Impact of Instruction Re-Ordering on the Correctness of Shared-Memory Programs

Lisa Higham and Jalal Kawash
Department of Computer Science
The University of Calgary, Canada
`higham@cpsc.ucalgary.ca`
Department of Computer Science
American University of Sharjah, UAE
`jkawash@aus.edu`

Abstract

Sequential consistency is an intuitive consistency model that simplifies reasoning about concurrent multiprocessor programs. Most implementations of high-performance multiprocessors, however, utilize mechanisms that allow instructions to execute out of order, resulting in consistency models that are weaker than sequential consistency and further complicating the job of programmers. This paper investigates all possible combinations of re-ordering of read and write instructions and their effects on the correctness of programs that are designed for sequential consistency. It shows that with certain combinations of re-orderings, any program that accesses shared memory through only reads and writes and that is correct assuming sequential consistency, can be transformed to a new program that does not use any explicit synchronization, and that remains correct in spite of the instruction re-ordering. With other combinations of re-ordering, such transformations do not exist, and even solutions to the mutual exclusion problem are impossible without resorting to explicit synchronization.

Keywords: Instruction re-ordering, Sequential consistency, Critical Sections, Memory consistency models, Mutual exclusion, High-performance multiprocessors.

1 Introduction

Designers of concurrent algorithms typically assume sequential consistency, a consistency model that is formalized by Lamport [20]. Sequential consistency requires that memory operations of all processors appear to be "executed in some sequential order, and the operations of each processor appear in this sequence in the order specified by its program" (program order). Sequential consistency is intuitive, but disallows many possible hardware and software optimizations.

Adve and Gharachorloo [2] identify several optimization techniques that cause instructions to be reordered so that they appear to execute out of program order. This is called *instruction re-ordering.* Write buffers with read bypasses, overlapping writes, non-blocking reads, and optimizing compilers can lead to all forms of instruction re-ordering. They also cite many commercial multiprocessors that utilize instruction re-ordering, such as the AlphaServer 8200/8400, Cray T3D/T3E, and SparcCenter 1000/2000 (See Figure 1). Other examples include the Java Virtual Machine (JVM), IBM PowerPC, Intel Itanium, and .Net. Instruction re-ordering aims at improving the system's performance but it relaxes sequential consistency, making the job of programming multiprocessors even harder.

As noted by Lamport [20], independent instructions applied to different (memory) locations cannot be reordered without changing possible outcomes. Therefore, multiprocessor systems that incorporate instruction re-ordering are also equipped with more powerful instructions than reads and writes, such as read-modify-write and memory barrier instructions. These synchronization primitives can be used to enforce orderings on instructions that otherwise might be re-ordered causing incorrect computation. Using these powerful instructions, however, is expensive; excessive use can result in inefficient implementations, possibly defeating the purpose of instruction re-ordering altogether.

Architecture	write-read re-ordering	write-write re-ordering	read-write re-ordering	read-read re-ordering
IBM 370 [2]	■			
SPARC TSO [26, 16]	■			■ [19]
SPARC PSO [26, 16]	■	■	■ [19]	■ [19]
SPARC RMO [26, 12]	■	■	■	■
IBM PowerPC [8]	■	■	■	■
DEC Alpha [9, 12]	■	■	■	■
JVM [22, 14]	■	■	■	■
Intel Itanium[18]	■	■	■	■
.Net [24]	■	■	■	■

Figure 1: Examples of some commercial systems that utilize instruction re-ordering

We assume that multiprocessors are *coherent* meaning that execution order maintains program order of instructions applied to the same memory location. If a read of one location precedes in program order a write to a different location and this read appears after this write in execution order, this is called *read-write re-ordering*. Reordering types *write-read, write-write, and read-read* are defined similarly. Call a shared memory multiprocessor program whose shared memory consists of only atomic locations (that is, variables that support only read and write instructions), a *(shared-memory) multi-program*. This paper answers a fundamental question concerning shared-memory multi-programs and instruction re-ordering:

> Under what conditions is there a general transformation that transforms any shared-memory multi-program that is correct under sequential consistency to another shared-memory multi-program that is still correct in spite of possible instruction re-ordering?

The results of this paper are summarized in Figure 2 (Section 4). Let $\mathcal{A}$ be a shared-memory multi-program that solves a given problem $\mathcal{P}$ under sequential consistency. This paper shows that:

1. For any combination of re-ordering types that excludes read-read re-ordering, there exists a (very simple) transformation, which transforms $\mathcal{A}$ to another shared-memory program that solves $\mathcal{P}$ in spite of the re-ordering. The transformation is general; it is correct for any shared-memory program under any combination of read-write, write-read, and write-write re-orderings.
2. The exclusion of the read-read re-ordering is sufficient but not necessary. For any combination of read-read and write-write re-ordering only, such a transformation still exists.
3. If both read-read and read-write (or both read-read and write-read) re-ordering combinations are possible, there is no shared-memory solution to mutual exclusion without using stronger objects than shared locations.
4. Since mutual exclusion using only shared locations is solvable under sequential consistency, a corollary is that there does not exist a general transformation that uses only shared locations and that guarantees correctness when both read-read and read-write (or both read-read and write-read) re-ordering combinations are possible.

Some of our previous work focused on addressing similar, but less general, questions concerning specific memory consistency models. Previously, we studied the possibility of implementing different coordination patterns on state-of-the-art multiprocessor systems [13, 14, 16]. We also derived necessary and sufficient condition for critical section coordination in a special form of processor consistency [15] and showed that even expected wait-free test-and-set is possible with this consistency model [17].

Other related studies [5, 7, 1, 4, 3, 6, 11, 21] provide programming strategies for high performance multiprocessors most of which rely on the wise usage of synchronization.

Before we target the posed questions in Section 3, Section 2 provides the definitions and a more precise model and other machinery used in this paper. Section 4 summarizes the paper and discusses further research.

2 Model and Definitions

2.1 Multiprocessors, programs and computations

We think of a multiprocessor as a set of processors that asynchronously accesses a bank of shared (memory) locations. Each processor executes a shared-memory program and consequently generates instruction invocations, each of which atomically reads from or atomically writes to one of these locations. The order of these instructions as specified by each processor's program is called *program order.*

For this paper, we consider multiprocessors whose execution of read and write instructions is indistinguishable from an execution in which all the instructions of each of its processors are performed in some total order, but this total order does not necessarily extend program order. We are interested in such machines because some optimization techniques permit instructions to be re-ordered so that they appear to execute out of program order. Typically, such re-orderings are designed so that they do not alter the outcome of a single processor computation. When more than one processors is executing, however, these re-orderings can corrupt the computation [20].

In order to facilitate our proofs in Section 3, we formalize the preceding intuition as follows. Processor p assigns a value ν to location x with the instruction $\mathrm{write}_p(x, \nu)$. The instruction $\mathrm{read}_p(x) = \nu$ returns to processor p the value ν for location x. Both of these instructions are *applied* to x. The subscripts in instructions $\mathrm{write}_p(x, \nu)$ and $\mathrm{read}_p(x) = \nu$ will be omitted when the processor executing the instruction is clear or irrelevant to the discussion. The *invocation component* of the instruction $\mathrm{read}(x) = \nu$ is $\mathrm{read}(x)$ and the *response component* is ν. For a write instruction, the invocation component is the same as the whole instruction and the response is an acknowledgment, which is ignored for this paper. A *processor computation* is any sequence of read and write instructions applied to a collection of shared locations, where the instruction invocations in the sequence arises from the processor's program and are in program order. A *(multiprocessor) computation* of multiprocessor P, is a collection of processor computations, one for each processor $p \in P$. The collection of shared-memory programs of its processors is the multiprocessor's *(shared-memory) multi-program.*

2.2 Consistency Guarantees

According to these definitions of a processor computation and a multiprocessor computation, the responses of read instructions are entirely arbitrary. But in practice we expect the responses of reads to somehow "make sense". This is captured with constraints on the computation which restrict the values that read invocations can return. A sequence of read and write instructions is *valid* if each read in the sequence returns the value of the most recent preceding write to the same location. We will require that for any multiprocessor computation there exists some valid sequence of all the instructions in the computation and this sequence bears some relationship to program order.

The strongest requirement considered in this paper is sequential consistency. A computation *satisfies sequential consistency* if there exists a valid sequence of all its instructions that extends program order.

Sequential consistency is weakened by relaxing the requirement that *all* instructions must appear in program order. We consider all classes of instruction re-ordering, except when the instructions are applied to the same location. (The property of maintaining program order between instructions that are applied to the same location is called *coherence*, and multiprocessor machines are typically coherent [2, 10].) Specifically, let C be a multiprocessor computation and O be all the instructions in C. Let $(O, \xrightarrow{prog})$ denote the program order relation on O. Let $o_1, o_2 \in O$ such that $o_1 \xrightarrow{prog} o_2$, and o_1 and o_2 are applied to different locations. Define the following four *basic* relations on O.

- if o_1 and o_2 are both read instructions then they are *read-read* related, and $(O, \xrightarrow{rr})$ denotes this relation.
- if o_1 is a read instruction and o_2 is a write instruction then they are *read-write* related, and $(O, \xrightarrow{rw})$ denotes this relation.

- if o_1 is a write instruction and o_2 is a read instruction then they are *write-read* related, and $(O, \xrightarrow{wr})$ denotes this relation.
- if o_1 and o_2 are both write instructions then they are *write-write* related, and $(O, \xrightarrow{ww})$ denotes this relation.

Unions of these relations are denoted by $(O, \xrightarrow{\Sigma x_i y_i})$ where $x_i, y_i \in \{r, w\}$. For example, $(O, \xrightarrow{rw+ww})$ abbreviates $(O, \xrightarrow{rw}) \cup (O, \xrightarrow{ww})$. We are primarily concerned with the 16 relations induced by removing combinations of these basic relations from the program order relation. We denote the transitive closure of the relation $(O, \xrightarrow{prog}) \backslash (O, \xrightarrow{\Sigma x_i y_i})$ by $(O, \xrightarrow{\overline{\Sigma x_i y_i}})$. For example, $(O, \xrightarrow{\overline{rw+rr}})$ is the transitive closure of the relation formed by removing from program order all pairs that are applied to different locations where the first instruction in the pair is a read.

Each of the basic relations are partial orders, but combinations might not be. For example, $(O, \xrightarrow{rw+wr})$ is not transitive. Since program order is antisymmetric, any subset of program order is antisymmetric, and thus the transitive closure of $(O, \xrightarrow{prog}) \backslash (O, \xrightarrow{\Sigma x_i y_i})$ is antisymmetric and transitive. Hence, each possible relation $(O, \xrightarrow{\overline{\Sigma x_i y_i}})$ is a partial order. Given a total order $(O, \xrightarrow{T})$ on a finite set O, there is exactly one sequence S satisfying: a precedes b in S if and only if $(a, b) \in (O, \xrightarrow{T})$. So we sometimes abuse terminology and use interchangeably a total order and the sequence that *realizes* that order.

Definition 2.1 *Let O be all the instructions of a computation C of a multiprocessor. Then, C* satisfies consistency model $M_{\overline{\Sigma x_i y_i}}$ *if there exists a valid total order $(O, \xrightarrow{L})$ such that $(O, \xrightarrow{\overline{\Sigma x_i y_i}}) \subseteq (O, \xrightarrow{L})$. A valid sequence of the instructions O that realizes any such total order is called a $M_{\overline{\Sigma x_i y_i}}$*-verifying sequence *for C.*

Observe that $(O, \xrightarrow{\overline{\emptyset}}) = (O, \xrightarrow{prog})$ so $M_{\overline{\emptyset}}$ is sequential consistency.

A multiprocessor *guarantees* a given consistency model if all its possible computations satisfy that consistency model. A shared-memory multi-program, P, together with its consistency model, CM, is called a *multiprocessor system*, and is denoted by (P, CM). For example, the computations of the system $(P, M_{\overline{rr+ww}})$ consists of each of the computations of the multi-program P for which there is a valid sequence of the computation's instructions that extends the partial order $(O, \xrightarrow{\overline{rr+ww}})$.

A consistency model $M_{\overline{\Sigma x_i y_i}}$ is *stronger* than $M_{\overline{\Sigma x_j y_j}}$ if $(O, \xrightarrow{\overline{\Sigma x_i y_i}}) \subseteq (O, \xrightarrow{\overline{\Sigma x_j y_j}})$ and $M_{\overline{\Sigma x_i y_i}}$ is *strictly stronger* than $M_{\overline{\Sigma x_j y_j}}$ if $(O, \xrightarrow{\overline{\Sigma x_i y_i}}) \subset (O, \xrightarrow{\overline{\Sigma x_j y_j}})$. The terms *weaker* and *strictly weaker* are defined similarly.

2.3 Tools and definitions for results

Consider a shared-memory multi-program P that is correct assuming sequential consistency. That is, any computation of the system (P, sequential consistency) meets a given specification $\mathcal{S}$. Our objective is to answer the question: "For what instantiations of $\Sigma x_i y_i$ is it possible to transform P to a new multi-program, $\tau(P)$, that still uses only shared locations such that any computation of $\tau(P)$ satisfying the weaker consistency model $M_{\overline{\Sigma x_i y_i}}$ meets the same specification $\mathcal{S}$?"

Call a transformation of a shared-memory multi-program that only inserts additional read and write instructions and otherwise leaves the original multi-program intact a *simple transformation*. Let P be a multi-program and τ be a simple transformation. A computation of P and a computation of $\tau(P)$ *correspond* if every read in P returns the same value in both computations. A transformed system $(\tau(P), MC')$ *implements* a system (P, MC) if every computation of $(\tau(P), MC')$ corresponds to some computation of (P, MC). Clearly, if every computation of $(\tau(P), MC')$ corresponds to one of (P, MC), and all computations of (P, MC) satisfy a given specification, then every computation of $(\tau(P), MC')$ also satisfies the specification. Thus our question becomes, "For what instantiations of $\Sigma x_i y_i$, is there a simple transformation τ such that for any shared-memory multi-program P, the system $(\tau(P), M_{\overline{\Sigma x_i y_i}})$ implements the system (P, sequential consistency)."

2.4 The critical section problem

The mutual exclusion problem is the most famous and well-studied problem in concurrency. Following Silberschatz *et. al.* [25], we refer to this problem as the critical section problem to distinguish the *problem* from the mutual exclusion *property.* In the critical section problem, a set of processors coordinate to share a resource, while ensuring that no two access the resource concurrently. Solutions to the critical section problem for memories that satisfy sequential consistency have been known since the 1960s; Raynal [23] provides an extensive survey.

Some of our results concern the possibility or impossibility of a shared-memory solution to the critical section problem under the weak consistency models introduced earlier. Each critical section processor p has the structure:

```
repeat
    ⟨remainder⟩
    ⟨enter⟩
    ⟨critical section⟩
    ⟨exit⟩
until done
```

The *critical section problem* is to provide code for ⟨enter⟩ and ⟨exit⟩ so that:

mutual exclusion: For any processor executing its ⟨critical section⟩ code, the outcome is indistinguishable from that processor executing in isolation.

progress: If any processor is executing ⟨enter⟩, then eventually some processor will execute ⟨critical section⟩.

Typically a fairness property is also required, but will not be needed in this paper.

3 Effects of Instruction Re-Ordering

3.1 Overview

Our proofs take two forms. For the re-ordering cases when an implementation exists, we provide a general purpose simple transformation and prove that this transformation implements any sequentially consistent shared-memory system. For the re-ordering cases when an implementation does not exist, we establish this by proving that under these possible re-orderings there is no solution to mutual exclusion that does not exploit something stronger than reads and writes. It follows that there is no general purpose implementation in this case since mutual exclusion does have such a solution under sequential consistency.

3.2 Possibilities

Define transformation $\tau(P)$ on a multi-program P as follows. Each write instruction, write$(x, \cdot)$, in P is replaced by the sequence of instructions (read(x)); write$(x, \cdot)$; read(x)) in $\tau(P)$. That is, each write instruction is replaced by itself, immediately preceded and followed by a read invocation to the same location. Read invocations are left unchanged.

Theorem 3.1 *For any shared-memory multi-program P, the system $(\tau(P), M_{\overline{rw+wr+ww}})$ implements the system (P, sequential consistency).*

Proof: It suffices to show that any computation C of the system $(\tau(P), M_{\overline{rw+wr+ww}})$ corresponds to some computation of the system (P, sequential consistency). Define C' to be the computation C with every read inserted by τ removed. Then C' is clearly a computation of P. We will show that C' is sequentially consistent. Let S be a $M_{\overline{rw+wr+ww}}$-verifying sequence for C. Define S' to be the sequence S with every read inserted by τ removed. First, S' is clearly valid because S is. Second, we show that S' preserves program order. Let $o_1 \xrightarrow{prog} o_2$ in computation C'. If both o_1 and o_2 are reads, then o_1 precedes o_2 in S because S extends the read-read order. If o_1 is a read and o_2 is a write, τ inserts a read, r, before o_2 to the same location as o_2 writes. Therefore, in S, o_1 precedes r because S extends read-read order, and r precedes o_2 because $M_{\overline{rw+wr+ww}}$ is stronger than coherence. A similar argument holds if o_1 is a write and o_2 is a read. If both o_1 and o_2 are writes, then τ inserts a read, r_1, after o_1 to the same location as o_1 writes and another read, r_2, before o_2 to the same location as o_2 writes. Therefore, again because S extends read-read order and coherence, in S, o_1 precedes r_1 which precedes r_2 which precedes o_2. Thus in all cases o_1 precedes o_2 in S and hence in S'. Thus S' is a sequentially consistent-verifying sequence for C' implying computation C of $(\tau(P), M_{\overline{rw+wr+ww}})$ corresponds to computation C' of (P, sequential consistency). ■

By Theorem 3.1, transformation τ will map any sequentially consistent program to a new program that meets the same specification for any consistency guarantee that is at least as strong as $M_{\overline{rw+wr+ww}}$. Of course, if the target consistency guarantee is strictly stronger than $M_{\overline{rw+wr+ww}}$, τ may add excessive additional read instructions. However, the same idea can be specialized to any such consistency model CM as follows. Say that a write instruction invocation, $w = \text{write}(x, \cdot)$, is *left (respectively, right) vulnerable* if there exists an instruction invocation o_1 (respectively, o_2) such that

1. $o_1 \xrightarrow{prog} w$ (respectively, $w \xrightarrow{prog} o_2$), and
2. CM does not guarantee that program order is maintained between w and o_1 (respectively, o_2), and
3. there does not exist a read(x), where $o_1 \xrightarrow{prog} \text{read}(x) \xrightarrow{prog} w$ (respectively, $w \xrightarrow{prog} \text{read}(x) \xrightarrow{prog} o_2$).

If w is left (respectively, right) vulnerable then the invocation w is transformed to $(\text{read}(x); w)$ (respectively, $(w, \text{read}(x))$). That is, w is replaced by itself, immediately preceded (respectively, followed) by a read invocation to the same location.

Using an argument essentially the same as that used in the proof of Theorem 3.1, such a transformation is easily seen to preserve program order for all the instructions in the original program, and will produce only computations that correspond to the sequentially consistent ones of the original program. Furthermore, observe that any write instruction that is vulnerable to re-ordering and is not protected with the additional read(s) invocations, could be re-ordered in the execution. Consequently the resulting computation could not, in general, be guaranteed to correspond to a sequentially consistent one. Hence we conclude that this implementation will produce the minimum number of additional instructions among all simple transformations to consistency model $M_{\overline{rw+wr+ww}}$ that are guaranteed correct for *all* shared-memory multi-programs.

Theorem 3.1 establishes that as long as read-read order is maintained, any multi-program can be transformed to tolerate any loss of any combination of the other three basic orders. The next theorem shows that the read-read re-ordering can also be tolerated provided it is not combined with either read-write or write-read re-ordering.

For any multi-program P, define transformation $\gamma(P)$ as follows. Let v be a new shared location not used by P.

- For every write instruction w such that the instruction immediately following w in program order is another write to a different location, w is transformed in $\gamma(P)$ to the sequence $(w; \text{read}(v))$. That is, each such write instruction is replaced by itself immediately followed by a read invocation of the new location v. Other write instructions are not altered.
- For every read instruction r such that the instruction immediately following r in program order is another read of a different location, r is transformed in $\gamma(P)$ to the sequence $(r; \text{write}(v, 0))$. That is, each such read instruction is replaced by itself, immediately followed by a write invocation of the new location v. Other read instructions are not altered.

Theorem 3.2 *For any shared-memory multi-program P, the system $(\gamma(P), M_{\overline{rr+ww}})$ implements the system (P, sequential consistency).*

Proof: The proof strategy is the same as that of Theorem 3.1. It suffices to show that any computation C of the system $(\gamma(P), M_{\overline{rr+ww}})$ corresponds to some computation of the system (P, sequential consistency). Define C' to be the computation C with every read and write to v removed. Then C' is clearly a computation of P. We will show that C' is sequentially consistent. Let S be a $M_{\overline{rr+ww}}$-verifying sequence for C. Define S' to be the sequence S with every read and write to v removed. First, S' is clearly valid because S is. Second, we show that S' preserves program order. Let $o_1 \xrightarrow{prog} o_2$ in computation C'. If one of o_1 and o_2 is a read and the other is a write, or they are both applied to the same location, then o_1 precedes o_2 in S because S extends read-write and write-read orders and coherence. Otherwise they are both of the same instruction type (read or write) applied to different locations. In this case, either there is an instruction of the opposite type between them in program order or one is inserted by γ. So, again, program order in S is enforced because S extends both read-write and write-read orders. Thus in all cases o_1 precedes o_2 in S and hence in S'. Thus S' is a sequentially consistent-verifying sequence for C'. This implies computation C of $(\gamma(P), M_{\overline{rr+ww}})$ corresponds to computation C' of $(P, \text{sequential consistency})$. ■

As was the case for τ, transformation γ could introduce excessive instructions if the consistency guarantee is strictly stronger than $M_{\overline{rr+ww}}$. But it is easily adapted so that only the necessary read and write instruction to the new location are added. For example, if the consistency model is stronger than $M_{\overline{rr}}$, all the read instructions introduced by γ are unnecessary.

Recall that the critical section problem has several well known solutions in sequentially consistent systems. Therefore, the following is an immediate corollary of Theorems 3.1 and 3.2.

Corollary 3.3 *Critical sections can be implemented using only read and write instructions to shared locations provided the consistency guarantee is at least as strong as either* $M_{\overline{rw+wr+ww}}$ *or* $M_{\overline{rr+ww}}$.

3.3 Impossibilities

Theorem 3.4 *There is no shared-memory solution to the critical section problem for any number of processors greater than or equal to 2 when the consistency model is weaker than or equal to* $M_{\overline{rr+wr}}$ *or weaker than or equal to* $M_{\overline{rr+rw}}$.

Proof: Assume for the sake of contradiction that a solution to the critical section problem exists under either consistency model $M_{\overline{rr+wr}}$ or $M_{\overline{rr+rw}}$. Consider any $\langle$critical section$\rangle$ code that contains at least one shared location *share*, and two processors, p and q, such that:

- p and q write different values, say ν_p and ν_q respectively, to *share* while in $\langle$critical section$\rangle$, and
- $\langle$critical section$\rangle$ is finite, and
- *share* is only used in $\langle$critical section$\rangle$, and
- each of p and q read *share* immediately after its first write to it in $\langle$critical section$\rangle$.

Clearly, the first three items hold for most critical sections since they are intended to prevent interference of one processor by another while writing critical data. The last assumption can be made without loss of generality because any critical section satisfying the first three items can be augmented with these reads without effecting the semantics of the critical section.

Consider computations of the system when processors p and q are participating and all remaining processors are in $\langle$remainder$\rangle$. If processor p is executing its $\langle$enter$\rangle$ while q also stays in $\langle$remainder$\rangle$, then by the progress property, p must subsequently execute $\langle$critical section$\rangle$, and then execute $\langle$exit$\rangle$, because $\langle$critical section$\rangle$ is finite. Let Computation 1 be a computation that is produced in this case. The sequence $o_1^p, o_2^p, ..., o_k^p$ is the sequence of read and write instructions produced by p's program from the beginning of $\langle$enter$\rangle$ up to the instruction preceding the first write to *share* in p's $\langle$critical section$\rangle$; $o_{k+3}^p, ..., o_j^p$ is the sequence of instructions produced from the rest of p's $\langle$critical section$\rangle$ following the read of *share* that immediately follows p's first write to *share* up to the end of $\langle$exit$\rangle$. Observe that read(*share*) must have response ν_p.

$$\textbf{Computation 1} \left\{ \begin{array}{ll} p: & o_1^p, o_2^p, ..., o_k^p; \text{write}(\textit{share}, \nu_p); \text{read}(\textit{share}) = \nu_p; o_{k+3}^p, ..., o_j^p \\ q: & \langle\text{remainder}\rangle \end{array} \right.$$

Similarly, if processor q is executing $\langle$enter$\rangle$ while p stays in $\langle$remainder$\rangle$, Computation 2 is produced, where read(*share*) has response ν_q. The program for $\langle$enter$\rangle$ up to the instruction preceding the first write to *share* in q's $\langle$critical section$\rangle$ produces the sequence of instructions $o_1^q, o_2^q, ..., o_l^q$; the program from after the read of *share* to the end of $\langle$exit$\rangle$ produces the sequence of instructions $o_{l+3}^q, ..., o_m^q$.

$$\textbf{Computation 2} \left\{ \begin{array}{ll} p: & \langle\text{remainder}\rangle \\ q: & o_1^q, o_2^q, ..., o_l^q; \text{write}(\textit{share}, \nu_q); \text{read}(\textit{share}) = \nu_q; o_{l+3}^q, ..., o_m^q \end{array} \right.$$

Case 1, $M_{\overline{rr+wr}}$: Consider Computation 3 where both p and q are participating but where ν_q is returned for the value of *share* to both processors. The sequences after read(*share*) $= \nu_q$ are elided because they are not needed for this case.

$$\textbf{Computation 3} \begin{cases} p: & o_1^p, o_2^p, ..., o_k^p; \text{write}(\textit{share}, \nu_p); \text{read}(\textit{share}) = \nu_q; ... \\ q: & o_1^q, o_2^q, ..., o_l^q; \text{write}(\textit{share}, \nu_q); \text{read}(\textit{share}) = \nu_q; ... \end{cases}$$

Clearly, this is not a correct computation of a critical section program because it violates the mutual exclusion property. To achieve the desired contradiction, we prove that Computation 3 satisfies $M_{\overline{rr+wr}}$, given that Computations 1 and 2 do.

Since the Computations 1 and 2 satisfy $M_{\overline{rr+wr}}$, each has a $M_{\overline{rr+wr}}$-verifying sequence. Let $S_p = o_{\alpha_1}^p, \cdots o_{\alpha_k}^p, \text{write}(\textit{share}, \nu_p)$ be a verifying sequence for Computation 1, up to but not including p's read of *share*. Since S_p extends read-write and write-write order the write to *share* must occur at the end of the sequence. Similarly, let $S_q = o_{\beta_1}^q, \cdots o_{\beta_l}^q, \text{write}(\textit{share}, \nu_q)$ be a $M_{\overline{rr+wr}}$-verifying sequence for Computation 2 also excluding everything after q's write of *share*.

Build a sequence of instructions S as follows. Initially, S is empty. Examine each o_i^q in S_q in order from $i = \beta_1$ to β_l. If o_i^q is a read that is not preceded by a write to the same location, append o_i^q to S and remove it from S_q. When there are no such reads left in S_q, append S_p to S. Finally, append the updated S_q (with the above reads removed) to S, then append the two reads of *share* to S.

S can be partitioned into four segments. The first consists entirely of reads by q returning initial values, the second consists entirely of S_p, the third consists entirely of S_q minus instructions in the first segment, and the fourth segment consists of the reads of *share* in any order.

To see that S is valid, note that the reads in the first segment necessarily return initial values because in Computation 2 only q is participating. So, the first segment is valid. The second segment is valid because it is S_p and the first segment does not contain any writes. The construction guarantees that each read left in the third segment returns the value written by the most recent write to the same location that precedes the read in the same segment. The reads in the fourth section return ν_q and this is the most recent write to *share*, by construction. Therefore, S is valid.

Let O be all the instructions in Computation 3. $(O, \xrightarrow{\overline{rr+wr}})$ is maintained in S_p and in S_q. The splitting of S_q only moves forward reads that are not preceded in S_q by any writes to the same location. So the moved reads do not violate $(O, \xrightarrow{\overline{rr+wr}})$. Therefore, S preserves $(O, \xrightarrow{\overline{rr+wr}})$. So S is a $M_{\overline{rr+wr}}$-verifying sequence for Computation 3.

Case 2, $M_{\overline{rr+rw}}$: Assume that Computations 1 and 2 satisfy $M_{\overline{rr+rw}}$. Consider a computation, Computation 4, where both p and q are participating but where ν_q is returned for the value of *share* to both processors. In this case, we will make use of ⟨exit⟩.

$$\textbf{Computation 4} \begin{cases} p: & o_1^p, o_2^p, ..., o_k^p; \text{write}(\textit{share}, \nu_p); \text{read}(\textit{share}) = \nu_q; o_{k+3}^p, ..., o_j^p \\ q: & o_1^q, o_2^q, ..., o_l^q; \text{write}(\textit{share}, \nu_q); \text{read}(\textit{share}) = \nu_q; o_{l+3}^q, ..., o_m^q \end{cases}$$

Again, this is not a correct computation of a critical section program because it violates the mutual exclusion property. As before, we prove that Computation 4 satisfies $M_{\overline{rr+wr}}$, given that Computations 1 and 2 do.

Let O be all the instructions in Computation 4 and let S_p and S_q be $M_{\overline{rr+rw}}$-verifying sequences respectively for Computations 1 and 2. Let S_p^- be S_p minus $\text{read}_p(\textit{share}) = \nu_p$ and let S_q^+ be S_q plus $\text{read}_p(\textit{share}) = \nu_q$ inserted immediately after $\text{write}(\textit{share}, \nu_q)$.

single type		two combinations		three combinations	
read-read	√	read-read, write-write	√	read-read, write-write, write-read	×
write-write	√	read-read, read-write	×	read-read, write-write, read-write	×
read-write	√	read-read, write-read	×	read-read, write-read, read-write	×
write-read	√	read-write, write-read	√	write-write, read-write, write-read	√
		write-write, write-read	√		
		write-write, read-write	√		

Figure 2: Summary of results

Define $S = S_p^-; S_q^+$. Sequence S is valid: *share* is only written in ⟨critical section⟩ implying the moved read(*share*) $= \nu_q$ by p is valid, and the rest is obviously valid. S corresponds to the scenario where p completes one cycle (⟨enter⟩, ⟨critical section⟩, and ⟨exit⟩) in isolation; then, q starts and completes a similar cycle. After p completes ⟨exit⟩, the state for q must be indistinguishable from Computation 2 where it ran in isolation. Hence S, is valid.

All the instructions, except read(*share*) $= \nu_q$ by p, occur in their original orders in the verifying sequences. Moving this read to the end is permitted by $(O, \xrightarrow{\overline{rr+rw}})$. Thus, Computation 4 satisfies $M_{\overline{rr+rw}}$.

In both cases, our assumption of a solution leads to a contradiction. Thus, there is no shared-memory solution to the critical section problem under either consistency model $M_{\overline{rr+wr}}$ or $M_{\overline{rr+rw}}$.

■

None of the arguments in Theorem 3.4 depended on fairness, nor on the size of the locations; so the impossibilities apply to unfair solutions and to locations of unbounded size.

4 Conclusion

This paper investigated the effect of instruction re-ordering on programs that are correct when no instructions are re-ordered. The paper conclusions are summarized in Figure 2. The possibilities (represented by √) in Figure 2 indicate the existence of a general transformation for any sequentially consistent program to a program that is still correct in spite of the indicated instruction re-ordering combination.

The simple implementations provided in this paper are general. They are also optimal for general transformations — these that apply to *any* multi-program that is correct for sequential consistency. However, optimality for general transformations does not necessarily imply optimality for individual multi-program instances. When given a *fixed* instance, it may be possible to apply further optimizations that exploit information from the given multi-program and the problem it solves. Such information (from both programs and problems) is unavailable to general transformers.

The impossibilities of Figure 2 (represented by ×) indicate that there is no shared-memory implementation of the critical section problem with the indicated combinations of instruction re-orderings. Since shared-memory solutions to the critical section problem exist for sequential consistency, general transformations that do not augment the specified program with explicit synchronization operations do not necessarily generate correct transformed programs. Hence in these cases, synchronization instructions must be inserted to the transformed program. Our results imply that the IBM PowerPC, DEC Alpha, JVM, and SPARC TSO, PSO, and RMO (Figure 1) require the use of explicit synchronization to correctly implement mutual exclusion. Hence, one of our future research directions is to augment the target program with memory barrier instructions and to minimize the number of such instructions.

References

[1] S. Adve. Using information from the programmer to implement system optimizations without violating sequential consistency. Technical Report ECE 9603, Department of Electrical and Computer Engineering, Rice University, March 1996.

[2] S. Adve and K. Gharachorloo. Shared memory consistency models: A tutorial. *IEEE Computer*, pages 66–76, December 1996.

[3] S. Adve, K. Gharachorloo, A. Gupta, J. Hennessy, and M. Hill. Sufficient system requirements for supporting the PL_{pc} memory model. Technical Report 1200 (or CSL-TR-93-595), University of Wisconsin-Madison (or Stanford University), 1993.

[4] S. Adve and M. Hill. Sufficient conditions for implementing data-race-free-1 memory model. Technical Report 1107, University of Wisconsin-Madison, 1992.

[5] M. Ahamad, G. Neiger, J. Burns, P. Kohli, and P. Hutto. Causal memory: Definitions, implementations, and programming. *Distributed Computing*, 9:37–49, 1995.

[6] H. Attiya, S. Chaudhuri, R. Friedman, and J. Welch. Shared memory consistency conditions for non-sequential execution: Definitions and programming strategies. *SIAM Journal of Computing*, 27(1):65–89, February 1998.

[7] H. Attiya and R. Friedman. Programming DEC-Alpha based multiprocessors the easy way. In *Proc. 6th Int'l Symp. on Parallel Algorithms and Architectures*, pages 157–166, 1994. Technical Report LPCR 9411, Computer Science Department, Technion.

[8] F. Corella, J. Stone, and C. Barton. A formal specification of the PowerPC shared memory architecture. Technical Report RC18638, IBM, 1994.

[9] C. C. Corportaion. *The Alpha Architecture Handbook.* Compaq Computer Corporation, 1998. Order number: EC-QD2KC-TE.

[10] M. Frigo. The weakest reasonable memory model. Master's thesis, Department of Electrical Engineering and Computer Science, MIT, 1998.

[11] K. Gharachorloo, S. Adve, A. Gupta, J. Hennessy, and M. Hill. Programming for different memory consistency models. *Journal of Parallel and Distributed Computing*, 15(4):399–407, August 1992.

[12] L. Higham, L. Jackson, and J. Kawash. Specifying memory consistency of write buffer multiprocessors. Technical Report 2004-758-23, Department of Computer Science, The University of Calgary, August 2004. Submitted for publication.

[13] L. Higham and J. Kawash. Critical sections and producer/consumer queues in weak memory systems. In *Proc. 1997 IEEE Int'l Symp. on Parallel Architectures, Algorithms, and Networks*, pages 56–63, December 1997.

[14] L. Higham and J. Kawash. Java: Memory consistency and process coordination (extended abstract). In *Proc. 12th Int'l Symp. on Distributed Computing, Lecture Notes in Computer Science volume 1499*, pages 201–215, September 1998.

[15] L. Higham and J. Kawash. Bounds for mutual exclusion with only processor consistency. In *Proc. of the 14th Int'l Conf. on Distributed Computing, Lecture Notes in Computer Science volume 1914*, pages 44–58, October 2000.

[16] L. Higham and J. Kawash. Memory consistency and process coordination for SPARC multiprocessors. In *Proc. of the 7th Int'l Conf. on High Performance Computing, Lecture Notes in Computer Science volume 1970*, pages 355–366, December 2000.

[17] L. Higham and J. Kawash. Implementing sequentially consistent programs on processor consistent platforms. In *Proc. 2004 Int'l IEEE Symp. on Parallel Architectures, Algorithms, and Networks*, pages 56–63, May 2004.

[18] Intel Corporation. Intel Itanium architecture software developers manual, volumes 1-3. 2002.

[19] J. Kawash. *Limitations and Capabilities of Weak Memory Consistency Systems.* Ph.D. dissertation, Department of Computer Science, The University of Calgary, January 2000.

[20] L. Lamport. How to make a multiprocessor computer that correctly executes multiprocess programs. *IEEE Trans. on Computers*, C-28(9):690–691, September 1979.

[21] L. Lamport. How to make a correct multiprocess program execute correctly on a multiprocessor. *IEEE Trans. on Computers*, 46(7):779–782, July 1997.

[22] T. Lindholm and F. Yellin. *The Java Virtual Machine Specification.* Addison-Wesley, 1997.

[23] M. Raynal. *Algorithms for Mutual Exclusion.* The MIT Press, 1986.

[24] A. D. Robison. Memory consistency and .Net. *Dr. Dobb's Journal*, pages 46–50, April 2003.

[25] A. Silberschatz and P. Galvin. *Operating System Concepts.* John Wiley & Sons, Inc., 1999.

[26] D. Weaver and T. Germond, editors. *The SPARC Architecture Manual version 9.* Prentice-Hall, 1994.