Laws of Order: Expensive Synchronization in Concurrent Algorithms Cannot be Eliminated

Hagit Attiya  
Technion  
hagit@cs.technion.il

Rachid Guerraoui  
EPFL  
rachid.guerraoui@epfl.ch

Danny Hendler  
Ben-Gurion University  
hendlerd@cs.bgu.ac.il

Petr Kuznetsov  
Deutsche Telekom Labs/TU Berlin  
pkuznets@acm.org

Maged M. Michael  
IBM T. J. Watson Research Center  
magedm@us.ibm.com

Martin Vecchev  
IBM T. J. Watson Research Center  
mtvechev@us.ibm.com

Abstract
Building correct and efficient concurrent algorithms is known to be a difficult problem of fundamental importance. To achieve efficiency, designers spend significant time trying to remove unnecessary and costly synchronization. However, not only is this manual trial-and-error process ad-hoc and error-prone, but it often leaves designers pondering the question of: is it inherently impossible to eliminate certain synchronization, or is it that I was unable to eliminate it on this attempt and I should keep trying?

In this paper we respond to this question. We prove that it is impossible to build correct concurrent implementations of classic and ubiquitous specifications such as sets, queues, stacks, mutual exclusion and read-modify-write operations, that completely eliminate certain expensive synchronization.

More specifically, we prove that one cannot avoid the use of: i) read-after-write (RAW), where a write to shared variable \( A \) is followed by a read to a different shared variable \( B \) or ii) atomic write-after-read (AWAR), where an atomic operation reads and then writes to shared locations. Unfortunately, enforcing any of these two patterns is expensive on virtually all mainstream processor architectures today. To enforce RAW, memory ordering—also called fence or barrier instructions—must be used. To enforce AWAR, atomic instructions such as compare-and-swap (or equivalent) are required. However, fences and atomic instructions are typically substantially slower—around 50 times—than regular instructions!

Although designers of concurrent algorithms frequently struggle to avoid RAW and AWAR, their attempts are often futile. Our result explains exactly in which cases avoiding RAW and AWAR is impossible. Failure to use such synchronization will mean that the algorithm is incorrect and there is no need to even attempt to verify its correctness. On the flip side, our result indicates on which data structures designers can focus their efforts on.

1. Introduction
The design of concurrent applications that avoid costly synchronization patterns is a cardinal programming challenge, requiring consideration of algorithmic concerns and architectural issues, and has implications to formal testing and verification.

Two common synchronization patterns that frequently arise in the design of concurrent algorithms are read after write (RAW) and atomic write after read (AWAR). The RAW pattern consists of a write to some shared variable \( A \), followed by a read to a different shared variable \( B \). The AWAR pattern consists of a read of some shared variable followed by a write to the same or a different shared variable, where the read and the write are atomic. Examples of the AWAR pattern include read-modify-write operations, such as successful Compare-and-Swap [27] (CAS) operations.

Unfortunately, on all mainstream processor architectures, the RAW and AWAR patterns are associated with expensive instructions. Modern processor architectures use relaxed memory models, where guaranteeing RAW order among accesses to independent memory locations requires the execution of memory ordering instructions—often called memory fences or memory barriers—that enforce RAW order.2 Also guaranteeing the atomicity of the AWAR pattern requires the use of atomic instructions. Typically, RAW fence and atomic instructions are substantially slower—around 50 times—than regular instructions, even under the most favorable caching conditions.

Due to these high overheads, designers of concurrent algorithms aim to avoid both the RAW and AWAR patterns, if possible. However, such attempts are often very time-consuming and unsuccessful: in many cases, after multiple empirical attempts, it turns out that it is impossible to avoid these patterns while ensuring correctness of the algorithm.

This raises an interesting and important practical question: Can we discover, formalize and prove the conditions under which at-

---

1 CAS operates on a single shared variable and takes as arguments an expected value and a new value. It atomically (1) reads the value of the variable; (2) compares the read value with the expected value; and (3) if equal then it writes the new value to the variable. It returns a Boolean value that indicates whether or not it succeeded, i.e., whether or not the write occurred.

2 RAW order requires the use of explicit fences or atomic instructions even on strongly ordered architectures (e.g., X86 and SPARC TSO) that automatically guarantee other types of ordering (read after read, write after read, and write after write).
tempts by algorithm designers to avoid the RAW and AWAR patterns, while ensuring algorithm correctness, are actually futile?

In this paper, we answer this question in a way that formally captures what were previously only empirical observations. We show that implementations of a wide class of concurrent algorithms must involve the expensive RAW or AWAR patterns.

We focus on two fundamental classes of specifications that are heavily used in practice: linearizability and mutual exclusion. Roughly speaking, our results state that it is impossible to build a linearizable RAW-AWAR-free concurrent algorithm of a non-commutative operation. Similarly, it proves that it is impossible to build a RAW-AWAR-free correct mutual exclusion algorithm. Both of these results are stated and elaborated on with formal rigor in the rest of the paper.

Our results are widely applicable as they talk about specifications, and not of particular implementations. That is, they are applicable to any implementation of the said specifications.

**Main Contributions.** The main contributions of the paper are the following:

- We define the class of specifications that our results apply to: deterministic sequential specifications and strongly non-commutative operations. We prove that it is impossible to build a linearizable implementation of such specifications, that is RAW-AWAR-free.
- We prove that many common operations on ubiquitous and fundamental abstract data types—such as sets, queues, work-stealing queues, stacks, and read-modify-write objects—satisfy our conditions on the specification and hence are subject to our results.
- We prove that it is impossible to build an algorithm that satisfies mutual exclusion, is deadlock-free and RAW-AWAR-free.

**Practical Implications.** Our results have important practical implications: it guides algorithm designers, suggests targeted improvements in hardware and can be used in tandem with classic program testing and verification:

- Designers of concurrent algorithms can use our results to determine when looking for a correct RAW-AWAR-free design is futile. Conversely, our results indicate when avoidance of these expensive patterns may be possible. Further, our results state exactly what changes in the semantics of the specification of the target algorithmic operations may make them amenable for a correct RAW-AWAR-free design.
- For processor architects, this result indicates the importance of optimizing the performance of atomic operations such as compare-and-swap, and in particular RAW fence instructions, which have historically received little attention for optimization.
- For formal testing and verification of concurrent algorithms, it is possible to use our result as a filter: if the algorithm does not contain RAW and AWAR (regardless of whether the architecture is sequentially consistent or not), then it is certainly incorrect and there is no need to even attempt to test it or verify it. Otherwise, we proceed as usual with standard testing and verification.

The remainder of the paper is organized as follows. We present an overview of our results with illustrative examples in Section 2. In Section 3, we present the formal model for our results. We present our main results for mutual exclusion in Section 4 and for linearizable objects in Section 5. In Section 6, we show that many widely used specifications fall into our class. We discuss related work in Section 7 and conclude the paper with Section 8.

In Section 3, we present the formal model for our results. We present our main results for mutual exclusion in Section 4 and for linearizable objects in Section 5. In Section 6, we show that many widely used specifications fall into our class. We discuss related work in Section 7 and conclude the paper with Section 8.

Let us illustrate these concepts on a simple example: a Hoare-style sequential specification of a classic Set, shown in Fig. 1.

First, this simple sequential specification is deterministic: if an add, remove and contains execute from a given state, they will always return the same result.

Second, both operations, add and remove are strongly non-commutative. That is, there exists an execution of the specification such that add can influence the result of add and remove can influence the result of remove. For example, let us begin with $S = \emptyset$. Then, if we perform an add(5), it will return true and a subsequent second add(5) will return false. However, if we change the order, and the second add(5) executes first, then it will return true while the first add(5) will return false. That is, add is a strongly non-commutative operation, as there exists another operation (in this case another add, but in general as we define later, it could be another operation) such that the two operations influence each other’s result. Similar reasoning applies to remove. However, contains is not a strongly non-commutative operation, as

$$\begin{align*}
\{S = \mathcal{A}\} & \text{ contains}(k) \quad \{ret = k \in A \land S = \mathcal{A}\} \\
\{S = \mathcal{A}\} & \text{ add}(k) \quad \{ret = k \notin A \land S = \mathcal{A} \cup \{k\}\} \\
\{S = \mathcal{A}\} & \text{ remove}(k) \quad \{ret = k \in A \land S = A \setminus \{k\}\}
\end{align*}$$

Figure 1. Sequential specification of a set. $S \subset \mathbb{N}$ denotes the contents of the set. $ret$ denotes the return value.

2. **Overview**

In this section, we first informally explain our result and then demonstrate its implications via several well-known concurrent algorithms. The discussion in this section is mostly informal. Formal details are provided in subsequent sections.

Our result focuses on two important practical specifications: mutual exclusion and linearizability [24].

**Mutual Exclusion.** The first part of the result states that it is impossible to design a deadlock-free algorithm where all executions of the algorithm are RAW-AWAR-free (i.e. efficient) and the algorithm satisfies the mutual exclusion specification [12, 31] (i.e. correct). By deadlock-free mutual exclusion specification, we mean that there cannot be more than one process inside a critical section at the same time, and if one or more processes compete for a critical section, at least one of them succeeds.

**Linearizability.** The second part of our result discusses linearizability [24]. Given a deterministic sequential specification Spec, and an unordered strongly non-commutative operation op of that specification, it says that we cannot design an algorithm where all executions of the algorithm are RAW-AWAR-free and the algorithm is linearizable with respect to the given sequential specification Spec.

Intuitively, an algorithm is linearizable with respect to a given sequential specification if each execution of the algorithm is equivalent to some sequential execution of the specification, where the order between the non-overlapping operations is preserved. The equivalence is defined by comparing the arguments and results of operations.

Informally, by a deterministic sequential specification we mean that if an operation executes from a given state, it will always return the same result. By a strongly non-commutative operation, we mean that in the specification, the operation op$_1$ can influence the result of another operation op$_2$ and op$_3$ can influence the result of op$_4$. By unordered we mean that the said op$_1$ and op$_2$ operations are performed by different processors (but still sequentially).

Let us illustrate these concepts on a simple example: a Hoare-style sequential specification of a classic Set, shown in Fig. 1.
its result can be influenced by a preceding add or remove, but its execution does not influence the result of any other operation.

Third, the specification may allow an operation to be executed by many processes, or only by a single process. For instance, in the case of work-stealing queues which we will discuss later, some operations can only be executed by a single process.

For the Set specification from Fig. 1, our result will state that any linearizable implementation of add and remove (i.e. the strongly non-commutative operations) must use RAW or AWAR in some sequential execution of the implementation. However, our result does not apply to the contains operation which intuitively makes sense as contains does not modify the shared state and should not require synchronization.

Implications While our result shows when expensive synchronization cannot be eliminated correctly, it is also helpful in guiding designers towards cases where they may be successful. In particular, a designer may focus on either of the following directions:

- **Determinism**: change the sequential specification, perhaps by considering non-deterministic specifications.
- **Correctness**: change the correctness criteria, perhaps by considering an alternative to linearizability.
- **Commutativity**: focus on operations that are not strongly non-commutative, i.e. contains instead of an add.
- **Ownership**: restrict the specification of an operation such that the operation can only be performed by a single process, instead of multiple processes.
- **Detectors**: come up with efficient detectors that can identify executions which are known to be commutative and hence can avoid the expensive synchronization in that case.

The first four of these pertain to the specification and the correctness criteria and we illustrate two of them (determinism and ownership) in the examples that follow.

The last one is focused on the implementation. To elaborate on the implementation point further, observe that our results talk about executions of operations, and namely those that are strongly non-commutative as illustrated in the example with the two add (5)’s executed sequentially. Suppose however that the sequence of two add (5)’s starts from a set that contains 5. Then, regardless of the order, both adds will always return false. If a designer is able to identify such cases (which adds more work on the critical path), then it might be possible to avoid the synchronization in some but not all executions of add. In practice however, this is difficult to realize as one would need to examine the state of the data structure non-atomically and be aware of potential overlapping operations.

Next, we illustrate the applicability of our result in practice via several well-known concurrent algorithms.

### 2.1 Compare and Swap

We begin with the universal compare-and-swap (CAS) construct, whose sequential specification is deterministic, and the operation is strongly non-commutative (for a formal proof, see Section 6). The sequential specification of CAS(v, o, n) says that it first compares v to o and if v = o, then n is assigned to v and CAS returns true. Otherwise, v is unchanged and CAS returns false. Here we use the * operator to denote address dereference.

The CAS specification can be implemented trivially with a linearizable algorithm that uses an atomic hardware instruction (also called CAS) and in that case, the implementation inherently includes the AWAR pattern.

Alternatively, CAS specification can be implemented by a linearizable algorithm using reads, writes, and hardware CAS, with the goal of avoiding the use of the hardware CAS in the common case of no contention. Such a linearizable algorithm is presented by Luchangco et al. [34]. Fig. 2 shows an adapted code snippet of the common path of that algorithm. While the algorithm succeeds in avoiding the AWAR pattern in the common case, it must include the RAW pattern in its common path. The write to b.X in line 16 must precede the read of b.X in line 17 or otherwise the algorithm will be incorrect.

Both examples confirm our result: AWAR or RAW was necessary. Of course, knowing that RAW and AWAR cannot be avoided in implementing CAS correctly is important as CAS is a fundamental building block for many classic concurrent algorithms.

### 2.2 Mutual Exclusion Locks

For mutual exclusion lock implementations, our result proves that a RAW or AWAR is required for the entry section.

The test-and-set is the most common lock implementation. Its lock acquire operation boils down to an AWAR pattern, by using an atomic operation, e.g., CAS, to atomically read a lock variable, check that it represents a free lock, and if so replace it with an indicator of a busy lock. Fig. 3 shows a simplified version of a test-and-set-lock. Similarly for all other locks that require the use of read-modify-write atomic operations in every lock acquire [2, 19, 36].

On the other hand, a mutual exclusion lock algorithm that avoids the AWAR pattern [6, 12, 41], must include the RAW pattern. For example, Fig. 4 shows a simplified snippet from the entry section of Dekker’s 2-way mutual exclusion algorithm.

```c
bool WFCAS(Val ev, Val nv) {
14:   if (ev == nv) return WFRead()==ev;
15:   Blk b = L;
16:   b.X = p;
17:   if (b.Y) goto 27;
... 

Figure 2. Adapted snippet from Luchangco et al.’s [34] wait-free CAS algorithm.

while (~CAS(Lock,FREE,BUSY));

Figure 3. A simplified snippet of a test-and-set lock acquire.

1:   flag[me] = true;
2:   while flag[~me] = true {
... 

Figure 4. A simplified snippet from the entry section of Dekker’s 2-way mutual exclusion algorithm.
```

### 2.3 Work Stealing Structures

Concurrent work stealing algorithms are highly popular in implementing various load balancing frameworks.

A work stealing structure holds a collection of work items and it has a single process as its owner. It supports three main
Other processes (thieves) may extract work items using an operation (i.e., the safety requirements of work stealing are that (1) each extract operation (i.e., put, take, and steal) that succeeds in extracting work items must return a valid work item, i.e., one that was previously put by the owner, and that (2) each inserted item is eventually extracted exactly once. In Section 6, we provide a formal proof for why the take and steal operations are strongly non-commutative operations.

In designing algorithms for work stealing, the highest priority is to optimize the owner’s operations, especially the common paths of such operations, as they are expected to be the most frequently executed parts of the operations. Examining known work stealing algorithms that avoid the AW AR pattern (i.e., avoid the use of complex atomic operations) in the common path of the owner’s operations, reveals that they all contain the RAW pattern in the common path of take operation that succeeds in extracting work items.

The work stealing algorithm by Chase and Lev [11] is representative of such algorithms [3, 17, 20, 21]. Fig. 5 shows a code snippet adapted from the common path of the take operation of that algorithm, with minor changes for the sake of consistency in presentation. The variables bottom and top are shared variables, and bottom is written only by the owner but may be read by other processes. Hence, there is no benefit from using single location complex atomic operations to operate on bottom. The key pattern in this code snippet is the RAW pattern in lines 4 and 5. The order of write to bottom in line 4 followed by the read of top in line 5 is necessary for the correctness of the algorithm.

Indeed, our result captures this fact and also the fact that reversing the order of these two instructions results in an incorrect algorithm.

**From deterministic to non-deterministic specifications** Our result dictates that in the standard case where we have the expected deterministic sequential specification of a work-stealing structure, it is impossible to avoid RAW and AW AR. However, as mentioned earlier, our result can guide us towards finding practical cases where we can indeed eliminate RAW and AW AR. One such relaxation is to allow non-deterministic specifications. Such a relaxation is exemplified by the idempotent work stealing introduced by Michael et al. [38]. This concept relaxes the semantics of work stealing to require that each inserted item is eventually extracted at least once instead of exactly once. Under this notion the authors managed to design algorithms for idempotent work stealing that avoid both the AW AR and RAW patterns in the owner’s operations.

Our result explains the underlying reason of why elimination of RAW and AW AR was possible: because we cannot write a deterministic sequential specification for the idempotent structures, our result now indicates that it may be possible to avoid RAW and AW AR which is substantiated by the algorithms in [38]. Fig. 6 shows the take operation of one of the idempotent algorithms (uses FIFO order). Note the absence of both the AW AR and RAW patterns. The shared variables head, tail, and tasks.array[] are read before writing to head, and no reads need to be atomic with the subsequent write.

### 2.4 FIFO Queue Example

In examining concurrent algorithms for multi-consumer FIFO queues, one notes that either locking or CAS is used in the common path of nontrivial dequeue operations that return a dequeued item. However, as we mentioned already, our result proves that mutually exclusion locking requires each execution of a successful lock acquire to include the AW AR or RAW pattern. All algorithms that avoid the use of locking in dequeue include a CAS operation in the common path of each nontrivial dequeue execution. Fig. 7 shows the simplified code snippet from the dequeue operation of the classic Michael and Scott’s lock-free FIFO queue [37]. Note that every execution that returns an item must execute CAS.

We observe that algorithms for multi-consumer dequeue include directly or indirectly at least one instance of the AW AR or RAW patterns (i.e. use either locking or CAS).

**Restricting Ownership** However, our result suggests that if we want to eliminate RAW and AW AR correctly, we can focus on restricting the processes that can execute an operation. For instance, we can specify that dequeue be executed by a single process, instead of multiple processes. Indeed, when we consider single-consumer FIFO queues, where no more than one process can execute the dequeue operation, we can obtain a correct implementation of dequeue which does not require RAW and AW AR.

Fig. 8 shows a single-consumer dequeue operation, adapted from Lamport’s single-producer single-consumer FIFO queue [30].

Note that the code avoids both RAW and AW AR. The variable head is private to the single consumer and its update is done by a regular write.

Once again, this example demonstrates a case where we used our result to guide the implementation. In particular, by changing the specification of an operation of the abstract data type—namely

---

1. The restriction or the lack of restriction on the number of concurrent producers does not affect the algorithm for the dequeue operation.
Data dequeue() {
  1:  if (tail == head) return EMPTY;
  2:  Data data := Q[head mod m];
  3:  head := head + 1 mod m;
  4:  return data;
}

Figure 8. Single-consumer AWAR-RAW-free dequeue operation adapted from Lamport’s FIFO queue [30].

from multi-consumer to single-consumer—it enabled us to create an implementation of the operation (i.e., dequeue) where we did not need RAW and AWAR.

3. Preliminaries
As our work contains results on both linearizable algorithms and mutual exclusion problems, in this section, we include the formalization that is common to both problems. Preliminaries that are specific to one of the results are included in the corresponding section.

3.1 Implementation
We now define all terms and properties that pertain to implementation, including transitions, executions and equivalence between states.

3.1.1 Language
For our implementations, we assume a standard sequential imperative programming language with the usual syntax and semantics, equipped with only basic constructs for concurrency: an atomic section and process creation. We assume the language does not allow nested atomic sections or process creation inside atomic sections. This language is simple but powerful enough to express all known practical concurrent algorithms today. Further, if necessary, with the atomic construct, one can implement all other atomic operations such as fetch-and-store, locks, universal constructs such as compare-and-swap, etc.

We use $\text{VarIds}_L$ to denote the set of process-local variable identifiers, $\text{VarIds}_G$ to denote the set of global identifiers (i.e. memory which can be accessed by each process including global variables and heap) and $\text{Lab}$ the set of program labels.

3.1.2 Semantics
A program state $\sigma$ is a tuple $(pc, locals, globals) \in \Sigma$, where:

- $\Sigma = PC \times Locals \times Globals$
- $PC = \text{ProcessIDs} \rightarrow \text{Lab}$
- $Locals = \text{ProcessIDs} \times \text{VarIds}_L \rightarrow \text{Val}$
- $Globals = \text{VarIds}_G \rightarrow \text{Val}$

A state $\sigma$ tracks the program counter for each process ($pc$), a mapping from process local variables to values ($locals$), and a mapping from global identifiers to values ($globals$).

We assume that program statements are labeled with unique labels and use $\text{stmt}(l)$ as the statement at program label $l$. We denote the set of initial states as $\text{Init} \subseteq \Sigma$.

Transition Relation
The behavior of a program is determined by its transition relation $TR \subseteq \Sigma \times ProcessIDs \times \text{Strat} \times \Sigma$, describing a set of transitions. For a transition $\pi_{tr} \in TR$, we denote its source state by $\text{src}(\pi_{tr})$, its executing process by $\text{proc}(\pi_{tr})$, its destination state by $\text{dst}(\pi_{tr})$, its executing statement by $\text{stmt}(\pi_{tr})$ and the unique program label where the statement resides as $\text{lab}(\pi_{tr})$. For atomic sections, we assume strong atomicity semantics: if a process is inside an atomic section in a given state, then no other process can take transitions from that state. To avoid formal clutter, our treatment of atomic sections is semi-formal (it can be modeled easily by including a field in the state, denoting the process currently inside an atomic section, but that will introduce more formal clutter). A program transition represents the intuitive fact that starting from a state $\text{src}(\pi_{tr})$, where no other process $p_j \neq \text{proc}(\pi_{tr})$ is inside an atomic section, process $\text{proc}(\pi_{tr})$ can execute the statement $\text{stmt}(\pi_{tr})$ and end up in a state $\text{dst}(\pi_{tr})$.

For our implementations, we assume a standard sequential imperative semantics where a process cannot modify the local state of other processes.

We now define all terms and properties that pertain to implementation, including transitions, executions and equivalence between states.

### Executions
An execution $\pi$ is a (possibly infinite) sequence of transitions $\pi_0, \pi_1, \ldots$, such that for every $i \geq 0$, $\pi_i \in TR$ and $\forall j \geq 1. \text{dst}(\pi_{j-1}) = \text{src}(\pi_j)$. An execution $\pi$ is initial if $\text{src}(\pi_0) \in \text{Init}$. We use first($\pi$) as a shortcut for $\text{src}(\pi_0)$, i.e. the first state in the execution $\pi$, and, last($\pi$) to denote the last state in the execution $\pi$, i.e. last($\pi$) = dst(first($\pi$)).

When an execution $\pi$ is performed in an execution $\pi$ then $t \in \pi$ returns true, otherwise it returns false. We use $\pi(i, j)$ to denote the sequence of all transitions in $\pi$ occurring between positions $i$ and $j$ (including the transitions at $i$ and $j$).

### Definitions
3.1.1 (Atomic Execution). We say that an execution $\pi$ is executed atomically by a process $p$ when $\pi$ is p-solo and either $|\pi| = 1$ or $\forall i. 1 \leq i < |\pi|$, $p$ is inside an atomic section in each state $\text{src}(\pi_i)$.

Note that every atomic execution by process $p$ is p-solo, but not every p-solo execution is atomic. That is, even if an execution is solo, for a given process, it could be the case that there is another
execution by a different process which contains one of the states of this execution (without the last one).

Given an execution $\pi$ and a transition $\pi_k \in \pi$, the maximal atomic subsequence of transition $\pi_k$ in $\pi$ is the (unique) subsequence $\pi_i \ldots \pi_j$, such that $\pi_i \ldots \pi_j$ is an atomic execution of process $\text{proc}(\pi_k)$, where $i \leq k \leq j$, and $\text{proc}(\pi_k)$ is outside an atomic section in states $\text{src}(\pi_i)$ and $\text{dst}(\pi_j)$.

Next, we define the notion of equivalent states with respect to a set of processes $P$. Intuitively, two states are equivalent with respect to a set of processes $P$, if the global states of the two states are identical and the local states of each process in both states is identical. More formally:

**Definition 3.2 (State Equivalence).** Given a set of processes $P \subseteq \text{ProcessIDs}$, two states $\sigma$ and $\sigma'$ are said to be equivalent with respect to $P$, denoted as $\sigma \equiv_P \sigma'$ if:

- $\forall p \in P$, $\text{pc}(p) = \text{pc}'(p)$
- $\forall (p, \text{var}) \in \text{dom(locals}_{\sigma})$, $\text{locals}_{\sigma}(p, \text{var}) = \text{locals}_{\sigma'}(p, \text{var})$
- enabled$(p) = \text{enabled'}(p)$

For convenience, when $P$ is a singleton $P = \{p\}$, we write $\sigma \equiv_p \sigma'$.

Next, we define read-after-write executions. Intuitively, these are executions where a process writes to global memory location $A$ and then, sometimes later, reads a global memory location that is different from $A$.

**Definition 3.3 (Read After Write Execution).** We say that a process $p$ performs a read-after-write in execution $\pi$, if $\exists i, j \leq |\pi|$ such that:

- $\pi_i \pi_j$ performs a global write by process $p$
- $\pi_j \pi_j$ performs a global read by process $p$
- $\text{mloc}(\pi_i) \neq \text{mloc}(\pi_j)$ (the memory locations are different).

We introduce the predicate $\text{RAWR}(\pi, p)$ which evaluates to true if $p$ performs a read-after-write in $\pi$ and to false otherwise. Note that in this definition there is no restriction on how the accesses are performed, atomically or not, it talks only about ordering of operations.

Next, we define atomic write-after-read executions. Intuitively, these are executions where a process first reads from a global memory location and then, sometimes later, writes to a global memory location and these two transitions occur atomically, that is, in-between these two transitions, no other process can perform any transitions. Note that unlike read-write executions, here, the global read and write does not access different memory locations.

**Definition 3.4 (Atomic Write After Read Execution).** We say that a process $p$ performs an atomic write-after-read in execution $\pi$, if $\exists i, j \leq |\pi|$ such that:

- $\pi_i \pi_j$ performs a global write by process $p$
- $\pi_j \pi_j$ performs a global read by process $p$
- $\text{mloc}(\pi_i) \neq \text{mloc}(\pi_j)$ is an atomic execution of process $p$

We introduce the predicate $\text{AWAR}(\pi, p)$ which evaluates to true if process $p$ performs an atomic write-after-read in $\pi$ and to false otherwise.

3.2 Specification

3.2.1 Histories

A history $H$ is defined as a finite sequence of actions, i.e. $H = \psi_1 \psi_2 \ldots \psi_n$, where an action is an invocation or a response of an operation by a given process, that is:

$$\psi = (p, \text{invoke op arg}) \mid (p, \text{response op ret})$$

where $p \in \text{ProcessIDs}$ is a process identifier, $op \in \text{OpId}$ is an operation identifier, $\text{arg}$ is the argument to the operation and $\text{ret}$ is the return value of the operation. For an action $a$, we use $\text{proc}(a)$ to denote the process, $\text{kind}(a)$ to denote the kind of the action (invoke or response), $\text{op}(a)$ to denote the name of the operation, $\text{arg}(a)$ to denote the arguments and $\text{ret}(a)$ to denote the return value. We use $H_i$ to denote the action at position $i$ in the history, where $0 \leq i < |H|$. For a process $p$, $H |p|$ is used to denote the sub-history of $H$ (which is a sub-sequence of $H$) that consists only of the actions of process $p$.

An invocation $(p, \text{invoke m arg})$ is said to be pending in a history $H$ if it has no matching response, that is, $\exists i, 0 \leq i < |H|$ such that $\text{proc}(H_i) = p$, $\text{kind}(H_i) = \text{invoke}$, $\text{op}(H_i) = m$ and $\forall j, i < j < |H|$, $\text{proc}(H_j) \neq p$ or $\text{kind}(H_j) \neq \text{response}$. A history $H$ is said to be complete if it has no pending calls. We use $\text{complete}(H)$ to denote the set of histories resulting after extending $H$ with matching responses to a subset of invocations that are pending in $H$ and then removing the remaining pending invocations. (A history $H$ is sequential if $H$ is empty $(H = \epsilon)$ or $H$ starts with an invocation action, i.e. $\text{kind}(H_0) = \text{invoke}$ and invocations and responses alternate. That is, $\forall i, 0 \leq i < |H| - 1, \text{kind}(H_i) \neq \text{kind}(H_{i+1})$ and each response is matched by an invocation that occurs immediately before it in $H$, i.e. $\forall i, 1 \leq i < |H|$, if $\text{kind}(H_i) = \text{response}$ then $\text{kind}(H_{i+1}) = \text{invoke}$ and $\text{proc}(H_i) = \text{proc}(H_{i+1})$. A complete sequential history $H$ where $|H| = 2$ is said to be a complete invocation of an operation $o$ if $\text{op}(H_0) = o$ and $\text{op}(H_1) = o$. In the case where $H$ is a complete sequential invocation, we use $\text{inv}(H)$ to denote the invoke action in $H$ and $\text{resp}(H)$ to denote the response action in $H$. A history $H$ is well-formed if for each process $p \in \text{ProcessIDs}$, $H |p|$ is sequential. In this work, we consider only well-formed histories.

3.2.2 Histories and Executions

For a transition $t$, if $\text{stmt}(t) = (\text{kind op arg})$ where $\text{kind} \in \{\text{invoke, response}\}$, then $t \mid (\text{inv}, \text{response})$ (abbreviated as $t \mid (\text{inv}, \text{response})$) denotes the action $(p, \text{kind op arg})$ where $p = \text{proc}(t)$. Otherwise, $t \mid (\text{inv})$, returns $\epsilon$ (the empty sequence). Accordingly, we can define a history $H(\pi)$ of an execution $\pi$ in the obvious way: $\pi \mid (\text{inv})$ denotes the sequence of invocations and responses occurring in $\pi$.

For a program $\text{Prog}$, we define its corresponding set of histories as $[\text{Prog}]_H = \{\pi \mid (\pi \in [\text{Prog}]_H)\}$. We use $[\text{Prog}]_H$ to denote the sequential histories in $[\text{Prog}]_H$. When clear from the context, we also sometimes use $\pi \mid (\text{inv})$ to denote projection onto invocations and responses of a specific operation $op$.

Given an execution $\pi$, we say that a transition $t \in \pi$ has a matching invocation in $\pi$, if there exists a transition $t_{\text{prev}} \in \pi$ such that $t_{\text{prev}} \mid (\text{inv}) \neq \epsilon$, $\text{kind}(t_{\text{prev}} \mid (\text{inv})) = \text{proc}(t_{\text{prev}})$, $t_{\text{prev}}$ precedes $t$ in $\pi$ and there does not exist a matching response action to $t_{\text{prev}} \mid (\text{inv})$ that is performed in-between $t_{\text{prev}}$ and $t$ in $\pi$. Note that $t_{\text{prev}}$ may be the same as $t$.

**Definition 3.5 (Well-formed Execution).** We say that an execution $\pi$ is well-formed if its history $\pi \mid (\text{inv})$ is well-formed, every transition in $t \in \pi$ has a matching invocation and for any transition $t \in \pi$ where $t \mid (\text{inv}) \neq \epsilon$, $\text{proc}(t)$ is outside of an atomic section in $\text{src}(t)$.

We say that $\pi$ is a complete sequential execution of an operation $op$ by process $p$, if $\pi$ is a well-formed execution, all transition in the history $\pi \mid (\text{inv})$ by process $p$.

A program is well-formed if it generates only well-formed executions. In this paper, we only consider well-formed programs.
4. Synchronization in Mutual Exclusion

In this section, we consider implementations that provide mutually exclusive access to a critical section among a set of processes. We show that every deadlock-free mutual exclusion implementation incurs RAW and AWAR patterns in certain executions.

A mutual exclusion implementation exports four kinds of external actions: an invocation \( \text{try} \) and a matching response \( \text{enter} \), and an invocation \( \text{exit} \) and a matching response \( \text{rem} \). We strengthen the notion of well-formedness and say that a history \( H \) is well-formed if, for every process \( p \), \( H \upharpoonright_p \) is a prefix of the cyclic sequence \( \text{try}, \text{enter}, \text{exit}, \text{rem}, \ldots \). Respectively an execution \( \pi \) is well-formed if it generates a well-formed history. We only consider well-formed executions here.

At the end of a finite execution \( \pi \), a process \( p \) is said to be in its \( \text{trying section} \) if its last external action in \( \pi \) is \( \text{try} \), in its \( \text{critical section} \) if its last external action is \( \text{enter} \), in its \( \text{exit section} \) if its last external action is \( \text{exit} \), and in its \( \text{remainder section} \) otherwise. Therefore, a process in a mutual exclusion algorithm is in its remainder section initially, and then cyclically goes through trying, critical, exit, and remainder sections. A process is called \( \text{active} \) if it is in its trying or exit section, and only active processes are allowed to take steps in an execution.

For the purpose of our lower bound, we assume the following weak formulation of the mutual exclusion problem \([12, 31]\). In addition to the classical mutual exclusion requirement, we only require that the implementation is deadlock-free, i.e., if a number of active processes concurrently compete for the critical section, at least one of them succeeds.

**Definition 4.1** (Mutual Exclusion). A deadlock-free mutual exclusion implementation guarantees:

- (Safety) There is no execution \( \pi \) such that two processes are in their critical sections at the end of \( \pi \).
- (Liveness) In every execution in which every active process takes sufficiently many steps:
  - If at least one process in its \( \text{trying section} \) and no process is in its \( \text{critical section} \), then at some point later some process enters its \( \text{critical section} \).
  - If at least one process in its \( \text{exit section} \), then at some point later some process enters its remainder section.

We say that \( \pi \) is a complete \( \text{trying section} \) of a process \( p \) if \( \pi \upharpoonright_p = \text{try} \cdot \text{enter} \).

**Theorem 4.2** (RAW and AWAR in Mutual Exclusion). Let \( \text{Prog} \) be a deadlock-free mutual exclusion implementation for two or more processes. Then for every complete \( p \)-solo \( \text{trying section} \) \( \pi_p \):

- \( \text{RAW}(\pi_p, p) = \text{true} \), or
- \( \text{AWAR}(\pi_p, p) = \text{true} \).

**Proof.** Let \( \pi_{base} \cdot \pi_q \) be an execution in \([\text{Prog}]\) such that \( \pi_q \) is a complete \( p \)-solo \( \text{trying section} \). By contradiction, assume that \( \pi_q \) does not contain a global write. Consider an execution \( \pi_{base} \cdot \pi_q \) such that \( \pi_q \) is \( p \)-free and it contains complete \( \text{trying section} \) of a process \( q \neq p \). Since \( \text{Prog} \) is deadlock-free, such an execution exists. Assume that the last action \( q \) in \( \pi_q \) is \( \text{enter} \) (it is in its \( \text{critical section} \) right after \( \pi_q \)). But \( \text{last}(\pi_{base}) \upharpoonright_q \subset \text{last}(\pi_{base} \cdot \pi_q) \) and, thus, there exists an execution \( \pi_{base} \cdot \pi_q \cdot \pi_r \) at the end of which both \( p \) and \( q \) are in their \( \text{critical sections} \) — a contradiction.

Thus, \( \pi_q \) contains a global write transition, an let \( t_w \) is the first global write transition in \( \pi_q \). Let \( \pi_q = \pi_{\pi_q} \cdot \pi_w \cdot \pi_1 \), where \( \pi_{\pi_q} \) is the maximal atomic subsequence of \( \pi_q \), that contains \( t_w \) (if \( t_w \) does not belong to an atomic construct, then \( \pi_{\pi_q} = t_w \)). We prove by contradiction and assume that \( \text{RAW}(\pi_q, p) = \text{false} \) and \( \text{AWAR}(\pi_q, p) = \text{false} \). Immediately, we observe that since \( \text{AWAR}(\pi_q, p) = \text{false} \) and \( t_w \) is the first write transition in \( \pi_q \), \( t_w \) is the first global transition in \( \pi_{\pi_q} \).

Further, since \( \pi_{\pi_q} \) contains no global writes, \( \forall q \in \text{ProcessIds} \) where \( q \neq p \), \( \text{first}(\pi_{\pi_q}) \not\leq \text{last}(\pi_{\pi_q}) \). Since \( \text{Prog} \) is deadlock-free, there exists an execution \( \pi_{base} \cdot \pi_0 \cdot \pi_q \) such that \( \pi_q \) is \( p \)-free and it contains a complete \( \text{trying section} \) of some process \( q \).

The assumption \( \text{RAW}(\pi_q, p) = \text{false} \) implies that no global read operation of \( p \) in \( \pi_{\pi_q} \cdot \pi_1 \) accesses a variable other than \( \text{valloc}(t_w) \). Note that the first action of \( \pi_{\pi_q} \) overwrites the only location that is read by \( p \) in \( \pi_{\pi_q} \cdot \pi_1 \). Thus, there exists an execution \( \pi_0 = \pi_{\pi_q} \cdot \pi_{\pi_w} \cdot \pi_0 \) in \([\text{Prog}]\) such that \( p \) does not distinguish \( \text{last}(\pi_1) \) and \( \text{last}(\pi_0) \). Therefore, the last action of \( p \) in \( \pi_0 \) is also \( \text{enter} \), while \( q \) is in its \( \text{critical section} \) — a contradiction.

Thus, either \( \text{RAW}(\pi_q, p) = \text{true} \) or \( \text{AWAR}(\pi_q, p) = \text{true} \).

5. Synchronization in Linearizable Algorithms

In this section, we state and prove a new result that affects the design of practical concurrent algorithms. Informally, our result states that strongly non-commutative operations must use certain kinds of synchronization: either by containing some inherent order between reads and writes (i.e. the RAW pattern) or by making parts of the execution atomic (i.e. the AWAR pattern). The impact of this result in practice is that in order to enforce these patterns, expensive synchronization is required. For instance, to enforce the RAW pattern when the program is running on a weak memory model, one must use an expensive store-load fence.

Before we state and prove our result, we first define linearizability, deterministic sequential specifications and strongly non-commutative operations.

**5.1 Linearizability**

Next, following \([22, 24]\) we define linearizable histories. A history \( H \) induces an irreflexive partial order \( \preceq \) on actions in the history: \( a \prec b \) if \( \text{kind}(a) = \text{response} \) and \( \text{kind}(b) = \text{invoke} \) and \( 0 \leq i < j < |H| \) such that \( H_i = a \) and \( H_j = b \). That is, response action \( a \) precedes invocation action \( b \) in \( H \). A history \( H \) is said to be linearizable with respect to a sequential history \( S \) if there exists a history \( H' \subseteq \text{complete}(H) \) such that:

1. \( \forall p \in \text{ProcessIds}, H' \upharpoonright_p = S \upharpoonright_p \)
2. \( \prec H \subseteq \prec S \)

We can naturally extend this definition to a set of histories. Let \( \text{Spec} \) be a sequential specification, a prefix-closed set of complete sequential histories (that is, if \( s \) is a sequential history in \( \text{Spec} \), then any prefix of \( s \) is also in \( \text{Spec} \)). Then, given a set of histories \( \text{Impl} \), we say that \( \text{Impl} \) is linearizable with respect to \( \text{Spec} \) if for any history \( H \in \text{Impl} \) there exists a history \( S \in \text{Spec} \) such that \( H \) is linearizable with respect to \( S \).

We say that a program \( \text{Prog} \) is linearizable with respect to a deterministic sequential specification \( \text{Spec} \) when \([\text{Prog}]_H \) is linearizable with respect to \( \text{Spec} \).

**5.2 Deterministic Sequential Specifications**

In this paper, similarly to \([9]\), we define deterministic sequential specifications. Given two sequential histories \( s_1 \) and \( s_2 \), let \( \text{maxprefix}(s_1, s_2) \) denote the longest common prefix of the two histories \( s_1 \) and \( s_2 \).

**Definition 5.1** (Deterministic Sequential Specifications). A sequential specification \( \text{Spec} \) is deterministic, if for all \( S_1, S_2 \subseteq \text{Spec} \), \( S_1 \neq S_2 \) and \( S = \text{maxprefix}(S_1, S_2) \), we have \( S = \varepsilon \) or \( \text{kind}(S_{[S-1]} \) \neq \text{invoke} \).

That is, a specification is deterministic, if we cannot find two different histories whose longest common prefix ends with an in-
vocation. If we can find such a prefix, then that would mean that there was a point in the execution of the two histories $S_1$ and $S_2$ up to which they behaved identically, but after they both performed the same invocation, they produced different results (or one had no continuation).

5.3 Strongly Non-Commutative Operation

Next, we define the notion of a strongly non-commutative operation. This notion strengthens the traditional notion of non-commutative operations and weakens the notion of a non-idempotent operation.

**Definition 5.2 (Strongly Non-Commutative Operation).** Let $Spec$ be a sequential specification of complete sequential histories. We say that an operation $op_1$ is strongly non-commutative in $Spec$ if there exists an operation $op_2$ (possibly equal to $op_1$) such that:

1. $base$ is a complete sequential history in $Spec$.
2. $s_1$ and $s_3$ are complete invocations of $op_1$.
3. $s_2$ and $s_3$ are complete invocations of $op_2$.
4. $arg(inv(s_1)) = arg(inv(s_3))$.
5. $arg(inv(s_2)) = arg(inv(s_3))$.
6. $base \cdot s_1 \in Spec$.
7. $base \cdot s_2 \in Spec$.
8. $base \cdot s_2 \cdot s_3 \in Spec$.
9. $base \cdot s_2 \cdot s_3 \in Spec$.
10. $ret(resp(s_1)) \neq ret(resp(s_3))$.
11. $ret(resp(s_2)) \neq ret(resp(s_3))$.

In other words, the operation $op_1$ is strongly non-commutative if there is another operation $op_2$ and a history $base$ in $Spec$ such that we can distinguish whether $op_2$ is applied right after $base$ or right after $op_2$ (which is applied after $base$). Similarly we can distinguish whether $op_2$ is applied right after $base$ or right after $op_1$ (which is applied after $base$). Note that $op_2$ may be the same operation as $op_1$.

We define $op_1$ as a strongly non-commutative unordered operation if in addition to the requirements in Definition 5.2, $proc(inv(s_1)) = proc(inv(s_4))$, $proc(inv(s_2)) = proc(inv(s_3))$, and $proc(inv(s_1)) \neq proc(inv(s_2))$.

**Non-Idempotent Operation vs. Strong Non-Commutative Operation**

Note that if we select $op_2$ to be the same operation as $op_1$ then we reach a special case which amounts to non-idempotent operations. That is, given $base$, if we apply $op_1$ twice in a row, the second invocation will return a different result than the first. Consider again the Set specification in Fig. 1. The operation $add$ is non-idempotent. As discussed in the example in Section 2, we can start with $S = \emptyset$ and say $base = e$. Then, if we perform two $add(5)$ in a row, each $add(5)$ will return a different result. Non-idempotent operations are strongly non-commutative, but not vice versa.

**Classic Non-Commutativity vs. Strong Non-Commutativity**

In the classic notion of non-commutativity (e.g. [45]), it is enough for one of the operations to not commute with the other, while here, it is required that both operations do not commute from the *same* prefix history. That is, if two operations do not commute, it does not mean that either of them is a strongly non-commutative operation. However, if an operation is strongly non-commutative, there exists another operation with which it does not commute (by definition). Consider again the Set specification in Fig. 1. Although $add$ and $contains$ do not commute, $contains$ is not a strongly non-commutative operation. That is, $add$ influences the result of $contains$, but $contains$ does not influence the result of $add$.

**Definition 5.3 (Strongly Non-Commutative Sequential Specification).** We say that a sequential specification $Spec$ is strongly non-commutative if there exists an operation which is strongly non-commutative in $Spec$.

5.4 RAW and AWAR Cannot be Eliminated

We begin with stating that if we perform a complete sequential operation from two equivalent states, then in both cases the operation will return the same results (i.e. the same history).

**Lemma 5.4 (Equivalent Histories).** Let $\pi$ be a complete sequential execution of $op$ by process $p$ and let $\sigma$ be a state such that $\sigma \not\models first(\pi)$. Then, there exists a complete sequential execution $qi$ of $op$, where $src(qi) = \sigma$ and $\pi_{ir} = qi_{ir}$.

This validity of this lemma follows directly from the fact that the transition relation is deterministic, that enabled transitions are taken, and that a process cannot access the local variables of another process.

In this work we focus on programs where the specification $Spec$ can be determined by the sequential executions of the program.

**Definition 5.5.** Given a sequential specification $Spec$ and a program $Prog$, we say that the program determines the specification when the sequential histories are the same as the specification $Spec$, that is, $[Prog]_{Hist} = Spec$.

With minor exceptions, this property holds for all practical algorithms that we are aware of. From here on, we assume that all sequential specifications $Spec$ are determined.

Next, we prove that if an operation is strongly non-commutative, then there must exist an execution of this operation that will write to global memory. The result is intuitive, as if the operation does not write to global memory, there will be no observable change to the state and hence the operation will not be non-commutative.

**Lemma 5.6 (Global Write).** Let $op_1$ be a strongly non-commutative unordered operation in a deterministic sequential specification $Spec$ and let $Prog$ be a linearizable implementation of $Spec$. Then, there exists a complete sequential execution $\pi_a$ of $op_1$ in $[Prog]$ by process $p \in ProcessIDs$ such that $t_a \in \pi_a$, where $t_a$ performs a global write.

**Proof.** From the premise that $op_1$ is a strongly non-commutative unordered operation, it means that there exists an operation $op_2$ with the properties as described in Definition 5.2. Let $\pi_a, \pi_b$, and $\pi_c$ be the complete sequential executions in $[Prog]$ such that $s_1 = \pi_a \models ir$, $s_2 = \pi_b \models ir$ and $s_3 = \pi_c \models ir$. The executions $\pi_a, \pi_b$, and $\pi_c$ are guaranteed to exist due to Definition 5.5. Let $p = proc(\pi_a)$ and $q = proc(\pi_c)$. As $op_1$ is unordered, we know that $p \neq q$.

Let us assume the execution $\pi_a$ does not contain a transition that performs a global write. Then, $first(\pi_a) \not\models last(\pi_a)$. However, by the premise we know that $last(\pi_a) = first(\pi_c)$, and hence $first(\pi_a) \not\models first(\pi_c)$. Thus, following Lemma 5.4, $\pi_a \models ir, \pi_c \models ir$. Hence, $s_2 = s_3$, which contradicts premise 11 in Definition 5.2. Therefore, there must exist a transition in $\pi_c$ that performs a global write.

Let us assume the execution $\pi_a$ contains a transition that performs a global write. Then, $first(\pi_a) \models last(\pi_a)$. However, by the premise we know that $last(\pi_a) = first(\pi_c)$, and hence $first(\pi_a) \models first(\pi_c)$. Thus, following Lemma 5.4, $\pi_a \models ir, \pi_c \models ir$. Hence, $s_2 = s_3$, which contradicts premise 11 in Definition 5.2. Therefore, there must exist a transition in $\pi_c$ that performs a global write.

Next, we prove that RAW and AWAR cannot be avoided in a wide class of linearizable implementations.

**Theorem 5.7 (RAW or AWAR in Linearizable Implementations).** Let $op_1$ be a strongly non-commutative unordered operation in a deterministic sequential specification $Spec$ and let $Prog$ be a linearizable implementation of $Spec$. Then, there exists a complete sequential execution $\pi_a$ of $op_1$ by process $p \in [Prog]$ such that:

- RAW($\pi_a, p) = true$, or
- AWAR($\pi_a, p) = true$
Proof. From the premise that \( op_1 \) is a strongly non-commutative unordered operation, it means that there exists a history base and operation \( op_2 \) with the properties described in Definition 5.2. Let \( p \Rightarrow base, \alpha, \beta, \pi, a, \text{and} \ pi \) be the complete sequential executions in \( Prog \) such that \( base \Rightarrow \pi_1, \pi_2 \Rightarrow \pi_3, \pi_4 \Rightarrow \pi_5 \), and \( s_1 = \pi_1 \sim \pi_2 \sim \pi_3 \sim \pi_4 \sim \pi_5 \). The executions \( base, \pi_1, \pi_2, \pi_3, \pi_4 \) are guaranteed to exist due to Definition 5.5. Let \( p = proact(\pi_s) \) and \( q = proact(\pi_t) \). As \( op_1 \) is unordered, we know that \( p \neq q \).

Let us proceed by contradiction and assume the conclusion is false, i.e. \( RAW(\pi_s, p) \) is false and \( RAW(\pi_t, p) \) is false. By Lemma 5.6, we know there exists a write transition \( t_w \in \pi_s \) that writes to global memory. Let \( \pi_a = \pi_f \cdot \pi_w \cdot \pi_t \), where \( \pi_w \) is the first global write transition in \( \pi_a \) and \( \pi_t \) is the maximal atomic subsequence of \( t_w \). Since \( RAW(\pi_t, p) \) is false and \( t_w \) is the first global write transition in \( \pi_s \), it follows that there can be no global read transitions in \( \pi_s \) that occur before \( t_w \) (otherwise we would contradict \( RAW(\pi_s, p) \) is false). This means that \( \pi_w \) is the first global (read or write) transition in \( \pi_a \).

From Definition 5.5, we know that in state first(\( \pi_t \)) process \( p \) is outside an atomic section and from the fact that \( \pi_w \) is a maximal atomic subsequence we know that in state last(\( \pi_t \)), \( p \) is also outside an atomic section. As \( \pi_f \) contains only transitions by process \( p \) which are not global writes, it follows that \( first(\pi_f) \not\preceq last(\pi_t) \). From the premise, we know that \( first(\pi_s) = first(\pi_t) \). Therefore, we know that \( first(\pi_s) \not\preceq last(\pi_t) \). From the fact that a complete sequential execution \( \pi_t \) by process \( q \) exists, and from \( first(\pi_s) \not\preceq last(\pi_t) \), we know that process \( q \) can begin a q-solo execution of \( op_2 \) from last(\( \pi_t \)) (transition relation requires \( q \) to execute a transition from last(\( \pi_t \))). Then, as the transition relation is deterministic, inductively, we can build a complete sequential execution \( \pi_b \) by process \( q \), where after any transition \( t \) performed by process \( q \), we know that \( dst(t') \not\subseteq dst(t) \) where \( t' \) is the transition in \( \pi_b \) at the same position as \( t \) resides in \( \pi_t \). That is, \( \pi_b \) mirrors the execution \( \pi_t \) and hence according to Lemma 5.4, \( \pi_b \) is complete. Transitivity, \( \pi_b \) is also complete.

As \( \pi_b \) is a q-solo well-formed complete sequential execution, it follows that \( enabled_{last(\pi_t)} \cdot p \equiv enabled_{last(\pi_b)} \cdot p \), that is, \( enabled_{last(\pi_t)} \cdot p \equiv enabled_{first(\pi_t)} \cdot p \). Similarly, we can build the \( p \)-solo execution \( \pi_u \cdot \pi_i \) such that \( \pi_u \cdot \pi_i \) contains the same sequence of statements as \( \pi_u \cdot \pi_i \). At any state in \( \pi_u \cdot \pi_i \), \( p \) is enabled to perform a transition and since we assumed that \( RAW(\pi_u, p) \) and \( RAW(\pi_u, p) \) are false, it implies that \( \pi_u \cdot \pi_i \) does not contain a read transition which reads a global location other than \( mlreq(t_w) \). As \( p \) overwrites its first global write transition in \( \pi_u \) at the only location that it reads afterwards in \( \pi_u \), \( \pi_i \) follows that \( \pi_u \cdot \pi_i \) and \( \pi_u \cdot \pi_i \) are indistinguishable to \( p \) and, thus, the result of \( op_1 \) returned in \( \pi_u \cdot \pi_i \) is the same as the result of \( op_1 \) in \( \pi_a \). That is, in the execution \( conc_{base} = \pi_f \cdot \pi_b \cdot \pi_u \cdot \pi_i \) which we just showed exists in \( Prog \), \( conc_{base} \equiv s_1 \).

Given that the implementation is linearizable with respect to a deterministic sequential specification, we know that the only possible valid linearizations of the two overlapping operations in \( base \cdot conc \) are: a) \( base \cdot conc \cup op_1 \cdot \pi_1 \cdot \pi_2 \), or b) \( base \cdot \pi_1 \cdot \pi_2 \cdot \pi_3 \cdot \pi_4 \cdot \pi_5 \). Let us proceed by considering each of these two cases:

Case a) when the linearization is \( base \cdot \pi_1 \cdot \pi_2 \cdot \pi_3 \cdot \pi_4 \cdot \pi_5 \), because the specification is deterministic, it follows that \( conc_{base} \equiv s_1 \) and \( \pi_1 \cdot \pi_2 \cdot \pi_3 \cdot \pi_4 \cdot \pi_5 \) is unique. However, we know that \( \pi_1 \cdot \pi_2 \cdot \pi_3 \cdot \pi_4 \cdot \pi_5 \) is the same as the result of \( op_1 \) in \( \pi_a \).

In case b) when the linearization is base \cdot \pi_1 \cdot \pi_2 \cdot \pi_3 \cdot \pi_4 \cdot \pi_5 \), because the specification is deterministic, it follows that \( \pi_1 \cdot \pi_2 \cdot \pi_3 \cdot \pi_4 \cdot \pi_5 \) is unique. However, we know that \( \pi_1 \cdot \pi_2 \cdot \pi_3 \cdot \pi_4 \cdot \pi_5 \) is unique, which contradicts condition 11 in Definition 5.2.

In case b) when the linearization is base \cdot \pi_1 \cdot \pi_2 \cdot \pi_3 \cdot \pi_4 \cdot \pi_5 \), because the specification is deterministic, it follows that \( \pi_1 \cdot \pi_2 \cdot \pi_3 \cdot \pi_4 \cdot \pi_5 \) is unique, which contradicts condition 10 in Definition 5.2.

Therefore, \( RAW(\pi_a, p) \equiv true \) or \( RAW(\pi_s, p) \equiv true \).
by process \(i\) on \(Q_j\), and let \(s_2\) and \(s_3\) be complete invocations of take by process \(i\). From Definition 6.3, \(\{base \cdot s_1, base \cdot s_2, base \cdot s_3, base \cdot s_4\} \subseteq Spec_{aw}, ret(s_1) = ret(s_2) = v,\) and \(ret(s_3) = ret(s_4) = empty.\) The claim now follows from Definition 5.2. \(\square\)

It follows from Lemma 6.4 and Definition 5.3 that \(Spec_{aw}\) is strongly non-commutative. It is easily shown that queues, hash-tables and sets are strongly non-commutative. The proofs are essentially identical to the proofs of Lemmas 6.2 and 6.4 and are therefore omitted.

6.3 Compare-and-Swap (CAS)

We now prove that CAS is strongly non-commutative.

**Definition 6.5 (Compare-and-swap Sequential Specification).** A compare-and-swap object \(C\) supports a single operation called CAS and stores a scalar value over some domain \(V\). The operation \(CAS(exp,new)\), for \(exp, new \in V\), induces the following state transition of the compare-and-swap object. If \(C\)'s value is \(exp\), \(C\)'s value is changed to \(new\) and the operation returns \(true\); otherwise, \(C\)'s value remains unchanged and the operation returns \(false\). The sequential specification of a compare-and-swap object is defined as above.

**Lemma 6.6 (CAS Strongly Non-Commutative).** The CAS operation is strongly non-commutative.

**Proof.** Let \(base \in Spec_C\) be a complete history after which \(C\)'s value is \(v\), let \(i, j\) be two processes, let \(s_1\) and \(s_3\) be complete invocations of \(CAS(v,v')\) by process \(i\), for some \(v \neq v' \in V\), and let \(s_2\) and \(s_4\) be complete invocations of \(CAS(v,v')\) by process \(j\). From Definition 6.5, \(\{base \cdot s_1, base \cdot s_2, base \cdot s_3, base \cdot s_4\} \subseteq Spec_C, ret(s_1) = ret(s_2) = true,\) and \(ret(s_3) = ret(s_4) = false.\) The claim now follows from Definition 5.2. \(\square\)

It follows from Lemma 6.6 and Definition 5.3 that \(Spec_{caw}\) is strongly non-commutative. It also follows from Lemma 6.6 that any software implementation of CAS is required to use either AWAR or RAW. Proving a similar result for all non-trivial read-modify-write operations (such as fetch-and-add, swap, test-and-set and load-link/store-conditional) is equally straightforward.

7. Related Work

Numerous papers present implementations of concurrent data structures, several of these are cited in Section 2. We refer the reader to Herlihy and Shavit’s book [23] for many other examples.

Contemporary architectures often execute instructions issued by a single process in an out-of-order manner, and provide fence or barrier instructions to order the execution (cf. [1, 33]). There is a plethora of fence and barrier instructions (see [35]). For example, DEC Alpha provides two different fence instructions, a memory barrier (MB) and a write memory barrier (WMB). PowerPC provides a lightweight (lwsync) and a heavyweight (sync) memory ordering fence instructions, where sync is full fence, while lwsync guarantees only RAW, WAR, and WAW ordering but not RAW. SPARC V9 RMO provides several flavors of fence instructions, through a MEMBAR instruction that can be customized (via four-bit encoding) to order a combination of previous read and write operations with respect to future read and write operations. Pentium 4 supports load fence (lfence), store fence (sfence) and memory fence (mfence) instructions. The mfence instruction can be used for implementing the RAW order.

Recently, there has been a renewed interest in formalizing memory models (cf. [40, 43, 44]), model checking [8], and verifying [42] programs that run on these models. Our result is complementary to this direction: it states that we may need to enforce certain order, i.e. RAW, regardless of what weak memory model is used. Further, our result can be used in tandem with program testing and verification: if RAW and AWAR is completely missing from a program that claims to satisfy certain specifications, then that program is certainly incorrect and there is no need to attempt to test it or verify it.

Kawash’s PhD thesis [29] (also in papers [25, 26]) investigates the ability of weak consistency models to solve mutual exclusion, with only read and write operations. This work shows that many weak models (Coherence, Causal consistency, P-RAM, Weak Ordering, SPARC consistency and Java Consistency) cannot solve mutual exclusion. Processor consistency [18] can solve mutual exclusion, but it requires multi-write registers; for two processes, solving mutual exclusion requires at least three variables, one of which is multi-writer. In contrast, we show that particular orders of operations or certain atomicity constraints must be enforced, regardless of the memory model; moreover, our results apply beyond mutual exclusion and hold for a large class of important objects.

Boehm [7] studies when memory operations can be reordered with respect to PThread-style locks, and shows that it is not safe to move memory operations into a locked region by delaying them past a lock call. On the other hand, memory operations can be moved into such a region by advancing them to be before an unlock call. Boehm also provides detailed evaluation of the cost of using fences. However, Boehm’s paper does not address the central subject of our paper, namely, the necessity that certain ordering patterns (RAW or AWAR) must be present inside the lock operations themselves.

Our proof technique employs the covering technique, originally used by Burns and Lynch [10] to prove a lower bound on the number of registers needed for solving mutual exclusion. This technique has many applications, both with read/write operations [4, 5, 13, 14, 28, 39], and with non-trivial atomic operations, such as compare&swap [15, 16]. Some steps of our proofs can be seen as a formalization of the arguments Lamport uses to derive a fast mutual exclusion algorithm [32].

8. Conclusion and Future Work

We presented an important result which states that it is impossible to build a wide range of concurrent algorithms that are both correct and do not use expensive synchronization. Our results have powerful implications for concurrent algorithm construction, hardware design, testing and verification. We focused on two common synchronization idioms: read-after-write (RAW) and atomic write after read (AWAR). We proved that mutual exclusion algorithms must contain either RAW or AWAR patterns in their entry sections. Similarly, we proved that linearizable implementations of strongly non-commutative operations must use RAW or AWAR. Finally, we proved that our result applies to many classic specifications such as stacks, sets, hash tables, queues, work-stealing structures and compare-and-swap operations. We believe that our result is practically useful as a guideline for algorithm writers and further, suggests targeted improvements in hardware design: in particular, optimizing the cost of store-load fences and compare-and-swap operations.

An interesting direction for future work is strengthening our result by considering not only a read after a write (or a write after a read), but also longer sequences of reads and writes. Another interesting direction is formulating a useful class of operations for which we do not need to use RAW and AWAR.
References


[34] Paul E. McKenney. Memory barriers: a hardware view for software hackers. Linux Technology Center, IBM Beaverton, June 2010.


