Reasoning about the Implementation of Concurrency Abstractions on x86-TSO

Scott Owens

University of Cambridge

Abstract. With the rise of multi-core processors, shared-memory concurrency has become a widespread feature of computation, from hardware, to operating systems, to programming languages such as C++ and Java. However, none of these provide sequentially consistent shared memory; instead they have relaxed memory models, which make concurrent programs even more challenging to understand. Programming language implementations run on hardware memory models, so VM and run-time system implementors must reason at both levels. Of particular interest are the low-level implementations of the abstractions that support language-level concurrency—especially because they invariably contain data races.

In this paper, we develop a novel principle for reasoning about assembly programs on our previous x86-TSO memory model, and we use it to analyze five concurrency abstraction implementations: two spinlocks (from Linux); a non-blocking write protocol; the double-checked locking idiom; and `java.util.concurrent`'s Parker. Our principle, called *triangular-race freedom*, strengthens the usual data-race freedom style of reasoning.

1 Introduction

Most techniques for reasoning about concurrent shared-memory programs assume *sequential consistency (SC)*: accesses by multiple threads to the shared memory occur in a global-time linear order that corresponds to an interleaving of their constituent statements [21]. Real multi-core and multiprocessor systems, however, incorporate performance optimizations that have observable non-SC consequences for the behavior of multithreaded concurrent programs. These processors are said to have *relaxed* memory models [1].

Figure 1 provides an example of relaxed memory behavior on modern Intel or AMD x86 processors: given two memory locations x and y (initially holding 0), if two threads labeled p and q respectively write 1 to x and y and then read from y and x, it is possible for both to read 0 *in the same execution*. It is easy to check that this result cannot arise from any interleaving of the two threads.

Programming languages, notably C++ and Java, also have relaxed memory models [16, 18] made necessary both by the relaxed memory models of the underlying hardware and by visible compiler optimizations. Such memory models are difficult to specify and subtle to implement, as evidenced by the flaws that

Initial: $[x] = 0 \wedge [y] = 0 \wedge x \neq y$	
p	q
1a: mov $[x]$←1	1c: mov $[y]$←1
1b: mov eax←$[y]$ (0)	1d: mov ebx←$[x]$ (0)
Allow: eax $= 0 \wedge$ ebx $= 0$	

Syntax: Throughout the paper we use Intel syntax for x86 assembly programs in a san serif font. For emphasis, we separate operands with "←" instead of "," when the left operand is assigned to. The eax, ebx, etc. are register names, and $[x]$ denotes the contents of memory location x, where x is a meta-variable. Thus, mov $[y]$←1 stores the value 1 to the memory location y, and mov eax←$[y]$ reads the value at memory location y, and stores it in register eax. We use the labels (1a: etc.) to identify instructions in the text. We sometimes indicate the result of a memory read in parentheses to the right of the instruction, when we are interested in a particular execution.

Fig. 1. An x86 program that exhibits relaxed behavior

have been discovered in Java's memory model from time to time [29, 35], but they are critical to supporting robust and efficient programs.

The peculiarities of a particular hardware memory model directly affect the implementation of language memory models. For example, Dice has observed that, at one point, Intel and AMD revised their x86 manuals to apparently allow some relaxed behaviors (the so called "independent reads of independent writes") that were previously thought to be forbidden [11]. Such behavior on the hardware would lead directly to Java-language-level behavior that violates Java's memory model. Solutions either required revising Java's memory model to allow the new behavior, or required the compiler to insert additional x86 synchronization primitives, harming performance. (Recent Intel and AMD manuals more explicitly forbid the behavior in question, and the actual processors did not allow it anyway.) Thus, a solid understanding of hardware memory models, both what they are and how to reason about them, is necessary for implementing concurrent languages, and for verifying such implementations: their compiler, their VM/runtime, and their concurrency libraries.

In this paper, we develop a principle, that we call *triangular-race freedom (TRF)*, for reasoning about programs run on relaxed memory models that are similar to SPARC's total store ordering (TSO) [34], including our x86-TSO memory model (see Sect. 3). We apply TRF to five example idioms that are typical of the low-level concurrency abstractions used to build support for higher-level memory models in language runtime systems, virtual machines, and operating systems:

- a simple spinlock, taken from the Linux kernel (Sects. 2 and 7),
- a ticketed spinlock, based on a newer Linux kernel (Sect. 7),
- a non-blocking write protocol [19], also known as "SeqLocks" [20] (Sect. 8),
- the double-checked locking idiom [32] (Sect. 8), and

– the implementation of blocking synchronization in the HotSpot JVM (the parker for `java.util.concurrent.locks.LockSupport`) [12] (Sect. 9).

Our TRF principle relies on particular features of TSO architectures to strengthen existing *data-race freedom (DRF)* [2] principles. (We explain DRF in Sections 2 and 5.) Thus, it applies to a wider class of programs that can contain data-races, including the idioms above. TRF also precisely characterizes the programs whose shared memory accesses are the same on TSO and SC memory models.

We first (Sect. 2) explain the intuition behind TRF and informally explain how it applies to our first spinlock example before moving on to the mathematical development of TRF, which begins with a presentation of x86-TSO, our model of the x86 architecture [27] (Sect. 3).

In summary, our contributions are:

– a DRF principle for x86-TSO (Sect. 5);
– our TRF principle, which precisely characterizes the programs with identical x86-TSO and sequentially consistent shared memory accesses (Sect. 6); and
– the application of TRF-based reasoning to the five examples mentioned above.

A sketch of the proof of our main theorem is in the appendix; for a full proof see `http://www.cl.cam.ac.uk/~so294/ecoop2010/`.

2 Using Triangular-race Freedom, Informally

Intuitively, a data race occurs whenever two threads access the same memory address, one of them is writing, and the two accesses are not separated by some synchronization operation (we refine this and make it precise in Sect. 5). For example, in Fig. 1 there are data races on both x and y. If none of a program's executions can encounter a data race, then the program is data-race free. One common idiom for ensuring data-race freedom is to put all shared memory accesses in critical sections.

Based on the observation that most programs should be DRF, relaxed memory models are often designed to guarantee that DRF programs have no observable, non-sequentially consistent behaviors. Saraswat et al. [30] call this the "fundamental property" of a memory model, and it has been proved to hold for a variety of relaxed models for both hardware and languages [2, 3, 5, 7, 14, 23]. For DRF programs on such models, existing verification technology, such as model checking [36] and concurrent separation logic [8, 26], can be applied soundly, and informal reasoning can be based entirely on sequentially consistent interleaving.

Crucially, the definition of data-race freedom uses a sequentially consistent memory model, so the relaxed model is not required in order to establish that a program is DRF. One need only check that no SC execution has a race to get a guarantee that any relaxed execution is equivalent to some SC execution. Furthermore, when proving the absence of races, one can rely on any facts established by existing SC-based reasoning techniques.

```
; The address of spinlock, x, is stored in register eax, and
; the value of the spinlock ([x]) is 1 iff it is unlocked.
acquire:  lock dec [eax]   ; atomic (tmp := [x] − 1
                           ;          [x] := tmp
                           ;          flag := tmp ≥ 0
                           ;          flush local write buffer)
          jns enter        ; if flag then goto enter
spin:     cmp [eax],0      ; flag := [x] ≤ 0
          jle spin         ; if flag then goto spin
          jmp acquire      ; goto acquire
enter:                     ; the critical section starts here
release:  mov [eax]←1      ; [x] := 1
```

Fig. 2. An x86 spinlock from Linux v2.6.24.7 (pseudocode to the right of ;)

Informally, we define a triangular race to be a data race between a read and write operation (we can exclude write-write data races) where the read operation is preceded by another write operation on the same thread, and there are no intervening hardware synchronization primitives (such as a barrier or atomic compare and exchange instruction). We prove that a program with no triangular races has only SC observable behaviors when run on a TSO memory model. Just as for data-race freedom, we need to check for triangular races only on an SC semantics for the program.

2.1 Application to a Spinlock

User-level programs are often intended to be *well-synchronized*: accesses to shared memory locations are protected by a mutual exclusion mechanism. However, the x86, like most processors, does not directly provide any mutual exclusion primitives; instead they must be implemented as part of the program (e.g., in a library of lock operations). Their implementations often contain data races, rendering a traditional DRF principle inapplicable. Here, we present a mutual exclusion implementation taken from Linux, and show how TRF-based reasoning can be applied to it. (Section 7 goes through this reasoning more formally.)

The Linux kernel, version 2.6.24.7, implements basic mutual exclusion with a spinlock (Fig. 2[1]). The spinlock is represented by a signed integer which is 1 if the lock is free and 0 or less if the lock is held. To acquire the lock, a hardware thread atomically decrements the integer. The lock prefix on the dec instruction ensures that the decrement executes atomically (see Sect. 3 for more details); locked instructions are also considered to be synchronization primitives in the definition of triangular races. (Furthermore, locked instructions are not to be confused with mutual exclusion locks such as the spinlock here.)

If the spinlock was free before the decrement, it is now held and the thread can proceed to the critical section. If it was held, the thread loops waiting for

[1] Section 3 will explain the **flush** that appears in the pseudo-code.

it to become free. Because there might be multiple threads waiting for the lock, once it is freed, the thread must again attempt to enter through the atomic decrement. To release the lock, the thread simply sets its value to 1.

In a previous version of the kernel, the releasing instruction (in this version, the mov) also had the lock prefix, as a defensive measure against possible relaxed memory behaviors (because it acts as a memory synchronization primitive). Its removal was suggested as a significant performance improvement, but at the time there was no clear picture of the semantics of x86 multiprocessors. After much discussion, and input from an Intel engineer, its removal was agreed [22]. The 2.6.24.7 spinlock without the additional lock is TRF, and hence exhibits only SC behaviors. Thus, the removal of the lock prefix was justified in this case.

We reason informally as follows. Suppose there was a triangular race that included the read at the spin line as one of the racing instructions. There would need to be a prior write on the spinning thread without a synchronization operation in between. However, any path to the spinning read must pass through the locked decrement, a synchronization operation. Because there are no writes between those two operations, this is not a triangular race. The only other read that could participate in a triangular race is from the locked decrement. However, we shall see that locked reads can never be part of a triangular race. Note that the release write and the spin read can participate in an ordinary data race.

2.2 Other Examples

In each of the other examples (Sects. 7–9), just as in the spinlock above, threads communicate by polling locations in shared memory. None of them are DRF because the notifying writes race with the reads that are polling for them. In each case, we apply the TRF principle by first identifying the potential data races. We then consider the possible executions of the reading thread starting from the most recent hardware synchronization primitive (e.g., a barrier or atomic compare and exchange). If a memory write can occur in-between, we have located a triangular race, and have reason to consider adding additional synchronization operations. Otherwise, we conclude that the program has no relaxed-memory-related bugs.

3 The x86-TSO Memory Model

Context Recently, we have formally described two memory models for the x86 architecture. Our first attempt, x86-CC (for causal consistency) [31], captured the then-current Intel and AMD documentation [4, 15], but it turned out to forbid some observed behaviors and also to permit other unobserved behaviors that could significantly complicate programming and reasoning (independent reads of independent writes, or IRIW [6]). In response to this, and to changes in Intel's documentation, we created a TSO based model, x86-TSO [27]. It is consistent with the concrete examples (called *litmus tests*) in Intel's and AMD's latest documentation [15, rev. 32, Sept. 2009] [4, rev. 3.15, Nov. 2009], with our observations of processor behavior, and with our knowledge of x86 folklore, programmer needs, and vendor intentions.

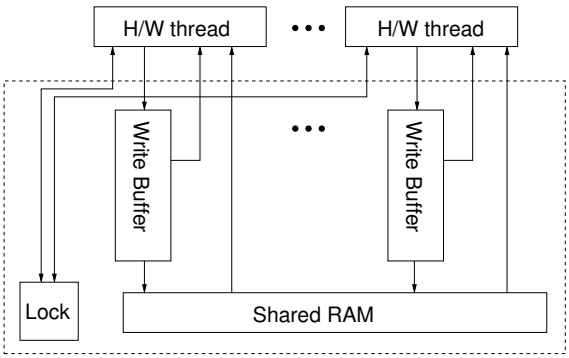


Fig. 3. x86-TSO block diagram

Scope We intend the x86-TSO model to capture the x86's programmer-visible architecture, and so it is significantly abstracted from any hardware implementation details. Furthermore, it covers typical user code and most kernel code (i.e., programs using coherent write-back memory), but we have made no attempt to cover exceptions, misaligned or mixed-size accesses, 'non-temporal' operations (e.g. movnti), self-modifying code, page-table changes, or device memory.

Exceptions and mixed-size accesses do occur in user code, so their absence somewhat limits the applicability of our TRF principle in its current form. Our hope is that these features can be fit into the TSO framework, and that the TRF principle can be extended to cover them. In contrast, the TRF principle would not extend to non-TSO memory types (e.g., write combining) that are used for things other than main memory.

The x86-TSO abstract machine We split the semantics of x86 programs into two parts: the instruction semantics and the memory model. The instruction semantics describes the local behavior of each stream of instruction execution in a system: the hardware threads at the top of Fig. 3.[2] The memory model describes the behavior of shared memory accesses in terms of the components in the lower, boxed part of Fig. 3.[3] Each of the two parts is modeled as a labeled transition system with labels drawn from *events* (Fig. 4). The semantics of the entire system is expressed as a CCS-style parallel composition of the two parts, synchronizing on non-τ labels.

To illustrate, consider the instruction add [56]←eax which adds the value of memory location 56 to the value of register eax, then stores the result back

[2] We use the term "hardware thread" to distinguish both from operating system level threads, and from physical processors, which could be executing multiple hardware threads in parallel.

[3] Our previously published models included register reads and writes in the memory model, but we omit them from this paper to avoid cluttering the presentation. Their behavior is entirely straightforward in x86, and they pose no technical challenges.

event $(e,\ f)$	::=	$\langle \mathsf{W}_p^i[x]v \rangle$	(a write of value v to address x by thread p)
	|	$\langle \mathsf{R}_p^i[x]v \rangle$	(a read of v from x by p)
	|	$\langle \mathsf{B}_p^i \rangle$	(an mfence memory barrier by p)
	|	$\langle \mathsf{L}_p^i \rangle$	(the start of an instruction with lock prefix by p)
	|	$\langle \mathsf{U}_p^i \rangle$	(the end of an instruction with lock prefix by p)
	|	$\langle \tau_p^i[x]v \rangle$	(an internal action of the storage subsystem, moving v from the write buffer on p to x in shared memory)

where i and j are issue indices, p and q identify hardware threads, x and y are memory addresses, and v and w are machine words.

Fig. 4. Events

to [56]. Supposing that 1 is stored at [56] and 2 in eax, an execution of this instruction comprises an event $\langle \mathsf{R}_p^i[56]1 \rangle$ followed by $\langle \mathsf{W}_p^j[56]3 \rangle$. The instruction semantics is internally responsible for specifying how the sum is calculated and for specifying which reads and writes to do. It also tags each event with the originating processor p and an issue index i, which together uniquely identify the event.[4] The memory model is responsible for supplying the values read from memory locations.

Because the overall semantics is split across the event interface, we are able to investigate the meta-theory of the memory model in this paper without becoming ensnared in complications arising from the x86 instruction set. See our previous work [31] for further details on the instruction semantics.

Turning to the memory model in detail, on a write event it records the written address and value in a FIFO write buffer associated with the writing hardware thread. At any time, the memory model can remove the oldest write from the buffer and store it to the shared memory. This action is indicated with a silent transition $\langle \tau_p^i[x]v \rangle$ which is ignored by the instruction semantics (the p, i, x, and v values are taken from the corresponding write event). On a read event the memory model first checks the corresponding thread's write buffer. If the buffer contains a write to the address read from, then the memory model uses the most recently buffered value for that address; otherwise it uses the value currently stored in the shared main memory. Each individual write to a buffer, write from a buffer to shared memory, or read (from a buffer or memory) occurs atomically, represented by a single event.

Returning to the introductory example (Fig. 1), both threads' writes (from instructions 1a and 1c) can be placed in their respective write buffers and not sent to main memory until after both processors have done their reads (1b and 1d). While the write to x is in p's buffer, q reads the value of x from shared memory, and likewise for y in q's buffer.

There are two kinds of events besides reads and writes. First, a barrier event $\langle \mathsf{B}_p^i \rangle$ forces the issuing hardware thread's write buffer to be emptied before proceeding to the next instruction. Barrier events are generated by mfence instruc-

[4] The issue index is used only to distinguish otherwise identical events on the same thread. We elide it when convenient.

Read from memory

$$\frac{\text{not\_blocked}(s,p) \wedge (s.M(x) = \textsc{Some}(v)) \wedge \text{no\_pending}(s.B(p), x)}{s \xrightarrow{\langle \mathsf{R}_p^i[x]v \rangle} s}$$

Read from write buffer

$$\frac{\begin{array}{l}\text{not\_blocked}(s,p) \wedge \\ (\exists b_1 b_2 j.(s.B(p) = b_1 +\!\!+ [(x,v,j)] +\!\!+ b_2) \wedge \text{no\_pending}(b_1, x))\end{array}}{s \xrightarrow{\langle \mathsf{R}_p^i[x]v \rangle} s}$$

Write to write buffer

$$\frac{}{s \xrightarrow{\langle \mathsf{W}_p^i[x]v \rangle} s \oplus \{\!|B := s.B(p \mapsto ([(x,v,i)] +\!\!+ s.B(p)))|\!\}}$$

Write from write buffer to memory

$$\frac{\text{not\_blocked}(s,p) \wedge (s.B(p) = b +\!\!+ [(x,v,i)])}{s \xrightarrow{\langle \tau_p^i[x]v \rangle} s \oplus \{\!|M := s.M(x \mapsto \textsc{Some}(v))|\!\} \oplus \{\!|B := s.B(p \mapsto b)|\!\}}$$

Lock

$$\frac{(s.L = \textsc{None}) \wedge (s.B(p) = [\,])}{s \xrightarrow{\langle \mathsf{L}_p^i \rangle} s \oplus \{\!|L := \textsc{Some}(p)|\!\}}$$

Unlock

$$\frac{(s.L = \textsc{Some}(p)) \wedge (s.B(p) = [\,])}{s \xrightarrow{\langle \mathsf{U}_p^i \rangle} s \oplus \{\!|L := \textsc{None}|\!\}}$$

Barrier

$$\frac{s.B(p) = [\,]}{s \xrightarrow{\langle \mathsf{B}_p^i \rangle} s}$$

Notation: SOME and NONE construct optional values, $(\cdot,\cdot)$ builds tuples, $[\,]$ builds lists, $+\!\!+$ appends lists, $\cdot \oplus \{\!|\cdot := \cdot|\!\}$ updates records, and $\cdot(\cdot \mapsto \cdot)$ updates functions.

Fig. 5. The x86-TSO memory model

tions. Second, a processor can lock ($\langle \mathsf{L}_p^i \rangle$) or unlock ($\langle \mathsf{U}_p^i \rangle$) the memory system; while the memory system is locked by a particular processor, no other processor can read from or write to memory. Locks and unlocks are used by the instruction semantics to implement locked instructions (including atomic increment lock inc, compare and exchange cmpxchg, and a limited set of others) which ensure that all memory accesses by the locked instruction happen together, atomically in the system. Lock and unlock events also function as barriers.

Figure 5 presents the transition rules for the memory model formally, where a state s is of the following record type (*addr* is the type of memory addresses, *tid* is the type of thread identifiers, and *idx* is the type of issue indices):

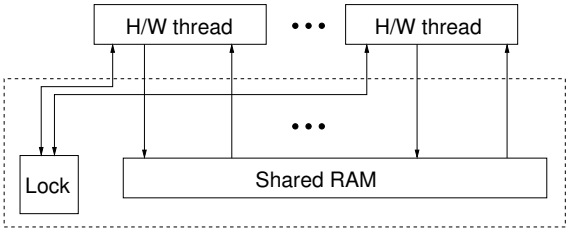
$$\{\!|M : addr \to (value \text{ OPTION});\ B : tid \to (addr \times value \times idx) \text{ LIST};\ L : tid \text{ OPTION}|\!\}$$

The rules use two auxiliary definitions: p is *not blocked* in machine state s if either it holds the lock or the lock is not held, and there are *no pending* writes in a buffer b for address x if there are no (x, v, i) elements in b. We also impose the following progress condition: each write put into a write buffer must eventually leave it.

4 Sequentially Consistent Programs

Section 3 presented an abstract machine for x86-TSO, our model of the observable memory behavior of x86 processors. Here, we define x86-SC, a simple model of the sequentially consistent memory behavior that one might wish x86 multiprocessors had, and that they do have for DRF and TRF programs.

The x86-SC abstract machine (below) separates the instruction semantics and memory model in exactly the same way as the x86-TSO abstract machine (Fig. 3). In fact, both memory models use the same instruction semantics. The only difference between the two is that the x86-SC abstract machine has no write buffers. Writes propagate to shared memory immediately upon execution, and all reads consult shared memory. We model this by requiring each $\langle \mathsf{W}_p^i[x]v \rangle$ event to be immediately followed by the corresponding $\langle \tau_p^i[x]v \rangle$ event. Thus, formally, an x86-SC execution is an x86-TSO execution where every write to a buffer is immediately flushed. However, when writing down x86-SC traces, we often omit the τ labels for clarity. Barrier events have no effect on x86-SC, but locked instructions still ensure that their constituent events happen atomically using lock and unlock events, just as in x86-TSO.



Our main goal is to characterize a class of programs to which we can apply reasoning techniques, both formal and informal, that are based on sequentially consistent semantics. The following defines one such class.

Definition 1 (resultSC). *A program is* result-sequentially-consistent (resultSC) *iff for every x86-TSO execution there exists an x86-SC execution with the same result. By same result we mean that, if the x86-TSO execution diverges, then the x86-SC one does too; and if the the x86-TSO execution converges, then the x86-SC one does too, with the same shared memory final state, where all write buffers must be empty in an x86-TSO final state.*

Result sequential consistency captures programs that, during x86-TSO execution, might have behaviors which are not equivalent to x86-SC behaviors, but

where that the final result does not depend upon the difference. However, our TRF approach is not powerful enough to detect such situations. To precisely characterize our approach, we define a notion of *memory equivalence* which additionally considers memory writes to be observable.

Two memory equivalent traces must have the same memory writes in the same order, and their corresponding read events must not only have the same values, but those values must have been put in place by the same write event. Traditional DRF principles ensure that every execution on a relaxed machine has a memory equivalent sequentially consistent trace. As we shall see (Sect. 6), TRF exactly characterizes the class of programs in which every x86-TSO execution has a memory equivalent x86-SC one.

To define memory equivalence, we first define the write that a read event *reads from*.

Definition 2 (Reads from). *Let $\prec$ order events according to a given execution trace. A read event* $\langle \mathsf{R}_p^i[x]v\rangle$ reads from *the preceding write that most recently wrote to x. It is the maximal element in* $\{\langle \mathsf{W}_p^j[x]w\rangle | \langle \mathsf{W}_p^j[x]w\rangle \prec \langle \mathsf{R}_p^i[x]v\rangle \wedge \langle \tau_p^j[x]w\rangle \not\prec \langle \mathsf{R}_p^i[x]v\rangle\}$ *(the set of writes in p's buffer, which is always empty in an x86-SC execution) if this set is nonempty. Otherwise, it is the maximal element in* $\{\langle \tau_q^j[x]w\rangle | \langle \tau_q^j[x]w\rangle \prec \langle \mathsf{R}_p^i[x]v\rangle\}$. *If there is no such event, then the read* reads from *the initial state.*

Definition 3 (Memory equivalence). *Two executions traces are* memory equivalent *iff they have the same subsequence of writes to shared memory (τ events), and there exists a bijection between the read events of each trace such that corresponding read events read from the same write event/initial state.*

Definition 4 (memorySC). *A program is* memory sequentially consistent (memorySC) *iff for each of its possible executions on x86-TSO, there exists a memory equivalent execution on x86-SC.*

In Sect. 8 we explore the gap between resultSC and memorySC.

5 Data Races

Although the details vary in the literature, data races are typically defined roughly as follows [1–3, 5–7, 14, 23, 30]. Two operations on different threads *compete* if they access the same memory address and one of them is a write, and two operations are *concurrent* if there is no temporal relationship between them. Rigorous definitions of when two events are concurrent take several forms, such as being unrelated by a happens-before relation [21], or adjacent in an SC execution trace—depending on the underlying formalism. A *data race* is then a pair of competing, concurrent operations. Synchronization primitives are handled by either ignoring certain races on them, or by augmenting the temporal information (e.g., happens before) with dependencies between synchronization primitives.

For example, if a program can reach a point where the next event could be either $\langle \mathsf{W}_p[x]v_1\rangle$ or $\langle \mathsf{R}_q[x]v_2\rangle$ (with $p \neq q$), then it has a data race. However,

if both accesses of x are in critical sections, then this point cannot be reached because both p and q cannot be in the critical section simultaneously.

Many relaxed memory models satisfy the "fundamental property" of Saraswat et al. [30] that DRF programs exhibit only sequentially consistent behavior. The key insight is that whenever there is a visible temporal dependency between two operations, if the memory model ensures that the dependency is respected throughout the system, then the entire system must have a consistent view of their ordering (i.e., the one prescribed by the dependency). Data-race freedom requires everything that could potentially observe system-wide inconsistencies (i.e., competing operations) to have a temporal dependency, thus rendering any internal inconsistencies unobservable.

Here, we focus on a single model, x86-TSO, and so we customize our notion of data race specifically for it. Firstly, we need not consider two competing, concurrent write events to be a data race. Because foreign writes are visible to a hardware thread only through a single global shared memory, the system intrinsically maintains global consistency for the ordering of writes—the program does not need to.

Secondly, we address synchronization by ignoring potential data races whose read events come from a locked instruction. Because the hardware thread's write buffer will be flushed at the end of the locked instruction, the reading thread's local perception of shared memory will be reflected in global memory before any other thread can access memory. (Note that the preceding reasoning does not apply to read events followed by barrier events, since other threads' events could observe memory in between the execution of the read and the barrier.) This is more general than some formulations of data races because a concurrent, competing write does not also need to be locked to avoid a data race.

Formally, we define a data race to be a prefix of an x86-SC execution, using the intuition that adjacent events from different threads in an execution trace could have occurred in the opposite order. We take care to ensure that a locked write and unlocked read can form a data race.

Definition 5 (Data race). *A* data race *is a prefix of an x86-SC execution with either of the following two shapes:*
$e_1 \dots e_n \langle \mathsf{R}_q[x]v \rangle \langle \mathsf{W}_p[x]w \rangle$ *or* $e_1 \dots e_n \langle \mathsf{R}_q[x]v \rangle \langle \mathsf{L}_p \rangle f_1 \dots f_m \langle \mathsf{W}_p[x]w \rangle$
where $p \neq q$ and none of the f_i are unlocks.

For example, two threads p and q attempting to increment x, which is initially 0, have a data race (matching the first shape): $\langle \mathsf{R}_q[x]0 \rangle \langle \mathsf{R}_p[x]0 \rangle \langle \mathsf{W}_q[x]1 \rangle$. If one increment is locked, say on p, there is still a race (matching the second shape): $\langle \mathsf{R}_q[x]0 \rangle \langle \mathsf{L}_p \rangle \langle \mathsf{R}_p[x]0 \rangle \langle \mathsf{W}_p[x]1 \rangle$. However, if both increments are locked, there is no race. There are only two sequential executions (the other is achieved by swapping p and q): $\langle \mathsf{L}_p \rangle \langle \mathsf{R}_p[x]0 \rangle \langle \mathsf{W}_p[x]1 \rangle \langle \mathsf{U}_p \rangle \langle \mathsf{L}_q \rangle \langle \mathsf{R}_q[x]1 \rangle \langle \mathsf{W}_q[x]2 \rangle \langle \mathsf{U}_q \rangle$. Neither of them match the shapes in Definition 5.

For a second example, consider a simple parallel prime sieve [17]. One processor writes to a shared array at indices that are multiples of 2 (greater than 2), another writes for 3, and so on. Although there are indices (e.g., 6) with

Initial: $[x] = 0 \wedge [y] = 0 \wedge x \neq y$	
p	q
6a: mov $[x]$←1	6b: mov $[y]$←1
	6c: mov eax←$[x]$ (0)

Fig. 6. A simple triangular race

competing, concurrent events, they are not data races because both events are writes.

As a corollary to our main theorem (Theorem 1 in Sect. 6) we have the following, which is stronger than traditional DRF principles because we do not consider write-write data races.

Corollary 1 (DRF theorem). *Every DRF x86 program is memorySC.*

The converse does not hold, as the two instruction program that reads x on one processor and writes x on another is not DRF, but is memorySC.

6 Triangular Races

If a read event occurs while its hardware thread's write buffer is empty, then the local perception of the state of shared memory must coincide with the global one. We use this idea to further strengthen our notion of a data race into a *triangular race*. A triangular race comprises a data race between $\langle \mathsf{R}_q[x]v_1 \rangle$ and $\langle \mathsf{W}_p[x]v_2 \rangle$ (where $p \neq q$) along with a preceding write event $\langle \mathsf{W}_q[y]w \rangle$ on the reading thread that could be in the local write buffer when the read happens. Thus, q issues no locks, unlocks, or barriers between $\langle \mathsf{W}_q[y]w \rangle$ and $\langle \mathsf{R}_q[x]v_1 \rangle$.

Returning to the initial example (Fig. 1), when a data race is encountered—suppose that 1a has executed, and there is now a race on y—the write to x could still be in p's write buffer. In contrast, if we add an mfence (barrier) instruction after each write, the buffers will be empty whenever a race is encountered, and the program will be memorySC.

To see how a triangular race can lead to non-sequentially consistent behavior, consider the simple triangular race in Fig. 6 which can perform the following sequence of actions on the x86-TSO machine.

1.		$\langle \mathsf{W}^1_q[y]1 \rangle$	buffer the write of 1 to y (6b)
2.		$\langle \mathsf{R}^2_q[x]0 \rangle$	read x from main memory (6c)
3.	$\langle \mathsf{W}_p[x]1 \rangle$		buffer the write of 1 to x (6a)
4.	$\langle \tau_p[x]1 \rangle$		write 1 to x in shared memory (6a)
5.		$\langle \tau_q[y]1 \rangle$	write 1 to y in shared memory (6b)

Suppose we wish to construct a memory equivalent x86-SC trace. #1 must immediately precede #5, and #3 must immediately precede #4 in an x86-SC execution. Furthermore, #3 must precede #1 to maintain memory equivalence (the writes of 6a to shared memory before those of 6b). Thus, the trace must

Initial: $[x] = 0 \wedge [y] = 0 \wedge x \neq y$		
p	q	q'
7a: mov $[x]$←1	7b: mov $[y]$←1	7d: mov ecx←$[x]$ (1)
	7c: mov ebx←$[x]$ (0)	7e: mov edx←$[y]$ (0)
Allow: ebx $= 0 \wedge$ ecx $= 1 \wedge$ edx $= 0 \wedge$ ecx $= 1 \wedge$ edx $= 0$		

Fig. 7. Observing write ordering leads to relaxed behavior

be #3#4#1#5 with #2 inserted at the start, between #4 and #1, or at the end. Only at the first does the read get value 0, and not 1. However, in that execution a read from instruction 6c precedes a write from 6b—out of program order. The instruction semantics does not permit this, and so this program is not memorySC.

It is resultSC, as the ordering #1#5#2#3#4, or simply 6b6c6a, demonstrates. The difference is that program's result is not affected by the shared memory ordering of these writes. However, the program can be extended to one whose result is; in Fig. 7 the indicated result can be reached only if the 7a write is sent to shared memory before the 7b one is, and since this is visible in the result, this program is not resultSC.

Our formal definition of a triangular race follows that of a data race. Note that a triangular race is also a data race.

Definition 6 (Triangular race). *A* triangular race *is a prefix of an x86-SC execution with either of the following two shapes:*
$e_1 \dots e_m \langle \mathsf{W}_q[y]v_1 \rangle \langle \mathsf{R}_q[z_1]w_1 \rangle \dots \langle \mathsf{R}_q[z_n]w_n \rangle \langle \mathsf{R}_q[x]v_2 \rangle \langle \mathsf{W}_p[x]v_3 \rangle$ *or*
$e_1 \dots e_m \langle \mathsf{W}_q[y]v_1 \rangle \langle \mathsf{R}_q[z_1]w_1 \rangle \dots \langle \mathsf{R}_q[z_n]w_n \rangle \langle \mathsf{R}_q[x]v_2 \rangle \langle \mathsf{L}_p \rangle f_1 \dots f_o \langle \mathsf{W}_p[x]v_3 \rangle$
where $x \neq y$ *and* $p \neq q$ *and* $x \notin \{z_1 \dots z_n\}$ *and none of the* f_i *are unlocks.*

We can now state our main theorem, whose proof we defer until the appendix.[5]

Theorem 1 (TRF theorem). *An x86 program is memorySC iff it is TRF.*

Because Theorem 1 is an equivalence, any extension of TRF must necessarily admit some programs with x86-TSO executions that are not memory equivalent to any x86-SC executions. We return to this point in Sect. 8.

7 Locking Primitives

We now return to the spinlock of Fig. 2 and use Theorem 1 to prove that it works; then we address a more sophisticated variant. For this section, we assume that some set of addresses is distinguished as holding spinlocks.

[5] Theorem 1 relies on some facts about the x86 instruction semantics, including that locked instructions cannot access multiple different addresses.

We first define when a program is using a spinlock properly, as follows.

Definition 7 (Spinlock well-synchronized). *A program is* spinlock well-synchronized *with respect to a particular spinlock implementation iff for every x86-SC execution, and for every pair of competing events that are not on a spinlock, there is a spinlock that is released and then acquired between them.*

We omit mention of which threads the release and acquire are on in Definition 7. Because a spinlock well-synchronized program must have a lock and unlock between competing events on every execution, the unlock must necessarily be on the thread of the first competing event, and the lock must necessarily be on the second. Otherwise, there would be another execution that has the lock/unlock and competing event in the opposite order, and so not between the two competing events. We use the assumption of spinlock well-synchronization exclusively to apply the following lemma.

Lemma 1. *In a spinlock well-synchronized program, any data race is on a spinlock's address.*

Proof. Using the contrapositive, suppose there is a data race $\ldots\langle \mathsf{R}_p[y]v\rangle\langle \mathsf{W}_q[y]w\rangle$ (or $\ldots\langle \mathsf{R}_p[y]v\rangle\langle \mathsf{L}_q\rangle\ldots\langle \mathsf{W}_q[y]w\rangle$) where y is not the address of a spinlock. Then there are two competing events $\langle \mathsf{R}_p[y]v\rangle\langle \mathsf{W}_q[y]w\rangle$ without a spinlock release and the acquire in between, and so the program is not well-synchronized. □

Theorem 2. *If an x86 program is spinlock well-synchronized (with respect to the spinlock in Fig. 2) and the locations of spinlocks are only accessed by the code in Fig. 2, then it is memorySC.*

Proof. By Theorem 1 it suffices to show that there are no triangular races. Since a triangular race is a data race, by Lemma 1 and by assumption, we only need to analyze the possible data races on a spinlock x that involve only code in Fig. 2 and show that none can be triangular. The instruction semantics guarantees that the trace of a thread after entering acquire or release is included in the following (using regular expression notation):

acquire: $(\langle \mathsf{L}_p\rangle\langle \mathsf{R}_p[x]w_1\rangle\langle \mathsf{W}_p[x]w_2\rangle\langle \mathsf{U}_p\rangle\langle \mathsf{R}_p[x]v_1\rangle\ldots\langle \mathsf{R}_p[x]v_n\rangle\langle \mathsf{R}_p[x]1\rangle)^*$
$(\langle \mathsf{L}_p\rangle\langle \mathsf{R}_p[x]1\rangle\langle \mathsf{W}_p[x]0\rangle\langle \mathsf{U}_p\rangle|\epsilon)$

release: $\langle \mathsf{W}_p[x]1\rangle$

Any race must include a $\langle \mathsf{R}_p[x]w\rangle$ event from acquire. Every such event is immediately preceded by either $\langle \mathsf{L}_p\rangle$, $\langle \mathsf{U}_p\rangle$, or $\langle \mathsf{R}_p[x]w'\rangle$, none of which are permitted by a triangular race (nor are events from other threads permitted). □

7.1 A Ticketed Spinlock

Recently, the Linux kernel changed from the spinlock in Fig. 2 to a fairer, ticketed spinlock (Fig. 8).[6] To acquire the lock, a thread first atomically increments

[6] Figure 8 differs from the Linux version by storing the two fields that comprise the lock in separate words, instead of in the lower and upper halves of the same word.

```
; The address of the next ticket to give out, y, is stored in register ebx, and
; the address of the ticket currently being served, x, is stored in register eax.
acquire:  mov ecx←1              ; tkt := 1
          lock xadd [ebx]←ecx    ; atomic (tkt := [y]
                                 ;         [y] := tkt + 1
                                 ;         flush local write buffer)
    spin: cmp [eax],ecx          ; flag := ([x] = tkt)
          je enter               ; if flag then goto enter
          jmp spin               ; goto spin
   enter:                        ; the critical section starts here
release:  inc [eax]              ; [x] := [x] + 1
```

Fig. 8. A ticketed x86 spinlock inspired by Linux v2.6.31

the ticket using a locked instruction, and then loops until its ticket is ready to be served. To release the lock, it increments the ticket being served. The key difference from the previous spinlock is that the release instruction both reads and writes. It does not need to be atomic, because two threads cannot be attempting to simultaneously release the spinlock (assuming that threads only try to release spinlocks that they hold).

To show that this lock guarantees sequential consistency, we will need to know that it ensures mutual exclusion on x86-SC.

Definition 8 (Correctly locked). *A program is* correctly locked *if each of its x86-SC execution traces satisfies the following properties.*

1. *The locations of spinlocks are only accessed by the code in Fig. 8.*
2. *For each hardware thread p, control only enters* release *on p to release a lock $\langle x, y\rangle$ when previously, control left* acquire *on p to acquire $\langle x, y\rangle$, without another release of $\langle x, y\rangle$ on p in between (i.e., threads only release locks they hold).*

Lemma 2 (Spinlock mutual exclusion). *In a correctly locked x86 program, if a hardware thread reaches the* enter *line of a spinlock, no other thread can reach the* enter *line until the first thread completes the increment from* release*.*

Proof outline. By standard sequentially consistent reasoning techniques; we do not go into detail here. The difference in the next ticket and the currently served ticket is the number of threads that have entered acquire, but not finished release. We assume there are fewer than 2^{32} hardware threads.

Theorem 3. *If a correctly locked x86 program is spinlock well-synchronized (with respect to Fig. 8), then it is memorySC.*

Proof. As in the proof of Theorem 2, we analyze the possible data races on the spinlock's data and show that none of them can be triangular.
acquire: $\langle \mathsf{L}_p\rangle\langle \mathsf{R}_p[y]w_1\rangle\langle \mathsf{W}_p[y]w_2\rangle\langle \mathsf{U}_p\rangle\langle \mathsf{R}_p[x]v_1\rangle \ldots \langle \mathsf{R}_p[x]v_n\rangle$
release: $\langle \mathsf{R}_p[x]v_1\rangle\langle \mathsf{W}_p[x]v_2\rangle$

No race involving $\langle \mathsf{R}_p[y]w \rangle$ can be a triangular race, because this event is always immediately preceded by $\langle \mathsf{L}_p \rangle$. Neither can a race involving $\langle \mathsf{R}_p[x]w \rangle$ from acquire because it is immediately preceded by either $\langle \mathsf{U}_p \rangle$ or $\langle \mathsf{R}_p[x]w' \rangle$. The $\langle \mathsf{R}_p[x]v \rangle$ in release might be preceded by appropriate events from the critical section to be a triangular race. There are no writes to x in in acquire to race with, so the race must be on another processor doing a release. Thus, any potential triangular race must be as follows:
$\ldots \langle \mathsf{R}_q[x]v_1 \rangle \ldots e_1 \ldots e_n \langle \mathsf{R}_p[x]v_2 \rangle \langle \mathsf{W}_q[x]v_3 \rangle$
where the e_i are all on thread p, $p \neq q$, and there are no events on q between the given read and write (i.e., assume this is the read event from the same inc instruction as the write).

Because the program is correctly locked, p and q both acquire x before $\langle \mathsf{R}_p[x]v_2 \rangle$, although p's acquire might occur before or after $\langle \mathsf{R}_q[x]v_1 \rangle$, and there are no releases in between. Thus, both threads have acquired the lock before either thread has finished releasing the lock, contradicting Lemma 2. □

8 Examples with Potential Triangular Races

In this section, we investigate two examples that are not well-synchronized. In both cases, barriers can be added to ensure triangular-race freedom, but at a performance cost. The examples also illustrate the difference between memorySC and resultSC programs (Sect. 4). We argue, reasoning directly with x86-TSO, that without barriers they are resultSC in some contexts, but not in others. We then comment on how our examples illustrate a general publication idiom.

Non-Blocking Write Protocol Figure 9 presents a non-blocking write protocol [19] similar to Linux's SeqLocks [20]. In this instance, two memory addresses y_1 and y_2 make up a conceptual register that a single hardware thread can write to, and any number of other threads can attempt to read from. A version number is stored at x. The writing thread maintains the invariant that the version number is odd during writing by incrementing it before the start of and after the finish of writing. A reader checks that the version number is even before attempting to read (otherwise it could see an inconsistent result by reading while y_1 and y_2 are being written). After reading, the reader checks that the version has not changed, thereby ensuring that no write has overlapped the read.

We want to see how a triangular race could occur in a program using this protocol. Notice that the Reader code does not write to memory. Thus, a program where the reading processors only access memory via the code at Reader is trivially TRF. However, there are data races between the writer and a reader on x, y_1, and y_2, and if the reading processor has written to memory before initiating the read, these become triangular races.

If we are concerned with memory sequential consistency, then there is no choice but to prefix the entry to the read operation with a barrier to prevent any preceding writes from taking part in a triangular race. However, it might be that even without the barrier, the whole program is resultSC. For example,

```
; The address of the current version x is stored in register eax, and
; its contents at y1 and y2.
; The version, [x], is odd while the writer is writing, and even otherwise.
Writer:  mov ebx←1          ; tmp := 1
         xadd [eax]←ebx     ; tmp := [x]
                            ; [x] := tmp + 1
         mov [y1]←v1        ; [y1] := v1
         mov [y2]←v2        ; [y2] := v2
         inc ebx            ; tmp := tmp + 1
         mov [eax]←ebx      ; [x] := tmp

Reader:  mov ebx←[eax]      ; tmp := [x]
         mov ecx←ebx        ; tmp2 := tmp
         and ecx←1          ; tmp2 := tmp2&1
         cmp ecx,0          ; flag := (tmp2 ≠ 0)
         jne read           ; if flag then goto Reader
         mov ecx←[y1]       ; result1 := [y1]
         mov edx←[y2]       ; result2 := [y2]
         cmp [eax],ebx      ; flag := ([x] ≠ tmp)
         jne read           ; if flag then goto Reader
```

Fig. 9. A versioning non-blocking write protocol

consider the following illustrative pseudocode where the readers communicate with each other using spinlock-based (Fig. 2) synchronization. Recall that the release operation does not contain a barrier.

p_1	p_2	p_3
Writer	acquire spinlock y	Reader
	mov $[x]$←eax	acquire spinlock y
	release spinlock y	mov ebx ←$[x]$
	Reader	release spinlock y

This is essentially the same pattern as the Fig. 7 example with locks added. As there, we can observe non-sequentially consistent behavior if p_2's reads do not observe p_1's writes, but p_3's read does, as follows on x86-TSO (reading from top to bottom):

1.		acquire spinlock y
2.		write to x put into buffer
3.		spinlock release y into buf.
4.		Reader gets old value
5.	Writer writes and flushes	
6.		Reader read new value
7.		start acquire spinlock y

After 7, p_3 will loop trying to acquire the lock until p_2 releases it by flushing the buffered unlocking write to y to shared memory. However, the write to x must be flushed first, and thus p_3's read from x will see the new value. This behavior can be accounted for on x86-SC if p_2 runs all of its instructions, followed by p_1,

```
; The address of the object x is stored in memory at location [eax].
; An uninitialized object is represented by the address 0.
ensureinit:  cmp [eax],0          ; flag := x is initialized
             jne initialized      ; if flag then goto initialized
             ---------------------; acquire a spinlock
             cmp [eax],0          ; flag := x is initialized
             jne unlock           ; if flag then goto initialized
             ---------------------; writes to initialize the object,
                                  ; leaving its address in ebx
             mov [eax]←ebx        ; x := initialized value
unlock:      ---------------------; release the spinlock
initialized:                      ; Now the object can be used
```

Fig. 10. Double-checked Locking

and finally p_3. The difference is that on the x86-TSO trace, all of p_1's writes reach shared memory before p_2's write to x, whereas that does not happen in the x86-SC trace.

If the final result might depend on the ordering of p_1's and p_2's writes as well as the values read by p_2 and p_3, then the program might not be resultSC. In this case, a barrier after p_2 releases the spinlock would be required to maintain sequentially consistent reasoning.

Double-checked Locking Double-checked locking [32] is an optimization idiom for objects whose one-time initialization must occur in a critical section, but further accesses are not restricted. It is famously unsound in Java and C++ [13, 24] due to compiler re-orderings, exceptions, etc.; however, it does work on many processors, including x86. Figure 10 presents the idiom; an object x is never accessed without first ensuring that it has been initialized with ensureinit.

On x86-SC, one of three things can happen when ensuring initialization.

1. Read x, finding it initialized; proceed to use x.
2. Read x, finding it un-initialized; lock; read x, finding it initialized; unlock; proceed to use x.
3. Read x, finding it un-initialized; lock; read x, finding un-initialized; write to initialize; write to x; unlock; proceed to use x.

For correct operation, the third option should happen at most once (i.e., the object should not be initialized multiple times), and whenever the object is found to be initialized (options 1 and 2), it should actually be initialized. It works on x86-SC, since the second and subsequent entrants to the critical section will see the initialized pointer.

Turning to whether it is TRF, the write to x by a thread in the third case can race against the first read from x by another thread in one of the first two cases. (Note that the initializing writes cannot be part of a race because no other thread can read them until after they read the initializing thread's write to x.) Just as with the non-blocking writer above, this race is triangular if, and only

if, the read is preceded by a write, but not a barrier or locked instruction. Thus, double-checked locking works for x86-TSO if a barrier is always executed before attempting the read.

We conjecture that typical uses of double-checked locking without the barrier are resultSC. Suppose that the initializing write to x is in p's write buffer, and another processor q reads the un-initialized value of x. It will try to acquire the lock, and once acquired x will be seen to be initialized, following similar reasoning to the unlock in the non-blocking writer example above. So even though its local behavior may differ (taking the lock, whereas it would not on x86-SC), this should not affect the result. However, this is by no means guaranteed, and establishing it might require fully general x86-TSO reasoning. For example, if p can re-acquire the spinlock from some other point in the code (perhaps it is protecting more than one object) before q enters, then we must prove that p will eventually release the lock.

Publication Idioms The unlocking write from the spinlocks and the initializing write to x in the double-checked locking example both demonstrate a general publication idiom. To publish local changes, a hardware thread writes to a single shared location that other threads can observe. Once another thread sees the write, it has permission to view the new data (in the spinlock's case, the data is whatever was written in the critical section). Before the write, the other threads wait for the data, either directly, as in the spinlock, or indirectly, as in the double-checked locking idiom (by subsequently waiting for a spinlock). Our analysis of these examples indicates that such publication idioms need careful, potentially TSO-specific analysis, unless the reads that detect publication are preceded by barriers.

9 A JVM Bug Due to a Triangular Race

A recent blog posting by Dice [12] discusses his discovery of the cause of a concurrency bug in the HotSpot JVM's implementation of blocking synchronization for `java.util.concurrent`. On a certain fast-path through the `Parker::park` method, a missing mfence instruction allowed a wake-up call to be lost, leading to the possibility of hung threads. The blog post explains the bug with 7 carefully chosen execution steps which are directly in terms of write buffers, using an x86-TSO-like model. These steps span two calls to the `park` method on one thread and a call to the `unpark` method on the other.

Here we briefly present the bug, we explain why the buggy program contains a triangular race, and we explain why the addition of mfence instructions to repair the bug also removes the triangular race.

Figure 11 presents a simplified version of the `Parker`. A thread calls the `park` method when it wants to wait for some condition to hold. Other threads call the `unpark` method after they make the condition hold, waking the first thread. However, unlike a semaphore, the thread waiting in `park` can awaken and return

```
class Parker {
  volatile int _counter = 0;
  pthread_mutex_t _mutex [1];    pthread_cond_t _cond [1];
};

void Parker::park() {
  if (_counter > 0) {
    _counter = 0;
    // mfence needed here
    return;
  }
  if (pthread_mutex_trylock(_mutex) != 0) return;
  if (_counter > 0) { // no wait needed
    _counter = 0;
    pthread_mutex_unlock(_mutex);
    return;
  }
  pthread_cond_wait(_cond, _mutex);
  _counter = 0;
  pthread_mutex_unlock(_mutex);
}

void Parker::unpark() {
  pthread_mutex_lock(_mutex);
  int s = _counter;
  _counter = 1;
  pthread_mutex_unlock(_mutex);
  if (s < 1) pthread_cond_signal(_cond);
}
```

Fig. 11. A simplified `Parker` from HotSpot (written in C++) taken from [12]

without any corresponding call to `unpark`. Thus, the parked thread must, upon awakening, check that the condition holds, and call `park` again if it does not.

For example, if thread p is awaiting the condition `x == 0`, and has a parker `pk`, the parker would be used as follows:

```
while !(x == 0) pk.park();
```

Another thread q would signal the first thread thus:

```
x = 0;  mfence();  pk.unpark();
```

In terms of read and write events, p does the following, assuming that `pk._counter` and `x` start at 1 (perhaps some other thread has already set `x` to 0, called `unpark`, and then set `x` to 1). It first reads from the location of `x`, then, in the call to `park`, reads from the location of `pk._counter` (suppose it is at address y). It next writes to the location of `pk._counter` and returns, reading the location of `x` again. In event notation: $\langle \mathsf{R}^i_p[x]1\rangle\langle \mathsf{R}_p[y]1\rangle\langle \mathsf{W}_p[y]0\rangle\langle \mathsf{R}^j_p[x]1\rangle$.

Suppose thread q unparks p with the following sequence: $\langle \mathsf{W}_q[x]0\rangle\langle \mathsf{B}_q\rangle \ldots$ (eliding the events from the call to `unpark`). Then the following interleaving (where q simply follows p) has a triangular race:
$\ldots \langle \mathsf{R}^i_p[x]1\rangle\langle \mathsf{R}_p[y]1\rangle\langle \mathsf{W}_p[y]0\rangle\langle \mathsf{R}^j_p[x]1\rangle\langle \mathsf{W}_q[x]0\rangle\langle \mathsf{B}_q\rangle \ldots$

The bug is made manifest when the subsequent call to `park` happens after the call to `unpark` is completed. However, finding the triangular race does not require looking into the internals of `unpark` or considering multiple calls to `park`. In general, the presence of a triangular race does not guarantee that a bug has been found, but in this instance there would be good reason for suspicion since the ordering of writes to x and y (the locations in the triangular race) are key to the correct functioning of the algorithm.

A direct way to spot the triangular race is to notice the data race on x and then to check for preceding writes on the same thread as the read part of the data race. A call to `park` can precede the read from x, and it can write to `_counter` and immediately exit, forming a triangular race. In fact the write `_counter`, read `x` pattern on p combined with the write `x`, mfence, read `_counter` pattern on q (looking into the implementation of `unpark`) is analogous to Fig. 1, but with a single mfence.

With an additional mfence added to the `park` method, immediately following the write to `_counter`, the sequence becomes:
$\ldots \langle \mathsf{R}^i_p[x]1\rangle\langle \mathsf{R}_p[y]1\rangle\langle \mathsf{W}_p[y]0\rangle\langle \mathsf{B}_p\rangle\langle \mathsf{R}^j_p[x]v\rangle\langle \mathsf{W}_q[x]0\rangle\langle \mathsf{B}_q\rangle \ldots$
This is not a triangular race, but it is a data race. In fact, the corrected code is TRF (as long as `pthread_mutex_unlock` contains an mfence or an instruction with the lock prefix, or mfences are added after those writes to `_counter` as well). We argue this informally, but we cannot formally prove it without a model of the pthreads primitives.

Any triangular race must contain a data race where the reading operation is preceded on the same thread by a write without an intervening mfence or lock. The only read on an unparking thread is from `_counter` and that is immediately preceded by an mfence before the call. Hence, the racing read must be on the parking thread. The reads from `x` and `_counter` on the parking thread, with one exception, are only preceded by writes from a previous call to `park` in the loop, and those writes are now all followed by mfence instructions. The exception is a data race on x, where the read is the first one on entry into the `while` loop, and the preceding write occurs before the while loop on p. This is exactly the sort of triangular race discussed in Sect. 8.

10 Related work

Burkhardt and Musuvathi [9] characterize "store buffer safety" for TSO programs in terms of sequentially consistent traces by explicitly building the happens-before relation of both SC and TSO memory models at every step (using vector clocks). Thus, although only SC executions are considered, the relaxed memory model cannot be ignored, as it can in our approach. We conjecture that their store buffer safety property is in-between our memorySC and resultSC properties. It

allows the ordering memory writes to vary when unobserved, but everything else must be the same (e.g., reads must read from the same writes).

Cohen and Schirmer [10] have proposed a programming discipline to ensure sequential consistency for TSO programs. Like us, they do not consider any synchronization operations beyond what the hardware provides, and their discipline captures the same intuition as our triangular races regarding the write buffer being empty when reading. Their discipline is based on ownership (e.g., writes to locally owned, unshared memory do not need to be flushed before a read), and so notions of ownership and ghost operations pervade their programs and memory model semantics. However, the ordering of writes to locally owned memory locations cannot be observed by other threads, so ownership information gives them an approach to verifying resultSC for some programs which are not memorySC.

Park and Dill [28] verify programs by model checking them directly on the semantics of TSO, and the related-but-more-relaxed PSO and RMO. Shasha and Snir [33] show how to transform a program so that it has only sequentially consistent executions on a relaxed memory architecture.

11 Future Work and Conclusion

Our focus in this paper has been on creating a semantic foundation for reasoning about programs above TSO-like relaxed memory models. We have demonstrated the usefulness of our TRF principle on a variety of low-level concurrency algorithms that are important to the implementors of languages that support shared memory concurrency. However, formal reasoning directly on traces can be tedious, so a program logic or sound static analyzer specialized to proving triangular-race freedom might make the application of TRF more convenient.

Currently, DRF-style principles, including TRF, can be applied only to programs are globally DRF (or TRF). If a small piece contains a race, then the entire program must be reasoned about with relaxed-memory-specific techniques. Ideally, this relaxed reasoning could be applied to the (small) part of the program that requires it (such as we did in Sect. 8), and SC-reasoning used for the rest. To support this approach, a compositional DRF principle would be invaluable.

Our work has illustrated the importance of considering how relaxed executions are equivalent to sequentially consistent ones. We hope a careful study of which equivalences support which kinds of reasoning will be a fruitful direction for creating new DRF-style principles.

Acknowledgements We thank Peter Sewell for discussions, Francesco Zappa Nardelli and Susmit Sarkar for their comments on drafts. We also thank Dave Dice for bringing the "Parker" bug to our attention, and thank the ECOOP reviewers for their suggestions. We acknowledge funding from EPSRC grant EP/F036345.

A Proof Sketches for Theorem 1

See `http://www.cl.cam.ac.uk/~so294/ecoop2010/` for complete proofs. First we prove the completeness of TRF with respect to memorySC.

Lemma 3. *If a program has a triangular race, then it is not memorySC.*

Proof. Given a triangular race without lock:
$e_1 \dots e_m \langle \mathsf{W}_q^i[y]v_1\rangle \langle \tau_q^i[y]v_1\rangle \langle \mathsf{R}_q[z_1]w_1\rangle \dots \langle \mathsf{R}_q[z_n]w_n\rangle \langle \mathsf{R}_q^{j'}[x]v_2\rangle \langle \mathsf{W}_p^j[x]v_3\rangle \langle \tau_p^j[x]v_3\rangle$ where $x \neq y$ and $p \neq q$ and $x \notin \{z_1 \dots z_n\}$, move the τ^i event to the end, giving $e_1 \dots e_m \langle \mathsf{W}_q^i[y]v_1\rangle \langle \mathsf{R}_q[z_1]w_1'\rangle \dots \langle \mathsf{R}_q[z_n]w_n'\rangle \langle \mathsf{R}_q^{j'}[x]v_2\rangle \langle \mathsf{W}_p^j[x]v_3\rangle \langle \tau_p^j[x]v_3\rangle \langle \tau_q^i[y]v_1\rangle$. This is a valid x86-TSO execution because if any of the $z_1 \dots z_n$ equal y, their value will be read from the write buffer. Any memory equivalent x86-SC execution must perform p's write to x before q's write to y. Respecting the program order of the instruction semantics, and this constraint requires us to move $\langle \mathsf{W}_p^j[x]v_3\rangle$ to before q's write to y. But now the read from x must read from this write event, whereas it could not have before (the write event cannot be pushed back before the one that it used to read from without violating memory equivalence). The case where $\langle \mathsf{W}_p^j[x]v_3\rangle$ is locked is similar, but $\langle \tau_q^i[y]v_1\rangle$ must appear after the unlock. To place the it after the entire locked instruction, we rely on the fact that there is no read from y in it, which is guaranteed by the instruction semantics: a locked event can only read and write a single address (x here). □

Due to space constraints, we do not present our axiomatic model, but refer the reader to our previous work [27]. We write $\prec_X$ and $\prec_E$ to indicate memory order and program order of an execution witness and event structure, respectively. The following lemma says there is a sequentially consistent counterpart for any valid execution that satisfies an axiomatic version of TRF.

Definition 9 (Axiomatic TR). *An execution witness X has an* axiomatic TR *is there are events that satisfy the following:* $p \neq q \wedge x \neq y \wedge (\langle \mathsf{W}_q[y]v_2\rangle \prec_E \langle \mathsf{R}_q[x]v_3\rangle) \wedge (\langle \mathsf{W}_p[x]v_1\rangle \prec_X \langle \mathsf{W}_q[y]v_2\rangle) \wedge (\langle \mathsf{W}_p[x]v_1\rangle \not\prec_X \langle \mathsf{R}_q[x]v_3\rangle) \wedge (\forall \langle \mathsf{W}_q[x]v_4\rangle \in E.\ (\langle \mathsf{W}_q[x]v_4\rangle \prec_E \langle \mathsf{R}_q[x]v_3\rangle) \Rightarrow (\langle \mathsf{W}_p[x]v_1\rangle \not\prec_X \langle \mathsf{W}_q[x]v_4\rangle))$.

Lemma 4. *Suppose that E is a well-formed event structure; that X is a valid execution for E; and that X has no axiomatic TR. Then, there exists a valid sequentially consistent execution X' with the same reads-from map, initial state, and write ordering as X.*

Proof sketch. We have mechanically verified this lemma in the HOL-4 proof assistant [25]. The proof comprises 4 phases. First, we construct a equivalent notion of valid execution that that is less strict about memory ordering dependencies on locked events. Second, we show that the subset of these valid executions where $\langle \mathsf{W}_p[x]v\rangle \prec_E \langle \mathsf{R}_p[y]w\rangle \Rightarrow \langle \mathsf{W}_p[x]v\rangle \prec_X \langle \mathsf{R}_p[y]w\rangle$ are exactly the sequentially consistent executions. Third, given X we construct a transitive memory ordering $\prec_{X'} = (\prec_X|_{writes} \cup \prec_E \cup X.\text{rfmap})^+$. Fourth, we complete $\prec_{X'}$ for locked accesses. For example, if $e_1 \prec_X e_2$ and e_1 and e_3 are in the same locked instruction, then we add $e_3 \prec_X e_2$ and enough other dependencies to maintain transitivity. We then prove that X' is a valid execution, relying on the axiomatic TRF assumption to show that it satisfies X's reads-from map.

Now we prove the soundness of TRF for memorySC.

Lemma 5. *If a program is not memorySC, then it has a triangular race.*

Proof sketch. Let $e \ldots$ be an x86-TSO trace with no memory equivalent x86-SC trace. By Theorem 3 from [27], there exists a valid execution X. By the contrapositive of Lemma 4 (noting that memory equivalence implies that the reads-from map and write ordering of corresponding execution witnesses is the same) there is an axiomatic TR. Consider the $\prec_E$- and $\prec_X$- closed prefix of the events mentioned in the axiomatic TR. This is a finite valid execution, so we can proceed by induction, trying to show that if a valid execution contains an axiomatic TR, then it has a triangular race. Remove from the execution the read event in the TR; if there is still a TR, we are done. Otherwise there are none, and by Lemma 4, we can build a sequential execution, and add the read back at the end (it could read from a different write, but that does not matter since we consider no subsequent events on q). Thus, we have an x86-SC execution (with the same write ordering and rfmap, save for i_3): $\ldots \langle \mathsf{W}_p^{i_1}[x]v_1\rangle \ldots \langle \mathsf{W}_q^{i_2}[y]v_2\rangle \ \ldots \langle \mathsf{R}_q^{i_3}[x]v_4\rangle$ where $x \neq y$ and $p \neq q$. Remove all events between i_1 and i_3 that are not on q (leaving any of i_1's fellow locked events in place, if any). The result is still an execution unless one of q's remaining reads read from one of the removed writes. However, in that case, the removed write is $\prec_X$ q's read, and so $i_1 \prec_X i_3$, and there is no TR. Any remaining lock, unlock or barrier events between i_2 and i_3 would have caused $i_1 \prec_X i_3$, so there are none. There are no remaining writes to x between i_1 and i_3, or else there not would have been a TR. Hence there are no reads from x, or else $i_1 \prec_X i_3$. Thus, we can move i_1, along with any potential fellow locked events to after i_3 (again relying on the instruction property mentioned in Lemma 3's proof if i_1 is locked).

References

1. S. V. Adve and K. Gharachorloo. Shared memory consistency models: A tutorial. *IEEE Computer*, 29(12):66–76, 1996.
2. S. V. Adve and M. D. Hill. A unified formalization of four shared-memory models. *IEEE Trans. Parallel Distrib. Syst.*, 4(6):613–624, 1993.
3. M. Ahamad, G. Neiger, J. E. Burns, P. Kohli, and P. W. Hutto. Causal memory: Definitions, implementation, and programming. *Distributed Computing*, 9(1), 1995.
4. *AMD64 Architecture Programmer's Manual (3 vols).* Advanced Micro Devices, Sept. 2007. rev. 3.14.
5. D. Aspinall and J. Ševčík. Formalising Java's data race free guarantee. In *Theorem Proving in Higher Order Logics*, volume 4732 of *LNCS*, pages 22–37. Springer, 2007.
6. H.-J. Boehm and S. V. Adve. Foundations of the C++ concurrency memory model. In *Proc. Prog. Language Design and Implementation*, pages 68–78. ACM, 2008.
7. G. Boudol and G. Petri. Relaxed memory models: An operational approach. In *Proc. Principles of Programming Languages*, pages 392–403. ACM, 2009.
8. S. Brookes. A semantics for concurrent separation logic. *Theor. Comput. Sci.*, 375(1-3):227–270, 2007.
9. S. Burckhardt and M. Musuvathi. Effective program verification for relaxed memory models. In *CAV*, volume 5123 of *LNCS*, pages 107–120. Springer, 2008.
10. E. Cohen and N. Schirmer. A better reduction theorem for store buffers. arXiv:0909.4637v1, 2009.
11. D. Dice. Java memory model concerns on Intel and AMD systems. `http://blogs.sun.com/dave/` (accessed 2009/12/13), Jan. 2008.

12. D. Dice. A race in LockSupport park() arising from weak memory models. http://blogs.sun.com/dave/ (accessed 2009/12/13), Nov. 2009.
13. The "double-checked locking is broken" declaration. http://www.cs.umd.edu/~pugh/java/memoryModel/DoubleCheckedLocking.html.
14. R. Friedman. *Consistency Conditions for Distributed Shared Memories*. PhD thesis, Technion: Israel Institute of Technology, 1994.
15. *Intel 64 and IA-32 Architectures Software Developer's Manual*. Intel Corporation, April (vol 1,2A,2B; rev.27), Feb. (vol.3A,3B; rev.26) 2008.
16. ISO/IEC 14882, programming languages - C++. WG21 n2800, Oct. 2008.
17. C. B. Jones. Accommodating interference in the formal design of concurrent object-based programs. *Form. Methods Syst. Des.*, 8(2):105–122, 1996.
18. JSR 133: Java memory model and thread specification revision. http://jcp.org/en/jsr/detail?id=133.
19. H. Kopetz and J. Reisinger. The non-blocking write protocol NBW: A solution to a real-time synchronisation problem. In *Real-Time Systems Symposium*, 1993.
20. C. Lameter. Effective synchronization on Linux/NUMA systems. In *Gelato Conference*. Silicon Graphics, Inc., 2005. http://www.lameter.com/gelato2005.pdf.
21. L. Lamport. How to make a multiprocessor computer that correctly executes multiprocess programs. *IEEE Trans. Computers*, 28(9):690–691, 1979.
22. Linux kernel mailing list, Nov. 1999. Subj.: spin_unlock optimization(i386).
23. V. Luchango. *Memory Consistency Models for High Performance Distributed Computing*. PhD thesis, Massachusetts Institute of Technology, 2001.
24. S. Meyers and A. Alexandrescu. C++ and the perils of double-checked locking. *Dr. Dobbs Journal*, July–August 2004.
25. M. Norrish and K. Slind. Hol-4. http://hol.sourceforge.net/.
26. P. W. O'Hearn. Resources, concurrency, and local reasoning. *Theor. Comput. Sci.*, 375(1-3):271–307, 2007.
27. S. Owens, S. Sarkar, and P. Sewell. A better x86 memory model: x86-TSO. In *TPHOLs*, volume 5674 of *LNCS*, pages 391–407. Springer, Aug. 2009.
28. S. Park and D. L. Dill. An executable specification and verifier for relaxed memory order. *IEEE Trans. Computers*, 48(2):227–235, 1999.
29. W. Pugh. The Java memory model is fatally flawed. *Concurrency - Practice and Experience*, 12(6):445–455, 2000.
30. V. A. Saraswat, R. Jagadeesan, M. M. Michael, and C. von Praun. A theory of memory models. In *Principles and Practice of Parallel Programming*, 2007.
31. S. Sarkar, P. Sewell, F. Zappa Nardelli, S. Owens, T. Ridge, T. Braibant, M. O. Myreen, and J. Alglave. The semantics of x86-CC multiprocessor machine code. In *Proc. Principles of Programming Languages*, pages 379–391. ACM, 2009.
32. D. C. Schmidt and T. Harrison. Double-checked locking. In *Pattern Languages of Program Design 3*. Addison-Wesley, 1997.
33. D. Shasha and M. Snir. Efficient and correct execution of parallel programs that share memory. *ACM Trans. Program. Lang. Syst.*, 10(2):282–312, 1988.
34. SPARC International, Inc. *The SPARC Architecture Manual: Version 8*. Prentice Hall, 1992.
35. J. Ševčík and D. Aspinall. On validity of program transformations in the Java memory model. In *ECOOP 2008*, volume 5142 of *LNCS*. Springer, July 2008.
36. W. Visser, K. Havelund, G. P. Brat, S. Park, and F. Lerda. Model checking programs. *Autom. Softw. Eng.*, 10(2):203–232, 2003.