Trading Fences with RMRs and Separating Memory Models[*] (Extended Abstract)

Hagit Attiya
Dept. of Computer Science
Technion
hagit@cs.technion.ac.il

Danny Hendler
Dept. of Computer Science
Ben-Gurion University
hendlerd@cs.bgu.ac.il

Philipp Woelfel
Dept. of Computer Science
University of Calgary
woelfel@cpsc.ucalgary.ca

ABSTRACT

Out-of-order execution of instructions is a common optimization technique for multicores and multiprocessors, which is governed by the *memory model* of the architecture. Relatively strong memory models, like TSO (supported by x86 and AMD), only allow reads to bypass earlier writes, while other models, like RMO (supported by ARM, POWER and Alpha) and PSO (supported by older SPARC), also allow the *reordering of writes to different locations*. These reorderings can be prevented by the use of costly *fence* instructions.

In this paper we prove that when writes can be reordered (e.g, in RMO or even PSO), there is a tradeoff between the number of fences, f, and the number of *remote memory references* (*RMR*s), r, for a large class of objects, including locks, counters and queues:

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

For example, when one of these objects is implemented using a constant number of fences (e.g., in the Bakery lock), the tradeoff implies that a linear number of RMRs is required (as indeed is the case with the Bakery lock). This gives a complexity separation between the memory models that allow write reordering and those that prohibit it, since a recent paper shows that a lock can be implemented in the stronger TSO memory model, with a small, constant number of fences, and a logarithmic number of RMRs.

The lower bound uses an information theoretic argument, relating the *encoding* of $n!$ distinguishable executions to the number of fences and RMRs performed in the course of these executions.

We also present a family of algorithms matching the lower bound, which explicitly enforce the required ordering, and hence, are correct even with weak memory models. This shows that the tradeoff is tight, and indicates that for many important objects, fences are mostly needed for avoiding reordering of writes.

[*]The first author is supported by a grant from Yad-Hanadiv Foundation; the first and second authors are supported by the Israel Science Foundation (grants 1227/10 and 1749/14); the third author is partially supported by the Canada Research Chairs program, as well as the Discover Grants and Discovery Accelerator Supplements programs of the Natural Sciences and Engineering Research Council of Canada (NSERC).

Categories and Subject Descriptors

D.1.3 [**Programming Techniques**]: Concurrent Programming—*Concurrent programming*; F.1.2 [**Computation By Abstract Devices**]: Modes of Computation—*Parallelism and concurrency*

General Terms

Algorithms, Architecture

Keywords

Shared memory; fences; total store ordering

1. INTRODUCTION

Modern multicores and multiprocessors optimize code so as to execute certain instructions out of program order. The memory model supported by the architecture dictates which operation pairs can be reordered; for example, the *total store order* (*TSO*) model [13, 18] permits to perform a read from address a before an earlier write to address $b \neq a$. This reordering is also allowed in the weaker *partial store ordering* (*PSO*) model [18], which permits writes to be reordered. An even weaker model, called *relaxed memory ordering* (*RMO*) [18], allows to reorder any pair of instructions. (See Table 1 in [1] and [19, 20].)

To ensure the correctness of a concurrent algorithm, it is possible to prohibit the reordering of memory instructions, by inserting a *fence* instruction (also called a *barrier*) between them. A fence incurs significant overhead, but there are cases when it is unavoidable since it has been shown [7] that many concurrent objects, including locks (mutual exclusion), counters and queues, must ensure that there is *a read after a write to a different location*, unless strong atomic operations such as, e.g., *compare-and-swap* (CAS) are used (and these also incur significant overhead).

It is possible to acquire a lock using a constant number of fences. For example, in Lamport's well-known Bakery algorithm [17] (Algorithm 1), the acquisition of the lock (as well as its release) requires only a constant number of fences. However, acquiring the Bakery lock also requires reading a linear number of different memory locations, even when the process runs alone without interruption from other processes. Even worse, these reads are classified as *remote*, since they cannot be served from the process' local cache or memory segment. Counting the *remote memory references* (RMRs), namely, reads and writes that cannot be served from the local cache or memory segment of the process, is considered a good way to compare local-spin concurrent algorithms (see e.g., [2–5, 10, 14–16, 23]).

In this paper we prove that when writes may be reordered, a *linear* number of RMRs is needed when the number of fences is constant, showing that the Bakery lock is asymptotically optimal.

Algorithm 1: n-process Bakery lock for process i, $0 \le i < n$

```
shared $C[0, \dots, n-1]$: array of bits
shared $T[0, \dots, n-1]$: array of integers
Procedure Acquire(){
write($C[i]$,$1$) ; fence()
$tmp := 1 + \max\{T[0, \dots, n-1]\}$
write($C[i]$,$0$) ; fence()
write($T[i]$,$tmp$) ; fence()
foreach $j = 0, \dots, n-1, j \ne i$ do
    wait until $C[j] == 0$
    wait until $T[j] == 0$ or $(T[i], i) < (T[j], j)$
end
}
Procedure Release(){
write($T[i]$,$0$) ; fence() }
```

That is, one cannot win on both the fence and RMR complexities of read/write algorithms for many fundamental objects, including locks, counters and queues. This is a special case of a general result we prove: if f and r denote the numbers of fences respectively RMRs performed per passage (one acquisition and release of the lock), then in the worst-case

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

Hence, to get an optimal, logarithmic number of RMRs per passage the number of fences per passage must also be logarithmic, matching the complexity of the tournament-tree lock [23].

When writes are performed in program order, there is a lock in which each passage incurs a *constant* number of fences and a *logarithmic* number of RMRs [8] (which is optimal [9]). This gives an exponential separation between models without reordering of writes (like TSO) and weaker models in which write reordering is allowed (like PSO, Power and RMO).

Our lower bound holds for the two standard theoretical models for measuring RMR complexity: the *Distributed Shared Memory* (DSM) and the *Cache Coherent* (CC) model. In the DSM model, shared memory is partitioned into segments, and each segment is local to one process. Only accesses to non-local memory segments incur RMRs. In the CC model, processes are equipped with caches, and RMRs correspond to cache-misses. Contrary to, e.g., the tight RMR lower bound in [9], our lower bound proof is for a shared memory model combining the power of the DSM and the CC models. Specifically, processes are equipped with caches, and an access of a memory location by a process incurs an RMR only if the location is not local to that process *and* it is a cache miss. Thus, our tradeoff holds for modern NUMA architectures with caches.

The tradeoff is tight and we present a family of algorithms that trade the number of fences with the number of RMRs: for every f, $1 \le f \le \log n$, a passage incurs $O(f)$ fences and $O(fn^{1/f})$ RMRs. These algorithms are correct in any memory model, since they explicitly order reads and writes with fences.

To prove the tradeoff, we construct an execution E_π for each permutation π of $[n] := \{0, \dots, n-1\}$, and we encode these executions so that *the code length of an execution is a function of both the number of fences and the number of RMRs* incurred in it.

In every execution E_π, processes access the shared object once (for example, to increment a counter or acquire the lock) according to their order in π. Each execution E_π is encoded by a sequence of *commands*, some of which receive integer parameters, so that, roughly, the number of commands in the encoding is linear in the total number of fences performed in the execution, denoted $\beta(E_\pi)$, while the total sum of parameter values is linear in the number of RMRs, denoted $\rho(E_\pi)$. As we show, the length of the encoding is bounded from above by $\beta(E_\pi)(\log \frac{\rho(E_\pi)}{\beta(E_\pi)} + 1)$.

In E_π, processes are unaware of processes that appear after them in the permutation π. As we prove, this ensures that π can be reconstructed from E_π. Since an execution code uniquely determines an execution (and therefore a permutation over $[n]$), there are $n!$ different execution codes. Therefore, $\beta(E)(\log \frac{\rho(E)}{\beta(E)} + 1)$, for at least one of these executions, is asymptotically at least logarithmic in $n!$, that is, in $\Omega(n \log n)$.

Our proof significantly extends the technique used to prove the $\Omega(n \log n)$ lower bound on the RMR complexity of mutual exclusion [9, 11]. That proof also goes by encoding executions corresponding to different permutations in a way that allows to uniquely reconstruct them, but is only able to capture the RMR complexity. In contrast, the codes constructed by our proof capture both fence and RMR complexities: A sequence of w write operations issued by a process p may be delayed by the system and committed to memory only when a fence is performed. When writes can be reordered, these writes can be committed *in any order*, and our encoding technique exploits this freedom to encode a *batch of writes* in its entirety by using a constant number of commands, instead of encoding each write separately. These commands encode the exact manner in which the writes in a batch should be interleaved within the steps of preceding processes. Some of these commands receive a parameter that represents the number of write steps to which they apply. Thus, the values of these parameters may be encoded by using $O(\log w)$ bits. Each fence, on the other hand, is encoded using a constant number of bits.

Very informally, we capture the tradeoff between fences and RMRs as follows. The number of code-bits representing fences changes linearly with the number of fences ($\beta(E_\pi)$). However, as the number of fences decreases, the average size of write batches $\left(\frac{\rho(E_\pi)}{\beta(E_\pi)}\right)$ increases. If the number of code bits has to remain in $\Theta(n \log n)$, a linear decrease in the number of fences implies a linear increase in $\log \frac{\rho(E_\pi)}{\beta(E_\pi)}$, which is an exponential increase in the number of RMRs.

2. MODEL

We consider an asynchronous shared memory system where n processes with IDs $0, \dots, n-1$ communicate by accessing shared registers from a totally ordered set $\mathcal{R}$ with values from a domain $\mathcal{D} \supseteq \{\sqcup\}$. For simplicity but without loss of generality let $\mathcal{R} = \mathcal{D} - \{\sqcup\} = \mathbb{N}$. Initially, each register has value $\sqcup$.

Each process p is equipped with a *write-buffer* [21] that stores an (initially empty) unordered set $WB_p \subseteq \mathcal{R} \times \mathcal{D}$ (without duplicates). Processes execute an algorithm A in which they perform the operations `read(`R`)`, `write(`R, x`)` or `fence()`, where $(R, x) \in \mathcal{R} \times \mathcal{D}$. A `write(`$R, x$`)` operation by process p writes (R, x) to p's write-buffer, replacing WB_p with $(WB_p - \{(R, x') \mid x' \in \mathcal{D}\}) \cup \{(R, x)\}$. If p executes `read(`R`)` at a time when there is a write (R, x) in WB_p, then the read is *served from the write-buffer* and the value x is returned; if no such pair is in WB_p, then the value of register R is returned, and thus the read is *served from shared memory*. At any point in time, if WB_p contains some write (R, x), then that write may be *committed to shared memory* (by the system), meaning that (R, x) is removed from WB_p, and the value of R changes to x. A `fence()` operation does not affect the shared memory or write-buffers directly; it only guarantees that the system does not allow the calling process to take any further steps before its write-buffer is empty.

To facilitate our proofs, we assume w.l.o.g. that there is a special `return()` operation that takes one *return* value, a non-negative integer. Each process has to execute `return()` exactly once, and after executing `return(x)` the process enters a *final state with value x*.

An *execution* is a (possibly infinite) sequence of *steps* executed by processes. Each step is either a *read, write, fence, or return step* that happens when the process executes a `read()`, `write()`, `fence()`, or `return()` operation, respectively; or it is a *commit* of a write from p's write-buffer (to shared memory). Unlike other step types, commit steps are not instructions in a process' program and their position in the execution is controlled by the system; for notational convenience, we regard them as process steps.

A process p *participates* in an execution E, if E contains at least one step by p. If only processes $p_1, \dots, p_k$ participate in execution E, then we say E is an execution *by* $p_1, \dots, p_k$. An execution is *complete*, if it contains a return step by each participating process. For any execution E and any set $P \subseteq [n]$ of processes, $E|_P$ denotes the execution obtained from E by removing all steps by processes in $[n] - P$.

A *system configuration* is the state of the entire system, i.e., it comprises the state of each process, each register, and each write-buffer. The *initial* system configuration is denoted C_{init}. For system configuration C, $next_p(C)$ denotes the operation that process $p \in [n]$ is poised to execute; we write $next_p(C) = \emptyset$, if in C process p is in a final state. Process p's write-buffer in system configuration C is denoted $WB_p(C)$, and the value stored in register $R \in \mathcal{R}$ is denoted $R(C)$. An execution E is *permissible by algorithm A starting from system configuration C* if E may result as processes execute algorithm A starting from C. If E is permissible by A starting from the initial system configuration, C_{init}, then we say it is *A-permissible*. A system configuration C is *reachable*, if there exists an A-permissible execution E, such that C is reached from C_{init} by executing E.

A *schedule* is a (possibly infinite) sequence $\sigma = (\sigma_1, \sigma_2, \dots)$, where each element σ_i in the schedule is a pair in $[n] \times (\mathcal{R} \cup \{\bot\})$. A schedule σ, together with an algorithm A and a system configuration C, uniquely determines an execution $Exec_A(C;\sigma)$ as follows: If σ is the empty schedule, then $Exec_A(C;\sigma)$ is the empty execution. Now suppose σ contains exactly one element $(p,R) \in [n] \times (\mathcal{R} \cup \{\bot\})$. If p is in a final state in C, i.e., $next_p(C) = \emptyset$, then again, $Exec_A\big(C;(p,R)\big)$ is the empty execution. Now suppose $next_p(C) \neq \emptyset$. Then $Exec_A\big(C;(p,R)\big)$ consists of a single step s determined as follows:

- If there is a value $x \in \mathcal{D}$ such that $(R,x) \in WB_p(C)$, then s is the step that commits (R,x) (clearly this is only possible if $R \neq \bot$).
- Otherwise, if $next_p(C)$ is a `fence()` operation and $WB_p(C) \neq \emptyset$, then s is a commit step in which write $(R,x) \in WB_p(C)$ gets committed, where R is the register with the smallest identifier for which there is a write in $WB_p(C)$. (Recall that $\mathcal{R}$ is totally ordered.)
- In all other cases, s is a step that corresponds to operation $next_p(C)$, i.e., s is a read, write, fence, or return step, but not a commit step.

Each step in an execution E will be defined as either *local* or *remote*. Our definition is such that every remote step in E corresponds to an RMR in both the DSM and the CC models (although not necessarily vice versa). Therefore, our lower bound applies for both the CC and DSM models. The set $\mathcal{R}$ of all registers is partitioned into n *memory segments*, $\mathcal{R}_0, \dots, \mathcal{R}_{n-1}$, each of infinite size. In the following we assume (w.l.o.g.) for presentation simplicity that in any execution, all arguments of `write()` operations are distinct. A `read(`R`)` step by process p is local, if either $R \in \mathcal{R}_p$, or the read operation returns value x and p has previously executed `write(`R, x`)` or read the value x from R; all other `read()` steps are remote. All `write()` and `fence()` steps are local. A commit of a write (R,x) is local, if either $R \in \mathcal{R}_p$, or p previously committed a write $(\cdot, R)$ to shared memory, and since then no other process committed a write $(\cdot, R)$; all other commits are remote. A process p *accesses* process q's local memory, if p reads a register $R \in \mathcal{R}_q$, and that read is served from shared memory, or when it commits a write to $R \in \mathcal{R}_q$.

Algorithm A satisfies *weak obstruction-freedom* (see [9]), if for every process p and every reachable configuration C such that every process other than p is either in its initial or in its final state in C, p enters a final state in $Exec_A(C,\sigma)$, for every infinite $\{p\}$-only schedule σ (which contains only pairs (p,R)). Note that deadlock freedom implies weak obstruction-freedom.

3. ALGORITHMS

A *lock* supports two methods: *Acquire*, which allows a process to grab the lock, and *Release*, which is invoked after using the lock. Any lock satisfies the following requirements: (1) at most a single process has the lock in any given time (*mutual exclusion*), (2) if some process invokes *Acquire* then, eventually, some process gets the lock (*deadlock freedom*), and (3) a process completes the *Release* method in a finite number of steps (*finite exit*).

The well-known Bakery [17] and tournament-tree [22, 23] locks realize the two extreme cases of the tradeoff between the numbers of RMRs and fences. Even in an uncontended situation, Acquiring the Bakery lock (Algorithm 1) requires three writes, each followed by a fence to ensure it is visible in the shared memory, and reading the values written by all other processes, each requiring an RMR since these locations are read for the first time. Releasing the lock requires a single write (followed by a fence). (This algorithm also works under the RMO model.) Therefore, a passage through the Bakery lock requires only a constant number of fences but a linear number of RMRs. Thus, $f \log(r/f + 1) \in \Theta(\log n)$. On the other hand, a passage through a tournament-tree lock incurs $O(\log n)$ fences and $O(\log n)$ RMRs, hence, $f \log(r/f + 1) \in \Theta(\log n)$.

To obtain intermediate points on the spectrum of the tradeoff, when the number of fences f is between a constant and $\log n$, note that Equation (1) is asymptotically matched when

$$r \in O(f \cdot n^{1/f}) \ . \qquad (2)$$

Viewing the bound in this manner immediately inspires a generalized tournament Algorithm GT_f, $1 \leq f \leq \log n$, which uses a tree of height f and branching factor $n^{1/f}$. The tree has n leaves, each one statically assigned to a single process. To win the lock in GT_f, a process has to win the locks in all the f internal nodes along the path from its leaf to the root. In each node, there are (at most) $n^{1/f}$ processes, which compete using a Bakery lock for $n^{1/f}$ processes. Clearly, GT_1 is the Bakery algorithm and $GT_{\log n}$ is essentially the binary tournament-tree algorithm. (See Figure 1.)

The resulting number of fences per passage is therefore linear in f and the resulting number of RMRs per passage is $O(f \cdot n^{1/f})$. Thus, the numbers of fences and RMRs satisfy Equation (2), showing that the bound of Equation (1) is tight.

4. ORDERING ALGORITHMS AND OVERVIEW OF THE MAIN RESULT

The lower bound holds for a general class of objects, for which all algorithms must order processes in specific, clean executions. Informally, an algorithm is *ordering* if for any permutation $\pi = (p_0, \dots, p_{n-1})$ over $[n]$ and execution E by $\{p_0, \dots, p_m\}$, $0 \leq$

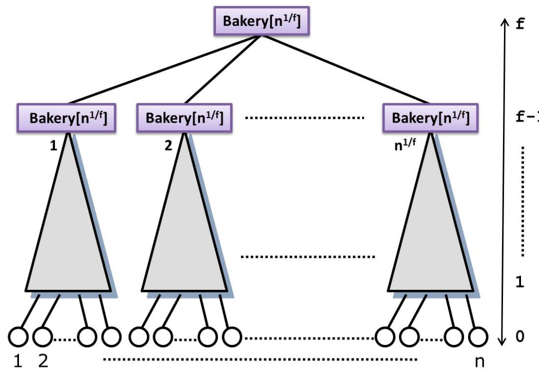

Figure 1: Schematic view of algorithm GT_f; Bakery[$n^{1/f}$] denotes the Bakery lock for $n^{1/f}$ processes.

$m < n$, if processes $\{p_0, \ldots, p_{m-1}\}$ are unaware of p_m in E,[1] and each process p_i returns i, $0 \le i < m$, then p_m returns m in E. In particular, we can prove by induction that if the execution is *sequential*, namely, each process p_{i-1}, $0 < i \le m$, returns before p_i starts, then each process p_i returns i.

For a permutation $\pi = (p_0, \ldots, p_{n-1})$ over $[n]$, let $[\pi]_k$ denote the sequence $(p_0, \ldots, p_{k-1})$ of length k.

DEFINITION 4.1. *Algorithm A is* ordering*, if for any permutation $\pi = (p_0, \ldots, p_{n-1})$ over $[n]$, any $k \in [n]$, and any complete A-permissible execution E by $\{p_0, \ldots, p_k\}$: if $E|_{\{p_0,\ldots,p_{k-1}\}}$ is A-permissible and each process p_i, $i \in [k]$ returns value i in E, then p_k returns value k in E.*

The following algorithm, `Count`, is a simple ordering algorithm that uses a single lock: Upon entering the critical section, each process reads a shared register C (initialized to 0), adds one to it, and writes back the result, followed by a fence; the process then exits the critical section and returns the value read from C. The sequence of values returned by processes (and the associated sequence of process identifiers) is a permutation, and it is not hard to see that `Count` is ordering. Clearly, the numbers of fences and RMRs in `Count` are asymptotically the same as those of a single passage through the lock it uses. More details appear in the full version of the paper, which also shows that other objects—queue, counter and fetch-and-increment—may be similarly used for constructing ordering algorithms. Thus, our tradeoff holds for implementations of these objects.

We capture the tradeoff between the *total* number of fences and RMRs in an execution, in which each process accesses the object exactly once (e.g., each process acquires and releases the lock once). Recall that $\rho(E)$, $\beta(E)$, respectively, denote the total number of RMRs and fences performed by the processes in such an execution E. The next theorem states our main result.

THEOREM 4.2. *If A is an ordering algorithm, then it has an execution E, in which every process executes the algorithm once, and $\beta(E)(\log(\rho(E)/\beta(E)) + 1) \in \Omega(n \log n)$.*

Dividing by n implies that at least for one process in E, the tradeoff between the number of fences, f, and number of RMRs, r, satisfies $f\big(\log(r/f) + 1\big) \in \Omega(\log n)$.

[1] Formally, this means that the execution E projected on $\{p_0, \ldots, p_{k-1}\}$ is also an execution of the algorithm.

command	meaning
proceed	take steps until there is a fence with non-empty write buffer
commit	commit all write steps
wait-hidden-commit(k)	wait until k write steps are hidden by earlier processes
wait-read-finish(k)	wait for k early processes that read a write to complete
wait-local-finish(k)	wait for k early processes that access local memory to complete

Table 1: Commands used in the encoding and their meaning.

Recall that we prove the tradeoff by constructing an execution E_π, for each permutation π of $[n] := \{0, \ldots, n-1\}$, and encoding these executions so that the code length of an execution is a function of the number of fences and the number of RMRs incurred in it.

We start with an informal description of how, given a permutation $\pi = (p_0, p_1, \ldots, p_{n-1})$, we construct and encode a unique execution. Our construction ensures that no process is aware of the participation of later processes (that follow it in π) in the execution, that is, it maintains the property that process p_i is unaware whether a later process in the permutation, p_j, $j > i$, participates in E_π. Since the algorithm is ordering, this allows to prove that each process p_i returns the value i in E_π. It follows that the sequence of values returned in E_π and $E_{\pi'}$ are different for $\pi \neq \pi'$, implying that π can be uniquely determined from E_π.

Each execution E_π is encoded by a sequence of *commands*, with possible parameters. Roughly speaking, the number of commands in the encoding is in $O(\beta(E_\pi))$, while the total sum of parameter values is in $O(\rho(E_\pi))$. We prove that E_π is uniquely encoded using $B(E_\pi) = O\Big(\beta(E_\pi)\big(\log(\rho(E_\pi)/\beta(E_\pi)) + 1\big)\Big)$ bits. Since the $n!$ different executions must have $n!$ different codes, the function $B(E_\pi)$ in at least one of these executions is asymptotically at least $\log n!$, that is, in $\Omega(n \log n)$.

Executions are constructed and *encoded* iteratively (see Section 5.2). In the first iteration, the steps of process p_0 (as it runs solo) are determined and encoded. Since p_0 is unaware of other processes, it does not have to wait for any other process. Then, the steps of process p_1 are determined and encoded. These steps are interleaved within the steps of p_0, so process p_1 may have to wait for process p_0 (we soon explain when). We proceed in this manner, until the steps of all processes have been incorporated into execution E_π, and we have created *command stacks* $St_0, \ldots, St_{n-1}$, where, for $j \in [n]$, St_j encodes the steps of process p_j and, collectively, $St_0, \ldots, St_{n-1}$ encode the manner in which steps by different processes should be interleaved.

Assume we have constructed execution E^i_π of processes $p_0, \ldots, p_i$. Roughly, as long as p_{i+1} does not issue a fence step when its write-buffer is non-empty, it may continue taking steps. The resulting sequence of steps is encoded by a single *proceed* command. When p_{i+1} issues a fence step and its write-buffer is non-empty, then the batch of writes to be committed to shared-memory is encoded by three commands, at most. The *wait-hidden-commit*(k) command means that k write-steps by p_{i+1} should be *hidden* (i.e., overwritten before being read) before other write-steps may be committed; the *wait-read-finish*(k) command means that there are k early processes that must complete their execution before p_{i+1} is allowed to commit writes to registers read by these processes; finally, the *commit* command means that p_{i+1} can commit the rest of the writes in its buffer (if any) and proceed with its execution. (Table 1 summarizes the commands.)

In more detail, this is how we interleave the steps of process p_{i+1} into execution E^i_π and construct E^{i+1}_π.

- Before process p_{i+1} is allowed to take its first step, it has to wait for all earlier processes that access its local memory segment in E^i_π to terminate. This is encoded by the *wait-local-finish* command, whose parameter is the number of processes for whose completion p_{i+1} must wait.
- As long as process p_{i+1} performs read, write, or fence instructions that have no impact (i.e., fences issued when the write-buffer of p_{i+1} is empty), then p_{i+1} may proceed taking steps. This is encoded by the parameterless *proceed* command. (To avoid infinite executions, a process is not scheduled to take such steps if it is in a state where it would not enter a final state when running by itself.)
- Otherwise, p_{i+1}'s next step is a fence instruction and its write-buffer is not empty. This is the only case in which writes by p_{i+1} are committed to shared memory. In this case, the batch of write steps taken by p_{i+1} since its preceding fence (or since it began its execution) is collectively encoded by using a constant number of commands according to the following principles:
  – Whenever possible, writes about to be committed by p_{i+1} are scheduled so they will be hidden by writes committed in E^i_π by earlier processes before they are read. Upon encoding a batch of writes to be committed by p_{i+1}, the number of writes that may be overwritten is encoded as a parameter of the *wait-hidden-commit* command. These writes will be committed before any other writes in p_{i+1}'s write-buffer and will be interleaved in the execution so that they are hidden.
  – If there are no writes that may be hidden in p_{i+1}'s write-buffer, but there are writes to registers from which an earlier process reads in E^i_π, then the number of such early processes is a parameter of the *wait-read-finish* command. Process p_{i+1} will commit these writes only after these early processes terminate their execution.
  – Otherwise, if p_{i+1}'s write-buffer is not empty, then the writes in it should be committed. This is encoded by the parameterless *commit* command.
  – Otherwise, p_{i+1} can take its next step. This is encoded by the parameterless *proceed* command.

5. PROOF OF THE LOWER BOUND

Fix an ordering algorithm A and assume w.l.o.g. that every process executes a `fence()` operation just before it enters a final state, i.e., before its `return()` operation. (Since the proof fixes algorithm A, in the following we write permissible instead of permissible by A, and omit A from the notation defined earlier.)

Somewhat counter-intuitively, we first describe the decoding of executions (Section 5.1). The reason is that the encoding, which is done inductively (in Section 5.2), relies on interpreting the decoding of earlier parts of the encoding.

5.1 Codes: Commands and Their Decoding

We now describe codes and how they are *decoded*, that is, how they are used to construct an execution. The codes use the following *commands*: *commit*, *proceed*, *wait-hidden-commit*(k), *wait-local-finish*(k, S), and *wait-read-finish*(k, S), where $k \in \mathbb{N}$ and $S \subseteq [n]$ are parameters.

Each process p has a *command stack* St_p, which stores a sequence of commands. Let $top(St_p)$ denote the element on top of stack St_p. An *extended configuration* is an $(n + 1)$-tuple $\Gamma = (C; St_0, \ldots, St_{n-1})$, where C is a system configuration and $St_0, \ldots, St_{n-1}$ are command stacks. If $\vec{S} = (St_1, \ldots, St_n)$ is a command stack sequence and $q \in [n]$, then $\vec{S}|_q$ is process q's stack St_q.

We now show how an extended configuration $\Gamma = (C; St_0, \ldots, St_{n-1})$ uniquely determines an execution $E(\Gamma)$, which is permissible starting from system configuration C. From Γ, we determine a zero- or one-step schedule, σ, and a new sequence $(St'_0, \ldots, St'_{n-1})$ of command stacks. $E(\Gamma)$ is defined recursively. If σ is a zero-element schedule, then $E(\Gamma)$ is the empty execution. Otherwise, σ is a one-element schedule (representing a step by some process i); let C' be the system configuration reached at the end of execution $Exec(C; \sigma)$, then:

$$E(\Gamma) = Exec(C; \sigma) \circ E(C'; St'_0, \ldots, St'_{n-1}),$$

where St'_i, $i \in [n]$, is a new command stack obtained by popping elements from and/or pushing elements to St_i. Our recursive definition does not guarantee that $E(\Gamma)$ is finite for every Γ, i.e, the recursion may not terminate; however, the codes Γ we will construct later will only yield finite executions $E(\Gamma)$. To completely define $E(\Gamma)$, it suffices to state for each extended configurations Γ whether $E(\Gamma)$ is the empty execution, and otherwise to define σ and $St'_0, \ldots, St'_{n-1}$ as a function of Γ.

We partition the set of processes into sets of *finished*, *commit enabled*, *non-commit enabled*, and *waiting* processes, defined as follows. Process p is *finished* in configuration C, if it is in a final state in C; $NbFinal(C)$ denotes the number of processes that are finished in system configuration C. Process p is *commit enabled* in $(C; St_0, \ldots, St_{n-1})$, if

$$top(St_p) = commit \wedge next_p(C) = \texttt{fence()} \wedge WB_p(C) \neq \emptyset.$$

Process p is *non-commit enabled* if $top(St_p) = proceed$, p enters a final state in every p-only execution from C, and one of the following holds:

- $next_p(C)$ is a `read()` or `write()` operation;
- $next_p(C)$ is `return(`r`)`, where $r = NbFinal(C)$; or
- $next_p(C)$ is `fence()` and $WB_p(C) = \emptyset$.

Process p is *waiting* in all other cases (it is not finished, commit enabled, or non-commit enabled).

Decoding Rule (D1): Commit Step. Suppose in configuration C with command stacks $St_0, \ldots, St_{n-1}$ there is at least one commit enabled process. Let p be the one with the smallest ID. Let R be the smallest register such that (R, x) is in $WB_p(C)$. If there are processes q' such that $top(St_{q'}) = wait\text{-}hidden\text{-}commit(k)$, $k > 0$, and $WB_{q'}(C)$ contains a write to R, then let q be the one with the smallest ID and let $p^* = q$. Otherwise, let $p^* = p$. Then we define $\sigma = (p^*, R)$. I.e., in $Exec(C; \sigma)$, p^* commits its write to R.

A commit step is *hidden* if executed by a waiting process p^*; otherwise, the commit step is *visible*.

The command stacks St'_i, $i \in [n]$ are determined as follows.

(D1a) For $i = p^*$, if $top(St_{p^*}) = commit$ (i.e., $p^* = p$) and $|WB_p(C)| = 1$, then we obtain St'_{p^*} from St_{p^*} by simply popping the top element from the stack.
(*Informally: a commit of a batch of writes from the write-buffer of p was now completed.*)

(D1b) For $i = p^*$, if $top(St_{p^*}) = wait\text{-}hidden\text{-}commit(k)$ (i.e., $p^* = q$), then we pop that element from the stack, and if $k - 1 > 0$, we push $wait\text{-}hidden\text{-}commit(k - 1)$ back on the stack.
(*Informally: another write commit by q is about to be hidden - by p. If additional write commits must be hidden, then push* $wait\text{-}hidden\text{-}commit(k-1)$ *back to q's stack, other-*

wise no more of q's commits in the current batch should be hidden.)

(D1c) For $i \neq p^*$, if $R \in \mathcal{R}_i$, $i \in [n]$, and $top(St_i) = \textit{wait-local-finish}(\ell, S)$, then we obtain St'_i from St_i by replacing that top element with $\textit{wait-local-finish}(\ell, S \cup \{p^*\})$.
(*Informally: if there is a process i $\neq p^*$ that is waiting for earlier processes who access its local segment to terminate, then i should wait for the termination of p^*.*)

(D1d) All other command stacks remain unchanged.

Decoding Rule (D2): Read, Write, Return, or Fence Step. Now suppose there is no commit enabled process, but there is at least one non-commit enabled one. Let p be the non-commit enabled process with smallest ID. Then σ is $(p, \perp)$, i.e., $Exec(C; \sigma)$ is the permissible one-step execution in which p executes its read, write, return, or fence step. We now show how to determine St'_i for all $i \in [n]$. Recall that C' is the system configuration reached at the end of $E(C; \sigma)$. First, we consider process p.

(D2a) If $next_p(C') \in \{\texttt{fence()}, \texttt{return()}, \emptyset\}$, then we obtain St'_p from St_p by popping the *proceed* command from the top of St_p. Otherwise, $St'_p = St_p$.
(*Informally: if the next step of p is read or write, then the top of p's stack remains proceed. Otherwise, the next command (if any) reaches the top of p's stack.*)

For each other process $q \in [n] - \{p\}$, the command stack St'_q is obtained from St_q as follows.

(D2b) If $Exec(C; \sigma)$ is a return step and $top(St_q)$ is $\textit{wait-read-finish}(k, S)$ or $\textit{wait-local-finish}(k, S)$, where S contains p, then we pop the top element from the stack, and if $k-1 > 0$ we push $\textit{wait-read-finish}(k-1, S)$ (respectively $\textit{wait-local-finish}(k-1, S)$) back on the stack.
(*Informally: if p's execution terminated, then every process q that waits for p to terminate no longer needs to wait for it. If q needs to wait for the termination of additional processes, then parameter k is updated to k − 1, otherwise the next command in q's stack becomes effective.*)

(D2c) If $top(St_q)$ is $\textit{wait-read-finish}(k, S)$ and $Exec(C; \sigma)$ is a read step in which process p reads register R from shared memory (not the write buffer), where $R \in WB_q(C)$, then we replace the top element of St_q with $\textit{wait-read-finish}(k, S \cup \{p\})$.
(*Informally: if there are processes q $\neq$ p that are about to commit a write to R and are waiting for the termination of earlier processes who read a register they are about to write to, then each such process should wait for the termination of p.*)

(D2d) If $top(St_q)$ is $\textit{wait-local-finish}(k, S)$ and in $Exec(C; \sigma)$ process p reads a register from $\mathcal{R}_q$, then we replace the top stack element with $\textit{wait-local-finish}(k, S \cup \{p\})$.
(*Informally: if there is a process q that is waiting for earlier processes who access its local segment to terminate, then q should wait for the termination of p.*)

(D2e) In all other cases, q's command stack remains unchanged, i.e., $St'_q = St_q$.

Decoding Rule (D3): End of Execution. If all processes are waiting or finished, $E(\Gamma)$ is the empty execution.

5.2 Encoding an Execution

Recall that C_{init} is the initial system configuration, and fix a permutation $\pi = (p_0, \ldots, p_{n-1})$ over $[n]$. We construct inductively command stack sequences $\vec{S}_0, \vec{S}_1, \vec{S}_2, \ldots$, where the base case is $\vec{S}_0 = (\emptyset, \ldots, \emptyset)$.

For the inductive step, suppose we have constructed $\vec{S}_i$ for some integer $i \geq 0$. Let C_i be the system configuration reached at the end of $E_i = E(C_{init}; \vec{S}_i)$. In other words, C_i is the configuration reached after the execution determined by $(C_{init}; \vec{S}_i)$ is constructed (decoded) according to the decoding rules specified in Section 5.1. If in configuration C_i process p_{n-1} is in a final state, the inductive construction ends. Otherwise, let τ_i be the largest index in $[n]$ such that $\vec{S}_i|_{p_{\tau_i}} \neq \emptyset$, and $\tau_i = -1$ if no such index exists. We let

$$\ell = \begin{cases} \tau_i + 1 & \text{if } \tau_i = -1 \text{ or if } p_{\tau_i} \text{ is in a final state in } C_i\text{; and} \\ \tau_i & \text{otherwise.} \end{cases} \tag{3}$$

We obtain $\vec{S}_{i+1}$ from $\vec{S}_i$ by adding exactly one command cmd_{i+1} to the *bottom* of process p_ℓ's command stack; all other command stacks remain unchanged. I.e., $\vec{S}_{i+1}|_{p_\ell}$ is $\vec{S}_i|_{p_\ell}$ with cmd_{i+1} added at the bottom, and for all $k \in [n] - \{\ell\}$, $\vec{S}_{i+1}|_{p_k} = \vec{S}_i|_{p_k}$. There are several cases to determine cmd_{i+1}:

Case (E1): $\vec{S}_i|_{p_\ell} = \emptyset$ and there are $\lambda > 0$ distinct processes different from p_ℓ that access registers in $\mathcal{R}_{p_\ell}$ during E_i. Then $cmd_{i+1} = \textit{wait-local-finish}(\lambda, \emptyset)$.

Case (E2): Either $\vec{S}_i|_{p_\ell} \neq \emptyset$ or no process different from p_ℓ accesses a register in $\mathcal{R}_{p_\ell}$ during E_i. We distinguish between two sub-cases.

Case (E2a): $next_{p_\ell}(C_i) \neq \texttt{fence()}$ or $WB_{p_\ell}(C_i) = \emptyset$. Then $cmd_{i+1} = \textit{proceed}$.

Case (E2b): $next_{p_\ell}(C_i) = \texttt{fence()}$ and $WB_{p_\ell}(C_i) \neq \emptyset$. Let E_i^* be the prefix of E_i that ends when p_ℓ's command stack is empty for the first time, and E_i^{**} the postfix of $E(C_{init}; \vec{S}_i)$ such that $E_i = E(C_{init}; \vec{S}_i) = E^* \circ E^{**}$. (Lemma 5.1 (I6) below implies that E_i^* always exists.) Let γ be the number of distinct registers appearing in $WB_{p_\ell}(C_i)$, to which at least one process commits a write during E^{**}. Let ζ be the number of distinct processes that read during E^{**} at least one register that appears in $WB_{p_\ell}(C_i)$. Then

$$cmd_{i+1} = \begin{cases} \textit{wait-hidden-commit}(\gamma) & \text{if } \gamma > 0; \\ \textit{wait-read-finish}(\zeta, \emptyset) & \text{if } \gamma = 0 \text{ and } \zeta > 0\text{; and} \\ \textit{commit} & \text{if } \gamma = \zeta = 0. \end{cases}$$

The next lemma summarizes several important structural properties of the stack sequences $\vec{S}_i$ and the corresponding executions E_i. Its proof, by induction, is omitted due to space restrictions.

LEMMA 5.1. *The following properties hold:*

(I1) For all $k \in [n]$, $\vec{S}_i|_{p_k} = \emptyset$ if and only if $k > \tau_i$.

(I2) For all $k \in [n]$, in system configuration C_i process p_k is in a final state if $k < \tau_i$, it is in its initial state if $k > \tau_i$, and if p_k is in a final state, then the value of that state is k.

(I3) $E(C_{init}; \vec{S}_i)|_{[n]-\{p_{\tau_i}\}} = E(C_{init}; \vec{S}_{i-1})|_{[n]-\{p_{\tau_i}\}}$, if $i \geq 1$.

(I4) For all $r \in [n]$, each stack $\vec{S}_i|_{p_r}$ contains at most one $\textit{wait-local-finish}()$ command, and only at the top of the stack.

(I5) Suppose $\vec{S}_i|_{p_r} \neq \emptyset$, $r \in [n]$. Then $\vec{S}_i|_{p_r}$ contains a $\textit{wait-local-finish}(\lambda)$ command if and only if there are exactly

λ processes in $\{p_1,\ldots,p_{r-1}\}$, that access registers in $\mathcal{R}_{p_r}$ during $E(C_i;\vec{S}_i)$.

(I6) Execution $E(C_{init};\vec{S}_i)$ is finite and when it ends, p_{τ_i}'s command stack is empty.

(I7) For each $k \in [n]$, $E_i|_{\{p_0,\ldots,p_k\}} = E(C_{init};\vec{S}_i^{(k)})$, where $\vec{S}_i^{(k)} = (\vec{S}_i|_{p_0},\ldots,\vec{S}_i|_{p_k},\emptyset,\ldots,\emptyset)$. In particular, $E_i|_{\{p_0,\ldots,p_k\}}$ is permissible starting from C_{init}.

(I8) If the bottom element of $\vec{S}_i|_{p_{\tau_i}}$ is wait-read-finish$(\zeta,\emptyset)$ for some $\zeta \in \mathbb{N}$, and $(C_i^{\emptyset};\vec{S}_i^{\emptyset})$ is the extended configuration reached, when p_{τ_i}'s stack is empty for the first time during $E(C_{init};\vec{S}_i)$, then $next_{p_{\tau_i}}(C_i^{\emptyset}) =$ `fence()`, $WB_{p_{\tau_i}}(C_i^{\emptyset}) \neq \emptyset$, and throughout $E(C_i^{\emptyset};\vec{S}_i^{\emptyset})$ no process other than p_{τ_i} accesses a register that appears in $WB_{p_{\tau_i}}(C_i^{\emptyset})$.

(I9) If the bottom element of $\vec{S}_i|_{p_{\tau_i}}$ is wait-hidden-commit(γ), $\gamma \in \mathbb{N}$, and $(C_i^{\emptyset};\vec{S}_i^{\emptyset})$ is the extended configuration reached, when p_{τ_i}'s stack is empty for the first time during $E(C_{init};\vec{S}_i)$, then $next_{p_{\tau_i}}(C_i^{\emptyset}) =$ `fence()` and throughout $E(C_i^{\emptyset};\vec{S}_i^{\emptyset})$ no process other than p_{τ_i} commits a write to a register that appears in $WB_{p_{\tau_i}}(C_i^{\emptyset})$.

(I10) If stack $\vec{S}_i$ contains a command cmd right below a wait-read-finish() command, then cmd = commit; if cmd is right below a wait-hidden-commit() command, then cmd is either a wait-read-finish(), proceed, or commit command; and if cmd is right below a commit command, then cmd is a proceed command.

Property (I7) implies that in execution E_i processes $p_0,\ldots,p_k$ are not influenced by the steps made by processes $p_{k+1},\ldots,p_{n-1}$. This is central for establishing all other properties. Properties (I4) and (I10) imply that the number of *proceed* commands on p's stack is proportional to the number of fence steps p executes.

We conclude this section with a simple claim that will be useful for the proof of our main theorem, in Section 5.3. The proof of the claim is omitted due to space restrictions.

CLAIM 5.2. *In configuration C_i all processes $p_0,\ldots,p_{\ell-1}$ are in a final state, process p_ℓ is not in a final state, and $p_{\ell+1},\ldots,p_{n-1}$ are in their initial states. In particular, $WB_{p_k}(C_i) = \emptyset$ for all $k \in [n] - \{\ell\}$.*

5.3 Proof of Theorem 4.2

In this section, we prove the lower bound stated in Theorem 4.2. For the ease of notation, whenever we refer to one of the statements (I1)-(I10), we mean the corresponding statement of Lemma 5.1.

We will show that for each permutation $\pi = (p_0,\ldots,p_{n-1})$ the inductive construction of stacks $\vec{S}_0^\pi,\vec{S}_1^\pi,\ldots$ ends eventually, i.e., there exists an $m_\pi \in \mathbb{N}$ such that in $E(C_{init};\vec{S}_{m_\pi}^\pi)$ all processes enter a final state. From (I2), in that execution each process p_k, $k \in [n]$, enters a final state with value k. Hence, $\vec{S}_{m_\pi}^\pi$ uniquely encodes permutation π, and thus there exists a permutation π so that we need at least $\Omega(n\log n)$ bits to encode all stacks of $\vec{S}_{m_\pi}^\pi$. In this section, we show how the number of commands of $\vec{S}_{m_\pi}^\pi$ and the sum of their parameter values, and thus the shortest encoding of $\vec{S}_{m_\pi}^\pi$, yields a lower bound for the total number of remote and fence steps executed during $E(C_{init};\vec{S}_{m_\pi}^\pi)$.

For a stack S, let $|S|$ denote the number of elements in S. For now, we consider only an arbitrary fixed permutation $\pi = (p_0,\ldots,p_{n-1})$ over $[n]$. For $i = 0,1,\ldots$ let $\vec{S}_i$ be the stack sequence constructed as described at the beginning of Section 5.2 for this permutation. Further, as before let $E_i = E(C_{init};\vec{S}_i)$.

Consider a non-empty command stack $\vec{S}_i|_p$, and let cmd be a command on the stack. Note that if cmd is a *wait-local-finish*(k,Q) or *wait-read-finish*(k,Q) command, then $Q = \emptyset$ (see cases (E1) and (E2b) in the construction).

We assign each command cmd on $\vec{S}_i|_p$ a *value* val(cmd). Each of the commands *proceed* and *commit* have value 1. For $k \in \mathbb{N}$, each of the commands *wait-hidden-commit*(k), *wait-local-finish*$(k,\emptyset)$, and *wait-read-finish*$(k,\emptyset)$ has value k. The value of command stack $\vec{S}_i|_p$, val$(\vec{S}_i|_p)$, is the sum of values of its commands.

In the following sections, we relate the values of commands on the stacks of $\vec{S}_i$, as well as the sizes of the stacks, to the number of remote steps executed during E_i. In Section 5.3.1, we show that the sum of values of *wait-read-finish*() commands on $\vec{S}_i$ is bounded asymptotically by the number of remote steps in E_i. In Section 5.3.2, we show the same for the sum of values of *wait-hidden-commit*() and *wait-local-finish*() commands. Thus, by averaging over these three types of commands, we obtain that the sum of values of commands is bounded within a constant factor of the number of remote steps. In Section 5.3.3, we show that the sum of stack sizes, and thus the total number of commands, is bounded asymptotically by the number of `fence` steps. These results yield Theorem 4.2, as we discuss in Section 5.3.4.

5.3.1 *Analysis of Wait-Read-Finish() Commands*

The aim of this section is to bound from below the number of remote steps of the final execution E_{m_π} constructed for permutation π by the sum of values of *wait-read-finish*() commands on the stacks of $\vec{S}_{m_\pi}^\pi$. The following lemma summarizes this bound.

LEMMA 5.3. *Suppose in $E_j = E(C_{init};\vec{S}_j)$ all participating processes enter a final state. Let V be the sum of values of wait-read-finish() commands on stacks of $\vec{S}_j$. Then E_j contains at least $\lceil V/2 \rceil$ remote steps.*

To prove this lemma, we use an amortized analysis, in which we charge remote read steps of some processes to other processes. The rest of this section is devoted to the proof of this lemma. We first start with a simple observation, which follows immediately from (I4) and (I5) and the fact that p_ℓ must execute a write step to add some element to its write-buffer.

OBSERVATION 5.4. *During any execution $E(C_{init};\vec{S}_i)$, after process p_ℓ has executed its first step, no process p_b, $b < \ell$, accesses any register in $\mathcal{R}_{p_\ell}$. In particular no process p_b, $b < \ell$, accesses a register $R \in \mathcal{R}_{p_\ell}$ after p_ℓ has added R to its write-buffer for the first time.*

In the following, let τ_j be defined as in Section 5.2, i.e., it is the largest index in $[n]$ such that $\vec{S}_j|_{p_{\tau_j}} \neq \emptyset$, and $\tau_j = -1$ if no such index exists.

CLAIM 5.5. *If during $E(C_{init};\vec{S}_j)$ an extended configuration $(C;\vec{S})$ is reached, such that as a result of the last step leading to $(C;\vec{S})$ a wait-read-finish$(\zeta,\emptyset)$ command cmd appears on top of p_{τ_j}'s stack, then for all $b < \tau_j$ and all $R \in WB_{p_{\tau_j}}(C)$,*

(a) process p_b commits no write to R during $E(C;\vec{S})$; and

(b) process p_b does not read R during $E(C;\vec{S})$ after the point in which cmd was popped from the top of p_{τ_j}'s stack without being pushed back to it.

PROOF. Let $i < j$ be the index such that $\vec{S}_{i+1}$ is constructed from $\vec{S}_i$ by adding the command $cmd_{i+1} = \textit{wait-read-finish}(\zeta, \emptyset)$ to the bottom of $\vec{S}_i|_{p_\ell}$. Let $\ell = \tau_i$ and note that $\ell = \tau_j$. Since cmd_{i+1} is a $\textit{wait-read-finish}()$ command, Encoding Rule (E2b) was applied to construct $\vec{S}_{i+1}$ (with $\gamma = 0$). In particular, if $(X; \vec{T})$ is the extended configuration reached during $E(C_{init}; \vec{S}_i)$, when p_ℓ's stack is empty for the first time, then

$$\text{throughout } E(X;\vec{T}) \text{ no process } p_b,\ b<\ell \text{ commits a write to a register that appears in } WB_{p_\ell}(X). \quad (4)$$

Since $S_j|_{p_\ell}$ is equal to S_i but with additional commands added on the bottom of the stack (i.e., below cmd_i), and all other stacks are identical in S_j and S_i, executions $E(C_{init}; \vec{S}_j)$ and $E(C_{init}; \vec{S}_i)$ are identical up to the point when p_ℓ's stack becomes empty in $E(C_{init}; \vec{S}_i)$, which is the point in $E(C_{init}; \vec{S}_j)$ when cmd_{i+1} becomes the top element of p_ℓ's stack. Thus, both executions have a common prefix D, such that $E(C_{init}; \vec{S}_j) = D \circ E(C; \vec{S})$ and $E(C_{init}; \vec{S}_i) = D \circ E(X; \vec{T})$. In particular, $X = C$, and thus $WB_{p_\ell}(C) = WB_{p_\ell}(X)$. Moreover, by (I3) $E(C; \vec{S})|_{[n]-\{p_\ell\}} = E(X; \vec{T})|_{[n]-\{p_\ell\}}$, so Part (a) of the claim follows from (4)

For Part (b) note that with the same arguments as above, $E(C_{init}; \vec{S}_{i+1})$ and $E(C_{init}; \vec{S}_j)$ have a common prefix D' such that $E(C_{init}; \vec{S}_{i+1}) = D' \circ E(C^{\emptyset}_{i+1}; \vec{S}^{\emptyset}_{i+1})$, where $(C^{\emptyset}_{i+1}; \vec{S}^{\emptyset}_{i+1})$ is the extended configuration reached when p_ℓ's stack is empty for the first time. Then D is a prefix of D' that ends when system configuration X is reached. By the decoding rules, p_ℓ takes no steps while a $\textit{wait-read-finish}()$ command is on top of its stack, and $\textit{wait-read-finish}()$ commands can only be replaced by other $\textit{wait-read-finish}()$ commands or removed. Hence, in $E(C_{init}; \vec{S}_{i+1})$, p_ℓ takes no steps between the point when configuration X and when configuartion $C^{\emptyset}_{i+1}$ is reached, and so in both system configurations p_ℓ's state is the same. In particular $WB_{p_\ell}(C^{\emptyset}_{i+1}) = WB_{p_\ell}(X) = WB_{p_\ell}(C)$. By (I8), process p_b, $b < \ell$, does not access a register $R \in WB_{p_\ell}(C)$ throughout $E(C_{init}; \vec{S}_{i+1})$ after the prefix D', and by (I7) the same is true in $E(C_{init}; \vec{S}_j)$. □

To prove Lemma 5.3, we charge read steps of some processes to the processes whose stacks contain $\textit{wait-read-finish}()$ commands.

Let Γ be some extended configuration. We say that the pair $(p_b, R) \in [n] \times \mathcal{R}$ is *charged to* process p_ℓ in $E(\Gamma)$, if $R \notin \mathcal{R}_{p_b}$, p_b reads register R during $E(\Gamma)$ at some point t, it does not read it again after point t, and $\ell > b$ is the smallest index such that at point t R appears in p_ℓ's write buffer. Clearly, every pair (p_b, R) can only be charged to one process.

For any execution E, let $\textit{Write}_p(E)$ denote the set of registers on which p executes write steps during E, i.e., registers that p adds to its write-buffer at some point during E.

CLAIM 5.6. *Suppose $\vec{S}_j|_{p_\ell}$, $\ell \in [n]$, contains K $\textit{wait-read-finish}()$ commands $\omega_1, \ldots, \omega_K$. Then in $E_j = E(C_{init}; \vec{S}_j)$ there are at least*

$$\mathrm{val}(\omega_1) + \cdots + \mathrm{val}(\omega_K) - |\textit{Write}_{p_\ell}(E_j) - \mathcal{R}_{p_\ell}|$$

distinct process/register pairs charged to process p_ℓ.

PROOF. If $\vec{S}_j|_{p_\ell} = \emptyset$, then the claim is trivially true. Hence, assume that $\vec{S}_j|_{p_\ell} \neq \emptyset$.

Recall that $\vec{S}_{i+1}$ is obtained from $\vec{S}_i$ by adding an element to the bottom of the stack of process p_{τ_i}, and τ_i is non-decreasing. Let $j' \le j$ be the largest index such that $\vec{S}_{j'}$ is obtained from $\vec{S}_{j'-1}$ by adding a command to (the bottom of) p_ℓ's stack. Then $\ell = \tau_{j'}$. Moreover, $\vec{S}_{j'}|_{p_s} = \vec{S}_j|_{p_s}$ for $s \le \ell$ and by (I1) $\vec{S}_{j'}|_{p_s} = \emptyset$ for $s > \ell$. Hence, $\vec{S}_{j'}|_{p_\ell}$ contains also exactly the $\textit{wait-read-finish}()$ commands $\omega_1, \ldots, \omega_K$, and since by (I7) (applied to $\vec{S}_j$), $E(C; \vec{S}_{j'})|_{\{p_1,\ldots,p_\ell\}} = E(C; \vec{S}_j)|_{\{p_1,\ldots,p_\ell\}}$, it suffices to prove the claim for $\vec{S}_{j'}$. For the ease of notation assume w.l.o.g. $j = j'$, i.e., $\ell = \tau_j$.

Assume w.l.o.g. that $\omega_1, \ldots, \omega_K$ appear in this order (from top to bottom) on $\vec{S}_j|_{p_\ell}$. Let t_k be the point in time during $E(C_{init}; \vec{S}_j)$ when ω_k, $k \in \{1, \ldots, K\}$, is on top of the stack for the first time, and let $(X_k; \vec{T}_k)$ be the extended configuration reached at that point. By (I6), at the end of $E(C_{init}; \vec{S}_j)$, p_ℓ's stack is empty, so at some point $t'_k > t_k$ the command that is initially below ω_k moves to the top or the stack becomes empty. (By Decoding Rule (D2d) in each step in which p_ℓ's stack has a $\textit{wait-read-finish}()$ command on top, either the stack doesn't change, or the top command gets replaced by a different $\textit{wait-read-finish}()$ command or is removed.) Note that by the deocoding rules process p_ℓ cannot execute any step as long as a $\textit{wait-read-finish}()$ command is on top of its stack, so its write buffer remains $WB_{p_\ell}(X_k)$ throughout $[t_k, t'_k]$. Let W_k denote the set of registers appearing in $WB_{p_\ell}(X_k)$ throughout $[t_k, t'_k]$. Then

$$\bigcup_{1 \le k \le K} W_k \subseteq \textit{Write}_{p_\ell}(E_j). \quad (5)$$

Recall that ω_k is a $\textit{wait-read-finish}(\mathrm{val}(\omega_k), \emptyset)$ command. Let Q_k be the set of pairs $(q_a, R) \in [n] \times \mathcal{R}$, $a \neq \ell$, such that during $[t_k, t'_k]$ process q_a reads register R and R appears in $WB_{p_\ell}(X_k)$. Let $\textit{Reg}(Q_k)$ denote the set of registers such that a pair $(\cdot, R)$ appears in Q_k. In the full version of the paper, we show that there are exactly $\mathrm{val}(\omega_k)$ processes, such that during $[t_k, t'_k]$ each of them reads a register in W_k and enters a final state during $[t_k, t'_k]$. Hence, $|Q_k| \ge \mathrm{val}(\omega_k)$.

For every register R there can be at most one pair $(q, R) \in Q_k$ such that $R \in \mathcal{R}_q$. Moreover, from Observation 5.4, we conclude that $R \notin \mathcal{R}_{p_\ell}$. Hence, we have

$$\big|\{(q_a, R) \in Q_k \mid R \notin \mathcal{R}_q\}\big| + |(W_k \cap \textit{Reg}(Q_k)) - \mathcal{R}_{p_\ell}| \ge |Q_k| \ge \mathrm{val}(\omega_k). \quad (6)$$

Now note that if $R \in \textit{Reg}(Q_k)$, then according to Claim 5.5 (b), no process p_b, $b < \ell$, accesses register R after t'_k . Hence, since time interval $[t_{k+1}, t'_{k+1}]$ starts only after $[t_k, t'_k]$ is finished $R \notin \textit{Reg}(Q_{k'})$ for any $k' > k$. I.e., $\textit{Reg}(Q_{k'}) \cap \textit{Reg}(Q_k) = \emptyset$ for all $1 \le k < k' \le K$, and thus

$$\begin{aligned}\sum_{1 \le k \le K} |(W_k \cap \textit{Reg}(Q_k)) - \mathcal{R}_{p_\ell}| &= \Big|\bigcup_{1 \le k \le K} \big((W_k \cap \textit{Reg}(Q_k)) - \mathcal{R}_{p_\ell}\big)\Big| \\ &\le \Big|\big(\bigcup_{1 \le k \le K} W_k\big) - \mathcal{R}_{p_\ell}\Big| \le |\textit{Write}_{p_\ell}(E_j) - \mathcal{R}_{p_\ell}|.\end{aligned}$$

Thus, (6) implies $\sum_{1 \le k \le K} \mathrm{val}(\omega_k) \le |\textit{Write}_{p_\ell}(E_j) - \mathcal{R}_{p_\ell}| + \sum_{1 \le k \le K} \big|\{(q_a, R) \in Q_k \mid R \notin \mathcal{R}_q\}\big|$.

Therefore, to complete the proof of the lemma, it suffices to show that every pair $(q_a, R) \in Q_k$, $R \notin \mathcal{R}_{q_a}$, is charged to p_ℓ. Since $(p_a, R) \in Q_k$, p_a reads a register $R \in W_k = WB_{p_\ell}(X_k)$ at some point in $[t_k, t'_k]$ during $E_j = E(C_{init}; \vec{S}_j)$. Let t be the last point in $[t_k, t'_k]$ at which p_a reads register R. Then at that point R is in p_ℓ's write-buffer. By Claim 5.5 (b) p_a does not read R after t'_k, so t is the last point at which p_a reads R throughout the entire execution E_j . Since the stacks of processes $p_{\ell+1}, \ldots, p_{n-1}$ are empty, none of them takes a step in this interval (as is the case for p_ℓ), and so $a < \ell$. Thus, if (p_a, R) is not charged to p_ℓ, then it must be charged to a different process p_b, where $a < b < \ell$. I.e., at point t during $E(C_{init}; \vec{S}_j)$ R is also in p_b's write-buffer. By Claim 5.2, p_b's write-buffer is empty in configuration C_j, which is reached at the end of execution $E(C_{init}; \vec{S}_j)$, so at some point $t' > t > t_k$ p_b commits a write to R. But since at point t_k register R appears in p_ℓ's write-buffer and ω_k is on top of its stack, this contradicts Claim 5.5 (a). □

We are now ready to prove Lemma 5.3.

PROOF OF LEMMA 5.3. Let Z_q denote the number of pairs (q, R) that are charged to processes, and let $Z = Z_0 + \cdots + Z_{n-1}$. Recall that if (q, R) is charged to some process, then by definition $R \notin \mathcal{R}_q$, and q reads register R at least once during E_j. Since the first (read or commit) access by q of a register in $\mathcal{R} - \mathcal{R}_q$ must be a remote step, q executes at least Z_q remote steps during E_j. Hence, Z is a lower bound for the total number of remote steps executed during E_j.

Let $W_q = |Write_q(E_j) - \mathcal{R}_q|$ and $W = W_0 + \cdots + W_{n-1}$. Recall that just before a process executes a return step, it must execute a final fence step. Hence, since q enters a final state during E_j, all its writes get committed during E_j. Since for every register $R \in \mathcal{R} - \mathcal{R}_q$ the first commit by q to R must be a remote commit, the total number of remote commit steps q executes during E_j is at least W_q, and so W is also a lower bound for the total number of remote steps executed during E_j.

By definition, every pair (q, R) can only be charged to one process, so by Claim 5.6 we have $Z + W \geq V$, and thus either $Z \geq \lceil V/2 \rceil$ or $W \geq \lceil V/2 \rceil$. □

5.3.2 *Analysis of Wait-Hidden-Commit() and Wait-Local-Finish() Commands*

We now show that the number of remote steps in execution E_j is bounded asymptotically from below by sum of values of *wait-hidden-commit*() and of *wait-local-finish*() commands used to encode E_j.

LEMMA 5.7. *Let $\vec{S}_j$ be a command stack sequence and let V_1 denote the sum of values of wait-hidden-commit() commands and V_2 the sum of values of wait-local-finish() commands. Then $E_j = E(C_{init}; \vec{S}_j)$ contains at least $\max\{V_1/2, V_2\}$ remote steps.*

To prove the lemma, we first consider commit steps executed by waiting processes and relate them to the sum of values of *wait-hidden-commit*() commands.

CLAIM 5.8. *Let Γ be an extended configuration, and suppose in some step s of $E(\Gamma)$ a waiting process p commits a write to some register R. Then after step s process p does not commit to R again, and no non-commit step gets executed until in some step a commit enabled process $q \neq p$ has committed a write to R.*

PROOF. Let $(C; \vec{S})$ be the extended configuration reached during $E(\Gamma)$ such that s is the first step of $E(C; \vec{S})$. Since p is waiting in $(C; \vec{S})$, by Decoding Rule (D1), there must be some other process $q \neq p$, that is commit enabled in $(C; \vec{S})$, and it contains a write to R in its write-buffer $WB_q(C)$. From the decoding rules it also follows that a commit enabled process will remain commit enabled until it committed all the writes in its write-buffer. By Decoding Rule (D1), as long as there is at least one commit enabled process in an extended configuration, the first step will always be a commit step. In particular, in $E(C; \vec{S})$ no process can execute any non-commit step before q has committed all its writes in $WB_q(C)$. It follows that all read, fence, write, or return steps of $E(C; \vec{S})$ must occur after q has committed to R. Moreover, since at any time p's write-buffer can contain only one write to each register, after executing step s p cannot commit to R again until it has added a new write $(R, \cdot)$ to its write-buffer in a write step, which, as argued above, cannot happen until q has committed its write to R. □

Let Γ be an extended configuration. Recall that a commit step s during $E(\Gamma)$ is *hidden* if s is executed by a waiting process. Every other commit step is called *visible*. Note that a commit step by p is hidden, if and only if the command cmd that is on top of p's stack when s gets executed is a *wait-hidden-commit*() command, and it is visible if and only if $cmd = commit$.

CLAIM 5.9. *Let p be a process and $R \in \mathcal{R} - \mathcal{R}_p$. In execution $E(C_{init}; \vec{S}_j)$ process p executes at least as many remote steps on R as it executes hidden commits on R.*

PROOF. Let $s_1, s_2, \ldots, s_k$ be the hidden commit steps on R that process p executes (in this order) during $E(C_{init}; \vec{S}_j)$. By Claim 5.8, any two hidden commit steps s_t, s_{t+1}, by p are separated by some visible commit step s^* of a process $q \neq p$. Hence, if s_{t+1} is not a remote commit by p, then p must have executed a remote commit to R after q's commit step s^* and before its own commit step s_{t+1}. Since in addition, the very first commit by p to R is remote, p executes at least k remote commit steps. □

CLAIM 5.10. *Let R be some register, and let h_R be the number of hidden commit steps on R executed during some execution $E(C_{init}; \vec{S}_j)$. Then the number of remote steps executed on R during that execution is at least $h_R/2$.*

PROOF. Let $h_{R,p}$ denote the number of hidden commits executed by process p on register R, and z the process for which $R \in \mathcal{R}_z$. By Claim 5.9, for each process $p \in [n] - \{z\}$ all $h_{R,p}$ hidden commits by p are remote steps.

By Claim 5.8, any hidden commit step by z on R is followed by a visible commit by a process $q \neq z$, which gets executed before z can execute any other hidden commit. Thus, each hidden commit by z is followed by a remote step on R. Hence, the execution contains at least $h_{R,z}$ remote steps on R.

Using the bound above for processes $p \neq z$, we obtain that the number of remote steps on R is at least $\max\left\{h_{R,z}, \sum_{p \in [n]-\{p\}} h_{R,p}\right\} \geq h_R/2$. □

PROOF OF LEMMA 5.7. By (I6), at the end of $E(C_{init}; \vec{S}_j)$ all command stacks are empty. Hence, it follows immediately from Decoding Rule (D1b) that the total number of hidden commit steps executed during that execution is at least V_1. Then by Claim 5.10, E_j contains at least $V_1/2$ hidden commit commands.

By (I4), each stack of $\vec{S}_j$ contains at most one *wait-local-finish*() command. If $\vec{S}_j|_p$ contains a *wait-local-finish*(γ_p) command, then by (I5) there are at least γ_p processes in $[n] - \{p\}$ that access registers in $\mathcal{R}_p$ during E_j. Since for each such process the first access of a register in $\mathcal{R}_p$ is a remote step, it follows that there are at last γ_p remote steps on registers in $\mathcal{R}_p$. Hence, by summing over all processes p, we obtain that E_j contains at least V_2 remote steps. □

5.3.3 *Fence Steps vs. Stack Size*

The following lemma states that the number of fence steps executed in an execution E_j is asymptotically at least as large as the number of elements on all stacks used to encode E_j.

LEMMA 5.11. *In execution $E(C_{init}; \vec{S}_j)$ process p_ℓ executes at least $\lceil (|S| - 1)/4 \rceil - 3$ fence steps, where $S = \vec{S}_j|_{p_\ell}$.*

PROOF. By (I4) S contains at most one $\mathit{wait\text{-}local\text{-}finish}()$ command. By (I10), below a $\mathit{wait\text{-}read\text{-}finish}()$ command there can only be a *commit* command and below a $\mathit{wait\text{-}hidden\text{-}commit}()$ command there can only be a $\mathit{wait\text{-}read\text{-}finish}()$, *proceed*, or *commit* command. Finally, also by (I10), below a *commit* command there can only be a *proceed* command. Hence, if we remove the single $\mathit{wait\text{-}local\text{-}finish}()$ command, then among the remaining commands at least every fourth one must be *proceed*. I.e., S contains at least $\lceil(|S|-1)/4\rceil$ *proceed* commands. By (D2a), a *proceed* command on p_ℓ's stack can only be removed due to a step in which p_ℓ becomes poised to execute a `fence()` or `return()` operation, or in which it enters a final state. Clearly, throughout $E(C_{init}; \vec{S}_j)$ a *proceed* command can be removed at most once for the reason that p_ℓ becomes poised to execute a `return()` operation, and at most once for the reason that p_ℓ enters a final state. Hence, the claim follows from the fact that by (I6) p_ℓ's stack is empty at the end of $E(C_{init}; \vec{S}_j)$. $\square$

5.3.4 Putting Things Together

Consider a permutation π and the command stack sequences $\vec{S}_0^\pi, \vec{S}_1^\pi, \ldots$, constructed in Section 5.2, for this permutation. Since $\vec{S}_0^\pi$ consists only of empty stacks, and in each step of the construction exactly one command is added to some stack, the stacks of $\vec{S}_i^\pi$ contain in total i commands. The iterative construction ends only when a command stack sequence $\vec{S}_{m_\pi}^\pi$ has been constructed such that during $E_{m_\pi}^\pi = E(C_{init}; \vec{S}_{m_\pi}^\pi)$, all processes enter a final state. If then construction does not end at all, then by Lemma 5.11 the execution has an unbounded number of fence steps. Hence, assume that the iterative construction ends with a command stack sequence $\vec{S}_{m_\pi}^\pi$, for every permutation π.

Let $v_1^\pi, v_2^\pi, \ldots, v_{m_\pi}^\pi$ be the values of the m_π commands on the n stacks of $\vec{S}_{m_\pi}^\pi$, and $v_\pi = v_1^\pi + \cdots + v_{m_\pi}^\pi$. Then we can encode $\vec{S}_{m_\pi}$ using at most

$$\log(v_1^\pi) + \cdots + \log(v_{m_\pi}^\pi) + O(m_\pi + n) \le m_\pi \cdot \log(v_\pi/m_\pi) + O(m_\pi + n) \text{ bits.} \quad (7)$$

By (I2), if $\pi = (p_0, \ldots, p_{n-1})$, then during $E_m^\pi = E(C_{init}; \vec{S}_{m_\pi}^\pi)$ each process p_k, $k \in [n]$, enters a final state with value k. Therefore, the stack sequence $\vec{S}_{m_\pi}^\pi$ uniquely identifies permutation π. Since there are $n!$ permutations, we need on average $\Omega(n \log n)$ bits to encode each permutation. Hence, for at least one permutation π, we obtain from (7)

$$m_\pi \cdot \big(\log(v_\pi/m_\pi) + 1\big) = \Omega(n \log n).$$

Since there are in total m_π commands on all stacks of $\vec{S}_{m_\pi}$, we have from Lemma 5.11 that E_π contains $\Omega(m_\pi)$ fence steps. From Lemmas 5.3 and 5.7, and by averaging the sum of values of $\mathit{wait\text{-}read\text{-}finish}()$, $\mathit{wait\text{-}hidden\text{-}commit}()$, and $\mathit{wait\text{-}local\text{-}finish}()$ commands, it follows that $\Omega(v_\pi)$ steps of E_π are remote steps. This completes the proof of Theorem 4.2.

6. SUMMARY

We proved an inherent tradeoff between the number of fences and the number of remote accesses that must be incurred in implementations of certain concurrent operations, e.g., on locks, counters and queues. The proof assumes a model in which each process has both a local segment of shared memory and a cache. Thus, Theorem 4.2 holds for both *cache-coherent* and *distributed shared-memory* architectures [6]. Following [9, 12], our lower bound applies also to algorithms that may use *comparison* primitives, such as CAS, in addition to reads and writes.

7. REFERENCES

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

[2] D. Alistarh, J. Aspnes, S. Gilbert, and R. Guerraoui. The complexity of renaming. In *FOCS*, pages 718–727, 2011.

[3] J. Anderson and Y.-J. Kim. An improved lower bound for the time complexity of mutual exclusion. *Dist. Comp.*, 15(4):221–253, 2002.

[4] J. H. Anderson and Y.-J. Kim. Adaptive mutual exclusion with local spinning. In *DISC*, pages 29–43, 2000.

[5] J. H. Anderson and Y.-J. Kim. Nonatomic mutual exclusion with local spinning. In *PODC*, pages 3–12, 2002.

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

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

[8] H. Attiya, D. Hendler, and S. Levy. An $O(1)$-barriers optimal RMRs mutual exclusion algorithm. In *PODC*, pages 220–229, 2013.

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

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

[11] R. Fan and N. Lynch. An $\Omega(n \log n)$ lower bound on the cost of mutual exclusion. In *PODC*, pages 275–284, 2006.

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

[13] Intel Corporation. *Intel® 64 and IA-32 Architectures Software Developer's Manual*. Number 253669-033US. December 2009.

[14] P. Jayanti. Adaptive and efficient abortable mutual exclusion. In *PODC*, pages 295–304, 2003.

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

[16] Y.-J. Kim and J. Anderson. A time complexity bound for adaptive mutual exclusion. In *DISC*, pages 1–15, 2001.

[17] L. Lamport. A new solution of Dijkstra's concurrent programming problem. *Comm. ACM*, 17(8):453–455, 1974.

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

[19] P. E. McKenney. Memory ordering in modern microprocessors, part I. *Linux Journal*, 2005(136):2, 2005.

[20] P. E. McKenney. Memory barriers: a hardware view for software hackers. *Linux Tech. Center, IBM Beaverton*, 2010.

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

[22] G. L. Peterson and M. J. Fischer. Economical solutions for the critical section problem in a distributed system. In *STOC*, pages 91–97, 1977.

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