted $last\_writer(v, H)$. If v is not written by any event in H, then we let $last\_writer(v, H) = \perp$ and $last\_writer\_event(v, H) = \perp$.

We use $\langle e, \ldots \rangle$ to denote a computation that begins with the event e, $\langle e, \ldots, f \rangle$ to denote a computation beginning with event e and ending with event f, and $\langle\rangle$ to denote the empty computation. We use $H \circ G$ to denote the computation obtained by concatenating computations H and G. An *extension* of computation H is a computation of which H is a prefix. For a computation H and a set of processes Y, $H \mid Y$ denotes the subcomputation of H that contains all events in H of processes in Y.[7] Computations H and G are *equivalent with respect to* Y if and only if $H \mid Y = G \mid Y$. A computation H is a Y*-computation* if and only if $H = H \mid Y$. For simplicity, we abbreviate the preceding definitions when applied to a singleton set of processes. For example, if $Y = \{p\}$, then we use $H \mid p$ to mean $H \mid \{p\}$ and p-computation to mean $\{p\}$-computation. Two computations H and G are *congruent*, denoted $H \sim G$, if either both H and G are empty, or if $H = \langle e \rangle \circ H'$ and $G = \langle f \rangle \circ G'$, where $e \sim f$ and $H' \sim G'$.

Until this point, we have placed no restrictions on the set of computations C of a shared-memory system $\mathcal{S} = (C, P, V)$ (other than restrictions pertaining to the kinds of events that are allowed in an individual computation). The restrictions we require are as follows.

P1: If $H \in C$ and G is a prefix of H, then $G \in C$.
— *Informally, every prefix of a valid computation is also a valid computation.*

P2: If $H \circ \langle e_p \rangle \in C$, $G \in C$, $G \mid p = H \mid p$, and if $value(v, G) = value(v, H)$ holds for all $v \in Rvar(e_p)$, then $G \circ \langle e_p \rangle \in C$.
— *Informally, if two computations H and G are not distinguishable to process p, if p can execute event e_p after H, and if all variables in $Rvar(e_p)$ have the same values after H and G, then p can execute e_p after G.*

P3: If $H \circ \langle e_p \rangle \in C$, $G \in C$, and $G \mid p = H \mid p$, then $G \circ \langle e'_p \rangle \in C$ for some event e'_p such that $e_p \sim e'_p$.
— *Informally, if two computations H and G are not distinguishable to process p, and if p can execute event e_p after H, then p can execute a congruent operation after G.*

P4: For any $H \in C$, $H \circ \langle e_p \rangle \in C$ implies that $\alpha = value(v, H)$ holds, for all $(v, \alpha) \in Rvar(e_p)$.
— *Informally, only the last value written to a variable can be read.*

P5: For any $H \in C$, if both $H \circ \langle e_p \rangle \in C$ and $H \circ \langle e'_p \rangle \in C$ hold for two events e_p and e'_p, then $e_p = e'_p$.
— *Informally, each process is deterministic. This property is included in order to simplify bookkeeping in our proofs.*

Property P3 precisely defines the class of allowed events. In particular, if process p is enabled to execute a certain statement, then that statement must generate the same operation regardless of the execution of other processes. For example, if a is a local *shared* variable and x and y are remote variables, then the following statement is *not* allowed.

statement $s8$: **if** $a = 0$ **then** $x := 1$ **else** $y := 1$ **fi**

This is because the event generated by $s8$ may have either $\mathsf{write}(x)$ or $\mathsf{write}(y)$ as its operation, depending on the value possibly written to a by other processes.

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

[7] The subcomputation $H \mid Y$ is not necessarily a valid computation in a given system $\mathcal{S}$, that is, an element of C. However, we can always consider $H \mid Y$ to be a computation in a technical sense, *i.e.*, it is a sequence of events.

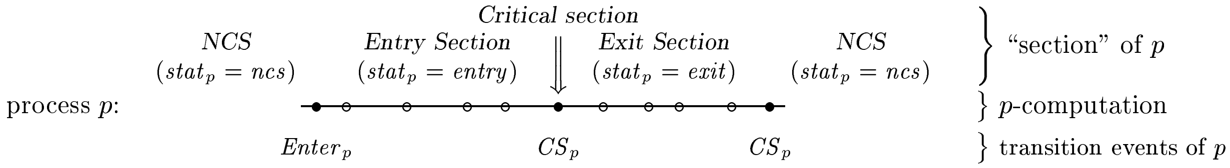

Figure 2: Transition events of a mutual exclusion system. In this figure, NCS stands for "noncritical section," a circle ($\circ$) represents a non-transition event, and a bullet ($\bullet$) represents a transition event.

2.2 Mutual-exclusion Systems

We now define a special kind of shared-memory system, namely mutual exclusion systems, which are our main interest. A *mutual exclusion system* $\mathcal{S} = (C, P, V)$ is a shared-memory system that satisfies the properties below.

Each process p has a local auxiliary variable $stat_p$ that represents which section in the mutual exclusion algorithm p is currently in: $stat_p$ ranges over ncs (for noncritical section), $entry$, or $exit$, and is initially ncs. (For simplicity, we assume that each critical-section execution is vacuous.) Process p also has three "dummy" auxiliary variables ncs_p, $entry_p$, and $exit_p$. These variables are accessed only by the following events.

$$
\begin{aligned}
Enter_p &= [\mathsf{write}(entry_p),\ \{\},\ \{(stat_p, entry),\ (entry_p, 0)\},\ p] \\
CS_p &= [\mathsf{write}(exit_p),\ \{\},\ \{(stat_p, exit),\ (exit_p, 0)\},\ p] \\
Exit_p &= [\mathsf{write}(ncs_p),\ \{\},\ \{(stat_p, ncs),\ (ncs_p, 0)\},\ p]
\end{aligned}
$$

Event $Enter_p$ cause p to transit from its noncritical section to its entry section. Event CS_p cause p to transit from its entry section to its exit section.[8] Event $Exit_p$ cause p to transit from its exit section to its noncritical section. This behavior is depicted in Fig. 2.

We define variables $entry_p$, $exit_p$, and ncs_p to be remote to all processes. This assumption allows us to simplify bookkeeping, because it implies that each of $Enter_p$, CS_p, and $Exit_p$ is congruent only to itself. (This is the sole purpose of defining these three variables.)

The allowable transitions of $stat_p$ are as follows: for all $H \in C$,

$$
\begin{array}{rcll}
H \circ \langle Enter_p \rangle \in C & \text{if and only if} & value(stat_p, H) = ncs, & \\
H \circ \langle CS_p \rangle \in C & \text{only if} & value(stat_p, H) = entry, & \text{and} \\
H \circ \langle Exit_p \rangle \in C & \text{only if} & value(stat_p, H) = exit. &
\end{array}
$$

In our proof, we only consider computations in which each process enters and then exits its critical section at most once. Thus, we henceforth assume that each computation contains at most one $Enter_p$ event for each process p. In addition, a mutual exclusion system is required to satisfy the following.

Exclusion: For all $H \in C$, if both $H \circ \langle CS_p \rangle \in C$ and $H \circ \langle CS_q \rangle \in C$ hold, then $p = q$.

Progress: Given $H \in C$, let $X = \{q \in P\colon value(stat_q, H) \neq ncs\}$. If X is nonempty, then there exists an X-computation G such that $H \circ G \circ \langle e_p \rangle \in C$, where $p \in X$ and e_p is either CS_p (if $value(stat_p, H) = entry$) or $Exit_p$ (if $value(stat_p, H) = exit$).

The Exclusion property is equivalent to (mutual) exclusion, which was informally defined in Sec. 1. Although we assume that each critical-section execution is vacuous, we can certainly "augment" the algorithm by replacing each event CS_p by a set of events that represents p's critical-section execution. If two events CS_p and CS_q are simultaneously "enabled" after a computation H, then we can interleave the critical-section executions of p and q, thus violating mutual exclusion. The Exclusion property states that such a situation does not arise.

[8] Each critical-section execution of p is captured by the single event CS_p, so $stat_p$ changes directly from $entry$ to $exit$.

The Progress property is implied by livelock-freedom, although it is strictly weaker than livelock-freedom. In particular, it allows the possibility of infinitely extending H such that no active process p executes CS_p or $Exit_p$. This weaker formalism is sufficient for our purposes.

2.3 Cache-coherent Systems

On cache-coherent shared-memory systems, some remote-variable accesses may be handled locally, without causing interconnection network traffic. Our lower bound proof applies to such systems without modification. This is because we do not count every remote event, but only certain "critical" events that generate cache misses. (Actually, as explained below, some events that we consider critical might not generate cache misses in certain system implementations, but this has no asymptotic impact on our proof.)

Precisely defining the class of such events in a way that is applicable to the myriad of cache implementations that exist is exceedingly difficult. We partially circumvent this problem by assuming idealized caches of infinite size: a cached variable may be updated or invalidated but it is never replaced by another variable. Note that, in practice, cache size and associativity limitations should only *increase* the number of cache misses. In addition, in order to keep the proof manageable, we allow cache misses to be both undercounted *and* overcounted. In particular, as explained below, in any realistic cache system, at least a constant fraction (but not necessarily all) of all critical events generate cache misses. Thus, a single cache miss may be associated with $\Theta(1)$ critical events, resulting in overcounting up to a constant factor. Note that this overcounting has no effect on our asymptotic lower bound. Also, an event that generates a cache miss may be considered noncritical, resulting in undercounting, which may be of more than a constant factor. Note that this undercounting can only strengthen our asymptotic lower bound. Therefore, an asymptotic lower bound on the number of critical events is also an asymptotic lower bound on the number of actual cache misses.

Our definition of a critical event is given below. This definition is followed by a rather detailed explanation in which various kinds of caching protocols are considered.

Definition: Let $\mathcal{S} = (C, P, V)$ be a mutual exclusion system. Let e_p be an event in $H \in C$. Then, we can write H as $F \circ \langle e_p \rangle \circ G$, where F and G are subcomputations of H. We say that e_p is a *critical event* in H if and only if one of the following conditions holds:

Transition event: e_p is one of $Enter_p$, CS_p, or $Exit_p$.

Critical read: There exists a variable v, remote to p, such that $op(e_p) = \mathsf{read}(v)$ and $F \,|\, p$ does not contain a read from v.
— *Informally, e_p is the first event of p that reads v in H.*

Critical write: There exists a variable v, remote to p, such that e_p is a remote write event on v (*i.e.*, $op(e_p) = \mathsf{write}(v)$), and $last\_writer(v, F) \neq p$.
— *Informally, e_p is a remote write event on v, and either e_p is the first event that writes to v in H (i.e., $last\_writer(v, F) = \perp$), or e_p overwrites a value that was written by another process.*

Critical successful comparison: There exists a variable v, remote to p, such that e_p is a successful comparison event on v (*i.e.*, $op(e_p) = \mathsf{compare}(v, \alpha)$ for some value of α and $v \in Wvar(e_p)$), and $last\_writer(v, F) \neq p$.
— *Informally, e_p is a successful comparison event on v, and either e_p is the first event that writes to v in H (i.e., $last\_writer(v, F) = \perp$), or e_p overwrites a value that was written by another process.*

Critical unsuccessful comparison: There exists a variable v, remote to p, such that e_p is an unsuccessful comparison event on v (*i.e.*, $op(e_p) = \mathsf{compare}(v, \alpha)$ for some value of α and $v \notin Wvar(e_p)$), $last\_writer(v, F) \neq p$, and either
(i) $F \,|\, p$ does not contain an unsuccessful comparison event on v, or
(ii) F can be written as $F_1 \circ \langle f_q \rangle \circ F_2$, where $f_q = last\_writer\_event(v, F)$, such that $F_2 \,|\, p$ does not contain an unsuccessful comparison event on v.

— *Informally, e_p must read the initial value of v (if $last\_writer(v, F) = \perp$) or a value that is written by another process q. Moreover, either* **(i)** *e_p is the first unsuccessful comparison on v by p in H, or* **(ii)** *e_p is the first such event by p after some other process has written to v (via f_q).*[9] □

Note that state transition events do *not* actually cause cache misses; these events are defined as critical events because this allows us to combine certain cases in the proofs that follow. A process executes only three transition events per critical-section execution, so defining transition events as critical does not affect our asymptotic lower bound.

Note that it is possible that the first read of v by p, the first write or successful comparison event on v by p, *and* the first unsuccessful comparison event on v by p (*i.e.*, Case (i) in the definition above) are all considered critical. Depending on the system implementation, the second and third of these events to occur might not generate a cache miss. However, even in such a case, the first such event will always generate a cache miss, and hence at least a third of all such "first" critical events will actually incur real interconnect traffic. Hence, considering all of these events to be critical has no asymptotic impact on our lower bound.

All caching protocols are based on either a *write-through* or a *write-back* scheme. In a write-through scheme, all writes go directly to shared memory. In a write-back scheme, a remote write to a variable v may create a cached copy of v, so that subsequent writes to v do not cause cache misses. With either scheme, if cached copies of v exist on other processors when such a write occurs, then to ensure consistency, these cached copies must be either *invalidated* or *updated*. In the rest of this subsection, we consider in some detail the question of whether our notion of a critical write and a critical comparison is reasonable under the various caching protocols that arise from these definitions.

First, consider a system in which there are no comparison events, in which case it is enough to consider only critical write events. If a write-through scheme is used, then all remote write events cause interconnect traffic, so consider a write-back scheme. In this case, a write e_p to a remote variable that is not the first write to v by p is considered critical only if $last\_writer(v, F) = q$ holds for some $q \neq p$, which implies that v is stored in a local cache line of process q. (Since all caches are assumed to be infinite, $last\_writer(v, F) = q$ implies that q's cached copy of v has not been invalidated.) In such a case, e_p must either invalidate or update the cached copy of v (depending on the means for ensuring consistency), thereby generating interconnect traffic.

Next, consider comparison events. A successful comparison event on a remote variable v writes a new value to v. Thus, the reasoning given above for ordinary writes applies to successful comparison events as well. This leaves only unsuccessful comparison events. Recall that an unsuccessful comparison event on a remote variable v does not actually write v. Thus, the reasoning above does *not* apply to such events.

In the remainder of this discussion, let e_p denote an unsuccessful comparison event on a remote variable v, where Case (ii) in the definition applies. Then, some other process q writes to v (via a write or successful comparison event, or even a local, read, or unsuccessful comparison event, if v is local to q) prior to e_p but after p's most recent unsuccessful comparison event on v, and also after p's most recent successful comparison and/or remote write event on v. Consider the interconnect traffic generated, assuming an invalidation scheme for ensuring cache consistency. In this case, p's previous cached copy of v is invalidated prior to e_p, so e_p must generate interconnect traffic in order to read the current value of v, unless an earlier read of v by p (after q's write) exists. Thus, e_p fails to generate interconnect traffic only if there is an earlier read of v by p (after q's write), say f_p, that does. Note that f_p is either a "first" read of v by p or a noncritical read. The former case may happen at most once per remote variable; in the latter case, we can "charge" the interconnect traffic generated by f_p to e_p.

The last possibility to consider is that of an unsuccessful comparison event e_p implemented within a caching protocol that uses updates to ensure consistency. In this case, q's write in the scenario above updates p's cached copy, and hence e_p may not generate interconnect traffic. (Note that, for interconnect traffic to be avoided in this case, the hardware must be able to distinguish a failed comparison event on a cached variable from a successful comparison event or a failed comparison on a non-cached variable.)

[9] This definition is more complicated than those for critical writes and successful comparisons because an unsuccessful comparison event on v by p does not actually write v. Thus, if a sequence of such events is performed by p while v is not written by other processes, then only the first such event should be considered critical.

Therefore, our lower bound does *not* apply to a system that uses updates to ensure consistency and that has the ability to execute failed comparison events on cached variables without generating interconnect traffic. (If an update scheme is used, but the hardware is incapable of avoiding interconnect traffic when executing such failed comparison events, then our lower bound obviously still applies.) Such systems were termed LFCU ("Local Failed Comparison with write-Update") systems earlier in Sec. 1. As mentioned there, an algorithm with $O(1)$ time complexity in such systems is presented in Sec. 5. This algorithm shows that LFCU systems fundamentally *must* be excluded from our proof.

As a final comment on our notion of a critical event, notice that whether an event is considered critical depends on the particular computation that contains the event, specifically the prefix of the computation preceding the event. Therefore, when saying that an event is (or is not) critical, the computation containing the event must be specified.

3 Lower-bound Proof Strategy

In Sec. 4, we show that for any mutual exclusion system $\mathcal{S} = (C, P, V)$, there exists a computation H such that some process p executes $\Omega(\log N / \log \log N)$ critical events to enter and exit its critical section, where $N = |P|$. In this section, we sketch the key ideas of the proof.

3.1 Process Groups and Regular Computations

Our proof focuses on a special class of computations called "regular" computations. A regular computation consists of events of two groups of processes, "active processes" and "finished processes." Informally, an active process is a process in its entry section, competing with other active processes; a finished process is a process that has executed its critical section once, and is in its noncritical section. (Recall that we only consider computations in which each process executes is critical section at most once.) These properties follow from Condition RF4, given later in this section.

Definition: Let $\mathcal{S} = (C, P, V)$ be a mutual exclusion system, and H be a computation in C. We define $\mathrm{Act}(H)$, the set of *active processes* in H, and $\mathrm{Fin}(H)$, the set of *finished processes* in H, as follows.

$$\begin{aligned}\mathrm{Act}(H) &= \{p \in P \colon H \,|\, p \neq \langle\rangle \text{ and } \langle \mathit{Exit}_p \rangle \text{ is } \textit{not} \text{ in } H\} \\ \mathrm{Fin}(H) &= \{p \in P \colon H \,|\, p \neq \langle\rangle \text{ and } \langle \mathit{Exit}_p \rangle \text{ is in } H\}\end{aligned}$$ □

In Sec. 3.2, a detailed overview of our proof is given. Here, we give cursory overview, so that the definitions that follow will make sense. Initially, we start with a regular computation in which all the processes in P are active. The proof proceeds by inductively constructing longer and longer regular computations, until the desired lower bound is attained. The regularity condition defined below ensures that no participating process has "knowledge" of any other process that is active.[10] This has two consequences: we can "erase" any active process (*i.e.*, remove its events from the computation) and still get a valid computation; "most" active processes have a "next" non-transition critical event. In each induction step, we append to each of the n active processes (except at most one) one next critical event. These next critical events may introduce unwanted information flow, *i.e.*, these events may cause an active process to acquire knowledge of another active process, resulting in a non-regular computation. Informally, such information flow is problematic because an active process p that learns of another active process may start busy waiting. If p busy waits via a local spin loop, then it might *not* execute any more critical events, in which case the induction fails.

In some cases, we can eliminate all information flow by simply erasing some active processes. Sometimes erasing alone does not leave enough active processes for the next induction step. In this case, we partition the active processes into two categories: "invisible" processes and "promoted" processes. The invisible processes (that are not erased — see below) will constitute the set of active processes for the next regular computation in the induction. No process is allowed to have knowledge of another process that is invisible. The promoted processes are processes that have been selected to "roll forward." A process that is rolled

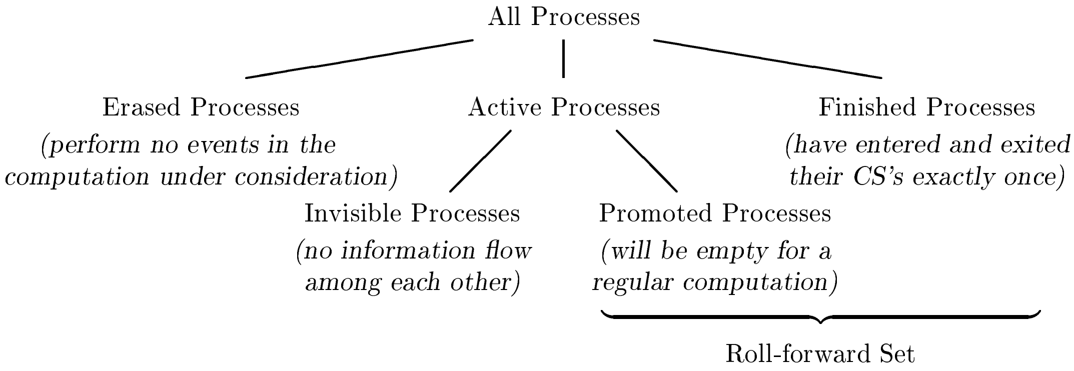

Figure 3: Process groups.

forward finishes executing its entry, critical, and exit sections, and returns to its noncritical section. (Both of these techniques, erasing and rolling forward, have been used previously to prove several other lower bounds for concurrent systems [1, 8, 18].) Processes *are* allowed to have knowledge of promoted or finished processes. Although invisible processes may have knowledge of promoted processes, once all promoted processes have finished execution, the regularity condition holds again (*i.e.*, all active processes are invisible). The various process groups we consider are depicted in Fig. 3 (the roll-forward set is discussed below).

The promoted and finished processes together constitute a "roll-forward set," which must meet Conditions RF1–RF5 below. Informally, Condition RF1 ensures that an invisible process is not known to any other processes; RF2 and RF3 bound the number of possible conflicts caused by appending a critical event; RF4 ensures that the invisible, promoted, and finished processes behave as explained above; RF5 ensures that we can erase any invisible process, maintaining that critical events (that are not erased) remain critical.

Definition: Let $\mathcal{S} = (C, P, V)$ be a mutual exclusion system, H be a computation in C, and RFS be a subset of P such that $\mathrm{Fin}(H) \subseteq RFS$ and $H \,|\, p \neq \langle\rangle$ for each $p \in RFS$. We say that RFS is a valid *roll-forward set* (*RF-set*) of H if and only if the following conditions hold.

RF1: Assume that H can be written as $E \circ \langle e_p \rangle \circ F \circ \langle f_q \rangle \circ G$.[11] If $p \neq q$ and there exists a variable $v \in \mathit{Wvar}(e_p) \cap \mathit{Rvar}(f_q)$ such that F does not contain a write to v (*i.e.*, $\mathit{last\_writer\_event}(v, F) = \perp$), then $p \in RFS$ holds.
— *Informally, if a process p writes to a variable v, and if another process q reads* that *value from v without any intervening write to v, then $p \in RFS$ holds.*

RF2: For any event e_p in H and any variable v in $\mathit{var}(e_p)$, if v is local to another process q ($\neq p$), then either $q \notin \mathrm{Act}(H)$ or $\{p, q\} \subseteq RFS$ holds.
— *Informally, if a process p accesses a variable that is local to another process q, then either q is not an active process in H, or both p and q belong to the roll-forward set RFS. Note that this condition does not distinguish whether q actually accesses v or not, and conservatively requires q to be in RFS (or erased) even if q does not access v. This is done in order to simplify bookkeeping.*

RF3: Consider a variable $v \in V$ and two different events e_p and f_q in H. Assume that both p and q are in $\mathrm{Act}(H)$, $p \neq q$, there exists a variable v such that $v \in \mathit{var}(e_p) \cap \mathit{var}(f_q)$, and there exists a write to v in H. Then, $\mathit{last\_writer}(v, H) \in RFS$ holds.
— *Informally, if a variable v is accessed by more than one processes in $\mathrm{Act}(H)$, then the last process in H to write to v (if any) belongs to RFS.*

[10] A process p has knowledge of another process q if p has read from some variable a value that is written either by q or another process that has knowledge of q.

[11] Here and in similar sentences hereafter, we are considering *every* way in which H can be so decomposed. That is, any pair of events e_p and f_q inside H such that e_p comes before f_q defines a decomposition of H into $E \circ \langle e_p \rangle \circ F \circ \langle f_q \rangle \circ G$, and RF1 must hold for any such decomposition.

RF4: For any process p such that $H \mid p \neq \langle\rangle$,

$$value(stat_p,\ H) = \begin{cases} entry & \text{if } p \in \mathrm{Act}(H) - RFS \\ entry \text{ or } exit & \text{if } p \in \mathrm{Act}(H) \cap RFS \\ ncs & \text{otherwise } (i.e.,\ p \in \mathrm{Fin}(H)). \end{cases}$$

Moreover, if $p \in \mathrm{Fin}(H)$, then the last event by p in H is $Exit_p$.
— *Informally, if a process p participates in H ($H \mid p \neq \langle\rangle$), then at the end of H, one of the following holds:* **(i)** *p is in its entry section and has not yet executed its critical section ($p \in \mathrm{Act}(H) - RFS$);* **(ii)** *$p$ is in the process of "rolling forward" and is in its entry or exit section ($p \in \mathrm{Act}(H) \cap RFS$); or* **(iii)** *$p$ has already finished its execution and is in its noncritical section (i.e., $p \in \mathrm{Fin}(H)$).*

RF5: For any event e_p in H, if e_p is a critical write or a critical comparison in H, then e_p is also a critical write or a critical comparison in $H \mid (\{p\} \cup RFS)$.
— *Informally, if an event e_p in H is a critical write or a critical comparison, then it remains critical if we erase all processes not in RFS and different from p.* □

Condition RF5 is used to show that the property of being a critical write/comparison is conserved when considering certain related computations. Recall that, if e_p is not the first event by p to write to v, then for it to be critical, there must be a write to v by another process q in the subcomputation between p's most recent write (via a remote write or a successful comparison event) and event e_p. Similarly, if e_p is not the first unsuccessful comparison by p on v, then for it to be critical, there must be a write to v by another process q in the subcomputation between p's most recent unsuccessful comparison on v and event e_p. RF5 ensures that if q is not in RFS, then other process q' exists that *is* in RFS and that writes to v in the subcomputation in question.

Note that a valid RF-set can be "expanded": if RFS is a valid RF-set of computation H, then any set of processes that participate in H, provided that it is a superset of RFS, is also a valid RF-set of H.

The invisible and promoted processes (which partition the set of active processes) are defined as follows.

Definition: Let $\mathcal{S} = (C, P, V)$ be a mutual exclusion system, H be a computation in C, and RFS be a valid RF-set of H. We define $\mathrm{Inv}_{RFS}(H)$, the set of *invisible processes* in H, and $\mathrm{Pmt}_{RFS}(H)$, the set of *promoted processes* in H, as follows.

$$\begin{aligned} \mathrm{Inv}_{RFS}(H) &= \mathrm{Act}(H) - RFS \\ \mathrm{Pmt}_{RFS}(H) &= \mathrm{Act}(H) \cap RFS \end{aligned}$$ □

For brevity, we often omit the specific RF-set if it is obvious from the context, and simply use the notation $\mathrm{Inv}(H)$ and $\mathrm{Pmt}(H)$. Finally, the regularity condition can be defined as "all the processes we wish to roll forward have finished execution."

Definition: A computation H in C is *regular* if and only if $\mathrm{Fin}(H)$ is a valid RF-set of H. □

3.2 Detailed Proof Overview

The overall structure of our proof is depicted in Fig. 4. Initially, we start with H_1, in which $\mathrm{Act}(H_1) = P$, $\mathrm{Fin}(H_1) = \{\}$, and each process p has one critical event, $Enter_p$. We inductively construct longer and longer regular computations until our lower bound is met. At the j^{th} induction step, we consider a computation H_j such that $\mathrm{Act}(H_j)$ consists of n ($\leq N$) processes, each of which executes j critical events in H_j. We show that if $j > c \log n$, where c is a fixed constant, then the lower bound has already been attained. Thus, in the inductive step, we assume $j \leq c \log n$. Based on the existence of H_j, we construct a regular computation H_{j+1} such that $\mathrm{Act}(H_{j+1})$ consists of $\Omega(n/\log^2 n)$ processes,[12] each of which executes $j+1$ critical events in H_{j+1}. The construction method, formally described in Lemma 7 (in Sec. 4), is explained below.

[12] We use $\log^2 n$ to denote $(\log n)^2$.

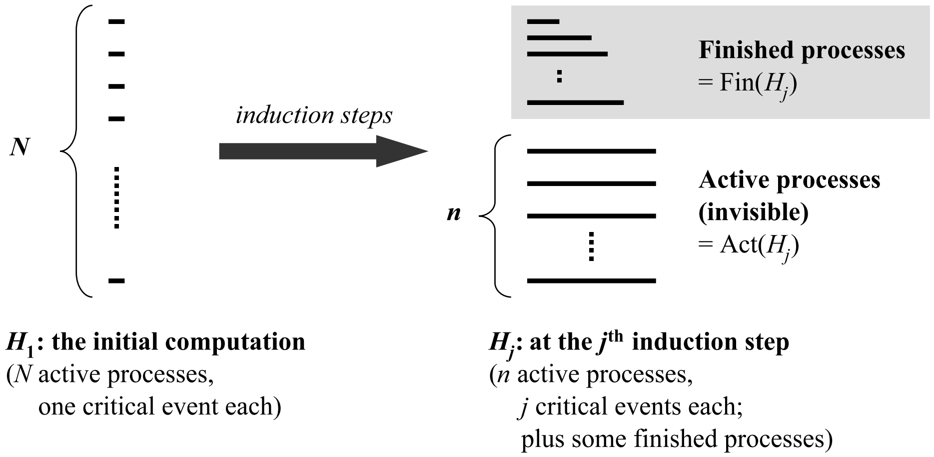

Figure 4: Induction steps. In this and subsequent figures, a computation is depicted as a collection of bold horizontal lines, with each line representing the events of a single process.

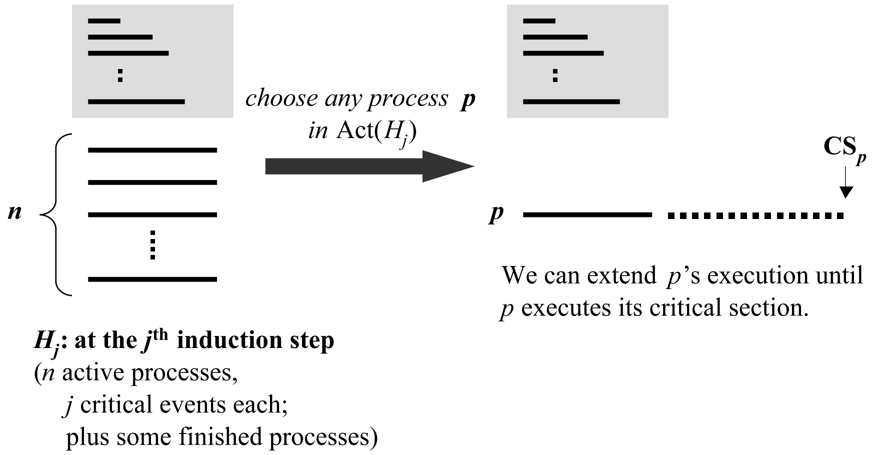

Figure 5: For each active (invisible) process p in a regular computation, if p runs in isolation (*i.e.*, with only finished processes), p eventually executes CS_p.

Since computation H_j is regular, no active process has knowledge of other active processes. Therefore, we can "erase" any active process and still get a valid computation (Lemma 1). Moreover, if we choose an active process p and erase all other active processes, then by the Progress property, p eventually executes CS_p, as shown in Fig. 5. We claim that every process p in $\mathrm{Act}(H_j)$, except at most one, executes at least one additional critical event before it executes CS_p. This claim is formally stated and proved in Lemma 5; here we give an informal explanation.

Assume, to the contrary, that we have two distinct processes, p and q, each of which may execute its CS event, if executed alone as shown in Fig. 5, by first executing only noncritical events. It can be shown that noncritical events of invisible processes cannot cause any information flow among these processes (Lemma 4). That is, an invisible process cannot become "visible" as a result of executing noncritical events. In particular, a local event of process p cannot cause information flow, because, by RF2, no other invisible process can access p's local variables. In addition, a remote read event of v by p is noncritical only if p has already read v. If another process q has written v after p's last read, then by RF3, the last process to write v is in RFS, *i.e.*, it is not invisible. Similar arguments apply to remote write or comparison events.

It follows that process p is unaware of q until it executes its critical section, and *vice versa*. Therefore, we can let both p and q execute *concurrently* after $H_j \mid (\{p, q\} \cup \mathrm{Fin}(H_j))$ — in particular, we can append $F_p \circ F_q$ (or $F_q \circ F_p$), constructing a computation that may be followed by *both* CS_p and CS_q, clearly violating the Exclusion property.[13]

Thus, among the n processes in $\mathrm{Act}(H_j)$, at least $n-1$ processes can execute an additional critical event before entering its critical section. We call these events "next" critical events, and denote the corresponding set of processes by Y.

The processes in Y collectively execute at most nj critical events in H_j. Let $\mathcal{E}$ be the set of all these events plus all the next critical events. Since we assumed $j \leq c \log n$, we have $|\mathcal{E}| \leq nj + n \leq (c \log n + 1)n$. Among the variables accessed by these events, we identify V_{HC}, the set of variables that experience "high contention," as those that are remotely accessed by at least $d \log^2 n$ critical events in $\mathcal{E}$, where d is another constant to be specified. Because of the Atomicity Property, each event can access at most one remote variable, so we have $|V_{\mathrm{HC}}| \leq (c \log n + 1)n/(d \log^2 n) \leq (c+1)n/(d \log n)$. Next, we partition the processes in Y depending on whether their next critical events access a variable in V_{HC}:

$$\begin{aligned} P_{\mathrm{HC}} &= \{y \in Y \colon y\text{'s next critical event accesses some variable in } V_{\mathrm{HC}}\}, \\ P_{\mathrm{LC}} &= Y - P_{\mathrm{HC}}. \end{aligned}$$

Because H_j is regular and $Y \subseteq \mathrm{Act}(H_j)$, we can erase any process in Y. Hence, we can erase the smaller of P_{HC} and P_{LC} and still have $\Omega(n)$ remaining active processes. We consider these cases separately.

Erasing strategy. Assume that we erased P_{HC} and saved P_{LC}. This situation is depicted in Fig. 6. In order to construct a regular computation, we want to eliminate any information flow. There are two cases to consider.

If p's next critical event reads a value that is written by some other active process q, then information flow clearly arises, and hence must be eliminated, by erasing either p or q. The other case is more subtle: if p's next critical event *overwrites* a value that is written by q, then although no information flow results, problems may arise later. In particular, assume that, in a later induction step, yet another process r reads the value written by p. In this case, simply erasing p does not eliminate all information flow, because then r would read a value written by q instead. In order to simplify bookkeeping, we simply assume that a conflict arises whenever any next critical event accesses any variable that is accessed by a critical event (past or next) of any other processes. That is, we consider all four possibilities, write followed by read, write followed by write, read followed by read, and read followed by write, to be conflicts.

Define V_{LC} as the set of variables remotely accessed by the next critical events of P_{LC}. Then clearly, every variable in V_{LC} is a "low contention" variable, and hence is accessed by at most $d \log^2 n$ different critical events (and, hence, different processes). Therefore, the next event by a process in P_{LC} can conflict with at most $d \log^2 n$ processes. By generating a "conflict graph" and applying Turán's theorem (Theorem 1), we can find a set of processes Z such that $|Z| = \Omega(n/\log^2 n)$, and among the processes in Z, there are no conflicts. By retaining Z and erasing all other active processes, we can eliminate all conflicts. Thus, we can construct H_{j+1}.

Roll-forward strategy. Assume that we erased P_{LC} and saved P_{HC}. This situation is depicted in Fig. 7. In this case, the erasing strategy does not work, because eliminating all conflicts will leave us with at most $|V_{\mathrm{HC}}|$ processes, which may be $o(n/\log^2 n)$.

Every next event by a process in P_{HC} accesses a variable in V_{HC}. For each variable $v \in V_{\mathrm{HC}}$, we arrange the next critical events that access v by placing write, comparison, and read events in that order. Then, all next write events of v, except for the last one, are overwritten by subsequent writes, and hence cannot create any information flow. (That is, even if some other process later reads v, it cannot gather any information of these "next" writers, except for the last one.) Furthermore, we can arrange comparison events such that at most one of them succeeds, as follows.

[13] It is crucial that both F_p and F_q are free of critical events. For example, if F_q contains a critical event, then it may read a variable that is written by p in F_p, or write a variable that is read by p in F_p. In the former case, $F_p \circ F_q$ causes information flow; in the latter, $F_q \circ F_p$ does.

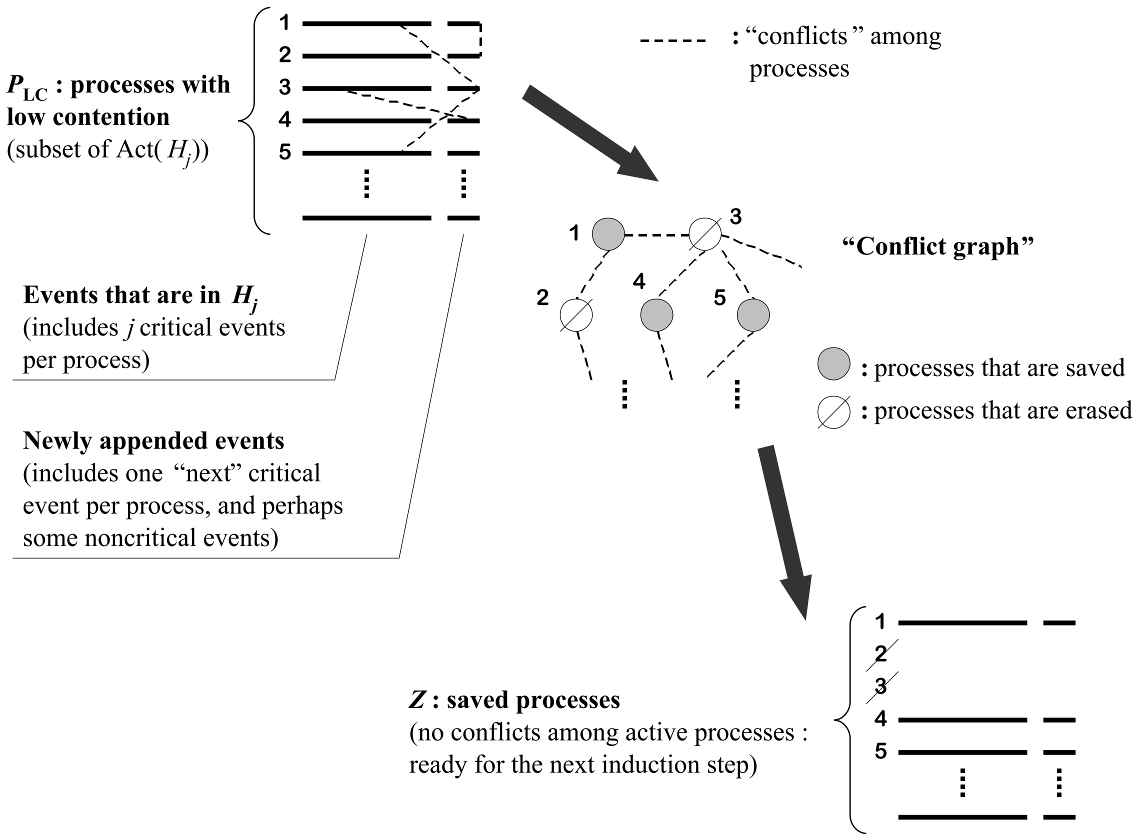

Figure 6: Erasing strategy. For simplicity, processes in $\mathrm{Fin}(H_j)$ are not shown in the remaining figures.

Assume that the value of v is α after all the next write events are executed. We first append all comparison events with an operation that can be written as $\mathsf{compare}(v, \beta)$ such that $\beta \neq \alpha$. These comparison events must fail. We then append all the remaining comparison events, namely, events with operation $\mathsf{compare}(v, \alpha)$. The first successful event among them (if any) changes the value of v. Thus, all subsequent comparison events must fail.

Thus, among the next events accessing some such $v \in V_{\mathrm{HC}}$, the only information flow that arises is from the "last writer" event $LW(v)$ and from the "successful comparison" event $SC(v)$ to all other next comparison and read events of v. We define $\overline{G}$ to be the resulting computation with the next critical events arranged as above, and define the RF-set RFS as $\{LW(v), SC(v)\colon v \in V_{\mathrm{HC}}\} \cup \mathrm{Fin}(H_j)$. Then, by definition, the set of promoted processes $\mathrm{Pmt}(\overline{G})$ consists of $LW(v)$ and $SC(v)$ for each $v \in V_{\mathrm{HC}}$. Because $|V_{\mathrm{HC}}| \leq (c+1)n/(d \log n)$, we have $|\mathrm{Pmt}(\overline{G})| \leq 2|V_{\mathrm{HC}}| \leq 2(c+1)n/(d \log n)$. We then roll the processes in $\mathrm{Pmt}(\overline{G})$ forward (*i.e.*, schedule only these processes, and temporarily pause all other processes, until every process in $\mathrm{Pmt}(\overline{G})$ reaches its noncritical section) by inductively constructing computations $G_0, G_1, \ldots, G_k$ (where $G_0 = \overline{G}$), such that each computation G_{j+1} contains one more critical event (of some process in $\mathrm{Pmt}(\overline{G})$) than G_j. (During the inductive construction, we may erase some active processes in order to eliminate conflicts generated by newly appended events, as explained below.) The computation $\overline{G}$ and the construction of $G_0, \ldots, G_k$ are depicted in Fig. 8.

If any process p in $\mathrm{Pmt}(\overline{G})$ executes at least $\log n$ critical events before returning to its noncritical section, then the lower bound easily follows. (It can be shown that $j + \log n = \Omega(\log N / \log \log N)$. The formal argument is presented in Theorem 2.) Therefore, we can assume that each process in $\mathrm{Pmt}(\overline{G})$ performs fewer than $\log n$ critical events while being rolled forward. Because $|\mathrm{Pmt}(\overline{G})| \leq 2(c+1)n/(d \log n)$, it follows that all the processes in $\mathrm{Pmt}(\overline{G})$ can be rolled forward with a total of $O(n)$ critical events.

Since each process in $\mathrm{Pmt}(\overline{G})$ is eventually rolled forward and reaches its noncritical section (see Fig. 8),

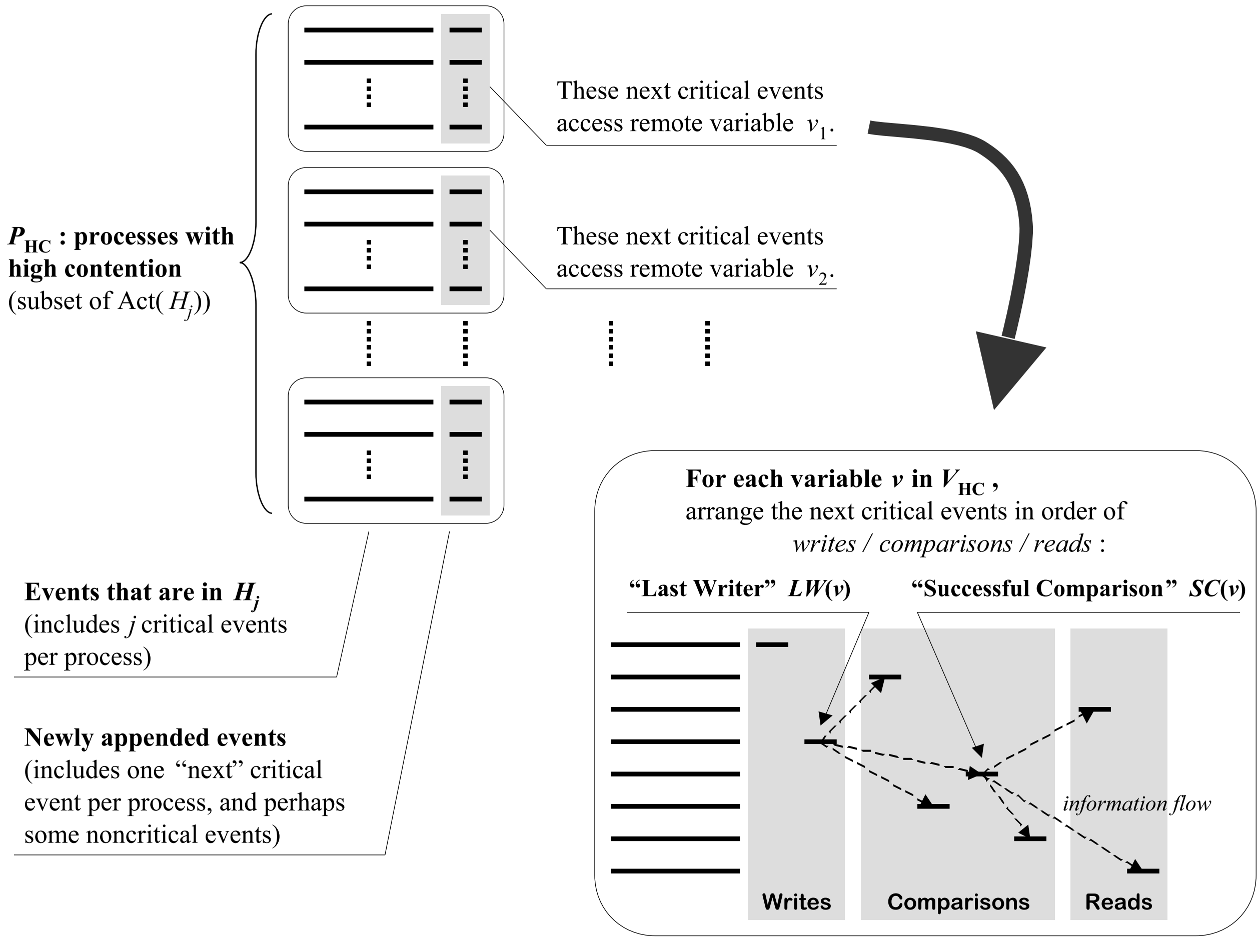

Figure 7: Roll-forward strategy. Part I.

we do not have to prevent information flow among these processes. (Once all processes in $\mathrm{Pmt}(\overline{G})$ are rolled forward, other processes may freely read variables written by them — knowledge of another process in its noncritical section cannot cause an active process to block.) As before, it can be shown that noncritical events do not generate any information flow that has to be prevented. (In particular, if a noncritical remote read by a process p is appended, then p must have previously read the same variable. By Condition RF3, if the last writer is another process, then that process is in the RF-set, and hence is allowed to be known to other processes.) Therefore, the only case to consider is when a critical event of $\mathrm{Pmt}(\overline{G})$ reads a variable v that is written by another active process $q \notin \mathrm{Pmt}(\overline{G})$, *i.e.*, $q \in \mathrm{Inv}(\overline{G})$.

If there are multiple processes in $\mathrm{Act}(\overline{G})$ that write to v in $\overline{G}$, then Condition RF3 guarantees that the last writer in $\overline{G}$ belongs to the RF-set, *i.e.*, we have $q \notin \mathrm{Inv}(\overline{G})$. On the other hand, if there is a single process in $\mathrm{Act}(\overline{G})$ that writes to v in $\overline{G}$, then that process must be q, and information flow can be prevented by erasing q. It follows that each critical event of $\mathrm{Pmt}(\overline{G})$ can conflict with, and thus erase, at most one process in $\mathrm{Inv}(\overline{G})$.

Therefore, the entire roll-forward procedure erases $O(n)$ processes from $\mathrm{Inv}(\overline{G})$. Because $|\,\mathrm{Inv}(\overline{G})| = \Theta(n) - |\,\mathrm{Pmt}(\overline{G})|$ and $|\,\mathrm{Pmt}(\overline{G})| = O(n)$, we can adjust constant coefficients so that $|\,\mathrm{Pmt}(\overline{G})|$ is at most n multiplied by a small constant (< 1), in which case $\Omega(n)$ processes (*i.e.*, processes in $\mathrm{Inv}(\overline{G})$) survive after the entire procedure. Thus, we can construct H_{j+1}.

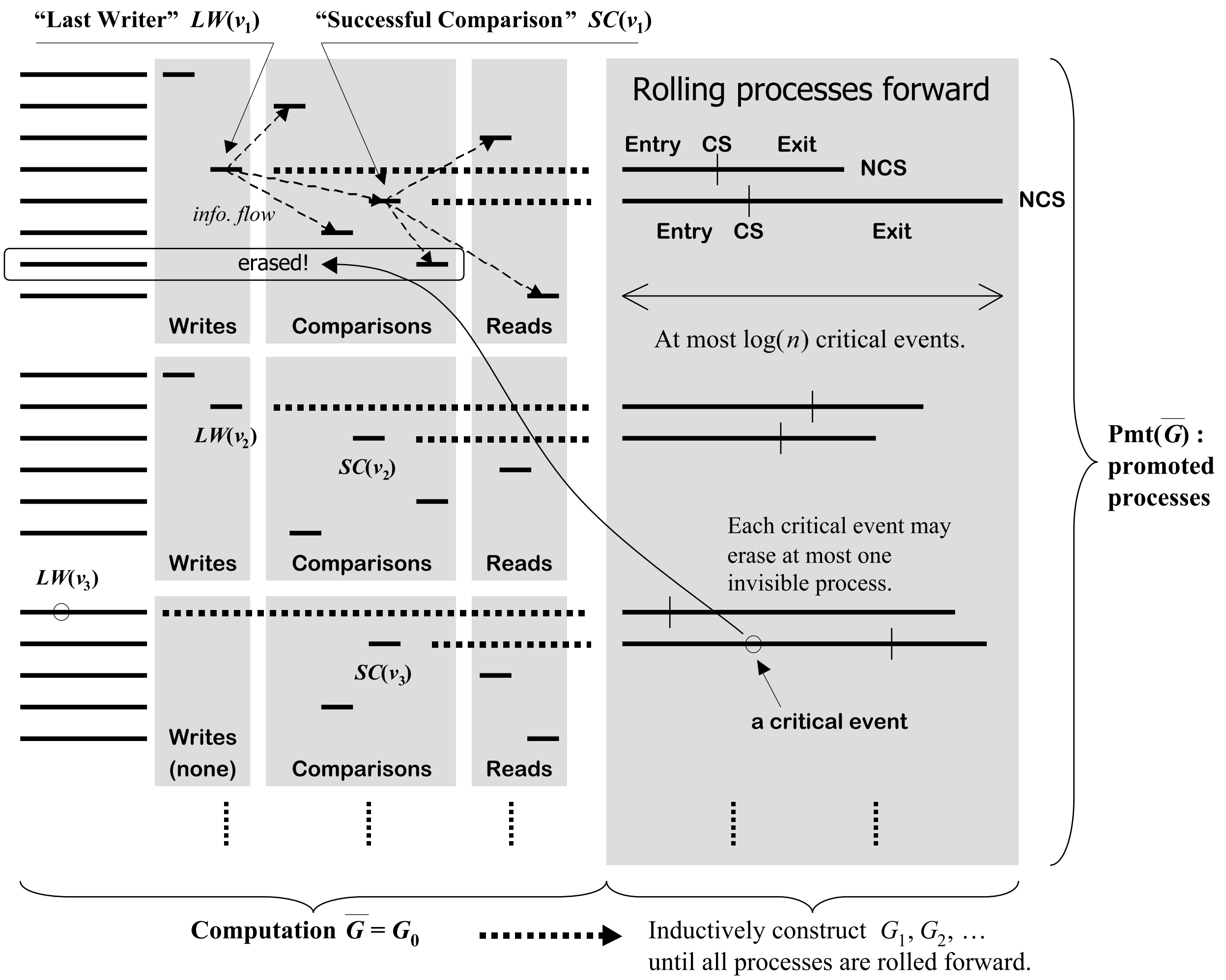

Figure 8: Roll-forward strategy. Part II.

4 Detailed Lower-bound Proof

In this section, we present our lower-bound theorem. We begin by stating several lemmas. Full proofs for Lemmas 1–6 can be found in an appendix. In many of these proofs, computations with a valid RF-set RFS are considered. When this is the case, we omit RFS when quoting properties RF1–RF5. For example, "H satisfies RF1" means that "H satisfies RF1 when the RF-set under consideration is RFS." Throughout this section, as well as in the appendix, we assume the existence of a fixed mutual exclusion system $\mathcal{S} = (C, P, V)$.

According to Lemma 1, stated next, any invisible process can be safely "erased."

Lemma 1 *Consider a computation H and two sets of processes RFS and Y. Assume the following:*

- $H \in C$; (1)
- *RFS is a valid RF-set of H;* (2)
- $RFS \subseteq Y$. (3)

Then, the following hold: $H \,|\, Y \in C$; RFS is a valid RF-set of $H \,|\, Y$; an event e in $H \,|\, Y$ is a critical event if and only if it is also a critical event in H.

Proof sketch: Because H satisfies RF1, if a process p is not in RFS, no process other than p reads a value written by p. Therefore, $H \mid Y \in C$. Conditions RF1–RF5 can be individually checked to hold in $H \mid Y$, which implies that RFS is a valid RF-set of $H \mid Y$.

To show that an event e_p in $H \mid Y$ is a critical event if and only if it is also a critical event in H, it is enough to consider critical writes and comparisons. (Transition events and critical reads are straightforward.) In this case, RF5 implies that e_p is critical in $H \mid Y$ if and only if it is also critical in $H \mid \{p\} \cup RFS$, which in turn holds if and only if it is also critical in H. □

The next lemma shows that the property of being a critical event is conserved across "similar" computations. Informally, if process p cannot distinguish two computations H and H', and if p may execute a critical event e_p after H, then it can also execute a critical event e'_p after $H' \circ G$, where G is a computation that does not contain any events by p. Moreover, if G satisfies certain conditions, then $H' \circ G \circ \langle e'_p \rangle$ satisfies RF5, preserving the "criticalness" of e'_p across related computations.

Lemma 2 *Consider three computations H, H', and G, a set of processes RFS, and two events e_p and e'_p of a process p. Assume the following:*

- $H \circ \langle e_p \rangle \in C$; (4)
- $H' \circ G \circ \langle e'_p \rangle \in C$; (5)
- *RFS is a valid RF-set of H;* (6)
- *RFS is a valid RF-set of H';* (7)
- $e_p \sim e'_p$; (8)
- $p \in \mathrm{Act}(H)$; (9)
- $H \mid (\{p\} \cup RFS) = H' \mid (\{p\} \cup RFS)$; (10)
- $G \mid p = \langle\rangle$; (11)
- *no events in G write any of p's local variables;* (12)
- *e_p is critical in $H \circ \langle e_p \rangle$.* (13)

Then, e'_p is critical in $H' \circ G \circ \langle e'_p \rangle$. Moreover, if the following conditions are true,

(A) *$H' \circ G$ satisfies RF5;*

(B) *if e_p is a comparison event on a variable v, and if G contains a write to v, then $G \mid RFS$ also contains a write to v.*

then $H' \circ G \circ \langle e'_p \rangle$ also satisfies RF5.

Proof sketch: It is enough to consider the following case: e_p is a critical write or a critical comparison on v such that $last\_writer(v, H) = q$ holds for some process q, where $q \neq \bot$ and $q \neq p$. (As before, if e_p is a transition event or critical read, then the reasoning is straightforward.) If e_p is a critical write, then by applying RF3 to H, we can show $q \in RFS$. Thus, by (10), q also writes to v in H' after p's last write to v. Hence, e'_p is critical in $H' \circ G \circ \langle e'_p \rangle$.

Assume that e_p is a critical comparison. If G contains a write to v, then by (11), e_p is the first comparison event on v by p after G, and hence is critical by definition. On the other hand, if G does not contain a write to v, then by (12), we can show that e_p and e'_p read the same value for each variable they read, and hence by P2 and P5, we have $e_p = e'_p$. Thus, e'_p is a successful (respectively, unsuccessful) comparison if and only if e_p is also a successful (respectively, unsuccessful) comparison. As in the case of a critical write, by applying RF3 to H, we can show $q \in RFS$. Using this fact, the conditions given in the definition of a critical successful/unsuccessful comparison can be individually checked to hold for e'_p.

If Condition (A) holds, then in order to show that $H' \circ G \circ \langle e'_p \rangle$ satisfies RF5, it suffices to consider e'_p. Condition (B) guarantees that if e'_p is critical because of a write to v in G, then e'_p is also critical in $(H' \circ G \circ \langle e'_p \rangle) \mid (\{p\} \cup RFS)$. □

The next lemma provides means of appending an event e_p of an active process, while maintaining RF1 and RF2. This lemma is used inductively in order to extend a computation with a valid RF-set. Specifically, (20) guarantees that RF2 is satisfied, and (21) forces any information flow to originate from a process in RFS, thus satisfying RF1. (Note that, if $q = \bot$, $q = p$, or $v_{\mathrm{rem}} \notin Rvar(e_p)$ holds, then no information flow occurs.) The proof of this lemma is mainly technical in nature and is omitted here.

Lemma 3 *Consider two computations H and G, a set of processes RFS, and an event e_p of a process p. Assume the following:*

- $H \circ G \circ \langle e_p \rangle \in C$; (14)
- *RFS is a valid RF-set of H;* (15)
- $p \in \mathrm{Act}(H)$; (16)
- *$H \circ G$ satisfies RF1 and RF2;* (17)
- *G is an $\mathrm{Act}(H)$-computation;* (18)
- $G \,|\, p = \langle\rangle$; (19)
- *if e_p remotely accesses a variable v_{rem}, then the following hold:*
  - *if v_{rem} is local to a process q, then either $q \notin \mathrm{Act}(H)$ or $\{p, q\} \subseteq RFS$, and* (20)
  - *if $q = last\_writer(v_{\mathrm{rem}}, H \circ G)$, then one of the following hold: $q = \bot$, $q = p$, $q \in RFS$, or $v_{\mathrm{rem}} \notin Rvar(e_p)$.* (21)

Then, $H \circ G \circ \langle e_p \rangle$ satisfies RF1 and RF2.

The next lemma gives us means for extending a computation by appending noncritical events.

Lemma 4 *Consider a computation H, a set of processes RFS, and another set of processes $Y = \{p_1, p_2, \ldots, p_m\}$. Assume the following:*

- $H \in C$; (22)
- *RFS is a valid RF-set of H;* (23)
- $Y \subseteq \mathrm{Inv}_{RFS}(H)$; (24)
- *for each p_j in Y, there exists a computation L_{p_j}, satisfying the following:*
  - *L_{p_j} is a p_j-computation;* (25)
  - $H \circ L_{p_j} \in C$; (26)
  - *L_{p_j} has no critical events in $H \circ L_{p_j}$, that is, no event in L_{p_j} is a critical event in $H \circ L_{p_j}$.* (27)

Define L to be $L_{p_1} \circ L_{p_2} \circ \cdots \circ L_{p_m}$. Then, the following hold: $H \circ L \in C$, RFS is a valid RF-set of $H \circ L$, and L contains no critical events in $H \circ L$.

Proof sketch: For each j, define L^j to be $L_{p_1} \circ L_{p_2} \circ \cdots \circ L_{p_j}$. The lemma can be proved by induction on j. At each induction step, it is assumed that $H \circ L^j \in C$, RFS is a valid RF-set of $H \circ L^j$, and L^j contains no critical events in $H \circ L^j$. Because $L_{p_{j+1}}$ contains no critical events in $H \circ L_{p_{j+1}}$, it can be appended to $H \circ L^j$ to get $H \circ L^{j+1}$ for the next induction step. (As mentioned at the end of Sec. 3, appending a noncritical event cannot cause any undesired information flow from invisible processes to processes in RFS.) $\square$

The next lemma states that if n active processes are competing for entry into their critical sections, then at least $n - 1$ of them execute at least one more critical event before entering their critical sections.

Lemma 5 *Let H be a computation. Assume the following:*

- *$H \in C$, and* (28)
- *H is regular (i.e., $\mathrm{Fin}(H)$ is a valid RF-set of H).* (29)

Define $n = |\operatorname{Act}(H)|$. Then, there exists a subset Y of $\operatorname{Act}(H)$, where $n-1 \le |Y| \le n$, satisfying the following: for each process p in Y, there exist a p-computation L_p and an event e_p by p such that

- $H \circ L_p \circ \langle e_p \rangle \in C$; (30)
- *L_p contains no critical events in $H \circ L_p$;* (31)
- $e_p \notin \{Enter_p, CS_p, Exit_p\}$; (32)
- $\operatorname{Fin}(H)$ *is a valid RF-set of $H \circ L_p$;* (33)
- *e_p is a critical event by p in $H \circ L_p \circ \langle e_p \rangle$.* (34)

Proof sketch: First, we construct, for each process p in $\operatorname{Act}(H)$, a computation L_p and an event e_p that satisfy (30) and (31). Then, we show that every event e_p thus constructed, except at most one, satisfies (32). The other conditions can be easily proved and will be omitted here.

Define $H_p = H \,|\, (\{p\} \cup \operatorname{Fin}(H))$. Because H is regular, $\operatorname{Fin}(H)$ is a valid RF-set of H. Hence, by Lemma 1, H_p is in C and $\operatorname{Fin}(H)$ is a valid RF-set of H_p. Since $p \in \operatorname{Act}(H)$, we have $\operatorname{Act}(H_p) = \{p\}$ and $\operatorname{Fin}(H_p) = \operatorname{Fin}(H)$. Therefore, by the Progress property, there exists a p-computation F_p such that $H_p \circ F_p \circ \langle CS_p \rangle \in C$. If F_p has a critical event, then let e'_p be the first critical event in F_p, and let L_p be the prefix of F_p that precedes e'_p (*i.e.*, $F_p = L_p \circ \langle e'_p \rangle \circ \cdots$). Otherwise, define L_p to be F_p and e'_p to be CS_p.

Now we have a p-computation L_p and an event e'_p by p, such that $H_p \circ L_p \circ \langle e'_p \rangle \in C$, in which L_p has no critical events and e'_p is a critical event. Because L_p has no critical events in $H_p \circ L_p$, it can be shown that $H \circ L_p \in C$ and that L_p has no critical events in $H \circ L_p$. Because $H \circ L_p$ and $H_p \circ L_p$ are equivalent with respect to p, by P3, there exists an event e_p by p such that $e_p \sim e'_p$ and $H \circ L_p \circ \langle e_p \rangle \in C$.

We now claim that at most one process in $\operatorname{Act}(H)$ fails to satisfy (32). Because $p \in \operatorname{Act}(H)$ and H is regular, e_p cannot be $Enter_p$ or $Exit_p$. By the Exclusion property, there can be at most one $p \in \operatorname{Act}(H)$ such that $e_p = CS_p$. □

The following lemma is used to roll processes forward. It states that as long as there exist promoted processes, we can extend the computation with one more critical event of some promoted process, and at most one invisible process must be erased due to the resulting information flow.

Lemma 6 *Consider a computation H and set of processes RFS. Assume the following:*

- $H \in C$; (35)
- *RFS is a valid RF-set of H;* (36)
- $\operatorname{Fin}(H) \subsetneq RFS$ (*i.e.*, $\operatorname{Fin}(H)$ *is a proper subset of* RFS). (37)

Then, there exists a computation G satisfying the following.

- $G \in C$; (38)
- *RFS is a valid RF-set of G;* (39)
- *G can be written as $H \,|\, (Y \cup RFS) \circ L \circ \langle e_p \rangle$, for some choice of Y, L, and e_p, satisfying the following:*
  - *Y is a subset of* $\operatorname{Inv}(H)$ *such that* $|\operatorname{Inv}(H)| - 1 \le |Y| \le |\operatorname{Inv}(H)|$, (40)
  - $\operatorname{Inv}(G) = Y$, (41)
  - *L is a* $\operatorname{Pmt}(H)$*-computation,* (42)
  - *L has no critical events in G,* (43)
  - $p \in \operatorname{Pmt}(H)$, *and* (44)
  - *e_p is critical in G;* (45)
- $\operatorname{Pmt}(G) \subseteq \operatorname{Pmt}(H)$; (46)
- *An event in $H \,|\, (Y \cup RFS)$ is critical if and only if it is also critical in H.* (47)

Proof sketch: Let $H' = H \mid RFS$ and $Z = \mathrm{Pmt}(H)$. Then, by the definition of an active process, $\mathrm{Act}(H') = (\mathrm{Act}(H) \cap RFS) = \mathrm{Pmt}(H) = Z$. By Lemma 1, $H' \in C$ and RFS is a valid RF-set of H'.

Therefore, by applying the Progress property, we can construct a Z-computation F such that $H' \circ F \circ \langle \bar{e}_r \rangle \in C$, where r is a process in Z and $\bar{e}_r$ is either CS_r or $Exit_r$. If F has a critical event, then let e'_p be the first critical event in F, and let L be the prefix of F that precedes e'_p (*i.e.*, $F = L \circ \langle e'_p \rangle \circ \cdots$). Otherwise, define L to be F and e'_p to be $\bar{e}_r$. Because F is a Z-computation, $p \in Z$.

Now we have a Z-computation L and an event e'_p by $p \in Z$, such that $H' \circ L \circ \langle e'_p \rangle \in C$, L has no critical events in $H' \circ L \circ \langle e'_p \rangle$, and e'_p is a critical event in $H' \circ L \circ \langle e'_p \rangle$. It can be shown that $H \circ L \in C$ and that L has no critical events in $H \circ L$. (This follows because H and H' are equivalent with respect to Z.) Because $H \circ L$ and $H' \circ L$ are equivalent with respect to p, by P3, there exists an event e''_p by p such that $e''_p \sim e'_p$ and $H \circ L \circ \langle e''_p \rangle \in C$.

Because e'_p is a critical event in $H' \circ L \circ \langle e'_p \rangle$ and e''_p accesses the same variables as e'_p, it can be shown that e''_p is a critical event in $H \circ L \circ \langle e''_p \rangle$. Let v be the remote variable accessed by e''_p. If v is local to a process q in $\mathrm{Inv}(H)$, or if $q = last\_writer(v, H \circ L)$ is in $\mathrm{Inv}(H)$, then we can "erase" process q and construct a computation G that satisfies the requirements stated in the lemma. (If both conditions hold simultaneously, then by RF2, q is identical in both cases.) The event e_p that is appended to obtain G is congruent to e''_p, *i.e.*, $e_p \sim e''_p$. $\square$

The following theorem is due to Turán [19].

Theorem 1 (Turán) *Let $\mathcal{G} = (V, E)$ be an undirected graph, with vertex set V and edge set E. If the average degree of $\mathcal{G}$ is d, then an independent set*[14] *exists with at least $\lceil |V|/(d+1) \rceil$ vertices.* $\square$

The following lemma provides the induction step that leads to the lower bound in Theorem 2.

Lemma 7 *Let H be a computation. Assume the following:*

- *$H \in C$, and* (48)
- *H is regular (i.e., $\mathrm{Fin}(H)$ is a valid RF-set of H).* (49)

Define $n = |\mathrm{Act}(H)|$. Also assume that

- *$n > 1$, and* (50)
- *each process in $\mathrm{Act}(H)$ executes exactly c critical events in H, where $c \leq \log n - 1$.* (51)

Then, one of the following propositions is true.

Pr1: *There exist a process p in $\mathrm{Act}(H)$ and a computation F in C such that*

- *$F \circ \langle Exit_p \rangle \in C$;*
- *F does not contain $\langle Exit_p \rangle$;*
- *p executes at least $(c + \log n)$ critical events in F.*

Pr2: *There exists a regular computation G in C such that*

- $\mathrm{Act}(G) \subseteq \mathrm{Act}(H)$; (52)
- $|\mathrm{Act}(G)| \geq \min\left(\dfrac{n}{6} - \dfrac{n}{2\log n} - \dfrac{1}{2},\ \dfrac{n-1}{2\cdot(12\log^2 n + 1)}\right)$; (53)
- *each process in $\mathrm{Act}(G)$ executes exactly $(c+1)$ critical events in G.* (54)

[14] An *independent set* of a graph $\mathcal{G} = (V, E)$ is a subset $V' \subseteq V$ such that no edge in E is incident to two vertices in V'.

Proof: We first apply Lemma 5. Assumptions (28) and (29) stated in Lemma 5 follow from (48) and (49), respectively. It follows that there exists a set of processes Y such that

- $Y \subseteq \mathrm{Act}(H)$, and (55)
- $n - 1 \leq |Y| \leq n$, (56)

and for each process $p \in Y$, there exist a computation L_p and an event e_p by p, such that

- $H \circ L_p \circ \langle e_p \rangle \in C$; (57)
- L_p is a p-computation; (58)
- L_p contains no critical events in $H \circ L_p$; (59)
- $e_p \notin \{Enter_p, CS_p, Exit_p\}$; (60)
- $\mathrm{Fin}(H)$ is a valid RF-set of $H \circ L_p$; (61)
- e_p is a critical event by p in $H \circ L_p \circ \langle e_p \rangle$. (62)

For each $p \in Y$, by (58), (59), and $p \in Y \subseteq \mathrm{Act}(H)$, we have

$$\mathrm{Act}(H \circ L_p) = \mathrm{Act}(H) \quad \wedge \quad \mathrm{Fin}(H \circ L_p) = \mathrm{Fin}(H). \tag{63}$$

By (50) and (56), Y is nonempty.

If Proposition Pr1 is satisfied by any process in Y, then the theorem is clearly true. Thus, we will assume, throughout the remainder of the proof, that there is no process in Y that satisfies Pr1. Define $\mathcal{E}_H$ as the set of critical events in H of processes in Y.

$$\mathcal{E}_H = \{f_q \text{ in } H\colon f_q \text{ is critical in } H \text{ and } q \in Y\} \tag{64}$$

Define $\mathcal{E} = \mathcal{E}_H \cup \{e_p\colon p \in Y\}$, *i.e.*, the set of all "past" and "next" critical events of processes in Y. From (51), (55), and (56), it follows that

$$|\mathcal{E}| = (c+1)|Y| \leq (c+1)n. \tag{65}$$

Now define V_{HC}, the set of variables that experience "high contention" (*i.e.*, those that are accessed by "sufficiently many" events in $\mathcal{E}$), as follows.

$$V_{\mathrm{HC}} = \{v \in V\colon \text{there are at least } 6\log^2 n \text{ events in } \mathcal{E} \text{ that remotely access } v\} \tag{66}$$

Since, by the Atomicity Property, each event in $\mathcal{E}$ can access at most one remote variable, from (51) and (65), we have

$$|V_{\mathrm{HC}}| \leq \frac{|\mathcal{E}|}{6\log^2 n} \leq \frac{(c+1)n}{6\log^2 n} \leq \frac{n}{6\log n}. \tag{67}$$

Define P_{HC}, the set of processes whose "next" event accesses a variable in V_{HC}, as follows.

$$P_{\mathrm{HC}} = \{p \in Y\colon e_p \text{ accesses a variable in } V_{\mathrm{HC}}\} \tag{68}$$

We now consider two cases, depending on $|P_{\mathrm{HC}}|$.

Case 1: $|P_{\mathrm{HC}}| < \frac{1}{2}|Y|$ (erasing strategy)

— *In this case, we start with $Y' = Y - P_{\mathrm{HC}}$, which consists of at least $(n-1)/2$ active processes. We construct a "conflict graph" $\mathcal{G}$, made of the processes in Y'. By applying Theorem 1, we can find a subset Z of Y' such that their critical events do not conflict with each other.*

Let $Y' = Y - P_{\mathrm{HC}}$. By (55), we have

$$Y' \subseteq \mathrm{Act}(H). \tag{69}$$

By (56) and Case 1, we also have

$$|Y'| \;=\; \big(|Y| - |P_{\text{HC}}|\big) \;>\; \left(|Y| - \frac{1}{2}Y\right) \;=\; \frac{1}{2}|Y| \;\geq\; \frac{n-1}{2}. \tag{70}$$

We now construct an undirected graph $\mathcal{G} = (Y', E_{\mathcal{G}})$, where each vertex is a process in Y'. To each process y in Y' and each variable $v \in var(e_y)$ that is remote to y, we apply the following rules.

- **R1:** If v is local to a process z in Y', then introduce edge $\{y, z\}$.
- **R2:** If there exists an event $f_p \in \mathcal{E}$ that remotely accesses v, and if $p \in Y'$, then introduce edge $\{y, p\}$.

Because each variable is local to at most one process, and since (by the Atomicity Property) an event can access at most one remote variable, Rule R1 can introduce at most one edge for each process in Y. Since $y \in Y'$, we have $y \notin P_{\text{HC}}$, which, by (68), implies $v \notin V_{\text{HC}}$. Hence, by (66), it follows that there are at most $6\log^2 n - 1$ events in $\mathcal{E}$ that remotely access v. Therefore, since an event can access at most one remote variable, Rule R2 can introduce at most $6\log^2 n - 1$ edges for each process in Y.

Combining Rules R1 and R2, at most $6\log^2 n$ edges are introduced for each process in Y. Since each edge is counted twice (for each of its endpoints), the average degree of $\mathcal{G}$ is at most $12\log^2 n$. Hence, by Theorem 1, there exists an independent set Z such that

$$Z \subseteq Y', \quad \text{and} \tag{71}$$

$$|Z| \geq \frac{|Y'|}{(12\log^2 n + 1)} \geq \frac{n-1}{2 \cdot (12\log^2 n + 1)}, \tag{72}$$

where the latter inequality follows from (70).

Next, we construct a computation G, satisfying Proposition Pr2, such that $\text{Act}(G) = |Z|$.

Define H' as

$$H' = H \,|\, (Z \cup \text{Fin}(H)). \tag{73}$$

By (69) and (71), we have

$$Z \subseteq Y' \subseteq Y \subseteq \text{Act}(H), \tag{74}$$

and hence,

$$\text{Act}(H') = Z \subseteq \text{Act}(H) \quad \wedge \quad \text{Fin}(H') = \text{Fin}(H). \tag{75}$$

We now apply Lemma 1, with 'RFS' $\leftarrow \text{Fin}(H)$ and 'Y' $\leftarrow Z \cup \text{Fin}(H)$. Among the assumptions stated in Lemma 1, (1) and (2) follow from (48) and (49), respectively; (3) is trivial. It follows that

- $H' \in C$, (76)
- $\text{Fin}(H)$ is a valid RF-set of H', and (77)
- an event in H' is critical if and only if it is also critical in H. (78)

Our goal now is to show that H' can be extended so that each process in Z has one more critical event. By (75), (77), and by the definition of a finished process,

$$\text{Inv}_{\text{Fin}(H)}(H') = \text{Act}(H') = Z. \tag{79}$$

For each $z \in Z$, define F_z as

$$F_z = (H \circ L_z) \,|\, (Z \cup \text{Fin}(H)). \tag{80}$$

By (74), we have $z \in Y$. Thus, applying (57), (58), (59), and (61) with 'p' $\leftarrow z$, it follows that

- $H \circ L_z \circ \langle e_z \rangle \in C$; (81)
- L_z is a z-computation; (82)

- L_z contains no critical events in $H \circ L_z$; (83)
- $\mathrm{Fin}(H)$ is a valid RF-set of $H \circ L_z$. (84)

By P1, (81) implies
$$H \circ L_z \in C. \tag{85}$$

We now apply Lemma 1, with 'H' $\leftarrow H \circ L_z$, 'RFS' $\leftarrow \mathrm{Fin}(H)$, and '$Y$' $\leftarrow Z \cup \mathrm{Fin}(H)$. Among the assumptions stated in Lemma 1, (1) and (2) follow from (85) and (84), respectively; (3) is trivial. It follows that

- $F_z \in C$, and (86)
- an event in F_z is critical if and only if it is also critical in $H \circ L_z$. (87)

Since $z \in Z$, by (73), (80), and (82), we have
$$F_z = H' \circ L_z.$$

Hence, by (83) and (87),

- L_z contains no critical events in $F_z = H' \circ L_z$. (88)

Let $m = |Z|$ and index the processes in Z as $Z = \{z_1,\ z_2,\ \ldots,\ z_m\}$. Define $L = L_{z_1} \circ L_{z_2} \circ \cdots \circ L_{z_m}$. We now use Lemma 4, with 'H' $\leftarrow H'$, 'RFS' $\leftarrow \mathrm{Fin}(H)$, '$Y$' $\leftarrow Z$, and 'p_j' $\leftarrow z_j$ for each $j = 1, \ldots, m$. Among the assumptions stated in Lemma 4, (22)–(24) follow from (76), (77), and (79), respectively; (25)–(27) follow from (82), (86), and (88), respectively, with 'z' $\leftarrow z_j$ for each $j = 1, \ldots, m$. This gives us the following.

- $H' \circ L \in C$; (89)
- $\mathrm{Fin}(H)$ is a valid RF-set of $H' \circ L$; (90)
- L contains no critical events in $H' \circ L$. (91)

To this point, we have successfully appended a (possibly empty) sequence of noncritical events for each process in Z. It remains to append a "next" critical event for each such process. Note that, by (82) and the definition of L,

- L is a Z-computation. (92)

Thus, by (75) and (91), we have
$$\mathrm{Act}(H' \circ L) = \mathrm{Act}(H') = Z \quad \wedge \quad \mathrm{Fin}(H' \circ L) = \mathrm{Fin}(H') = \mathrm{Fin}(H). \tag{93}$$

By (73) and the definition of L, it follows that

- for each $z \in Z$, $(H \circ L_z) \mid (\{z\} \cup \mathrm{Fin}(H)) = (H' \circ L) \mid (\{z\} \cup \mathrm{Fin}(H))$. (94)

In particular, $H \circ L_z$ and $H' \circ L$ are equivalent with respect to z. Therefore, by (57), (89), and repeatedly applying P3, it follows that, for each $z_j \in Z$, there exists an event e'_{z_j}, such that

- $G \in C$, where $G = H' \circ L \circ E$ and $E = \langle e'_{z_1},\ e'_{z_2},\ \ldots,\ e'_{z_m} \rangle$; (95)
- $e'_{z_j} \sim e_{z_j}$. (96)

By the definition of E,

- E is a Z-computation. (97)

By (60), (93), and (96), we have

$$\mathrm{Act}(G) = \mathrm{Act}(H' \circ L) = Z \quad \wedge \quad \mathrm{Fin}(G) = \mathrm{Fin}(H' \circ L) = \mathrm{Fin}(H). \tag{98}$$

By (60), (62), and (96), it follows that for each $z_j \in Z$, both e_{z_j} and e'_{z_j} access a common remote variable, say, v_j. Since Z is an independent set of $\mathcal{G}$, by Rules R1 and R2, we have the following:

- for each $z_j \in Z$, v_j is not local to any process in Z; (99)
- $v_j \neq v_k$, if $j \neq k$.

Combining these two, we also have:

- for each $z_j \in Z$, no event in E other than e'_{z_j} accesses v_j (either locally or remotely). (100)

We now establish two claims.

Claim 1: For each $z_j \in Z$, if we let $q = \mathit{last\_writer}(v_j, H' \circ L)$, then one of the following holds: $q = \perp$, $q = z_j$, or $q \in \mathrm{Fin}(H)$.

Proof of Claim: It suffices to consider the case when $q \neq \perp$ and $q \neq z_j$ hold, in which case there exists an event f_q by q in $H' \circ L$ that writes to v_j. By (73) and (92), we have $q \in Z \cup \mathrm{Fin}(H)$. We claim that $q \in \mathrm{Fin}(H)$ holds in this case. Assume, to the contrary,

$$q \in Z. \tag{101}$$

We consider two cases. First, if f_q is a critical event in $H' \circ L$, then by (91), f_q is an event of H', and hence, by (78), f_q is also a critical event in H. By (74) and (101), we have $q \in Y$. Thus, by (64), we have $f_q \in \mathcal{E}_H$, and hence $f_q \in \mathcal{E}$ holds by definition. By (99) and (101), v_j is remote to q. Thus, f_q *remotely* writes v_j. By (101) and $z_j \in Z$, we have

$$\{q, z_j\} \subseteq Z, \tag{102}$$

which implies $\{q, z_j\} \subseteq Y'$ by (71). From this, our assumption of $q \neq z_j$, and by applying Rule R2 with 'y' $\leftarrow z_j$ and 'f_p' $\leftarrow f_q$, it follows that edge $\{q, z_j\}$ exists in $\mathcal{G}$. However, (102) then implies that Z is not an independent set of $\mathcal{G}$, a contradiction.

Second, assume that f_q is a noncritical event in $H' \circ L$. Note that, by (99) and (101), v_j is remote to q. Hence, by the definition of a critical event, there exists a critical event $\bar{f}_q$ by q in $H' \circ L$ that remotely writes to v_j. However, this leads to contradiction as shown above. $\square$

Claim 2: Every event in E is critical in G. Also, G satisfies RF5 with 'RFS' $\leftarrow \mathrm{Fin}(H)$.

Proof of Claim: Define $E_0 = \langle\rangle$; for each positive j, define E_j to be $\langle e'_{z_1}, e'_{z_2}, \ldots, e'_{z_j}\rangle$, a prefix of E. We prove the claim by induction on j, applying Lemma 2 at each step. Note that, by (95) and P1, we have the following:

$$H' \circ L \circ E_j \circ \langle e'_{z_{j+1}}\rangle = H' \circ L \circ E_{j+1} \in C, \quad \text{for each } j. \tag{103}$$

Also, by the definition of E_j, we have

$$E_j \mid z_{j+1} = \langle\rangle, \quad \text{for each } j. \tag{104}$$

At each step, we assume

- $H' \circ L \circ E_j$ satisfies RF5 with 'RFS' $\leftarrow \mathrm{Fin}(H)$. (105)

The induction base ($j = 0$) follows easily from (90), since $E_0 = \langle\rangle$.

Assume that (105) holds for a particular value of j. Since $z_{j+1} \in Z$, by (74), we have

$$z_{j+1} \in Y, \tag{106}$$

and $z_{j+1} \in \mathrm{Act}(H)$. By applying (63) with 'p' $\leftarrow z_{j+1}$, and using (106), we also have $\mathrm{Act}(H \circ L_{z_{j+1}}) = \mathrm{Act}(H)$, and hence

$$z_{j+1} \in \mathrm{Act}(H \circ L_{z_{j+1}}). \tag{107}$$

By (104), if any event e'_{z_k} in E_j accesses a local variable v of z_{j+1}, then e'_{z_k} accesses v *remotely*, and hence $v = v_k$ by definition. However, by (99), v_k cannot be local to z_{j+1}. It follows that

- no events in E_j access any of z_{j+1}'s local variables. (108)

We now apply Lemma 2, with 'H' $\leftarrow H \circ L_{z_{j+1}}$, '$H'$' $\leftarrow H' \circ L$, 'G' $\leftarrow E_j$, 'RFS' $\leftarrow \mathrm{Fin}(H)$, '$e_p$' $\leftarrow e_{z_{j+1}}$, and 'e'_p' $\leftarrow e'_{z_{j+1}}$. Among the assumptions stated in Lemma 2, (5), (7), (9), (11), and (12) follow from (103), (90), (107), (104), and (108), respectively; (8) follows by applying (96) with 'z_j' $\leftarrow z_{j+1}$; (6) and (10) follow by applying (84) and (94), respectively, with 'z' $\leftarrow z_{j+1}$; and (4) and (13) follow by applying (57) and (62), respectively, with 'p' $\leftarrow z_{j+1}$, and using (106). Moreover, Assumption (A) follows from (105), and Assumption (B) is satisfied vacuously (with 'v' $\leftarrow v_{j+1}$) by (100).

It follows that $e'_{z_{j+1}}$ is critical in $H' \circ L \circ E_j \circ \langle e'_{z_{j+1}} \rangle = H' \circ L \circ E_{j+1}$, and that $H' \circ L \circ E_{j+1}$ satisfies RF5 with 'RFS' $\leftarrow \mathrm{Fin}(H)$. $\square$

We now claim that $\mathrm{Fin}(H)$ is a valid RF-set of G. Condition RF5 was already proved in Claim 2.

- **RF1 and RF2:** Define E_j as in Claim 2. We establish RF1 and RF2 by induction on j, applying Lemma 3 at each step. At each step, we assume

  - $H' \circ L \circ E_j$ satisfies RF1 and RF2 with 'RFS' $\leftarrow \mathrm{Fin}(H)$. (109)

  The induction base ($j = 0$) follows easily from (90), since $E_0 = \langle\rangle$.

  Assume that (109) holds for a particular value of j. Note that, by (100), we have $\mathit{last\_writer}(v_{j+1}, H' \circ L \circ E_j) = \mathit{last\_writer}(v_{j+1}, H' \circ L)$. Thus, by (93) and Claim 1,

  - if we let $q = \mathit{last\_writer}(v_{j+1}, H' \circ L \circ E_j)$, then one of the following holds: $q = \perp$, $q = z_{j+1}$, or $q \in \mathrm{Fin}(H) = \mathrm{Fin}(H' \circ L)$. (110)

  We now apply Lemma 3, with 'H' $\leftarrow H' \circ L$, 'G' $\leftarrow E_j$, 'RFS' $\leftarrow \mathrm{Fin}(H)$, '$e_p$' $\leftarrow e'_{z_{j+1}}$, and 'v_{rem}' $\leftarrow v_{j+1}$. Among the assumptions stated in Lemma 3, (14), (15), (17), (19), and (21) follow from (103), (90), (109), (104), and (110), respectively; (16) follows from (93) and $z_{j+1} \in Z$; (18) follows from (93) and (97); (20) follows from (99) and (93). It follows that $H' \circ L \circ E_{j+1}$ satisfies RF1 and RF2 with 'RFS' $\leftarrow \mathrm{Fin}(H)$.

- **RF3:** Consider a variable $v \in V$ and two different events f_q and g_r in G. Assume that both q and r are in $\mathrm{Act}(G)$, $q \neq r$, and that there exists a variable v such that $v \in \mathit{var}(f_q) \cap \mathit{var}(g_r)$. (Note that, by (98), $\{q, r\} \subseteq Z$.) We claim that these conditions can actually never arise simultaneously, which implies that G vacuously satisfies RF3.

  Since v is remote to at least one of q or r, without loss of generality, assume that v is remote to q. We claim that there exists an event $\bar{f}_q$ in $\mathcal{E}$ that accesses the same variable v. If f_q is an event of E, we have $f_q = e'_{z_j}$ for some $z_j \in Z$, and $e_{z_j} \in \mathcal{E}$ holds by definition; define $\bar{f}_q = e_{z_j}$ in this case. If f_q is a noncritical event in $H' \circ L$, then by definition of a critical event, there exists a critical event $\bar{f}_q$ in $H' \circ L$ that remotely accesses v. If f_q is a critical event in $H' \circ L$, then define $\bar{f}_q = f_q$. (Note that, if $\bar{f}_q$ is a critical event in $H' \circ L$, then by (78) and (91), $\bar{f}_q$ is also a critical event in H, and hence, by $q \in Z$, (74), and the definition of $\mathcal{E}$, we have $\bar{f}_q \in \mathcal{E}$.)

It follows that, in each case, there exists an event $\bar{f}_q \in \mathcal{E}$ that remotely accesses v. If v is local to r, then by Rule R1, $\mathcal{G}$ contains the edge $\{q, r\}$. On the other hand, if v is remote to r, then we can choose an event $\bar{g}_r \in \mathcal{E}$ that remotely accesses v, in the same way as shown above. Hence, by Rule R2, $\mathcal{G}$ contains the edge $\{q, r\}$. Thus, in either case, p and q cannot simultaneously belong to Z, a contradiction.

- **RF4:** By (90) and (98), it easily follows that G satisfies RF4 with respect to $\mathrm{Fin}(H)$.

Finally, we claim that G satisfies Proposition Pr2. By (98), we have $\mathrm{Act}(G) = Z \subseteq \mathrm{Act}(H)$, so G satisfies (52). By (72), we have (53). By (51), (78), and (91), each process in Z executes exactly c critical events in $H' \circ L$. Thus, by Claim 2, G satisfies (54).

Case 2: $|P_{\mathrm{HC}}| \geq \frac{1}{2}|Y|$ (roll-forward strategy)

— *In this case, we start with P_{HC}, which, by (56), consists of at least $(n-1)/2$ active processes. We first erase the processes in K, defined below, to satisfy RF2. Appending the critical events e_p for each p in $S = P_{\mathrm{HC}} - K$ gives us a non-regular computation $\overline{G}$. We then select a subset of P_{HC}, which consists of $LW(v)$ and $SC(v)$ for each $v \in V_{\mathrm{HC}}$, as the set of promoted processes. We roll these processes forward, inductively generating a sequence of computations $G_0, G_1, \ldots, G_k$ (where $G_0 = \overline{G}$), where the last computation G_k is regular. We erase at most $n/3$ processes during the procedure, which leaves $\Theta(n)$ active processes in G_k.*

Define K, the erased (or "killed") processes, S, the "survivors," and H', the resulting computation, as follows.

$$\begin{aligned} K &= \{p \in P_{\mathrm{HC}}\colon \text{there exists a variable } v \in V_{\mathrm{HC}} \text{ such that } v \text{ is local to } p\} && (111)\\ S &= P_{\mathrm{HC}} - K && (112)\\ H' &= H \,|\, (S \cup \mathrm{Fin}(H)) && (113) \end{aligned}$$

Because each variable is local to at most one process, from (67) and (111), we have

$$|K| \leq \frac{n}{6 \log n}. \qquad (114)$$

By (55), (68) and (112), we have

$$S \subseteq P_{\mathrm{HC}} \subseteq Y \subseteq \mathrm{Act}(H), \qquad (115)$$

and hence,

$$\mathrm{Act}(H') = S \subseteq \mathrm{Act}(H) \quad \wedge \quad \mathrm{Fin}(H') = \mathrm{Fin}(H). \qquad (116)$$

We now apply Lemma 1, with 'RFS' $\leftarrow \mathrm{Fin}(H)$ and 'Y' $\leftarrow S \cup \mathrm{Fin}(H)$. Among the assumptions stated in Lemma 1, (1) and (2) follow from (48) and (49), respectively; (3) is trivial. It follows that

- $H' \in C$, (117)
- $\mathrm{Fin}(H)$ is a valid RF-set of H', and (118)
- an event in H' is critical if and only if it is also critical in H. (119)

Our goal now is to show that H' can be extended to a computation $\overline{G}$ (defined later in Fig. 9), so that each process in S has one more critical event. By (116), (118), and by the definition of a finished process,

$$\mathrm{Inv}_{\mathrm{Fin}(H)}(H') = \mathrm{Act}(H') = S. \qquad (120)$$

For each $s \in S$, define F_s as

$$F_s = (H \circ L_s) \,|\, (S \cup \mathrm{Fin}(H)). \qquad (121)$$

By (115), we have $s \in Y$. Thus, applying (57), (58), (59), and (61) with 'p' $\leftarrow s$, it follows that

- $H \circ L_s \circ \langle e_s \rangle \in C$; (122)

- L_s is an s-computation; (123)
- L_s contains no critical events in $H \circ L_s$; (124)
- $\mathrm{Fin}(H)$ is a valid RF-set of $H \circ L_s$. (125)

By P1, (122) implies

$$H \circ L_s \in C. \tag{126}$$

We now apply Lemma 1, with 'H' $\leftarrow H \circ L_s$, 'RFS' $\leftarrow \mathrm{Fin}(H)$, and '$Y$' $\leftarrow S \cup \mathrm{Fin}(H)$. Among the assumptions stated in Lemma 1, (1) and (2) follow from (126) and (125), respectively; (3) is trivial. It follows that

- $F_s \in C$, and (127)
- an event in F_s is critical if and only if it is also critical in $H \circ L_s$. (128)

Since $s \in S$, by (113), (121), (123), and (127), we have

- $F_s = H' \circ L_s \in C$. (129)

Hence, by (124) and (128),

- L_s contains no critical events in $F_s = H' \circ L_s$. (130)

By (111) and (112), no variable in V_{HC} is local to any process in S. Therefore, by (68) and (112),

- for each process s in S, e_s *remotely* accesses a variable in V_{HC}. (131)

We now show that the events in $\{L_s : s \in S\}$ can be "merged" by applying Lemma 4. We arbitrarily index S as $\{s_1,\ s_2,\ \ldots,\ s_m\}$, where $m = |S|$. (Later, we construct a specific indexing of S to reduce information flow.) Let $L = L_{s_1} \circ L_{s_2} \circ \cdots \circ L_{s_m}$. Apply Lemma 4, with '$H$' $\leftarrow H'$, 'RFS' $\leftarrow \mathrm{Fin}(H)$, '$Y$' $\leftarrow S$, and 'p_j' $\leftarrow s_j$ for each $j = 1, \ldots, m$. Among the assumptions stated in Lemma 4, (22)–(24) follow from (117), (118), and (120), respectively; (25)–(27) follow from (123), (129), and (130), respectively, with 's' $\leftarrow s_j$ for each $j = 1, \ldots, m$. This gives us the following.

- $H' \circ L \in C$; (132)
- $\mathrm{Fin}(H)$ is a valid RF-set of $H' \circ L$; (133)
- L contains no critical events in $H' \circ L$. (134)

By (113) and the definition of L, we also have,

- for each $s \in S$, $(H \circ L_s) \,|\, (\{s\} \cup \mathrm{Fin}(H)) = (H' \circ L) \,|\, (\{s\} \cup \mathrm{Fin}(H))$; (135)
- for each $s \in S$, $(H' \circ L) \,|\, s = (H \circ L_s) \,|\, s$. (136)

We now re-index the processes in S so that information flow among them is minimized. The re-indexing method is expressed as an algorithm in Fig. 9. This algorithm is described in Claim 3 below. (The event ordering produced by the algorithm was illustrated earlier in Fig. 7.) The algorithm constructs an indexing $(s^1,\ s^2,\ \ldots,\ s^m)$ of S and two computations $\overline{G}$ and E, such that

- $\overline{G} \in C$, where $\overline{G} = H' \circ L \circ E$ and $E = \langle e'_{s^1},\ e'_{s^2},\ \ldots,\ e'_{s^m} \rangle$; (137)
- $e'_{s^j} \sim e_{s^j}$. (138)

By the definition of E,

- E is an S-computation. (139)

```
begin
    j := 0;   E_0 := ⟨⟩;
    for each v ∈ V_HC do
        W(v) := {};   C_1(v) := {};   C_2(v) := {};   R(v) := {}
    od;
    for each v ∈ V_HC do
        for each s ∈ S such that op(e_s) = write(v) do
            append(s);   add e'_s to W(v)
        od;
        α(v) := value(v, H' ∘ L ∘ E_j);
        for each s ∈ S such that op(e_s) = compare(v, β) for any β ≠ α(v) do
            append(s);   add e'_s to C_1(v)
        od;
        for each s ∈ S such that op(e_s) = compare(v, α(v)) do
            append(s);   add e'_s to C_2(v)
        od;
        for each s ∈ S such that op(e_s) = read(v) do
            append(s);   add e'_s to R(v)
        od
    od;
    E := E_m;
    G‾ = H' ∘ L ∘ E
end

procedure append(s : a process)
    INVARIANT: 0 ≤ j < m = |S|;
               E_j = ⟨e'_{s^1}, e'_{s^2}, ..., e'_{s^j}⟩;
               (s^1, s^2, ..., s^j) is a sequence of distinct processes in S;
               H' ∘ L ∘ E_j ∈ C;
               e'_{s^k} ~ e_{s^k} and s ≠ s^k for each k = 1, 2, ..., j.
    s^{j+1} := s;
        — By (122), (132), (136), and Property P3, there exists an event e'_{s^{j+1}}
          such that H' ∘ L ∘ E_j ∘ ⟨e'_{s^{j+1}}⟩ ∈ C and e'_{s^{j+1}} ~ e_{s^{j+1}} hold.
    E_{j+1} := E_j ∘ ⟨e'_{s^{j+1}}⟩;
    j := j + 1
end
```

Figure 9: Algorithm for arranging events of S so that information flow is sufficiently low.

By (60), (134), and (138), $L \circ E$ does not contain any transition events. Moreover, by the definition of L and E, $(L \circ E) \,|\, p \neq \langle\rangle$ implies $p \in S$, for each process p. Combining these assertions with (116), we have

$$\mathrm{Act}(\overline{G}) = \mathrm{Act}(H' \circ L) = \mathrm{Act}(H') = S \quad \wedge \quad \mathrm{Fin}(\overline{G}) = \mathrm{Fin}(H' \circ L) = \mathrm{Fin}(H') = \mathrm{Fin}(H). \tag{140}$$

We now state and prove two claims regarding $\overline{G}$. Claim 3 follows easily by examining the algorithm.

Claim 3: For each $v \in V_{\mathrm{HC}}$, the algorithm in Fig. 9 constructs four (possibly empty) sets of events, $W(v)$, $C_1(v)$, $C_2(v)$, and $R(v)$, and a value, $\alpha(v)$. All events in E that access v appear contiguously, in the following order.

- events in $W(v)$: each event e'_s in $W(v)$ satisfies $op(e'_s) = \mathsf{write}(v)$;
- events in $C_1(v)$: each event e'_s in $C_1(v)$ satisfies $op(e'_s) = \mathsf{compare}(v, \beta_s)$ for some $\beta_s \neq \alpha(v)$;
- events in $C_2(v)$: each event e'_s in $C_2(v)$ satisfies $op(e'_s) = \mathsf{compare}(v, \alpha(v))$;
- events in $R(v)$: each event e'_s in $R(v)$ satisfies $op(e'_s) = \mathsf{read}(v)$.

Moreover, in the computation $\overline{G}$, after all events in $W(v)$ are executed, and before any event in $C_2(v)$ is executed, v has the value $\alpha(v)$. All events in $C_1(v)$ (if any) are unsuccessful comparisons. At most one event in $C_2(v)$ is a successful comparison. (Note that a successful comparison event writes a value other than $\alpha(v)$, by definition. Thus, if there is a successful comparison, then all subsequent comparison events must fail.) For each $v \in V_{\mathrm{HC}}$, define $LW(v)$, the "last write," and $SC(v)$, the "successful comparison," as follows:

$$LW(v) = \begin{cases} \text{the last event in } W(v), & \text{if } W(v) \neq \{\}, \\ \mathit{last\_writer\_event}(v, H' \circ L), & \text{if } W(v) = \{\}; \end{cases}$$

$$SC(v) = \begin{cases} \text{the successful comparison in } C_2(v), & \text{if } C_2(v) \text{ contains a successful comparison,} \\ \perp, & \text{otherwise.} \end{cases}$$

Then, the last process to write to v (if any) is either $SC(v)$ (if $SC(v)$ is defined) or $LW(v)$ (otherwise). □

Before establishing our next claim, Claim 4, we define RFS as

$$\begin{aligned} RFS \;=\; & \mathrm{Fin}(H) \\ & \cup \{\mathit{owner}(LW(v))\colon v \in V_{\mathrm{HC}} \text{ and } LW(v) \neq \perp\} \\ & \cup \{\mathit{owner}(SC(v))\colon v \in V_{\mathrm{HC}} \text{ and } SC(v) \neq \perp\}. \end{aligned} \tag{141}$$

By (68), (111), (112), and (138), we have the following:

- for each $s \in S$, if e'_s remotely accesses v, and if v is local to a process q, then $q \notin S$. (142)

Note that "expanding" a valid RF-set does not falsify any of RF1–RF5. Therefore, using (133), (140), and $\mathrm{Fin}(H) \subseteq RFS \subseteq \mathrm{Fin}(H) \cup S$, it follows that

- RFS is a valid RF-set of $H' \circ L$. (143)

We now establish Claim 4, stated below.

Claim 4: Every event in E is critical in $\overline{G}$. Also, $\overline{G}$ satisfies RF5.

Proof of Claim: Define $E_0 = \langle\rangle$; for each positive j, define E_j to be $\langle e'_{s^1},\ e'_{s^2}, \ldots,\ e'_{s^j}\rangle$, a prefix of E. We prove the claim by induction on j, applying Lemma 2 at each step. Note that, by (137) and P1, we have the following:

$$H' \circ L \circ E_j \circ \langle e'_{s^{j+1}}\rangle = H' \circ L \circ E_{j+1} \in C, \quad \text{for each } j. \tag{144}$$

Also, by the definition of E_j, we have

$$E_j \mid s^{j+1} = \langle\rangle, \quad \text{for each } j. \tag{145}$$

At each step, we assume

- $H' \circ L \circ E_j$ satisfies RF5. (146)

The induction base ($j = 0$) follows easily from (143), since $E_0 = \langle\rangle$.
Assume that (146) holds for a particular value of j. Since $s^{j+1} \in S$, by (115), we have

$$s^{j+1} \in Y, \tag{147}$$

and $s^{j+1} \in \mathrm{Act}(H)$. By applying (63) with 'p' $\leftarrow s^{j+1}$, and using (147), we also have $\mathrm{Act}(H \circ L_{s^{j+1}}) = \mathrm{Act}(H)$, and hence

$$s^{j+1} \in \mathrm{Act}(H \circ L_{s^{j+1}}). \tag{148}$$

Also, by (142),

- no events in E_j access any of s^{j+1}'s local variables. (149)

We use Lemma 2 twice in sequence in order to prove Claim 4. First, by P3, and applying (122), (132), and (136) with 's' $\leftarrow s^{j+1}$, it follows that there exists an event $e''_{s^{j+1}}$, such that

- $H' \circ L \circ \langle e''_{s^{j+1}} \rangle \in C$, and (150)
- $e''_{s^{j+1}} \sim e_{s^{j+1}}$. (151)

We now apply Lemma 2, with 'H' $\leftarrow H \circ L_{s^{j+1}}$, '$H'$' $\leftarrow H' \circ L$, 'G' $\leftarrow \langle\rangle$, '$RFS$' $\leftarrow \mathrm{Fin}(H)$, '$e_p$' $\leftarrow e_{s^{j+1}}$, and 'e'_p' $\leftarrow e''_{s^{j+1}}$. Among the assumptions stated in Lemma 2, (5) and (7)–(9) follow from (150), (133), (151), and (148), respectively; (11) and (12) hold vacuously by 'G' $\leftarrow \langle\rangle$; (4), (6), and (10) follow by applying (122), (125), and (135), respectively, with 's' $\leftarrow s^{j+1}$; (13) follows by applying (62) with 'p' $\leftarrow s^{j+1}$, and using (147). It follows that

- $e''_{s^{j+1}}$ is critical in $H' \circ L \circ \langle e''_{s^{j+1}} \rangle$. (152)

Before applying Lemma 2 again, we establish the following preliminary assertions. Since $\mathrm{Fin}(H) \subseteq RFS$, by applying (125) with 's' $\leftarrow s^{j+1}$, it follows that

- RFS is a valid RF-set of $H \circ L_{s^{j+1}}$. (153)

We now establish a simple claim.

> **Claim 4-1:** If $e_{s^{j+1}}$ is a comparison event on a remote variable v, and if E_j contains a write to v, then $E_j \,|\, RFS$ also contains a write to v.
>
> **Proof of Claim:** If $e_{s^{j+1}}$ is a comparison event on v, then by (138) and Claim 3, we have $e'_{s^{j+1}} \in C_1(v) \cup C_2(v)$. By (142), no event in E_j may locally access v. Hence, by Claim 3, if an event e'_{s^k} (for some $k \leq j$) in E_j writes to v, then we have either $e'_{s^k} \in W(v)$ or $e'_{s^k} = SC(v)$. If $e'_{s^k} = SC(v)$, then since $s^k \in RFS$ holds by (141), Claim 4-1 is satisfied. On the other hand, if $e'_{s^k} \in W(v)$, then $W(v)$ is nonempty. Moreover, since all events in $W(v)$ are indexed before any events in $C_1(v) \cup C_2(v)$, E_j contains all events in $W(v)$. Thus, by (141) and the definition of LW, both E_j and $E_j \,|\, RFS$ contain $LW(v)$, an event that writes to v. □

We now apply Lemma 2 again, with 'H' $\leftarrow H' \circ L$, 'H'' $\leftarrow H' \circ L$, 'G' $\leftarrow E_j$, 'e_p' $\leftarrow e''_{s^{j+1}}$, and 'e'_p' $\leftarrow e'_{s^{j+1}}$. Among the assumptions stated in Lemma 2, (4)–(7) and (11)–(13) follow from (150), (144), (143), (143), (145), (149), and (152), respectively; (10) is trivial; (8) follows from (151) and by applying (138) with 's^j' $\leftarrow s^{j+1}$; (9) follows from (140) and $s^{j+1} \in S$. Moreover, Assumption (A) follows from (146), and Assumption (B) follows from Claim 4-1.

It follows that $e'_{s^{j+1}}$ is critical in $H' \circ L \circ E_j \circ \langle e'_{s^{j+1}} \rangle = H' \circ L \circ E_{j+1}$, and that $H' \circ L \circ E_{j+1}$ satisfies RF5. □

We now show that RFS is a valid RF-set of $\overline{G}$. Condition RF5 was already proved in Claim 4.

- **RF1 and RF2:** Define E_j as in Claim 4. We establish RF1 and RF2 by induction on j, applying Lemma 3 at each step. At each step, we assume

  - $H' \circ L \circ E_j$ satisfies RF1 and RF2. (154)

  The induction base ($j = 0$) follows easily from (143), since $E_0 = \langle\rangle$.

  Assume that (154) holds for a particular value of j. Assume that $e'_{s^{j+1}}$ remotely accesses variable v.

  By Claim 3, if $e'_{s^{j+1}}$ remotely reads a variable v, then the following holds: $s^{j+1} \in C_1(v) \cup C_2(v) \cup R(v)$; every event in $W(v)$ is contained in E_j; $\mathit{last\_writer}(v, H' \circ L \circ E_j)$ is one of $LW(v)$ or $SC(v)$ or $\perp$. Therefore, by (141), we have the following:

- if $e'_{s^{j+1}}$ remotely reads v, and if we let $q = \mathit{last\_writer}(v, H' \circ L \circ E_j)$, then either $q = \perp$ or $q \in \mathit{RFS}$ holds. (155)

We now apply Lemma 3, with 'H' $\leftarrow H' \circ L$, 'G' $\leftarrow E_j$, 'e_p' $\leftarrow e'_{s^{j+1}}$, and 'v_{rem}' $\leftarrow v$. Among the assumptions stated in Lemma 3, (14), (15), (17), (19), and (21) follow from (144), (143), (154), (145), and (155), respectively; (16) follows from (140) and $s^{j+1} \in S$; (18) follows from (140) and (139); (20) follows from (140) and (142). It follows that $H' \circ L \circ E_{j+1}$ satisfies RF1 and RF2.

- **RF3:** Consider a variable $v \in V$ and two different events f_p and g_q in $\overline{G}$. Assume that both p and q are in $\mathrm{Act}(\overline{G})$, $p \neq q$, that there exists a variable v such that $v \in \mathit{var}(f_p) \cup \mathit{var}(g_q)$, and that there exists a write to v in $\overline{G}$. Define $r = \mathit{last\_writer}(v, \overline{G})$. Our proof obligation is to show that $r \in \mathit{RFS}$.

  By (140), we have $\{p, q\} \subseteq S$. If there exists an event e'_s in E that remotely accesses v, then by Claim 3, $\mathit{last\_writer\_event}(v, \overline{G})$ is either $SC(v)$ (if $SC(v) \neq \perp$) or $LW(v)$ (otherwise). (Since we assumed that there exists a write to v, they both cannot be $\perp$.) Thus, by (141), we have the following:

  - if there exists an event e'_s in E such that e'_s remotely accesses v, then $r \in \mathit{RFS}$. (156)

  We now consider three cases.

  - Consider the case in which both f_p and g_q are in $H' \circ L$.
    If there exists an event e'_s in E such that $v \in \mathit{Wvar}(e'_s)$, then we claim that v is remote to s. Assume, to the contrary, that v is local to s. Since at least one of p or q is different from s, without loss of generality, assume $p \neq s$. Since $p \in S$ and, by (115), $S \subseteq \mathrm{Act}(H)$, we have $p \notin \mathrm{Fin}(H)$. Thus, by (133) and by applying RF2 with 'RFS' $\leftarrow \mathrm{Fin}(H)$ to f_p in $H' \circ L$, we have $s \notin \mathrm{Act}(H' \circ L)$. However, by (140), $\mathrm{Act}(H' \circ L) = S$, which contradicts $s \in S$ (which follows from (139), since e'_s is an event of E). It follows that v is remote to s, and hence $r \in \mathit{RFS}$ by (156).
    On the other hand, if there exists no event e'_s in E such that $v \in \mathit{Wvar}(e'_s)$ holds, then we have $r = \mathit{last\_writer}(v, H' \circ L)$. By (133) and applying RF3 with 'RFS' $\leftarrow \mathrm{Fin}(H)$ to f_p and g_q in $H' \circ L$, we have $\mathit{last\_writer}(v, H' \circ L) \in \mathrm{Fin}(H) \subseteq \mathit{RFS}$.
  - Consider the case in which f_p is in $H' \circ L$ and $g_q = e'_{s^k}$, for some $s^k \in S$. By (140) and our assumption that p and q are both in $\mathrm{Act}(\overline{G})$, we have $p \in \mathrm{Act}(H' \circ L)$ and $q \in \mathrm{Act}(H' \circ L)$. If v is local to q, then by (133), and by applying RF2 with 'RFS' $\leftarrow \mathrm{Fin}(H)$, to f_p in $H' \circ L$ we have $q \notin \mathrm{Act}(H' \circ L)$, a contradiction. Thus, v is remote to q, and hence $r \in \mathit{RFS}$ by (156).
  - Consider the case in which $f_p = e'_{s^j}$ and $g_q = e'_{s^k}$, for some s^j and s^k in S. Since v is remote to at least one of s^j or s^k, we have $r \in \mathit{RFS}$ by (156).

- **RF4:** By (60), (133), and (140), it easily follows that $\overline{G}$ satisfies RF4 with respect to RFS.

Therefore, we have established that

- RFS is a valid RF-set of $\overline{G}$. (157)

By (140) and (141), it follows that $\mathrm{Pmt}_{\mathit{RFS}}(\overline{G})$ consists of processes $\mathit{owner}(LW(v))$ and $\mathit{owner}(SC(v))$, for each variable $v \in V_{\mathrm{HC}}$. Thus, clearly $|\mathrm{Pmt}_{\mathit{RFS}}(\overline{G})| \leq 2|V_{\mathrm{HC}}|$ holds. Hence, from (67), we have

$$|\mathrm{Pmt}_{\mathit{RFS}}(\overline{G})| \leq n/(3 \log n). \tag{158}$$

We now let the processes in $\mathrm{Pmt}(\overline{G})$ finish their execution by inductively appending critical events of processes in $\mathrm{Pmt}(\overline{G})$, thus generating a sequence of computations $G_0, G_1, \ldots, G_k$ (where $G_0 = \overline{G}$), satisfying the following:

- $G_j \in C$; (159)
- RFS is a valid RF-set of G_j; (160)

- $\mathrm{Pmt}(G_j) \subseteq \mathrm{Pmt}(\overline{G})$, (161)
- each process in $\mathrm{Inv}(G_j)$ executes exactly $c+1$ critical events in G_j; (162)
- the processes in $\mathrm{Pmt}(\overline{G})$ collectively execute exactly $|\mathrm{Pmt}(\overline{G})| \cdot (c+1) + j$ critical events in G_j; (163)
- $\mathrm{Inv}(G_{j+1}) \subseteq \mathrm{Inv}(G_j)$ and $|\mathrm{Inv}(G_{j+1})| \geq |\mathrm{Inv}(G_j)| - 1$ if $j < k$; (164)
- $\mathrm{Fin}(G_j) \subsetneq \mathit{RFS}$ if $j < k$, and $\mathrm{Fin}(G_j) = \mathit{RFS}$ if $j = k$. (165)

At each induction step, we apply Lemma 6 to G_j in order to construct G_{j+1}, until $\mathrm{Fin}(G_j) = \mathit{RFS}$ is established, at which point the induction is completed. The induction is explained in detail below.

Induction base ($j = 0$): Since $G_0 = \overline{G}$, (159) and (160) follow from (137) and (157), respectively. Condition (161) is trivial.

By (51), (119), and (134), each process in S executes exactly c critical events in $H' \circ L$. Thus, by Claim 4, it follows that each process in S executes exactly $c+1$ critical events in $\overline{G}$. Since $\mathrm{Inv}(\overline{G}) \subseteq S$, $\overline{G}$ satisfies (162). Since $\mathrm{Pmt}(\overline{G}) \subseteq S$, $\overline{G}$ satisfies (163).

Induction step: At each step, we assume (159)–(163). If $\mathrm{Fin}(G_j) = \mathit{RFS}$, then (165) is satisfied and we finish the induction, by letting $k = j$.

Assume otherwise. We apply Lemma 6 with 'H' $\leftarrow G_j$. Assumptions (35)–(37) stated in Lemma 6 follow from (159), (160), and $\mathrm{Fin}(G_j) \neq \mathit{RFS}$. The lemma implies that a computation G_{j+1} exists satisfying (159)–(165), as shown below.

Condition (159) and (160) follow from (38) and (39), respectively. Since G_j satisfies (161), by (46), G_{j+1} also satisfies (161). Since $\mathrm{Inv}(G_{j+1}) \subseteq \mathrm{Inv}(G_j)$ by (40) and (41), by (43) and (47), and applying (162) to G_j, it follows that G_{j+1} satisfies (162). By (43)–(47), and applying (161) and (163) to G_j, it follows that G_{j+1} satisfies (163). Condition (164) follows from (40) and (41). Thus, the induction is established.

We now show that $k < n/3$. Assume otherwise. By applying (163) to G_k, it follows that there exists a process $p \in \mathrm{Pmt}(\overline{G})$ such that p executes at least $c + 1 + k/|\mathrm{Pmt}(\overline{G})|$ critical events in G_k. Because $k \geq n/3$, by (158), p executes at least $c + 1 + \log n$ critical events in G_k. From (165) and $p \in \mathrm{Pmt}(\overline{G}) \subseteq \mathit{RFS}$, we get $p \in \mathrm{Fin}(G_k)$. Hence, by (160), and by applying RF4 to p in G_k, it follows that the last event by p is Exit_p. Therefore, G_k can be written as $F \circ \langle \mathit{Exit}_p \rangle \circ \cdots$, where F is a prefix of G_k such that p executes at least $c + \log n$ critical events in F. However, p and F then satisfy Proposition Pr1, a contradiction.

Finally, we show that G_k satisfies Proposition Pr2. The following derivation establishes (53).

$$\begin{aligned}
|\mathrm{Act}(G_k)| &= |\mathrm{Inv}_{\mathit{RFS}}(G_k)| && \{\text{by (165), } \mathit{RFS} = \mathrm{Fin}(G_k)\text{, thus } \mathrm{Act}(G_k) = \mathrm{Inv}_{\mathit{RFS}}(G_k)\} \\
&\geq |\mathrm{Inv}_{\mathit{RFS}}(G_0)| - k && \{\text{by repeatedly applying (164)}\} \\
&= |\mathrm{Act}(\overline{G}) - \mathit{RFS}| - k && \{\text{by the definition of “Inv”; note that } \overline{G} = G_0\} \\
&= |S - \mathit{RFS}| - k && \{\text{by (140)}\} \\
&= |S - (\mathrm{Pmt}(\overline{G}) \cup \mathrm{Fin}(H))| - k \\
& && \{\text{because } \mathit{RFS} = \mathrm{Pmt}(\overline{G}) \cup \mathrm{Fin}(\overline{G})\text{, and } \mathrm{Fin}(\overline{G}) = \mathrm{Fin}(H) \text{ by (140)}\} \\
&= |S - \mathrm{Pmt}(\overline{G})| - k && \{\text{because } S \cap \mathrm{Fin}(H) = \{\} \text{ by (140)}\} \\
&= |(P_{\mathrm{HC}} - K) - \mathrm{Pmt}(\overline{G})| - k && \{\text{by (112)}\} \\
&\geq |P_{\mathrm{HC}}| - |K| - |\mathrm{Pmt}(\overline{G})| - k \\
&\geq \frac{|Y|}{2} - \frac{n}{6\log n} - \frac{n}{3\log n} - \frac{n}{3} && \{\text{by Case 2, (114), (158), and } k < n/3\} \\
&\geq \frac{n-1}{2} - \frac{n}{2\log n} - \frac{n}{3} && \{\text{by (56)}\} \\
&= \frac{n}{6} - \frac{n}{2\log n} - \frac{1}{2}.
\end{aligned}$$

```
H_1 := ⟨Enter_1, Enter_2, ..., Enter_N⟩;      n_1 := N;      j := 1;
repeat forever
LOOP INVARIANT: H_j ∈ C, H_j is regular, n_j = |Act(H_j)|, and each process in Act(H_j)
                executes exactly j critical events in H_j.
    if j > log n_j − 1 then
        let k := j, and exit the algorithm
    else      /* j ≤ log n − 1 */
        apply Lemma 7 with 'H' ← H_j;
        if (Pr1) holds then
            let k := j, and exit the algorithm
        else      /* (Pr2) holds */
            — There exists a regular computation G in C such that |Act(G)| = Ω(n_j / log^2 n_j) and each
              process in Act(G) executes exactly j + 1 critical events in G. Define Z = Act(G).
            H_{j+1} := G; n_{j+1} := |Z|; j := j + 1
    fi fi
od
```

Figure 10: Algorithm for constructing $H_1, H_2, \ldots, H_k$.

Moreover, by (160) and (165), we have $\mathrm{Act}(G_k) = \mathrm{Inv}(G_k)$. Thus, by (115), (140), and (164), we have $\mathrm{Act}(G_k) \subseteq \mathrm{Inv}(\overline{G}) \subseteq \mathrm{Act}(\overline{G}) = S \subseteq \mathrm{Act}(H)$, which implies (52). Finally, (162) implies (54). Therefore, G_k satisfies Proposition Pr2. $\square$

Theorem 2 *For any mutual exclusion system $\mathcal{S} = (C, P, V)$, there exist a process p in P and a computation H in C such that $H \circ \langle Exit_p \rangle \in C$, H does not contain $Exit_p$, and p executes $\Omega(\log N / \log\log N)$ critical events in H, where $N = |P|$.*

Proof: Let $H_1 = \langle Enter_1,\ Enter_2,\ \ldots,\ Enter_N \rangle$, where $P = \{1, 2, \ldots, N\}$. By the definition of a mutual exclusion system, $H_1 \in C$. It is obvious that H_1 is regular and each process in $\mathrm{Act}(H) = P$ has exactly one critical event in H_1. Starting with H_1, we repeatedly apply Lemma 7 and construct a sequence of computations $H_1,\ H_2,\ \ldots,\ H_k$, such that each process in $\mathrm{Act}(H_j)$ has j critical events in H_j. The construction algorithm is shown in Fig. 10.

For each computation H_j such that $1 \le j < k$, we have the following inequality:

$$n_{j+1} \ge \frac{cn_j}{\log^2 n_j} \ge \frac{cn_j}{\log^2 N},$$

where c is some fixed constant. This in turn implies

$$\log n_{j+1} \ge \log n_j - 2 \log\log N + \log c. \tag{166}$$

By iterating over $1 \le j < k$, and using $n_1 = N$, (166) implies

$$\log n_k \ge \log N - 2(k-1) \log\log N + (k-1) \log c. \tag{167}$$

We now consider two possibilities, depending on how the algorithm in Fig. 10 terminates. First, suppose that H_k satisfies $k > \log n_k - 1$. Combining this inequality with (167), we have

$$k > \frac{\log N + 2\log\log N - \log c - 1}{2 \log\log N - \log c + 1} = \Theta\left(\frac{\log N}{\log\log N}\right).$$

Therefore, each process in $\mathrm{Act}(H_k)$ executes $\Omega(\log N / \log\log N)$ critical events in H_k. By the Progress property, we can extend H_k to construct a computation that satisfies the theorem.

The other possibility is that $k \le \log n_k - 1$ holds and H_k satisfies Proposition Pr1. In this case, a process p and a computation F exist such that $F \circ \langle Exit_p \rangle \in C$, F does not contain $\langle Exit_p \rangle$, and p executes at least

```
Shared variables
    CheckPtr: 0..N − 1;
    Lock: boolean initially false;
    Priority: array[0..N − 1] of boolean initially false;
    Trying: array[0..N − 1] of boolean initially false

process p ::   /* 0 ≤ p < N */
while true do
0:   Noncritical Section;
1:   Trying[p] := true;
2:   repeat /* null */ until test_and_set(Lock) ∨ Priority[p];
3:   Priority[p] := false;
4:   Trying[p] := false;
5:   Critical Section;
6:   s := CheckPtr;
7:   CheckPtr := s + 1 mod N;
8:   if Trying[s] then
9:       Priority[s] := true
     else
10:      Lock := false
od
```

Figure 11: A mutual exclusion algorithm with $O(1)$ time complexity in LFCU systems. Each shared variable is remote to all processes.

$k + \log n_k$ critical events in F. By combining $k \le \log n_k - 1$ with (167), we have

$$\log n_k \ge \frac{\log N + 4 \log\log N - 2 \log c}{2 \log\log N - \log c + 1} = \Theta\left(\frac{\log N}{\log\log N}\right).$$

Therefore, computation F satisfies the theorem. □

5 A Constant-time Algorithm for LFCU Systems

In this section, we present a simple starvation-free mutual exclusion algorithm with $O(1)$ time complexity in LFCU systems, provided that each event that writes shared variables generates $O(1)$ interconnect traffic to update cached copies. In bus-based systems, this is a reasonable assumption, since an update message can be broadcast with a constant cost. On the other hand, in non-bus-based systems, an update may generate $\omega(1)$ interconnect traffic. However, the complexity involved in broadcasting a word-sized message in a non-bus-based network, while managing cache coherence, renders this approach exceedingly problematic. Indeed, we do not know of any commercial multiprocessor system that has a non-bus-based architecture and that uses a write-update cache protocol.

The algorithm, which is shown in Fig. 11, is a slight modification of the basic *test-and-set* lock. The atomic *test-and-set* primitive used in the algorithm is defined by the following pseudo-code.

```
test-and-set(bit: boolean) returns boolean
    if bit = false then bit := true; return true
    else return false
```

In the algorithm, two shared variables are used, *CheckPtr* and *Lock*, and two shared arrays, *Priority* and *Trying*. We assume that each of these variables is remote to all processes. Thus, our algorithm does not require the existence of locally-allocated shared memory as in a DSM system. Variable *Trying*[p] is *true* if and only if process p is executing within lines 2–4 of its entry section. Variable *Lock* is *false* if the *test-and-set* lock is available, and *true* otherwise. Variable *Priority*[p] is *true* if and only if p has been given priority to enter its critical section, as explained below. Variable *CheckPtr* cycles through $0..N-1$ in order to determine the process to be given priority.

When a process p leaves its noncritical section, it sets $Trying[p] = true$ at line 1. It then enters the busy-waiting loop at line 2. Process p may enter its critical section either by performing a successful *test-and-set* or by finding $Priority[p] = true$ at line 2. The *Priority* variables are used to prevent starvation: if other processes execute concurrently with p, then there is a possibility that p's *test-and-set* always fails. In order to prevent starvation, variable *CheckPtr* cycles through $0..N-1$, as seen in lines 6 and 7. (Note that these lines are executed before any other process is allowed to enter its critical section, so *CheckPtr* is incremented sequentially.) If p continues to wait at line 2 while other processes execute their critical sections, then eventually (specifically, after at most N critical-section executions) some process reads $CheckPtr = p - 1 \ \mathbf{mod}\ N$ at line 6, and establishes $Priority[p] = true$ at line 9. To prevent violations of the Exclusion property, *Lock* is not changed to *false* in this case. This idea of giving priority to processes that might otherwise wait forever is also used in [7, 9].

It is straightforward to formalize these arguments and prove that the algorithm of Fig. 11 is a correct, starvation-free mutual exclusion algorithm. We now prove that its time complexity is $O(1)$ per critical-section execution in LFCU systems. Clearly, time complexity is dominated by the number of remote memory references generated by line 2. In an LFCU system, the *test-and-set* invocations in line 2 generate $O(1)$ remote memory references. This is because a failed *test-and-set* generates a cached copy of *Lock*, and any subsequent update of *Lock* by a process at line 10 updates this cached copy. The reads of $Priority[p]$ in line 2 also generate $O(1)$ remote memory references. In particular, the first read of $Priority[p]$ creates a cached copy. If $Priority[p] = true$, then the loop terminates. If $Priority[p] = false$, then subsequent reads of $Priority[p]$ are handled in-cache, until $Priority[p]$ is updated by another process. Other processes update $Priority[p]$ only by establishing $Priority[p] = true$. Once this is established, p's busy-waiting loop terminates. We conclude that the algorithm generates $O(1)$ remote memory references per critical-section execution in LFCU systems, as claimed.

6 Concluding Remarks

We have established a lower bound of $\Omega(\log N/\log\log N)$ remote memory references for mutual exclusion algorithms based on reads, writes, or comparison primitives; for algorithms with comparison primitives, this bound only applies in non-LFCU systems. Our bound improves an earlier lower bound of $\Omega(\log\log N/\log\log\log N)$ established by Cypher. We conjecture that $\Omega(\log N)$ is a tight lower bound for the class of algorithms and systems to which our lower bound applies; this conjecture remains an open issue.

It should be noted that Cypher's result guarantees that there exists no algorithm with *amortized* $\Theta(\log\log N/\log\log\log N)$ time complexity, while ours does not. This is because his bound is obtained by counting the total number of remote memory references in a computation, and by then dividing this number by the number processes participating in that computation. In contrast, our result merely proves that there exists a computation H and a process p such that p executes $\Omega(\log N/\log\log N)$ critical events in H. Therefore, our result leaves open the possibility that the *average* number of remote memory references per process is less than $\Theta(\log N/\log\log N)$. We leave this issue for future research.

There has been much recent interest in adaptive mutual exclusion algorithms [2, 7, 9, 13, 15, 17]. In such algorithms, time complexity (under some measure) is required to be a function of the number of contending processes. In recent work [13], we used proof techniques similar to those presented in this paper to establish the following lower-bound result pertaining to adaptive algorithms.

> *For any k, there exists some N such that, for any N-process mutual exclusion algorithm based on reads, writes, or conditional primitives, a computation exists involving $\Theta(k)$ processes in which some process executes $\Omega(k)$ remote operations to enter and exit its critical section.*

This result precludes an algorithm with $O(\log k)$ time complexity, where k is "point contention," *i.e.*, the maximum number of processes concurrently active at the *same* state. The problem of designing an $O(\log k)$ algorithm using only reads and writes had been mentioned previously in at least two papers [2, 7]. The lower-bound result quoted above shows that such an algorithm cannot exist. It is important to point out that this result does not establish $\Omega(k)$ as a lower bound. Instead, it shows that $\Omega(k)$ time complexity is required *provided* N is sufficiently large. Deducing a precise characterization of time complexity as a function of contention is yet another interesting open issue.

In other recent work [4], we considered algorithms in which memory is accessed by nonatomic read and write operations; such an operation executes over an interval of time and therefore several such operations by different processes may overlap one another. For any such algorithm, we showed that there exists a single-process execution in which the lone competing process performs $\Omega(\log N/\log\log N)$ remote memory references that access $\Omega(\sqrt{\log N/\log\log N})$ distinct variables in order to enter its critical section, where N is the number of processes. These bounds show that adaptive algorithms are impossible if variable accesses are nonatomic, even if caching techniques are used to avoid interconnect accesses. In the same paper, we presented a nonatomic algorithm with $\Theta(\log N)$ RMR time complexity. We believe that the above-mentioned $\Omega(\log N/\log\log N)$ lower bound for nonatomic algorithms can be improved to $\Omega(\log N)$, matching the algorithm, but establishing this remains as future work.

It is possible to generalize our lower-bound proof for systems with multi-valued and/or multi-variable comparison primitives. *Two-valued compare-and-swap* (2VCAS) and *double compare-and-swap* (DCAS) are examples of such primitives. 2VCAS uses two compare values a and b; a single variable v is compared to both and a new value is written to v if either comparison succeeds. DCAS operates on two different variables u and v, using two associated compare values a and b, respectively; new values are assigned to u and v if and only if both $u = a$ and $v = b$ hold. In order to adapt our proof for systems in which such primitives are used, only the following change is needed: in the roll-forward strategy, for each variable, we select $O(1)$ processes with successful comparison events, instead of just one, and let all other processes execute unsuccessful comparison events. Therefore, we now roll $O(1)$ processes forward per variable. With this change, our asymptotic lower bound remains unchanged. In a recent paper [3], we showed that there exists a class of comparison primitives that includes 2VCAS for which a $\Theta(\log N/\log\log N)$ algorithm is possible. (To the best of our knowledge, none of the primitives in this class has been implemented on a real machine. DCAS, which was supported on some generations of the Motorola 68000 processor family, is not in this class.) Thus, there exist comparison primitives for which our lower bound is tight.

Acknowledgements: We are grateful to Faith Fich for her comments on an earlier version of this paper. We are also grateful to the anonymous referees for their diligence in reviewing such a long paper and for the helpful suggestions they made.

References

[1] Y. Afek, P. Boxer, and D. Touitou. Bounds on the shared memory requirements for long-lived and adaptive objects. In *Proceedings of the 19th Annual ACM Symposium on Principles of Distributed Computing*, pages 81–89. ACM, July 2000.

[2] J. Anderson and Y.-J. Kim. Adaptive mutual exclusion with local spinning. In *Proceedings of the 14th International Symposium on Distributed Computing*, pages 29–43. Lecture Notes in Computer Science 1914, Springer, October 2000.

[3] J. Anderson and Y.-J. Kim. Local-spin mutual exclusion using fetch-and-ϕ primitives. Manuscript, 2002.

[4] J. Anderson and Y.-J. Kim. Nonatomic mutual exclusion with local spinning. In *Proceedings of the 21st Annual ACM Symposium on Principles of Distributed Computing*, pages 3–12. ACM, July 2002.

[5] J. Anderson and J.-H. Yang. Time/contention tradeoffs for multiprocessor synchronization. *Information and Computation*, 124(1):68–84, January 1996.

[6] T. Anderson. The performance of spin lock alternatives for shared-memory multiprocessors. *IEEE Transactions on Parallel and Distributed Systems*, 1(1):6–16, January 1990.

[7] H. Attiya and V. Bortnikov. Adaptive and efficient mutual exclusion. In *Proceedings of the 19th Annual ACM Symposium on Principles of Distributed Computing*, pages 91–100. ACM, July 2000.

[8] J. Burns and N. Lynch. Mutual exclusion using indivisible reads and writes. In *Proceedings of the 18th Annual Allerton Conference on Communication, Control, and Computing*, pages 833–842, 1980.

[9] M. Choy and A. Singh. Adaptive solutions to the mutual exclusion problem. *Distributed Computing*, 8(1):1–17, 1994.

[10] R. Cypher. The communication requirements of mutual exclusion. In *Proceedings of the Seventh Annual Symposium on Parallel Algorithms and Architectures*, pages 147–156, June 1995.

[11] E. Dijkstra. Solution of a problem in concurrent programming control. *Communications of the ACM*, 8(9):569, 1965.

[12] G. Graunke and S. Thakkar. Synchronization algorithms for shared-memory multiprocessors. *IEEE Computer*, 23:60–69, June 1990.

[13] Y.-J. Kim and J. Anderson. A time complexity bound for adaptive mutual exclusion. In *Proceedings of the 15th International Symposium on Distributed Computing*, pages 1–15. Lecture Notes in Computer Science 2180, Springer, October 2001.

[14] J. Mellor-Crummey and M. Scott. Algorithms for scalable synchronization on shared-memory multiprocessors. *ACM Transactions on Computer Systems*, 9(1):21–65, February 1991.

[15] M. Merritt and G. Taubenfeld. Speeding Lamport's fast mutual exclusion algorithm. *Information Processing Letters*, 45:137–142, 1993.

[16] D. Patterson and J. Hennessy. *Computer Organization and Design: The Hardware/Software Interface.* Morgan Kaufmann Publishers, second edition, 1997.

[17] E. Styer. Improving fast mutual exclusion. In *Proceedings of the 11th Annual ACM Symposium on Principles of Distributed Computing*, pages 159–168. ACM, August 1992.

[18] E. Styer and G. Peterson. Tight bounds for shared memory symmetric mutual exclusion. In *Proceedings of the 8th Annual ACM Symposium on Principles of Distributed Computing*, pages 177–191. ACM, August 1989.

[19] P. Turán. On an extremal problem in graph theory (in Hungarian). *Mat. Fiz. Lapok*, 48:436–452, 1941.

[20] J.-H. Yang and J. Anderson. Fast, scalable synchronization with minimal hardware support. In *Proceedings of the 12th Annual ACM Symposium on Principles of Distributed Computing*, pages 171–182. ACM, August 1993.

Appendix: Detailed Proofs for Lemmas 1–6

In this appendix, full proofs are presented for Lemmas 1–6. As before, we omit RFS when quoting properties RF1–RF5 and assume the existence of a fixed mutual exclusion system $\mathcal{S} = (C, P, V)$.

Lemma 1 *Consider a computation H and two sets of processes RFS and Y. Assume the following:*

- $H \in C$; (1)
- *RFS is a valid RF-set of H;* (2)
- $RFS \subseteq Y$. (3)

Then, the following hold: $H \,|\, Y \in C$; RFS is a valid RF-set of $H \,|\, Y$; an event e in $H \,|\, Y$ is a critical event if and only if it is also a critical event in H.

Proof: By (2), H satisfies RF1–RF5. Since H satisfies RF1, if a process p is not in RFS, no process other than p reads a value written by p. Therefore, by inductively applying P2, we have $H \,|\, Y \in C$.

We now prove that RFS is a valid RF-set of $H \,|\, Y$.

- **RF1:** Assume that H can be written as $E \circ \langle e_p \rangle \circ F \circ \langle f_q \rangle \circ G$, and that $H \,|\, Y$ can be written as $(E \,|\, Y) \circ \langle e_p \rangle \circ (F \,|\, Y) \circ \langle f_q \rangle \circ (G \,|\, Y)$. Also assume that $p \neq q$ and that there exists a variable $v \in \mathit{Wvar}(e_p) \cap \mathit{Rvar}(f_q)$ such that $F \,|\, Y$ does not contain a write to v. We claim that F does not contain a write to v, in which case, by applying RF1 to H, we have $p \in RFS$.

  Assume, to the contrary, that F has a write to v. Define $g_r = \mathit{last\_writer\_event}(v, F)$. Since $F \,|\, Y$ does not contain g_r, we have $r \notin Y$. Since $H \,|\, Y$ contains f_q, we have $q \in Y$. Therefore, $r \neq q$. By applying RF1 to g_r and f_q in H, we have $r \in RFS$, which contradicts $r \notin Y$.

- **RF2:** Consider an event e_p in $H \,|\, Y$, and a variable v in $\mathit{var}(e_p)$. Assume that v is local to another process q ($\neq p$). By applying RF2 to H, we have either $q \notin \mathrm{Act}(H)$ or $\{p, q\} \subseteq RFS$. Since $\mathrm{Act}(H \,|\, Y) \subseteq \mathrm{Act}(H)$, $q \notin \mathrm{Act}(H)$ implies $q \notin \mathrm{Act}(H \,|\, Y)$.

- **RF3:** Consider a variable $v \in V$ and two different events e_p and f_q in $H \,|\, Y$. Assume that both p and q are in $\mathrm{Act}(H \,|\, Y)$, $p \neq q$, there exists a variable v such that $v \in \mathit{var}(e_p) \cap \mathit{var}(f_q)$, and there exists a write to v in $H \,|\, Y$. We claim that $\mathit{last\_writer}(v, H \,|\, Y) \in RFS$ holds.

  By definition, events e_p and f_q also exist in H, and there exists a write to v in H. Since $\mathrm{Act}(H \,|\, Y) \subseteq \mathrm{Act}(H)$, both p and q are in $\mathrm{Act}(H)$. Therefore, by applying RF3 to H, we have $\mathit{last\_writer}(v, H) \in RFS$. Thus, since $RFS \subseteq Y$, the last event to write to v is identical in both H and $H \,|\, Y$. Hence, $\mathit{last\_writer}(v, H \,|\, Y) \in RFS$ holds.

- **RF4:** From $\mathrm{Act}(H \,|\, Y) \subseteq \mathrm{Act}(H)$, $\mathrm{Fin}(H \,|\, Y) \subseteq \mathrm{Fin}(H)$, and the fact that H satisfies RF4, it easily follows that $H \,|\, Y$ satisfies RF4.

- **RF5:** If an event e_p in $H \,|\, Y$ is critical in $H \,|\, Y$, then by the definition of a critical event, it is also critical in H. (Note that every event that is contained in H but not in $H \,|\, Y$ is executed by some process different from p. Adding such events to $H \,|\, Y$ cannot make e_p noncritical.) Thus, by applying RF5 to H, e_p is also critical in $H \,|\, (\{p\} \cup RFS)$.

Finally, we claim that an event e_p in $H \,|\, Y$ is a critical event if and only if it is also a critical event in H. If e_p is critical in $H \,|\, Y$, then as shown above (in the reasoning for RF5), it is also critical in H. On the other hand, if e_p is critical in H, then by applying RF5 to H, it is also critical in $H \,|\, (\{p\} \cup RFS)$. Since e_p is an event of $H \,|\, Y$, we have $p \in Y$, and hence $\{p\} \cup RFS \subseteq Y$. Thus, by the definition of a critical event, e_p is also critical in $H \,|\, Y$. □

Lemma 2 *Consider three computations H, H', and G, a set of processes RFS, and two events e_p and e'_p of a process p. Assume the following:*

- *$H \circ \langle e_p \rangle \in C$;* (4)
- *$H' \circ G \circ \langle e'_p \rangle \in C$;* (5)
- *RFS is a valid RF-set of H;* (6)
- *RFS is a valid RF-set of H';* (7)
- *$e_p \sim e'_p$;* (8)
- *$p \in \mathrm{Act}(H)$;* (9)
- *$H \,|\, (\{p\} \cup RFS) = H' \,|\, (\{p\} \cup RFS)$;* (10)
- *$G \,|\, p = \langle\rangle$;* (11)
- *no events in G write any of p's local variables;* (12)
- *e_p is critical in $H \circ \langle e_p \rangle$.* (13)

Then, e'_p is critical in $H' \circ G \circ \langle e'_p \rangle$. Moreover, if the following conditions are true,

(A) *$H' \circ G$ satisfies RF5;*

(B) *if e_p is a comparison event on a variable v, and if G contains a write to v, then $G \,|\, RFS$ also contains a write to v.*

then $H' \circ G \circ \langle e'_p \rangle$ also satisfies RF5.

Proof: First, define

$$\overline{H} = H' \circ G \circ \langle e'_p \rangle. \tag{168}$$

Note that, by (9), (10) and (11), we have the following:

$$\begin{aligned} H \,|\, p &= (H' \circ G) \,|\, p, \quad \text{and} && (169) \\ p &\in \mathrm{Act}(H'). && (170) \end{aligned}$$

If Condition (A) is true, then in order to show that $\overline{H}$ satisfies RF5, it suffices to consider event e'_p. If e'_p is a critical read or transition event, then it is clearly critical in $\overline{H} \,|\, (\{p\} \cup RFS)$. Thus, our remaining proof obligations are to show

- e'_p is critical in $\overline{H}$;
- if e'_p is a critical write or a critical comparison, and if Condition (B) is true, then e'_p is also a critical write or a critical comparison in $\overline{H} \,|\, (\{p\} \cup RFS)$.

We consider four cases, depending on the kind of critical event e_p is.

Transition event: If e_p is a transition event, then by (8) and the definition of congruence, e'_p is also a transition event.

Critical read: If e_p is a critical read in $H \circ \langle e_p \rangle$, then there exists a variable v, remote to p, such that $op(e_p) = \mathsf{read}(v)$ and $H \,|\, p$ does not contain a read from v. Thus, by (8) and (169), e'_p is also a critical read in $\overline{H}$.

Before considering the remaining two cases, note that, if e_p is a critical write or a critical comparison in $H \circ \langle e_p \rangle$, then there exists a variable v, remote to p, such that $v \in var(e_p)$ and

$$last\_writer(v, H) \neq p. \tag{171}$$

Critical write: Assume that e_p is a critical write in $H \circ \langle e_p \rangle$. We consider two cases. First, if p does not write v in H, then by (169),

- p does not write v in $H' \circ G$. (172)

Thus, we have $\mathit{last\_writer}(v, H' \circ G) \neq p$, and hence, by (8) and the definition of a critical write, e'_p is a critical write in $\overline{H}$. Moreover, by (172), we also have $\mathit{last\_writer}(v, (H' \circ G) \,|\, (\{p\} \cup \mathit{RFS})) \neq p$. Thus, e'_p is also a critical write in $\overline{H} \,|\, (\{p\} \cup \mathit{RFS})$.

Second, if p writes to v in H, then define $\bar{e}_p$ to be the last event by p that writes to v in H. Define $q = \mathit{last\_writer}(v, H)$ and $f_q = \mathit{last\_writer\_event}(v, H)$. By (171), we have $q \neq p$ and $q \neq \perp$. If $q \in \mathrm{Act}(H)$, then by (6) and (9), and applying RF3 to $\bar{e}_p$ and f_q in H, we have $q \in \mathit{RFS}$. On the other hand, if $q \in \mathrm{Fin}(H)$, then clearly $q \in \mathit{RFS}$ by (6). Thus, in either case, in H, there exists a write to v (that is, f_q) by some process in RFS after the last write to v by p (that is, $\bar{e}_p$). Therefore, by (10), the same is true in H', and hence by (11), we have $\mathit{last\_writer}(v, H' \circ G) \neq p$. It follows, by (8) and the definition of a critical write, that e'_p is a critical write in $\overline{H}$.

Moreover, since $q \in \mathit{RFS}$, there also exists a write to v (that is, f_q) after the last write to v by p ($\bar{e}_p$) in $\overline{H} \,|\, (\{p\} \cup \mathit{RFS})$. It follows that e'_p is also a critical write in $\overline{H} \,|\, (\{p\} \cup \mathit{RFS})$.

Critical comparison: Assume that e_p is a critical comparison on v in $H \circ \langle e_p \rangle$. We consider three cases.

Case 1: If G contains a write to v, then by (11) and the definition of a critical comparison, e'_p is clearly a critical comparison in $\overline{H}$. Moreover, if (B) is true, then since G contains a write to v, $G \,|\, \mathit{RFS}$ also contains a write to v, and hence e'_p is also a critical comparison in $\overline{H} \,|\, (\{p\} \cup \mathit{RFS})$.

Case 2: If p does not access v in H (*i.e.*, for every event f_p in $H \,|\, p$, $v \notin \mathit{var}(f_p)$), then by (169), p does not access v in $H' \circ G$. Thus, by (8) and the definition of a critical comparison, e'_p is a critical comparison in $\overline{H}$. Moreover, it is clear that p does not access v in $(H' \circ G) \,|\, (\{p\} \cup \mathit{RFS})$. Thus, e'_p is also a critical comparison in $\overline{H} \,|\, (\{p\} \cup \mathit{RFS})$.

Case 3: Assume that p accesses v in H, and that G does not contain a write to v. Then, there exists an event f_p in H such that $v \in \mathit{var}(f_p)$. Moreover, since v is the only remote variable in $\mathit{var}(e_p)$, by (12), it follows that

- for each variable u in $\mathit{Rvar}(e_p)$, G does not contain a write to u. (173)

We now establish two claims.

Claim 1: $\mathit{last\_writer}(v, H) = \mathit{last\_writer}(v, H')$ and $\mathit{last\_writer\_event}(v, H) = \mathit{last\_writer\_event}(v, H')$. Moreover, either both $\mathit{last\_writer}(v, H)$ and $\mathit{last\_writer}(v, H')$ are $\perp$, or both are in $\{p\} \cup \mathit{RFS}$.

Proof of Claim: Suppose that v is written in H and let g_{q_v} be the last event to do so. If $q_v \neq p$ and if $q_v \in \mathrm{Act}(H)$, then by (6) and (9), and by applying RF3 to f_p and g_{q_v} in H, we have $q_v \in \mathit{RFS}$. On the other hand, if $q_v \in \mathrm{Fin}(H)$, then clearly $q_v \in \mathit{RFS}$ holds by (6). Finally, if $q_v = p$, then clearly we have $q_v \in \{p\} \cup \mathit{RFS}$. Thus, in any case, we have $q_v \in \{p\} \cup \mathit{RFS}$. Similarly, if $g_{q'_v}$ is the last event to write v in H', then by (7) and (170), we have $q'_v \in \{p\} \cup \mathit{RFS}$. Therefore, by (10), it follows that the last event to write v (if any) is the same in H and H'. Hence, the claim follows. $\square$

Claim 2: For each variable u in $\mathit{Rvar}(e_p)$, $\mathit{last\_writer\_event}(u, H) = \mathit{last\_writer\_event}(u, H')$ holds.

Proof of Claim: If u is remote to p, then by the Atomicity Property, we have $u = v$. Thus, in this case, the claim follows by Claim 1.

So, assume that u is local to p. If there exists an event g_{q_u} in H that writes u, then by (6) and (9), and by applying RF2 to g_{q_u} in H, it follows that either $q_u = p$ or $q_u \in RFS$ holds. Similarly, if an event $g_{q'_u}$ writes u in H', then by (7) and (170), we have either $q'_u = p$ or $q'_u \in RFS$. Therefore, by (10), it follows that the last event to write u (if any) is the same in H and H'. $\square$

From (173) and Claim 2, it follows that

- for each variable u in $Rvar(e_p)$, $value(u, H) = value(u, H' \circ G)$ holds. (174)

By (4), (169), (174), and P2, we have $H' \circ G \circ \langle e_p \rangle \in C$. Combining with (5), and using P5, we have $e_p = e'_p$. In particular,

- e'_p is a successful (respectively, unsuccessful) comparison in $\overline{H}$ if and only if e_p is also a successful (respectively, unsuccessful) comparison in $H \circ \langle e_p \rangle$. (175)

Define $q = last\_writer(v, H)$ and $g_q = last\_writer\_event(v, H)$. By (171) and Claim 1, we have

$$q = \bot \quad \vee \quad q \in RFS - \{p\}, \tag{176}$$

and $g_q = last\_writer\_event(v, H')$. Thus, by (10),

$$q \neq \bot \;\Rightarrow\; H \,|\, \{p, q\} = H' \,|\, \{p, q\}. \tag{177}$$

Since G does not contain a write to v, we also have

$$last\_writer(v, H' \circ G) = q \quad \wedge \quad last\_writer\_event(v, H' \circ G) = g_q. \tag{178}$$

If e_p is a critical successful comparison in $H \circ \langle e_p \rangle$, then by (175), (176), (178), and the definition of a critical successful comparison, e'_p is clearly a critical successful comparison in both $\overline{H}$ and $\overline{H} \,|\, (\{p\} \cup RFS)$.

Similarly, if e_p is a critical unsuccessful comparison in $H \circ \langle e_p \rangle$, then by definition, either $H \,|\, p$ does not contain an unsuccessful comparison event by p on v, or $q \neq \bot$ and H does not contain an unsuccessful comparison event by p on v *after* g_q. In the former case, by (169), $(H' \circ G) \,|\, p$ does not contain an unsuccessful comparison event on v; in the latter case, by (11) and (177), $H' \circ G$ does not contain an unsuccessful comparison event by p on v *after* g_q.

Therefore, In either case, by (175), it follows that e'_p is a critical unsuccessful comparison in both $\overline{H}$ and $\overline{H} \,|\, (\{p\} \cup RFS)$. $\square$

Lemma 3 *Consider two computations H and G, a set of processes RFS, and an event e_p of a process p. Assume the following:*

- *$H \circ G \circ \langle e_p \rangle \in C$;* (14)
- *RFS is a valid RF-set of H;* (15)
- *$p \in \mathrm{Act}(H)$;* (16)
- *$H \circ G$ satisfies RF1 and RF2;* (17)
- *G is an $\mathrm{Act}(H)$-computation;* (18)
- *$G \,|\, p = \langle\rangle$;* (19)
- *if e_p remotely accesses a variable v_{rem}, then the following hold:*
  - *if v_{rem} is local to a process q, then either $q \notin \mathrm{Act}(H)$ or $\{p, q\} \subseteq RFS$, and* (20)
  - *if $q = last\_writer(v_{\mathrm{rem}}, H \circ G)$, then one of the following hold: $q = \bot$, $q = p$, $q \in RFS$, or $v_{\mathrm{rem}} \notin Rvar(e_p)$.* (21)

Then, $H \circ G \circ \langle e_p \rangle$ satisfies RF1 and RF2.

Proof: Define

$$\overline{H} = H \circ G \circ \langle e_p \rangle. \tag{179}$$

By (16) and (19), we have

$$p \in \text{Act}(H \circ G). \tag{180}$$

By (18), we also have $\text{Act}(H \circ G) \subseteq \text{Act}(H)$. Thus, by (180),

$$\text{Act}(\overline{H}) \subseteq \text{Act}(H \circ G) \subseteq \text{Act}(H). \tag{181}$$

Now we prove each of RF1 and RF2 separately.

- **RF1:** Since, by (17), $H \circ G$ satisfies RF1, it suffices to consider the following case: $\overline{H}$ can be written as $E \circ \langle f_q \rangle \circ F \circ \langle e_p \rangle$; $p \neq q$; there exists a variable $v \in \mathit{Wvar}(f_q) \cap \mathit{Rvar}(e_p)$; and F does not contain a write to v. Our proof obligation is to show $q \in \mathit{RFS}$.

  If v is local to p, then by (17), and applying RF2 to f_q in $H \circ G$, we have either $p \notin \text{Act}(H \circ G)$ or $q \in \mathit{RFS}$. Thus, by (180), we have $q \in \mathit{RFS}$. On the other hand, if v is remote to p, then we have $v_{\text{rem}} = v$, which implies $q = \mathit{last\_writer}(v_{\text{rem}}, H \circ G)$, $q \neq \perp$, $q \neq p$, and $v_{\text{rem}} \in \mathit{Rvar}(e_p)$. Thus, by (21), we have $q \in \mathit{RFS}$.

- **RF2:** Consider an event f_q in $\overline{H}$, and a variable v in $\mathit{var}(f_q)$. Assume that v is local to another process $r \neq q$. Our proof obligation is to show that either $r \notin \text{Act}(\overline{H})$ or $\{q, r\} \subseteq \mathit{RFS}$ holds.

  If f_q is an event of $H \circ G$, then by (17), and applying RF2 to f_q in $H \circ G$, we have either $r \notin \text{Act}(H \circ G)$ or $\{q, r\} \subseteq \mathit{RFS}$. By (181), $r \notin \text{Act}(H \circ G)$ implies $r \notin \text{Act}(\overline{H})$.

  On the other hand, if $f_q = e_p$, then we have $p = q$ and $v_{\text{rem}} = v$. By applying (20) with 'q' $\leftarrow r$, we have either $r \notin \text{Act}(H)$ or $\{p, r\} = \{q, r\} \subseteq \mathit{RFS}$. By (181), $r \notin \text{Act}(H)$ implies $r \notin \text{Act}(\overline{H})$. $\square$

In order to prove Lemma 4, we need several more lemmas, presented here. The next lemma shows that appending a noncritical event of an active process does not invalidate a valid RF-set.

Lemma A1 *Consider a computation H, a set of processes RFS, and an event e_p of a process p. Assume the following:*

- *$H \circ \langle e_p \rangle \in C$;* (182)
- *RFS is a valid RF-set of H;* (183)
- *$p \in \text{Act}(H)$;* (184)
- *e_p is noncritical in $H \circ \langle e_p \rangle$.* (185)

Then, RFS is a valid RF-set of $H \circ \langle e_p \rangle$.

Proof: First, note that $\text{Act}(H) = \text{Act}(H \circ \langle e_p \rangle)$ and $\text{Fin}(H) = \text{Fin}(H \circ \langle e_p \rangle)$, because $p \in \text{Act}(H)$ and $e_p \neq \mathit{Exit}_p$. If e_p remotely accesses a remote variable, then we will denote that variable as v_{rem}. By (185) and the definition of a critical event,

- if e_p remotely accesses v_{rem}, then there exists an event $\bar{e}_p$ in H that remotely accesses v_{rem}. (186)

Thus, by (183), and applying RF2 to $\bar{e}_p$ in H, it follows that

- if e_p remotely accesses v_{rem} and v_{rem} is local to another process q, then either $q \notin \text{Act}(H)$ or $\{p, q\} \subseteq \mathit{RFS}$ holds. (187)

Define $z = \mathit{last\_writer}(v_{\mathrm{rem}}, H)$ and $f_z = \mathit{last\_writer\_event}(v_{\mathrm{rem}}, H)$. We claim that one of the following holds: $z = \perp$, $z = p$, or $z \in \mathit{RFS}$. Assume, to the contrary, that $z \neq \perp$, $z \neq p$, and $z \notin \mathit{RFS}$. Then, by (183), $z \notin \mathit{RFS}$ implies $z \notin \mathrm{Fin}(H)$. Since $H \,|\, z \neq \langle\rangle$, we have $z \in \mathrm{Act}(H)$. Thus, by (184), and applying RF3 to $\bar{e}_p$ and f_z in H, we have $z \in \mathit{RFS}$, a contradiction. Thus, we have shown that

- if e_p remotely accesses v_{rem}, and if $z = \mathit{last\_writer}(v_{\mathrm{rem}}, H)$, then one of the following hold: $z = \perp$, $z = p$, or $z \in \mathit{RFS}$. (188)

We now consider each condition RF1–RF5 separately.

- **RF1 and RF2:** Define $G = \langle\rangle$. It follows trivially from (183) that

  - $H \circ G$ satisfies RF1 and RF2. (189)

  We now apply Lemma 3. Assumptions (14)–(21) stated in Lemma 3 follow from (182), (183), (184), (189), $G = \langle\rangle$, $G = \langle\rangle$, (187), and (188), respectively. It follows that $H \circ \langle e_p \rangle$ satisfies RF1 and RF2.

- **RF3:** Consider a variable $v \in V$ and two different events f_q and g_r in $H \circ \langle e_p \rangle$. Assume that both q and r are in $\mathrm{Act}(H) = \mathrm{Act}(H \circ \langle e_p \rangle)$, $q \neq r$, that there exists a variable v such that $v \in \mathit{var}(f_q) \cap \mathit{var}(g_r)$, and that there exists a write to v in $H \circ \langle e_p \rangle$. Let

  $$s = \mathit{last\_writer}(v, H \circ \langle e_p \rangle). \tag{190}$$

  Our proof obligation is to show that $s \in \mathit{RFS}$ holds.

  - First, assume that v is local to p. Since at least one of q or r is different from p, by (183) and (184), and applying RF2 to H, we have $p \in \mathit{RFS}$. If $s = p$, then we have $s \in \mathit{RFS}$. On the other hand, if $s \neq p$, then (190) implies that e_p does not write v. Hence, we have $s = \mathit{last\_writer}(v, H)$. Therefore, by (183) and (184), and applying RF2 to $\mathit{last\_writer\_event}(v, H)$, we have $s \in \mathit{RFS}$.
  - Second, assume that v is remote to p, and consider the case in which both f_q and g_r are in H. If $v \in \mathit{Wvar}(e_p)$, then by (185) and (190), we have $\mathit{last\_writer}(v, H) = s = p$. (Otherwise, e_p would be either a critical write or a critical successful comparison by definition.) On the other hand, if $v \notin \mathit{Wvar}(e_p)$, then clearly we have $\mathit{last\_writer}(v, H) = s$. Therefore, in either case, there is a write to v in H. Therefore, by (183), and applying RF3 to f_q and g_r in H, we have $\mathit{last\_writer}(v, H) \in \mathit{RFS}$, and hence $s \in \mathit{RFS}$.
  - Third, assume that v is remote to p, and consider the case in which one of f_q or g_r is e_p. Without loss of generality, we can assume that f_q is in H, $g_r = e_p$, $v = v_{\mathrm{rem}}$, and $q \neq p$. We consider two cases.

    If e_p writes v_{rem}, then by (190), we have $s = p$. Moreover, by (185), we have $\mathit{last\_writer}(v_{\mathrm{rem}}, H) = p$. (Otherwise, e_p would be either a critical write or a critical successful comparison by definition.) Applying RF3 to $\mathit{last\_writer\_event}(v, H)$ (by p) and f_q in H, we have $\mathit{last\_writer}(v_{\mathrm{rem}}, H) \in \mathit{RFS}$, which implies $s \in \mathit{RFS}$.

    Otherwise, if e_p does not write v_{rem}, then since $v_{\mathrm{rem}} \in \mathit{var}(e_p)$, we have $v_{\mathrm{rem}} \in \mathit{Rvar}(e_p)$. By (183), (184), and (186), and applying RF3 to $\bar{e}_p$ and f_q, we have $\mathit{last\_writer}(v_{\mathrm{rem}}, H) \in \mathit{RFS}$. Since e_p does not write to v_{rem}, we have $s = \mathit{last\_writer}(v_{\mathrm{rem}}, H) \in \mathit{RFS}$.

- **RF4:** Since H satisfies RF4, and since e_p is not one of Enter_p, CS_p, or Exit_p, it easily follows that $H \circ \langle e_p \rangle$ also satisfies RF4.

- **RF5:** This condition follows trivially from (183) and (185). □

Corollary A1 *Consider a computation H, a set of processes RFS, and another computation L. Assume the following:*

- *$H \circ L \in C$;* (191)

- *RFS is a valid RF-set of H;* (192)
- *L is an* Act(H)*-computation;* (193)
- *L has no critical events in* $H \circ L$. (194)

Then, RFS is a valid RF-set of $H \circ L$.

Proof: The proof of Corollary A1 easily follows by induction on the length of L, applying Lemma A1 at each induction step. ☐

The following lemma shows that if two computations H and H' are "similar enough" with respect to a process p, and if a noncritical event e_p can be appended to H, then it can also be appended to H' without modification.

Lemma A2 *Consider two computations H and H', a set of processes RFS, and an event e_p of a process p. Assume the following:*

- $H \circ \langle e_p \rangle \in C$; (195)
- $H' \in C$; (196)
- *RFS is a valid RF-set of H;* (197)
- *RFS is a valid RF-set of* H'; (198)
- $p \in \mathrm{Act}(H)$; (199)
- $H \,|\, (\{p\} \cup \mathit{RFS}) = H' \,|\, (\{p\} \cup \mathit{RFS})$; (200)
- e_p *is noncritical in* $H \circ \langle e_p \rangle$. (201)

Then, the following hold: $H' \circ \langle e_p \rangle \in C$; *RFS is a valid RF-set of both* $H \circ \langle e_p \rangle$ *and* $H' \circ \langle e_p \rangle$; e_p *is a noncritical event in* $H' \circ \langle e_p \rangle$.

Proof: By (199) and (200), we have

$$p \in \mathrm{Act}(H'). \tag{202}$$

First, we prove that $H' \circ \langle e_p \rangle \in C$ holds. Because $H \,|\, p = H' \,|\, p$, by P2, it suffices to show that for each variable v in $\mathit{Rvar}(e_p)$, $\mathit{last\_writer\_event}(v, H) = \mathit{last\_writer\_event}(v, H')$.

- If v is local to p, then by (197), (198), and (199), for any event f_q in either H or H', by applying RF2 to f_q, $v \in \mathit{var}(f_q)$ implies $q \in \{p\} \cup \mathit{RFS}$. Thus, by (200), the last event to write to v is identical in both H and H'.

- If v is remote to p, then by (201) and by the definition of a critical event, there exists an event $\bar{e}_p$ by p in H such that $v \in \mathit{var}(\bar{e}_p)$. We consider two cases.

  First, assume that there exists a write to v in H. Define $q = \mathit{last\_writer}(v, H)$ and $f_q = \mathit{last\_writer\_event}(v, H)$. We claim that either $q = p$ or $q \in \mathit{RFS}$. If $q \in \mathrm{Fin}(H)$, then by (197), $q \in \mathit{RFS}$ follows. On the other hand, if $q \in \mathrm{Act}(H)$ and $q \neq p$ hold, then by (197) and (199), and applying RF3 to $\bar{e}_p$ and f_q in H, we have $q \in \mathit{RFS}$.

  Similarly, if there exists a write to v in H', then define $q' = \mathit{last\_writer}(v, H')$. Since $H \,|\, (\{p\} \cup \mathit{RFS}) = H' \,|\, (\{p\} \cup \mathit{RFS})$, H also contains a write to v if and only if H' contains a write to v, and the last event to write to v is identical in H and H'.

Thus, we have

$$H' \circ \langle e_p \rangle \in C. \tag{203}$$

By Lemma A1, *RFS* is a valid RF-set of $H \circ \langle e_p \rangle$, which establishes our second proof obligation. Assumptions (182)–(185) stated in Lemma A1 follow from (195), (197), (199), and (201), respectively.

We now claim that e_p is noncritical in $H' \circ \langle e_p \rangle$. Assume, to the contrary, that e_p is critical in $H' \circ \langle e_p \rangle$. Apply Lemma 2, with '$H$' $\leftarrow H'$, 'H'' $\leftarrow H$ (*i.e.*, with H and H' interchanged), 'G' $\leftarrow \langle\rangle$, and '$e'_p$' $\leftarrow e_p$. Among the assumptions stated in Lemma 2, (4)–(7), (9), (10) follow from (203), (195), (198), (197), (202), and (200), respectively; (8) is trivial; (11) and (12) follow from $G = \langle\rangle$; (13) follows from our assumption that e_p is critical in $H' \circ \langle e_p \rangle$. From the lemma, it follows that e_p is a critical event in $H \circ \langle e_p \rangle$, a contradiction. Therefore,

- e_p is noncritical in $H' \circ \langle e_p \rangle$. (204)

Finally, by applying Lemma A1 with 'H' $\leftarrow H'$, it follows that *RFS* is a valid RF-set of $H' \circ \langle e_p \rangle$. Assumptions (182)–(185) stated in Lemma A1 follow from (203), (198), (202), and (204), respectively. □

Corollary A2 *Consider two computations H and H', two sets of processes RFS and Z, and another computation L. Assume the following:*

- *$H \circ L \in C$;* (205)
- *$H' \in C$;* (206)
- *RFS is a valid RF-set of H;* (207)
- *RFS is a valid RF-set of H';* (208)
- *$Z \subseteq \mathrm{Act}(H)$;* (209)
- *$H \mid (Z \cup RFS) = H' \mid (Z \cup RFS)$;* (210)
- *L is a Z-computation;* (211)
- *L has no critical events in $H \circ L$.* (212)

Then, the following hold: $H' \circ L \in C$; RFS is a valid RF-set of both $H \circ L$ and $H' \circ L$; L has no critical events in $H' \circ L$.

Proof: The proof of Corollary A2 easily follows by induction on the length of L, applying Lemma A2 at each induction step. □

Lemma 4 *Consider a computation H, a set of processes RFS, and another set of processes $Y = \{p_1, p_2, \ldots, p_m\}$. Assume the following:*

- *$H \in C$;* (22)
- *RFS is a valid RF-set of H;* (23)
- *$Y \subseteq \mathrm{Inv}_{RFS}(H)$;* (24)
- *for each p_j in Y, there exists a computation L_{p_j}, satisfying the following:*
  - *L_{p_j} is a p_j-computation;* (25)
  - *$H \circ L_{p_j} \in C$;* (26)
  - *L_{p_j} has no critical events in $H \circ L_{p_j}$.* (27)

Define L to be $L_{p_1} \circ L_{p_2} \circ \cdots \circ L_{p_m}$. Then, the following hold: $H \circ L \in C$, RFS is a valid RF-set of $H \circ L$, and L contains no critical events in $H \circ L$.

Proof: First, note that (24) implies

$$Y \subseteq \mathrm{Act}(H). \tag{213}$$

Define $L^0 = \langle\rangle$; for each positive j, define L^j to be $L_{p_1} \circ L_{p_2} \circ \cdots \circ L_{p_j}$. We prove the lemma by induction on j. At each step, we assume

- $H \circ L^j \in C$, (214)

- RFS is a valid RF-set of $H \circ L^j$, and (215)
- L^j contains no critical events in $H \circ L^j$. (216)

The induction base ($j = 0$) follows easily from (22) and (23), since $L^0 = \langle\rangle$.

Assume that (214)–(216) hold for a particular value of j. By definition, $L^j \,|\, p_{j+1} = \langle\rangle$, and hence

$$H \,|\, (\{p_{j+1}\} \cup RFS) = (H \circ L^j) \,|\, (\{p_{j+1}\} \cup RFS). \tag{217}$$

We use Corollary A2, with 'H'' $\leftarrow H \circ L^j$, 'Z' $\leftarrow \{p_{j+1}\}$, and 'L' $\leftarrow L_{p_{j+1}}$. Among the assumptions stated in Corollary A2, (206)–(208) and (210) follow from (214), (23), (215), and (217), respectively; (209) follows from (213) and $p_{j+1} \in Y$; (205), (211), and (212) follow from (26), (25), and (27), respectively, each applied with 'p_j' $\leftarrow p_{j+1}$. This gives us the following:

- $H \circ L^{j+1} = (H \circ L^j) \circ L_{p_{j+1}} \in C$;
- RFS is a valid RF-set of $H \circ L^{j+1}$;
- $L_{p_{j+1}}$ contains no critical events in $H \circ L^{j+1}$. (218)

By (216) and (218), it follows that L^{j+1} contains no critical events in $H \circ L^{j+1}$. $\square$

Lemma 5 *Let H be a computation. Assume the following:*

- *$H \in C$, and* (28)
- *H is regular (i.e.,* $\mathrm{Fin}(H)$ *is a valid RF-set of H).* (29)

Define $n = |\,\mathrm{Act}(H)|$. Then, there exists a subset Y of $\mathrm{Act}(H)$, where $n - 1 \le |Y| \le n$, satisfying the following: for each process p in Y, there exist a p-computation L_p and an event e_p by p such that

- *$H \circ L_p \circ \langle e_p\rangle \in C$;* (30)
- *L_p contains no critical events in $H \circ L_p$;* (31)
- *$e_p \notin \{Enter_p, CS_p, Exit_p\}$;* (32)
- $\mathrm{Fin}(H)$ *is a valid RF-set of $H \circ L_p$;* (33)
- *e_p is a critical event by p in $H \circ L_p \circ \langle e_p\rangle$.* (34)

Proof: First, we construct, for each process p in $\mathrm{Act}(H)$, a computation L_p and an event e_p that satisfy (30) and (31). Then, we show that every event e_p thus constructed, except at most one, satisfies (32). The other conditions can be easily proved thereafter.

For each process p in $\mathrm{Act}(H)$, define H_p as

$$H_p = H \,|\, (\{p\} \cup \mathrm{Fin}(H)). \tag{219}$$

We apply Lemma 1, with 'RFS' $\leftarrow \mathrm{Fin}(H)$, and '$Y$' $\leftarrow \{p\} \cup \mathrm{Fin}(H)$. Among the assumptions stated in Lemma 1, (1) and (2) follow from (28) and (29), respectively; (3) is trivial. It follows that H_p is in C and

- $\mathrm{Fin}(H)$ is a valid RF-set of H_p. (220)

Since $p \in \mathrm{Act}(H)$, by (219), we have

$$\mathrm{Act}(H_p) = \{p\} \quad \wedge \quad \mathrm{Fin}(H_p) = \mathrm{Fin}(H). \tag{221}$$

Thus, by (220), and applying RF4 to H, we have

$$value(stat_q, H_p) = \begin{cases} ncs, & \text{for all } q \ne p \\ entry, & \text{for } q = p. \end{cases}$$

Therefore, by the Progress property, there exists a p-computation F_p such that $H_p \circ F_p \circ \langle CS_p \rangle \in C$. If F_p has a critical event in $H_p \circ F_p \circ \langle CS_p \rangle$, then let e'_p be the first critical event in F_p, and let L_p be the prefix of F_p that precedes e'_p (*i.e.*, $F_p = L_p \circ \langle e'_p \rangle \circ \cdots$). Otherwise, define L_p to be F_p and e'_p to be CS_p. By P1, we have $H_p \circ L_p \circ \langle e'_p \rangle \in C$ and $H_p \circ L_p \in C$.

We have just constructed a computation L_p and an event e'_p by p, such that

- $H_p \circ L_p \circ \langle e'_p \rangle \in C$, (222)
- $H_p \circ L_p \in C$, (223)
- L_p is a p-computation, (224)
- L_p has no critical events in $H_p \circ L_p$, and (225)
- e'_p is a critical event in $H_p \circ L_p \circ \langle e'_p \rangle$. (226)

The following assertion follows easily from (219).

$$(H_p \circ L_p) \,|\, (\{p\} \cup \mathrm{Fin}(H)) = (H \circ L_p) \,|\, (\{p\} \cup \mathrm{Fin}(H)). \tag{227}$$

We now use Corollary A2, with 'H' $\leftarrow H_p$, 'H'' $\leftarrow H$, 'RFS' $\leftarrow \mathrm{Fin}(H)$, '$Z$' $\leftarrow \{p\}$, and 'L' $\leftarrow L_p$. Assumptions (205)–(212) stated in Corollary A2 follow from (223), (28), (220), (29), (221), (227), (224), and (225), respectively. Thus, we have the following:

- $H \circ L_p \in C$; (228)
- $\mathrm{Fin}(H)$ is a valid RF-set of $H_p \circ L_p$; (229)
- $\mathrm{Fin}(H)$ is a valid RF-set of $H \circ L_p$; (230)
- L_p has no critical events in $H \circ L_p$, which establishes (31).

Because $H \circ L_p$ and $H_p \circ L_p$ are equivalent with respect to p, by (222), (228), and P3, there exists an event e_p of p such that

- $e_p \sim e'_p$, and (231)
- $H \circ L_p \circ \langle e_p \rangle \in C$, which establishes (30).

We now claim that at most one process in $\mathrm{Act}(H)$ fails to satisfy (32). Because $p \in \mathrm{Act}(H)$ and H is regular, by RF4, $value(stat_p, H) = entry$ holds. Thus, by the definition of a mutual exclusion system, e_p cannot be $Enter_p$ or $Exit_p$. It suffices to show that there can be at most one process in $\mathrm{Act}(H)$ such that $e_p = CS_p$.

Assume, to the contrary, that there are two distinct processes p and q in $\mathrm{Act}(H)$, such that $e_p = CS_p$ and $e_q = CS_q$. Note that (29) implies that $\mathrm{Inv}_{\mathrm{Fin}(H)}(H) = \mathrm{Act}(H)$, and hence

$$\{p, q\} \subseteq \mathrm{Inv}_{\mathrm{Fin}(H)}(H). \tag{232}$$

By applying Lemma 4 with 'RFS' $\leftarrow \mathrm{Fin}(H)$, '$Y$' $\leftarrow \{p, q\}$, 'L_{p_1}' $\leftarrow L_p$, and 'L_{p_2}' $\leftarrow L_q$, we have $H \circ L_p \circ L_q \in C$. Among the assumptions stated in Lemma 4, (22)–(24) follow from (28), (29), and (232), respectively; (25)–(27) follow from (224), (228), (31), respectively, each with 'p' $\leftarrow p$ and then 'p' $\leftarrow q$.

Since $H \circ L_p$ is equivalent to $H \circ L_p \circ L_q$ with respect to p, and since CS_p does not read any variable, by P2, we have $H \circ L_p \circ L_q \circ \langle CS_p \rangle \in C$. Similarly, we also have $H \circ L_p \circ L_q \circ \langle CS_q \rangle \in C$. Hence $H \circ L_p \circ L_q$ violates the Exclusion property, a contradiction.

Therefore, there exists a subset Y of $\mathrm{Act}(H)$ such that $n - 1 \le |Y| \le n$ and each process in Y satisfies (30), (31), and (32).

We claim that each process p in Y satisfies (33). Note that, since $p \in Y$ and $Y \subseteq \mathrm{Act}(H)$, by (224),

- L_p is a Y-computation. (233)

Condition (33) now follows from Corollary A1, with 'RFS' $\leftarrow \mathrm{Fin}(H)$, and '$L$' $\leftarrow L_p$. Assumptions (191)–(194) stated in the corollary follow from (228), (29), (233), and (31), respectively.

Finally, we prove (34). Note that, by (221), (224), and (225), we have

$$\mathrm{Act}(H_p \circ L_p) = \{p\}. \tag{234}$$

Condition (34) now follows from Lemma 2, with 'H' $\leftarrow H_p \circ L_p$, 'H'' $\leftarrow H \circ L_p$, 'G' $\leftarrow \langle\rangle$, '$RFS$' $\leftarrow \mathrm{Fin}(H)$, '$e_p$' $\leftarrow e'_p$, and 'e'_p' $\leftarrow e_p$. Assumptions (4)–(13) stated in Lemma 2 follow from (222), (30), (229), (230), (231), (234), (227), $G = \langle\rangle$, $G = \langle\rangle$, and (226), respectively. □

Lemma 6 *Consider a computation H and set of processes RFS. Assume the following:*

- *$H \in C$;* (35)
- *RFS is a valid RF-set of H;* (36)
- $\mathrm{Fin}(H) \subsetneq RFS$ *(i.e.,* $\mathrm{Fin}(H)$ *is a proper subset of RFS).* (37)

Then, there exists a computation G satisfying the following.

- *$G \in C$;* (38)
- *RFS is a valid RF-set of G;* (39)
- *G can be written as $H \,|\, (Y \cup RFS) \circ L \circ \langle e_p \rangle$, for some choice of Y, L, and e_p, satisfying the following:*
  - *Y is a subset of* $\mathrm{Inv}(H)$ *such that* $|\mathrm{Inv}(H)| - 1 \le |Y| \le |\mathrm{Inv}(H)|$, (40)
  - $\mathrm{Inv}(G) = Y$, (41)
  - *L is a* $\mathrm{Pmt}(H)$*-computation,* (42)
  - *L has no critical events in G,* (43)
  - $p \in \mathrm{Pmt}(H)$, *and* (44)
  - *e_p is critical in G;* (45)
- $\mathrm{Pmt}(G) \subseteq \mathrm{Pmt}(H)$; (46)
- *An event in $H \,|\, (Y \cup RFS)$ is critical if and only if it is also critical in H.* (47)

Proof: Define $Z = \mathrm{Pmt}(H)$. Then, by definition, $Z \subseteq \mathrm{Act}(H)$. Define H' as

$$H' = H \,|\, RFS. \tag{235}$$

Apply Lemma 1, with 'Y' $\leftarrow RFS$. Assumptions (1)–(3) stated in Lemma 1 follow from (35)–(37), respectively.

- RFS is a valid RF-set of H'. (236)

Also, by (235), we have

$$\mathrm{Act}(H') = Z \quad \wedge \quad \mathrm{Fin}(H') = \mathrm{Fin}(H). \tag{237}$$

Therefore, by (236), and applying RF4 to H', we have

$$value(stat_q, H') = \begin{cases} entry \text{ or } exit, & \text{if } q \in Z, \\ ncs, & \text{if } q \notin Z. \end{cases}$$

Therefore, by the Progress property, there exists a Z-computation F such that $H' \circ F \circ \langle f_r \rangle \in C$, where r is a process in Z and f_r is either CS_r or $Exit_r$. If F has a critical event in $H' \circ F \circ \langle f_r \rangle$, , then let e'_p be the first critical event in F, and let L be the prefix of F that precedes e'_p (*i.e.*, $F = L \circ \langle e'_p \rangle \circ \cdots$). Otherwise, define L to be F and e'_p to be f_r. By P1, we have $H' \circ L \circ \langle e'_p \rangle \in C$ and $H' \circ L \in C$. Because F is a Z-computation, we have $p \in Z$, which implies

- $p \in Z \subseteq \mathrm{Act}(H)$, and (238)
- $p \in RFS$. (239)

We have just constructed a computation L and an event e'_p by p, such that

- $H' \circ L \circ \langle e'_p \rangle \in C$, (240)
- $H' \circ L \in C$, (241)
- L is a Z-computation, (242)
- L has no critical events in $H' \circ L$, and (243)
- e'_p is a critical event in $H' \circ L \circ \langle e'_p \rangle$. (244)

The following assertion follows easily from (235) and (242). (Note that $Z \subseteq RFS$ holds by definition.)

$$(H' \circ L) \,|\, RFS = (H \circ L) \,|\, RFS. \tag{245}$$

We now use Corollary A2, with 'H' $\leftarrow H'$ and 'H'' $\leftarrow H$. Among the assumptions stated in Corollary A2, (205)–(209), (211), and (212) follow from (241), (35), (236), (36), (237), (242), and (243), respectively; (210) follows from (245) and $Z \subseteq RFS$. Thus, we have the following:

- $H \circ L \in C$; (246)
- RFS is a valid RF-set of $H' \circ L$; (247)
- RFS is a valid RF-set of $H \circ L$; (248)
- L has no critical events in $H \circ L$. (249)

Note that, by (242), (249), and $Z \subseteq \mathrm{Act}(H)$, we have

$$\mathrm{Act}(H \circ L) = \mathrm{Act}(H) \quad \wedge \quad \mathrm{Fin}(H \circ L) = \mathrm{Fin}(H). \tag{250}$$

In particular, by (238),

$$p \in \mathrm{Act}(H \circ L). \tag{251}$$

Because $H \circ L$ and $H' \circ L$ are equivalent with respect to p, by (240), (246), and P3, there exists an event e''_p of p such that

- $e''_p \sim e'_p$, and (252)
- $H \circ L \circ \langle e''_p \rangle \in C$. (253)

We now use Lemma 2, with 'H' $\leftarrow H' \circ L$, 'H'' $\leftarrow H \circ L$, 'G' $\leftarrow \langle\rangle$, '$e_p$' $\leftarrow e'_p$, and 'e'_p' $\leftarrow e''_p$. Among the assumptions stated in Lemma 2, (4)–(8) and (11)–(13) follow from (240), (253), (247), (248), (252), 'G' $\leftarrow \langle\rangle$, '$G$' $\leftarrow \langle\rangle$, and (244), respectively; (9) follows from (238) and (243); (10) follows from (239) and (245). It follows that

- e''_p is critical in $H \circ L \circ \langle e''_p \rangle$. (254)

We now establish (38)–(47) by considering two cases separately.

Case 1: e''_p is a transition event. In this case, define $Y = \mathrm{Inv}(H)$, $e_p = e''_p$, and $G = H \circ L \circ \langle e_p \rangle$. Note that, by (36), we have

$$H = H \,|\, (Y \cup RFS) \quad \wedge \quad G = H \,|\, (Y \cup RFS) \circ L \circ \langle e_p \rangle. \tag{255}$$

We claim that these definitions satisfy (38)–(47). Conditions (38) and (42)–(44) follow from (253), (242), (249), and (238), respectively. Condition (40) is trivial.

We now establish (39). Before proving RF1–RF5, we need to prove that $\mathrm{Fin}(G) \subseteq RFS$ and that $G \,|\, q \neq \langle\rangle$ holds for each $q \in RFS$. Condition (248) implies $\mathrm{Fin}(H \circ L) \subseteq RFS$. By the definition of G, we have $\mathrm{Fin}(G) \subseteq \mathrm{Fin}(H \circ L) \cup \{p\}$. Thus, by (239), we have $\mathrm{Fin}(G) \subseteq RFS$. Condition (248) also implies that $G \,|\, q \neq \langle\rangle$ holds for each $q \in RFS$. We now check each of RF1–RF5.

- **RF1, RF2, and RF3:** Since appending a transition event does not invalidate any of RF1, RF2, and RF3, it easily follows, by (248), that G satisfies these conditions.
- **RF4:** By (248), it suffices to show that p satisfies RF4, which follows easily from (239) and (251). In particular, if $e_p = CS_p$, then $p \in \mathrm{Pmt}(G)$ and $value(stat_p, G) = exit$ hold; if $e_p = Exit_p$, then $p \in \mathrm{Fin}(G)$ and $value(stat_p, G) = ncs$ hold. (Note that e_p cannot be $Enter_p$, because by (248) and (251), and applying RF4 to p in $H \circ L$, we have $value(stat_p, H \circ L) \neq ncs$.)
- **RF5:** It suffices to show that e_p is also a critical event in $G \,|\, RFS$. However, since e_p is a transition event, this is immediate.

It follows that (39) holds.

We now conclude Case 1 by establishing (41) and (45)–(47). By (250), we have $\mathrm{Inv}(H \circ L) = (\mathrm{Act}(H \circ L) - RFS) = (\mathrm{Act}(H) - RFS) = \mathrm{Inv}(H)$. Moreover, (239) implies that appending e_p to $H \circ L$ cannot change the set of invisible processes. Thus, we have (41). Condition (45) holds by definition, since e_p is a transition event. In order to prove (46), note that any process in $\mathrm{Fin}(H)$ is also in $\mathrm{Fin}(G)$ by the definition of a finished process. Thus, we have $\mathrm{Pmt}(G) = (RFS - \mathrm{Fin}(G)) \subseteq (RFS - \mathrm{Fin}(H)) = Z$. Finally, (255) implies that (47) is trivially true.

Case 2: e''_p is *not* a transition event. In this case, there exists a variable v_{ce} (for "critical event"), remote to p, that is accessed by e''_p. If v_{ce} is local to a process in $\mathrm{Inv}(H)$, let x_{loc} be the process that v_{ce} is local to; otherwise, let $x_{\mathrm{loc}} = \perp$. Similarly, if $last\_writer(v_{\mathrm{ce}}, H \circ L) \in \mathrm{Inv}(H)$ holds, let $x_w = last\_writer(v_{\mathrm{ce}}, H \circ L)$ and $f_{x_w} = last\_writer\_event(v_{\mathrm{ce}}, H \circ L)$; otherwise, let $x_w = \perp$ and $f_{x_w} = \perp$. By definition,

- if $x_{\mathrm{loc}} \neq \perp$, then $x_{\mathrm{loc}} \in \mathrm{Inv}(H) \subseteq \mathrm{Act}(H)$. (256)
- if $x_w \neq \perp$, then $x_w \in \mathrm{Inv}(H) \subseteq \mathrm{Act}(H)$. (257)

We now establish the following simple claim.

Claim 1: If $x_{\mathrm{loc}} \neq \perp$ and $x_w \neq \perp$, then $x_{\mathrm{loc}} = x_w$.

Proof of Claim: Assume, to the contrary, that $x_{\mathrm{loc}} \neq \perp$, $x_w \neq \perp$, and $x_{\mathrm{loc}} \neq x_w$ hold. Then, f_{x_w} *remotely* writes to v_{ce}. Hence, by (248), and applying RF2 to f_{x_w} in $H \circ L$, we have either $x_{\mathrm{loc}} \notin \mathrm{Act}(H \circ L)$ or $\{x_w, x_{\mathrm{loc}}\} \in RFS$. However, by (250) and (256), we have $x_{\mathrm{loc}} \in \mathrm{Act}(H \circ L)$ and $x_{\mathrm{loc}} \notin RFS$, a contradiction. (Note that, by (36), $x_{\mathrm{loc}} \in \mathrm{Inv}(H)$ implies $x_{\mathrm{loc}} \notin RFS$.) □

We now define Y as follows. By Claim 1, Y is well-defined.

$$Y = \begin{cases} \mathrm{Inv}(H) - \{x_{\mathrm{loc}}\}, & \text{if } x_{\mathrm{loc}} \neq \perp; \\ \mathrm{Inv}(H) - \{x_w\}, & \text{if } x_w \neq \perp; \\ \mathrm{Inv}(H), & \text{if } x_{\mathrm{loc}} = \perp \text{ and } x_w = \perp. \end{cases} \tag{258}$$

Let $\overline{G} = H \,|\, (Y \cup RFS) \circ L$. (Informally, $\overline{G}$ is a computation that is obtained by erasing x_{loc} and x_w from H. By erasing x_{loc}, we preserve RF2. By erasing x_w, we preserve RF3 and eliminate potential information flow. Since L has no critical events in $\overline{G}$ [see (269) below], appending L does not create information flow.) We now establish a number of assertions concerning $\overline{G}$, after which we define G. By (242) and $Z \subseteq RFS$, we have

$$\overline{G} = H \,|\, (Y \cup RFS) \circ L = (H \circ L) \,|\, (Y \cup RFS). \tag{259}$$

We now apply Lemma 1, with 'H' $\leftarrow H \circ L$ and 'Y' $\leftarrow Y \cup RFS$. Among the assumptions stated in Lemma 1, (1) and (2) follow from (246) and (248), respectively; (3) is trivial. Thus, we have the following:

- $\overline{G}$ is in C, (260)
- RFS is a valid RF-set of $\overline{G}$, and (261)
- an event in $\overline{G}$ is critical if and only if it is also critical in $H \circ L$. (262)

By (239), we have $RFS = \{p\} \cup RFS$. Thus, by (251) and (259), we have the following:

- $p \in \mathrm{Act}(\overline{G})$, and (263)
- $\overline{G} \,|\, (\{p\} \cup RFS) = (H \circ L) \,|\, (\{p\} \cup RFS)$. (264)

If $x_{\mathrm{loc}} \neq \perp$, then we have $x_{\mathrm{loc}} \notin Y$ by (258), and also $x_{\mathrm{loc}} \notin RFS$, since $x_{\mathrm{loc}} \in \mathrm{Inv}(H)$. Thus, by (259), it follows that

- $\overline{G} \,|\, x_{\mathrm{loc}} = \langle\rangle$, if $x_{\mathrm{loc}} \neq \perp$. (265)

Similarly,

- $\overline{G} \,|\, x_w = \langle\rangle$, if $x_w \neq \perp$. (266)

By (264), $\overline{G}$ is equivalent to $H \circ L$ with respect to p. Therefore, by (253), (260), and P3, there exists an event e_p such that

- $e_p \sim e_p''$, and (267)
- $\overline{G} \circ \langle e_p\rangle \in C$. (268)

Define G to be $\overline{G} \circ \langle e_p\rangle = H \,|\, (Y \cup RFS) \circ L \circ \langle e_p\rangle$. We claim that G satisfies the lemma. To show this, we need a few additional assertions. By (249), (259), and (262), it follows that

- L has no critical events in $\overline{G}$, and (269)
- an event in $H \,|\, (Y \cup RFS)$ is critical if and only if it is also critical in H. (270)

Also, by (263), and since e_p is not a transition event,

$$\mathrm{Act}(G) = \mathrm{Act}(\overline{G}). \tag{271}$$

We now prove that G satisfies the lemma. Each of the conditions (38), (40), (42), (44), and (47) follows easily from (268), (258), (242), (238), and (270), respectively. Since $\overline{G}$ is a prefix of G, (43) follows from (269). By (258), (259), and (269), the active processes in $\overline{G}$ include those in Y, which are invisible, and any promoted processes in RFS; hence, $\mathrm{Act}(\overline{G}) - RFS = Y - RFS = Y$. Moreover, the processes in RFS that are active in $H \circ L$ are also active in $\overline{G}$; hence, $\mathrm{Act}(\overline{G}) \cap RFS = \mathrm{Act}(H \circ L) \cap RFS$. Thus, by (250) and (271), we have

$$\begin{aligned} \mathrm{Inv}(G) &= \mathrm{Act}(G) - RFS = \mathrm{Act}(\overline{G}) - RFS = Y - RFS = Y, \quad \text{and} \\ \mathrm{Pmt}(G) &= \mathrm{Act}(G) \cap RFS = \mathrm{Act}(\overline{G}) \cap RFS = \mathrm{Act}(H \circ L) \cap RFS = \mathrm{Act}(H) \cap RFS = Z, \end{aligned}$$

which imply (41) and (46).

In order to prove (45), we apply Lemma 2 with 'H' $\leftarrow H \circ L$, 'H'' $\leftarrow \overline{G}$, 'G' $\leftarrow \langle\rangle$, '$e_p$' $\leftarrow e_p''$, and 'e_p'' $\leftarrow e_p$. Assumptions (4)–(13) stated in Lemma 2 follow from (253), (268), (248), (261), (267), (251), (264), 'G' $\leftarrow \langle\rangle$, '$G$' $\leftarrow \langle\rangle$, and (254), respectively. Moreover, Assumption (A) follows from (261), and (B) holds vacuously by 'G' $\leftarrow \langle\rangle$. It follows that e_p is critical in G, *i.e.*, (45) holds, and

- G satisfies RF5. (272)

This leaves only (39) to be proved. Let $x' = \mathit{last\_writer}(v_{\mathrm{ce}}, \overline{G})$ and $f_{x'} = \mathit{last\_writer\_event}(v_{\mathrm{ce}}, \overline{G})$. We begin by establishing the following claim.

Claim 2: $x' = \perp$ or $x' \in \mathit{RFS}$.

Proof of Claim: Assume, to the contrary, that $x' \neq \perp$ and $x' \notin \mathit{RFS}$ hold. If $f_{x'}$ is an event of L, then by (242) and the definition of Z, we have $x' \in Z \subseteq \mathit{RFS}$, a contradiction. Thus, $f_{x'}$ is an event of $H \,|\, (Y \cup \mathit{RFS})$. Since we assumed $x' \notin \mathit{RFS}$, by (258), we have $x' \in Y \subseteq \mathrm{Inv}(H)$. If $x' = \mathit{last\_writer}(v_{\mathrm{ce}}, H \circ L)$, then since $x' \in \mathrm{Inv}(H)$, we have $x' = x_w$ by the definition of x_w, which is impossible by (266). Thus, we have $x' \neq \mathit{last\_writer}(v_{\mathrm{ce}}, H \circ L)$, which implies

$$x_w \neq x'. \tag{273}$$

Note that, by (257) and (250), we have

$$x_w \neq \perp \;\Rightarrow\; x_w \in \mathrm{Act}(H \circ L). \tag{274}$$

Also, since $x' \in \mathrm{Inv}(H) \subseteq \mathrm{Act}(H)$, by (250),

$$x' \in \mathrm{Act}(H \circ L). \tag{275}$$

We now show that $\mathit{last\_writer}(v_{\mathrm{ce}}, H \circ L) \in \mathit{RFS}$ holds. If $x_w = \perp$, then since there exists a write to v (*i.e.*, $f_{x'}$) in $H \circ L$, the definition of x_w implies $\mathit{last\_writer}(v_{\mathrm{ce}}, H \circ L) \notin \mathrm{Inv}(H)$, which in turn implies $\mathit{last\_writer}(v_{\mathrm{ce}}, H \circ L) \in \mathit{RFS}$ by (248) and (250). On the other hand, if $x_w \neq \perp$, then by (248), (273), (274), (275), and by applying RF3 to $f_{x'}$ and f_{x_w} in $H \circ L$, we have $\mathit{last\_writer}(v_{\mathrm{ce}}, H \circ L) \in \mathit{RFS}$.

Because $\mathit{last\_writer}(v_{\mathrm{ce}}, H \circ L) \in \mathit{RFS}$ holds, by (259), the last event to write to v_{ce} is identical in $H \circ L$ and $\overline{G}$. Thus, we have $x' \in \mathit{RFS}$, a contradiction. □

We now establish (39) by showing that RFS is a valid RF-set of G. Condition RF5 was already proved in (272). Before proving RF1–RF4, we need to prove that $\mathrm{Fin}(G) \subseteq \mathit{RFS}$ and that $G \,|\, q \neq \langle\rangle$ holds for each $q \in \mathit{RFS}$. Condition (261) implies $\mathrm{Fin}(\overline{G}) \subseteq \mathit{RFS}$. Since e_p is not a transition event, we have $\mathrm{Fin}(G) = \mathrm{Fin}(\overline{G})$, and hence $\mathrm{Fin}(G) \subseteq \mathit{RFS}$. Condition (261) also implies that $G \,|\, q \neq \langle\rangle$ holds for each $q \in \mathit{RFS}$. We now check each of RF1–RF4.

- **RF1 and RF2:** We use Lemma 3 to prove these two conditions. First, we need the following claim.

  Claim 3: If v_{ce} is local to a process q, then either $q \notin \mathrm{Act}(\overline{G})$ or $\{p, q\} \subseteq \mathit{RFS}$ holds.

  Proof of Claim: If v_{ce} is local to a process q, then one of the following holds: $q \in \mathrm{Inv}(H)$, $q \in \mathit{RFS}$, or $H \,|\, q = \langle\rangle$. If $q \in \mathrm{Inv}(H)$, then $q = x_{\mathrm{loc}}$ by definition, and hence, by (265), we have $q \notin \mathrm{Act}(\overline{G})$. If $q \in \mathit{RFS}$, then by (239), we have $\{p, q\} \subseteq \mathit{RFS}$. By (250) and (259), we have $\mathrm{Act}(\overline{G}) \subseteq \mathrm{Act}(H \circ L) = \mathrm{Act}(H)$, and hence $H \,|\, q = \langle\rangle$ implies $q \notin \mathrm{Act}(\overline{G})$. □

  We now use Lemma 3, with 'H' $\leftarrow \overline{G}$, 'G' $\leftarrow \langle\rangle$, and '$v_{\mathrm{rem}}$' $\leftarrow v_{\mathrm{ce}}$. Among the assumptions stated in Lemma 3, assumptions (14)–(17) follow from (268), (261), (263), and (261), respectively; (18) and (19) are trivial; (20) follows from Claim 3; (21) (with 'q' $\leftarrow x'$) follows from Claim 2. It follows that G satisfies RF1 and RF2.

- **RF3:** Consider a variable $v \in V$ and two different events g_q and h_r in G. Assume that both q and r are in $\mathrm{Act}(G)$, $q \neq r$, that there exists a variable v such that $v \in \mathit{var}(g_q) \cap \mathit{var}(h_r)$, and that there exists a write to v in G. Define $s = \mathit{last\_writer}(v, G)$. Our proof obligation is to show that $s \in \mathit{RFS}$.

  Since $p \in Z \subseteq \mathit{RFS}$, it suffices to consider the case in which $s \neq p$, in which case we also have the following: e_p does not write v, $s = \mathit{last\_writer}(v, \overline{G})$, and there exists a write to v in $\overline{G}$.

- First, consider the case in which both g_q and h_r are in $\overline{G}$. By (271), we have $q \in \mathrm{Act}(\overline{G})$ and $r \in \mathrm{Act}(\overline{G})$. Thus, by (261), and by applying RF3 to g_q and h_r in $\overline{G}$, we have $s \in RFS$.
- Second, consider the case in which one of g_q or h_r is e_p. Without loss of generality, we can assume that g_q is in $\overline{G}$ and $h_r = e_p$. Then, we have $q \neq p$ and $r = p$. If v is local to p, then by (261), (263), $s \neq p$, and by applying RF2 to $last\_writer\_event(v, \overline{G})$ by s in $\overline{G}$, we have $s \in RFS$. On the other hand, if v is remote to p, then by the Atomicity Property, we have $v = v_{\mathrm{ce}}$ and $s = last\_writer(v_{\mathrm{ce}}, \overline{G}) = x'$, in which case, by Claim 2, we have $s = \perp$ or $s \in RFS$. Since there exists a write to v_{ce} in $\overline{G}$ by assumption, we have $s \neq \perp$, and hence $s \in RFS$.

- **RF4:** Since $\overline{G}$ satisfies RF4 by (261), and since e_p is not a transition event, RF4 follows trivially. $\square$