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
TU Berlin/Deutsche Telekom Labs
pkuznets@acm.org

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

Martin Vechev
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 try to remove unnecessary and costly synchronization. However, not only is this manual trial-and-error process ad-hoc, time consuming 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 concurrent implementations of classic and ubiquitous specifications such as sets, queues, stacks, mutual exclusion and read-modify-write operations, that completely eliminate the use of expensive synchronization.

We prove that one cannot avoid the use of either: i) read-after-write (RAW), where a write to shared variable A is followed by a read to a different shared variable B without a write to B in between, or ii) atomic write-after-read (AWAR), where an atomic operation reads and then writes to shared locations. Unfortunately, enforcing RAW or AWAR is expensive on all current mainstream processors. To enforce RAW, memory ordering–also called fence or barrier–instructions must be used. To enforce AWAR, atomic instructions such as compare-and-swap are required. However, these instructions are typically substantially slower than regular instructions.

Although algorithm designers frequently struggle to avoid RAW and AWAR, their attempts are often futile. Our result characterizes the cases where avoiding RAW and AWAR is impossible. On the flip side, our result can be used to guide designers towards new algorithms where RAW and AWAR can be eliminated.

Categories and Subject Descriptors D.1.3 [*Concurrent Programming*]; E.1 [*Data*]: Data Structures

General Terms Algorithms, Theory

Keywords Concurrency, Algorithms, Lower Bounds, Memory Fences, Memory Barriers

POPL'11, January 26–28, 2011, Austin, Texas, USA.

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 with 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 process writing to some shared variable A, followed by the same process reading a different shared variable B, without that process writing to B in between. The AWAR pattern consists of a process reading some shared variable followed by the process writing to a shared variable (the write could be to the same shared variable as the read), where the entire read-write sequence is atomic. Examples of the AWAR pattern include read-modify-write operations such as a Compare-and-Swap [26] (CAS).

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.[1] Guaranteeing the atomicity of AWAR requires the use of atomic instructions. Typically, fence and atomic instructions are substantially slower than regular instructions, even under the most favorable caching conditions.

Due to these high overheads, designers of concurrent algorithms aim to avoid both RAW and AWAR patterns. However, such attempts are often unsuccessful: in many cases, even after multiple attempts, it turns out impossible to avoid these patterns while ensuring correctness of the algorithm.

This raises an interesting and important practical question:

Can we discover and formalize the conditions under which avoiding RAW and AWAR, while ensuring correctness, is futile?

In this paper, we answer this question formally. We show that implementations of a wide class of concurrent algorithms must involve RAW or AWAR. In particular, we focus on two widely used

[1] 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).

specifications: linearizable objects [23] and mutual exclusion [11]. Our results are applicable to any algorithm claiming to satisfy these specifications.

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

- We prove that it is impossible to build a linearizable implementation of a strongly non-commutative method that satisfies a deterministic sequential specification, in a way that sequential executions of the method are free of RAW and AWAR (Section 5).
- We prove that common methods on ubiquitous and fundamental abstract data types–such as sets, queues, work-stealing queues, stacks, and read-modify-write objects–are strongly non-commutative and are subject to our results (Section 6).
- We prove that it is impossible to build an algorithm that satisfies mutual exclusion, is deadlock-free and avoids both RAW and AWAR (Section 4).

Practical Implications. Our results have several implications:

- Designers of concurrent algorithms can use our results to determine when looking for a design without RAW and AWAR is futile. Conversely, our results indicate when avoidance of these patterns may be possible.
- For processor architects, our result indicates the importance of optimizing the performance of atomic operations such as compare-and-swap and RAW fence instructions, which have historically received little attention for optimization.
- For synthesis and verification of concurrent algorithms, our result is potentially useful in the sense that a synthesizer or a verifier need not generate or attempt to verify algorithms that do not use RAW and AWAR for they are certainly incorrect.

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 necessary formal machinery. We present our result for mutual exclusion in Section 4 and for linearizable objects in Section 5. In Section 6, we show that many widely used specifications satisfy the conditions outlined in Section 5 and hence are subject to our result. We discuss related work in Section 7 and conclude the paper with Section 8.

2. Overview

In this section we explain our results informally, give an intuition of the formal proof presented in later sections and show concurrent algorithms that exemplify our result.

As mentioned already, our result focuses on two practical specifications for concurrent algorithms: mutual exclusion [11, 31] and linearizability [23].

Informally, our result states that if we are to build a mutual exclusion algorithm or a linearizable algorithm, then in certain sequential executions of that algorithm, we must use either RAW or AWAR. That is, if all executions of the algorithm do not use RAW or AWAR, then the algorithm is incorrect.

2.1 Mutual Exclusion

Consider the classic mutual exclusion template shown in Fig. 1. Here we have N processes ($N > 1$), with each process acquiring a lock, entering the critical section, and finally releasing the lock. The specification for mutual exclusion states that we cannot have multiple processes in their critical section at the same time. The template does not show the actual code that each process must execute in its lock, critical, and unlock sections. Further, the code executed by different processes need not be identical.

Process 0:	Process 1:		Process N-1:
$lock_0$: ...	$lock_1$: ...	.....	$lock_{N-1}$: ...
CS_0: ...	CS_1: ...	.....	CS_{N-1}: ...
$unlock_0$: ...	$unlock_1$: ...	.....	$unlock_{N-1}$: ...

Figure 1. N-process mutual exclusion template, for $N > 1$.

```
lock_i:
  while ( ¬ CAS(Lock,FREE,BUSY) );
```

Figure 2. Illustrating AWAR: a simplified snippet of a test-and-set lock acquire.

```
lock_i:
  flag[i] = true;
  while (flag[¬i])...
```

Figure 3. Illustrating RAW: simplified snippet from the lock section of Dekker's 2-way mutual exclusion algorithm (here $0 \leq i < 2$).

Our result states that whenever a process has sequentially executed its $lock$ section, then this execution must use RAW or AWAR. Otherwise, the algorithm does not satisfy the mutual exclusion specification and is incorrect.

2.1.1 Informal Proof Explanation

Let us now give an intuition for the proof on a simplified case where the system is in its initial state, i.e., all processes are just about to enter their respective $lock$ sections, but have not yet done so. Let us pick an arbitrary process i, $0 \leq i < N$, and let process i sequentially execute its $lock_i$ section, enter the critical section CS_i, and then stop. Let us assume that process i did not perform a shared write when it executed its $lock_i$ section.

Now, let us select another process $j \neq i$, $0 \leq j < N$. As process i did not write to the shared state, there is no way for process j to know where process i is. Therefore, process j can fully execute its own $lock_j$ section and enter the critical section CS_j. Now, both processes are inside the critical section, violating mutual exclusion. Therefore we have shown that each process must perform a shared write in its $lock$ section.

Let us now repeat the same exercise and assume that all processes are in the initial state, where they are all just about to enter their respective $lock$ sections, but have not yet done so. We know that each process must write to shared memory in the sequential execution of its lock section. Let us again pick process i to execute its $lock_i$ section sequentially. Assume that process i writes to shared location named X. Now, let us assume that the $lock_i$ section is executed by process i sequentially without using RAW and AWAR. Since there is no AWAR, it means that the write to X cannot be executed atomically with a previous shared read (be it a read from X or another shared location). There could still be a shared read in $lock_i$ that precedes the write to X, but that read cannot execute atomically with the write to X. Let us now have process i execute until it is about to perform its first shared write operation, and then stop. Now, let process j perform a full sequential execution of its $lock_j$ section (this is possible as process i has not yet written to shared memory so process j is not perturbed). Process j now enters its critical section CS_j and stops. Process i now resumes its $lock_i$

section and immediately performs the shared write to X. Once process i writes to X, it over-writes any changes to X that process j made. This means that if process i is to know where process j is, it must read a shared memory location other than X. However, we assumed that there is no RAW which means that process i cannot read a shared location other than X, without previously having written to that location. In turn, this implies that process i cannot observe where process j is, that is, process j cannot influence the execution of process i. Hence, process i continues and completes its $lock_i$ section and enters its critical section, leading to a violation of mutual exclusion. Therefore, any sequential execution of a $lock$ section requires the use of either AWAR or RAW.

2.1.2 Examples

Here, we show several examples of mutual exclusion algorithms that indeed use either RAW or AWAR in their lock sections. These examples are specific implementations that highlight the applicability of our result, namely that implementation of algorithms that satisfy the mutual exclusion specification cannot avoid both RAW and AWAR.

One of the most common lock implementations is based on the test-and-set atomic sequence. 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. 2 shows a simplified version of a test-and-set-lock. Similar pattern is used in all other locks that require the use of read-modify-write atomic operations in every lock acquire [2, 18, 36].

On the other hand, a mutual exclusion lock algorithm that avoids AWAR [6, 11, 41], must use RAW. For example, Fig. 3 shows a simplified snippet from the lock section of Dekker's algorithm [11] for 2-process mutual exclusion. A process that succeeds in entering its critical section must first raise its own flag and then read the other flag to check that the other process's flag is not raised. Thus, the lock section involves a RAW pattern.

2.2 Linearizability

The second part of our result discusses linearizable algorithms [23]. Intuitively, an algorithm is linearizable with respect to a sequential specification if each execution of the algorithm is equivalent to some sequential execution of the specification, where the order between the non-overlapping methods is preserved. The equivalence is defined by comparing the arguments and results of method invocations.

Unlike mutual exclusion where *all* sequential executions of a certain method (i.e., the $lock$ section) must use either RAW or AWAR, in the case of linearizability, only *some* sequential executions of specific methods must use either RAW or AWAR. We quantify these methods and their executions in terms of properties on *sequential* specifications. Any algorithm implementation that claims to satisfy these properties on the sequential specifications is subject to our results. The two properties are:

- *Deterministic sequential specifications*: Informally, we say that a sequential specification is deterministic if a method executes from the same state will always produce the same result. Many classic abstract data types have deterministic specifications: sets, queues, etc.
- *Strongly non-commutative methods*: Informally, a method m_1 is said to be strongly non-commutative if there exists some state in the specification from which m_1 executed sequentially by process p can influence the result of a method m_2 executed sequentially by process q, $q \neq p$, and vice versa, m_2 can influence the result of m_1 from the same state. Note that m_1 and m_2 are performed by different processes.

$$\{\mathcal{S} = \mathcal{A}\} \quad \texttt{contains}(k) \quad \{ret = k \in \mathcal{A} \wedge \mathcal{S} = \mathcal{A}\}$$

$$\{\mathcal{S} = \mathcal{A}\} \quad \texttt{add}(k) \quad \{ret = k \notin \mathcal{A} \wedge \mathcal{S} = \mathcal{A} \cup \{k\}\}$$

$$\{\mathcal{S} = \mathcal{A}\} \quad \texttt{remove}(k) \quad \{ret = k \in \mathcal{A} \wedge \mathcal{S} = \mathcal{A} \setminus \{k\}\}$$

Figure 4. Sequential specification of a set. $\mathcal{S} \subset \mathbb{N}$ denotes the contents of the set. ret denotes the return value.

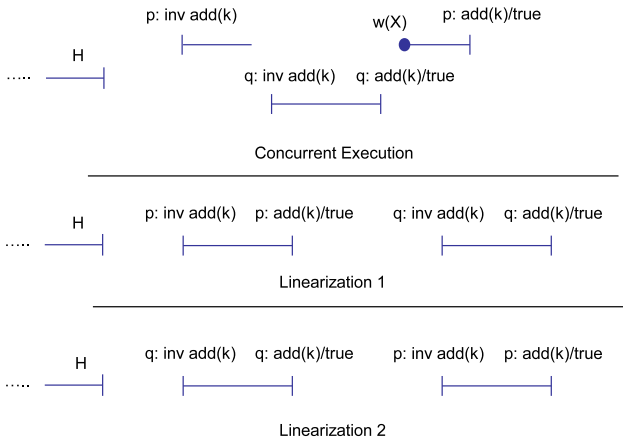

Figure 5. Illustration of the reasoning for why RAW is required in linearizable algorithms.

Our result states that if we have an implementation of a strongly non-commutative method m, then there are *some sequential* executions of m that must use RAW or AWAR. That is, if all sequential executions of m do not use RAW or AWAR, then the algorithm implementation is not linearizable with respect to the given sequential specification.

Let us illustrate these concepts with an example: a Hoare-style sequential specification of a classic Set, shown in Fig. 4 where each method can be executed by more than one process.

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

Second, both methods, `add` and `remove` are strongly non-commutative. For `add`, *there exists* an execution of the specification by a process such that `add` can influence the result of `add` which is executed by another process. For example, let us begin with $\mathcal{S} = \emptyset$. Then, if process p performs an `add(5)`, it will return *true* and a subsequent `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 method as there exists another method where both method invocations influence each other's result starting from some state (i.e.,$\mathcal{S} = \emptyset$). In this case it happens to be another `add` method, but in general the two methods could be different. Similar reasoning shows why `remove` is strongly non-commutative. However, `contains` is *not* a strongly non-commutative method, as even though its result can be influenced by a preceding `add` or `remove`, its execution cannot influence the result of any of the three methods `add`, `remove` or `contains`, regardless of the state from which `contains` starts executing.

For the Set specification, our result states that *any* linearizable implementation of the strongly non-commutativie methods `add` and `remove` *must* use RAW or AWAR in *some sequential* execution of the implementation. For example, let us consider a sequential execution of `add(k)` starting from a state where $k \notin \mathcal{S}$. Then this sequential execution must use RAW or AWAR. However, our result does not apply to the sequential execution of `add(k)` where $k \in \mathcal{S}$. In that case, regardless of whether `add(k)` is performed, the result of any other subsequent method performed right after `add(k)` is unaffected.

2.2.1 Informal Proof Explanation

The proof steps in the case of linearizable implementations are very similar to the ones already outlined in the case of mutual exclusion implementations. Intuitively, if a method is strongly non-commutative, then any of its sequential executions must perform a shared write. Otherwise, there is no way for the method to influence the result of any other method that is executed after it, and hence the method cannot be strongly non-commutative. Let us illustrate how we reason about why RAW or AWAR should be present on our set example.

By contradiction, let us assume that RAW and AWAR are not present. Consider the concurrent execution in Fig. 5. Here, some prefix of the execution marked as H has completed and at the end of H, $k \notin \mathcal{S}$. Then, process p invokes method `add(k)` and executes it up to the first shared write (to a location called X), and then p is preempted. Then, another process q performs a full sequential execution of `add(k)` (for the same k) which returns *true*. After that, p resumes its execution and immediately performs the shared write, and completes its execution of `add(k)` and also returns *true*. The reason why both returned *true* is similar to the case for mutual exclusion: the write to X by p overwrites any writes to X that q has made and as we assumed that RAW is not allowed, it follows that process p cannot read any locations other than X in its subsequent steps without having previously written to them. Hence, both `add(k)`'s return the same value $true$.

Now, if the algorithm is linearizable, there could only be two valid linearizations as shown in Fig. 5. However, it is easy to see that both linearizations are incorrect as they do not conform to the specification: if $k \notin \mathcal{S}$ at the end of H, then according to the set specification, executing two `add(k)`'s sequentially in a row cannot lead to both `add(k)`'s returning the same result. Therefore, either RAW or AWAR must be present in some sequential executions of `add(k)`.

More generally, as we will see in Section 5, we show this for any deterministic specification, not only for sets. We will see that the central reason why both linearizations are not allowed is because the result of `add(k)` executed by process q is not influenced by the preceding `add(k)` executed by process p, violating the assumption that `add()` is a strongly non-commutative method.

2.2.2 Practical Implications

While our result shows when it is impossible to eliminate both RAW and AWAR, the result can also be used to guide the search for linearizable algorithms where it may be possible to eliminate RAW and AWAR, by changing one or more of the following dimensions:

- *Deterministic Specification*: change the sequential specification, perhaps by considering non-deterministic specifications.
- *Strong Non-Commutativity*: focus on methods that are not strongly non-commutative, i.e.,`contains` instead of `add`.
- *Single-Owner*: restrict the specification such that a method can only be performed by a single process, instead of multiple processes (as we will see later, technically, this is also part of the strong non-commutativity definition).

```
   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 6. Adapted snippet from Luchagco et al.'s [34] wait-free CAS algorithm.

- *Execution Detectors*: design efficient detectors that can identify executions which are known to be commutative.

The first three of these pertain to the specification and we illustrate two of them (deterministic specification and single-owner) in the examples that follow. The last one is focused on the implementation. As mentioned already, for linearizability our result holds for *some sequential* executions. However, when implementing an algorithm, it may be difficult to differentiate the sequential executions of a given method for which the result holds and those for which it does not. However, if a designer is able to come up with an efficient mechanism to identify these cases, it may be possible to avoid RAW and AWAR in the executions where it may not be required. For instance, if the method can check that $k \in \mathcal{S}$ before `add(k)` is performed, then for those sequential executions of `add(k)` it may not need to use neither RAW nor AWAR.

Even though our result only talks about *some* sequential executions, in practice, it is often difficult to design efficient tests that differentiate sequential executions, and hence, it often ends up the case that RAW or AWAR is used on *all* sequential executions of a strongly non-commutative linearizable method.

2.3 Examples: Linearizable Algorithms

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

2.3.1 Compare and Swap

We begin with the universal compare-and-swap (CAS) construct, whose sequential specification is deterministic, and the method is strongly non-commutative (for a formal proof, see Section 6). The sequential specification of CAS(m, o, n) says that it first compares $[m]$ to o and if $[m] = o$, then n is assigned to $[m]$ and CAS returns *true*. Otherwise, $[m]$ 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, the 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. 6 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, the algorithm does indeed include the RAW pattern in its common path. To ensure correctness, the write to `b.X` in line 16 *must* precede the read of `b.Y` in line 17.

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

```
    WorkItem take() {
1:    b = bottom;
2:    CircularArray a = activeArray;
3:    b = b - 1;
4:    bottom = b;
5:    t = top;
      ...
```

Figure 7. Snippet adapted from the `take` method of Chase-Lev's work stealing algorithm [10].

```
    WorkItem take() {
1:    h = head;
2:    t = tail;
3:    if (h = t) return EMPTY;
4:    task = tasks.array[h%tasks.size];
5:    head = h+1;
6:    return task;
    }
```

Figure 8. The `take` method from Michael et al.'s idempotent work stealing FIFO queue [38].

2.3.2 Work Stealing Structures

Concurrent work stealing algorithms are popular algorithms for implementing 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 methods: `put`, `take`, and `steal`. Only the owner can insert and extract work items via methods `put` and `take`. Other processes (thieves) may extract work items using `steal`.

In designing algorithms for work stealing, the highest priority is to optimize the owner's methods, especially the common paths of such methods, as they are expected to be the most frequently executed parts of the methods. Examining known work stealing algorithms that avoid the AWAR pattern (i.e., avoid the use of complex atomic operations) in the common path of the owner's methods [3, 16, 19, 20], reveals that they all contain the RAW pattern in the common path of the `take` method that succeeds in extracting work items.

The work stealing algorithm by Chase and Lev [10] is representative of such algorithms. Fig. 7 shows a code snippet adapted from the common path of the `take` method 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. The key pattern in this code snippet is the RAW pattern in lines 4 and 5. The order of the write to `bottom` in line 4 followed by the read of `top` in line 5 is necessary for the correctness of the algorithm. Reversing the order of these two instructions results in an incorrect algorithm. In subsequent sections, we will see why correct implementations of the `take` and `steal` methods must use either RAW or AWAR.

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 both RAW and AWAR. However, as mentioned earlier, our result can guide us towards finding practical cases where we can indeed eliminate RAW and AWAR. Indeed, relaxing the deterministic specification may allow us to come up with algorithms that avoid both RAW and AWAR. 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 AWAR and RAW patterns in the owner's methods.

Our result explains the underlying reason of why the elimination of RAW and AWAR was possible: because the sequential specification of idempotent structures is necessarily non-deterministic, our result now indicates that it may be possible to avoid RAW and AWAR. Indeed, this is confirmed by the algorithms in [38]. Fig. 8 shows the `take` method of one of the idempotent algorithms. Note the absence of both AWAR and RAW in this code. 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.

```
    Data dequeue() {
1:    h = head;
2:    t = tail;
3:    next = h.next;
4:    if head ≠ h goto 1;
5:    if next = null return EMPTY;
6:    if h = t {CAS(tail,t,next) ; goto 1; }
7:    d = next.data;
8:    if ¬CAS(head,h,next) goto 1;
9:    return d;
    }
```

Figure 9. Simplified snippet of `dequeue` on lock-free FIFO queue [37].

```
    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 10. Single-consumer `dequeue` method adapted from Lamport's FIFO queue which does not use RAW and AWAR [30].

2.3.3 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` methods that return a dequeued item. However, as we mentioned already, our result proves that mutual exclusion locking requires each sequential execution of a successful lock acquire to include AWAR or RAW. All algorithms that avoid the use of locking in `dequeue` include a CAS operation in the common path of each nontrivial `dequeue` execution. Fig. 9 shows the simplified code snippet from the `dequeue` method 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 AWAR or RAW patterns (i.e., use either locking or CAS).

From Multi-Owner to Single-Owner Our results suggest that if we want to eliminate RAW and AWAR, we can focus on restricting the processes that can execute a method. For instance, we can specify that `dequeue` can be executed only be a single process. Indeed, when we consider single-consumer FIFO queues, where no more than one process can execute the `dequeue` method, we can obtain a correct implementation of `dequeue` which does not require RAW and AWAR.

$$
\begin{array}{rcll}
x, y & \in & Var & \\
m & \in & \mathit{MID} & \\
l & \in & \mathit{Lab} & \\
B & \in & BExp & ::= \quad ... \\
E & \in & NExp & ::= \quad ... \\
C & \in & Com & ::= \quad l : x = E \mid l : x = G[E] \mid \\
 & & & \quad\; l : G[E] = E \mid l : \textbf{if } B \textbf{ goto } l \mid \\
 & & & \quad\; l : \textbf{beg-atomic} \mid l : \textbf{end-atomic} \\
 & & & \quad\; l : \textbf{entry } m\, \vec{x} \mid l : \textbf{exit } m\, x \\
 & & & \quad\; C; C \\
P & & & ::= \quad C \parallel ... \parallel C
\end{array}
$$

Figure 11. Language Syntax

Fig. 10 shows a single-consumer `dequeue` method, adapted from Lamport's single-producer single-consumer FIFO queue [30].[2] Note that the code avoids both RAW and AWAR. 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 a method of the abstract data type–namely from multi-consumer to single-consumer–it enabled us to create an implementation of the method (i.e., `dequeue`) where we did not need RAW and AWAR.

3. Preliminaries

In this section, we present the formal machinery necessary to specify and prove our results later.

3.1 Language

The language shown in Fig. 11 is a basic assembly language with labeled statements: assignments, sequencing and conditional goto's. We do not elaborate on the construction of numerical and boolean expressions, which are standard. The language is also equipped with the following features:

- Statements for beginning and ending of atomic sections. Using these, one can implement various universal constructs such as compare-and-swap.
- Parallel composition of sequential commands.
- We use *G* to model global memory via a one dimensional array.
- Two statements are used to denote the start (i.e., entry statement) and end of a method (i.e., exit statement).

We use Var to denote the set of local variables for each process, *MID* to denote a finite set of method identifiers, *Lab* the set of program labels and *PID* a finite set of process identifiers. We assume the set of values obtained from expression evaluation includes at least the integers and the booleans.

3.2 Semantics

A *program state* σ is a tuple $\langle pc, locals, G, inatomic\rangle \in \Sigma$:

- $\Sigma = PC \times Locals \times Globals \times InAtomic$
- $PC = \mathit{PID} \rightharpoonup \mathit{Lab}$
- $Locals = \mathit{PID} \times Var \rightharpoonup \mathit{Val}$
- $Globals = \mathit{Val} \rightharpoonup \mathit{Val}$
- $InAtomic = \mathit{PID} \cup \bot$

[2] The restriction or the lack of restriction on the number of concurrent producers does not affect the algorithm for the `dequeue` method.

A state σ tracks the program counter for each process (pc), a mapping from process local variables to values ($locals$), the contents of global memory (*G*) and whether a process executes atomically ($inatomic$). If no process executes atomically then $inatomic$ is set to $\bot$. We denote the set of initial states as $Init \subseteq \Sigma$ (Initially $inatomic$ is set to $\bot$ for all states in $Init$).

Transition Function We assume standard small-step operational semantics given in terms of transitions between states [45]. The behavior of a program is determined by a partial transition function $TF\colon \Sigma \times \mathit{PID} \rightharpoonup \Sigma$. Given a state σ and a process p, $TF(\sigma, p)$ returns the unique state, if it exists, that the program will evolve into once p executes its enabled statement. When convenient, we sometimes use the function TF as a relation.

For a transition $t \in TF$, we denote its source state by $src(t)$, its executing process by $proc(t)$, its destination state by $dst(t)$, its executing statement by $stmt(t)$. A program transition represents the intuitive fact that starting from a state $src(t)$, process $proc(t)$ can execute the statement $stmt(t)$ and end up in a state $dst(t)$, that is, $TF(src(t), proc(t)) = dst(t)$. We say that a transition t performs a global read (resp. write) if $stmt(t)$ reads from *G* (resp. writes to) and use $mloc(t)$ to denote the global memory location that the transition accesses. If the transition does not read or write a global location, then $mloc(t)$ returns $\bot$. That is, only in the case where a transition t accesses a global memory location does $mloc(t)$ return a non-$\bot$ value, otherwise, $mloc(t)$ always returns $\bot$.

We enforce strong atomicity semantics: for any state σ, process p can make a transition from σ only if $inatomic_\sigma = \bot$ or $inatomic_\sigma = p$. For a transition t, if $stmt(t) =$ **beg-atomic**, then $inatomic_{dst(t)} = proc(t)$. Similarly, if $stmt(t) =$ **end-atomic**, $inatomic_{dst(t)} = \bot$. We use $enabled_\sigma$ to denote the set of processes that can make a transition from σ. If $inatomic_\sigma \neq \bot$, then $enabled_\sigma = \{inatomic_\sigma\}$, otherwise $enabled_\sigma = \mathit{PID}$.

The statement **entry** $m\, \vec{x}$ is used to denote the start of a method invoked with a sequence of variables which contain the arguments to the method (denoted by $\vec{x}$). The statement **exit** $m\ x$ is used to denote the end of a method m. These statements do not affect the program state (except the program counter). The meaning of the other statements in the language is standard.

Executions An execution π is a (possibly infinite) sequence of transitions $\pi_0, \pi_1, \ldots$, where $\forall i \geq 0$, $\pi_i \in TF$ and $\forall j \geq 1$. $dst(\pi_{j-1}) = src(\pi_j)$. We use *first*($\pi$) as a shortcut for $src(\pi_0)$, i.e., the first state in the execution π, and, *last*(π) to denote the last state in the execution π, i.e., *last*(π) $= dst(\pi_{|\pi|-1})$. If a transition t is performed in an execution π then $t \in \pi$ is *true*, otherwise it is *false*.

For a program Prog, we use $[\![\text{Prog}]\!]$ to denote the set of executions for that program starting from initial states (e.g. states in $Init$). Next, we define what it means for an execution $\pi \in [\![\text{Prog}]\!]$ to be *atomic*:

Definition 3.1 (Atomic Execution)**.** *We say that an execution π is executed atomically by process p when:*

- *All transitions are performed by p:*
  $\forall i.\ 0 \leq i < |\pi|.\ proc(\pi_i) = p$.
- *All transitions are atomic:*
  $|\pi| = 1$ *or* $\forall i.\ 1 \leq i < |\pi|,\ inatomic_{src(\pi_i)} = p$.

We use $\pi_{(i,j)}$ to denote the substring of π occurring between positions i and j (including the transitions at i and j).

Definition 3.2 (Maximal Atomic Cover)**.** *Given an execution π and a transition $\pi_k \in \pi$, the* maximal atomic cover *of π_k in π is the unique substring $\pi_{(i,j)}$ of π, where:*

- *$\pi_{(i,j)}$ is executed atomically by $proc(\pi_k)$, where $i \leq k \leq j$.*

- $inatomic_{src(\pi_i)} = \bot$.
- $inatomic_{dst(\pi_j)} = \bot$.

Intuitively, we can understand the maximal atomic cover as taking a transition and extending it in both directions until we reach a leftmost state and a rightmost state where no process is inside an atomic section in either of these two states.

Next, we define read-after-write executions:

Definition 3.3 (Read After Write Execution)**.** *We say that a process p performs a* read-after-write *in execution π, if $\exists i, j.\ 0 \leq i < j < |\pi|$ such that:*

- *π_i performs a global write by process p.*
- *π_j performs a global read by process p.*
- *$mloc(\pi_i) \neq mloc(\pi_j)$ (the memory locations are different).*
- *$\forall k.\ i < k < j$, if $proc(\pi_k) = p$, then $mloc(\pi_k) \neq mloc(\pi_j)$.*

Intuitively, these are executions where somewhere in the execution the process writes to global memory location A and then, sometimes later, reads a global memory location B that is different from A, and in-between the process does not access B. Note that there could be transitions in π performed by processes other than p. Note that in this definition there is no restriction on whether the global accesses are performed atomically or not, the definition only concerns itself with the ordering of accesses and not their atomicity.

We introduce the predicate $\texttt{RAW}(\pi, p)$ which evaluates to *true* if p performs a *read-after-write* in execution π and to *false* otherwise.

Next, we define atomic write-after-read executions. 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 accesses occur atomically, that is, in-between these two accesses, no other process can perform any transitions. Note that unlike read-after-write executions, here, the global read and write need *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 π, if $\exists i, j.\ 0 \leq i < j < |\pi|$ such that:*

- *process p performs a global read in π_i.*
- *process p performs a global write in π_j.*
- *$\pi_{(i,j)}$ is executed atomically by process p.*

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

3.3 Specification

3.3.1 Histories

A *history* H is defined as a finite sequence of actions, i.e., $H = \psi; \psi ...; \psi$, where an action denotes the start and end of a method:

$$\psi \ = \ (p, \textbf{entry}\ m\ \vec{a}) \mid (p, \textbf{exit}\ m\ r)$$

where $p \in$ *PID* is a process identifier, $m \in$ *MID* is a method identifier, $\vec{a}$ is a sequence of arguments to the method and r is the return value. For an action ψ, we use $proc(\psi)$ to denote the process, $kind(\psi)$ to denote the kind of the action (entry or exit), and $m(\psi)$ to denote the name of the method. We use H_i to denote the action at position i in the history, where $0 \leq i < |H|$. For a process p, $H \lfloor_p$ is used to denote the subsequence of H consisting only of the actions of process p. For a method m, $H \lfloor_m$ is used to denote the subsequence of H consisting only of the actions of method m.

A method entry $(p, \textbf{entry}\ m_1\ \vec{a_1})$ is said to be *pending* in a history H if it has no matching exit, that is, $\exists i.\ 0 \leq i < |H|$ such that $proc(H_i) = p$, $kind(H_i) = entry$, $m(H_i) = m_1$ and $\forall j.\ i < j < |H|$, $proc(H_j) \neq p$ or $kind(H_j) \neq exit$ or $m(H_j) \neq m_1$. A history H is said to be *complete* if it has no pending calls. We use *complete*(H) to denote the set of histories resulting after extending H with matching exits to a subset of entries that are pending in H and then removing the remaining pending entries.

A history H is *sequential* if H is empty ($H = \epsilon$) or H starts with an entry action, i.e., $kind(H_0) = entry$ and if $|H| > 1$, entries and exits alternate. That is, $\forall i.\ 0 \leq i < |H| - 1, kind(H_i) \neq kind(H_{i+1})$ and each exit is matched by an entry that occurs immediately before it in H, i.e., $\forall i.\ 1 \leq i < |H|$, if $kind(H_i) = exit$ then $kind(H_{i-1}) = entry$ and $proc(H_i) = proc(H_{i-1})$. A complete sequential history H is said to be a *complete invocation* of a method m_1 iff $|H| = 2$, $m(H_0) = m_1$ and $m(H_1) = m_1$. In the case where H is a complete sequential invocation, we use $entry(H)$ to denote the entry action in H and $exit(H)$ to denote the exit action in H. A history H is *well-formed* if for each process $p \in$ *PID*, $H \lfloor_p$ is sequential. In this work, we consider only well-formed histories.

3.4 Histories and Executions

Given an execution π, we use the function $hs(\pi)$ to denote the history of π. $hs(\pi)$ takes as input an execution π and produces a sequence of actions by iterating over each transition $t \in \pi$ in order, and extracting $proc(t)$ and $stmt(t)$. If $stmt(t)$ is an entry statement of a method m, then the transition t contributes the action $(proc(t), \textbf{entry}\ m\ \vec{a})$, where $\vec{a}$ is the sequence of values obtained from evaluating the variables used in the sequence $stmt(t)$, in state $src(t)$. Similarly, for exit statements. If the transition t does not perform an entry or an exit statement, then it contributes ϵ.

For a program Prog, we define its corresponding set of histories as $[\![\text{Prog}]\!]_H = \{hs(\pi) \mid \pi \in [\![\text{Prog}]\!]\}$. We use $[\![\text{Prog}]\!]_{HS}$ to denote the sequential histories in $[\![\text{Prog}]\!]_H$.

A transition $t \in \pi$ is said to be a *method* transition if it is performed in-between method entry and exit. That is, there exists a preceding transition $t_{prev} \in \pi$ that performs an entry statement with $proc(t_{prev}) = proc(t)$, such that $proc(t)$ does not perform an exit statement in-between t_{prev} and t in π. We say that t_{prev} is a matching entry transition for t. Note that t_{prev} may be the same as t. A transition that is not a method transition is said to be a *client* transition.

Definition 3.5 (Well-formed Execution)**.** *We say that an execution π is well-formed if:*

- hs(π) *is well-formed.*
- *Any client transition $t \in \pi$:*
  - $mloc(t) = \bot$.
  - $stmt(t) \neq$ ***beg-atomic*** *and* $stmt(t) \neq$ ***end-atomic***.
- *For any transition $t \in \pi$, if $stmt(t)$ is an exit statement, then $inatomic_{src(t)} = \bot$.*
- *For any transition $t_r \in \pi$, if t_r is a method transition that reads a local variable other than the variables specified in the statement of its matching entry transition t_m, then there exists a transition t_w performed by process $proc(t)$, in-between t_m and t_r, such that t_w writes to that local variable.*

That is, a well-formed execution is one where its history is well-formed, only method transitions are allowed to access global memory or perform atomic statements, when a method exit statement is performed, the $inatomic$ should be $\bot$, and methods must initialize local variables which are not used for argument passing before using them.

We say that π is a *complete sequential execution* of a method m by process p, if π is a well-formed execution and $hs(\pi)$ is a complete invocation of m by process p. Note that π may contain client transitions (both by process p and other processes).

A program Prog is *well-formed* if ⟦Prog⟧ contains 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 either the RAW or the AWAR pattern in certain executions.

A mutual exclusion implementation exports the following methods: $MID = \{lock_0, unlock_0, \ldots, lock_{n-1}, unlock_{n-1}\}$, where $n = |PID|$. In this setting, we strengthen the definition of well-formed executions by requiring that each process $p \in PID$ only invokes methods $lock_p$ and $unlock_p$ in an alternating fashion. That is, for any execution $\pi \in$ ⟦Prog⟧ and for any process $p \in PID$, $hs(\pi) \downarrow_p$ is such that lock and unlock operations alternate, i.e., $lock_p, unlock_p, lock_p, \ldots$.

Given an execution $\pi \in$ ⟦Prog⟧ and $H = hs(\pi) \downarrow_p$, we say that p is in its trying section if it has started, but not yet completed a $lock_p$ operation, i.e., $m(H_{|H|-1}) = lock_p$ and $kind(H_{|H|-1}) = entry$. We say that p is in its critical section if it has completed $lock_p$ but has not yet started $unlock_p$, that is, $m(H_{|H|-1}) = lock_p$ and $kind(H_{|H|-1}) = exit$. We say that p is in its exit section if it has started $unlock_p$ but has not yet finished it, that is, $m(H_{|H|-1}) = unlock_p$ and $kind(H_{|H|-1}) = entry$. Otherwise we say that p is in the remainder section (initially all processes are in their remainder sections). A process is called *active* if it is in its trying or exit section.

For the purpose of our lower bound, we assume the following weak formulation of the mutual exclusion problem [11, 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 Prog guarantees:*

- *Safety: For all executions $\pi \in$ ⟦Prog⟧, it is always the case that at most one process is in its critical section at a time, that is, for all $p, q \in$ PID, if p is in its critical section in* $\text{hs}(\pi) \downarrow_p$ *and q is in its critical section in* $\text{hs}(\pi) \downarrow_q$*, then $p = q$.*
- *Liveness: In every execution in which every active process takes sufficiently many steps: i) if at least one process is in its trying section and no process is in its critical section, then at some point later some process enters its critical section, and ii) if at least one process is in its exit section, then at some point later some process enters its remainder section.*

Theorem 4.2 (RAW or AWAR in Mutual Exclusion)**.** *Let Prog be a deadlock-free mutual exclusion implementation for two or more processes (*$|\text{PID}| > 1$*). Then, for every complete sequential execution π of $lock_p$ by process p:*

- $\texttt{RAW}(\pi, p) =$ true*, or*
- $\texttt{AWAR}(\pi, p) =$ true

Proof. Let $\pi_{base} \cdot \pi \in$ ⟦Prog⟧ such that π is a complete sequential execution of $lock_p$ by process p. It follows that no process $q \in PID$, $q \neq p$, is in its critical section in $hs(\pi_{base}) \downarrow_q$ (otherwise mutual exclusion would be violated). It also follows that p is not active in $hs(\pi_{base}) \downarrow_p$,

By contradiction, assume that π does not contain a global write. Consider an execution $\pi_{base} \cdot \rho$ such that process p does not perform transitions in ρ and every active process takes sufficiently many steps in ρ until some process $q \neq p$ completes its $lock_q$ section, i.e., q is in its critical section in $hs(\pi_{base} \cdot \rho) \downarrow_q$. The execution $\pi_{base} \cdot \rho \in$ ⟦Prog⟧ since Prog is deadlock-free.

Since p does not write to a shared location in π, $G_{last(\pi_{base})} = G_{last(\pi_{base} \cdot \pi)}$. Further, the local state of all processes other than p in $last(\pi_{base})$ is the same as their local state in $last(\pi_{base} \cdot \pi)$, i.e., $\forall q \in PID, \forall var \in Var$, if $q \neq p$, then $locals_{last(\pi_{base})}(q, var) = locals_{last(\pi_{base} \cdot \pi)}(q, var)$. Also, we know $enabled_{last(\pi_{base})} = enabled_{last(\pi_{base} \cdot \pi)}$ as transitions by process p do not access local variables of other processes. Hence, we can build the execution $\pi_{nc} = \pi_{base} \cdot \pi \cdot \rho'$ where ρ' is the execution with the same sequence of statements as ρ (i.e., process p does not perform transitions in ρ'). Hence, $hs(\pi_{nc}) \downarrow_q = hs(\pi_{base} \cdot \rho) \downarrow_q$, that is, q is in its critical section in $hs(\pi_{nc}) \downarrow_q$. But p is also in its critical section in $hs(\pi_{nc}) \downarrow_p$ — a contradiction.

Thus, π contains a global write, and let t_w be the *first* global write transition in π. Let $\pi = \pi_f \cdot \pi_w \cdot \pi_l$, where π_w is the maximal atomic cover of t_w in π. We proceed by contradiction and assume that $\texttt{RAW}(\pi, p) =$ *false* and $\texttt{AWAR}(\pi, p) =$ *false*. Since $\texttt{AWAR}(\pi, p) =$ *false* and t_w is the first write transition in π, it follows that t_w is the first global transition in π_w.

Since π_f contains no global writes, $G_{first(\pi_f)} = G_{last(\pi_f)}$. Applying the same arguments as before, there exists an execution $\pi_{base} \cdot \pi_f \cdot \tau \in$ ⟦Prog⟧ such that some process q, $q \neq p$, is in its critical section in $hs(\pi_{base} \cdot \pi_f \cdot \tau) \downarrow_q$.

The assumption $\texttt{RAW}(\pi, p) =$ *false* implies that no global read transition by process p in $\pi_w \cdot \pi_l$ accesses a variable other than $mloc(t_w)$ without having previously written to it. Note that t_w overwrites the only location that can be read by p in $\pi_w \cdot \pi_l$. Thus, applying the same arguments as before, there exists an execution $\pi_c = \pi_{base} \cdot \pi_f \cdot \tau \cdot \pi'_w \cdot \pi'_l$ in ⟦Prog⟧ such that q is in its critical section in $hs(\pi_c) \downarrow_q$ and p is in its critical section in $hs(\pi_c) \downarrow_p$ —a contradiction.

Thus, either $\texttt{RAW}(\pi, p) =$ *true* or $\texttt{AWAR}(\pi, p) =$ *true*. □

5. Synchronization in Linearizable Algorithms

In this section we state and prove that certain sequential executions of strongly non-commutative methods of algorithms that are linearizable with respect to a deterministic sequential specification must use RAW or AWAR.

5.1 Linearizability

Following [21, 23] we define linearizable histories. A history H induces an irreflexive partial order $<_H$ on actions in the history: $a <_H b$ if $kind(a) = exit$ and $kind(b) = entry$ and $\exists i, j.\ 0 \leq i < j < |H|$ such that $H_i = a$ and $H_j = b$. That is, exit action a precedes entry action b in H. A history H is said to be *linearizable* with respect to a sequential history S if there exists a history $H' \in complete(H)$ such that:

1. $\forall p \in PID, H' \downarrow_p = S \downarrow_p$
2. $<_H \subseteq <_S$.

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

We say that a program Prog is linearizable with respect to a sequential specification $Spec$ when $\llbracket \text{Prog} \rrbracket_H$ is linearizable with respect to $Spec$.

5.2 Deterministic Sequential Specifications

In this paper, similarly to [8], we define deterministic sequential specifications. Given two sequential histories s_1 and s_2, let $\texttt{maxprefix}(s_1, s_2)$ denote the longest common prefix of the two histories s_1 and s_2.

Definition 5.1 (Deterministic Sequential Specifications)**.** *A sequential specification $Spec$ is deterministic, if for all $s_1, s_2 \in Spec$, $s_1 \neq s_2$ and $\hat{s} = \texttt{maxprefix}(s_1, s_2)$, we have $\hat{s} = \epsilon$ or $kind(\hat{s}_{|\hat{s}|-1}) \neq entry$.*

That is, a specification is deterministic, if we cannot find two different histories whose longest common prefix ends with an entry. 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 entry, they produced different results (or one had no continuation).

5.3 Strong Non-Commutativity

We define a *strongly non-commutative* method as follows:

Definition 5.2 (Strongly Non-Commutative Method)**.** *We say that a method m_1 is* strongly non-commutative *in a sequential specification $Spec$ if there exists a method m_2 (possibly the same as m_1), and there exist histories base, s_1, s_2, s_3, s_4 such that:*

1. *s_1 and s_4 are complete invocations of m_1 with $entry(s_1) = entry(s_4)$ and $exit(s_1) \neq exit(s_4)$.*
2. *s_2 and s_3 are complete invocations of m_2 with $entry(s_2) = entry(s_3)$ and $exit(s_2) \neq exit(s_3)$.*
3. *$proc(entry(s_1)) \neq proc(entry(s_2))$.*
4. *base is a complete sequential history in Spec.*
5. *base $\cdot\, s_2 \cdot s_4 \in Spec$.*
6. *base $\cdot\, s_1 \cdot s_3 \in Spec$.*

In other words, the method m_1 is strongly non-commutative if there is another method m_2 and a history *base* in $Spec$ such that we can distinguish whether m_1 is applied right after *base* or right after m_2 (which is applied after *base*). Similarly we can distinguish whether m_2 is applied right after *base* or right after m_1 (which is applied after *base*). Note that m_2 may be the same method as m_1.

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

Assumption 1. $Spec = [\![Prog]\!]_{HS}$.

5.4 RAW and AWAR for Linearizability

Next, we state and prove the main result of this section:

Theorem 5.3 (RAW or AWAR in Linearizable Algorithms)**.** *Let m_1 be a strongly non-commutative method in a deterministic sequential specification $Spec$ and let $Prog$ be a linearizable implementation of $Spec$. Then there exists a complete sequential execution π_a of m_1 by process p such that:*

- $\texttt{RAW}(\pi_a, p) =$ true*, or*
- $\texttt{AWAR}(\pi_a, p) =$ true

Proof. From the premise that m_1 is a strongly non-commutative method and Assumption 1, we know that there exist executions $\pi_{base_1} \cdot \pi_a \cdot \pi_c \in [\![\text{Prog}]\!]$ and $\pi_{base_2} \cdot \pi_b \cdot \pi_d \in [\![\text{Prog}]\!]$ such that:

1. $hs(\pi_{base_1})$ and $hs(\pi_{base_2})$ are complete sequential histories.
2. $hs(\pi_{base_1}) = hs(\pi_{base_2})$.
3. π_a and π_d are complete sequential executions of m_1.
4. π_b and π_c are complete sequential executions of m_2.
5. $entry(hs(\pi_a)) = entry(hs(\pi_d))$.
6. $entry(hs(\pi_b)) = entry(hs(\pi_c))$.
7. $exit(hs(\pi_a)) \neq exit(hs(\pi_d))$.
8. $exit(hs(\pi_b)) \neq exit(hs(\pi_c))$.
9. $proc(entry(hs(\pi_a))) \neq proc(entry(hs(\pi_b)))$.

From the fact that executions in the program are well-formed, we know that if π, $\rho \in [\![\text{Prog}]\!]$ and $hs(\pi)$, $hs(\rho)$ are complete sequential invocations such that $entry(hs(\pi)) = entry(hs(\rho))$, and $G_{first(\pi)} = G_{first(\rho)}$, it follows that $hs(\pi) = hs(\rho)$ and $G_{last(\pi)} = G_{last(\rho)}$. That is, if a process completes the same method invocation from two program states with identical global memory, the method will always produce the same result and global memory (Fact 1). Fact 1 follows directly from the fact that transitions are deterministic, processes cannot access the local state of another process, arguments to both methods are the same, and the starting global states are the same.

From Fact 1 and $hs(\pi_{base_1})$, $hs(\pi_{base_2})$ being complete sequential histories, we can show that $G_{last(\pi_{base_1})} = G_{last(\pi_{base_2})}$. That is, from $first(\pi_{base_1}) = first(\pi_{base_2})$ (both are initial states), we can inductively show that any complete sequential invocation preserves the fact that the global state in the last states of the two executions are the same. From $G_{last(\pi_{base_1})} = G_{last(\pi_{base_2})}$, it follows that $G_{first(\pi_a)} = G_{first(\pi_b)}$.

Let $p = proc(entry(hs(\pi_a)))$ and $q = proc(entry(hs(\pi_b)))$. We first prove that a method transition performed by process p must perform a global write in π_a. Let us assume the execution π_a does not contain a method transition where process p performs a global write. As π_a is well-formed, we know that any client transitions performed in π_a do not access global memory. It then follows that $G_{first(\pi_a)} = G_{last(\pi_a)}$. However, from the premise we know that $G_{last(\pi_a)} = G_{first(\pi_c)}$ and hence $G_{first(\pi_a)} = G_{first(\pi_c)}$. Transitively, we know that $G_{first(\pi_b)} = G_{first(\pi_c)}$. From item 4 above we know that π_b and π_c are complete sequential executions of m_2 with $entry(hs(\pi_b)) = entry(hs(\pi_c))$ (item 6). Then, it follows from Fact 1 that $exit(hs(\pi_b)) = exit(hs(\pi_c))$ which contradicts with item 8. Therefore, there must exist a method transition in π_a by process p that performs a global write.

Let us proceed by contradiction and assume that both $\texttt{RAW}(\pi_a, p)$ = *false* and $\texttt{AWAR}(\pi_a, p)$ = *false*. Let $\pi_a = \pi_f \cdot \pi_w \cdot \pi_\ell$, where t_w is the *first* method transition in π_a that writes to global memory and π_w is the maximal atomic cover of t_w in π_a. As π_a is well-formed, we know that all transitions in π_w are method transitions. Since $\texttt{AWAR}(\pi_a, p) =$ *false* and t_w is the first global write transition in π_a, it follows that there can be no global read transitions in π_w that occur before t_w (otherwise we would contradict AWAR). This means that t_w is the first *global* read or write transition in π_w.

As π_f does not contain global writes, it follows that $G_{first(\pi_a)} = G_{last(\pi_f)}$. From the premise we know that $G_{first(\pi_a)} = G_{first(\pi_b)}$ and hence $G_{first(\pi_b)} = G_{last(\pi_f)}$. As π_w is a maximal atomic cover, we know that $inatomic_{last(\pi_f)} = \bot$.

From the fact that client transitions cannot synchronize (they cannot execute atomic statements or access global memory), and that a process cannot access the local variables of another process, it follows that process q can execute m_2 concurrently with m_1 from state $last(\pi_f)$. That is, there exists an execution $\pi_{base_1} \cdot \pi_f \cdot \pi'_b \in [\![\text{Prog}]\!]$ where π'_b is a complete sequential execution of m_2 by process q such that $entry(hs(\pi_b)) = entry(hs(\pi'_b))$. As $G_{first(\pi'_b)} = G_{last(\pi_f)}$, it follows that $G_{first(\pi'_b)} = G_{first(\pi_b)}$. Then, by Fact 1, it follows that $hs(\pi'_b) = hs(\pi_b)$.

As $p \in enabled_{first(\pi'_b)}$ and π'_b is a complete sequential execution by process q, it follows that $p \in enabled_{last(\pi'_b)}$. Process p can now continue execution of method m_1 and build the execution $\pi_{base_1} \cdot \pi_{conc} \in [\![\text{Prog}]\!]$ where $\pi_{conc} = \pi_f \cdot \pi'_b \cdot \pi'_w \cdot \pi'_\ell$.

By assumption, we know that $\texttt{RAW}(\pi_a, p)$ and $\texttt{AWAR}(\pi_a, p)$ are *false* and it follows that $\pi_w \cdot \pi_\ell$ does not contain a method transition which reads a global memory location other than $mloc(t_w)$ without previously having written to it. In both, π_w and π'_w, process p overwrites the only global memory location $mloc(t_w)$ that it can read without previously having written to it. Then, $\pi_w \cdot \pi_\ell$ and $\pi'_w \cdot \pi'_\ell$ will contain the same sequence of statements, with all global transitions accessing and reading/writing identical values. Thus, $hs(\pi_{conc}) \lfloor_{m_1} = hs(\pi_a)$.

Given that the implementation is linearizable the two possible linearizations of $\pi_{base_1} \cdot \pi_{conc}$ are:

1. $hs(\pi_{base_1}) \cdot hs(\pi_{conc}) \lfloor_{m_1} \cdot hs(\pi'_b)$. We already established that $hs(\pi_{conc}) \lfloor_{m_1} = hs(\pi_a)$ and $hs(\pi'_b) = hs(\pi_b)$, and hence by substitution we get $hs(\pi_{base_1}) \cdot hs(\pi_a) \cdot hs(\pi_b)$. From the premise, we know that $hs(\pi_{base_1}) \cdot hs(\pi_a) \cdot hs(\pi_c) \in Spec$. As the specification is deterministic, it follows that $hs(\pi_b) = hs(\pi_c)$, a contradiction with item 8.
2. $hs(\pi_{base_1}) \cdot hs(\pi'_b) \cdot hs(\pi_{conc}) \lfloor_{m_1}$. We already established that $hs(\pi_{conc}) \lfloor_{m_1} = hs(\pi_a)$ and $hs(\pi'_b) = hs(\pi_b)$ and hence by substitution we get $hs(\pi_{base_1}) \cdot hs(\pi_b) \cdot hs(\pi_a)$. From the premise, we know that $hs(\pi_{base_2}) \cdot hs(\pi_b) \cdot hs(\pi_d) \in Spec$ and $hs(\pi_{base_1}) = hs(\pi_{base_2})$. As the specification is deterministic, it follows that $hs(\pi_a) = hs(\pi_d)$, a contradiction with item 7.

Therefore, $\texttt{RAW}(\pi_a, p) = \textit{true}$ or $\texttt{AWAR}(\pi_a, p) = \textit{true}$. □

5.5 A Note on Commutativity and Idempotency

The notion of strongly non-commutative method is related to traditional notions of non-commutative methods [44] and non-idempotent methods. Let us again consider Definition 5.2.

Non-Idempotent Method vs. Strongly Non-Commutative Method If it is the case that method m_2 is the same as method m_1, then the definition instantiates to *non-idempotent* methods. That is, given *base*, if we apply m_1 twice in a row, the second invocation will return a different result than the first. Consider again the Set specification in Fig. 4. The method `add` is non-idempotent. As discussed in the example in Section 2, we can start with $\mathcal{S} = \emptyset$ and $base = \epsilon$. Then, if we perform two methods `add(5)` in a row, each one of the `add(5)`'s will return a different result.

Classic Non-Commutativity vs. Strong Non-Commutativity In the classic notion of non-commutativity [44], it is enough for one of the methods to not commute with the other, while here, it is required that both methods do not commute from the *same* prefix history. In the classic case, if two methods do not commute, it does not mean that either of them is a strongly non-commutative method. However, if a method is strongly non-commutative, then it is always the case that there exists another method with which it does not commute (by definition). Consider again the Set specification in Fig. 4. Although `add` and `contains` do not commute, `contains` is not a strongly non-commutative method. That is, `add` influences the result of `contains`, but `contains` does *not* influence the result of `add`.

6. Strongly Non-Commutative Specifications

In this section we provide a few examples of well-known sequential specifications that contain strongly non-commutative methods as defined in Definition 5.2.

6.1 Stacks

Definition 6.1 (Stack Sequential Specification)**.** *A stack object S supports two methods: push and pop. The state of a stack is a sequence of items $S = \langle v_0, ..., v_k \rangle$. The stack is initially empty. The push and pop methods induce the following state transitions of the sequence $S = \langle v_0, \ldots, v_k \rangle$, with appropriate return values:*

- *push(v_{new}): changes S to be $\langle v_0, ..., v_k, v_{new} \rangle$ and returns* ack.
- *pop(): if S is non-empty, changes S to be $\langle v_0, ..., v_{k-1} \rangle$ and returns v_k. If S is empty, returns empty and S remains unchanged.*

We let $Spec_s$ denote the sequential specification of a stack object as defined above.

Lemma 6.2 (Pop is Strongly Non-Commutative)**.** *The* pop *stack method is strongly non-commutative.*

Proof. Let $base \in Spec_s$ be a complete sequential history after which $S = \langle v \rangle$ for some v. Let p and q be two processes, let s_1 and s_4 be complete invocations of *pop* by p, and let s_2 and s_3 be complete invocations of *pop* by q. From Definition 6.1, $\{base \cdot s_1 \cdot s_3, base \cdot s_2 \cdot s_4\} \subset Spec_s$, $ret(s_1) = ret(s_2) = v$, and $ret(s_3) = ret(s_4) = empty$. The claim now follows from Definition 5.2. □

It also follows from Definition 5.2 that *push* methods are not strongly non-commutative.

6.2 Work Stealing

As we now prove, the (non-idempotent) work stealing object, discussed in section 2.2, is an example of an object for which two different methods are strongly non-commutative.

Definition 6.3 (Work Stealing Sequential Specification)**.** *A work stealing object supports three methods: put, take, and steal. The state of each process i is a sequence of items $Q_i = \langle v^i_0, ..., v^i_{k_i} \rangle$. All queues are initially empty. The put and take methods are performed by each process i on its local queue Q_i and induce on it the following state transitions, with appropriate return values:*

- *put(v_{new}): changes Q_i to be $\langle v_{new}, v^i_0, ..., v^i_{k_i} \rangle$ and returns* ack.
- *take(): if Q_i is non-empty, it changes Q_i to be $\langle v^i_1, ..., v^i_{k_i} \rangle$ and returns v^i_0. If Q_i is empty, it returns empty and Q_i remains unchanged.*

The steal *method is performed by each process i on some queue $Q_j = \langle v^j_0, ..., v^j_{k_j} \rangle$ for $j \neq i$. if Q_j is non-empty, it changes Q_j to be $\langle v^j_0, ..., v^i_{k_j - 1} \rangle$ and returns $v^j_{k_j}$. If Q_j is empty, it returns empty and Q_j remains unchanged. We let $Spec_{ws}$ denote the sequential specification of a work stealing object as defined above.*

Lemma 6.4 (Take & Steal are Strongly Non-Commutative)**.** *The* take *and* steal *methods are strongly non-commutative.*

Proof. Let $base \in Spec_{ws}$ be a complete sequential history after which $Q_j = \langle v \rangle$ for some value v and process j. Let $i \neq j$ be some process other than j, let s_1 and s_4 be complete invocations of *steal* 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, $\{base \cdot s_1 \cdot s_3, base \cdot s_2 \cdot s_4\} \subset Spec_s$, $ret(s_1) = ret(s_2) = v$, and $ret(s_3) = ret(s_4) = empty$. The claim now follows from Definition 5.2. □

It is easily shown that specifications for *queue*s, *hash-table*s and *set*s have strongly non-commutative methods. 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 method called CAS and stores a scalar value over some domain $\mathcal{V}$. The method* CAS*(exp,new), for $exp, new \in \mathcal{V}$, 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 method returns true; otherwise, C's value remains unchanged and the method returns false.*
We let $Spec_C$ denote the sequential specification of a compare-and-swap object as defined above.

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

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

It 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 specifications (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 [22] for many other examples.

Modern architectures often execute instructions issued by a single process out-of-order, 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 a full fence, while `lwsync` guarantees all other orders except 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 enforcing the RAW order.

Herlihy [22] proved that linearizable wait-free implementations of many widely-used concurrent data-structures, such as counters, stacks and queues, must use AWAR. These results do not mention RAW and do not apply to obstruction-free [15] implementations of such objects or to implementations of mutual exclusion, however, whereas our results do.

Recently, there has been a renewed interest in formalizing memory models (cf. [40, 42, 43]), and model checking and synthesizing programs that run on these models [29]. 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 both RAW and AWAR are missing from a program that claims to satisfy certain specifications, then that program is certainly incorrect and there is no need test it or verify it.

Kawash's PhD thesis [28] (also in papers [24, 25]) investigates the ability of *weak consistency* models to solve mutual exclusion, with only reads and writes. 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 [17] 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 linearizable 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. 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.

Our proof technique employs the *covering* technique, originally used by Burns and Lynch [9] to prove a lower bound on the number of registers needed for solving mutual exclusion. This technique had many applications, both with read / write operations [4, 5, 12, 14, 27, 39], and with non-trivial atomic operations, such as compare&swap [13]. Some steps of our proofs can be seen as a formalization of the arguments Lamport uses to derive a fast mutual exclusion algorithm [32].

In terms of our result for mutual exclusion, while one might guess that some form of RAW should be used in the entry code of read/write mutual exclusion, we are not aware of any prior work that states and proves this claim. Burns and Lynch [9] show that you need to have *n* registers, and as part of their proof show that a process needs to write, but they do not show that after it writes, the process must read from a different memory location. Lamport [32] also only hints to it. These works neither state nor prove the claim we are making (and they also do not discuss AWAR).

8. Conclusion and Future Work

In this work, we focused on two common synchronization idioms: read-after-write (RAW) and atomic write after read (AWAR). Unfortunately, enforcing any of these two patterns is costly on all current processor architectures.

We showed that it is impossible to eliminate both RAW and AWAR in the sequential execution of a lock section of any mutual exclusion algorithm. We also proved that RAW or AWAR must be present in some of the sequential executions of strongly non-commutative methods that are linearizable with respect to a deterministic sequential specification. Further, we proved that many classic specifications such as stacks, sets, hash tables, queues, work-stealing structures and compare-and-swap operations have strongly non-commutative operations, making implementations of these specifications subject to our result. Finally, as RAW or AWAR cannot be avoided in most practical algorithms, our result suggests that it is important to improve the hardware costs of store-load fences and compare-and-swap operations, the instructions that enforce RAW and AWAR.

An interesting direction for future work is taking advantage of our result by weakening its basic assumptions in order to build useful algorithms that do not use RAW and AWAR.

9. Acknowledgements

We thank Bard Bloom and the anonymous reviewers for valuable suggestions which improved the quality of the paper. Hagit Attiya's research is supported in part by the *Israel Science Foundation* (grants number 953/06 and 1227/10). Danny Hendler's research is supported in part by the *Israel Science Foundation* (grants number 1344/06 and 1227/10).

References

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

[2] Thomas E. Anderson. The performance of spin lock alternatives for shared-money multiprocessors. *IEEE Trans. Parallel Distrib. Syst.*, 1(1):6–16, 1990.

[3] Nimar S. Arora, Robert D. Blumofe, and C. Greg Plaxton. Thread scheduling for multiprogrammed multiprocessors. In *Proceedings of the Tenth Annual ACM Symposium on Parallel Algorithms and Architectures, SPAA*, pages 119–129, June 1998.

[4] Hagit Attiya, Faith Fich, and Yaniv Kaplan. Lower bounds for adaptive collect and related objects. In *Proceedings of the Twenty-Third Annual ACM Symposium on Principles of Distributed Computing*, pages 60–69, 2004.

[5] Hagit Attiya, Alla Gorbach, and Shlomo Moran. Computing in totally anonymous asynchronous shared memory systems. *Information and Computation*, 173(2):162–183, March 2002.

[6] Yoah Bar-David and Gadi Taubenfeld. Automatic discovery of mutual exclusion algorithms. In *Proceedings of the 17th International Conference on Distributed Computing, DISC*, pages 136–150, 2003.

[7] Hans-J. Boehm. Reordering constraints for pthread-style locks. In *Proceedings of the Twevelth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPoPP*, pages 173–182, 2007.

[8] Sebastian Burckhardt, Chris Dern, Madanlal Musuvathi, and Roy Tan. Line-up: a complete and automatic linearizability checker. In *PLDI '10: Proceedings of the 2010 ACM SIGPLAN conference on Programming language design and implementation*, pages 330–340, New York, NY, USA, 2010. ACM.

[9] James Burns and Nancy Lynch. Bounds on shared memory for mutual exclusion. *Information and Computation*, 107(2):171–184, December 1993.

[10] David Chase and Yossi Lev. Dynamic circular work-stealing deque. In *Proceedings of the Seventeenth Annual ACM Symposium on Parallelism in Algorithms and Architectures, SPAA*, pages 21–28, July 2005.

[11] Edsger W. Dijkstra. Solution of a problem in concurrent programming control. *Commun. ACM*, 8(9):569, 1965.

[12] Faith Ellen, Panagiota Fatourou, and Eric Ruppert. Time lower bounds for implementations of multi-writer snapshots. *Journal of the ACM*, 54(6):30, 2007.

[13] Faith Fich, Danny Hendler, and Nir Shavit. On the inherent weakness of conditional primitives. *Distributed Computing*, 18(4):267–277, 2006.

[14] Faith Fich, Maurice Herlihy, and Nir Shavit. On the space complexity of randomized synchronization. *Journal of the ACM*, 45(5):843–862, September 1998.

[15] Faith Fich, Victor Luchangco, Mark Moir, and Nir Shavit. Obstruction-free step complexity: Lock-free dcas as an example. In *DISC*, pages 493–494, 2005.

[16] Matteo Frigo, Charles E. Leiserson, and Keith H. Randall. The implementation of the cilk-5 multithreaded language. In *Proceedings of the ACM SIGPLAN 1998 Conference on Programming Language Design and Implementation, PLDI*, pages 212–223, June 1998.

[17] James R. Goodman. Cache consistency and sequential consistency. Technical report, 1989. Technical report 61.

[18] Gary Graunke and Shreekant S. Thakkar. Synchronization algorithms for shared-memory multiprocessors. *IEEE Computer*, 23(6):60–69, 1990.

[19] Danny Hendler, Yossi Lev, Mark Moir, and Nir Shavit. A dynamic-sized nonblocking work stealing deque. *Distributed Computing*, 18(3):189–207, 2006.

[20] Danny Hendler and Nir Shavit. Non-blocking steal-half work queues. In *Proceedings of the Twenty-First Annual ACM Symposium on Principles of Distributed Computing*, pages 280–289, July 2002.

[21] Maurice Herlihy. Wait-free synchronization. *ACM Trans. Program. Lang. Syst.*, 13(1):124–149, 1991.

[22] Maurice Herlihy and Nir Shavit. *The art of multiprocessor programming*. Morgan Kaufmann, 2008.

[23] Maurice Herlihy and Jeannette Wing. Linearizability: a correctness condition for concurrent objects. *ACM Trans. Program. Lang. Syst.*, 12(3):463–492, 1990.

[24] Lisa Higham and Jalal Kawash. Java: Memory consistency and process coordination. In *DISC*, pages 201–215, 1998.

[25] Lisa Higham and Jalal Kawash. Bounds for mutual exclusion with only processor consistency. In *DISC*, pages 44–58, 2000.

[26] *IBM System/370 Extended Architecture, Principles of Operation*, 1983. Publication No. SA22-7085.

[27] Prasad Jayanti, King Tan, and Sam Toueg. Time and space lower bounds for nonblocking implementations. *SIAM Journal on Computing*, 30(2):438–456, 2000.

[28] Jalal Kawash. *Limitations and Capabilities of Weak Memory Consistency Systems*. PhD thesis, University of Calgary, January 2000.

[29] Michael Kuperstein, Martin Vechev, and Eran Yahav. Automatic inference of memory fences. In *Formal Methods in Computer Aided Design*, 2010.

[30] Leslie Lamport. Specifying concurrent program modules. *ACM Trans. Program. Lang. Syst.*, 5(2):190–222, April 1983.

[31] Leslie Lamport. The mutual exclusion problem: part II - statement and solutions. *J. ACM*, 33(2):327–348, 1986.

[32] Leslie Lamport. A fast mutual exclusion algorithm. *ACM Trans. Comput. Syst.*, 5(1):1–11, 1987.

[33] Jaejin Lee. *Compilation Techniques for Explicitly Parallel Programs*. PhD thesis, Department of Computer Science, University of Illinois at Urbana-Champaign, 1999.

[34] Victor Luchangco, Mark Moir, and Nir Shavit. On the uncontended complexity of consensus. In *Proceedings of the 17th International Conference on Distributed Computing*, pages 45–59, October 2003.

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

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

[37] Maged M. Michael and Michael L. Scott. Simple, fast, and practical non-blocking and blocking concurrent queue algorithms. In *Proceedings of the Fifteenth Annual ACM Symposium on Principles of Distributed Computing*, pages 267–275, May 1996.

[38] Maged M. Michael, Martin T. Vechev, and Vijay Saraswat. Idempotent work stealing. In *Proceedings of the Fourteenth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPoPP*, pages 45–54, February 2009.

[39] Shlomo Moran, Gadi Taubenfeld, and Irit Yadin. Concurrent counting. *Journal of Computer and System Sciences*, 53(1):61–78, August 1996.

[40] Scott Owens, Susmit Sarkar, and Peter Sewell. A better x86 memory model: x86-tso. In *TPHOLs*, pages 391–407, 2009.

[41] Gary L. Peterson. Myths about the mutual exclusion problem. *Inf. Process. Lett.*, 12(3):115–116, 1981.

[42] Vijay A. Saraswat, Radha Jagadeesan, Maged M. Michael, and Christoph von Praun. A theory of memory models. In *Proceedings of the 12th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPOPP*, pages 161–172, March 2007.

[43] Susmit Sarkar, Peter Sewell, Francesco Zappa Nardelli, Scott Owens, Tom Ridge, Thomas Braibant, Magnus O. Myreen, and Jade Alglave. The semantics of x86-cc multiprocessor machine code. In *POPL*, pages 379–391, 2009.

[44] William E. Weihl. Commutativity-based concurrency control for abstract data types. *IEEE Trans. Computers*, 37(12):1488–1505, 1988.

[45] Glynn Winskel. *The Formal Semantics of Programming Languages*. MIT Press, 1993.