Tight RMR Lower Bounds for Mutual Exclusion and Other Problems

(Extended Abstract)

Hagit Attiya[*] Danny Hendler[†] Philipp Woelfel [‡]

ABSTRACT

We investigate the remote memory references (RMRs) complexity of deterministic processes that communicate by reading and writing shared memory in asynchronous cache-coherent and distributed shared-memory multiprocessors.

We define a class of algorithms that we call *order encoding*. By applying information-theoretic arguments, we prove that every order encoding algorithm, shared by n processes, has an execution that incurs $\Omega(n \log n)$ RMRs. From this we derive the same lower bound for the mutual exclusion, bounded counter and store/collect synchronization problems. The bounds we obtain for these problems are tight. It follows from the results of [10] that our lower bounds hold also for algorithms that can use comparison primitives and load-linked/store-conditional in addition to reads and writes. Our mutual exclusion lower bound proves a longstanding conjecture of Anderson and Kim.

Categories and Subject Descriptors

D.1.3 [**Software**]: Programming Techniques—*Concurrent programming*; F.1.2 [**Theory of Computation**]: Computation by Abstract Devices—*Modes of Computation* [Parallelism and concurrency]; F.2.2 [**Theory of Computation**]: Analysis of Algorithms and Problem Complexity—*Nonnumerical Algorithms and Problems*

General Terms

Algorithms, Theory

Keywords

information theory, shared-memory, lower bound techniques, mutual exclusion, bounded counter, store/collect object

[*]Department of Computer Science, Technion; Supported in part by the *Israel Science Foundation* (grant number 953/06). hagit@cs.technion.ac.il

[†]Department of Computer Science, Ben-Gurion University, hendlerd@cs.bgu.ac.il

[‡]Department of Computer Science, University of Calgary, woelfel@cpsc.ucalgary.ca

1. INTRODUCTION

The memory hierarchy in shared-memory multi-processor systems places some of the memory locally, while the rest of the memory is remote, e.g., in other processing units or in dedicated storage. For example, in *cache-coherent* (CC) systems, each processor maintains local copies of shared variables in its cache; the consistency of copies in different caches is ensured by a coherence protocol. At any given time, a variable is *local* to a process if the coherence protocol guarantees that the corresponding cache contains an up-to-date copy of the variable, and is *remote* otherwise. In *distributed shared-memory* (DSM) systems, on the other hand, each shared variable is permanently locally accessible to a single processor and remote to all other processors.

References to remote memory are orders of magnitude slower than accesses to the local memory module; they generate traffic on the processor-to-memory interconnect, which can be a bottleneck and cause further slow-down. For these reasons, the performance of many algorithms for shared memory multiprocessor systems depends critically on the number of references to remote memory they generate [7, 14].

The *remote memory references* complexity measure, abbreviated *RMR cost*, charges an algorithm with one unit for each access of remote memory, namely, reading from or writing to memory locations that do not reside locally at the process; writing, reading or even spinning on a memory location that is locally available is considered free. The RMR cost is considered to accurately predict the actual performance of a concurrent algorithms deployed in a multiprocessing system.

The familiar way to measure time complexity, by counting the total number of shared memory accesses, regardless of whether they are local or remote, is inadequate for blocking problems such as mutual exclusion: For such problems, a process may perform an unbounded number of memory accesses while performing a local spin loop, waiting for another process [1]. Instead, recent mutual exclusion work mostly uses the RMR cost to evaluate algorithms (see, e.g., [2, 5, 6, 13]).

We prove tight lower bounds on the RMR cost of key synchronization problems—mutual exclusion, bounded counters, and store/collect.

Our Technique and Contributions. We define *order encoding* algorithms. Roughly speaking, an algorithm is order encoding if, for each of its executions E, the order in which the operations that were performed in E "took effect" can be determined based on the state of the shared memory after E terminates.

Our key technical result (Theorems 1 and 9) establishes that every n-process order encoding algorithm has an execution that incurs $\Omega(n \log n)$ RMRs. To prove this, we construct a set $\mathcal{E}$ of $n!$ stylized executions of the order encoding algorithm. The set $\mathcal{E}$ has the properties that (a) different executions of $\mathcal{E}$ result in different shared-memory states, and (b) each execution $E \in \mathcal{E}$ can be represented by a bit-string whose length is bounded from above (up to a constant factor) by the total number of RMRs performed in E. Since $|\mathcal{E}| = n!$, a simple information theoretic argument shows that the RMR cost of at least one of the executions of $\mathcal{E}$ is $\Omega(n \log n)$.

The reader may wonder whether an algorithm implementing one of the problems we consider, e.g. a mutual exclusion algorithm, must be order encoding: an execution of a mutual exclusion algorithm does *not* necessarily record the order of its operations in shared memory! Indeed, we do not argue directly about the algorithms for the problems we consider. Rather, we present "wrappers" to obtain corresponding order encoding algorithms.

The mutual exclusion problem is at the core of distributed computing theory [16]. Using the technique described above, we prove a lower bound of $\Omega(n \log n)$ RMRs on any implementation of mutual exclusion from reads and writes (Theorem 11). This bound, as well as all other bounds we present, holds for both the CC and the DSM models. This result confirms a longstanding conjecture of Anderson and Kim [4]. We then prove a similar result for bounded counters (Theorem 13) and for store/collect objects (Theorem 15).

In [10] it is proven that any CC or DSM algorithm using read and write operations, load-linked/store-conditional (LL/SC), and comparison primitives, can be simulated by an algorithm that uses only read and write operations, with just a constant blowup in the RMR complexity. It follows that the lower bounds hold also if the algorithms can use comparison primitives and LL/SC in addition to reads and writes.

Related Work. Considerable research has focused on proving RMR bounds for the mutual exclusion problem. Anderson and Yang presented an $O(\log n)$ RMRs mutual exclusion algorithm [17]. Our results establish that this is the best possible. Anderson and Kim proved a lower bound of $\Omega(\log n / \log\log n)$ on the RMR complexity of mutual exclusion algorithms that use reads and writes only [2], improving on a previous lower bound of $\Omega(\log\log n / \log\log\log n)$ obtained by Cypher [8].

Fan and Lynch [9] proved an $\Omega(n \log n)$ lower bound on the *state change* cost of mutual exclusion. This does not imply a corresponding RMR lower bound for either the CC or the DSM models. Nevertheless, their technique introduced several novel ideas, which our technique borrows and extends.

For lack of space, we refer the interested reader to [3] for a comprehensive discussion of RMR upper and lower bounds.

2. MODEL

Our model of computation is based on [12]. A concurrent system models an asynchronous shared memory system where n deterministic *processes* communicate by executing *operations* on shared *variables*.

Each process is a sequential execution path that performs a sequence of *steps*, each of which invokes a single operation on a single variable and receives a corresponding *response*. The operations allowed in our model are *read*, *write*, *comparison primitives* [2, 10] (a.k.a. conditional operations) and *load-linked/store-conditional*. If step σ invokes an operation on variable v we say that σ *accesses* v. A variable that supports the read and write operations only is called a *register*.

In this paper we consider the cache-coherent (CC) and the distributed shared-memory (DSM) computation models. Each processor in a CC machine maintains local copies of shared variables inside its cache, whose consistency is ensured by a coherence protocol. At any given time a variable is remote to a processor if the corresponding cache does not contain an up-to-date copy of the variable.

Our lower bounds apply to both families of CC coherence algorithms: *write-through* and *write-back* [15]. Quoting from [10]: "In a write-through protocol, to read a register R a process p must have a (valid) cached copy of R. If it does, p reads that copy without causing an RMR; otherwise, p causes an RMR that creates a cached copy of R. To write R, p causes an RMR that invalidates (i.e., effectively deletes) all other cached copies of R and writes R to main memory.

In a write-back protocol, each cached copy is held in either "shared" or "exclusive" mode. To read a register R, a process p must hold a cached copy of R 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 R held in exclusive mode, typically by downgrading the status to shared and, if the exclusive copy was modified, writing R back to memory; and (b) creates a cached copy of R held in shared mode. To write R, p must have a cached copy of R 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 R and writes any modified copy held in exclusive mode back to memory; and (b) creates a cached copy of R held in exclusive mode."

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 register is (forever) *local* to a single processor and *remote* to all others. An access of a remote register is an RMR. 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.

We say that a step σ by process p that accesses a variable v is *local* if σ can be applied on a local copy of v, without using the processor-to-memory interconnect. Otherwise we say that σ is a *remote memory reference* (RMR).

An *execution* is a sequence of steps performed by processes as they execute their algorithms. A process p is *idle* after an execution E if, right after E, p is not in the midst of executing its algorithm; that is, either p did not start performing its algorithm in E or p terminates its algorithm in E.

3. LOWER BOUND IN THE CC MODEL

The algorithms we consider in the following only use read and write operations. However, by the results of [10], all our lower bounds hold also for algorithms that can use also comparison operations and LL/SC.

3.1 Order Encoding Algorithms and the Main Theorem

We consider a set $\mathcal{P} = \{p_1, \ldots, p_n\}$ of n processes, where each process p_i executes algorithm A_i. Our proof technique assumes that the algorithms A_i may invoke the following two types of *special operations*, in addition to reads and writes: A *record*-operation, performed by p_i, receives an arbitrary value as input and *records* it by writing it to a register that is only accessible to p_i. An *order*-operation, by process p_i, is a read from a register R_i^*, that is only accessible by p_i. We use order-operations to determine the order in which these operations are executed during the execution, in a way that is made precise later. The last operation performed by each algorithm A_i is a record-operation. This is the only record-operation performed by A_i. A_i is also required to perform the order-operation exactly once.

We emphasize that the above special operations do not limit the set of algorithms to which our lower bounds apply. They are only inserted to *wrapper* versions of these algorithms and our technique is applied to these wrapper versions.

Let $\mathcal{E}$ be the set of all executions in which each process p_i either does not participate or executes its algorithm A_i exactly once. For an execution $E \in \mathcal{E}$, we write $p_i \prec_E p_j$ if process p_i executes its order-operation in E before process p_j does. If all n processes complete their algorithms in E then $\prec_E$ is a total order. The *record-vector* of an execution E is the vector $\vec{R}[E] = (\gamma_1, \ldots, \gamma_n)$, where γ_i is the value recorded by process p_i in its last step in E; γ_i is denoted $R_i[E]$.

Let $L = (q_1, \ldots, q_k)$ be a list of distinct processes in $\mathcal{P}$. The *L-solo execution*, denoted $E_{sol}(L) \in \mathcal{E}$, is the execution in which the set of participating processes is $\{q_1, \ldots, q_k\}$ and, for $1 \le i < k$, process q_i terminates its algorithm before process q_{i+1} executes its first step. If L is a permutation $\pi \in S_n$, we simply write L_π.

DEFINITION 1. *A sequence of algorithms $(A_1, \ldots, A_n)$ is called* "order encoding" *if for all permutations $\pi \in S_n$, for all lists $L = (p_{\pi(1)}, \ldots, p_{\pi(k)})$ of length k, $1 \le k \le n$, and for all executions E in which exactly the processes $p_{\pi(1)}, \ldots, p_{\pi(k)}$ participate and terminate, the following two properties hold:*

(a) If there are $1 \le i < j \le k$, such that $p_{\pi(j)} \prec_E p_{\pi(i)}$, then $\vec{R}[E] \neq \vec{R}[E_{sol}(L)]$.

(b) If $R_{\pi(k)}[E] \neq R_{\pi(k)}[E_{sol}(L)]$, then $R_{\pi(j)}[E] \neq R_{\pi(j)}[E_{sol}(L)]$, for some j, $1 \le j < k$.

Property (a) implies that at least one process can distinguish between an execution E and an L-solo execution E' if the orders $\prec_E$ and $\prec'_E$ differ. Property (b) implies that if the last process in the permutation L distinguishes between E and E', then some earlier process already distinguishes between them.

The intuition is the following: In an L-solo execution E, where $L = (p_1, \ldots, p_k)$, we have $p_i \prec_E p_{i+1}$, for every i, $1 \le i < k$. Moreover, if $j > i$, then process p_i does not "know" whether or not process p_j participates in E. Now consider another execution E', in which we "speed up" process p_k slightly, so that it executes its order-operation just before p_{k-1} does but is still "unnoticed" by processes $p_1, \ldots, p_{k-2}$. Properties (a) and (b) imply that the records of both p_{k-1} and p_k are different in E and E'. Our proof uses this property to capture the required communication between p_{k-1} and p_k.

Our results hold for algorithms that satisfy the following weak progress requirement.

DEFINITION 2. *An implementation $(A_1, \ldots, A_n)$ satisfies* weak obstruction-freedom *if for every process p_i and every execution E after which all processes other than p_i are idle, if p_i runs by itself executing algorithm A_i after E then it eventually terminates.*

Weak obstruction-freedom may be satisfied by an implementation regardless of whether or not it uses locks. It is easily seen that both deadlock-freedom and obstruction-freedom [11] imply weak obstruction-freedom. In what follows, we consider algorithms that satisfy weak obstruction-freedom. The rest of this section is devoted to proving the following theorem.

THEOREM 1. *Let $(A_1, \ldots, A_n)$ be order encoding algorithms that satisfy weak obstruction-freedom. Then there is an execution $E \in \mathcal{E}$ with $\Omega(n \log n)$ RMR cost.*

In the following we define the concept of *critical operations*. The number of critical operations performed during an execution is a lower bound on the number of RMRs incurred during that execution. Our proof technique encodes executions such that the code length is proportional to the number of critical operations.

DEFINITION 3. *Consider an operation σ of process p on register R, that occurs right after some execution prefix E. If σ is a read-operation, then it is* critical *if p has not accessed R in E, or p has accessed R in E, and some other process has written R since p's last access of R. If σ is a write-operation, then it is* critical *if p has not written R in E, or if p has written R in E and some other process has written R since p's last write to R.*

A similar notion of critical events was used by Anderson and Kim [2]; all operations that are critical according to our definition are also critical according to their definition (but not vice versa). A critical operation incurs an RMR in both write-through and write-back CC systems [2, pp. 226–227].

The general structure of our proof is the following. We construct $n!$ different executions, each corresponding to a permutation $\pi \in S_n$ and denoted E_π. Our construction maintains the property that each process $p_{\pi(i)}$ does not "know" whether any processes $p_{\pi(j)}$, for $j > i$, participate in E_π. In that case, it is easily seen (as we prove later) that process $p_{\pi(1)}$ has to act exactly as it does in $E_{sol}(L_\pi)$ and must also produce the same record. By an inductive argument, we show that *all* processes must produce the same record in E_π and in $E_{sol}(L_\pi)$. Indeed, assume that processes $p_{\pi(1)}, \ldots, p_{\pi(i)}$ produce the same record in both executions. Process $p_{\pi(i+1)}$ does not "know" whether any process other than $p_{\pi(1)}, \ldots, p_{\pi(i)}, p_{\pi(i+1)}$ participates in E_π. Hence if it records different values in E_π and in $E_{sol}(L_\pi)$ it might violate property (b) of Definition 1. It follows that the record-vectors of E_π and $E_{sol}(L_\pi)$ are equal.

Since for each permutation π the record vectors of E_π and $E_{sol}(L_\pi)$ are equal, it follows that all executions E_π are different. We then show that we can uniquely *encode* each execution E_π and that the bit-length of each such encoding is bounded from above (up to a constant factor) by the total number of critical operations that are executed during E_π. Hence, the $n!$ different executions E_π lead to $n!$ different codes. Therefore, the total number of critical operations in at least one of these executions is asymptotically at least the logarithm of the number of different records produced by these executions, i.e., $\Omega(\log(n!)) = \Omega(n \log n)$. Below, we describe in detail the structure of the executions E_π.

3.2 Codes and their Interpretation

In this section, we define the structure of the executions E_π, and we define commands, such that a sequence of commands can be interpreted as an execution.

Consider an execution E. We say that a process p *loops locally* after E, if p does not terminate nor does it execute any critical operations in a solo-run that starts after E.

3.2.1 Rounds and Sub-Rounds.

In any execution E_π, processes proceed in *rounds*. Each round is composed of a sequence of 4 *sub-rounds*: *read*, *early-write*, *late-write*, and *non-critical* sub-rounds. The very first sub-round is a read sub-round, followed by an early-write, then a late-write, and finally, a non-critical sub-round. After a round completes, the next round (if any) starts with another read sub-round, and so on.

The idea is that critical operations occur in one of the first three sub-rounds. If the critical operation is a read, then it occurs in the read sub-round, and if it is a write, it occurs in one of the write sub-rounds. The two different types of write sub-rounds allows to control which of the writes may become "visible" in the sense that they are not overwritten before some other process may read what was written. The executions we construct guarantee that whenever a process writes to a register in the early-write round, then some other process writes to the same register in the late-write round. Hence, writes in the early-write rounds are always "invisible".

Each process may or may not *wait* during a round. If it waits, then it does not execute any steps in any of the four sub-rounds of the round. If it does not wait, it may execute at most one single step during the read, early-write, and late-write sub-rounds altogether, and then it may execute additional steps during the non-critical sub-round.

Now consider a round, and let $P \subseteq \mathcal{P}$ be the set of processes participating in that round. For process $p \in P$, let σ_p be the operation p is about to perform. Formally, σ_p is a pair (op, reg), where op is either "read" or "write", and reg identifies the register on which this operation operates. Let $\lambda_p \in \{$"critical","non-critical"$\}$ denote whether σ_p would be a critical operation, *if it were executed as the first operation of the next round*. Note that although λ_p may be "non-critical", σ_p *may* be critical when p actually executes it, because by that time some other process may have already accessed the register which σ_p accesses (our construction guarantees that this does not occur). On the other hand, from Definition 3, if $\lambda_p =$"critical" then σ_p is guaranteed to be a critical operation when p performs it.

Let $P' = \{p \in P | \lambda_p = \text{"critical"}\}$. During the first three sub-rounds of a round (namely, the read, early-write, and late-write sub-rounds) each of the processes in P' performs its operation σ_p. No other operations are performed during these three sub-rounds. Those processes $p \in P'$ for which σ_p is a read-operation execute that operation in the read sub-round. We let these processes perform their read steps in increasing order of process IDs. After the read sub-round, those processes $p \in P'$ for which σ_p is a write-operation perform their writes. Some of these writes will be designated as *early writes*, and these will be performed in the early-write sub-round in increasing process ID order. The remaining writes are *late* writes and will be performed in the late-write sub-round, in increasing process ID order. As we prove later, the early writes performed in each round are overwritten by the late writes of that round.

The final sub-round is the non-critical sub-round, in which we schedule each of the remaining processes of P' in increasing order of their ID. We let each such process p perform steps in the non-critical sub-round only if it is not about to loop locally.[1] In this case, we let p run solo until it is either about to perform a critical operation or until it terminates.

Order-operations play a special role by forcing processes to deviate from the round-scheme described above. A process p that executes its order-operation continues to run solo until it terminates. We prove that, in the executions we consider below, p cannot loop locally in such a solo run.

3.2.2 *Commands*

We associate a sequence s_i of *commands* with each process p_i such that a list $s_1, \ldots, s_n$ of command sequences determines an execution E_π. Each command is associated with a specific round of E_π, in which it *becomes effective*. It may then remain effective during some of the following rounds, until it *ceases to be effective* after some round.

Two types of commands exist: Wait commands and Participate commands. A Participate command indicates that the process will participate in the round in which the command becomes effective. The Participate command receives a single argument. If the process' first operation in that round is a critical write, then the argument indicates whether the write step is scheduled in the early-write or in the late-write sub-round. If the first operation of the process is not a critical write, then the argument is ignored. Thus, the Participate-commands are (1) Participate(late), and (2) Participate(early).

A Wait command indicates that the process should wait in the round in which the command becomes effective. The process may have to wait also in some of the following rounds. Thus, once the Wait command becomes effective, it may remain effective in a few additional rounds.

We now describe the Wait commands in detail and explain how we determine how long they remain effective. Consider a Wait command of process p that becomes effective in round i. Let R be the register that is about to be accessed by p's next operation.

1. Wait(next late-write): This command remains effective until (and including) the first round $j \geq i$, such that in round $j+1$ some process (other than p) will write to R in the late-write sub-round.

2. Wait(reader/writer termination): This command remains effective until (and including) the first round $j > i$, in which one of the processes that access (read or write) R during a round $i' \in \{i+1, \ldots, j\}$ terminates.

Roughly speaking, Wait commands are used as follows. The first type of Wait commands is used for "hiding" a critical write by process p. If process p is about to perform a critical write to register R and this register is later written by an "earlier" process q (i.e., $q \prec p$) whose write is not "hidden", then a Wait(next-write) command will make sure that p's write will be scheduled as an early write in the same round in which q writes to R. Thus p's early write will be overwritten by q's late write.

Hiding critical writes is not always possible and the second type of Wait commands is used in such situations. If process p is about to write to a register that is read later in the execution by an "earlier" process q, and if p's write cannot be "hidden", then the Wait(reader/writer termination) command is used to make sure that p waits until q terminates before it is allowed to perform its critical write. This way we can eliminate information flow from p to "earlier" processes. Another situation, where we use this second type of Wait command is the following: If R is local to process q at the time when p writes to R, then p's write operation changes the locality of R. Hence, q's next write operation on R would now become a critical operation. The way we encode the commands, we have to be sure that the "criticality" of q's operation is not affected by "later" processes. We ensure this, by letting p wait until q's execution is finished.

3.2.3 *The Interpretation of Commands*

We now describe how command sequences uniquely specify executions. An execution $E = C(s_1, \ldots, s_n)$, described by the command sequences $s_1, \ldots, s_n$, is determined as follows.

Assume for some $z \geq 1$, that the first $z-1$ rounds of execution E have been determined and let E' denote the execution consisting of the first $z-1$ rounds (round 0 is defined to be the empty execution). Let σ_p denote p's next step after E', i.e., σ_p is a pair (op, reg), where op indicates whether the step is a read- or write-operation, and reg identifies the register on which this operation operates. Further, let $\lambda_p \in \{$critical, non-critical$\}$ indicate whether or not σ_p is critical if it is scheduled immediately after E'.

For each process p, we now determine which command (if any) will be effective in round z. The exact behavior of process p is then uniquely determined by the structure of rounds described above and by the following rules:

1. If a Wait command is effective in round z, then process p does not take any steps in this round.

[1] If the algorithms are finite-state, then it can be determined by simulation whether a process will loop locally or not. We do not assume that, however. Thus, if the algorithms are infinite-state, then our proof is existential rather than constructive. In this case our proof assumes the existence of an oracle that determines whether or not a process will loop locally starting from a certain point.

2. If a Participate(x) command ($x \in \{\text{early, late}\}$) is effective in round z, then process p executes σ_p in the read sub-round if it is a read-operation, and otherwise it executes σ_p in the x-write sub-round. It also executes non-critical steps in the non-critical sub-round (as described in Section 3.2.1), unless it is about to loop locally.

3. If no command is effective in round z, then process p executes only non-critical steps in the non-critical sub-round (as described in Section 3.2.1), unless it is about to loop locally.

Hence, to establish the uniqueness of the execution specification, it suffices to show how to determine which command (if any) is effective in each round. Let c'_p denote the last command of p that became effective in E' (if such a command exists), and let c_p denote the command following c'_p in p's sequence of commands (if it exists). If c'_p and c_p do not exist, then no command is effective in round z.

In round z, command c'_p may either remain effective or it may cease to be effective before round z starts. Even if c'_p ceases to be effective (or does not exist), c_p might not become effective in round z. In fact, c_p becomes effective in round z if and only if c'_p ceased to be effective before the start of round z and if $\lambda_p =$ critical.

It remains to describe how to decide whether c'_p ceases to be effective right after round $z-1$, or whether it remains effective during round z. If c'_p is a Wait(reader/writer termination) command that was effective in round $z-1$ then it is uniquely determined by the previous rounds whether it remains effective in round z (see the description of the Wait commands in Section 3.2.2). It remains to consider the case where $c'_p =$Wait(next late-write).

Let P' denote the set of processes p for which $c'_p =$Wait(next late-write) and c'_p was effective in round $z-1$. For all processes not in P' it is already known, which command is effective in round z, so we know whether a process $q \in P - P'$ writes in the late-write sub-round of round z. Thus, we can apply the following rule:

> For $p \in P'$ and $\sigma_p = (op, R)$, command c'_p ceases to be effective right after round $z-1$, if and only if there is a process $q \in P - P'$ that writes in the late-write sub-round of round z to register R.

This rule would not be compatible with our commands definition, if there were a process $p' \in P'$ that would now write in the late-write sub-round of round z to register R. But this cannot happen, as our encodings will guarantee that no process $p' \in P'$ will "write late" in round z. This is ensured by the following invariant:

Every Wait(next late-write) in the commands sequence of a process is immediately followed by a Participate(early) command. (1)

Hence, we know whether or not there is a process that "writes late" to R in round z *before* considering the processes of P'.

The specifications of rounds and commands and their interpretation guarantees that every list $(s_1, \ldots, s_n)$ of command sequences uniquely determines an execution $C(s_1, \ldots, s_n)$. Note that this does not imply that all processes terminate in this execution.

3.3 Execution Encoding

In this section we show how to find for every permutation π command sequences $s_1, \ldots, s_n$ defining an execution $E_\pi = C(s_1, \ldots, s_n)$ which has the same record vector as $E_{sol}(L_\pi)$.

DEFINITION 4. *Let E and E' be two executions, and let σ_i and σ'_i, for $i \in \mathbb{N}$, be the i-th step a process p makes during executions E and E', respectively; If p makes fewer than i steps in E (E'), then σ_i (σ'_i) is defined to be $\perp$. The executions E and E' are* indistinguishable *for process p, denoted $E \sim_p E'$, if the following statements hold for every $i \in \mathbb{N}$:*

1. *$\sigma_i = \perp$ if and only if $\sigma'_i = \perp$.*

2. *If σ_i and σ'_i are read-operations, then both operations return the same response.*

3. *σ_i is a critical operation if and only if σ'_i is a critical operation.*

The first property ensures that both processes perform an equal number of steps in E and E'. The second and third properties ensure that p is in the same state after performing i steps in E and in E'. If p is in the same state after i steps in E and E', then, clearly, the type of operation in the $(i+1)$'th step (i.e., read or a write) and the register on which it operates are the same in E and E'. Hence, $\sigma_{i+1} = \sigma'_{i+1}$.

For the rest of this section we fix a permutation π. Every process p_i executes algorithm A_i, and the sequence $(A_1, \ldots, A_n)$ of algorithms is order encoding and satisfies weak obstruction-freedom. We encode processes' steps in the order prescribed by π. That is, first we construct the command sequence $s_{\pi(1)}$ for $p_{\pi(1)}$, then the command sequence $s_{\pi(2)}$ for $p_{\pi(2)}$, and so on. The sequences $s_{\pi(1)}, \ldots, s_{\pi(k)}$ determine an execution $C(s_1 \ldots s_n)$, where $s_\pi(j)$ is the empty command sequence for $j > k$. Thus, the processes $p_{\pi(k+1)}, \ldots, p_{\pi(n)}$ do not take any steps in $C(s_1 \ldots s_n)$ (recall that the first operation of a process is always critical, and if a process' command sequence is empty, then it can only participate in the non-critical sub-rounds where it can't execute a critical operation).

In order to simplify notation, we let $q_i = p_{\pi(i)}$ and $a_i = s_{\pi(i)}$ for $1 \le i \le n$. We also let $D(a_1, \ldots, a_k)$ denote the execution $C(s_1, \ldots, s_n)$, where for all $j > k$, $s_{\pi(j)}$ is the empty command sequence.

3.3.1 Invariants

Our encoding needs to ensure that Invariant (1), defined in Section 3.2.3, holds. In addition, our encoding will maintain (as we prove) the following invariants. Assume we have constructed $s_1, \ldots, s_k$, then

$$D(a_1, \ldots, a_i) \sim_{q_i} D(a_1, \ldots, a_j) \quad \text{for every } i, j,\ 1 \le i \le j \le k, \text{ and} \tag{2}$$

$$\text{in } D(a_1, \ldots, a_k), \text{ all processes } q_1, \ldots, q_k \text{ terminate.} \tag{3}$$

When $k = n$ we have

$$D(a_1, \ldots, a_i) \sim_{q_i} D(a_1, \ldots, a_n) \quad (= C(s_1, \ldots, s_n)),$$

for every i, $1 \le i \le n$.

PROPOSITION 2. *Let $E^k_\pi = D(a_1, \ldots, a_k)$ and $L^k_\pi = (q_1, \ldots, q_k)$. If Invariants (2) and (3) hold then*

(a) *$\vec{R}[E^k_\pi] = \vec{R}[E_{sol}(L^k_\pi)]$.*

(b) *Each process q_i, $1 \le i < k$, terminates before process q_{i+1} executes its order operation.*

PROOF. *(a)* Let $(\alpha_1, \ldots, \alpha_k) = \vec{R}[E^k_\pi]$ and $(\beta_1, \ldots, \beta_k) = \vec{R}[E_{sol}(L^k_\pi)]$. From Invariant (2), executions E^k_π and $D(a_1)$ are indistinguishable to process q_1. They are thus also indistinguishable to q_1 from a solo-execution. Since q_1 terminates in E^k_π by Invariant (3), it has to record β_1. Thus, $\alpha_1 = \beta_1$.

Now assume that $(\alpha_1,\ldots,\alpha_i) = (\beta_1,\ldots,\beta_i)$ for $1 \le i \le k$. We prove that $\alpha_{i+1} = \beta_{i+1}$. First note that, by Invariant (2), $\vec{R}[D(a_1,\ldots,a_{i+1})] = (\alpha_1,\ldots,\alpha_{i+1})$. Moreover, from definitions, $\vec{R}[E_{sol}(L^{i+1}_\pi)] = (\beta_1,\ldots,\beta_{i+1})$, where $L^{i+1}_\pi = (q_1,\ldots,q_{i+1})$. Thus, by Definition 1 (b), if $\alpha_{i+1} \ne \beta_{i+1}$ then $(\alpha_1,\ldots,\alpha_i) \ne (\beta_1,\ldots,\beta_i)$, which is a contradiction.

(b) We prove the claim by induction on i. For $i = 0$ the claim holds vacuously. Assume $i \ge 1$. Consider the point in execution E^k_π when q_i executes its order operation. By induction hypothesis, processes $q_1,\ldots,q_{i-1}$ have already terminated. By definition of the round scheme, q_i runs solo starting from this point until it terminates, if it ever does. Since algorithms A_i satisfy weak obstruction-freedom, q_i eventually terminates when it runs solo after its order operation in $D(a_1,\ldots,a_i)$. By Invariant (2), q_i cannot distinguish E^k_π from $D(a_1,\ldots,a_i)$. It follows that q_i eventually terminates when it runs solo after its order operation also in E^k_π. Finally note that q_{i+1} could not have executed its order-operation before q_i: assuming otherwise implies $q_{i+1} \prec_E q_i$, and, by Definition 1 (a), the records of $\vec{R}[E^k_\pi]$ and $\vec{R}[E_{sol}(L^k_\pi)]$ must differ. This would contradict part (a). □

Part (a) of Proposition 2 establishes that every process p_i records the same values in executions $C(s_1,\ldots,s_n)$ and $E_{sol}(L_\pi)$. Let $E_\pi = C(s_1,\ldots,s_n)$, then the record-vector of E_π can be encoded by a string of length $O(\sum_{i=1}^n |s_i|)$. Assuming the above invariants hold (as we prove in the following section), it follows from Proposition 2 and Definition 1 (a) that every execution E_π yields a *distinct* record-vector.

3.3.2 Realization of the Encoding

We now describe how we construct the executions E_π and encode them so as to maintain Invariants (1)-(3) for $k = 1,\ldots,n$. For $k = 1$, we consider a solo run of process q_1, and we let t be the total number of critical operations it executes. We then let a_1 be the concatenation of t Participate(late) commands. Thus, $D(a_1)$ is a complete solo-run of process q_1, and during each round q_1 is performing exactly one critical operation (either a read in the read sub-round, or a late-write in the late-write sub-round). Moreover, by weak obstruction-freedom and since s_1 contains no Wait commands, Invariants (1), (2) and (3) hold for $k = 1$.

Now assume that $a_1,\ldots,a_{k-1}$, for $k \ge 2$, have been determined, so that Invariants (1), (2), and (3) hold. We construct the commands sequence $a_k = (c_1,\ldots,c_\ell)$ of process q_k. In order to satisfy (1), whenever we add a command Wait(next late-write), we add a command Participate(early) immediately after it. In this case, we abuse notation and let c_i denote a sequence consisting of both these commands.

Assume that we have constructed $a^t_k := (c_1,\ldots,c_t)$. We have to ensure the following two invariants.

$$D(a_1,\ldots,a_i) \sim_{q_i} D(a_1,\ldots,\ldots,a_{k-1},a^t_k) \text{ for every } i,\ 1 \le i < k, \text{ and} \tag{4}$$

$$\text{in } D(a_1,\ldots,\ldots,a_{k-1},a^t_k)\text{, all commands in } a^t_k \text{ first become effective and then cease to be effective.} \tag{5}$$

Note that this implies that Invariant (2) is true if we let $a_k = a^t_k$, but Invariant (3) is not necessarily true since q_k might not terminate during that execution. Certainly this is the case for $t = 0$ (i.e., if a^t_k is the empty sequence of commands) because in that case $D(a_1,\ldots,a_{k-1},a^0_k)$ is the same execution as $D(a_1,\ldots,a_{k-1})$ but a_k takes no steps in this execution.

However, if we can continuously extent the command sequence a^t_k such that (4) and (5) remain true, then we will eventually obtain a command sequence $a^{t^*}_k$, such that process q_k terminates in $D(a_1,\ldots,a_{k-1},a^{t^*}_k)$, too. This is because there exists an integer T such that after T rounds all processes $q_1,\ldots,q_{k-1}$ have terminated, and thus all commands in $a^{t^*}_k$ that become effective in a round later than T must be Participate commands in order not to violate (5). By weak obstruction-freedom, process q_k must eventually terminate.

So assume we have constructed $D^t = D(a_1,\ldots,a_{k-1},a^t_k)$ for some $t \ge 0$, such that (4) and (5) hold. If process q_k terminates in D^t, then we are done and we let $a_k = a^t_k$. Invariants (2) and (3) now hold.

Assume now that q_k does not terminate in D^t. Invariant (4) ensures that processes $q_1,\ldots,q_{k-1}$ act in D^t exactly as in $D(a_1,\ldots,a_{k-1})$. Hence, they all terminate. Moreover, by Invariant (5), there exists some round $z-1$ of execution D^t after which the last command of a^t_k ceases to be effective.

Consider the following round z. Let σ denote the operation that q_k is about to perform, and let $\lambda =$"critical" if and only if σ is critical when executed as the first operation of round z. Let R be the register accessed by σ. In the following we determine the next command c_{t+1} to be appended to q_k's sequence of commands. I.e., a^{t+1}_k is the concatenation of a^t_k and c_{t+1}, and $D^{t+1} = D(a_1,\ldots,a_{k-1},a^{t+1}_k)$. We then argue that

$$\forall 1 \le i < k : D^t \sim_{q_i} D^{t+1}. \tag{6}$$

This will imply Invariant (4) by transitivity of $\sim_{q_i}$. If c_{t+1} is a Wait command, we will also argue that Invariant (5) remains true. Potentially, there are three ways in which appending a command c_{t+1} to q_k's commands sequence may lead to a violation of Invariant (6):

1) σ is a write to R: then some other process q_i, $i < k$, may either read what q_k writes, or it may notice a "change in locality" of R (one of q_i's non-critical commands on R may become critical due to q_k's write).

2) σ is a write or read command and process q_k terminates: then this may change the semantics of the command sequence of some process q_i, $i < k$. This may only happen if, during the round in which q_k terminates, q_i's effective command is Wait(reader/writer termination), and q_i is on the verge of writing to register R. In this case, q_i's Wait command may cease to be effective in execution D^{t+1} before it ceases to be effective in D^t.

3) c_{t+1} is a Participate command, and q_k executes a non-critical write in the non-critical sub-round of round z, that will make the execution distinguishable for some process q_i, $i < k$. This cannot happen: If it does, then q_k must have executed a critical write on R in some earlier round $z' < z$, and no other process has written to R since then. It follows that q_k's write in round z' has already led to an execution that can be distinguished by process q_i from $D(a_1,\ldots,a_{k-1})$. This implies in turn that Invariant (4) was violated even before we added c_{t+1} to q_k's commands list.

We now prove that Proposition 2 precludes problem 2).

PROPOSITION 3. *Let D^* be the execution obtained from the command sequences $a_1,\ldots,a_{k-1},a^{t+1}_k$ by stopping q_k just before it is about to terminate (if it terminates). If $D^t \sim_{q_i} D^*$ for all $1 \le i < k$, then $D^t \sim_{q_i} D^{t+1}$ for all $1 \le i < k$.*

PROOF. From the definition of D^*, it is not possible that q_k's termination in D^* causes a Wait(reader/writer termination) command by process q_i, $i < k$, to become effective. If q_k is not stopped in D^*, then $D^* = D^{t+1}$. Assume, then, that q_k is stopped in D^* just before it terminates. It is enough to show that q_k terminates in D^* only after $q_1,\ldots,q_{k-1}$ terminate. By Invariant (3), all processes q_i, $i < k$, terminate in D^t and, since $D^t \sim_i D^*$, they terminate also in D^*. We are therefore under the conditions of Proposition 2. Applying Proposition 2, we get that processes $q_1,\ldots,q_{k-1}$ terminate in D^* before process q_k executes its order operation. □

We now describe how c_{t+1} is determined. The following cases exist.

Case A: $\lambda =$**"critical"**. Then σ will be a critical operation when it is executed. We handle read and write critical operations differently as follows.

Case A.1: σ is a read operation. Then c_{t+1} is set to be Participate(late). Thus, in D^{t+1}, σ is added to the read sub-round of round z and q_k's following non-critical operations are added to round z's non-critical sub-round. Clearly, σ cannot be observed by other processes. Also, from Definition 3, σ cannot change the criticality of other operations. Hence, Invariant (6) holds and therefore also Invariant (4). Since c_{t+1} is not a Wait command, and since q_k does not terminate in D^t, Invariant (5) holds vacuously.

Case A.2: σ is a write operation. We consider several subcases, depending on execution D^t. In what follows, we let q^W_{late} denote the first process that writes to R in the late-write sub-round of a round $z^W_{late} \geq z$ in D^t. If no such process exists, we denote $z^W_{late} = \infty$ and say that q^W_{late} is undefined.

Case A.2.1: $z^W_{late} = z$. Then we set c_{t+1} to Participate(early). Consider execution D^{t+1}. In round z, process q_k writes to register R in the early-write sub-round. In the same round, q_k's write will be overwritten by process q^W_{late} that writes to R in the late-write sub-round. Since no reads occur in the early- or late-write sub-rounds, no process ever reads the value written by q_k to R. Moreover, all writes that occur in the early- and late-write sub-rounds are critical. Thus, after the late-write of process q^W_{late} to register R, R is local to q^W_{late} regardless of the write by q_k. Thus Invariant (6) holds, and therefore also Invariant (4). Finally, since c_{t+1} is not a Wait command, Invariant (5) holds vacuously.

Case A.2.2: $z < z^W_{late} < \infty$. Then c_{t+1} is a sequence of two commands: a Wait(next late-write) command followed by a Participate(early) command. Thus, process q_k waits in D^{t+1} until some process writes to register R in the late-write sub-round of some round $z' \geq z$. The Participate(early) command becomes effective in round z'. Process q^W_{late} writes in execution D^t to register R in round $z^W_{late} > z$. Clearly, as long as q_k's Wait(next late-write) command remains effective, the execution is indistinguishable from D^t to all processes, so process q^W_{late} writes to R in round z^W_{late} also in D^{t+1}. It follows that $z' = z^W_{late}$. Thus, after sub-round $z'-1$, q_k's Wait(next late-write) command ceases to be effective and its Participate(early) becomes effective. Clearly, Invariant (5) holds in this case. Moreover, after round $z'-1$ we are under the conditions of Case A.2.1. and so Invariant (4) also holds.

Case A.2.3: $z^W_{late} = \infty$.

A.2.3 (a): If none of the processes in $\{q_1, \ldots, q_{k-1}\}$ access register R in a round $z' \geq z$, then we let c_{t+1} be Participate(late). Invariants (4) and (5) clearly hold in this case.

A.2.3 (b): Assume that process q_j, $1 \leq j < k$, accesses register R during round $z' \geq z$. In this case, we must not allow process q_k to write register R since we cannot "hide" this write. Therefore, q_k has to wait until the next process that will read R terminated. We set c_{t+1} to be Wait(reader/waiter termination). Since this is a Wait command, Invariant (4) holds. Additionally, it is clear that the Wait command ceases to be effective after the sub-round in which q_j terminates, or even before that. Hence Invariant (5) holds.

Case B: $\lambda =$**"non-critical".** From our construction, the next command does not become effective in round z. Instead, process q_k will simply participate in the following non-critical sub-round. Therefore no command is encoded for this round. Instead, we consider the first round $z' > z$ at which q_k's next command will become effective. Such a round must exist, since by (3) all processes $q_1, \ldots, q_{k-1}$ terminate in D^t, and once they have all terminated, q_k cannot loop locally.

3.3.3 Length of the Encoding

We now bound the the number of commands in the command sequences obtained above in terms of the number of critical operations that occur in $E_\pi := D(a_1, \ldots, a_n)$.

LEMMA 4. *The total number of critical operations in execution $D = D(a_1, \ldots, a_n)$ is $\Omega(\sum_{i=1}^{n} |a_i|)$, where $|a_i|$ is the length of command sequence a_i.*

We need the following simple observations.

OBSERVATION 5. *In the sequence a_k of commands, a* Wait*(reader/writer termination) command is always followed directly either by another* Wait*(reader/writer termination) command, or by a* Participate*(late) command.*

PROOF. By construction (Case A.2.3 (b)) after a round in which a Wait(reader/writer termination) command c becomes effective for process q_k, none of the processes in $\{q_1, \ldots, q_{k-1}\}$ performs a late-write on register R. (I.e., $z^W_{late} = \infty$.) This situation cannot change until the Wait command c ceases to be effective (and not even after that). Hence, the command that follows c must be either another Wait(reader/writer termination) command or a Participate(late) command. Clearly, c is followed by another command, because q_k terminates in $D(a_1, \ldots, a_k)$ (Invariant (3)). □

OBSERVATION 6. *If k processes access the same register R during an execution, then the total number of critical operations incurred on register R is at least $k-1$.*

PROOF. It follows right away from Definition 3, that after the first process has accessed register R, the first operation of each other process on register R is a critical operation. □

The remainder of this section is devoted to the proof of Lemma 4. First of all note that every command in the command sequences takes effect in some round. (This is implicit in the construction, but even if it weren't, we could simply remove every command that does not take effect without changing the execution.) By definition of the round-scheme, process q_i executes at least one critical operation for each Participate command in a_i. Moreover, by Invariant (1), the number of Wait(next late-write) commands in a_i is bounded by the number of Participate(early) commands in a_i. Hence, it suffices to show that the total number of Wait(reader/writer termination) commands in $a_1, \ldots, a_n$ is bounded up to a constant factor by the number of critical operations.

We can associate each Wait(reader/writer termination) command with a unique pair (b,z), where b is the process in whose command sequence this command appears, and z is the round at which the command takes effect in execution D.

Let $\mathcal{W}$ be the set of all pairs (b,z) associated with a Wait(reader/writer termination) command. Further, let $\mathcal{R}$ be the set of all registers, and recall that $\mathcal{P} = \{p_1, \ldots, p_n\}$ is the set of processes, and that $q_i = p_{\pi(i)}$.

CLAIM 7. *There exist two mappings $f : \mathcal{W} \to \mathcal{P}$ and $g : \mathcal{W} \to \mathcal{R}$ such that for all $(b,z) \in \mathcal{W}$ it holds*

(a) $f(b,z) \neq b$, and

(b) in execution D, process b writes register $g(b)$ and process $f(b)$ accesses register $g(b)$, and

(c) for all $R \in \mathcal{R}$ function f is injective on $g^{-1}(R)$.

PROOF. Consider a pair $(b,z) \in \mathcal{W}$, where $b = q_k$. By construction (see Case A.2.3 above), in round z of execution D process q_k is on the verge of critically writing to a register R, and none of the processes in $\{q_1,\ldots,q_{k-1}\}$ writes to register R in a late-write sub-round of a round $z' > z$. Also by the assumption of that case, there exists a process in $\{q_1,\ldots,q_{k-1}\}$ that accesses register R in a round $z' > z$. Among those, let q_j be the process that terminates first. We define $f(b,z) := q_j$ and $g(b,z) := R$. Obviously (a) and (b) are true because $q_j \neq b$ (by construction, b waits until q_j terminates), q_j accesses register R and b writes to register R.

We now show that f is injective on $W_R = g^{-1}(R)$. For the purpose of a contradiction assume that there exist two pairs (q_i,z_i) and (q_j,z_j) and a process $q_k \in \mathcal{P}$, such that $g(q_i,z_i) = g(q_j,z_j) = R$ and $f(q_i,z_i) = f(q_j,z_j) = q_k$. Let w_i and w_j be the two Wait(reader/writer termination) commands associated with the two pairs.

First assume that $i = j$, w.l.o.g. $z_j > z_i$. Hence, process q_i's Wait command w_i ceases to be effective before the sub-round in which its Wait command w_j takes effect. But by definition of f, w_i ceases to be effective right after the round in which process q_k terminates. Hence, when w_j becomes effective, process q_k has already terminated, so $f(q_j,z_j)$ cannot be that process—a contradiction.

Now assume that $i \neq j$, w.l.o.g. $i < j$. Then the assumption $f(q_i) = f(q_j) = q_k$ implies that q_k terminates in a round z', where $z' > z_i$ and $z' > z_j$, and the Wait commands w_i and w_j remain effective until then. By Observation 5 the commands w_i and w_j are both followed by a (possibly empty) sequence of additional Wait(reader/writer-termination) commands, and eventually one Participate(late) command. Hence, there exists a round $z_i^* \geq z' > \max\{z_j,z_i\}$ in which process q_i is writing to register R in a late-write sub-round.

Now consider the situation when we added command w_j to the command sequence of process q_j. Let a_j^t be the commands that appear in that command sequence before w_j. In execution $D(a_1,\ldots,a_{j-1},a_j^t)$, process q_i is acting exactly in the same way as in execution D (because of indistinguishability). When we added the Wait(reader/writer termination) command w_j to the sequence a_j^t, then we must have been in Case A.2.3 (b) for $z = z_j$. But the fact (recall $i < j$) that q_i is writing in execution $D(a_1,\ldots,a_{j-1},a_j^t)$ in round $z_i^* > z_j$ to register R in a late-write sub-round, means that we are actually in a case where $z_{late}^W \leq z_i^* < \infty$. This contradicts the assumption of Case A.2.3 (b). □

We argue that the claim implies that the number of critical operations during D is $\Omega(|\mathcal{W}|)$. Let $W_R = g^{-1}(R)$ for $R \in \mathcal{R}$. Since $\mathcal{W}$ is the disjoint union of all W_R, $R \in \mathcal{R}$, it suffices to show that the total number of critical operations on register R is $\Omega(|W_R|)$. Assume that $|W_R| > 0$ and let $Q = f(W_R)$. By (b), all processes in $Q \cup W_R$ access register R during execution D. Hence, by Observation 6 the total number of critical operations on register R is at least $|Q \cup W_R| - 1$. By (a), $|Q \cup W_R| \geq 2$, and by injectivity of f on W_R (c), we have $|Q| = |W_R|$. Hence, the total number of critical operations on register R is at least

$$|Q \cup W_R| - 1 \geq \max\{2, |W_R| - 1\} = \Omega(|W_R|).$$

PROOF OF THEOREM 1. Let E_π be the execution $D(a_1,\ldots,a_n)$ we have constructed for permutation π. By Proposition 2, the record-vector of E_π is the same as that of $E_{sol}(L_\pi)$. By Definition 1, the record-vectors $\vec{R}[E_{sol}(L_\pi)]$ are different for all permutations π. Hence, the record-vectors $\vec{R}[E_\pi]$, for $\pi \in S_n$, are all different. Since there are $n!$ permutations, there exists a permutation π, such that the encoding of $E_\pi = D(a_1,\ldots,a_n) = C(s_1,\ldots,s_n)$ uses at least $\log(n!) = \Omega(n\log n)$ bits. Since each encoding of a permutation consists of n sequences $s_1,\ldots,s_n$ of commands, and for each command there are only four possibilities, it follows that $|s_1| + \cdots + |s_n| = |a_1| + \cdots + |a_n| = \Omega(n\log n)$. By Lemma 4, the number of critical operations executed in E_π is $\Omega(n\log n)$. □

4. LOWER BOUND IN THE DSM MODEL

In this section we provide a high-level description of how the proof of Theorem 1, presented in Section 3 for the CC model, can be adapted in a simple manner to obtain an equivalent result for the Distributed Shared Memory (DSM) model. The complete proofs are deferred to the full paper.

Recall that the proof of Theorem 1 for the CC model uses the concept of critical operations. While the number of CC critical operations (as given in definition 3) is, in general, smaller than the number of RMRs, for the DSM model we define each RMR as a critical operation.

DEFINITION 5. *An operation σ of process p on register R is* critical *in a DSM system, if R is remote to p's processor.*

The general structure of the DSM proof is identical to that of the CC proof. A set of executions $\{E_\pi | \pi \in S_n\}$ is constructed and maintains the same properties maintained by the CC executions. The key change is in the set of commands used for execution encoding. Recall that two types of Wait commands are used for encoding CC executions: Wait(next late-write) and Wait(reader/writer termination). A third type of Wait commands is used for DSM encodings, motivated by the need to prevent information flow from "later" processes to "earlier" processes. Fix a permutation $\pi \in S_n$ and consider the steps taken by q_j, the j'th process by π. Assume that a set of earlier processes, $Q \in \{q_1,\ldots,q_{j-1}\}$, each access q_j's local registers. For E_π to possess the required properties, we schedule q_j's steps so that no process in Q reads a value written by q_j. Specifically, we must prevent processes in Q from reading *local writes* made by q_j. Unlike in the CC proof, here we cannot enforce this by encoding q_j's accesses of local registers since then the encoding length may asymptotically exceed the number of critical steps. Instead, we delay scheduling of q_j's steps until after all the processes in Q terminate. We use a new Wait(early-process-termination) command to enforce this. The code of process q_j therefore begins with k such wait commands, where $k = |Q|$.[2]

Another DSM scheduling requirement (symmetric, in a sense, to the previous one) is that we must not allow a process to communicate information to earlier processes via *their* local segments. If the next access by q_j is a write to a register R local to q_i, for some $i < j$, and if q_i accesses R in its execution, then q_i must terminate before q_j is allowed to proceed. This requirement is already met, however, by the encoding and execution construction for the CC model, since in such a case a Wait(reader/writer termination) command is appended to q_j's code on account of q_i's access of R. (See Case A.2.3 (b) in Section 3.3.2.)

The proofs of invariants 1-6 are similar to those for the CC model; they are actually somewhat simpler, since the locality of registers is fixed. The following lemma is the DSM equivalent of Lemma 4.

LEMMA 8. *The total number of critical operations in a DSM execution $D = D(a_1,\ldots,a_n)$ is $\Omega(\sum_{i=1}^n |a_i|)$, where $|a_i|$ is the length of command sequence a_i.*

[2]A single parameterized Wait(early-process-termination,k) command could have been used. We chose the non-parameterized encoding version since it simplifies our proofs.

PROOF SKETCH. The proof is very similar to that of Lemma 4 and, likewise, uses Observations 5 and 6. The proof of Observation 5 works for the DSM model without change. Observation 6 holds trivially in the DSM model, since each register is local to a single process and remote to all others. The rest of the proof is identical, except that we should now also prove that the addition of Wait(early-process-termination) commands does not not violate the claim. This follows easily from the fact that each Wait(early-process-termination) added to q_j's code on account of an access by q_i, for some $i < j$, can be amortized against the first critical step by q_i that accesses q_j's local segment. $\square$

THEOREM 9. *Let $(A_1, \ldots, A_n)$ be order encoding DSM algorithms that satisfy weak obstruction-freedom. Then there is an execution $E \in \mathcal{E}$ with $\Omega(n \log n)$ RMR cost.*

5. APPLICATIONS

5.1 Mutual Exclusion

The mutex object supports two operations: *Enter* and *Exit*. The *Enter* operation allows a process to enter the *critical section*; after completing the critical section, a process invokes an *Exit* operation. Any implementation of a mutex object must meet the following requirements: (1) at most a single process can be at its critical section in any given time (*mutual exclusion*), (2) if some process invokes *Enter* then, eventually, some process enters the critical section (*deadlock freedom*), and (3) a process completes the *Exit* procedure in a finite number of steps (*finite exit*).

We show that the algorithms A'_i, given in the pseudo-code below, are order encoding. Algorithms A'_i, $1 \le i \le n$ use one deadlock-free mutex object *ME*, implemented by algorithms $A_1, \ldots, A_n$, a shared register *counter* initialized to 0, and local register ℓ_i.

Algorithm 1: Pseudo-code of the algorithm A'_i.

```
1  ME.Enter;
2  $\ell_i$ ←READ(counter);WRITE($\ell_i + 1$ →counter);
3  order-operation ;
4  ME.Exit;
5  RECORD($\langle \ell_i \rangle$);
```

LEMMA 10. *Algorithms $A'_1, \ldots, A'_n$ are order-encoding.*

PROOF. Let $\pi \in S_n$ be a permutation and $L^k_\pi = (p_{\pi(1)}, \ldots, p_{\pi(k)})$. To prove property (a) of Definition 1, note that, in all executions, the i-th process entering the critical section records value i. Thus, in the L^k_π-solo execution $E_{solo}[L^k_\pi]$, process $p_{\pi(i)}$ records value i, for $1 \le i \le k$. Now let $1 \le i < j \le k$. In an execution E, where $p_{\pi(j)} \prec_E p_{\pi(i)}$, process $p_{\pi(j)}$ enters the critical section before $p_{\pi(i)}$ does and so records a smaller value than is recorded by $p_{\pi(i)}$. Hence, $\vec{R}[E] \neq \vec{R}[E_{solo}(L^k_\pi)]$.

To prove property (b), note that in any execution in which the processes $p_{\pi(1)}, \ldots, p_{\pi(k)}$ participate, the record vector is a permutation of $\{1, \ldots, k\}$. Thus, if the record vectors of two executions E and E' in which the processes $p_{\pi(1)}, \ldots, p_{\pi(k)}$ participate differ, then they must differ in at least two components. $\square$

The definitions imply that a deadlock-free mutual-exclusion implementation satisfies *weak obstruction-freedom*, and hence, so do algorithms $A'_1, \ldots, A'_n$. We get:

THEOREM 11. *Any deadlock-free implementation of mutual exclusion from read, write, and conditional operations has an execution whose RMR cost is $\Omega(n \cdot \log n)$.*

PROOF. Since algorithms $A'_1, \ldots, A'_n$ satisfy weak obstruction-freedom, by Lemma 10 and Theorem 1, there is an execution E' of $A'_1, \ldots, A'_n$ with an RMR cost of $\Omega(n \cdot \log n)$. Let E be the execution obtained from E' by removing all the steps performed by $A'_1, \ldots, A'_n$ that are not performed by $A_1, \ldots, A_n$. In E', each process performs at most 2 RMRs more than it does in E. It follows that E is an execution of $A_1, \ldots, A_n$ whose RMR cost is $\Omega(n \cdot \log n)$. $\square$

5.2 Bounded Counters

A b-bounded counter is an object that supports the operations *Increment* and *Reset*. The object stores a value in $\{0, \ldots, b\}$ and is initialized to 0. The operation *Increment* atomically increments the value of the object, unless it already has value b (in which case the value is not changed), and returns its previous value. The operation *Reset* resets the value of the object to 0 and returns the previous value. Using our lower bound technique, we show that the RMR cost of calling *Increment* and *Reset* once is $\Omega(n \cdot \log n)$.

We prove that the algorithms B_i, given in pseudo-code below, are order-encoding. Algorithms B_i, $1 \le i \le n$, use a single b-bounded counter object BC, for some $b \ge 2$, a shared integer *counter* initialized to 0, and registers ℓ_i and r_i.

Algorithm 2: Pseudo-code of the algorithm B_i.

```
1  $\ell_i \leftarrow -1$;
2  $r_i$ ←BC.Increment;
3  order-operation ;
4  if $r_i = 0$ then
5      $\ell_i$ ←READ(counter);
6      $\ell_i \leftarrow \ell_i + 1$;
7      WRITE($\ell_i$ → counter);
8      $r_i$ ← BC.Reset;
9  end
10 RECORD($\langle \ell_i, r_i \rangle$);
```

LEMMA 12. *Algorithms $B_1, \ldots, B_n$ are order-encoding.*

PROOF. Let $\pi \in S_n$ be a permutation, let $L^k_\pi = (p_{\pi(1)}, \ldots, p_{\pi(k)})$ and let E be an execution in which only the processes $p_{\pi(1)}, \ldots, p_{\pi(k)}$ participate. Clearly, in $E_{sol}(L^k_\pi)$ participating processes obtain response 0 from the *Increment* operation in line 3 and then obtain response 1 from the *Reset* operation in line 8. It follows that the value recorded by process $p_{\pi(j)}$ in $E_{sol}(L^k_\pi)$ is $\langle j, 1 \rangle$, for $j = 1, \ldots, k$.

We now prove Property (a) of Definition 1. Let E be an arbitrary execution and $1 \le i < j \le k$. For the purpose of a contradiction assume that $p_{\pi(j)} \prec_E p_{\pi(i)}$, and that the record vector of E is the same as that of $E_{sol}(L^k_\pi)$. Then the *Increment* operation must return 0 for both processes, $p_{\pi(i)}$ and $p_{\pi(j)}$, or otherwise one of them would record $\langle -1, x \rangle$ for $x \in \{1, \ldots, b\}$. But then $p_{\pi(i)}$ cannot execute its order-operation before $p_{\pi(j)}$ resets the counter in line 8. Hence, if $p_{\pi(i)}$ records $\langle a_1, 1 \rangle$ and $p_{\pi(j)}$ records $\langle a_2, 1 \rangle$, then $a_1 > a_2$. This is different from the record-vector produces by $E_{sol}(L^k_\pi)$, where $p_{\pi(i)}$ records $\langle i, 1 \rangle$ and $p_{\pi(j)}$ records $\langle j, 1 \rangle$.

To prove Property (b), assume that in execution E the record $\langle x, y \rangle$ of process $p_{\pi(k)}$ is different from $\langle k, 1 \rangle$. We show that there exists another process $p_{\pi(j)}$, $j \neq k$, whose record is different from $\langle j, 1 \rangle$. If $x = -1$, then process $p_{\pi(k)}$'s *Increment* operation in line 2 occurs after some other process $p_{\pi(j)}$ increments the counter BC from 0 to 1, and before $p_{\pi(j)}$'s *Reset* operation in line 8. Thus when that *Reset* occurs, then the counter has a value of b, and $p_{\pi(j)}$'s record is 2 in the second component.

If $x \neq k$ and $x \neq -1$, $p_{\pi(k)}$ is the x-th process that executes line 6, and not the k-th. Let $j = x$. Clearly, if process $p_{\pi(j)}$ executes line

6, then the first component of its record is not j, and if it does not execute line 6, then the first component of its record is -1.

Finally, assume that $x = k$ but $y \neq 1$. Since $x \neq -1$, process $p_{\pi(k)}$ increments the counter from 0 to 1 in line 2, and no other process can reset it until $p_{\pi(k)}$ executes line 8. Since $y \neq 1$, this *Reset* operation returns 2 (and thus $y = 2$), and some other process $p_{\pi(j)}$ increments the counter after $p_{\pi(k)}$ executes line 2 and before $p_{\pi(k)}$ executes line 8. But then $p_{\pi(j)}$'s *Increment* operation returns 1, and $p_{\pi(j)}$'s record is $\langle -1, 1 \rangle$. $\square$

Since the algorithms $B_1, \ldots, B_n$ satisfy weak obstruction-freedom, we obtain with the same arguments as in the proof of Theorem 11:

THEOREM 13. *Any weak obstruction-free implementation of a b-bounded counter, for $b > 2$, from read and write operations has an execution in which each process calls* Increment *and* Reset *once, whose RMR cost is* $\Omega(n \cdot \log n)$.

5.3 Store/Collect Objects

A one-shot (single-writer) store/collect object supports the operations *Store* and *Collect*. The object has n components, initialized to $\perp$; each process can either store a value in its associated component or collect the values from all components; a collect should not miss a value of a preceding store nor include the value of a store that has not yet begun.

Formally, the object supports two operations: Store(v) by process p_i makes v be the value stored by p_i, while Collect$()$ returns a vector V of values, one for each process. If a Collect operation returns V and p_i is a process, then $V[i] = \perp$ only if no Store operation by p_i completes before the Collect; if $V[i] = v \neq \perp$, then v is the parameter of a Store operation *op* by p_i that does not follow the Collect.

We use our lower bound technique, to prove that the RMR cost of calling store and collect once is $\Omega(n \cdot \log n)$.

We show that the algorithms given in pseudo-code below, are order-encoding. Algorithms C_i, $1 \leq i \leq n$, use a single weak obstruction-free store/collect object, CL, and registers ℓ_i and r_i.

Algorithm 3: Pseudo-code of the algorithm C_i.

```
1 CL.Store(i); order-operation ;
2 $r_i$ ←CL.Collect; $\ell_i$ ←number of non-$\perp$ entries in $r_i$;
3 RECORD($\langle \ell_i \rangle$);
```

LEMMA 14. *Algorithms $C_1, \ldots, C_n$ are order-encoding.*

PROOF. Let $\pi \in S_n$ be a permutation, let $L^k_\pi = (p_{\pi(1)}, \ldots, p_{\pi(k)})$ and let E be an execution in which only the processes $p_{\pi(1)}, \ldots, p_{\pi(k)}$ participate. Clearly, in $E_{sol}(L^k_\pi)$, process $p_{\pi(j)}$, $1 \leq j \leq k$, collects non-$\perp$ values exactly from processes $p_{\pi(1)}, \ldots, p_{\pi(j)}$. Hence, the value recorded by process $p_{\pi(j)}$ in $E_{sol}(L^k_\pi)$ is $\langle j \rangle$, for $j = 1, \ldots, k$.

We now prove Property (a) of Definition 1. Let E be an arbitrary execution and $1 \leq i < j \leq k$, such that $p_{\pi(j)} \prec_E p_{\pi(i)}$; assume that i and j are the minimal indices with this property. Then process $p_{\pi(i)}$ reads a non-$\perp$ value from process $p_{\pi(j)}$, and since i is minimal it follows that $p_{\pi(i)}$ reads non-$\perp$ values from $p_{\pi(1)}, \ldots, p_{\pi(i)}$.

To prove Property (b), assume that in execution E process $p_{\pi(k)}$ records a value different than k, i.e., $p_{\pi(k)}$ misses the value stored by some process $p_{\pi(i)}$, $1 \leq i < k$. The definition of a store / collect object implies that the collect of $p_{\pi(k)}$ starts before the store of $p_{\pi(i)}$ completes and hence before the collect of $p_{\pi(i)}$ starts. Let $p_{\pi(j)}$, $1 \leq j < k$, be the process whose collect starts last in E, then clearly, $p_{\pi(j)}$ records $k \neq j$, and Property (b) follows. $\square$

Since the algorithms $C_1, \ldots, C_n$ satisfy weak obstruction-freedom, we get in a manner similar to Theorem 11:

THEOREM 15. *Any weak obstruction-free implementation of a store/collect object, from read, write and conditional operations has an execution in which each process calls* Store *and* Collect *once whose RMR cost is* $\Omega(n \cdot \log n)$.

6. SUMMARY

We used a novel technique for proving lower bounds on the RMR cost for a variety of synchronization problems to derive an $\Omega(n \log n)$ lower bound for mutual exclusion, bounded counters and store/collect. Further research is to find additional applications for our technique or extend it to accommodate randomization and stronger primitives.

Acknowledgements

We would like to thank the referees for their helpful comments.

7. REFERENCES

[1] R. Alur and G. Taubenfeld. Results about fast mutual exclusion. RTSS 1992, pp. 12–21.

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

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

[4] J. H. Anderson and Y.-J. Kim. Fast and scalable mutual exclusion. DISC 1999, pp. 180–194.

[5] J. H. Anderson and Y.-J. Kim. Adaptive mutual exclusion with local spinning. DISC 2000, pp. 29–43.

[6] J. H. Anderson and Y.-J. Kim. Nonatomic mutual exclusion with local spinning. PODC 2002, pp. 3–12.

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

[8] R. Cypher. The communication requirements of mutual exclusion. SPAA 1995, pp. 147–156.

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

[10] W. M. Golab, V. Hadzilacos, D. Hendler, and P. Woelfel. Constant-RMR implementations of CAS and other synchronization primitives using read and write operations. PODC 2007, pp. 3–12.

[11] M. Herlihy, V. Luchangco, and M. Moir. Obstruction-free synchronization: Double-ended queues as an example. ICDCS 2003, pp. 522–529.

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

[13] Y.-J. Kim and J. Anderson. A time complexity bound for adaptive mutual exclusion. DISC 2001, pp. 1–15.

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

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

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

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