The Price of being Adaptive*

[Extended Abstract]

Ohad Ben-Baruch
Department of Computer-Science
Ben-Gurion University of the Negev
Be'er Sheva, Israel
bbohad@gmail.com

Danny Hendler
Department of Computer-Science
Ben-Gurion University of the Negev
Be'er Sheva, Israel
hendlerd@cs.bgu.ac.il

ABSTRACT

Mutual exclusion is a fundamental distributed coordination problem. Shared-memory mutual exclusion research focuses on *local-spin* algorithms and uses the *remote memory references* (RMRs) metric. To ensure the correctness of concurrent algorithms in general, and mutual exclusion algorithms in particular, it is often required to prohibit certain re-orderings of memory instructions that may compromise correctness, by inserting *memory fence* (a.k.a. *memory barrier*) instructions. Memory fences incur non-negligible overhead and may significantly increase time complexity.

A mutual exclusion algorithm is *adaptive to total contention* (or simply adaptive), if the time complexity of every passage (an entry to the critical section and the corresponding exit) is a function of *total contention*, that is, the number of processes, k, that participate in the execution in which that passage is performed. We say that an algorithm A is *f-adaptive* (and that f is an *adaptivity function* of A), if the time complexity of every passage in A is $O\big(f(k)\big)$. Adaptive implementations are desirable when contention is much smaller than the total number of processes, n, sharing the implementation.

Recent work [5] presented the first read/write mutual exclusion algorithm with asymptotically optimal complexity under both the RMRs and fences metrics: each passage through the critical section incurs O(log n) RMRs and a constant number of fences. The algorithm works in the popular Total Store Ordering (TSO) model. The algorithm of [5] is non-adaptive, however, and they posed the question of whether there exists an adaptive mutual exclusion algorithm with the same complexities.

We provide a negative answer to this question, thus capturing an inherent cost of adaptivity. In fact, we prove a stronger result: adaptive read/write mutual exclusion algorithms with constant fence complexity do not exist, regardless of their RMR complexity. This result follows from a general tradeoff that we establish for such algorithms, between the fence complexity and the growth rate of adaptivity functions. Specifically, we prove that the fence complexity of any such algorithm with a linear (or sub-linear) adaptivity function is $\Omega(\log \log n)$. The tradeoff holds for implementations that may use compare-and-swap operations, in addition to reads and writes.

We show that our results apply also to obstruction-free implementations of well-known objects, such as counters, stacks and queues.

Categories and Subject Descriptors

D.1.3 [**Programming Techniques**]: Concurrent Programming—*Parallel programming*; D.2.8 [**Analysis of Algorithms and Problem Complexity**]: Tradeoffs between Complexity Measures; F.1.3 [**Complexity Measures and Classes**]: Relations among complexity measures

General Terms

Theory, Performance

Keywords

Mutual exclusion; shared-memory; lower bounds; total store ordering; time complexity; remote memory reference (RMR)

1. INTRODUCTION

In the *mutual exclusion* problem, a set of processes must coordinate their accesses to a *critical section* (CS) so that, at any point in time, at most a single process is inside the CS. Introduced by Dijkstra in 1965 [9], the mutual exclusion problem is a fundamental Distributed Computing problem and is still the focus of intense research [2, 23].

For more than 20 years, shared-memory mutual exclusion research has investigated the *remote memory references* (RMR) complexity of local-spin mutual exclusion algorithms; much of this work focuses on (deterministic) read/write mutual exclusion (e.g. [8, 14, 15, 17, 25]). Anderson and Yang were the first to present an n-process mutual exclusion algorithm, where every *passage* (an entry to the critical section and the corresponding exit) incurs $O(\log n)$ RMRs. This is optimal [7].

A mutual exclusion algorithm A is adaptive, if its RMR complexity is a function of the number of active processes.

*Partially supported by the Israel Science Foundation (grants 1227/10, 1749/14) and by the Lynne and William Frankel Center for Computing Science at Ben-Gurion University.

PODC'15, July 21–23, 2015, Donostia-San Sebastián, Spain.

ACM 978-1-4503-3617-8 /15/07 ...$15.00.
DOI: http://dx.doi.org/10.1145/2767386.2767428.

More formally, an algorithm is *f-adaptive to total contention* (henceforth simply *adaptive*), if the RMR complexity of every passage is $O\big(f(k)\big)$, where k denotes *total contention*, that is, the number of processes that participate in the execution. It is *f-adaptive to interval contention* (respectively, *point contention*) if the RMR complexity of every passage $\mathcal{P}$ is $O\big(f(k)\big)$, where k is the number of processes that are active during $\mathcal{P}$ (respectively, the maximum number of processes that are concurrently active at some point in time during $\mathcal{P}$). We call f the *adaptivity function* of A. Adaptive algorithms are desirable when the number of active processes is often significantly smaller than n, the total number of processes.

Mutual exclusion algorithms are almost always designed under the assumption that memory accesses are atomic, i.e. linearizable [12], or at least sequentially consistent [18]. In practice, however, modern compilers optimize code so as to issue certain instructions out of order, based on the memory model supported by the architecture.

The memory model dictates which operation pairs can be reordered [1, Figure 8]. For example, the widely-supported *total store ordering* (*TSO*) model [19] ensures that writes are not reordered, but it is possible to perform a read from address a before a write to address $b \neq a$ that is earlier in program order is performed.

The TSO model is supported by several common architectures, including SPARC [19] and x86 [13].[1] It is weaker than sequential consistency, and hence, also weaker than linearizability.

To ensure the correctness of a concurrent algorithm, it is possible to prohibit the reordering of memory instructions, by inserting a *fence* (also called a *barrier*) instruction between them. The use of fences was shown to be unavoidable for read/write mutual exclusion algorithms [3].

Since memory fences incur significant overhead, the number of fence instructions incurred by each passage of an algorithm (henceforth called its *fence complexity*) is a significant contributor to its time complexity, alongside the algorithm's RMR complexity.

Recent work by Attiya, Hendler and Levy [5] presented the first TSO mutual exclusion algorithm that is optimal in terms of both its RMR and fence complexities: each passage incurs a logarithmic number of RMRs and a constant number of fences. Their algorithm is not adaptive, however, and they posed the question of whether an adaptive TSO mutual exclusion algorithm with the same RMR and fence complexities exists. This is the question that we address in this work.

Our Contributions

We provide a negative answer to the question posed by [5]. In fact, we prove a stronger result: read/write mutual exclusion algorithms with constant fence complexity cannot be adaptive to total (hence also to interval- or point-) contention. This impossibility result holds regardless of the RMR complexity of the algorithm.

Our result follows from a general tradeoff that we establish between the fence complexity and the growth rate of adaptivity functions. Specifically, we prove that the fence complexity of any read/write algorithm with a linear (or sub-linear) adaptivity function is $\Omega(\log \log n)$. Our results apply for both the cache-coherent (CC) and the distributed shared-memory (DSM) models.

Following [5, 10], our tradeoff applies also to algorithms that may use *comparison* primitives, such as *compare-and-swap* (CAS), in addition to reads and writes. We show that our results also hold for obstruction-free [11] implementations of well-known objects, such as counters, stacks and queues.

Our results establish a time complexity separation between adaptive and non-adaptive implementations, thus capturing an inherent cost incurred by adaptive algorithms in the TSO model.

The rest of this paper is organized as follows. The model we use and required definitions are provided in Section 2. An overview of our proofs is presented in Section 3. The paper is concluded with a short discussion in Section 4.

2. MODEL AND DEFINITIONS

We assume the standard asynchronous shared memory model [12], in which a set of processes P communicate by applying operations to a set of shared variables V, each of which is assigned an initial value. We consider both the *cache-coherent* (CC) and the *distributed shared-memory* (DSM) computation models [2].

In the DSM model, each processor owns a segment of shared memory that can be locally accessed without traversing the processor-to-memory interconnect. Thus, every variable is permanently *local* to a single processor and *remote* to all others.[2] An access of a remote variable is an RMR.

In the CC model, all variables are remote to all processes. Each processor maintains copies of shared variables inside its private cache, whose consistency is ensured by a coherence protocol. Our results apply to both the *write-through* and *write-back* [22] CC coherence protocols. An access made by a process in a CC system is an RMR, if the corresponding cache does not contain an up-to-date copy of the variable.

Our model assumes that each variable is permanently local to *at most* a single process (and remote to all others) and thus applies to both DSM and CC systems. For variable v, we denote by $owner(v)$ the process to which v is local. We write $owner(v) = \perp$ if v is remote to all processes.

A primitive *operation* α is a read or write by some $p \in P$ issued to a variable $v \in V$. The operation α includes the value read or written. We write $\alpha = read(v)$ $(write(v))$ if α is a read (write) operation issued to variable v. Following [17], we define an *event* e as a sequence of primitive operations by some process p, such that *at most one operation is issued to a remote variable of* p. The remote variable accessed in e (if any) and the type of operation applied to it (read/write) is determined before any shared variable is accessed by e, that is, it depends only on p's internal state when e starts. We denote by $op(e) = read(v)$ $(write(v))$ the operation to the remote variable issued in e; we write $op(e) = \perp$ if there is no such operation in e. We say that e is a *read (write) event* if $op(e) = read(v)$ $(write(v))$ for some variable v. Later we extend the definition of an event by defining new types of special events.

An *execution fragment* is a (finite or infinite) sequence of events. An *execution* is an execution fragment that starts

[1] Owens, Sarkar and Sewell [20] prove Intel x86 is equivalent to Sparc TSO.

[2] For simplicity and without loss of generality, we assume that each of the processes participating in the algorithms we consider runs on a unique processor.

from the initial configuration, resulting when processes apply operations to the implemented object as they execute their algorithm. If a process has not completed its operation, it has exactly one enabled event, which is the next event it will execute, as specified by the algorithm it is using. We consider finite execution fragments, unless otherwise specified. For execution fragments E and F, the execution fragment EF denotes the concatenation of E and F. For execution E, we say that F is an *extension* of E if EF is an execution.

TOTAL STORE ORDERING (TSO)

We now present an operational model for the behavior of a shared-memory system with relaxed memory ordering, which is a simplified version of the model used by Park and Dill [21].

A set of n processes, $p_1, \ldots, p_n$, each with its own abstract *write buffer*, execute read and write memory operations in the order specified by their algorithm, called *program order*. Write operations may be delayed and executed after read operations following them in program order. This is modeled by having write operations go to the write buffer rather than directly to shared memory.

A *configuration* describes the state of a system: It contains the local state of each process, including its location in its algorithm and the contents of its write buffer. It also contains the value of each shared variable. In the *initial configuration*, all processes are in their initial state and their write buffers are empty; all shared variables hold their initial values.

In each step, a scheduling adversary picks a process and then decides whether to let it execute another event according to its algorithm or to *commit* the first write operation in its write buffer (if any). In the latter case, the write is committed by changing the value of the respective shared variable to the parameter of the write and the write *becomes visible*. The write operation is *committed* at this step.

What happens when a process p issues an event depends on the type of the event. A *fence* event e forces the adversary to commit all the writes in p's write buffer (if any) in the order they were issued. That is, whenever the adversary schedules p, it commits the next write from p's write buffer, as long as the buffer is not empty. We say that process p *completes fence* e *in execution* E if all the writes that were in p's write buffer when e was issued by p were committed in E.

Otherwise, event e is a sequence of operations. An operation is executed according to its type:

1. A write operation is placed at the write buffer. If there is already a write operation to this variable in p's write buffer, then the older write in the write buffer is replaced with the new one; otherwise, the write is placed at the end of the write buffer. The write operation is *issued* at this event.

2. A read operation returns the value of the variable, and the process changes its local state accordingly. If there is a write to this variable in the write buffer, the value is read from that write; otherwise, the value of the variable is read from shared memory. The read operation is *issued* at this event.

Let E be an execution fragment. For a set of processes Y, we denote by E^{-Y} the execution fragment obtained from E by removing all the events issued by processes in Y, and we say that the processes of Y are *erased* from E. We write $p = writer(v, E)$, and say that p is *visible* on v after E, if p is the last process to commit a write to v in E, and $writer(v, E) = \perp$ if there exists no such p. We say that an event $e \in E$ *accesses* a variable v if e is an event by process p that either commits a write to v or issues a read to v, and there is no copy of v in p's write buffer when e is executed (hence e accesses shared memory). We say that process p *accesses* variable v in E if there is an event by p in E that accesses v. We denote by $Accessed(v, E)$ the set of processes that accessed v in E. For event $e \in E$, e is a *remote event* in E if e accesses a remote variable, and a *local event* otherwise. Notice that whether an event e is considered remote depends on the execution containing the event.

We now capture the extent by which processes are aware of the participation of other processes in an execution. We do so by adapting a definition used for this purpose by [4].

DEFINITION 1. *We say that p is* aware *of q after E if either $p = q$ or if there is an event $e \in E$ by p that reads a shared memory variable v such that one of the following holds:*

1. *the last process to commit write to v before e is q;*

2. *the last process to commit write to v before e is r, and r is aware of q at the time it issued that write.*

The awareness-set *of p after E, denoted by $AW(p, E)$, is the set of processes that p is aware of after E.*

Intuitively, a process p is aware of the participation of another process q in an execution if there is (either direct or indirect) information flow from q to p in that execution via shared memory. For simplicity and without loss of generality, we assume that different write events write different values.

Mutual Exclusion Systems

Each process p has a private variable $section_p$ that represents which section in the mutual exclusion algorithm p is currently in. $section_p$ is initially ncs, indicating that p is in the non-critical section. Each process p has three special events which only p may execute, called transition events:

1. $Enter_p$ causes p to transit from its non-critical section to its entry section and sets $section_p = entry$. This event is enabled if and only if $section_p = ncs$.

2. CS_p causes p to transit from its entry section to its exit section and updates $section_p = exit$. (For notational simplicity and WLOG we assume that the execution of the critical section is instantaneous.) This event is enabled only if $section_p = entry$.

3. $Exit_p$ causes p to transit from its exit section to its non-critical section and updates $section_p = ncs$. This event is enabled only if $section_p = exit$.

For execution E and process p, we let $status(p, E)$ denote the value of $section_p$ after E. A mutual exclusion system is required to satisfy the following properties:

Exclusion For any execution E, if both CS_p and CS_q are extensions of E, then $p = q$.

Progress Given an execution E, let $X = \{q \in P | status(q, E) \neq ncs\}$. If $X = \{p\}$, then there exists a solo extension F by p such that $EFExit_p$ is an execution.

The exclusion property prevents multiple critical-section events from being simultaneously enabled. If two events CS_p and CS_q are simultaneously enabled after an execution E, then mutual exclusion may be violated. The exclusion property states that such a situation does not arise. The progress property we use was defined in [7] and is called *weak obstruction-freedom*. It is implied by deadlock-freedom and obstruction-freedom [11], although it is strictly weaker than both. In particular, it permits livelock. This weaker progress condition is sufficient for our purposes.

Next, we define the notion of a *critical event* and explain the relationship between a critical event and an RMR in different cache-coherence protocols.

DEFINITION 2. *Let $E = E_1 e E_2$ be an execution fragment, where e is an event by process p. We say that e is a* critical event *in E if one of the following holds:*

critical read: *e is a remote read of v and this is the first remote read of v by p (i.e., E_1 does not contains a remote read to v by p).*

critical write: *e is a remote write to v such that $writer(v, E_1) \neq p$ (i.e., e is the first remote write of v by p in E, or e overwrites a value written to v by another process).*

In the DSM model, each critical event accesses a remote variable, thus generating an RMR. In the CC model with a write-through protocol, writes always generate an RMR. In the CC model with a write-back protocol, if $writer(v, E_1) = q \neq p$ then v is stored in the local cache of q, thus p must invalidate or update the cached copy of v, generating an RMR. It follows that in both the write-through and write-back protocols, a critical write and a critical read that is the first access of v by p are both RMRs.

A first write followed by a first read are two critical events, but the read does not necessarily generate a cache miss. Nevertheless, since the first write is always an RMR, at least half of all critical events are RMRs. Consequently, if A is f-adaptive then each process may encounter at most $2f(k)$ critical events during a single passage, where k is total contention. We may therefore assume for simplicity that $f(k)$ bounds the number of critical events incurred by a process during a single passage.

3. PROOF OVERVIEW

Here we provide a detailed overview of our proofs. Complete proofs are provided in the full paper.

We fix an N-process f-adaptive mutual exclusion system $\mathcal{A}$. Our goal is to construct an execution in which there is a process that executes "many" fences while attempting to gain access to the critical section. The number of fences will be a function of f. We first present the definition of an *invisible-set*, a key notion in the constructing of this execution.

Given an execution E, we define two sets of processes. *Active processes*, denoted by $Act(E)$, is the set of processes that start a passage in E and are yet to complete it. Informally, an active process is a process in its entry section, trying to enter its critical section. *Finished processes*, denoted by $Fin(E)$, is the set of processes that completed a passage in E.

DEFINITION 3. *Let E be an execution and INV be a set of processes such that $INV \subseteq Act(E)$. We say that INV is an* invisible set *(IN-set), and we call a process in INV an invisible process, if the following conditions hold:*

IN1: $\forall p \in P: \quad AW(p, E) \cap INV \subseteq \{p\}$
Informally, no process is aware of any invisible process other than itself.

IN2: $\forall p \in INV: \quad status(p, E) = entry.$
informally, all invisible processes are in the entry section.

IN3: $\forall Y \subseteq INV$, *and for any $e \in E^{-Y}$: e is a critical event in E^{-Y} if and only if e is a critical event in E.*
Informally, erasing invisible processes does not affect the criticality of remaining events.

IN4: *For event $e \in E$ by process $p \in INV$, if p accesses a remote variable v in e then $owner(v) \notin Act(E)$.*
Informally, if a process p accesses a remote variable v local to some process q, then q is not an active process.

IN5: $\forall v \in V$: *If $|Accessed(v, E) \cap Act(E)| > 1$ then $writer(v, E) \notin INV$.*
Informally, if variable v has been accessed by more than a single active process, then v was not last written by an invisible process.

IN1 ensures that no process is aware of any invisible process (other than itself). This property allows us to erase from the execution we construct any invisible process, that is, to remove its events from the execution.

IN2 ensures that all invisible processes are in their entry section, trying to gain access to the critical section.

IN3 ensures that erasing invisible processes does not affect the number of critical events executed so far by processes that remain active.

IN4 ensures that no process can become aware of an invisible process by reading a variable local to it.

Let p be an invisible process that is visible on some variable v. IN5 ensures that if we need to erase p from the execution, no other invisible process becomes visible on v. Note that any subset of an IN-set is itself an IN-set.

In Section 3.1, we describe in detail the execution we construct. Before that, we present a few technical lemmas that are required for arguing about its properties.

LEMMA 1. *Let E be an execution and let $p \in P$ be a process such that $p \notin AW(q, E)$ for any $q \neq p$. Then E^{-p} is an execution.*

IN1 ensures that no process is aware of any invisible process. Hence, if E is an execution and p is an invisible process, Lemma 1 ensures us that when we erase the events of p from E we obtain a legal execution.

The following lemma considers executions obtained by erasing the events of a set of invisible processes. It establishes that processes whose events are not erased remain in the invisible set.

LEMMA 2. *Let E be an execution, $INV \subseteq P$ be an IN-set of E, and $Y \subseteq INV$. Then the following hold:*

- *E^{-Y} is an execution.*

- *Each $p \in Act(E^{-Y})$ executes the same critical events in E^{-Y} and in E.*
- *$INV \setminus Y$ is an IN-set of E^{-Y}.*

PROOF. We prove the lemma for the case $Y = \{p\}$, a singleton. The general case is easily proven by induction. From Lemma 1, E^{-p} is an execution, establishing the first property. By IN3, an event in E^{-p} is critical if and only if it is critical in E. Since each $q \in Act(E^{-p})$ executes the same events in E and in E^{-p}, it executes the same critical events in both executions, and the second property follows. We now prove that $INV \setminus \{p\}$ is an IN-set of E^{-p}:

IN1: IN1 clearly holds for p, as it does not participate in E^{-p}. Consider $q \neq p$. Since q executes the same events in E and in E^{-p}, we have $AW(q, E) = AW(q, E^{-p})$. By IN1, $AW(q, E) \cap INV \subseteq \{q\}$ and, in particular, $AW(q, E^{-p}) \cap (INV \setminus \{p\}) \subseteq \{q\}$, implying that IN1 holds for q in E^{-p}.

IN2: Every $q \in INV \setminus \{p\}$ executes the same events in E and E^{-p}, thus $status(q, E^{-p}) = status(q, E) = entry$.

IN3: Consider $Z \subseteq INV \setminus \{p\}$ and $e \in (E^{-p})^{-Z} = E^{-Z\cup\{p\}}$. Note that e is an event also in E and in E^{-p}. $Z, \{p\} \subseteq INV$ and INV is an IN-set of E. Therefore, it follows from IN3 applied to E that e is a critical event in $E^{-Z\cup\{p\}}$ if and only if e is a critical event in E. This proves IN3, since e is a critical event in E if and only if it is a critical event in E^{-p}.

IN4: Consider an event $e \in E^{-p}$ by a process q accessing a remote variable v. By IN4, $owner(v) \notin Act(E)$, hence $owner(v) \notin Act(E^{-p}) \subseteq Act(E)$.

IN5: Assume $|Accessed(v, E^{-p}) \cap Act(E^{-p})| > 1$ for some v. Since $Act(E^{-p}) \subseteq Act(E)$, we get $|Accessed(v, E) \cap Act(E)| > 1$ and, by IN5 applied to E, $writer(v, E) \notin INV$. The only events removed are by $p \in INV$, thus $writer(v, E^{-p}) = writer(v, E) \notin INV$ and, in particular, $writer(v, E^{-p}) \notin INV \setminus \{p\}$. □

Our proof constructs executions in which all active processes are in the invisible set. A useful property of invisible sets is the ability to extend an execution with non-critical events without affecting the set. Consider a process p in the invisible set of an execution E and consider any variable v. If v is local to p, IN4 ensures that no other process accessed v in E. If v is remote to p, IN5 ensure that if p already accessed v, then either p is the only active process that has accessed it, or that the last process to commit a write to v is not in the set. In any of these cases, accessing v does not cause any information flow that may violate any of the conditions IN1-IN5. This property is established by the following lemma.

LEMMA 3. *Let E be an execution and let $INV \subseteq P$ be an IN-set of E. Also, let F be an extension of E such that F contains no critical or transition events in EF. Then INV is an IN-set of EF.*

PROOF. We prove the claim for the case $F = f$, a singleton. The general case is easily proven by induction.

Let $p \in Act(E)$ be the process that executes f. Note that $Act(Ef) = Act(E)$, as f is not a transition event. If f is a fence or local event, then f does not access any remote variable and no process changes its state. By IN4, no process except p accesses any of p's local variables in E. Thus, as f does not access any remote variable, none of conditions IN1-IN5 is violated and INV is an IN-set of Ef as well.

Assume, then, that f is a remote event and let v be the remote variable accessed in f. As f is not critical in Ef, f is not the first event by p to access v, that is, p accesses v in E.

IN1: For any $p' \neq p$ we have $AW(p', Ef) = AW(p', E)$, thus IN1 holds for p' in Ef as well. Let $q = writer(v, E)$. If $q \in INV$ then, by IN5, $p = q$, otherwise both p and q access v in E, thus $writer(v, E) \notin INV$, in contradiction. Since $p = writer(v, E)$, accessing v does not change p's awareness-set.

If $q \notin INV$ then, by IN1, $AW(q, E) \cap INV = \emptyset$, thus p cannot become aware of any invisible process by accessing v. Altogether, p cannot become aware of any invisible process when executing e, that is, $AW(p, Ef) \cap INV \subseteq \{p\}$ and IN1 holds for p in Ef.

IN2: Since f is not a transition event, we have $\forall q \in INV$: $status(q, Ef) = status(q, E) = entry$.

IN3: Consider $Y \subseteq INV$. As IN3 holds for E, it is sufficient to prove that it also holds for f. Assume $f \in (Ef)^{-Y}$, implying that $p \notin Y$. In this case, p executes the same events in Ef and in $(Ef)^{-Y}$.

If f is a remote read, then f is not the first remote read by p to v in Ef and thus also not in $(Ef)^{-Y}$. Hence, f is non-critical in both.

If f commits a write to v, then $writer(v, E) = p$. It follows that $writer(v, E^{-Y}) = writer(v, E) = p$ and f is non-critical in both Ef and $(Ef)^{-Y}$.

IN4: IN4 holds in E and $Act(E) = Act(Ef)$, thus it holds for any variable $u \neq v$ in F. The only remote variable accessed in f is v. Since p accessed v in E, we get from IN4 applied to E: $owner(v) \notin Act(E) = Act(Ef)$.

IN5: IN5 holds in E, thus it holds for any variable not accessed in f. By IN4, no invisible process other than p accesses p's local variables in Ef, thus IN5 holds for p's local variables as well.

If f is a remote read of v, then $writer(v, Ef) = writer(v, E)$ and IN5 holds for v in Ef as well.

Otherwise, f commits a non-critical write to v, hence $writer(v, Ef) = writer(v, E) = p$. We have $writer(v, Ef) = writer(v, E)$ and $Accessed(v, Ef) = Accessed(v, E)$, since p accesses v in E. Since IN5 holds for v in E, IN5 holds for v in Ef as well. □

The following theorem, due to Turán [24], is required for proving the properties of our construction.

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

3.1 Execution Construction

Our construction starts with an execution H_0 where every process p executes the $Enter_p$ event only. We then inductively construct longer and longer executions. In execution H_i, exactly i processes have completed a passage through the CS and all active processes have completed exactly i fences and issued exactly l_i critical steps, for some $l_i \leq f(i)$. Our goal is to extend the execution so that as many processes as possible perform an additional fence.

The TSO model allows to delay issuing writes until a fence is performed. These writes may be preceded by reads that follow them in program order. This makes it possible to construct executions in which reads always precede writes

in-between fences. In turn, this execution structure allows us to restrict the knowledge gained by processes in-between fences and to retain a sufficiently large IN-set. Technically, the inductive construction of execution H_{i+1} from H_i is composed of a *read phase*, a *write phase*, and a *regularization phase* (see Figure 1).

We say that an execution is "regular", if all its active processes are invisible. All the executions H_i that we construct are regular. However, as we soon explain, regularity may be violated in the course of the write phase and only a weaker condition of *semi-regularity* is guaranteed to be satisfied during this phase. The write phase is followed by a regularization phase that restores the stronger condition. These notions are formalized by the following definition.

DEFINITION 4. *An execution E is* regular *if $Act(E)$ is an IN-set of E. If $Act(E)$ satisfies properties IN1-IN4 in E, we say that E is a* semi-regular *execution.*

Read phase:

In the read phase, we iteratively extend the execution by allowing active processes to preform additional critical reads. Starting with regular execution $G_0 = H_i$, we construct executions $G_1, G_2, \ldots, G_s$. We prove that executions G_k, for $k \in \{0, \ldots s\}$, satisfy the following conditions:

(1) G_k is a regular execution;

(2) Each $p \in Act(G_k)$ executes $l_i + k$ critical events in G_k.

(3) Each $p \in Act(G_k)$ completes i fences and does not yet issue its $(i+1)$'th fence event in G_k.

(4) $Fin(G_k) = Fin(H_i)$.

(5) $|Act(G_k)| \geq (|Act(G_{k-1})| - 1)/10$.

For $k > 0$, G_k is an extension of G_{k-1} in which all active processes (except for, possibly, a single process that may finish its entry section) run until they are about to execute either a fence or a critical read. In order to obtain such extension, we let each active process, in turn, run until it is about to execute its next critical or fence event. Lemma 3 ensures that the resulting execution is a regular execution.

If at least half of the active processes are about to execute a fence, then we erase the rest of the active processes. We then obtain an execution J_0 by letting each of the remaining active processes execute its fence instruction. In this case, the read phase ends and a write phase begins.

Otherwise, we consider the set of active processes that are about to execute a critical read. In order to eliminate information flow, we may need to erase a constant fraction of the active processes so that regularity is maintained. We do so by constructing a conflict graph.

The vertices of the conflict graph are the active processes. An active process p that executes a critical read may become aware of another active process q by either reading one of q's local variables (thus violating IN4) or by reading a variable last written by q (thus violating IN1). In both cases, we add an edge $\{p, q\}$ to the conflict graph. Thus, each process may introduce at most two new edges to the graph, hence the average degree of the graph is at most 4.

By Theorem 1, we can find an independent set of size at least $INV/5$. We construct G_k by erasing all the active processes except those in the independent set and then allowing the remaining processes to execute their reads (interleaved in an arbitrary order).

A key point in arguing about the properties of our construction is to bound from above the number of iterations, s, required before all remaining active processes are about to issue their next fence event.

Informally, this is done by using the following argument. Active processes are unaware of each other and may only become aware of finished processes. Consequently, the number of critical reads each of them may execute is bounded from above by a function of $i = |Fin(G_k)| = |Fin(H_i)|$. This follows from the fact that processes are allowed at most $f(i)$ critical events, since the algorithm is f-adaptive.

Therefore, if a sufficient number of processes start the read phase, eventually a large subset of processes cannot execute additional reads and must commit their writes by executing a fence. The following claim states the exact bound.

CLAIM 1. *The number of executions constructed during the read phase, s, is bounded from above by $f(i+1) - \ell_i$.*

Proof sketch: Assume towards a contradiction that the construction proceeds for k iterations, where $\ell_i + k > f(i+1)$. Erasing all the active processes from G_k except for a single process, p, yields an execution G in which at most $i+1$ processes (those in $Fin(G_k) \cup p$) participate, and in which p executes $\ell_i + k > f(i+1)$ critical events in the course of a single passage. This is a contradiction.

Write phase:

The write phase determines the order in which writes, issued by active processes since their previous fence was completed (or since they began their execution), are committed. We iteratively extend the execution by allowing active processes to preform additional critical writes.

Starting with J_0, we iteratively construct executions $J_1, J_2, \ldots, J_t$. We prove that each of the executions $J_0, \ldots, J_t$ satisfies the following conditions:

(1) J_k is a semi-regular execution, in which multiple writes by active processes to the same variable (if any) are scheduled in increasing order of process ID.

(2) Each $p \in Act(J_k)$ executes $l_i + s + k$ critical events in J_k.

(3) Each $p \in Act(J_k)$ completes i fences and does not yet complete its $(i+1)$'th fence in J_k.

(4) $Fin(J_k) = Fin(H_i)$.

(5) $|Act(J_k)| \geq \sqrt{|Act(J_{k-1})|}/4(l_i + s + k)$.

For $k > 0$, J_k is an extension of J_{k-1} in which we let each process run until it either completes its fence or it is about to perform another critical write.

If at least half of the processes complete their fence, we erase the rest of the active processes, constructing an execution L_0 in which all active processes have completed $i+1$ fences. In this case, the write phase ends and a regularization phase begins.

Otherwise, we consider the set of active processes that are about to perform another critical write. We proceed in one of two ways, according to where processes are about to write to.

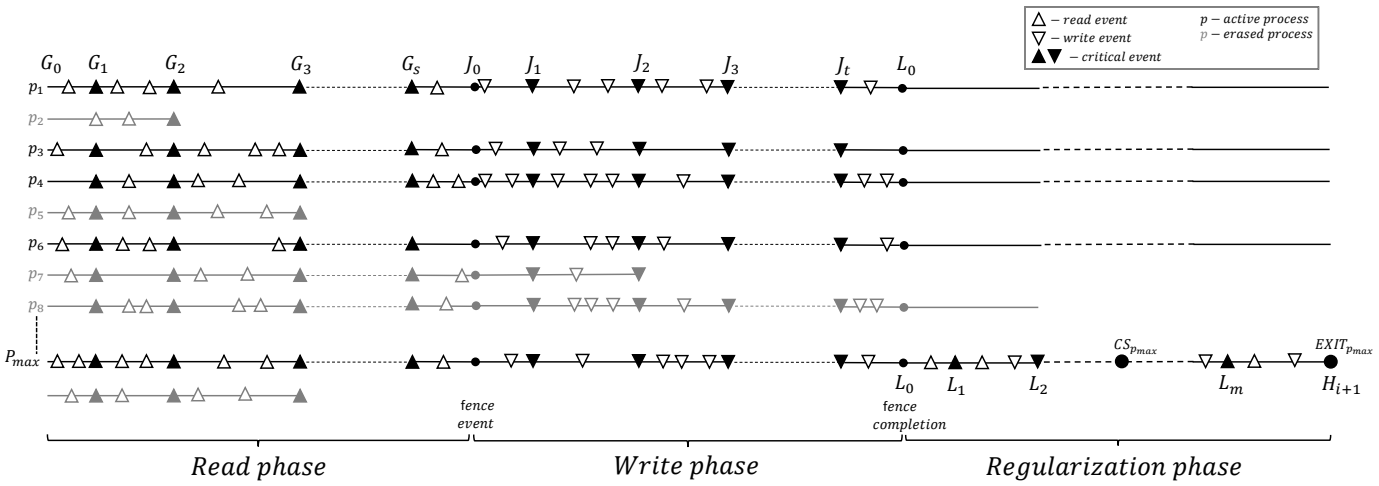

Figure 1: Structure of inductive construction. Gray-colored lines show events executed by erased processes.

In the *low-contention case*, at least $\sqrt{|Act(J_{k-1})|/2}$ of the processes are about to commit writes to different variables. In this case, we retain a single process per each such variable v and erase all other processes accessing v.

We also eliminate future information flow by erasing a fraction of the retained processes. The size of this fraction is a function of the number of critical events each process executed so far. Using the same technique as in the read phase, we build a conflict graph. Its vertices are the active processes. Note that processes do not gain new information in the course of a write phase, since they only commit writes (thus no violation of IN1 can occur).

An active processes p may conflict with another active process q if either p writes to one of q's local variables (violating IN4), or if p writes to a variable that has been accessed by q (violating IN5). In any of these cases, we add an edge $\{p, q\}$ to the conflict graph.

Each process accesses at most $\ell_i+s+k-1$ different remote variables in the course of J_{k-1}, thus each process presents at most ℓ_i+s+k new edges to the conflict graph. Consequently, the average degree of the graph is bounded from above by $2(\ell_i + s + k)$.

From Theorem 1, there exists an independent set of size at least $\sqrt{|Act(J_{k-1})|/2}/\big(2(\ell_i+s+k)+1\big)$. We obtain execution J_k by erasing all the active processes except for those in the independent set and by then allowing remaining processes to execute their critical writes (in an arbitrary order).

In the *high-contention case*, there is a variable v such that at least $\sqrt{|Act(J_{k-1})|/2}$ processes are about to commit their writes to v. In this case, we construct J_k by erasing the rest of the processes and by then allowing the remaining processes to commit their writes to v in an increasing order of their IDs. The process with the highest ID is visible on v in J_k.

Our construction of write phases is similar to a construction by Kim and Anderson [17], but unlike it, we consider fence complexity in addition to RMR complexity. Moreover, in our construction unlike in [17], writes committed during the same write phase are scheduled such that the process with the highest ID is visible on *all* high-contention variables, if any; this is guaranteed by the second part of Condition (1) above and is essential for obtaining our tradeoff.

Technically, this implies that some intermediate executions constructed during the write phase are allowed to be semi-regular but not regular: they violate invariant IN5 of Definition 3, since the last writer of high-contention variables is only allowed to finish its passage at the end of the phase. Ensuring the regularity of these intermediate executions would have required a large subset of processes to finish their passage (one per every high-contention write), which would weaken our complexity tradeoff.

Using the same argumentation as for the read phase, if a sufficient number of active processes start the phase, eventually a sufficiently large subset of these processes complete another fence and the write phase terminates.

Claim 2. *The number of executions constructed during the read and write phases is bounded from above by* $f(i+1)-\ell_i$. *In other words:* $\ell_i + s + t \leq f(i+1)$.

Regularization phase:

The regularization phase transforms the semi-regular (and possibly not regular) execution constructed by the write phase, L_0, into a regular execution. This is done by letting the active process with the largest ID, denoted p_{max}, finish its passage. Since p_{max} is visible on all the high-contention variables of the write-phase (if any), $Act(J_t) \setminus p_{max}$ is an IN-set.

Starting with L_0, we construct executions $L_1, L_2, \ldots, L_m$, H_{i+1}. We prove that each of the executions $L_0, \ldots, L_m$ satisfies the following conditions:

(1) $Act(L_k)$ can be written as $W_k \cup \{p_{max}\}$ ($p_{max} \notin W_k$).

(2) W_k is an IN-set of L_k.

(3) p_{max} executes $l_i + s + t + k$ critical events in L_k.

(4) Each $p \in W_k$ executes $l_i + s + t$ critical events in L_k.

(5) Each $p \in W_k$ completes $i+1$ fences in L_k and does not yet issue its $(i+2)$'nd fence event.

(6) $Fin(L_k) = Fin(H_i)$.

(7) $|Act(L_k)| \geq |Act(L_{k-1})| - 1$.

For $k \in \{1, \ldots, m\}$, we construct L_k from L_{k-1} by letting p_{max} run until it either terminates or until it is about to execute a critical event e. By Lemma 3, W_{k-1} is an IN-set of the resulting execution. In the latter case, in order to prevent information flow, we may need to erase the active process that owns the remote variable accessed by e or the process that is the last to have committed a write to it (there is at most a single such process).

All the active processes except p_{max} form an IN-set of the resulting execution, thus p_{max} is not aware of any other active process in executions L_k, for $0 \leq k \leq m$. Consequently, the number of critical events p_{max} may execute is a function of i, thus the number of intermediate executions constructed in the course of the regularization phase, m, is bounded from above, and eventually p_{max} finishes its passage. The resulting execution, denoted H_{i+1}, is regular, and each active process finishes $i+1$ fences in it. This completes the inductive step of our construction.

Claim 3. *The number of executions constructed during the regularization phase is bounded from above by $f(i+1)$. In other words, $m < f(i+1)$.*

Proof sketch: The proof is similar to the proof of Claim 1. We assume for contradiction that the construction can proceed for $k \geq f(i+1)$ iterations, producing execution L_k. By erasing all the active processes but p_{max} we get an execution in which at most $i+1$ processes take steps, in which p_{max} executes $\ell_i + s + t + k > f(i+1)$ critical events. This is a contradiction.

We now prove a lower bound on the size of the active processes set for the executions in our construction. As long as the lower bound guarantees that the active set is non-empty, the construction may proceed.

Theorem 2. *Let $i \in \mathbb{N}$ be such that $f(i) \leq \frac{N^{2^{-f(i)}}}{f(i)! \cdot 4^{f(i)+2i}}$. Then the following lower bound holds:*

$$|Act(H_i)| \geq \frac{N^{2^{-\ell_i}}}{\ell_i! \cdot 4^{\ell_i+2i}}$$

Proof sketch: The theorem is proved by induction on i. The base case $i = 0$ is trivial, as $|Act(H_0)| = N$. From induction hypothesis, we have:

$$|Act(H_i)| = |Act(G_0)| \geq \frac{N^{2^{-\ell_i}}}{\ell_i! \cdot 4^{\ell_i+2i}}$$

We now prove the claim for $i+1$. The induction step follows the phases of the construction. Note that we retain at least half of the processes when constructing J_0 from G_s, and L_0 from J_t. By the inequalities of condition (5) of the read and write phases construction:

$$|Act(G_k)| \geq \frac{|Act(G_{k-1})| - 1}{10}, \quad |Act(J_k)| \geq \frac{\sqrt{|Act(J_{k-1})|}}{4(\ell_i + s + k)}$$

The following lower bound is easy to derive using the above two inequalities inductively:

$$|Act(L_0)| \geq \frac{N^{2^{-(\ell_i+s+t)}}}{(\ell_i + s + t)! \cdot 4^{(\ell_i+s+t)+2i+1}}$$

Let $\ell_{i+1} = \ell_i + s + t$. Using the inequality $|Act(L_k)| \geq |Act(L_{k-1})| - 1$ inductively, and since $|Act(H_{i+1})| = |Act(L_m)| - 1$, we get:

$$|Act(H_{i+1})| \geq \frac{N^{2^{-\ell_{i+1}}}}{\ell_{i+1}! \cdot 4^{\ell_{i+1}+2i+1}} - (m+1). \tag{1}$$

From Claim 3 and from our assumption: $m + 1 \leq f(i+1) \leq \frac{N^{2^{-f(i+1)}}}{f(i+1)! \cdot 4^{f(i+1)+2(i+1)}}$. By Claim 2, $\ell_{i+1} \leq f(i+1)$, thus we may replace $f(i+1)$ with ℓ_{i+1} to get:

$$m + 1 \leq \frac{1}{4} \cdot \frac{N^{2^{-\ell_{i+1}}}}{\ell_{i+1}! \cdot 4^{\ell_{i+1}+2i+1}}. \tag{2}$$

Using Inequalities 1-2, we obtain the required result:

$$|Act(H_{i+1})| \geq \frac{N^{2^{-\ell_{i+1}}}}{\ell_{i+1}! \cdot 4^{\ell_{i+1}+2(i+1)}}.$$

In the appendix, we show that a weak obstruction-free mutual exclusion lock can be easily implemented from a weak obstruction-free implementation of either one of the following objects: counter, stack, or queue. Moreover, the implementation is such that any passage through the CS invokes a single operation on the respective object ($fetch\&increment$, *dequeue*, or *pop*) and has the same asymptotic RMR and fence complexities. We get:

Theorem 3. *Let $\mathcal{A}$ be an N-process weak obstruction-free f-adaptive implementation of a mutual-exclusion lock, counter, stack or queue, and let $i \in \mathbb{N}$ be such that $f(i) \leq \frac{N^{2^{-f(i)}}}{f(i)! \cdot 4^{f(i)+2i}}$. Then there exists an execution H with total contention $i+1$ and a process p such that p executes i fences in H during a single passage of its CS (respectively, in the course of performing a $fetch\&increment$, dequeue, or pop operation).*

Theorem 1 follows from Theorem 2, as under the conditions specified by the theorem the construction can proceed for i steps, yielding an execution H_i such that $Act(H_i) \geq 1$. Therefore there is an active process p after H_i, and from the properties of H_i, p is in a middle of a passage in H_i in which it completed i fences.

Moreover, by erasing all active processes but p from H_i we obtain an execution H in which p executes i fences in the course of a single passage, and the total contention of H is $i+1$, that is the number of fences p executes is linear in the total contention of the execution.

Corollary 1. *There exists no weak obstruction-free implementation of an adaptive mutual exclusion lock, counter, stack or queue with $O(1)$ fence complexity.*

Proof. Assume towards a contradiction that there exists such an f-adaptive algorithm $\mathcal{A}$, for some function f, such that no process executes c or more fences during a single passage/operation, for some constant c. We choose large enough N such that $f(c) \leq \frac{N^{2^{-f(c)}}}{f(c)! \cdot 4^{f(c)+2c}}$. By Theorem 3, there exists an execution H and a process p such that p executes c fences in H during a single passage/operation, contradicting our assumption. □

Kim and Anderson prove that a sub-linear adaptivity function is impossible [17]. We now present a lower bound on the fence complexity of the family of algorithms whose adaptivity function is linear.

COROLLARY 2. *Let $\mathcal{A}$ be an N-process f-adaptive implementation of a mutual-exclusion lock, counter, stack or queue, such that f is a linear function, that is $f(i) = c \cdot i$ for some constant c. Then the fence complexity of $\mathcal{A}$ is $\Omega(\log \log N)$.*

PROOF. By Theorem 3, it suffices to prove that, for $i = \Omega(\log \log N)$, the inequality $f(i) \leq \frac{N^{2^{-f(i)}}}{f(i)! \cdot 4^{f(i)+2i}}$ holds, thus there exists an execution E and a process p that executes $i = \Omega(\log \log N)$ fences during a single passage/operation in E.

$$\begin{aligned} c \cdot i \cdot (c \cdot i)! \cdot 4^{c \cdot i + 2i} &\leq N^{2^{-c \cdot i}} \\ \log(c \cdot i \cdot (c \cdot i)! \cdot 4^{c \cdot i + 2i}) &\leq 2^{-c \cdot i} \cdot \log N \\ \log\log(c \cdot i \cdot (c \cdot i)! \cdot 4^{c \cdot i + 2i}) &\leq -c \cdot i + \log\log N \\ \log\log(c \cdot i \cdot (c \cdot i)! \cdot 4^{c \cdot i + 2i}) + c \cdot i &\leq \log\log N. \end{aligned}$$

$\Rightarrow \log\log(c{\cdot}i{\cdot}(c{\cdot}i)!{\cdot}4^{c \cdot i + 2i}) + c{\cdot}i \leq \log\log((c{\cdot}i)^{2 \cdot c \cdot i}) + c{\cdot}i = \log(2 \cdot c \cdot i) + \log\log(c \cdot i) + c \cdot i \leq 3 \cdot c \cdot i$. It follows that the inequality holds for $i = \frac{1}{3c} \log\log N = \Omega(\log\log N)$ and the claim follows. □

The following corollary can be proven in a similar manner.

COROLLARY 3. *Let $\mathcal{A}$ be an N-process f-adaptive implementation of a mutual-exclusion lock, counter, stack or queue, such that f is an exponential function, that is $f(i) = 2^{c \cdot i}$ for some constant c. Then the fence complexity of $\mathcal{A}$ is $\Omega(\log\log\log N)$.*

4. DISCUSSION

We establish a time complexity separation between adaptive and non-adaptive implementations of mutual-exclusion locks, counters, stacks and queues, thus capturing an inherent cost incurred by adaptive algorithms in the TSO model.

This separation follows from a tradeoff that we prove between fence complexity and the growth rate of adaptivity functions. Specifically, we prove that the fence complexity of any read/write n-process algorithm with a linear (or sub-linear) adaptivity function is $\Omega(\log \log n)$. Our results apply for both the cache-coherent (CC) and the distributed shared-memory (DSM) models.

A corollary of our tradeoff is that constant fence-complexity adaptive implementations for these objects do not exist. Moreover, the impossibility result holds regardless of the RMR complexity of the algorithm. Following [5, 10], our tradeoff applies also to algorithms that may use comparison primitives, such as *compare-and-swap* (CAS), in addition to reads and writes.

Kim and Anderson presented an adaptive mutual exclusion algorithm whose RMR complexity is $O\big(min(k, \log n)\big)$, where k is point contention [16], hence it is f-adaptive for a linear f. The fence complexity of their algorithm is logarithmic. However, our tradeoff only implies $\log \log n$ fence complexity (see Corollary 2). Finding the tight tradeoff between fence complexity and the adaptivity function growth rate is an interesting research direction.

The memory model considered by this work is TSO. We remind the reader that TSO ensures that writes are not reordered, but it is possible to perform a read from address a before a write to address $b \neq a$ that is earlier in program order is performed. The *partial store ordering* (PSO) model, supported by older SPARC, is weaker than TSO, as it also allows the reordering of writes to different locations.

Recent work by Attiya, Hendler and Woelfel [6] showed that one cannot win on both the fence and RMR complexities of read/write PSO algorithms for many fundamental objects, including locks, counters and queues. They proved the following lower bound: let f and r respectively denote the numbers of fences and RMRs performed in an operation on such an object, then

$$f \cdot \log \frac{r}{f} + 1 \in \Omega(\log n). \tag{3}$$

They also showed that the bound is tight.

Attiya et al. [5] presented a TSO read/write mutual exclusion algorithm where each passage incurs a logarithmic number of RMRs and a constant number of fences. Inequality 3 establishes a complexity separation between the TSO and PSO models, since it follows from it that no such algorithm exists for the PSO model. Another interesting research direction is to find a tight tradeoff between the RMR-complexity and fence-complexity of adaptive PSO algorithms.

5. REFERENCES

[1] S. V. Adve and K. Gharachorloo. Shared memory consistency models: A tutorial. *computer*, 29(12):66–76, 1996.

[2] J. H. Anderson, Y. Kim, and T. Herman. Shared-memory mutual exclusion: major research trends since 1986. *Distributed Computing*, 16(2-3):75–110, 2003.

[3] H. Attiya, R. Guerraoui, D. Hendler, P. Kuznetsov, M. M. Michael, and M. T. Vechev. Laws of order: expensive synchronization in concurrent algorithms cannot be eliminated. In *POPL*, pages 487–498, 2011.

[4] H. Attiya and D. Hendler. Time and space lower bounds for implementations using -cas. In *In DISC*, pages 169–183, 2005.

[5] H. Attiya, D. Hendler, and S. Levy. An o(1)-barriers optimal rmrs mutual exclusion algorithm: extended abstract. In *ACM Symposium on Principles of Distributed Computing, PODC '13, Montreal, QC, Canada, July 22-24, 2013*, pages 220–229, 2013.

[6] H. Attiya, D. Hendler, and P. Woelfel. *The price of being adaptive. To appear in ACM Symposium on Principles of Distributed Computing, PODC, 2015.*

[7] H. Attiya, D. Hendler, and P. Woelfel. Tight RMR lower bounds for mutual exclusion and other problems. In *STOC*, pages 217–226, 2008.

[8] R. Danek and W. M. Golab. Closing the complexity gap between FCFS mutual exclusion and mutual exclusion. In *DISC*, pages 93–108, 2008.

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

[10] W. M. Golab, V. Hadzilacos, D. Hendler, and P. Woelfel. Rmr-efficient implementations of comparison primitives using read and write operations. *Distributed Computing*, 25(2):109–162, 2012.

[11] M. Herlihy, V. Luchangco, and M. Moir. Obstruction-free synchronization: Double-ended queues as an example. In *Proceedings of the 23rd IEEE International Conference on Distributed Computing Systems (ICDCS'03)*, pages 522–529, 2003.

[12] M. Herlihy and J. M. Wing. Linearizability: a correctness condition for concurrent objects. *ACM Trans. Prog. Lang. Syst.*, 12(3):463–492, June 1990.

[13] Intel Corporation. *Intel® 64 and IA-32 Architectures Software Developer's Manual*. December 2009.

[14] P. Jayanti. Adaptive and efficient abortable mutual exclusion. In *PODC*, pages 295–304, New York, NY, USA, 2003. ACM Press.

[15] P. Jayanti, S. Petrovic, and N. Narula. Read/write based fast-path transformation for FCFS mutual exclusion. In *SOFSEM*, pages 209–218, 2005.

[16] Y. Kim and J. H. Anderson. Adaptive mutual exclusion with local spinning. *Distributed Computing*, 19(3):197–236, 2007.

[17] Y. Kim and J. H. Anderson. A time complexity lower bound for adaptive mutual exclusion. *Distributed Computing*, 24(6):271–297, 2012.

[18] L. Lamport. How to make a correct multiprocess program execute correctly on a multiprocessor. *IEEE Trans. Computers*, 46(7):779–782, 1997.

[19] D. L.Weaver and T. Germond. *The SPARC Architecture Manual*. Prentice Hall, 1994.

[20] S. Owens, S. Sarkar, and P. Sewell. A better x86 memory model: x86-tso. In *Theorem Proving in Higher Order Logics, 22nd International Conference, TPHOLs 2009, Munich, Germany, August 17-20, 2009. Proceedings*, pages 391–407, 2009.

[21] S. Park and D. L. Dill. An executable specification and verifier for relaxed memory order. *IEEE Trans. Computers*, 48(2):227–235, 1999.

[22] D. A. Patterson and J. L. Hennessy. *Computer Organization & Design: The Hardware/Software Interface*. Morgan Kaufmann, 1994.

[23] G. Taubenfeld. *Synchronization Algorithms and Concurrent Programming*. Prentice Hall, 2006.

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

[25] J.-H. Yang and J. Anderson. A fast, scalable mutual exclusion algorithm. *Distributed Computing*, 9(1):51–60, 1995.

APPENDIX

A. ADDITIONAL OBJECTS

A *one-time* mutual exclusion algorithm is a ME algorithm in which each process completes at most a single passage. Since our proofs consider executions in which each process is allowed to complete a passage at most once, our results apply to one-time mutual exclusion algorithms.

LEMMA 4. *Let $\mathcal{C}$ be an object of one of the following types: counter, stack or queue. Then for any $N \in \mathbb{N}$, there exists an N-process one-time mutual exclusion algorithm $\mathcal{A}$ using $\mathcal{C}$ (and read/write variables), such that each passage through the CS invokes a single operation Op on $\mathcal{C}$ ($fetch\&increment$, dequeue, or pop respectively), and has the same RMR and fence complexities as those of Op, up to a constant additive factor, in both the DSM and CC models.*

PROOF. We first prove the lemma for counter, by presenting Algorithm 1 for an N-process one-time mutual exclusion.

Algorithm 1 One-time mutual exclusion

Shared Data: $release[N]$: initially $[1, 0, \cdots, 0]$.
$waiting[N]$: initially $[\perp, \perp, \cdots, \perp]$.
$spin[N]$: initially $[0, 0, \cdots, 0]$.
$\mathcal{C}$: a counter, initially 0.

program for process p:

```
1: v ← C.fetch&increment();
2: waiting[v] ← p;
3: if release[v] = 0 then
4:     await (spin[p] = 1)
   CS
5: release[v + 1] ← 1;
6: q ← waiting[v + 1];
7: if q ≠⊥ then
8:     spin[q] ← 1;
```

The correctness of the algorithm follows easily from the properties of the counter object.

We assume that each write in Algorithm 1 (lines 2-8 only) is followed by a fence instruction, and omit these fences from the code for presentation simplicity. Consequently, the fence complexity of Algorithm 1 is the same as that of the $fetch\&increment$ operation, up to a constant additive factor.

In the DSM model, variable $spin[p]$ resides in the local memory segment of process p. Note p busy-waits only when it waits for $spin[p]$ to be set. In the CC model (with either write-back or write-through), since once $spin[p]$ is set its value does not change, p may encounter at most 2 RMRs during the wait in line 4. Therefore, under both models, the ME algorithm has the same RMR complexity as of the $fetch\&increment$ operation, up to a constant additive factor.

It is easy to see that $\mathcal{C}$'s value never exceeds N in the course of executing Algorithm 1. It follows that $\mathcal{C}$ can be implemented using a single queue or stack in the following manner:

Queue: initialize $Q = \langle 0; 1; \cdots; N \rangle$. The $fetch\&increment$ operation is simply invoking $Q.dequeue()$.

Stack: initialize $S = \langle N; \cdots; 1; 0 \rangle$. The $fetch\&increment$ operation is simply invoking $S.pop()$.

Using Algorithm 1 with any of these implementations yields the required result. □