Noname manuscript No.
(will be inserted by the editor)

The Price of being Adaptive

Ohad Ben-Baruch · Danny Hendler

the date of receipt and acceptance should be inserted later

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 $O(f(k))$. Adaptive implementations are desirable when contention is much smaller than the total number of processes, n, sharing the implementation.

Recent work [6] 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 [6] is non-adaptive, however, and the authors 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.

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

An extended abstract of this paper appeared at PODC [8]. Partially supported by the Israel Science Foundation (grant 1749/14) and by the Lynne and William Frankel Center for Computing Science at Ben-Gurion University.

Ohad Ben-Baruch
Department of Computer-Science, Ben-Gurion University
Tel.: +972(0)524261187
E-mail: bbohad@gmail.com

Danny Hendler
Department of Computer-Science, Ben-Gurion University
Tel.: +972(0)864280387
E-mail: hendlerd@cs.bgu.ac.il

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 [12], the mutual exclusion problem is a fundamental Distributed Computing problem and is still the focus of intense research [2, 27].

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. [11, 18, 19, 21, 28]). 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 was shown to be optimal [4, 13].

A mutual exclusion algorithm A is adaptive, if its RMR complexity is a function of the number of active processes. More formally, an algorithm is *f-adaptive to total contention* (henceforth simply *adaptive*), if the RMR complexity of every passage is $O(f(k))$, 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 $O(f(k))$, 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 superior to non-adaptive ones when the number of active processes is typically 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 [16], or at least sequentially consistent [22]. 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 [23] 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 [23] and x86 [10]. [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 [5].

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 [6] 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 [6]. 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 [6, 15], 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 [17] 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 article is organized as follows. The model we use and required definitions are provided in Section 2. An overview of our proofs and results is presented in Section 3. Full and detailed proofs are presented in Section 4. Section 5 discusses additional objects such as stacks and queues. The paper is concluded with a short discussion in Section 6.

[1] Owens, Sarkar and Sewell [24] prove Intel x86 is equivalent to Sparc TSO.

2 Model and Definitions

We assume the standard asynchronous shared memory model [16], 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 a *remote memory reference* (RMR).

In the CC model, 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* [26] CC coherence protocols. Quoting from [14]: "In a write-through protocol, to read a variable v a process p must have a (valid) cached copy of v. If it does, p reads that copy without causing an RMR; otherwise, p causes an RMR that creates a cached copy of v. To write v, p causes an RMR that invalidates (i.e., effectively deletes) all other cached copies of v and writes v to main memory. In a write-back protocol, each cached copy is held in either shared or exclusive mode. To read a variable v, a process p must hold a cached copy of v in either mode. If it does, p reads that copy without causing an RMR. Otherwise, p causes an RMR that: (a) eliminates any copy of v held in exclusive mode, typically by downgrading the status to shared and, if the exclusive copy was modied, writing v back to memory; and (b) creates a cached copy of v held in shared mode. To write v, p must have a cached copy of v held in exclusive mode. If it does, p writes that copy without causing RMRs. Otherwise, p causes an RMR that: (a) invalidates all other cached copies of v and writes any modied copy held in exclusive mode back to memory; and (b) creates a cached copy of v held in exclusive mode."

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, which is always the case in the CC model. Notice that accessing a remote variable does not necessary generates an RMR (depends on the model), but simply implies that the variable is not part of the process's private segment (if there is such).

An *event* e is a read or write operation by some $p \in P$ issued to a variable $v \in V$. The event e includes the value read or written. We write $e = read(v)$ ($write(v)$) if e is a read (write) operation issued to variable v. Later we extend the definition of an event by defining new types of special events, that are used for modelling the mutual exclusion problem in the TSO model.

An *execution fragment* is a (finite or infinite) sequence of events. We use $\langle\rangle$ to denote the empty execution fragment. An *execution* is an execution fragment that starts 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. Let E and F be two execution fragments. The execution fragment EF denotes the concatenation of E and F. If E and EF are executions, we say that F is an extension of E. We say that F is a *sub-execution* of E, and write $F \preceq E$, if F is a (possibly non-contiguous) subsequence of E's events. 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 say that the processes of Y are *erased* from E. We denote by $E \mid Y$ the execution fragment obtained from E by removing all the events issued by processes not in Y (i.e., only the events issued by processes in Y are retained). When $Y = \{p\}$, we use the notation E^{-p} and $E \mid p$.

Fact 1.

1. $(E_1E_2)^{-Y} = E_1^{-Y}E_2^{-Y}$
2. $(E^{-Y})^{-Z} = E^{-Y \cup Z}$

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 [25].

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

[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.

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 (we say that the write *becomes visible*) and removing the write operation from the buffer. We say that the write operation is *committed* at this step and the execution is extended by a *write commit event*.

What happens when a process p issues an event depends on the type of the event:

1. 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.
2. A write operation is placed at the end of the write buffer. The write operation is *issued* at this event but is not yet made visible to other processes. It will only be made visible once the execution is extended by a corresponding commit event.
3. 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 the last such write; otherwise, if there is a (valid) cached copy of the variable in the process's private cache, the value is read from that copy; otherwise, the value of the variable is read from shared memory. The read operation is *issued* at this event.

For simplicity, we split the fence instruction into two successive *fence events*: a $BeginFence$ event, immediately followed (in program order) by an $EndFence$ event. $BeginFence$ initiates the execution of a fence as described above. $EndFence$ signifies that the fence execution has finished, that is, the write buffer of the process that performed the fence is now empty. For execution E and process p, we say that p *is executing a fence after* E, if the last fence event by p in E is $BeginFence$. Note that if p is executing a fence after E, then the only event p is allowed to execute is the next write in its write buffer, or $EndFence$ if the buffer is empty. Hence, if p is executing a fence after E, we write $mode(p, E) = write$, otherwise we write $mode(p, E) = read$. We say that p *completed* i fences in E if p executed i $EndFence$ events in E, that is p executed to completion i fences in E.

In our construction, we only consider executions in which, whenever the scheduler picks a process p for the next step, it will always let it execute another event rather than commit a write from its write buffer, as long as p is in between fences (i.e., not executing a fence). That is, the scheduler delays committing writes from the write buffer as long as possible. Hence, a process' mode indicates whether it is executing a fence (if the process is in write mode), in which case it may only commit writes from its write buffer, or it is in between fences (if the process is in read mode), in which case all its writes are delayed and the only shared memory operations performed on its behalf are reads.

Let E be an execution fragment. 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. We write $writer(v, E) = \perp$ if there exists no such p. We say that an event $e \in E$ by process p *accesses* a variable v if either 1) e commits a write to v, or 2) e is a read event to v that is performed when p's write-buffer does not contain a copy of v. Thus, events that issue writes to the write-buffer or read from the write-buffer are not considered variable accesses. 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.

A (read or write) event $e \in E$, executed by process p, is a *remote event* in E if it accesses a variable that is remote w.r.t. p, otherwise it is a *local event*. Notice that a remote event is not necessarily an RMR, as it might be that p has a valid copy of v in its cache. However, such an event has the potential of generating an RMR, and as such the proof will focus on such events. Whether or not an event is a remote access is determined based on the history of the process executing e, as stated below.

Fact 2. *Let E and F be two execution fragments and let p be a process such that $E \mid p = F \mid p$. Then for any event $e \in E$ by p, e is a remote event in E if and only if e is a remote event in F.*

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 [3].

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 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. Notice that the awareness-set of a process can only be extended along an execution. Moreover, it follows from Definition 1 that whenever a process p reads a variable v last written by some process q, all the processes that belonged to q's awareness set when it issued this write to v are added to p's awareness set.

Mutual Exclusion Systems

Each process p has a private variable $section_p$ that signifies 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. There are three *transition events* which each process p may execute:

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 [4] and is called *weak obstruction-freedom*. It is implied by deadlock-freedom and obstruction-freedom [17], although it is strictly weaker than both. In particular, it permits livelock. This weaker progress condition is sufficient for our proofs.

Next, we define the notion of a *critical event* and explain the relation 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 contain a remote read of v by p).

critical write: e commits a remote write to v and $writer(v, E_1) \neq p$ (i.e., e is the first remote write commit to v by p in E, or e overwrites a value committed to v by another process).

In the DSM model, each critical event generates an RMR since it accesses a remote variable. In the CC model with a write-through coherence protocol, write commits always generate an RMR. In the CC model with a write-back protocol, if $writer(v, E_1) = q \neq p$ then a copy of 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 therefore assume in the following for simplicity that $f(k)$ bounds the number of critical events incurred by a process during a single passage.

Observe that whether an event is considered critical depends on the particular execution that contains the event, and specifically on the process that executes the event and the prefix of the execution preceding the event. Consequently, when saying that an event is (or is not) critical , the execution containing the event must be specified.

We now define the notion of a *special* event. This is an extension to the notion of a critical event, used for capturing events of importance for our construction.

Definition 3 Let E be an execution such that E can be written as $E_1 e E_2$. We say that e is a *special event* in E if one of the following holds:

Critical event: e is a critical event in E.

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

Fence event: e is one of BeginFence or EndFence.

We say that two events e and f are *congruent*, and write $e \sim f$, if e and f are executed by the same process and either $e = f$ or both apply the same operation to the same variable. Informally, two events are congruent if they either execute the same transition or fence event, or if both are reads or both are writes of the same variable v (although the values they both read or write may differ).

3 Proof Overview

We now provide a detailed overview of our proofs. This is then followed by the full proofs.

We fix an 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 4 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, 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 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.

Our proofs mostly consider "regular" executions. Informally, these are executions where all active processes are invisible and in their entry sections. As we soon explain, sometimes we relax this requirement and permit "semi-regular" executions.

Definition 5 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.

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 H_i, for $i > 0$. In execution H_i, exactly i processes complete a passage through the CS and all active processes complete exactly i fences and issue exactly l_i critical events, 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 the execution of writes until a fence is performed, and 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).

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 $G_1, G_2, \ldots, G_s$.

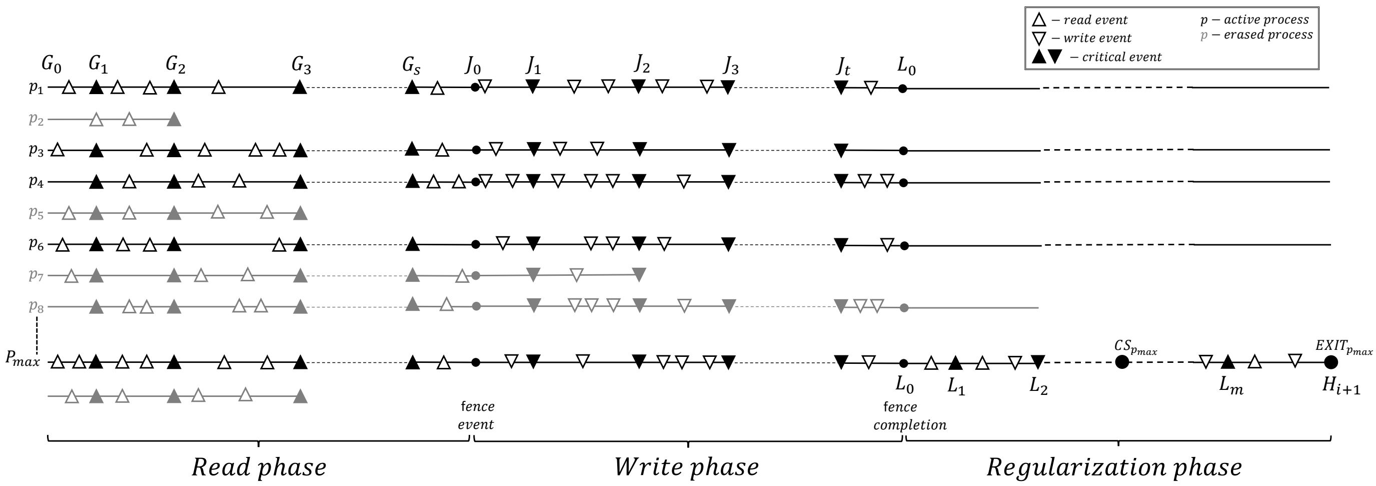

Fig. 1: Structure of inductive construction. Gray-colored lines show events executed by erased processes.

For $k > 0$, G_k is an extension of G_{k-1} in which all active processes run until they are about to execute a critical read (Lemma 5 establishes that such an extension exists) and then these reads are interleaved. Each such extension may require erasing a constant fraction of the active processes in order to eliminate information flow, such that the resulting execution is regular (see Claim 4.1.1). We prove that executions G_k, for $k \in \{0, \dots s\}$, satisfy the following conditions (see Lemma 6):

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

A key point in the proofs 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 (since they belong to an IN-set) 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 $i = |Fin(G_k)| = |Fin(H_i)|$ (an explicit bound is given in Claim 4.1). 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 resulting execution is denoted by J_0, whereupon the read phase ends and a write phase beings.

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 if this is the first fence), are committed. We iteratively extend the execution by allowing active processes to preform additional critical writes.

Starting with J_0, we iteratively construct executions $J_1, J_2, \dots, J_t$. We prove that each of the executions $J_0, \dots, J_t$ satisfies the following conditions (see Lemma **??**):

(1) J_k is a semi-regular execution, in which multiple writes by active processes to the same variable (if any) are ordered 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 did 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 is about to perform another critical write. We consider two cases according to where processes are about to write to. In the *low-contention case*, at least a square root fraction of the active processes are about to commit writes to different variables. In this case we retain a single process per such variable v (erasing all other processes accessing v) and 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. In the *high-contention case*, there is a variable v such that at least a square root fraction of the processes are about to commit their write to v. In this case we erase the rest of the processes and then allow these processes to commit their writes to v in an increasing order of their IDs.

Our construction of write phases is similar to a construction by Kim and Anderson [21], but unlike it, we consider fence complexity in addition to RMR complexity. Moreover, in our construction unlike in [21], writes committed during the same write phase are scheduled such that the process with the highest ID is visible on *all* of the phase' 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 requires that some intermediate executions constructed during the write phase are allowed to be semi-regular but not regular: they violate invariant IN5 of Definition 5, 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.

Processes do not gain new information in the course of a write phase, since they only commit writes. 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 (resulting in execution L_0) and the write phase terminates (an explicit bound is given in Claim 4.2).

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), $Act(L_0) \setminus p_{max}$ 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 (see Lemma 8):

(1) $Act(L_k)$ can be written as $W_k \cup \{p_{max}\}$ (where $p_{max} \notin W_k$);
(2) W_k is an IN-set of L_k;
(3) p_{max} executed $l_i + s + t + k$ critical events in L_k;
(4) Each $p \in W_k$ executed $l_i + s + t$ critical events in L_k;
(5) Each $p \in W_k$ completed $i+1$ fences in L_k and did 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. 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 that is the last to have written to it (by Claim 4.3.3, there is at most a single such process).

All of the active processes but p_{max} form an IN-set of the resulting execution (Claim 4.3.4), 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 (an explicit bound is given in Claim 4.3). The resulting execution, denoted H_{i+1}, is regular, and each active process finished $i+1$ fences. This completes the inductive step of our construction. We present the full proofs in Section 4.

3.1 Results

In Section 4, we prove the following theorem:

Theorem 1 *Let $\mathcal{A}$ be an N-process weak obstruction-free f-adaptive implementation of a mutual-exclusion lock and let $i \in \mathbb{N}$ be such that $f(i) \leq \dfrac{N^{2^{-f(i)}}}{f(i)! \cdot 4^{f(i)+2i}}$. Then there exists an execution H whose total contention is $i+1$ and a process p such that p executes i fences in H during a single passage of its CS.*

We then show (in Section 5) that a weak obstruction-free mutual exclusion lock can be easily implemented from a weak obstruction-free implementation of a counter, a stack or a 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 (see Lemma 9). It follows that Theorem 1 holds for stacks and queues as well.

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 $f(c) \leq \dfrac{N^{2^{-f(c)}}}{f(c)! \cdot 4^{f(c)+2c}}$.

By Theorem 1, 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. $\square$

Kim and Anderson prove that a sub-linear adaptivity function is impossible [21]. 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 1, it suffices to prove that for $i = \Omega(\log\log N)$ the inequality $f(i) \leq \dfrac{N^{2^{-f(i)}}}{f(i)!\cdot 4^{f(i)+2i}}$ 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.

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

The right-hand side of the above inequality can be bounded from above as follows:

$$\begin{aligned} &\log\log(c\cdot i\cdot(c\cdot i)!\cdot 4^{c\cdot i+2i})+c\cdot i \leq\\ &\quad\log\log((c\cdot i)^{2\cdot c\cdot i})+c\cdot i =\\ &\qquad\log(2\cdot c\cdot i)+\log\log(c\cdot i)+c\cdot i\leq 3\cdot c\cdot i. \end{aligned}$$

It follows that the inequality holds for $i = \frac{1}{3c}\log\log N = \Omega(\log\log N)$ and the claim follows. $\square$

A similar computation is used to prove the following lower bound.

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 $\Omega(\log\log\log N)$.*

Proof. By Theorem 1, it suffices to prove that, for some $i = \Omega(\log\log\log N)$, the inequality $f(i) \leq \dfrac{N^{2^{-f(i)}}}{f(i)!\cdot 4^{f(i)+2i}}$ holds, thus there exists an execution E and a process p such that p executes $i = \Omega(\log\log\log N)$ fences during a single passage/operation in E.

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

The right-hand side of the above inequality can be bounded from above as follows:

$$\begin{aligned} &\log\log(2^{c\cdot i}\cdot 2^{c\cdot i}!\cdot 4^{2^{c\cdot i}+2i})+2^{c\cdot i}\leq\\ &\quad\log\log((2^{c\cdot i})^{2\cdot 2^{c\cdot i}})+2^{c\cdot i} =\\ &\qquad c\cdot i+1+\log(c\cdot i)+2^{c\cdot i}\leq 2^{c\cdot i+1}. \end{aligned}$$

It follows that the inequality holds for $i = \frac{1}{c}(\log\log\log N - 1) = \Omega(\log\log\log N)$ and the claim follows. $\square$

4 Full Lower Bound Proofs

We start by stating a few lemmas and claims that are required for arguing about the properties of our construction, which we specify in a formal manner later. The proofs are technical and appear in the appendix.

Claim 1. *Let E be an execution fragment and $e \in E$ be an event issued by some process p.*

- *Assume e is a non-special event in E. Then for any execution fragment $F \preceq E$ such that $F \mid p = E \mid p$, e is a non-special event in F.*
- *Assume e is a special event in E. Then for any execution fragment F such that $E \preceq F$ and $F \mid p = E \mid p$, e is a special event in F.*

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.*

Lemma 2 *Let E be an execution and let INV be an IN-set of E. Let e be a $read(v)$ or $write(v)$ event. Assume $writer(v, E) \notin INV$ and $owner(v) \notin Act(E)$. Then INV satisfies IN1-IN4 of Definition 4 in Ee.*

Claim Let E be an execution and let INV be an IN-set of E. Let e be an extension of E by some process p such that e is a local event in Ee. Then INV is an IN-set of Ee.

Lemma 3 *Let E be an execution and let INV be an IN-set of E. Let F be an extension of E such that F contains no critical or transition event in EF. Then INV is an IN-set of EF.*

Lemma 4 *Let E be an execution, INV be an IN-set of E and $Y \subseteq INV$.*
Define $E' = E^{-Y}$. Then the following hold:

1. *E' is an execution;*
2. *$Act(E') = Act(E) \setminus Y$ and $Fin(E') = Fin(E)$;*
3. *$INV \setminus Y$ is an IN-set of E';*
4. *Each $p \in Act(E')$ executes the same critical events in E' and in E;*
5. *If $p \in Act(E')$ is about to execute a special event f_p after E, then p is about to execute a special event $e_p \sim f_p$ after E'.*

Lemma 5 *Let E be a regular execution. Then there exists an extension F such that the following hold:*

- *F contains no special events in EF;*
- *EF is a regular execution;*
- *Each $p \in Act(E)$ is about to execute a special event f_p after EF. Moreover, at most one process $p \in Act(E)$ is about to execute $f_p = CS_p$ after EF.*

The proof of the following theorem appears in [9].

Theorem 2 (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.*

We now prove a tradeoff between the fence complexity and the adaptivity function f. We start with the regular execution H_0 in which each process p have executed the $Enter_p$ event only, hence $Act(H_0) = P$ and $Fin(H_0) = \emptyset$. We then build longer executions $H_1, H_2, ...$ inductively. At each induction step, we construct H_{i+1} from H_i using three phases: read, write, and regularization. Each phase consists of a sequence of executions.

Every induction step starts with an execution H_i that meets the following conditions:

(a) H_i is a regular execution;
(b) Each $p \in Act(H_i)$ executes ℓ_i critical events in H_i, for some $\ell_i \leq f(i)$;
(c) $|Fin(H_i)| = i$;
(d) Each $p \in Act(H_i)$ completes i fences in H_i and $mode(p, H_i) = read$.

For simplicity, we slightly abuse notation and write ℓ instead of ℓ_i in the rest of this section.

4.1 Read phase

In the course of the read phase, we construct a sequence of executions $H_i = G_0, G_1, G_2, \ldots, G_s, J_0$.

Lemma 6 *At each step during the read phase we have an execution G_k satisfying the following conditions:*

(1) G_k is a regular execution;
(2) Each $p \in Act(G_k)$ executes $\ell + k$ critical events in G_k;
(3) Each $p \in Act(G_k)$ completes i fences in G_k and $mode(p, G_k) = read$;
(4) $Fin(G_k) = Fin(H_i)$;
(5) $|Act(G_k)| \geq (|Act(G_{k-1})| - 1)/10$.

First notice that $G_0 = H_i$ satisfies all the conditions in Lemma 6. Assume we already constructed G_{k-1} satisfying the conditions in Lemma 6. We let $G = G_{k-1}$ and $n = |Act(G_{k-1})|$ in the rest of this section (4.1), in which we specify the construction of the read phase and prove Lemma 6.

4.1.1 Construction: stage 1

By Lemma 5, there exists an extension F of G such that the following hold:

1. F contains no special events in GF;
2. GF is a regular execution;
3. F contains no transition events, therefore $Act(GF) = Act(G)$ and $Fin(GF) = Fin(G) = Fin(H_i)$;
4. Each $p \in Act(G)$ executes $\ell + k - 1$ critical events in GF;
5. F contains no fence events, hence each $p \in Act(G)$ completes i fences in GF and $mode(p, GF) = mode(p, G) = read$;
6. Each $p \in Act(G)$ is about to execute a special event f_p after GF. Moreover, at most a single process $q \in Act(G)$ is about to execute $f_q = CS_q$.

Denote by Y the set of processes in $Act(G)$ such that $f_p \neq CS_p$. We have $n - 1 \leq |Y| \leq n$. For each $p \in Y$, since $status(p, GF) = entry$ and $f_p \neq CS_p$ we get that f_p is not a transition event, and since $mode(p, GF) = read$ we get that f_p is either a read event or a $BeginFence$ event.
We define: $Z_1 = \{p \in Y \mid f_p = BeginFence\}$, $Z_2 = \{p \in Y \mid f_p \text{ is a read event}\}$.
It follows that $Y = Z_1 \cup Z_2$ and $Z_1 \cap Z_2 = \emptyset$, thus $|Y| = |Z_1| + |Z_2|$.

Case I: $|\mathbf{Z_1}| > |\mathbf{Y}|/\mathbf{2}$

We define $W = Z_1$. We have $|W| > |Y|/2 \geq (n-1)/2$, thus $|W| \geq n/2$.

Case II: $|\mathbf{Z_2}| \geq |\mathbf{Y}|/\mathbf{2}$

We construct an undirected graph $\mathcal{G}$ as follows: the vertices of $\mathcal{G}$ are the processes in Z_2. Consider $p \in Z_2$ and denote $f_p = read(v)$. We add an edge $\{p, q\}$

if there exists $q \in Z_2$ such that v is local to q or $writer(v, GF) = q$.
Since v is local to at most one process and has at most one last writer, p accounts for at most 2 edges in $\mathcal{G}$, thus the average degree in $\mathcal{G}$ is at most 4. By Theorem 2, there exists an independent set $W \subseteq Z_2$ in $\mathcal{G}$ such that:

$$|W| \geq |Z_2|/5 \geq |Y|/10 \geq (n-1)/10$$

4.1.2 Construction: stage 2

We have a set of processes $W \subseteq Act(GF)$. Define $\overline{W} = Act(GF) \setminus W$. By Lemma 4 with $'E' \leftarrow GF$ and $'Y' \leftarrow \overline{W}$, we have an execution $N = (GF)^{-\overline{W}}$ such that the following hold:

1. $W = Act(GF) \setminus \overline{W}$ is an IN-set of N;
2. $Act(N) = Act(GF) \setminus \overline{W} = W$, and $Fin(N) = Fin(GF) = Fin(H_i)$;
3. From 1 and 2: N is a regular execution;
4. Each $p \in W$ executed the same critical events in N and in GF, thus p executed $\ell + k - 1$ critical events in N;
5. For each $p \in W$, since $N \mid p = (GF) \mid p$ we get that p completed i fences in N and $mode(p, N) = mode(p, GF) = read$;
6. Each $p \in W$ is about to execute a special event $e_p \sim f_p$ after N.

We extend N by letting each $p \in W$ execute its next event in an arbitrary order. Denote this extension by D, and define $G_k = ND$. Notice that $Act(G_k) = Act(N) = W$ and $Fin(G_k) = Fin(N) = Fin(H_i)$ since D contains no transition events.

We now analyze the resulting execution, G_k, according to the cases defined in stage 1.

Case I

Define $s = k - 1$ and $J_0 = G_k$. For each $p \in W$ we have $e_p \sim f_p = BeginFence$. The following conditions hold.

1. D contains fence events only, thus by Lemma 3, W is an IN-set of $J_0 = ND$, i.e. J_0 is a regular execution;
2. Each $p \in Act(J_0)$ executed $\ell + s$ critical events in N and thus in J_0;
3. Each $p \in Act(J)$ completed i fences in J_0, and the last event by p in J_0 is BeginFence, i.e. $mode(p, J_0) = write$;
4. $Fin(J_0) = Fin(H_i)$;
5. $|Act(J_0)| = |W| \geq |Act(G_s)|/2$;

We are done with the read phase, and we proceed to the write phase.

Case II

By the definition of W, each $p \in W$ executed a single read event e_p in D.

Claim 4.1.1. *G_k is a regular execution.*

Proof. Consider $p \in W$, and denote $e_p = read(v)$.
property 1: $writer(v, N) \notin W$.
Denote $q = writer(v, GF)$. If $q \notin Act(GF)$ then after removing the events by processes in $\overline{W} \subseteq Act(GF)$ we still have $writer(v, N) = writer(v, GF) = q$ and $q \notin W \subseteq Act(GF)$. Otherwise $q \in Act(GF)$. GF is a regular execution in which q accessed v and $writer(v, GF) \in Act(GF)$, thus by IN5 q is the only process in $Act(GF)$ to access v. Notice that $q \notin W$ since there is an edge $\{p, q\}$ in $\mathcal{G}$, and $p \in W$, an independent set of $\mathcal{G}$. Therefore $q \in \overline{W}$, and after removing events by processes in $\overline{W}$ there is no process in $Act(GF)$ (and thus in W) to access v in N, that is $writer(v, N) \notin W$.
property 2: $owner(v) \notin W$:
Denote $q_v = owner(v)$. If $q_v \notin Z_2$ the claim clearly hold. Otherwise $q_v \in Z_2$ and $f_p \sim e_p = read(v)$, thus $\mathcal{G}$ contains an edge $\{p, q_v\}$ (notice that $p \neq q_v$ since p remotely reads v). Since $p \in W$ and W is an independent set we have $q_v \notin W$.

Denote by D_j the prefix of D that contains exactly j events. We prove by induction on j $(0 \leq j \leq |D|)$ that W is an IN-set of ND_j:
induction base $j = 0$: by our construction W is an IN-set of N.
Assume we already proved the claim for $j < |D|$. Notice that $ND_{j+1} = ND_j e_p$ for some $p \in W$, and denote $e_p = read(v)$. Since D_j contains no transition events, we have $Act(ND_j) = Act(N) = W$. D_j contains only read events, thus no write to v occurs in D_j, i.e. $writer(v, ND_j) = writer(v, N) \notin W$. Together with the fact that $owner(v) \notin W$, the conditions for Lemma 2 holds, and W satisfies IN1-IN4 in $ND_j e_p = ND_{j+1}$. As IN5 holds for W in ND_j it clearly holds for any variable $u \neq v$ in ND_{j+1}. Since e_p is a read event to v we get $writer(v, ND_{j+1}) = writer(v, ND_j) \notin W$ and IN5 holds for v in ND_{j+1}.

Using the last claim with $j = |D|$ we have that W is an IN-set of $ND = G_k$, and thus G_k is a regular execution. □

We now prove that G_k satisfies all the conditions in Lemma 6:

(1) G_k is a regular execution;
(2) Consider $p \in W = Act(G_k)$. p executed $\ell + k - 1$ critical events in N and a single event e_p in D. By our construction e_p is a critical event in Ne_p, $Ne_p \preceq G_k$, and $(Ne_p) \mid p = G_k \mid p$. Therefore, by

claim 1, e_p is a critical event in G_k. Altogether p executed $\ell + k$ critical events in G_k;
(3) Consider $p \in W$. p completed i fences in N and $mode(p, N) = read$. Since D contains a single read event by p we get that p completed i fences in G_k and $mode(p, G_k) = read$;
(4) $Fin(G_k) = Fin(N) = Fin(H_i)$;
(5) $|Act(G_k)| = |W| \geq (n-1)/10$.

Claim 4.1. *The number of steps in the read phase is bounded by $f(i+1) - \ell$, that is, $\ell + s \leq f(i+1)$.*

Proof. Assume towards a contradiction that during the read phase we build an execution G_k such that $\ell + k > f(i+1)$. Then G_k satisfies:

– G_k is a regular execution;
– Each $p \in Act(G_k)$ executed $\ell + k$ critical events in G_k;
– $Fin(G_k) = Fin(H_i)$, thus $|Fin(G_k)| = i$.

We choose an arbitrary $p \in Act(G_k)$ and denote $Y = Act(G_k) \setminus \{p\}$. Using Lemma 4 with $'E' \leftarrow G_k$ and $'Y' \leftarrow Y$, we have an execution $G'_k = G_k^{-Y}$ such that: $Act(G'_k) = \{p\}$ and $Fin(G'_k) = Fin(G_k) = Fin(H_i)$; p executes the same critical events in G_k and in G'_k, thus p executes $\ell+k$ critical events in G'_k. Hence, at most $i+1$ processes issue events in G'_k, i.e. the total contention of G'_k is at most $i+1$. However, p executes $\ell + k > f(i+1)$ critical events during a single passage in G'_k, a contradiction to our assumption that the algorithm is f-adaptive. □

4.2 Write phase

The read phase construct an execution J such that the last event by each process in $Act(J)$ is BeginFence. The write phase will determine the order in which processes are executing their write commits along the fence. Each process $p \in Act(J)$ have a list α_p of writes in its write buffer, which it will commit starting from J.

We first focus on the list α_p. Notice that α_p is also a solo run of p starting from J until the point where it finish executing its fence. Along the write phase, we first construct a set of processes out of $Act(J)$, such that the lists α_p of those processes satisfies certain properties. Then, we will use this set in order to extend J. In the course of the write phase we construct a sequence of sets $Act(J) = P_0 \supseteq P_1 \supseteq \ldots \supseteq P_t$ satisfying the Lemma below.

Lemma 7 *At each step of the write phase we have a set $P_k \subseteq Act(J)$ such that each process $p \in P_k$ has a prefix β_p of α_p and the following hold:*

(1) Each $p \in P_k$ is executing exactly k critical writes along β_p in the execution $J\beta_p$.
(2) Let $e \in \beta_p$ be a critical write in $J\beta_p$ by some process $p \in P_k$ to some variable v. Then, the following holds:
  (a) $owner(v) \notin P_k$;
  (b) $writer(v, J) \notin P_k$;
  (c) Either there is no $q \in P_k$ different then p which access v in $J\beta_q$, or that any process $q \in P_k$ have a write to v in β_q.
(3) $|P_k| \geq \sqrt{|P_{k-1}|}/4(\ell + s + k)$.

First notice that $P_0 = Act(J)$ satisfies all the above conditions with $\beta_p = \langle\rangle$ for any $p \in Act(J)$. Assume we already constructed P_{k-1} satisfying the conditions of Lemma 7. For any $p \in P_{k-1}$ we denote by β_p the prefix as promised by Lemma 7 throughout the rest of this section, in which we define the construction of the write phase, and prove Lemma 7.

4.2.1 Construction: stage 1

For any $p \in P_{k-1}$ we let p perform a solo run starting from $J\beta_p$ until its next event e_p is a critical write, or that p has finish executing α_p and its next event is $EndFence$. Denote this extension by γ_p. Notice that $\beta_p\gamma_p$ is indeed a prefix of α_p by its definition. Let Z be the set of all processes that have not finished executing α_p, that is, $Z = \{p \in P_{k-1} \mid \beta_p\gamma_p \neq \alpha_p\}$.

If $|Z| < |P_{k-1}|/2$, i.e., at least half of the processes in P_{k-1} have finished executing α_p, then we define $t = k-1$ and $W = P_{k-1} \setminus Z$, and we move to stage 3 of the write phase. By its definition, each process $p \in W$ executes t

critical writes along α_p in the execution $J\alpha_p$. Moreover, it is easy to verify that condition (2) of Lemma 7 holds for W with α_p, since it holds for $W \subseteq P_{k-1}$ with β_p, and $\alpha_p = \beta_p\gamma_p$ where γ_p contains no critical writes. Notice that no $p \in W$ can write to a remote variable in γ_p it did not access in $J\beta_p$, as such write is critical, and therefore condition (2.c) follows immediately.

Otherwise $|Z| \geq |P_{k-1}|/2$. We define V_{next} to be the set of variables that are about to be written in one of the next events e_p by the processes in Z. Formally, $V_{next} = \{v \in V \mid \exists p \in Z$ such that e_p remotely writes v$\}$. In order to construct P_k the following stage will handle the cases of low and high contention separately.

4.2.2 Construction: stage 2

Case I: $|\mathbf{V_{next}}| < \sqrt{|\mathbf{Z}|}$

By the pigeonhole principle, there exists a variable $u \in V_{next}$ and a set $P_k \subseteq Z$ of size $|P_k| \geq \sqrt{|Z|}$, such that e_p is a critical commit write to u for any $p \in P_k$.

Case II: $|\mathbf{V_{next}}| \geq \sqrt{|\mathbf{Z}|}$

For each $v \in V_{next}$ we select an arbitrary $p \in Z$ such that $e_p = write(v)$. Denote this set by Z'. Then, $|Z'| = |V_{next}| \geq \sqrt{|Z|}$. We construct an undirected graph $\mathcal{G}$ as follows: the vertices of $\mathcal{G}$ are the processes in Z'. Consider $p \in Z'$, and denote $e_p = write(v)$. For $q \in Z'$, we add an edge $\{p, q\}$ in $\mathcal{G}$ if either *a)* v is local to q; or *b)* q access v in $J\beta_q$.

Since each first access to a remote variable is critical, the number of different remote variables accessed by some process $p \in Z'$ is at most the number of critical events it executes. Therefore, the number of new edges introduce by rule b) for p is at most the number of critical events p executes in $J\beta_p$, which is $\ell + s + k - 1$. Since each variable is local at most one process, at most one new edge is introduce by rule a). Altogether, the average degree in $\mathcal{G}$ is at most $2(\ell+s+k)$. By Theorem 2, there exists an independent set $P_k \subseteq Z'$ in $\mathcal{G}$ such that:

$$|P_k| \geq \frac{|Z'|}{2(\ell + s + k) + 1} \geq \frac{\sqrt{|Z|}}{2(\ell + s + k) + 1}$$

We now prove that in both cases P_k satisfies all the conditions of Lemma 7, where for each $p \in P_k$ we choose $\beta'_p = \beta_p\gamma_p e_p$ to be the prefix of α_p.

Claim 4.2.1. *P_k satisfies Lemma 7.*

Proof. We first prove condition (2). Notice that P_k satisfies it with the prefixes β_p, since P_{k-1} did. Each β_p was extended such that p have one more critical write. In case I all processes have one more critical write to the same variable u. In case II all new critical writes are to different variables, where the independent set assures no such variable is accessed by some other process in P_k. This gives an intuition of why condition (2) holds with the new prefixes β'_p as well. Formally, let $e \in \beta'_p$ be a critical write in $J\beta'_p$ by some $p \in P_k$ to some variable v. We consider two cases:

Assume $e \in \beta_p$. Since (a) and (b) holds with P_{k-1} and $P_k \subseteq P_{k-1}$, it follows immediately they both holds with P_k as well. If any $q \in P_{k-1}$ have a write to v in β_q, then so is the case with β'_q, and condition (c) holds. Otherwise, consider some $q \in P_k$ different then p. By condition (c) q does not access v in $J\beta_q$. Assume towards a contradiction it does access v in $\gamma_q e_q$. Following condition (a) $q \neq owner(v)$, therefore this write is critical, i.e., it is e_q. In case I we get that $v = u$, and as so p have two different critical write, e and e_p, to the same variable v along a solo run β'_p, in contradiction. In case II, we get that p access the variable of e_q in $J\beta_p$, thus there is an edge $\{p, q\}$ in $\mathcal{G}$, contradicting the fact that $p, q \in P_k$, an independent set.

Otherwise $e = e_p$. In case I, any $q \in P_k$ have a critical write e_q to $u = v$, and therefore $q \neq owner(v)$. Moreover, e_q must be the first write to v in β'_q, as it is critical and β'_q is a solo run. It follows that $q \neq writer(v, J)$, and conditions (2) follows. In case II, consider some $q \in P_k$. If $q \neq owner(v)$ and it access v in $J\beta'_q$, then the first such access is critical. This critical event is different then e_q, since e_p and e_q writes to different variables, therefore it must be in $J\beta_q$. In such case, or if $q = owner(v)$, there is an edge $\{p, q\}$ in $\mathcal{G}$. This contradict the fact that P_k is an independent set. It follows that $q \neq owner(v)$, and it does not access v in $J\beta'_q$, and in particular $q \neq writer(v, J)$, and condition (2) holds.

Condition (1) follows from construction. By induction hypothesis, each $p \in P_k$ executes $k - 1$ critical writes along β_p in the run $J\beta_p$. We extended β_p with a solo run such that γ_p contains no critical events, end e_p is a critical event. Therefore p executes k critical events along $\beta'_p = \beta_p\gamma_p e_p$ in the run $J\beta'_p$.

For condition (3), in both cases we have a set P_k satisfying:

$$|P_k| \geq \frac{\sqrt{|Z|}}{2(\ell + s + k) + 1} \geq \frac{\sqrt{|P_{k-1}|/2}}{2(\ell + s + k) + 1} \geq \frac{\sqrt{|P_{k-1}|}}{4(\ell + s + k)}$$

□

This finish the inductive step of the write phase. At the last step we have large enough set of processes that have finish executing α_p. In this case, we move to stage 3, where we define how to use this set in order to construct the execution for the regularization phase.

4.2.3 Construction: stage 3

We have a set $W \subseteq Act(J)$ satisfying condition (2) of Lemma 7 with α_p, such that each $p \in W$ executes t critical writes along α_p in the execution $J\alpha_p$. We extend J by letting each process $p \in W$ perform its run α_p in some arbitrary way. Denote this extension by D, and let $p_r \in W$ be the last process to perform its α_p in D. First notice the D is indeed an extension of J - since all processes performs only writes, we can order them in any way to get an extension, as writes of one process does not affect writes of other.

Claim 4.2.2. *For any $Y \subseteq Act(J)$, $p \in W \setminus Y$, and $e \in \alpha_p$ the following holds: e is a critical event in $J\alpha_p$ if and only if it is critical in $(JD)^{-Y}$.*

Proof. Let Y, p be as in the claim. Consider $e \in \alpha_p$, a write to variable v.

Assume e is critical in $J\alpha_p$. By Lemma 7 we have $owner(v) \notin W$ and $q = writer(v, J) \notin W$. If $q \notin Y$, then $q = writer(v, J^{-Y})$. Otherwise, $q \in Y$. Since J is a regular execution, by IN5 q is the only process in $Act(J)$ to access v in J. After removing events by q we get $writer(v, J^{-Y}) \notin Act(J)$. In particular, in both cases we have $writer(v, J^{-Y}) \neq p$. Notice that e is the first write in α_p to v, since this event is critical in $J\alpha_p$. As a result, in D^{-Y} p executes α_p, where the first write to v in it is e, and all the events preceding α_p are by processes different then p. Therefore, either there is a write to v before e in D^{-Y} by some process different then p, or that the last write is in J^{-Y}. In both cases we get that e is a critical write in $J^{-Y}D^{-Y}$.

Assume e is not critical in $J\alpha_p$. If e is not the first write to v in α_p, then so is the case in D^{-Y} which contains α_p, and on both cases e is not a critical write. Otherwise, e is the first write to v in α_p. If $p = owner(v)$, then by its definition a write by p to v is not critical in any execution, and we done. Assume $p \neq owner(v)$, then it must be that $p = writer(v, J)$. Since J is regular, by IN4 $owner(v) \notin Act(J)$. Consider some $q \in W$ different then p, then $q \neq owner(v)$. If q does write to v in α_q, the first such write is critical in $J\alpha_q$. By Lemma 7 we get that $p = writer(v, J) \notin W$, in contradiction. As a result, no process in W besides p writes to v in D, and therefore in D^{-Y}. Meaning, the last write to v before e in $J^{-Y}D^{-Y}$ is in J^{-Y}. As we did not remove events by process p we have $p = writer(v, J^{-Y})$, and e is not critical in $J^{-Y}D^{-Y}$. □

Denote $\overline{W} = Act(J) \setminus W$, and let $L_0 = (JD)^{-\overline{W}}$. Since no process in $\overline{W}$ take steps in D we get $L_0 = J^{-\overline{W}}D$. First notice that L_0 is indeed an execution. Since J is a regular execution, by Lemma 4 with $'E' \leftarrow J$ and $'Y' \leftarrow \overline{W}$, we have a regular execution $J^{-\overline{W}}$. Moreover, D is an extension of $J^{-\overline{W}}$ - for any $p \in W$ we have $J \mid p = J^{-\overline{W}} \mid p$. Therefore, after both executions p have the same writes, α_p, in its write buffer. Furthermore, since all processes performs only writes in D, any order of these writes is an extension of $J^{-\overline{W}}$, as writes of one process does not affect writes of other.

Claim 4.2.3. *$W \setminus \{p_r\}$ is an IN-set of L_0.*

Proof. Denote $W' = W \setminus \{p_r\}$. Notice that $J^{-\overline{W}}$ is a regular execution, and $W' \subseteq Act(J^{-\overline{W}}) = W$. Therefore W' is an IN-set of $J^{-\overline{W}}$.

IN1: for any $p \in P$ we have $AW(p, J^{-\overline{W}}) \cap W' \subseteq \{p\}$. In addition, D contains only write events, therefore no process change its awareness-set along D, resulting $AW(p, J^{-\overline{W}}D) \cap W' \subseteq \{p\}$.

IN2: for any $p \in W'$ we have $status(p, J^{-\overline{W}}) = entry$. In addition, D contains only write events, therefore no process change its status in D and $status(p, J^{-\overline{W}}D) = entry$.

IN3: consider $Y \subseteq W'$, and consider $e \in (J^{-\overline{W}}D)^{-Y}$. If $e \in (J^{-\overline{W}})^{-Y}$, then since $J^{-\overline{W}}$ is regular, by IN3 we get: e is a critical event in $J^{-\overline{W}}$ if and only if it is critical in $(J^{-\overline{W}})^{-Y}$. Otherwise $e \in D^{-Y}$. Let p be the process to execute e, then by claim 4.2.2 we get: e is critical in $(JD)^{-(\overline{W} \cup Y)} = (J^{-\overline{W}}D)^{-Y}$ if an and if it is critical in $J\alpha_p$ if and only if it is critical in $(JD)^{-\overline{W}} = L_0$. Altogether we get that e is a critical event in L_0 if and only if it is critical in L_0^{-Y}.

IN4: Let $e \in L_0$ be an event by p accessing a remote variable v. w.l.o.g e is the first access by p to v in L_0, and as so it is a critical event. If $e \in J^{-\overline{W}}$, then by IN4 we get $owner(v) \notin Act(J^{-\overline{W}}) = W = Act(L_0)$. Otherwise, $e \in D$, a critical write in $(JD)^{-\overline{W}}$. By claim 4.2.2 e is a critical write in $J\alpha_p$, and therefore by condition (2) of Lemma 7 for W we get $owner(v) \notin W = Act(L_0)$.

IN5: Let v be a variable such that $p = writer(v, L_0) \in W'$. If v is local to p, then by IN4 there is no other process to write to v in L_0, and we done. Assume otherwise, then by IN4 $owner(v) \notin W$. Let e be the last critical write to v in L_0. Notice that e is an event by p, since it is the last process to write to v in L_0. If $e \in D$, then by claim 4.2.2 it is critical in $J\alpha_p$. By condition (2) of Lemma 7, either any $q \in W$ have a write to v in its α_q, and in particular, since D ends with α_{p_r} we get $writer(v, L_0) = p_r \notin W'$ in contradiction. Therefore, there is no $q \neq p$ in W to access v in $J\alpha_q$, and thus in L_0, and IN5 holds ($Accessed(v, L_0) \cap Act(L_0) = \{p\}$). If $e \in J^{-\overline{W}}$, then no process but p can write to v in D (since such write is critical, contradicting the fact that e is the last such write). Furthermore, we have $writer(v, J^{-\overline{W}}) = p$, and since $J^{-\overline{W}}$ is a regular execution, by IN5 p is the process in $Act(J^{-\overline{W}}) = W$ to access v in $J^{-\overline{W}}$. Altogether, p is the only process in W to access v in $L_0 = J^{-\overline{W}}D$, and IN5 holds. □

Each $p \in W$ commits all the writes in its write buffer along D, hence its next event is $EndFence$. We extend L_0 by letting each $p \in W$ execute $EndFence$ in some arbitrary order. By abuse of notation we denote the new execution by L_0 as well, since it retains all the properties of the previous L_0. Then L_0 satisfies the following conditions:

1. $Act(L_0) \setminus \{p_r\}$ is an IN-set of L_0;
2. Each $p \in Act(L_0) = W$ execute $\ell + s$ critical events in $J^{-\overline{W}}$, and by claim 4.2.2 another t critical events along D in $J^{-\overline{W}}D$. Therefore, p executes $\ell + s + t$ critical events in L_0;
3. Each $p \in Act(L_0) = W$ completes i fences in $J^{-\overline{W}}$, and have one more $EndFence$ as its last event in L_0. Thus, p completes $i + 1$ fences in L_0 and $mode(p, L_0) = read$;
4. $Fin(L_0) = Fin(J) = Fin(H_i)$;
5. $|Act(L_0)| \geq |P_k|/2$.

This conclude the write phase, and we proceed to the regularization phase. We now give an upper bound on the number of steps in the read and write phases.

Claim 4.2. *The number of steps in the read and write phases is bounded by $f(i+1) - \ell$. In other words: $\ell + s + t \leq f(i+1)$.*

Proof. Assume towards a contradiction that $\ell + s + t > f(i+1)$. Then we have an execution L_0 such that:

- $Act(L_0) \setminus \{p_r\}$ is an IN-set of L_0;
- p_r executes $\ell + s + t$ critical events in L_0;
- $Fin(L_0) = Fin(H_i)$, thus $|Fin(L_0)| = i$;

Using Lemma 4 with $'E' \leftarrow L_0$ and $'Y' \leftarrow Act(L_0) \setminus \{p_r\}$, we have an execution $L' = L_0^{-Y}$ such that $Act(L') = \{p_r\}$ and $Fin(L') = Fin(L_0)$. By IN3 for L_0, p_r executes the same critical events in L_0 and L', that is, p_r executes $\ell + s + t > f(i+1)$ critical events in L'. However, exactly $i + 1$ processes issue events in L', i.e. the total contention of L' is $i + 1$, while p_r executes more then $f(i+1)$ critical events during a single passage in L', a contradiction. □

4.3 Regularization phase

The write phase construct an execution L such that each process in $Act(L)$ completes $i + 1$ fences. Moreover, $Act(L)$ can be written as $Q \cup \{p_r\}$ where Q is an IN-set of L. The regularization phase will be used to let p_r complete its passage, such that we get a regular execution.

By Lemma 4 with $'E' \leftarrow L$ and $'Y' \leftarrow Q$ we get an execution L^{-Q} such that the following holds:

1. $Act(L^{-Q}) = Act(L) \setminus Q = \{p_r\}$;
2. $Fin(L^{-Q}) = Fin(L) = Fin(H_i)$, and therefore $|Fin(L^{-Q})| = i$.

Let α be a solo run of p_r starting from L^{-Q} until it finish its passage. Notice that such an extension exists by the progress property. We denote by Q^- the set of processes $q \in Q$ such that there exist $e \in \alpha$, a critical event in $L^{-Q}\alpha$ accessing variable v, such that either $q = owner(v)$ or $q = writer(v, L)$.

We would like to extend L with α, but it might be that a solo run of p_r after L and L^{-Q} is different. However, it is enough to erase the processes in Q^- from L to guarantee that α is also a solo run of p_r after L^{-Q^-}.

Denote $Q = Q^- \cup Q^+$ a disjoint union, that is, $Q^+ = Q \setminus Q^-$. By Lemma 4 with $'E' \leftarrow L$ and $'Y' \leftarrow Q^-$ we get that L^{-Q^-} is an execution such that $Act(L^{-Q^-}) = Q^+ \cup \{p_r\}$ and Q^+ is an IN-set of L^{-Q^-}.

Claim 4.3.1. *For any variable v accessed in α: $writer(v, L^{-Q^-}) \notin Q$.*

Proof. Assume towards a contradiction there is a variable v such that p_r access v and $q = writer(v, L^{-Q^-}) \in Q$. Since no process in Q^- takes steps in L^{-Q^-} it follows that $q \notin Q^-$. By the definition of Q^- we can conclude that $q \neq writer(v, L)$. Let p be $writer(v, L)$, then $p \neq q$. It must be that $p \in Q^-$, otherwise p executes the same events in L and in L^{-Q^-}, and therefore it is the last process to write to v in both, that is, $p = writer(v, L) = writer(v, L^{-Q^-}) = q$ in contradiction. Altogether, we have two different processes $p, q \in Q$ accessing v in L. Since Q is an IN-set of L, by IN5 $p = writer(v, L) \notin Q$, in contradiction. □

Using the claim above, for any variable v accessed in α we have $q = writer(v, L^{-Q^-}) \notin Q$. That is, the last write to v in L^{-Q^-} is by a process not in Q^+, and therefore after erasing the processes in Q^+ from this execution, q still executes the same events in the resulting execution, and thus the last write to v in it is the same event by q. Meaning, in both L^{-Q^-} and $(L^{-Q^-})^{-Q^+} = L^{-Q}$ the last event to write to v is identical.

As a result, p_r reads the same values from the same variables if running solo after both L^{-Q^-} and L^{-Q}. Since α is a solo run of p_r starting from L^{-Q} it is also a solo run starting from L^{-Q^-}. Define $H_{i+1} = L^{-Q^-}\alpha$.

Claim 4.3.2. *H_{i+1} is a regular execution.*

Proof. As $Act(L^{-Q^-}) = Q^+ \cup \{p_r\}$, and in α we let p_r finish its passage, we have $Act(H_{i+1}) = Q^+$. Notice that Q^+ is an IN-set of L^{-Q^-}. We prove it is also an IN-set of H_{i+1}.

IN1: for any $p \in P$ we have $AW(p, L^{-Q^-}) \cap Q^+ \subseteq \{p\}$. If $p \neq p_r$ it does not take any steps in α, and IN1 follows. In addition, in $L^{-Q}\alpha$ no process in Q^+ take steps, therefore p_r does not become aware of any such process during this execution. As such, and since p_r executes the exact same events in $L^{-Q}\alpha$ and in H_{i+1}, we have $AW(p_r, H_{i+1}) \cap Q^+ = \emptyset$ and we done.

IN2: for any $p \in Q^+$ we have $status(p, L^{-Q^-}) = entry$. Since p takes no steps in α we get $status(p, H_{i+1}) = entry$ as well.

IN3: consider $Y \subseteq Q^+$. Then $H_{i+1}^{-Y} = (L^{-Q^-})^{-Y}\alpha$. Using IN3 with L^{-Q^-}, for any $e \in (L^{-Q^-})^{-Y}$: e is a critical event in $(L^{-Q^-})^{-Y}$ if and only if it is a critical event in L^{-Q^-}. Since (L^{-Q^-}) is a prefix of H_{i+1} it holds also with H_{i+1}.
For $e \in \alpha$ accessing a variable v, following the above proof for H_{i+1} being an execution, the last event to write to v in L^{-Q} and in L^{-Q^-} is the same. In particular, denote $q = writer(v, L^{-Q^-})$, then $q \notin Q$. Since we remove only events by processes in q we get $q = writer(v, (L^{-Q^-})^{-Y})$. Now, notice that e is a critical read in H_{i+1} if and only if it is a critical read in H_{i+1}^{-Y}. This follow from the fact that a critical read is the first remote read of some variable by p_r, and it executes the exact same events on both executions. Otherwise, it is a critical write in H_{i+1} if and only if it is the first write by p_r to v in α and $p_r \neq writer(v, L^{-Q^-}) = q$ (since α is a solo run by p_r, and any subsequent write to v is not critical). This happens if and only if e is the first write
For $e \in \alpha$, first notice that e is a critical read in H_{i+1} if and only if it is a critical read in H_{i+1}^{-Y}. This follow from the fact that a critical read is the first remote read of some variable by p_r, and it executes the exact same events on both executions. Assume e is a critical write in H_{i+1}^{-Y}. By claim 1, since $H_{i+1}^{-Y} \preceq H_{i+1}$ and p executes the same events on both, it follows that e is a critical write in H_{i+1}. It remains to prove the other direction. Assume e is a critical write in H_{i+1} to some v. Following the above proof for H_{i+1} being an execution, since p access v in α, then the last event to write to v in L^{-Q} and in L^{-Q^-} is the same. In particular, denote $q = writer(v, L^{-Q^-})$, then $q \notin Q$. We remove only events by processes in Q, thus the last write to v in $(L^{-Q^-})^-Y$ is still q. Since α is a solo run by p_r, and $e \in \alpha$ is a solo run it follows that e is the first write by p_r to v in α.

Moreover, e is critical in H_{i+1} because $q = writer(v, L^{-Q^-}) \neq p_r$. For proving

□

In the regularization phase we construct a sequence of executions $L_0, L_1, \dots, L_m, H_{i+1}$. Denote $\ell_{i+1} = \ell + s + t$.

Lemma 8 *In each step, we have an execution L_k such that the following conditions hold:*

(1) $Act(L_k) = W_k \cup \{p_r\}$ where W_k is an IN-set of L_k;
(2) p_r executes $\ell_{i+1} + k$ critical events in L_k;
(3) Each $p \in W_k$ executes ℓ_{i+1} critical events in L_k;
(4) Each $p \in W_k$ completes $i+1$ fences in L_k and $mode(p, L_k) = read$;
(5) $Fin(L_k) = Fin(H_i)$;
(6) $|Act(L_k)| \geq |Act(L_{k-1})| - 1$.

First notice that L_0 satisfies all the conditions of Lemma 8. Assume we already constructed L_{k-1} satisfying the conditions of Lemma 8. We denote $L = L_{k-1},\ n = |Act(L_{k-1})|$ throughout the rest of this section, in which we define the construction of the regularization phase and prove Lemma 8.

Lemma 3 implies that an extension containing no critical or transition events does not effect the IN-set, that is the IN-set remains the same after the extension. It is easy to verify that a transition event by a process not in the IN-set does not affect it as well (no variable is accessed, and the only process that changes its state is not in the IN-set). We therefore conclude that an extension by processes not in the IN-set which contains no critical events does not change the IN-set, and the next corollary follows.

Corollary 4 *Let F be an extension of L by p_{max} such that F contains no critical events in LF. Then W_{k-1} is an IN-set of LF.*

Let F be a solo extension of L by p_{max}, where p_{max} executes until it either terminates (that is, executes $Exit_{p_{max}}$), or until it is about to issue a critical event f. First, we prove that such an extension exists.

Assume towards a contradiction that the solo run F by p_{max} after L is infinite, where p_{max} does not finish a passage in F, and F contains no critical events in LF. Consider a finite prefix F' of F. p_{max} does not finish a passage in F, thus $Act(LF') = Act(L)$. F' contains no critical events in LF', thus, by Corollary 4, W_{k-1} is an IN-set of LF'. Using Lemma 4 with $'E' \leftarrow LF'$, $'INV' \leftarrow W_{k-1}$ and $'Y' \leftarrow W_{k-1}$, we get an execution $L' = (LF')^{-W_{k-1}}$ such that $Act(L') = Act(LF') \setminus W_{k-1} = \{p_{max}\}$. Notice that L' can be written as $L^{-W_{k-1}}F'$, since F' is a solo run by $p_{max} \notin W_{k-1}$. We have an execution L' in which there is a solo run F' by p_{max}, where p_{max} is the only active process along F', and p_{max} does not finish a passage. Since this holds for any prefix of F, F' can be as long as we wish, thus contradicting the global progress property.

Case I

p_{max} finishes a passage in F.
Define $m = k - 1$ and $H_{i+1} = LF$. The following conditions hold:

1. Since p_{max} finishes its passage in F, $Act(H_{i+1}) = Act(L) \setminus \{p_{max}\} = W_m$, thus $|Act(H_{i+1})| = |Act(L_m)| - 1$;
2. By Corollary 4, W_m is an IN-set of H_{i+1}, thus H_{i+1} is a regular execution;
3. Each $p \in Act(H_{i+1})$ executes ℓ_{i+1} critical events in L, and thus in H_{i+1};
4. Since p_{max} finishes its passage in F, we get $Fin(H_{i+1}) = Fin(L) \cup \{p_{max}\} = Fin(H_i) \cup \{p_{max}\}$. Therefore $|Fin(H_{i+1})| = i + 1$;
5. Each $p \in Act(H_{i+1})$ completes $i+1$ fences in H_{i+1} and $mode(p, H_{i+1}) = read$.

We are done with the regularization phase, and thus with the entire inductive step.

Case II

p_{max} is about to execute a critical event f after LF. Since p_{max} does not finish its passage, $Act(LF) = Act(L)$ and $Fin(LF) = Fin(L) = Fin(H_i)$. F contains no critical events in LF, thus by corollary 4, W_{k-1} is an IN-set of LF. Let u be the remote variable p_{max} accesses in f. We define:

$$q = \begin{cases} writer(u, LF) & , writer(u, LF) \in W_{k-1} \\ \perp & , otherwise \end{cases}$$

$$q_u = \begin{cases} owner(u) & , owner(u) \in W_{k-1} \\ \perp & , otherwise \end{cases}$$

Denote $Q = \{q, q_u\}$ and $W_k = W_{k-1} \setminus Q$.

Claim 4.3.3. $|Q| \leq 1$ *(where we do not count $\perp$).*

Proof. Assume $|Q| = 2$, then $q, q_u \in W_{k-1}$ and $q \neq q_u$. Since $writer(u, LF) = q$ and $q \neq owner(u)$, q remotely accessed u in LF. W_{k-1} is an IN-set of LF, thus by IN4 $q_u \notin Act(LF)$ - a contradiction. □

Since $Q \subseteq W_{k-1}$, by Lemma 4 with $'E' \leftarrow LF$, $'INV' \leftarrow W_{k-1}$ and $'Y' \leftarrow Q$, we have: $N = (LF)^{-Q}$ is an execution, and the following hold:

1. $W_k = W_{k-1} \setminus Q$ is an IN-set of N.
2. $Act(N) = Act(LF) \setminus Q = W_k \cup \{p_{max}\}$, thus $|Act(N)| \geq |Act(L)| - 1$.
3. $Fin(N) = Fin(LF) = Fin(H_i)$.
4. Each $p \in W_k$ executes the same events in N and in LF, thus p completes $i+1$ fences in N and $mode(p, N) = read$.

5. Each $p \in Act(N)$ executes the same critical events in N and in LF. Since F contains no critical events in LF, each $p \in W_k$ executes ℓ_{i+1} critical events in N, and p_{max} executes $\ell_{i+1} + k - 1$ critical events in N.
6. p_{max} is about to execute a critical event $e \sim f$ after N.

We extend N by letting p_{max} execute e, and denote the resulting execution $L_k = Ne$.

Claim 4.3.4. *W_k is an IN-set of L_k.*

Proof. We start by proving two properties relating to variable u.

- property 1: $writer(u, N) \notin W_k$. If $writer(u, LF) \notin W_{k-1}$ then $q = \perp$, thus the events by $writer(u, LF)$ have not been removed from N and we get $writer(u, N) = writer(u, LF) \notin W_k \subseteq W_{k-1}$. Otherwise, $writer(u, LF) = q \in W_{k-1}$. W_{k-1} is an IN-set of LF, hence, by IN5, q is the only process in $Act(LF)$ to access u in LF (otherwise $writer(u, LF) \notin W_{k-1}$, a contradiction). Therefore, after removing the events by $q \in Q$ there is no process in W_{k-1} that accesses u in N, i.e. $writer(u, N) \notin W_k \subseteq W_{k-1}$.
- property 2: $owner(u, N) \notin Act(N)$. If $owner(u) \notin Act(LF)$, then $owner(u) \notin Act(N) \subseteq Act(LF)$. Otherwise, $owner(u) \in Act(LF)$. Since p_{max} remotely accesses u in f, we have $owner(u) \neq p_{max}$, thus $owner(u) \in W_{k-1}$. From our construction, $owner(u) \in Q$, and therefore $owner(u) \notin Act(N)$.

W_k is an IN-set of N, thus, by the last two properties and by Lemma 2, IN1-IN4 hold for W_k in $Ne = L_k$. As IN5 holds for W_k in N, it clearly holds for any variable $v \neq u$ in L_k. Consider now variable u. Either e does not commit a write to u, and thus $writer(u, L_k) = writer(u, N) \notin W_k$, or e is a commit write to u, and thus $writer(u, L_k) = p_{max} \notin W_k$. In both cases, IN5 holds for u in L_k. As a result, W_k is an IN-set of L_k. ☐

We now prove that L_k satisfies all the conditions of Lemma 8:

(1) e is not a transition event, thus $Act(L_k) = Act(N) = W_k \cup \{p_{max}\}$ (where $p_{max} \notin W_k$);
(2) By claim 4.3.4 W_k is an IN-set of L_k;
(3) p_{max} eamxecutes $\ell_{i+1} + k - 1$ critical events in N, and e is a critical event in Ne. Therefore p_{max} executes $\ell_{i+1} + k$ critical events in L_k;
(4) Each $p \in W_k$ executes ℓ_{i+1} critical events in L_k;
(5) Each $p \in W_k$ executes the same events in L_k and in N, thus p completed $i + 1$ fences in L_k, and $mode(p, L_k) = read$;
(6) $Fin(L_k) = Fin(N) = Fin(H_i)$;
(7) $|Act(L_k)| = |Act(N)| \geq |Act(L)| - 1$.

Claim 4.3. *The number of steps in the regularization phase is bounded by $f(i + 1)$.*
(In other words, $m \leq f(i + 1)$.)

Proof. Assume towards a contradiction that during the regularization phase we build an execution L_k such that $k > f(i + 1)$. Then L_k satisfies:

- $Act(L_k)$ can be written as $W_k \cup \{p_{max}\}$ (where $p_{max} \notin W_k$);
- W_k is an IN-set of L_k;
- $Fin(L_k) = Fin(H_i)$, thus $|Fin(L_k)| = i$.
- p_{max} executes $\ell_{i+1} + k$ critical events in L_k.

Using Lemma 4 with $'E' \leftarrow L_k$ and $'INV', 'Y' \leftarrow W_k$, we have an execution $L'_k = L_k^{-W_k}$ such that: $Act(L'_k) = Act(L_k) \setminus W_k = \{p_{max}\}$ and $Fin(L'_k) = Fin(L_k) = Fin(H_i)$; p_{max} executes the same critical events in L'_k and in L_k, thus p_{max} executes $\ell_{i+1} + k$ critical events in L'_k.
Hence, at most $i + 1$ processes issue events in L'_k, i.e. the total contention of L'_k is at most $i + 1$. However, p_{max} executed $\ell_{i+1} + k > f(i+1)$ critical events during a single passage in L'_k, contradicting our assumption that the algorithm is f-adaptive. ☐

4.4 Construction Bounds

We now present an analysis for the size of $Act(H_i)$ based on the upper bounds on the number of steps in for each phase. We will prove a lower bound under some restriction on the growth rate of the adaptivity function f.

Theorem 3 *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.
We assume WLOG that the adaptivity function f is non-decreasing. We prove the theorem by induction on i. For $i = 0$, we have $|Act(H_0)| \geq N$ which is trivially true.

Let $i + 1$ be such that:

$$f(i+1) \leq \frac{N^{2^{-f(i+1)}}}{f(i+1)! \cdot 4^{f(i+1)+2(i+1)}}$$

Since f is non-decreasing:

$$f(i) \leq f(i+1) \leq \frac{N^{2^{-f(i+1)}}}{f(i+1)! \cdot 4^{f(i+1)+2(i+1)}} \leq \frac{N^{2^{-f(i)}}}{f(i)! \cdot 4^{f(i)+2i}}.$$

Hence i satisfies the condition in Theorem 3, and by the induction hypothesis, $|Act(H_i)| \geq \frac{N^{2^{-\ell_i}}}{\ell_i! \cdot 4^{\ell_i+2i}}$.

The induction step is partitioned into several sub-steps, corresponding to the phases in the construction of H_{i+1} from H_i. In each sub-step, we establish a lower bound on the number of active processes in the intermediate executions during the respective phase, based on the lower bound established for the phases preceding it.

Read phase: $|Act(G_k)| \geq \frac{N^{2^{-(\ell_i+k)}}}{(\ell_i+k)! \cdot 4^{\ell_i+k+2i}}$.
By induction on k.
Base case $k=0$: then $G_0 = H_i$ and the claim holds.
Induction step: assume we proved the claim for $k-1$. By condition (5) of Lemma 6:

$$|Act(G_k)| \geq \frac{|Act(G_{k-1})|-1}{10} \geq \frac{\frac{N^{2^{-(\ell_i+k-1)}}}{(\ell_i+k-1)! \cdot 4^{\ell_i+k-1+2i}} - 1}{10} \geq \frac{N^{2^{-(\ell_i+k)}}}{(\ell_i+k)! \cdot 4^{\ell_i+k+2i}}$$

where the last inequality holds as long as $\ell_i + k \geq 3$, which may be assumed since ℓ_i increases from phase to phase and k increases in the course of the read phase.

Write phase: $|Act(J_k)| \geq \frac{N^{2^{-(\ell_i+s+k)}}}{(\ell_i+s+k)! \cdot 4^{\ell_i+s+k+2i} \cdot 2}$
By induction on k.
Base case $k=0$:
$|Act(J_0)| \geq \frac{|Act(G_s)|}{2} \geq \frac{N^{2^{-(\ell_i+s)}}}{(\ell_i+s)! \cdot 4^{\ell_i+s+2i} \cdot 2}$.
Induction step: assume we proved the claim for $k-1$. By condition (5) of Lemma **??**:

$$|Act(J_k)| \geq \frac{\sqrt{|Act(J_{k-1})|}}{4(\delta+k)} \geq \frac{\sqrt{\frac{N^{2^{-(\ell_i+s+k-1)}}}{(\ell_i+s+k-1)! \cdot 4^{\ell_i+s+k-1+2i} \cdot 2}}}{4(\ell_i+s+k)} \geq \frac{\frac{\sqrt{N^{2^{-(\ell_i+s+k-1)}}}}{(\ell_i+s+k-1)! \cdot 4^{\ell_i+s+k-1+2i} \cdot 2}}{4(\ell_i+s+k)} = \frac{N^{2^{-(\ell_i+s+k)}}}{(\ell_i+s+k)! \cdot 4^{\ell_i+s+k+2i} \cdot 2}$$

Regularization phase:

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

By induction on k.

Base case $k=0$:

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

Induction step: assume we proved the claim for $k-1$. By condition (7) of Lemma 8:

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

.

Therefore we have:

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

From claim 4.3 and our assumption,

$$m \leq f(i+1) \leq \frac{N^{2^{-f(i+1)}}}{f(i+1)! \cdot 4^{f(i+1)+2(i+1)}}$$

By Claim 4.2, $\ell_{i+1} \leq f(i+1)$. Therefore, we can replace $f(i+1)$ with ℓ_{i+1} to get:

$$m \leq \frac{N^{2^{-f(i+1)}}}{f(i+1)! \cdot 4^{f(i+1)+2(i+1)}} \leq \frac{N^{2^{-\ell_{i+1}}}}{\ell_{i+1}! \cdot 4^{\ell_{i+1}+2(i+1)}} = \frac{1}{4} \cdot \frac{N^{2^{-\ell_{i+1}}}}{\ell_{i+1}! \cdot 4^{\ell_{i+1}+2i+1}} \tag{2}$$

Plugging Inequality 2 into Inequality 1 yields the required lower bound:

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

□

We can now prove our main result.

Theorem 1 (repeated) *Let $\mathcal{A}$ be an N-process weak obstruction-free f-adaptive implementation of a mutual-exclusion lock 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 whose total contention is $i+1$ and a process p such that p executes i fences in H during a single passage of its CS.*

Proof. Since $l_i < f(i)$, it follows from Theorem 3 that $|Act(H_i)| \geq f(i) \geq 1$. This implies that our construction results in an execution H_i, in which there is a process $p \in Act(H_i)$ and, from the properties of H_i, p is in a middle of a passage in which it executed (and completed) i fences. Moreover, from Lemma 4, we are able to erase all active processes but p from H_i and 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. □

5 Additional Objects

A *counter* is an object whose domain is $\mathbb{N}$. It supports a single operation, *fetch&increment*. The state of a counter is a natural number. The *fetch&increment* operation atomically increments C and returns its previous value. An *m-limited-use* counter allows at most m operation instances of *fetch&increment*. Notice that any counter is also an m-limited-use counter, for any m.

A *queue* object supports two operations: *enqueue* and *dequeue*. Each enqueue operation receives input v from a non-empty set of values V. Each dequeue operation applied to a non-empty queue returns a value $v \in V$. The state of a queue is a sequence of items $S = \langle v_0; \cdots; v_k\rangle$, each of which is a value from V. The semantics of the enqueue and dequeue operations is the following.

- *enqueue*(v_{new}) changes S to be the sequence $S = \langle v_0; \cdots; v_k; v_{new}\rangle$.
- if S is not empty, a dequeue operation changes S to be the sequence $S = \langle v_1; \cdots; v_k\rangle$ and returns v_0. If S is empty, dequeue returns the special value empty.

A *stack* object supports two operations: *push* and *pop*. Each push operation receives input v from a non-empty set of values V. Each pop operation applied to a non-empty stack returns a value $v \in V$. The state of a stack is a sequence of items $S = \langle v_0; \cdots; v_k\rangle$, each of which is a value from V. The semantics of the push and pop operations is the following.

- *push*(v_{new}) changes S to be the sequence $S = \langle v_0; \cdots; v_k; v_{new}\rangle$.
- if S is not empty, a pop operation changes S to be the sequence $S = \langle v_0; \cdots; v_{k-1}\rangle$ and returns v_k. If S is empty, pop returns the special value empty.

A *one-time* mutual exclusion algorithm is a ME algorithm that allows each process to complete a passage at most once. Since our proofs consider executions in which each process is allowed to complete a passage at most once, our results can be applied to one-time mutual exclusion algorithms.

Lemma 9 *Let $\mathcal{C}$ be a weak obstruction-free object of one of one 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 (fetch&increment, dequeue, or pop respectively) operation on $\mathcal{C}$, and that has the same RMR and fence complexities (in both DSM and CC models) as the operation it invokes, up to a constant additive factor.*

Proof. We first prove the lemma for an N-limited-use counter (hence also for a regular counter), by presenting Algorithm 1 for an N-process one-time mutual exclusion.

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) is followed by a fence instruction, and omit these fences from the code for presentation simplicity. Consequently, Algorithm 1 has the same fence complexity of the *fetch&increment* operation, up to a constant additive factor.

In the DSM model, process p will hold $spin[p]$ in its local memory segment. Since the only busy-waiting p may perform is on $spin[p]$, the ME algorithm has the same RMR complexity as that of the *fetch&increment* operation, up to a constant additive factor. In the CC model (with either write-back or write-through), since once $spin[p]$ is updated to 1 its value does not change again, p may encounter at most 2 RMRs during the wait in line 4, hence the ME algorithm has the same RMR complexity as of the *fetch&increment* operation, up to a constant additive factor.

Algorithm 1 One-time mutual exclusion from counter.

Shared Data: $release[N]$: a boolean array, initially $[1, 0, \cdots, 0]$
$waiting[N]$: an integer array, initially $[\bot, \bot, \cdots, \bot]$
$spin[N]$: a boolean array, initially $[0, 0, \cdots, 0]$
$\mathcal{C}$: an N-limited-use counter, initially 0

program for process p:

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

An N-limited-use counter can be implemented using a single queue or stack S in the following manner:

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

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

Counter, stack and queue objects can be easily implemented using the mutual exclusion algorithm presented by Attia et al [6]. Thus, each operation on these objects incurs $O(\log N)$ RMRs and a constant number of fences, and this is optimal [4]. On the other hand, Lemma 9 implies that given an f-adaptive algorithm for any of these objects, an f-adaptive mutual exclusion algorithm can be obtained. Moreover, each passage through the CS invokes a single operation on the respective object, and has the same asymptotic fence complexity of the object. Hence, any lower bound on the fence complexity of the resulting mutual exclusion algorithm implies a lower bound on the fence complexity for the operation of the respective object.

6 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 [6, 15], 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 $O(min(k, \log n))$, where k is point contention [20], 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 [7] 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

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

They also showed that the bound is tight.

Attiya et al. [6] 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.

References

1. Adve SV, Gharachorloo K (1996) Shared memory consistency models: A tutorial. computer 29(12):66–76
2. Anderson JH, Kim Y, Herman T (2003) Shared-memory mutual exclusion: major research trends since 1986. Distributed Computing 16(2-3):75–110, DOI 10.1007/s00446-003-0088-6, URL http://dx.doi.org/10.1007/s00446-003-0088-6
3. Attiya H, Hendler D (2005) Time and space lower bounds for implementations using k-cas. In: In DISC, pp 169–183
4. Attiya H, Hendler D, Woelfel P (2008) Tight RMR lower bounds for mutual exclusion and other problems. In: STOC, pp 217–226
5. Attiya H, Guerraoui R, Hendler D, Kuznetsov P, Michael MM, Vechev MT (2011) Laws of order: expensive synchronization in concurrent algorithms cannot be eliminated. In: POPL, pp 487–498
6. Attiya H, Hendler D, Levy S (2013) 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, pp 220–229, DOI 10.1145/2484239.2484255, URL http://doi.acm.org/10.1145/2484239.2484255
7. Attiya H, Hendler D, Woelfel P (2015) Trading fences with rmrs and separating memory models. In: Proceedings of the 2015 ACM Symposium on Principles of Distributed Computing, PODC 2015, Donostia-San Sebastián, Spain, July 21 - 23, 2015, pp 173–182, DOI 10.1145/2767386.2767427, URL http://doi.acm.org/10.1145/2767386.2767427
8. Ben-Baruch O, Hendler D (2015) The price of being adaptive. In: Proceedings of the 2015 ACM Symposium on Principles of Distributed Computing, PODC 2015, Donostia-San Sebastián, Spain, July 21 - 23, 2015, pp 183–192, DOI 10.1145/2767386.2767428, URL http://doi.acm.org/10.1145/2767386.2767428
9. Bollobas B (2004) Extremal Graph Theory. Dover Publications, Incorporated
10. Corporation I (2009) Intel ® 64 and IA-32 Architectures Software Developer's Manual. Intel Corporation
11. Danek R, Golab WM (2008) Closing the complexity gap between FCFS mutual exclusion and mutual exclusion. In: DISC, pp 93–108
12. Dijkstra EW (1965) Solution of a problem in concurrent programming control. Communications of the ACM 8(9):569
13. Fan R, Lynch N (2006) An $\Omega(n \log n)$ lower bound on the cost of mutual exclusion. In: PODC, pp 275–284
14. Golab WM, Hadzilacos V, Hendler D, Woelfel P (2007) Constant-RMR implementations of CAS and other synchronization primitives using read and write operations. In: Proceedings of the Twenty-Sixth Annual ACM Symposium on Principles of Distributed Computing, PODC 2007, pp 3–12
15. Golab WM, Hadzilacos V, Hendler D, Woelfel P (2012) RMR-efficient implementations of comparison primitives using read and write operations. Distributed Computing 25(2):109–162
16. Herlihy M, Wing JM (1990) Linearizability: a correctness condition for concurrent objects. ACM Trans Program Lang Syst 12(3):463–492
17. Herlihy M, Luchangco V, Moir M (2003) Obstruction-free synchronization: Double-ended queues as an example. In: Proceedings of the 23rd IEEE International Conference on Distributed Computing Systems (ICDCS'03), pp 522–529
18. Jayanti P (2003) Adaptive and efficient abortable mutual exclusion. In: PODC, ACM Press, New York, NY, USA, pp 295–304, DOI http://doi.acm.org/10.1145/872035.872079
19. Jayanti P, Petrovic S, Narula N (2005) Read/write based fast-path transformation for FCFS mutual exclusion. In: SOFSEM, pp 209–218
20. Kim Y, Anderson JH (2007) Adaptive mutual exclusion with local spinning. Distributed Computing 19(3):197–236, DOI 10.1007/s00446-006-0009-6, URL http://dx.doi.org/10.1007/s00446-006-0009-6
21. Kim Y, Anderson JH (2012) A time complexity lower bound for adaptive mutual exclusion. Distributed Computing 24(6):271–297, DOI 10.1007/s00446-011-0152-6, URL http://dx.doi.org/10.1007/s00446-011-0152-6
22. Lamport L (1997) How to make a correct multiprocess program execute correctly on a multiprocessor. IEEE Trans Computers 46(7):779–782
23. LWeaver D, Germond T (1994) The SPARC Architecture Manual. Prentice Hall
24. Owens S, Sarkar S, Sewell P (2009) 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, pp 391–407
25. Park S, Dill DL (1999) An executable specification and verifier for relaxed memory order. IEEE Trans Computers 48(2):227–235, DOI 10.1109/12.752664, URL

http://doi.ieeecomputersociety.org/10.1109/12.752664
26. Patterson DA, Hennessy JL (1994) Computer Organization & Design: The Hardware/Software Interface. Morgan Kaufmann
27. Taubenfeld G (2006) Synchronization Algorithms and Concurrent Programming. Prentice Hall
28. Yang JH, Anderson J (1995) A fast, scalable mutual exclusion algorithm. Distributed Computing 9(1):51–60

Appendix A Proofs Omitted from Paper Body

Claim 1 (repeated) *Let E be an execution fragment, and $e \in E$ an event issued by some process p.*

- *Assume e is a non-special event in E. Then for any execution fragment $F \preceq E$ such that $F \mid p = E \mid p$, e is a non-special event in F.*
- *Assume e is a special event in E. Then for any execution fragment F such that $E \preceq F$ and $F \mid p = E \mid p$, e is a special event in F.*

Proof.

- Assume e is not a special event in E, then e is either a read or a write event.
  If e is a local event in E then e is a local event in F and the claim clearly holds. Otherwise, e is a remote event in both E and F. The following two cases exist.
  - $e = read(v)$: then, since e is not a critical event in E, e is not the first remote read of v by p in E, and since $F \mid p = E \mid p$, e is not the first remote read of v by p in F, thus e is a non-critical read in F.
  - $e = write(v)$: then, since e is not a critical event in E, p is the last process to commit a write to v before e in E, denote this write event by e'. Since $F \mid p = E \mid p$, e' occurs in F as well. There is no write commit between e' and e in E, and since $F \preceq E$ there is no write commit between e' and e in F as well. Therefore p is the last process to commit a write to v before e in F, thus e is a non-critical write in F as well.
- Assume e is a special event in E. If e is a transition or a fence event in E, then it is also a transition or a fence event in F and we are done. Assume, then, that e is either a critical read or a critical write event in E.
  - If $e = read(v)$, then e is the first remote read of v by p in E. Since $F \mid p = E \mid p$, e is the first remote read of v by p in F, thus a critical read in F as well.
  - If $e = write(v)$, then the last process to commit a write to v before e in E (if any) is not p. Since $F \mid p = E \mid p$, no write commit by p has been added to F, and since $E \preceq F$, no write commit by another process has been removed, thus the last process to commit a write to v before e in F (if any) is not p, so e is a critical write in F as well.

□

Lemma 1 (repeated) *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.*

Proof. We prove the claim by induction on the number of events in E^{-p}. The base case $E^{-p} = \langle\rangle$ is trivial. For the induction, let $E^{-p} = Ff$ such that $E = E_1 f E_2$, $F = E_1^{-p}$, and E_2 is a (possibly empty) solo execution by p. Assume that F is an execution, and let q be the process that executes f.

Process q executes the same events in E_1 and in F, thus it is in the same state after both executions and about to execute the same event f. If f is a transition or fence event, then Ff is clearly an execution. This is the case also if f is a write event. Otherwise, assume $f = read(v)$. The following two cases exist.

- Event f reads a copy of v from q's write buffer in execution E. In this case, since q is in the same state after F and E_1, it will read the same value from its write buffer after F.
- Otherwise, f reads v from the shared memory. Since $p \notin AW(q, E)$, the last process that commits a write to v before f in E_1 is not p. Hence, since all the processes except p execute the same events in E_1 and in F, v has the same writer and value after both executions. Consequently, f reads the same value in Ff and in $E_1 f$, so Ff is an execution.

□

Lemma 2 (repeated) *Let E be an execution and let INV be an IN-set of E. Let e be a $read(v)$ or $write(v)$ event. Assume $writer(v, E) \notin INV$ and $owner(v) \notin Act(E)$.*
Then INV satisfies IN1-IN4 of Definition 4 in Ee.

Proof. Denote $F = Ee$. First notice that $p \in Act(E)$, otherwise the only event p may execute after E is $Enter_p$. Thus we get $Act(F) = Act(E)$.

IN1: For any $p' \neq p$ we have $AW(p', F) = AW(p', E)$, thus IN1 holds for p' in F. Denote $q = writer(v, E)$, then $q \notin INV$ and, by IN1 $AW(q, E), \cap INV = \emptyset$. Writing v does not change p's awareness-set. Reading v expands p's awareness-set with $AW(q, E)$ at most, and thus p cannot become aware of any invisible process by reading v. Altogether, p does not become aware of any invisible process by accessing v, i.e., $AW(p, F) \cap INV \subseteq \{p\}$, and IN1 holds for p in F.

IN2: IN2 holds in E. Since e is not a transition event, no process changes its status during e and so IN2 holds in F.

IN3: Consider $Y \subseteq INV$. IN3 holds for any event in E, thus it suffices to prove that it holds for e as well. Assume $e \in F^{-Y}$, that is $p \notin Y$ and therefore $F \mid p = F^{-Y} \mid p$. If e is a local event in F, then e is a local event in F^{-Y} as well, and thus a non-critical event in both. Otherwise, assume e is a remote event in F. The following cases exist.

- If $e = read(v)$, then e is the first remote read of v by p in F if and only if it is the first remote read of v by p in F^{-Y}. Therefore e is a critical event in F if and only if it is a critical event in F^{-Y}.
- If $e = write(v)$, denote $q = writer(v, E)$. As $q \notin INV$ and $Y \subseteq INV$, we have $q \notin Y$, thus after removing all the events by processes in Y, we still have $writer(v, E^{-Y}) = writer(v, E) = q$. Consequently, either $q \neq p$ and e is critical in both F and F^{-Y}, or $p = q$ and e is non-critical in both executions.

IN4: IN4 holds in E and $Act(E) = Act(F)$, thus it holds for any variable $u \neq v$ in F. The only remote variable that may be accessed in e is v, and $owner(v) \notin Act(E) = Act(F)$. □

Claim 4 (repeated) *Let E be an execution and let INV be an IN-set of E. Let e be an extension of E by some process p such that e is a local event in Ee. Then INV is an IN-set of Ee.*

Proof. Denote $F = Ee$. First notice that $p \in Act(E)$, otherwise the only event p may execute after E is $Enter_p$, which is not a local event after E. Thus we get $Act(F) = Act(E)$.

IN1: For any $q \neq p$ we have $AW(q, F) = AW(q, E)$, thus IN1 holds for q in F. If v is remote to p, then, since e is a local event and does not access any remote variable, e either stores a write to p's write buffer or reads a copy of v from p's write buffer. In both cases, p's awareness set does not change as a result of executing e. Otherwise, v is local to p. In this case, since $p = owner(v) \in Act(E)$, we have by IN4 that p is the only process that accesses v in E (otherwise there is a remote access to v, thus by IN4 $p = owner(v) \notin Act(E)$, a contradiction). Hence, after accessing v (by either a read or a write event), p's awareness set does not change. It follows that $AW(p, F) = AW(p, E)$ and IN1 holds for p in F.

IN2: IN2 holds in E. Since e is not a transition event, no process changes its status after e, and IN2 holds in F.

IN3: Consider $Y \subseteq INV$. IN3 holds for all of E's events, thus it is sufficient to prove that it holds for e as well. If $e \in F^{-Y}$, then $p \notin Y$, thus $F \mid p = F^{-Y} \mid p$. Therefore e is a local event in both F and F^{-Y} and thus a non-critical event in both.

IN4: Since IN4 holds in E and no remote variable is accessed in e, and since $Act(F) = Act(E)$, IN4 holds in F as well.

IN5: IN5 holds in E, thus it holds in F for any variable that is not local to p, since p does not access any of these in e. Let v be a local variable of p accessed in e. Since IN4 holds in F and $p \in Act(F)$, there is no remote access to v in F. That is, p is the only process to access v in F, therefore $|Accessed(v, F)| = 1$, and IN5 holds for v in F. □

Lemma 3 (repeated) *Let E be an execution and let INV be an IN-set of E. Let F be an extension of E such that F contains no critical or transition event in EF. Then INV is an IN-set of EF.*

Proof. We first prove the claim for the case of a single event, $F = f$. The general case can be then proven by induction on the number of events in F. Let p be the process that executes f, then $p \in Act(E)$, otherwise the only event p may execute after E is $Enter_p$, which is a transition event. Thus $Act(Ef) = Act(E)$.

If f is a fence event, then f does not access any variable and no process changes its status. It is easily verified that in such case IN1-IN5 holds for INV in Ef as well, thus INV is an IN-set of Ef. If f is a local event, then by Claim 4, INV is an IN-set of Ef.

Assume, then, that f is a remote event and let v be the variable accessed in f. As f is not critical in Ef, this is not the first access of v by p, so p accesses v in E. Therefore, by IN4 applied to E, we have $owner(v) \notin Act(E)$. Denote $q = writer(v, E)$. If $q \notin INV$, then by Lemma 2, IN1-IN4 hold in Ef. Otherwise $q \in INV$, and by IN5 applied to E, q is the only active process that accessed v in E, so $p = q$. To conclude the proof, we now prove that IN1-IN4 hold in the latter case ($q \in INV$) and that IN5 holds in both cases.

IN1: For any $p' \neq p$ we have $AW(p', Ef) = AW(p', E)$, thus IN1 holds for p' in Ef. As for p, since $p = writer(v, E)$, accessing v does not change its awareness-set, that is $AW(p, Ef) = AW(p, E)$, and so IN1 holds for p in Ef as well.

IN2: As f is not a transition event, for any $p \in INV$: $status(p, Ef) = status(p, E) = entry$.

IN3: Consider $Y \subseteq INV$. As IN3 holds for E, it is sufficient to prove that it holds for f. Assume $f \in (Ef)^{-Y}$, that is $p \notin Y$ and $E \mid p = E^{-Y} \mid p$. If f is a remote read, then, from our assumption that it is not critical in Ef, f is not the first remote read by p of v in Ef and thus in $(Ef)^{-Y}$, that is, f is non-critical in both. Otherwise, since $p = writer(v, E)$ holds, $p = writer(v, E^{-Y})$ holds as well, therefore f is a non-critical write in both Ef and $(Ef)^{-Y}$.

IN4: IN4 holds in E and $Act(E) = Act(Ef)$, thus it holds for any variable $u \neq v$ accessed in Ef. The only variable accessed in f is v, and we already know that $owner(v) \notin Act(E) = Act(Ef)$.

IN5: IN5 holds in E, thus it holds for any variable $u \neq v$. As for v, if $q \notin INV$ then $f = read(v)$, otherwise f would have been a critical write, a contradiction. Hence $writer(v, Ef) = writer(v, E) = q \notin INV$, and we are done. Otherwise, $p = q \in INV$, and by IN5 applied to E we get that p is the only process to access v in E (otherwise $p = writer(v, E) \notin INV$, contradicting our assumption). Therefore p is the only process to access v in Ef, and IN5 holds for v in Ef. □

Lemma 4 (repeated) *Let E be an execution, INV be an IN-set of E and $Y \subseteq INV$.*
Define $E' = E^{-Y}$. Then the following hold:

1. *E' is an execution;*
2. *$Act(E') = Act(E) \setminus Y$ and $Fin(E') = Fin(E)$;*
3. *$INV \setminus Y$ is an IN-set of E';*
4. *Each $p \in Act(E')$ executes the same critical events in E' and in E;*
5. *If $p \in Act(E')$ is about to execute a special event f_p after E, then p is about to execute a special event $e_p \sim f_p$ after E'.*

Proof. We first prove the claim for the case $Y = \{p\}$, a single process.

1. Consider $q \in P$ different from p. Since $p \in INV$, by IN1 q is not aware of p in E, i.e. $p \notin AW(q, E)$. By Lemma 1 E^{-p} is an execution.
2. We removed the events of $p \in Act(E)$, thus $Act(E') = Act(E) \setminus \{p\}$ and $Fin(E') = Fin(E)$.
3. We prove $INV \setminus \{p\}$ is an IN-set of E':
   IN1: Consider $q \neq p$. Since $E \mid q = E' \mid q$, we have $AW(q, E) = AW(q, E')$. By IN1, $AW(q, E) \cap INV \subseteq \{q\}$, in particular $AW(q, E') \cap INV \setminus \{p\} \subseteq \{q\}$. p executes no events in E', thus $AW(p, E') = \emptyset$ and IN1 holds for p in E'.
   IN2: For any $q \in INV \setminus \{p\}$, we have $E' \mid q = E \mid q$, thus $status(q, E') = status(q, E) = entry$.
   IN3: Consider $Z \subseteq INV \setminus \{p\}$, and $e \in E'^{-Z} = E^{-Z \cup \{p\}}$. Notice that $e \in E, E^{-p}, E'^{-Z}$. Since $Z, \{p\} \subseteq INV$ and INV is an IN-set of E, by IN3 applied to E we have: e is a critical event in $E^{-Z \cup \{p\}} = E'^{-Z}$ if and only if e is a critical event in E if and only if e is a critical event in $E' = E^{-p}$.
   IN4: Consider an event $e \in E'$ by process q accessing a remote variable v. Since $e \in E$ accesses a remote variable v, by IN4 $owner(v) \notin Act(E)$, and thus $owner(v) \notin Act(E') \subseteq Act(E)$.
   IN5: Assume $|Accessed(v, E') \cap Act(E')| > 1$ for some v. Since $E' \preceq E$ and $Act(E') \subseteq Act(E)$ 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 $writer(v, E') = writer(v, E) \notin INV$, and in particular $writer(v, E') \notin INV \setminus \{p\}$.

4. Follows directly from IN4 in E and the fact that $p \in INV$.
5. Assume $q \in Act(E')$ is about to execute a special event f_q after E. Notice that $E' \mid q = E \mid q$, thus q is about to execute an event $e_q \sim f_q$ after E'. We now prove that e_q is a special event in $E'e_q$:
If f_q is a fence or transition event then so is e_q and we are done. If f_q is the first access by q to some remote variable v (either read or write) in Ee_q, then e_q is also so in $E'e_q$, and is therefore critical. If f_q is a critical write to a remote variable v, and this is not the first write committed by q to v in Ee_q, then q accessed v in E. Notice that $writer(v, E) \neq p$, otherwise $p, q \in Accessed(v, E) \cap Act(E)$ and by IN5 $writer(v, E) = p \notin INV$, a contradiction. The only events removed are by p, therefore we have $writer(v, E') = writer(v, E)$. Since f_q is critical event in Ef_q, we have $writer(v, E) \neq q$. Altogether $writer(v, E') \notin \{p, q\}$, hence e_q is a critical write in $E'e_q$ as well.

For the general case we prove the claim by induction on $|Y|$. The base case $|Y|= 1$ has just been proven. Assume we proved the claim for any $|Y|= n$, and consider $Y \subseteq INV$ such that $|Y|= n+1$. Fix an arbitrary $p \in Y$, and denote $Z = Y \setminus \{p\}$. Then $Z \subseteq INV$ and $|Z|= n$. Denote $E_Z = E^{-Z}$, then by induction hypothesis:

1. E_Z is an execution;
2. $Act(E_Z) = Act(E) \setminus Z$, and $Fin(E_Z) = Fin(E)$;
3. $INV \setminus Z$ is an IN-set of E_Z;
4. Each $q \in Act(E_Z)$ executed the same critical events in E_Z and in E;
5. If $q \in Act(E_Z)$ is about to execute a special event f_q after E, then q is about to execute a special event $f'_q \sim f_q$ after E_Z.

Notice that $E' = E_Z^{-p}$ and $p \in INV \setminus Z$, thus using the induction base with E_Z and $\{p\}$ we get:

– E' is an execution;
– $Act(E') = Act(E_Z) \setminus \{p\} = Act(E) \setminus Y$, and $Fin(E') = Fin(E_Z) = Fin(E)$;
– $(INV \setminus Z) \setminus \{p\} = INV \setminus Y$ is an IN-set of E';
– Each $q \in Act(E')$ executed the same critical events in E' and in E_Z, and thus the same critical events in E' and in E.
– If $q \in Act(E') \subseteq Act(E_Z)$ is about to execute a special event f_q after E, then q is about to execute a special event $f'_q \sim f_q$ after E_Z, thus q is about to execute a special event $e_q \sim f'_q \sim f_q$ after E'.

□

Lemma 5 (repeated) *Let E be a regular execution. Then there exists an extension F such that the following hold:*

– *F contains no special events in EF;*
– *EF is a regular execution;*
– *Each $p \in Act(E)$ is about to execute a special event f_p after EF. Moreover, at most one process $p \in Act(E)$ is about to execute $f_p = CS_p$ after EF.*

Proof. First we prove the following claim: Consider $p \in Act(E)$. Then there is a solo extension F_p by p such that F_p contains no special event in EF_p, and p is about to execute a special event f_p after EF_p.
We let p run solo after E until p's first special event f_p. Assume towards a contradiction that such an extension does not exist, i.e. p executes an infinite run F_p after E such that F_p contains no special event in EF_p (notice that p cannot finish running without executing the special event $Exit_p$). For any finite prefix H of F_p, H contains no special events in EH, thus $Act(EH) = Act(E)$, and by Lemma 3 $Act(EH)$ is an IN-set of EH. Denote $Y = Act(EH) \setminus \{p\}$. Using Lemma 4 with $'E' \leftarrow EH$, $'INV' \leftarrow Act(EH)$ and $'Y' \leftarrow Y$, we have an execution $E_p = (EH)^{-Y}$ such that $Act(E_p) = Act(EH) \setminus Y = \{p\}$. Since H is a solo-execution by $p \notin Y$, E_p can be written as $E^{-Y}H$. Therefore p executes a solo-execution H after E^{-Y}, where p is the only active process along H, and p does not finish a passage in H. Since this holds for any prefix of F_p, H can be as long as we wish, contradicting the global progress property.

Denote $Act(E) = p_1, p_2, \ldots, p_n$. We prove by induction on $i = 0, 1, 2, \ldots, n$ that there is an extension F_i of E such that:

1. F_i contains no special events in EF_i;
2. EF_i is a regular execution;
3. For every $1 \leq j \leq i$, p_j is about to execute a special event f_{p_j} after EF_i;

The base case $i = 0$ is trivial ($F_0 = \langle\rangle$). Assume we already constructed F_i as above for $i < n$. We now construct F_{i+1}:
Denote $p = p_{i+1}$. F_i contains no transition events, thus $Act(EF_i) = Act(E)$, and as a result $p \in Act(EF_i)$. Since EF_i is a regular execution, we can use the claim proven above for a single process: there is an extension F_p by p such that F_p contains no special events in EF_iF_p, and p is about to execute a special event f_p after EF_iF_p. Denote $F_{i+1} = F_iF_p$, then the following hold:

1. F_{i+1} contains no special events in EF_{i+1};
2. Since F_{i+1} contains no special events, we have $Act(EF_{i+1}) = Act(E)$, and using Lemma 3 we have that $Act(E)$ is an IN-set of EF_{i+1}, therefore EF_{i+1} is a regular execution;
3. We already know that p is about to execute a special event f_p after EF_{i+1}. Consider $q = p_j$ for some $j \leq$

i. By our assumption q is about to execute an event f_q after EF_i, and since $(EF_i) \mid q = (EF_{i+}) \mid q$, it is about to execute f_q after EF_{i+1}. As f_q is a special event in $EF_i f_q$, and $(EF_{i+1} f_q) \mid q = (EF_i f_q) \mid q$, by claim 1 f_q is a special event in $EF_{i+1} f_q$.

Substituting i with n we get an extension F such that:

- F contains no special events in EF;
- EF is a regular execution;
- Each $p \in Act(E)$ is about to execute a special event f_p after EF.
  At most one process $p \in Act(E)$ is about to execute $f_p = CS_p$, otherwise there are two distinct processes about to execute CS event after EF, contradicting the exclusion property.

□