A Time Complexity Lower Bound for Adaptive Mutual Exclusion*

Yong-Jik Kim
Tmax Soft Research Center
272-6 Seohyeon-dong, Seongnam-si
Gyeonggi-do, Korea 463-824
Email: jick@tmax.co.kr

James H. Anderson
Department of Computer Science
University of North Carolina at Chapel Hill
Chapel Hill, NC 27599-3175
Email: anderson@cs.unc.edu

September 2005

Abstract

We consider the time complexity of adaptive mutual exclusion algorithms, where "time" is measured by counting the number of remote memory references required per critical-section access. We establish a lower bound that precludes a deterministic algorithm with $O(\log k)$ time complexity (in fact, any deterministic $o(k)$ algorithm), where k is "point contention."

*Work supported by NSF grants CCR 9732916, CCR 9972211, CCR 9988327, ITR 0082866, and CCR 0208289. This work was presented in preliminary form at the 15th International Symposium on Distributed Computing [13], where it received the best student paper award.

1 Introduction

In this paper, we consider the time complexity of adaptive mutual exclusion algorithms. A mutual exclusion algorithm is *adaptive* if its time complexity is a function of the number of contending processes [4, 8, 11, 14, 15]. Two notions of contention have been considered in the literature: "interval contention" and "point contention" [1]. The *interval contention* over a computation H is the number of processes that are active in H, *i.e.*, that execute outside of their noncritical sections. The *point contention* over H is the maximum number of processes that are active at the *same state* in H. Note that point contention is always at most interval contention. Throughout this paper, we let N denote the number of processes in the system. Also, unless stated otherwise, k denotes the point contention experienced by an arbitrary process while it is active.

The time complexity measure considered in this paper is motivated by work on local-spin synchronization algorithms. In local-spin algorithms, all busy waiting is by means of read-only loops in which one or more locally-accessible "spin variables" are repeatedly tested. The ability to locally access a shared variable is provided on both distributed shared-memory (DSM) and cache-coherent (CC) machines, as illustrated in Figure 1. In a DSM machine, each processor has its own memory module that can be accessed without accessing the global interconnection network. On such a machine, a shared variable can be made locally accessible by storing it in a local memory module. In a CC machine, each processor has a private cache, and some hardware protocol is used to enforce cache consistency (*i.e.*, to ensure that all copies of the same variable in different local caches are consistent). On such a machine, a shared variable becomes locally accessible by migrating to a local cache line.

Because our main interest is local-spin algorithms, we determine the time complexity of a mutual exclusion algorithm by counting the number of remote memory references generated by one process to enter and then exit its critical section. A *remote memory reference* (RMR) is a memory access that requires a traversal of the global processors-to-memory interconnect. This complexity measure is known as the *RMR time complexity measure* [6].

In prior work, we presented an adaptive mutual exclusion algorithm with $O(min(k, \log N))$ RMR time complexity that is based only on reads and writes [4]. (A similar algorithm has also been presented by Afek *et al.* [3].) In other prior work, we established a worst-case RMR time bound of $\Omega(\log N/\log\log N)$ for mutual exclusion algorithms (adaptive or not) based on reads, writes, or comparison primitives[1] such as test-and-set and compare-and-swap [5]. This result shows that the $\Theta(\log N)$ worst-case RMR time complexity of our $O(min(k, \log N))$ algorithm is close to optimal (specifically, within a factor of $\Theta(\log\log N)$). In fact, we believe it *is* optimal: we conjecture that $\Omega(\log N)$ is a tight lower bound for this class of algorithms.

The $\Omega(\log N/\log\log N)$ lower bound mentioned above does not mention k, so it tells us very little about RMR time complexity under low contention. The best we can say is that $\Omega(\log k/\log\log k)$ remote references are required. In particular, the $\Omega(\log N/\log\log N)$ lower bound is established by inductively considering longer and longer computations, the first of which involves N processes, and the last of which may involve fewer processes. If we start instead with k process, then a computation is obtained with $O(k)$ processes (and hence $O(k)$ point contention at each state) in which some process performs $\Omega(\log k/\log\log k)$ remote references.

If $\Omega(\log N)$ is a tight lower bound, as conjectured above, then presumably a lower bound of $\Omega(\log k)$ would follow as well. This suggests two interesting possibilities: in all likelihood, either $\Omega(\min(k, \log N))$ is in fact a tight lower bound (*i.e.*, the algorithm in [4] is optimal), or it is possible to design an adaptive algorithm with $O(\log k)$ RMR time complexity (*i.e.*, $\Omega(\log k)$ is tight). Indeed, the problem of designing an $O(\log k)$ algorithm using only reads and writes has been mentioned in at least two papers [4, 8].

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

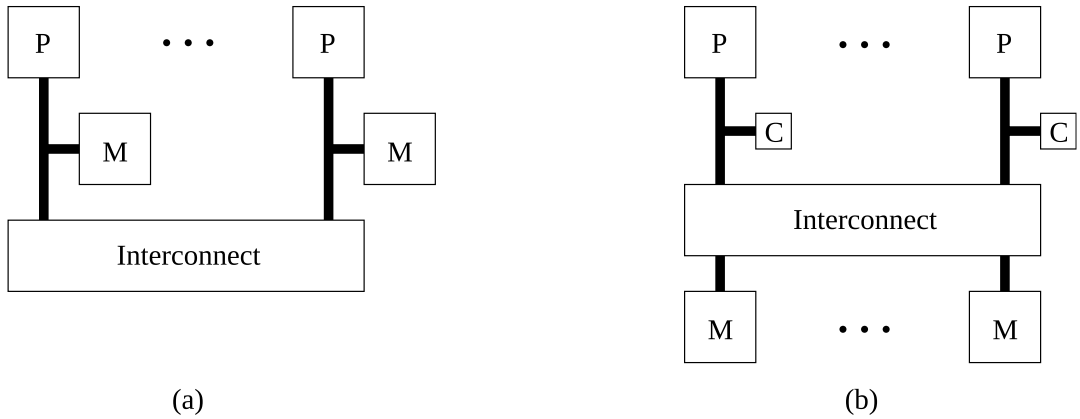

Figure 1: **(a)** DSM model. **(b)** CC model. In both insets, 'P' denotes a processor, 'C' a cache, and 'M' a memory module.

In this paper, we show that an $O(\log k)$ algorithm in fact does not exist. In particular, we prove the following.

> *Given any k, define $\bar{N} = \bar{N}(k) = (2k+4)^{2(2^k-1)}$. For any $N \geq \bar{N}$, and for any N-process mutual exclusion algorithm based on reads, writes, or comparison primitives, a computation exists involving $\Theta(k)$ processes in which some process performs $\Omega(k)$ remote memory references to enter and exit its critical section.*

Our proof of this result extends techniques used by us and others in several earlier papers [2, 5, 7, 9, 10, 12, 16].

The rest of the paper is organized as follows. In Section 2, our system model is defined. Our lower bound proof is then sketched in Section 3. A formal proof of it is given in Section 4. We conclude in Section 5.

2 Definitions

In this section, we provide definitions pertaining to atomic shared-memory systems that will be used in obtaining our lower bound. In the following subsections, we define our model of an atomic shared-memory system (Section 2.1), state the properties required of a mutual exclusion algorithm implemented within this model (Section 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 (Section 2.3). The same model was used earlier by us to establish the previously-mentioned $\Omega(\log N/\log\log N)$ lower bound [5]. Therefore, most of the material in this section is taken directly from [5].

2.1 Atomic Shared-Memory Systems

Our model of an atomic shared-memory system is similar to that used by Anderson and Yang [7].

An *atomic 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 an atomic shared-memory system, we must formally 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.

Informally, an *event* is a particular execution of an atomic statement of some process that involves reading and/or writing 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.[2] Thus, we allow arbitrarily powerful operations on local variables. Since our proof applies to systems with reads, writes, and comparison primitives, it is important to formally define the notion of a comparison primitive. We define a *comparison primitive* to be 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 $g(temp,\ old,\ new) = old$. We call an execution of such a primitive a *comparison event*. As we shall see, our formal 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 $s4$:	*compare-and-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.

[2]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.

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 $[p, \mathsf{Op}, R, W]$, where $p \in P$ (the set of processes). We call Op the *operation* of event e, denoted $op(e)$. Op determines what kind of event e is, and 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 $\{v \mid (v, \alpha) \in R\}$, and $Wvar(e)$, the set of variables written by e, to be $\{v \mid (v, \alpha) \in W\}$. 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 $e = [p, \mathsf{Op}, R, W]$, 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 $e = [p, \mathsf{Op}, R, W]$ 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 our lower bound proof. (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 $\mathsf{compare}(v, 0)$.

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

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 $e = [p, \mathsf{Op}, R, W]$ and $f = [q, \mathsf{Op}', R', W']$ 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 $writer\_event(v, H)$,[3] and its owner is denoted $writer(v, H)$. If v is not written by any event in H, then we let $writer(v, H) = \perp$ and $writer\_event(v, H) = \perp$.

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

[3] 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.)

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.[4] If G is a subcomputation of H, then $H - G$ is the computation obtained by removing all events in G from H. 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 an atomic 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 write(x) or write(y) as its operation, depending on the value possibly written to a by other processes.

2.2 Mutual Exclusion Systems

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

[4] 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.

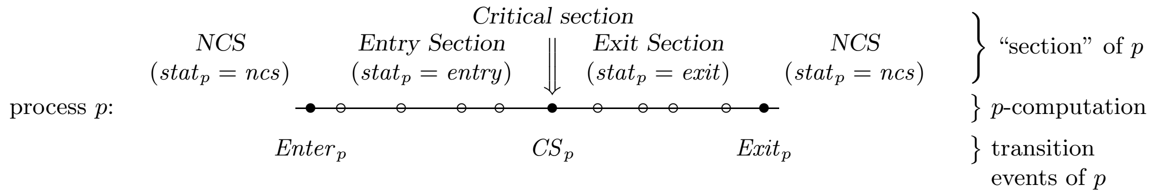

Figure 2: Transition events of an atomic mutual exclusion system. In this figure, NCS stands for "noncritical section," a circle ($\circ$) represents a non-transition event, and a bullet ($\bullet$) represents a transition event.

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 $Enter_p$ causes p to transit from its noncritical section to its entry section. Event CS_p causes p to transit from its entry section to its exit section.[5] Event $Exit_p$ causes p to transit from its exit section to its noncritical section. This behavior is depicted in Figure 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 $H \in C$,

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

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, an atomic 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 $H \in C$, define $X = \{q \in P \mid value(stat_q, H) \neq ncs\}$. 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 precludes multiple critical-section events from being simultaneously "enabled." 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.

[5] 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 (CC) 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.) The notion of a critical event presented here is taken directly from [5].

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 an atomic 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 $F \mid p$ 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 $writer(v, F) \neq p$.

— *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., $writer(v, F) = \perp$), 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 $writer(v, F) \neq p$.

— *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., $writer(v, F) = \perp$), 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 $v \notin Wvar(e_p)$), $writer(v, F) \neq p$, and either

(i) $F \mid p$ 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 $f_q = writer\_event(v, F)$, such that $F_2 \mid p$ does not contain an unsuccessful comparison event on v.

— *Informally, e_p must read the initial value of v (if $writer(v, F) = \perp$) 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).*[6] □

Note that state transition events do *not* actually cause cache misses; these events are defined as critical so that certain cases can be combined 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.

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 $writer(v, F) = q$ 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, $writer(v, F) = q$ 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

[6] 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.

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.) 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.) In fact, an algorithm with $O(1)$ time complexity in such systems is presented in [5].

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 Proof Strategy

In Section 4, we show that for any positive k, there exists some $\bar{N}$ such that, for any mutual exclusion system $\mathcal{S} = (C, P, V)$ with $|P| \geq \bar{N}$, there exists a computation H such that some process p experiences point contention k and executes at least k critical events to enter and exit its critical section. 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. The $\Omega(\log / \log\log N)$ lower bound mentioned earlier was also proved by considering such computations, so most of the definitions in this subsection are taken directly from [5]. 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 consider only 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.

$$\begin{aligned}\mathrm{Act}(H) &= \{p \in P \mid H \mid p \neq \langle\rangle \text{ and } \langle \mathit{Exit}_p \rangle \text{ is } \mathit{not} \text{ in } H\} \\ \mathrm{Fin}(H) &= \{p \in P \mid H \mid p \neq \langle\rangle \text{ and } \langle \mathit{Exit}_p \rangle \text{ is in } H\}\end{aligned} \qquad \square$$

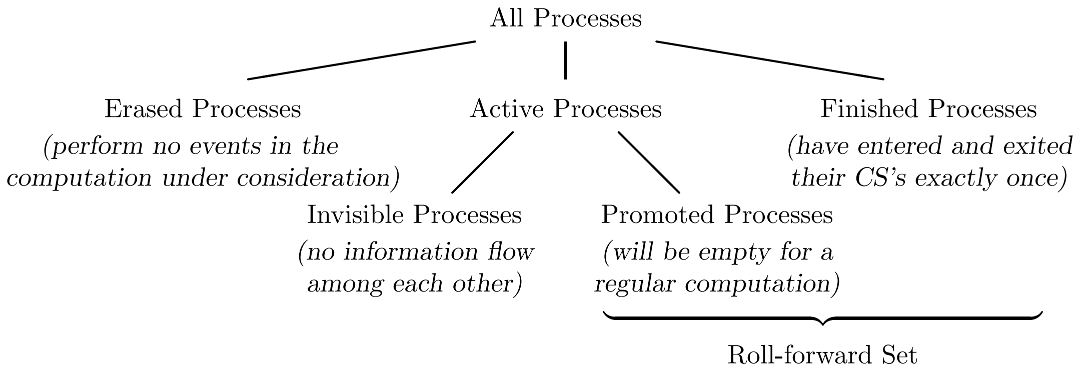

Figure 3: Process groups.

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.[7] 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 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 other lower bounds related to the mutual exclusion problem [5, 7, 9, 10, 12], as well as several other lower bounds for concurrent systems [2, 16].) 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 Figure 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

[7] 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.

of P such that $\mathrm{Fin}(H) \subseteq \mathit{RFS}$ and $H \mid p \neq \langle\rangle$ for each $p \in \mathit{RFS}$. 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 $E \circ \langle e_p \rangle \circ F \circ \langle f_q \rangle \circ G$.[8] 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.*, $\mathit{writer\_event}(v, F) = \perp$), then $p \in \mathit{RFS}$ 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 $p \in \mathit{RFS}$ 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 $\{p, q\} \subseteq \mathit{RFS}$ 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, $\mathit{writer}(v, H) \in \mathit{RFS}$ 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.*

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

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

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 ($p \in \mathrm{Act}(H) - \mathit{RFS}$);* (**ii**) *$p$ is in the process of "rolling forward" and is in its entry or exit section ($p \in \mathrm{Act}(H) \cap \mathit{RFS}$); 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 $H \mid (\{p\} \cup \mathit{RFS})$.

— *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

[8] 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.

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.

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

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. $\Box$

3.2 Detailed Proof Overview

Initially, we start with a regular computation H_1, where $\mathrm{Act}(H_1) = P$, $\mathrm{Fin}(H_1) = \{\}$, and each process has one critical event. We then inductively show that other longer computations exist, the last of which establishes our lower bound. Each computation is obtained by rolling forward or erasing some processes. We assume that P is large enough to ensure that enough non-erased processes remain after each induction step for the next step to be applied. The precise bound on $|P|$ is given in Theorem 2.

At the j^{th} induction step, we consider a computation H_j such that $\mathrm{Act}(H_j)$ consists of n processes that execute j critical events each. We construct a regular computation H_{j+1} such that $\mathrm{Act}(H_{j+1})$ consists of $\Omega(\sqrt{n}/k)$ processes, each of which executes $j+1$ critical events in H_{j+1}. The construction method, formally described in Lemma 7, is explained below. In constructing H_{j+1} from H_j, we may erase some processes and roll *at most two* processes forward. At the end of step $k-1$, we have a regular computation H_k in which each active process executes k critical events and $\mathrm{Fin}(H_k) \le 2(k-1)$. Since active processes have no knowledge of each other, we may erase all but one active process from H_k and obtain a valid computation. This computation has exactly one active process and at most $2(k-1)$ finished processes. Thus, its contention is at most $2k-1$. Moreover, the remaining active process performs k critical events, proving the desired lower bound.

We now describe how H_{j+1} is constructed from H_j. Let $n = |\mathrm{Act}(H_j)|$. As shown in Lemma 5, 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. We consider two cases, based on the variables remotely accessed by these next critical events.

Erasing strategy. Assume that there exist $\Omega(\sqrt{n})$ distinct variables that are remotely accessed by some next critical events. For each such variable v, we select one process whose next critical event accesses v. Let Y' be the set of selected processes. This situation is depicted in Figure 4. We now eliminate remaining possible conflicts among processes in Y' by constructing a "conflict graph" $\mathcal{G}$ as follows.

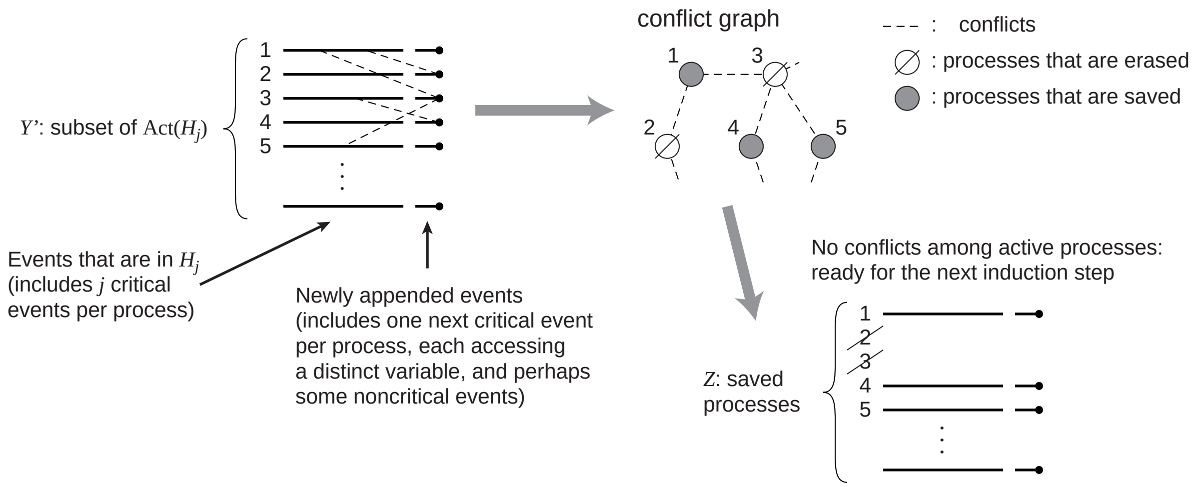

Figure 4: Erasing strategy. For simplicity, processes in $\mathrm{Fin}(H_j)$ are not shown.

Each process p in Y' is considered a vertex in $\mathcal{G}$. By induction, process p has j critical events in $\mathrm{Act}(H_j)$ and one next critical event. For each of the $j+1$ critical events of p, **(i)** if the event accesses the same variable as the next critical event of some other process q, introduce edge (p, q). In addition, **(ii)** if the next critical event of p remotely accesses a local variable of q, also introduce edge (p, q).

Since each process in Y' accesses a distinct remote variable in its next critical event, it is clear that each process generates at most $j+1$ edges by rule (i) and at most one edge by rule (ii). By applying Turán's theorem (Theorem 1), we can find a subset Z of Y' such that $|Z| = \Omega(\sqrt{n}/j)$ and their critical events do not conflict with each other. 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 the number of distinct variables that are remotely accessed by some next critical events is $O(\sqrt{n})$. This situation is depicted in Figure 5. Since there are $\Theta(n)$ next critical events, there exists a variable v that is remotely accessed by next critical events of $\Omega(\sqrt{n})$ processes. Let Y_v be the set of these processes. First, we retain Y_v and erase all other active processes. Let the resulting computation be H'. We then arrange the next critical events of Y_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.

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 (that are not erased so far), 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.

Let p_{LW} and p_{SC} be the owner of $LW(v)$ and $SC(v)$, respectively. (Depending on the computation, we may

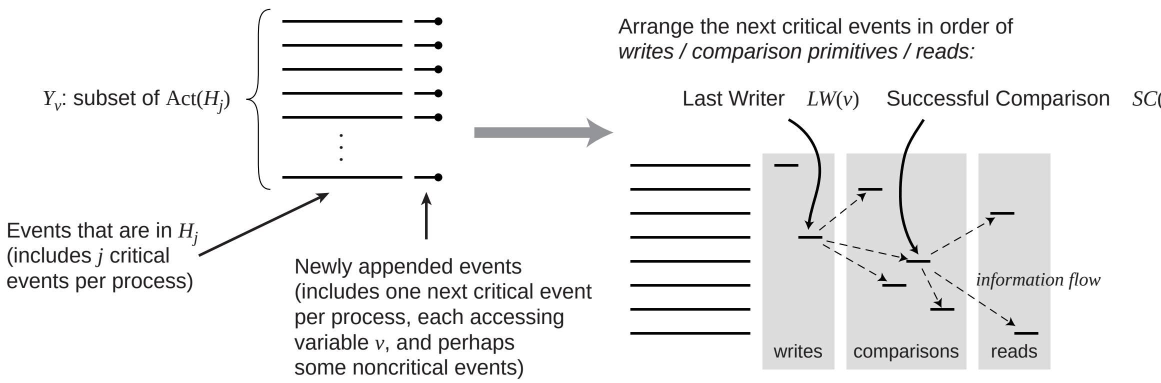

Figure 5: Roll-forward strategy. For simplicity, processes in $\mathrm{Fin}(H_j)$ are not shown.

have only one of them, or neither.) We then roll p_{LW} and p_{SC} forward by generating a regular computation G from H' such that $\mathrm{Fin}(G) = \mathrm{Fin}(H') \cup \{p_{\mathrm{LW}}, p_{\mathrm{SC}}\}$.

If either p_{LW} or p_{SC} executes at least k critical events before reaching its noncritical section, then the $\Omega(k)$ lower bound easily follows. Therefore, we can assume that either of p_{LW} and p_{SC} performs fewer than k critical events while being rolled forward. Each critical event of p_{LW} or p_{SC} that is appended to H' may generate information flow only if it reads a variable v that is written by another process in H'. Condition RF3 guarantees that if there are multiple processes that write to v, the last writer in H' is not active. Because information flow from an inactive process is allowed, a conflict arises only if there is a single process that writes to v in H'. Thus, each critical event of p_{LW} or p_{SC} conflicts with at most one process in Y_v, and hence can erase at most one process. (Appending a noncritical event to H' cannot cause any processes to be erased. In particular, if a noncritical remote read by p_{LW} (respectively, p_{SC}) is appended, then p_{LW} (respectively, p_{SC}) must have previously read the same variable. By RF3, if the last writer is another process, then that process is not active.)

Therefore, the entire roll-forward procedure erases fewer than $2k$ processes from $\mathrm{Act}(H') = Y_v$. We can assume $|P|$ is sufficiently large to ensure that $\sqrt{n} > 4k$. This ensures that $\Omega(\sqrt{n})$ processes survive after the entire procedure. (Actually, as seen in Theorem 2, we only ensure that $\Omega(\sqrt{n}/k)$ processes survive, in order to simplify bookkeeping. This results in a larger bound on $|P|$. However, it is only of secondary interest, since our main goal is a lower bound on the number of critical events.) Thus, we can construct H_{j+1}.

4 Detailed Lower Bound Proof

In this section, we establish our lower-bound theorem. Throughout this section, we assume the existence of a fixed mutual exclusion system $\mathcal{S} = (C, P, V)$. We began by stating six lemmas concerning mutual exclusion systems as defined here that were proved previously (in particular, in the paper that establishes the $\Omega(\log N/ \log\log N)$ lower bound mentioned earlier) [5].

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: $H \mid Y \in C$; RFS is a valid RF-set of $H \mid Y$; an event e in $H \mid Y$ is a critical event if and only if it is also a critical event 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

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 $q = \perp$, $q = p$, or $v_{\mathrm{rem}} \notin Rvar(e_p)$ holds, then no information flow occurs.)

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)
- *$G \mid p = \langle\rangle$;* (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* $q = \mathit{writer}(v_{\mathrm{rem}}, H \circ G)$*, then one of the following hold:* $q = \perp$, $q = p$, $q \in \mathit{RFS}$*, or* $v_{\mathrm{rem}} \notin \mathit{Rvar}(e_p)$. (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)
- $Y \subseteq \mathrm{Inv}_{\mathit{RFS}}(H)$; (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* $H \circ L$.

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 $n = |\mathrm{Act}(H)|$*. 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)
- $e_p \notin \{\mathit{Enter}_p, \mathit{CS}_p, \mathit{Exit}_p\}$; (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)

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

The following theorem is due to Turán [17].

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[9] exists with at least $\lceil |V|/(d+1) \rceil$ vertices.* □

The remaining lemma is unique to the lower bound established here and thus is presented with a full proof. This lemma provides the induction step that leads to the lower bound in Theorem 2.

Lemma 7 *Let k be a positive integer, and H be a computation. Assume the following:*

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

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

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

Then, one of the following propositions is true.

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

- *$F \circ \langle Exit_p \rangle \in C$;*
- *F does not contain $\langle Exit_p \rangle$;*
- *at most $|\text{Fin}(H) + 2|$ processes participate in F;*
- *p executes at least k critical events in F.*

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

- *$\text{Act}(G) \subseteq \text{Act}(H)$;* (52)
- *$|\text{Fin}(G)| \le |\text{Fin}(H) + 2|$;* (53)
- *$|\text{Act}(G)| \ge \min(\sqrt{n}/(2c+3),\ \sqrt{n} - 2k - 3)$;* (54)
- *each process in $\text{Act}(G)$ executes exactly $(c+1)$ critical events in G.* (55)

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

- $Y \subseteq \text{Act}(H)$, and (56)
- $n - 1 \le |Y| \le n$, (57)

[9]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'.

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

- $H \circ L_p \circ \langle e_p \rangle \in C$; (58)
- L_p is a p-computation; (59)
- L_p contains no critical events in $H \circ L_p$; (60)
- $e_p \notin \{Enter_p, CS_p, Exit_p\}$; (61)
- $\mathrm{Fin}(H)$ is a valid RF-set of $H \circ L_p$; (62)
- e_p is a critical event by p in $H \circ L_p \circ \langle e_p \rangle$. (63)

For each $p \in Y$, by (59), (60), and $p \in Y \subseteq \mathrm{Act}(H)$, we have

$$\mathrm{Act}(H \circ L_p) = \mathrm{Act}(H) \quad \wedge \quad \mathrm{Fin}(H \circ L_p) = \mathrm{Fin}(H). \tag{64}$$

By (50) and (57), 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.

$$\mathcal{E}_H = \{f_q \text{ in } H \mid f_q \text{ is critical in } H \text{ and } q \in Y\}. \tag{65}$$

Define $\mathcal{E} = \mathcal{E}_H \cup \{e_p \mid p \in Y\}$, *i.e.*, the set of all "past" and "next" critical events of processes in Y. From (51), (56), and (57), it follows that

$$|\mathcal{E}| = (c+1)|Y| \leq (c+1)n. \tag{66}$$

Define V_{next} as the set of variables remotely accessed by some "next" critical events:

$$V_{\text{next}} = \{v \in V \mid \text{there exists } p \in Y \text{ such that } e_p \text{ remotely accesses } v\}. \tag{67}$$

We consider two cases, depending on the size of V_{next}.

Case 1: $|V_{\text{next}}| \geq \sqrt{n}$ (erasing strategy)

— *In this case, we construct a subset Y' of Y by selecting one process for each variable in V_{next}. Clearly, $|Y'| = |V_{\text{next}}|$. We then construct a "conflict graph" $\mathcal{G}$, where each vertex is a process 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. By applying Lemma 1 to H and $Z \cup \mathrm{Fin}(H)$, and extending the resulting computation H' with next critical events, we construct a computation G that satisfies Proposition Pr2.*

By definition, for each variable v in V_{next}, there exists a process p in Y such that e_p remotely accesses v. Therefore, we can arbitrarily select one such process for each variable v in V_{next} and construct a set Y' of processes such that

- $Y' \subseteq Y$, (68)
- if $p \in Y'$, $q \in Y'$ and $p \neq q$, then e_p and e_q access different remote variables, and (69)
- $|Y'| = |V_{\text{next}}| \geq \sqrt{n}$. (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 per process. Since, by (51), y executes exactly c critical events in H, by (69), Rule R2 can introduce at most c edges per process.

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

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

$$|Z| \geq |Y'|/(2c+3) \geq \sqrt{n}/(2c+3), \tag{72}$$

where the latter inequality follows from (70).

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

Define H' as

$$H' = H \mid (Z \cup \mathrm{Fin}(H)). \tag{73}$$

By (56), (68), and (71), we have

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

and hence,

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

We now apply Lemma 1, with '*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 (48) and (49), respectively; (3) is trivial. It follows that

- $H' \in C$, (76)
- $\mathrm{Fin}(H)$ 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,

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

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

$$F_z = (H \circ L_z) \mid (Z \cup \mathrm{Fin}(H)). \tag{80}$$

By (74), we have $z \in Y$. Thus, applying (58), (59), (60), and (62) 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 (given in Section 2.1), (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 (81), (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 (61), (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 (61), (63), 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 facts, 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 $q = \mathit{writer}(v_j, H' \circ L)$, 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 (65), 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. $\Box$

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 (64) 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 (58) and (63), 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 $\mathit{writer}(v_{j+1}, H' \circ L \circ E_j) = \mathit{writer}(v_{j+1}, H' \circ L)$. Thus, by (93) and Claim 1,

  - if we let $q = \mathit{writer}(v_{j+1}, H' \circ L \circ E_j)$, 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 the 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), which implies $\mathrm{Act}(G) = Z \subseteq \mathrm{Act}(H)$, G satisfies (52) and (53). By (72), we have (54). By (51), (78), and (91), each process in Z executes exactly c critical events in $H' \circ L$. Thus, by Claim 2, G satisfies (55).

Case 2: $|V_{\mathrm{next}}| \leq \sqrt{n}$ (roll-forward strategy)

— *In this case, there exists a variable v that is remotely accessed by next critical events of at least $\sqrt{n}-1$ processes. Let Y_v be the set of these processes. We retain Y_v and erase all other active processes. Let the resulting computation be H'. We then roll forward processes p_{LW} and p_{SC} of Y_v to generate a regular computation G. If either p_{LW} or p_{SC} executes k or more critical events before finishing its execution, the resulting computation satisfies Proposition Pr1. Otherwise, fewer than $2k$ processes are erased during the procedure, which makes G satisfy Proposition Pr2, with at least $\sqrt{n}-2k$ active processes.*

For each variable v_j in V_{next}, define $Y_{v_j} = \{p \in Y \mid e_p \text{ remotely accesses } v_j\}$. By (57) and (67), $|V_{\mathrm{next}}| \leq \sqrt{n}$ implies that there exists a variable v in V_{next} such that $|Y_v| \geq (n-1)/\sqrt{n}$ holds. (In the rest of Case 2, we consider v to be a fixed variable.) Then, the following holds:

$$|Y_v| \geq (n-1)/\sqrt{n} > \sqrt{n} - 1. \tag{111}$$

Define

$$H' = H \mid (Y_v \cup \mathrm{Fin}(H)). \tag{112}$$

Using $Y_v \subseteq Y \subseteq \mathrm{Act}(H)$, we also have

$$\mathrm{Act}(H') = Y_v \subseteq \mathrm{Act}(H) \quad \wedge \quad \mathrm{Fin}(H') = \mathrm{Fin}(H). \tag{113}$$

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

- $H' \in C$, (114)
- $\mathrm{Fin}(H)$ is a valid RF-set of H', and (115)
- an event in H' is critical if and only if it is also critical in H. (116)

Our goal now is to show that H' can be extended to a computation $\overline{G}$ (defined later), so that each process in Y_v has one more critical event. By (113), (115), and by the definition of a finished process,

$$\mathrm{Inv}_{\mathrm{Fin}(H)}(H') = \mathrm{Act}(H') = Y_v. \tag{117}$$

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

$$F_s = (H \circ L_s) \mid (Y_v \cup \mathrm{Fin}(H)). \tag{118}$$

Since $Y_v \subseteq Y$, we have $s \in Y$. Thus, applying (58), (59), (60), and (62) with 'p' $\leftarrow s$, it follows that

- $H \circ L_s \circ \langle e_s \rangle \in C$; (119)
- L_s is an s-computation; (120)
- L_s contains no critical events in $H \circ L_s$; (121)
- $\mathrm{Fin}(H)$ is a valid RF-set of $H \circ L_s$. (122)

By P1, (119) implies

$$H \circ L_s \in C. \tag{123}$$

We now apply Lemma 1, with 'H' $\leftarrow H \circ L_s$, 'RFS' $\leftarrow \mathrm{Fin}(H)$, and '$Y$' $\leftarrow Y_v \cup \mathrm{Fin}(H)$. Among the assumptions stated in Lemma 1, (1) and (2) follow from (123) and (122), respectively; (3) is trivial. It follows that

- $F_s \in C$, and (124)
- an event in F_s is critical if and only if it is also critical in $H \circ L_s$. (125)

Since $s \in Y_v$, by (112), (118), (120), and (124), we have

- $F_s = H' \circ L_s \in C$. (126)

Hence, by (121) and (125),

- L_s contains no critical events in $F_s = H' \circ L_s$. (127)

We now show that the events in $\{L_s \mid s \in Y_v\}$ can be "merged" by applying Lemma 4. We arbitrarily index Y_v as $\{s_1, s_2, \ldots, s_m\}$, where $m = |Y_v|$. (Later, we construct a specific indexing of Y_v 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' $\leftarrow \mathrm{Fin}(H)$, '$Y$' $\leftarrow Y_v$, and 'p_j' $\leftarrow s_j$ for each $j = 1, \ldots, m$. Among the assumptions stated in Lemma 4, (22)–(24) follow from (114), (115), and (117), respectively; (25)–(27) follow from (120), (126), and (127), respectively, with 's' $\leftarrow s_j$ for each $j = 1, \ldots, m$. This gives us the following.

- $H' \circ L \in C$; (128)
- $\mathrm{Fin}(H)$ is a valid RF-set of $H' \circ L$; (129)
- L contains no critical events in $H' \circ L$. (130)

By (112) and the definition of L, we also have,

- for each $s \in Y_v$, $(H \circ L_s) \mid (\{s\} \cup \mathrm{Fin}(H)) = (H' \circ L) \mid (\{s\} \cup \mathrm{Fin}(H))$; (131)
- for each $s \in Y_v$, $(H' \circ L) \mid s = (H \circ L_s) \mid s$. (132)

We now re-index the processes in Y_v so that information flow among them is minimized. We place next critical events of Y_v by placing write, comparison, and read events in that order. Furthermore, we can arrange comparison events such that at most one of them succeeds, as explained in Section 3. Let $(s^1, s^2, \ldots, s^m)$ be the indexing of Y_v thus constructed, and E be the appended computation that consists of next critical events by processes in Y. Then, we have the following:

- $\overline{G} \in C$, where $\overline{G} = H' \circ L \circ E$ and $E = \langle e'_{s^1}, e'_{s^2}, \ldots, e'_{s^m} \rangle$; (133)
- $e'_{s^j} \sim e_{s^j}$. (134)

By the definition of E,

- E is an Y_v-computation. (135)

By (61), (130), and (134), $L \circ E$ does not contain any transition events. Moreover, by the definition of L and E, $(L \circ E) \mid p \neq \langle\rangle$ implies $p \in Y_v$, for each process p. Combining these assertions with (113), we have

$$\begin{aligned} \mathrm{Act}(\overline{G}) &= \mathrm{Act}(H' \circ L) = \mathrm{Act}(H') = Y_v \quad \wedge \\ \mathrm{Fin}(\overline{G}) &= \mathrm{Fin}(H' \circ L) = \mathrm{Fin}(H') = \mathrm{Fin}(H). \end{aligned} \tag{136}$$

We now state and prove two claims regarding $\overline{G}$. Claim 3 follows easily from the re-indexing of Y_v and construction of E, described above.

Claim 3: Events in E appear in the following order, where α is a fixed value in the range of v and $W(v)$, $C_1(v)$, $C_2(v)$, and $R(v)$ are sets of events.

- 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 $\beta_s \neq \alpha$;
- events in $C_2(v)$: each event e'_s in $C_2(v)$ satisfies $op(e'_s) = \mathsf{compare}(v, \alpha)$;
- 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 α. 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 α, by definition. Thus, if there is a successful comparison, then all subsequent comparison events must fail.) Define $LW(v)$, the "last write," and $SC(v)$, the "successful comparison," as follows:

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

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

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

Before establishing our next claim, Claim 4, we define p_{LW} and p_{SC} as $owner(LW(v))$ and $owner(SC(v))$, respectively. If $LW(v)$ (respectively, $SC(v)$) equals $\perp$, then p_{LW} (respectively, p_{SC}) also equals $\perp$. We also define RFS as

$$RFS \;=\; \mathrm{Fin}(H) \;\cup\; \{p \mid p \in \{p_{\mathrm{LW}}, p_{\mathrm{SC}}\} \text{ and } p \neq \perp\}. \tag{137}$$

By the definition of Y_v, for each $p \in Y_v$, e_p remotely accesses v. In particular,

- for each $p \in Y_v$, v is remote to p. (138)

Note that "expanding" a valid RF-set does not falsify any of RF1–RF5. Therefore, using (129), (136), and $\mathrm{Fin}(H) \subseteq RFS \subseteq \mathrm{Fin}(H) \cup Y_v$, it follows that

- RFS is a valid RF-set of $H' \circ L$. (139)

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 (133) and P1, we have the following:

$$H' \circ L \circ E_j \circ \langle e'_{s^{j+1}}\rangle = H' \circ L \circ E_{j+1} \in C, \quad \text{for each } j. \tag{140}$$

Also, by the definition of E_j, we have

$$E_j \mid s^{j+1} = \langle\rangle, \quad \text{for each } j. \tag{141}$$

At each step, we assume

- $H' \circ L \circ E_j$ satisfies RF5. (142)

The induction base ($j = 0$) follows easily from (139), since $E_0 = \langle\rangle$.

Assume that (142) holds for a particular value of j. Since $s^{j+1} \in Y_v \subseteq Y$, we have

$$s^{j+1} \in Y, \tag{143}$$

and $s^{j+1} \in \mathrm{Act}(H)$. By applying (64) with 'p' $\leftarrow s^{j+1}$, and using (143), we also have $\mathrm{Act}(H \circ L_{s^{j+1}}) = \mathrm{Act}(H)$, and hence

$$s^{j+1} \in \mathrm{Act}(H \circ L_{s^{j+1}}). \tag{144}$$

Also, by (138),

- no events in E_j access any of s^{j+1}'s local variables. (145)

We use Lemma 2 twice in sequence in order to prove Claim 4. First, by P3, and applying (119), (128), and (132) 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 (146)
- $e''_{s^{j+1}} \sim e_{s^{j+1}}$. (147)

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 (146), (129), (147), and (144), respectively; (11) and (12) hold vacuously by 'G' $\leftarrow \langle\rangle$; (4), (6), and (10) follow by applying (119), (122), and (131), respectively, with 's' $\leftarrow s^{j+1}$; (13) follows by applying (63) with 'p' $\leftarrow s^{j+1}$, and using (143). It follows that

- $e''_{s^{j+1}}$ is critical in $H' \circ L \circ \langle e''_{s^{j+1}}\rangle$. (148)

Before applying Lemma 2 again, we establish the following preliminary assertions. Since $\mathrm{Fin}(H) \subseteq RFS$, by applying (122) with 's' $\leftarrow s^{j+1}$, it follows that

- RFS is a valid RF-set of $H \circ L_{s^{j+1}}$. (149)

We now establish a simple claim.

> **Claim 4-1:** If $e_{s^{j+1}}$ is a comparison event on v, and if E_j contains a write to v, then $E_j \mid RFS$ also contains a write to v.

Proof of Claim: By (134) and Claim 3, we have $e'_{s^{j+1}} \in C_1(v) \cup C_2(v)$. Hence, by Claim 3, if an event e'_{s^k} (for some $k \le j$) 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 (137), 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 (137), both E_j and $E_j \mid RFS$ 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 (146), (140), (139), (139), (141), (145), and (148), respectively; (10) is trivial; (8) follows from (147) and by applying (134) with 's^j' $\leftarrow s^{j+1}$; (9) follows from (136) and $s^{j+1} \in Y_v$. Moreover, Assumption (A) follows from (142), 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. (150)

  The induction base ($j = 0$) follows easily from (139), since $E_0 = \langle\rangle$.

  Assume that (150) holds for a particular value of j. By Claim 3, if $e'_{s^{j+1}}$ reads v, then the following holds: $e'_{s^{j+1}} \in C_1(v) \cup C_2(v) \cup R(v)$; every event in $W(v)$ is contained in E_j; $writer(v, H' \circ L \circ E_j)$ is one of $LW(v)$ or $SC(v)$ or $\perp$. Therefore, by (137), we have the following:

  - if $e'_{s^{j+1}}$ remotely reads v, and if we let $q = writer(v, H' \circ L \circ E_j)$, then either $q = \perp$ or $q \in RFS$ holds. (151)

  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 (140), (139), (150), (141), and (151), respectively; (16) follows from (136) and $s^{j+1} \in Y_v$; (18) follows from (136) and (135); (20) follows from (136) and (138). It follows that $H' \circ L \circ E_{j+1}$ satisfies RF1 and RF2.

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

  By (136), we have $\{p, q\} \subseteq Y_v$. If $u = v$, then by Claim 3, $writer\_event(u, \overline{G})$ is either $SC(u)$ (if $SC(u) \ne \perp$) or $LW(u)$ (otherwise). (Since we assumed that there exists a write to u, they both cannot be $\perp$.) Thus, by (137), we have $r \in RFS$.

  On the other hand, assume $u \ne v$. 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 $u \in Wvar(e'_s)$, then since $u \ne v$, u is local to s. Since at least one of p or q is different from s, without loss of generality, assume $p \ne s$. Since $p \in Y_v$ and $Y_v \subseteq \mathrm{Act}(H)$, we have $p \notin \mathrm{Fin}(H)$. Thus, by (129) 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 (136), $\mathrm{Act}(H' \circ L) = Y_v$, which contradicts $s \in Y_v$ (which follows from (135), since e'_s is an event of E).

It follows that there exists no event e'_s in E such that $u \in \mathit{Wvar}(e'_s)$ holds. Thus, we have $r = \mathit{writer}(u, H' \circ L)$. By (129) and applying RF3 with 'RFS' $\leftarrow \mathrm{Fin}(H)$ to f_p and g_q in $H' \circ L$, we have $\mathit{writer}(u, H' \circ L) \in \mathrm{Fin}(H) \subseteq \mathit{RFS}$.

- Consider the case in which f_p is in $H' \circ L$ and $g_q = e'_{s^k}$, for some $s^k \in Y_v$. By (136) and our assumption that p and q are both in $\mathrm{Act}(\overline{G})$, we have $p \in \mathrm{Act}(H' \circ L)$ and $q \in \mathrm{Act}(H' \circ L)$. Since $u \neq v$, u is local to q. However, by (129), 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.
- Consider the case in which $f_p = e'_{s^j}$ and $g_q = e'_{s^k}$, for some s^j and s^k in Y_v. Since u is remote to at least one of s^j or s^k, we have $u = v$, a contradiction.

- **RF4:** By (61), (129), and (136), it easily follows that $\overline{G}$ satisfies RF4 with respect to RFS.

Therefore, we have established that

- RFS is a valid RF-set of $\overline{G}$. (152)

By (136) and (137), we have

$$\mathrm{Pmt}_{\mathit{RFS}}(\overline{G}) \;=\; \{p \mid p \in \{p_{\mathrm{LW}}, p_{\mathrm{SC}}\} \text{ and } p \neq \perp\}.$$

In particular,

$$|\mathrm{Pmt}_{\mathit{RFS}}(\overline{G})| \leq 2. \tag{153}$$

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 $G_0,\ G_1,\ \ldots,\ G_l$ (where $G_0 = \overline{G}$), satisfying the following:

- $G_j \in C$; (154)
- RFS is a valid RF-set of G_j; (155)
- $\mathrm{Pmt}(G_j) \subseteq \mathrm{Pmt}(\overline{G})$; (156)
- each process in $\mathrm{Inv}(G_j)$ executes exactly $c+1$ critical events in G_j; (157)
- the processes in $\mathrm{Pmt}(\overline{G})$ collectively execute exactly $|\mathrm{Pmt}(\overline{G})| \cdot (c+1) + j$ critical events in G_j; (158)
- $\mathrm{Inv}(G_{j+1}) \subseteq \mathrm{Inv}(G_j)$ and $|\mathrm{Inv}(G_{j+1})| \geq |\mathrm{Inv}(G_j)| - 1$ if $j < l$; (159)
- $\mathrm{Fin}(G_j) \subsetneq \mathit{RFS}$ if $j < l$, and $\mathrm{Fin}(G_j) = \mathit{RFS}$ if $j = l$. (160)

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 $G_0 = \overline{G}$, (154) and (155) follow from (133) and (152), respectively. Condition (156) is trivial.

By (51), (116), and (130), each process in Y_v executes exactly c critical events in $H' \circ L$. Thus, by Claim 4, it follows that each process in Y_v executes exactly $c+1$ critical events in $\overline{G}$. Since $\mathrm{Inv}(\overline{G}) \subseteq Y_v$, $\overline{G}$ satisfies (157). Since $\mathrm{Pmt}(\overline{G}) \subseteq Y_v$, $\overline{G}$ satisfies (158).

Induction step: At each step, we assume (154)–(158). If $\mathrm{Fin}(G_j) = \mathit{RFS}$, then (160) is satisfied and we finish the induction, by letting $l = j$.

Assume otherwise. We apply Lemma 6 with 'H' $\leftarrow G_j$. Assumptions (35)–(37) stated in Lemma 6 follow from (154), (155), and $\mathrm{Fin}(G_j) \neq RFS$. The lemma implies that a computation G_{j+1} exists satisfying (154)–(160), as shown below.

Condition (154) and (155) follow from (38) and (39), respectively. Since G_j satisfies (156), by (46), G_{j+1} also satisfies (156). Since $\mathrm{Inv}(G_{j+1)} \subseteq \mathrm{Inv}(G_j)$ by (40) and (41), by (43) and (47), and applying (157) to G_j, it follows that G_{j+1} satisfies (157). By (43)–(47), and applying (156) and (158) to G_j, it follows that G_{j+1} satisfies (158). Condition (159) follows from (40) and (41). Thus, the induction is established. □

We now show that $l < 2k$. Assume otherwise. By (153), and by applying (158) to G_l, it follows that there exists a process $p \in \mathrm{Pmt}(\overline{G})$ (*i.e.*, p is either p_{LW} or p_{SC}) such that p executes at least $c+1+k$ critical events in G_l. From (160) and $p \in \mathrm{Pmt}(\overline{G}) \subseteq RFS$, we get $p \in \mathrm{Fin}(G_l)$. Let $\overline{F} = G_l \mid RFS$. By Lemma 1, and applying (154) and (155), we have the following:

- $\overline{F} \in C$;
- RFS is a valid RF-set of $\overline{F}$;
- p executes at least $c+1+k$ critical events in $\overline{F}$.

Since $p \in \mathrm{Fin}(G_l)$, by applying RF4 to p in G_l, it follows that the last event of $G_l \mid p$ is $Exit_p$. Since $G_l \mid p = \overline{F} \mid p$, $\overline{F}$ can be written as $F \circ \langle Exit_p \rangle \circ \cdots$, where F is a prefix of $\overline{F}$ such that p executes at least $c+k$ critical events in F. However, p and F then satisfy Proposition Pr1, a contradiction.

Finally, we show that G_l satisfies Proposition Pr2. The following derivation establishes (54).

$$
\begin{aligned}
|\mathrm{Act}(G_l)| &= |\mathrm{Inv}_{RFS}(G_l)| && \{\text{by (160), } RFS = \mathrm{Fin}(G_l)\text{, thus } \mathrm{Act}(G_l) = \mathrm{Inv}_{RFS}(G_l)\} \\
&\geq |\mathrm{Inv}_{RFS}(G_0)| - l && \{\text{by repeatedly applying (159)}\} \\
&= |\mathrm{Act}(\overline{G}) - RFS| - l && \{\text{by the definition of “Inv”; note that } \overline{G} = G_0\} \\
&= |Y_v - RFS| - l && \{\text{by (136)}\} \\
&= |Y_v - (\mathrm{Pmt}(\overline{G}) \cup \mathrm{Fin}(H))| - l && \{\text{because } RFS = \mathrm{Pmt}(\overline{G}) \cup \mathrm{Fin}(\overline{G})\text{, and} \\
& && \quad \mathrm{Fin}(\overline{G}) = \mathrm{Fin}(H) \text{ by (136)}\} \\
&= |Y_v - \mathrm{Pmt}(\overline{G})| - l && \{\text{because } Y_v \cap \mathrm{Fin}(H) = \{\} \text{ by (136)}\} \\
&> |Y_v| - 2 - 2k && \{\text{by (153) and } l < 2k\} \\
&> \sqrt{n} - 2k - 3. && \{\text{by (111)}\}
\end{aligned}
$$

Moreover, by (155) and (160), we have $\mathrm{Act}(G_l) = \mathrm{Inv}(G_l)$. Thus, by (136) and (159), we have $\mathrm{Act}(G_l) \subseteq \mathrm{Inv}(\overline{G}) \subseteq \mathrm{Act}(\overline{G}) = Y_v \subseteq \mathrm{Act}(H)$, which implies (52). By (137) and (160), we have (53). Finally, (157) implies (55). Therefore, G_l satisfies Proposition Pr2. □

Theorem 2 *Let $\bar{N}(k) = (2k+4)^{2(2^k-1)}$. For any mutual exclusion system $\mathcal{S} = (C, P, V)$ and for any positive number k, if $|P| \geq \bar{N}(k)$, then there exists a computation H such that at most $2k-1$ processes participate in H and some process p executes at least k critical operations in H to enter and exit its critical section.*

Proof: Let $H_1 = \langle Enter_1,\ Enter_2,\ \ldots,\ Enter_N \rangle$, where $P = \{1, 2, \ldots, N\}$ and $N \geq \bar{N}(k)$. 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 $(H_1,\ H_2,\ \ldots)$, such that each process in $\mathrm{Act}(H_j)$ has j critical events in H_j. We repeat the process until either H_k is constructed or some H_j satisfies Proposition Pr1 of Lemma 7.

If some H_j ($j \le k-1$) satisfies Proposition Pr1, then consider the first such j. By our choice of j, each of $H_1, \ \ldots, \ H_j - 1$ satisfies Proposition Pr2 of Lemma 7. Therefore, since $|\mathrm{Fin}(H_1)| = 0$, we have $|\mathrm{Fin}(H_j)| \le 2(j-1) \le 2k-4$. It follows that computation $F \circ \langle \mathit{Exit}_p \rangle$, generated by applying Lemma 7 to H_j, satisfies Theorem 2.

The remaining possibility is that each of $H_1, \ \ldots, \ H_{k-1}$ satisfies Proposition Pr2. We claim that, for $1 \le j \le k$, the following holds:

$$|\mathrm{Act}(H_j)| \ge (2k+4)^{2(2^{k+1-j}-1)}. \tag{161}$$

The induction basis ($j = 1$) directly follows from $\mathrm{Act}(H) = P$ and $|P| \ge \bar{N}(k)$. In the induction step, assume that (161) holds for some j ($1 \le j < k$), and let $n_j = |\mathrm{Act}(H_j)|$. Note that each active process in H_j executes exactly j critical events. By (161), we also have $n_j > (2k+4)^2$, which in turn implies that $\sqrt{n_j} - 2k - 3 > \sqrt{n_j}/(2k+4)$. Therefore, by (54), we have

$$|\mathrm{Act}(H_{j+1})| \ \ge \ \min(\sqrt{n_j}/(2j+3), \ \sqrt{n_j} - 2k - 3) \ \ge \ \sqrt{n_j}/(2k+4),$$

from which the induction easily follows.

Finally, (161) implies $|\mathrm{Act}(H_k)| \ge 1$, and Proposition Pr2 implies $|\mathrm{Fin}(H_k)| \le 2(k-1)$. Therefore, select any arbitrary process p from $\mathrm{Act}(H_k)$. Define $G = H_k \mid (\mathrm{Fin}(H_k) \cup \{p\})$. Clearly, at most $2k-1$ processes participate in G. By applying Lemma 1 with 'H' $\leftarrow H_k$ and 'Y' $\leftarrow \mathrm{Fin}(H_k) \cup \{p\}$, we have the following: $G \in C$, and an event in G is critical if and only if it is also critical in H_k. Hence, because p executes k critical events in H_k, G is a computation that satisfies Theorem 2. □

5 Concluding Remarks

We have established a lower bound that eliminates the possibility of an adaptive mutual exclusion algorithm based on reads, writes, or comparison primitives with $O(\log k)$ RMR time complexity.

We believe that $\Omega(\mathit{min}(k, \log N))$ is probably a tight lower bound for the class of algorithms considered in this paper (which would imply that the algorithm in [5] is optimal). One relevant question is whether the results of this paper be combined with those of [5] to come close to an $\Omega(\mathit{min}(k, \log N))$ bound, *i.e.*, can we at least conclude that $\Omega(\mathit{min}(k, \log N/\log\log N))$ is a lower bound? Unfortunately, the answer is no. We have shown that $\Omega(k)$ RMR time complexity is required *provided* N is sufficiently large. This leaves open the possibility that an algorithm might have $\Theta(k)$ RMR time complexity for very "low" levels of contention, but $o(k)$ RMR time complexity for "intermediate" levels of contention. Although our lower bound does not preclude such a possibility, we find it highly unlikely.

References

[1] Y. Afek, H. Attiya, A. Fouren, G. Stupp, and D. Touitou. Long-lived renaming made adaptive. In *Proceedings of the 18th Annual ACM Symposium on Principles of Distributed Computing*, pages 91–103. ACM, May 1999.

[2] 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.

[3] Y. Afek, G. Stupp, and D. Touitou. Long-lived adaptive splitter and applications. *Distributed Computing*, 15(2):67–86, 2002.

[4] 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-Verlag, October 2000.

[5] J. Anderson and Y.-J. Kim. An improved lower bound for the time complexity of mutual exclusion. *Distributed Computing*, 15(4):221–253, December 2003.

[6] J. Anderson, Y.-J. Kim, and T. Herman. Shared-memory mutual exclusion: Major research trends since 1986. *Distributed Computing*, 16:75–110, 2003.

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

[8] 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.

[9] 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.

[10] J. Burns and N. Lynch. Bounds on shared memory for mutual exclusion. *Information and Computation*, 107(2):171–184, December 1993.

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

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

[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-Verlag, October 2001.

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

[15] 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.

[16] 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.

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