Prism: A Principle-Based Sequential Memory Model for Microsoft Native Code Platforms

Working Draft, Version 0.8 — July 31, 2006

Herb Sutter, Developer Division (hsutter@microsoft.com)

Note to reviewers

This document is an in-progress snapshot. All input will be much appreciated. Points that are known to need further discussion include the following:

In general, what guarantees should hold in the presence of races? (See §4.2.) For example, subject to other rules, should we:

- Allow invented writes (incl. values) that cannot occur in an SC execution? (See Examples 3.1.3 through 3.1.7.) The answer in the current draft is no, except that we assume all loops containing only ordinary memory operations will terminate and so enable some motion across those loops.
- Allow write reordering for ordinary writes? The answer in the current draft is yes.

The current draft makes critical regions asymmetric: The start is a full fence, whereas the end is only a release. That is, code can move into a critical region from below but not from above. For the reason, see Example 3.2.5. There is a claim in [Boehm 2005a] that the difference can be measurable on some platforms (esp. P4), with some performance numbers.

Thanks! – Herb

Acknowledgments

Special thanks to Jim Hogg, David Callahan, and Hans Boehm for their extensive helpful input and reviews of drafts of this paper as it evolved. We would also like to thank the following people and teams who have graciously provided input and insights in the form of draft reviews and/or hallway and email discussion, all of which has improved the quality of this document:

- From Visual Studio: Carol Eidt, Kevin Frei, Kang Su Gatlin, Vinod Grover, Phil Lucido.
- From Microsoft Research: Tim Harris, Leslie Lamport, David Tarditi, Yuan Yu.
- From Live: Chris Brumme.
- From Office of the CTO: Burton Smith.
- From SQL: Slava Oks.
- From Windows: Neill Clift, Jonathan Morrison.

Contents

1 Overview

1.1 Motivation

> *A multiprocessing system on a single computer involves problems similar to those of a distributed system because of the unpredictable order in which certain events can occur. … We have found that problems often arise because people are not fully aware of this fact and its implications.* — [Lamport 1978]

> *Chip [and compiler] designers are under so much pressure to deliver ever-faster CPUs [and optimizations] that they'll risk changing the meaning of your program, and possibly break it, in order to make it run faster.* — [Sutter 2005]

> *I personally believe that for mainstream computing, weak memory models will never catch on with human developers. Human productivity and software reliability are more important than the increment of performance and scaling these models provide.* — [Brumme 2003]

The purpose of this paper is to specify a single memory model for all native code on Microsoft platforms, including the source code, compilers and tools, and supported hardware platforms for Windows XP/Vista (client and server), Windows Live, Windows Mobile (Smartphone and Pocket PC), and Xbox. Henceforth, native source code will rely only on the guarantees of this model, and compilers will emit instructions and barriers as necessary to ensure the model's guarantees hold on supported target hardware. It is intended that the .NET managed memory model be implementable in terms of this underlying native code model.

A *memory model* describes a) how memory reads and writes *may be performed by a processor* relative to their program order, and b) how writes by one processor *may become visible to other processors*. Both aspects affect the valid optimizations that can be performed by compilers, caches, and physical processors, and therefore a key role of the memory model is to define the tradeoff between programmability (stronger guarantees for programmers) and performance (greater flexibility for reordering program memory operations).

In the past, Microsoft has had no well-specified memory model for native code; the model has been whatever the particular combination of compiler(s) and run-time hardware happened to do, which is at best unreliable and nonportable. The result has been that teams write code that contains latent bugs (including potential security vulnerabilities) and/or explicit special-purpose cases for different hardware which increases testing and porting costs. Similar problems have been encountered and at least partly addressed for managed code in .NET [Hogg 2005, Morrison 2005, Morrison 2005a] and Java [Pugh 2000, JSR-133 2004]. Note that today programmers cannot consistently write correct lock-based code when compiler optimizations invent writes that do not appear in the source code and so cannot be correctly locked by the programmer (see Example 3.1.8).

This paper proposes a memory model for all Microsoft native code, including source code, compilers and tools, and hardware platforms, that we believe corrects some fundamental problems, notably that today we do not have sufficient guarantees to write correct lock-based code, and achieves two key goals: (1) It is easy to understand for programmers, and equivalent to sequential consistency for race-free code. (2) It is easy to specify clearly for implementers, and allows greater optimization flexibility than current "strong" models. In particular, a primary goal is to allow wide (but not maximum) latitude for local optimizations without global knowledge of the complete program.

There are many well-considered memory models available in the literature and in working implementations. This section describes the approach we chose for this paper and how it differs from other approaches. See also §5 for comparisons between this paper and specific related work.

1.2 The Elevator Speech Paragraph

The primary goals of this paper are (1) to support a *simple and teachable programming model*, and (2) allow *wide (but not maximum) latitude for local optimizations* that can be performed without global knowledge of the complete program, (3) that is the same across *all Microsoft native platform targets* (including tools and hardware). The approach is to guarantee sequential consistency for correctly synchronized programs, which means sequential consistency at checkpoints marked by special ("interlocked") operations, including locks and transaction boundaries in a transactional memory system.

1.3 Model Scope and Components

We consider a program that is compiled and executed on one or more processors sharing a single uniform memory. The memory model focuses on the following:

- **Program reads and writes:** Reads and writes of program objects specified in program source code.
- **Memory reads and writes:** Reads and writes of actual memory locations in the shared memory.
- **Transformations from program to memory reads and writes:** Transformations that the intermediate layers shown in Figure 1 are and are not allowed to perform, individually and in combination.

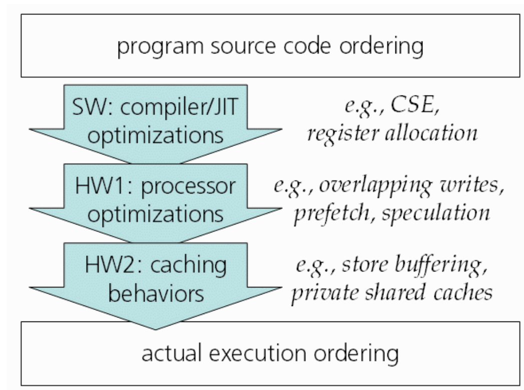

Figure 1: Levels of optimization/reordering

The memory model abstracts away the effects of intermediate implementation details of a given execution environment, such as NUMA architectures and cache structures. Compilers are required to maintain correct semantics for a given target processor by emitting the necessary instructions for that processor, including processor-specific memory ordering operations (e.g., load-with-acquire, fences).

1.4 Program vs. Hardware Focus

We believe that reasoning should start with the program, not with the hardware. This paper takes the approach of first coming up with a clear programming model based on simple abstractions, and then trying to specify the memory model in a way that permits implementations wide optimization latitude. (In particular, we believe that programming models that require programmers to know why and how to write explicit fences or barriers have proven too difficult for even expert programmers to use reliably. See for example [Win32prg 2006], which arose independently while we were writing this paper, as one current example of how even experienced programmers routinely encounter difficulty reasoning about even full fences, which are the simplest variety of barrier.)

The memory models in academic literature and commercial implementations are largely hardware-centric, not programmer-centric. Most papers begin with a list of specific optimizations they want to allow in the processor, cache, and other hardware, and then describe various "escape hatches" by which programmers can constrain the hardware's latitude and opt out of specific effects in specific ways. For example, [Adve 1995] Figure 8 lists a variety of such escape hatches in commercial systems, ranging from many flavors of explicit fences and memory barriers to special serialization instructions that require compilers to insert otherwise-redundant reads and writes in baroque ways to preserve intended program semantics. Not only are these escape hatches inconsistent and incompatible across platforms, but more seriously they have proven to be too difficult for even expert programmers to use reliably in practice, and so we do not consider such low-level mechanisms to be viable operations to expose in a programming model. (We also believe that starting with an explicit list of known optimizations may actually constrain hardware optimization opportunities, because hardwiring current techniques into the memory model is sometimes done at the expense of flexibility for future ideas.)

1.5 Uniform Treatment of Compiler and Hardware Optimizations

We believe that optimizations at all levels should be treated uniformly, and that the memory model need not and should not distinguish between reordering performed by the compiler and reordering performed by the processor, cache, or other hardware (see Figure 1). For example, successive reads from a variable **x** could be eliminated at level SW (e.g., by a compiler loading the value of **x** into a register) or at level HW2 (e.g., by loading the value of **x** into a processor-local cache), and because they have the same effect we conclude that for any given case if one is allowed then the other has to be allowed. Similarly, successive writes to different variables could be reordered at level SW by the compiler or at level HW1 by the processor, and again in any given case if one is allowed then the other has to be allowed.

Therefore, we will consider only program reads and writes and how they may be transformed to actual reads and writes of shared memory. In practice, the only thing that matters to the programmer is that the system behaves as though: (a) the order in which memory operations are actually performed is equivalent to some sequential execution according to program source order; and (b) each write is visible to all processors at the same time. This paper therefore focuses only on how to maintain that illusion, and does not mention specific caching strategies, barriers, etc., and thereby we also attempt to avoid overspecifying and overconstraining the allowed optimizations at all of these levels. Compilers conforming to this memory model are required to perform appropriate code generation to emit any hardware-specific instructions or directives required for correct execution on a particular architecture.

1.6 Sequential Consistency For Correctly Synchronized Programs

Fundamentally, programmers assume *sequential consistency* (SC) [Lamport 1979], where each processor performs its memory operations in program order, and only one processor at a time performs an operation on the monolithic shared memory. Two consequences are that: (a) each memory operation becomes instantaneously visible to all processors, and (b) in any execution, memory operations performed by different processors are interleaved.

This memory model is designed to preserve the expected sequentially consistent behavior for correctly synchronized programs. (This approach is similar to models like DRF0. [Adve 1990]) In particular, "correctly synchronized" means that every mutable object that is visible to multiple threads is either: a) correctly protected by a lock (or, in a transactional memory system, by an atomic block); or else b) marked as interlocked with atomic semantics. For a discussion of guarantees in the presence of races, see §4.2.

1.7 Atomic vs. Message Visibility

This memory model does not make the assumption that writes are atomically visible,[1] because we want this memory model to be applicable to clusters and other message-based environments. Therefore this model permits writes to be treated as asynchronous messages without violating sequential consistency and Rule R6. In other models, including the managed memory model, atomic visibility of writes is necessary to guarantee causality for Examples 3.4.1 to 3.4.3, which in this model are preserved by R6.

---

[1] Usually the term "atomic" is used to describe a read or write of a variable or memory location, and means that no intermediate value will be observable by other processors. Occasionally, as here, "atomic" is used to describe the visibility of a write, and means that a given write becomes visible to all other processors simultaneously (which this model does not require).

2 Model

2.1 Principles

The intent of this memory model is to enable a simple statement of the programmer's responsibility that developers can understand and use to reason reliably about the meaning of their programs, supported by an underlying model that is easy to specify clearly and implement correctly at all levels and that allows for maximal local optimizations without global knowledge of the whole program.

2.1.1 Correctness

> *But I also knew, and forgot, Hoare's dictum that premature optimization is the root of all evil in programming.* — [Knuth 1989]
>
> *It is far easier to make a correct program fast than it is to make a fast program correct.* — Various

The principal question is, "what do we teach programmers?" The answer has to be simple. We propose:

> **Principle P1: Enable a teachable programming model.** The programmer shall ensure that every object that is simultaneously visible to multiple threads and mutable is either: a) correctly protected by a lock (e.g., a traditional lock, or an atomic block in a transactional memory system); or else b) marked as interlocked with atomic, read-acquire, and write-release semantics. If these conditions are met, any execution shall be sequentially consistent with no races.

A programmer who follows P1 does not need to know anything further about this memory model, and can stop reading here. We believe that programming models more complex than P1 (e.g., requiring explicit fences) have been proven in practice to be too difficult for even experienced systems programmers to use reliably. Even with this simple model, the vast majority of programmers should use only part (a).

> **Principle P2: Enable a simple specification.** The memory model shall be built on the interlocked write as the key primitive that acts as a checkpoint to guarantee a set of ordinary writes shall become visible to another thread or processor that perform a corresponding interlocked read. An interlocked read or write can be used by performing it directly on an interlocked program object, or indirectly by acquiring or releasing a lock.

Informally, an interlocked read enters a critical section, and an interlocked write exits a critical section. A write event of interest is either a single interlocked write or a group of ordinary writes made visible by the next interlocked write by the same observer in program order, and the memory model guarantees sequential consistency for all write events in a correctly synchronized program while allowing unconstrained local optimization within a group.

2.1.2 Causality

> *The concept of time is fundamental to our way of thinking. It is derived from the more basic concept of the order in which events occur.* — [Lamport 1978]

The physical universe is an orderly system of events and observers based on causality, and causality is necessary for a system that humans can reason about reliably. In particular, in the physical universe:

> **Principle P3: Causality.** An observer shall not observe an event before any other event that causally precedes it (its cause or potential cause). All observers shall observe causally related events in the same order.

Even though relativistic and quantum effects introduce strange complications, they do not violate these simple guarantees. For example, time dilation can cause different observers to observe causally related events as happening at different times and speeds, but observers can never observe causally related events as happening in different orders. There is reordering latitude: Different observers can, and routinely do, observe causally unrelated events in different orders. Events also have reordering restrictions only with respect to observers and frames of reference that can observe them, and "private" unobserved events may experience an uncertainty that does not affect causality.

These ideas apply directly to shared-memory computing, which likewise is a system of events and observers, where some memory events are private and some are causally related to other events. Only in races, incomplete events can be observed with limited local distortion (for detailed discussion of this design point, see §4.2):

> **Principle P4: Events and races.** An event is an individual interlocked read or write, or a batch of ordinary reads and writes performed by the same observer between successive interlocked operations. Only in a race, an observer may observe a distorted batch whose writes appear to be performed in a different order, but not with different values, than in a sequentially consistent execution.

This memory model derives from the basic principles P1-P4, and like the physical universe it allows causally related events (writes) to become visible to different observers at different times but not in different orders, and even in races events may be distorted but not have values that never existed.

2.2 Rules

2.2.1 Correctness

First, we define the "as if" rule for race-free programs:

> **Rule R1: As if.** In a program that does not contain a race, any transformation that does not change the program's effects and cannot be detected by the program is valid.

Informally, if no valid program that relies only on the guarantees set out in this memory model can tell the difference, then there is no difference. For example, optimizers can eliminate unreachable code and dead code (ordinary writes that are never read).

Note that in this paper we do not consider reads and writes of unshared memory locations, which correspond to physical events that cannot be observed by other observers; these may be reordered subject to normal sequential optimization constraints (notably R1 applied to sequential code, including that sequential data and control dependencies are satisfied).

2.2.2 Ordinary and Interlocked Operations

A *program* always refers to the program source code. A *bitfield* is a type instance that is specified in the program to be represented in memory using a specific number of bits. An *object* (or, equivalently, *variable*) is a single type instance declared in the program that is not a bitfield, or any sequence of bitfields declared contiguously in the program. Informally, an object is any single object or variable expressed in the source code, except that adjacent bitfields are considered to be a single object. An interlocked *object* is an object that is specially designated as such by the programmer. A *program read or write* is a write that appears in the program and is performed on a specific object.

An *observer* is a sequential portion of a program (e.g., a thread) whose program reads and writes have a total ordering according to the program's source code. [2] Informally, an observer is a piece of sequential code with a single consistent frame of reference.[3] A *shared object* is an object that is declared interlocked or that can be the target of program reads or writes performed by more than one observer; conservatively, every object is considered shared unless it is not interlocked and can be proved to be accessible to only one observer (e.g., through language-specific programmer annotations, or through escape analysis or other deduction).

A *memory location* is an atomically updatable region of memory. A *shared memory location* is a memory location that is visible to more than one observer. Every object is stored in one or more memory locations, and no memory location stores any parts of two different objects.

An interlocked *memory location* is a memory location that is used to store an interlocked object. An interlocked *read or write* is a read or write of an interlocked memory location, and is generated from a single program read or write of an interlocked object. Per P1, we require:

> **Rule R2 (=P1.b): Interlocked atomicity.** An interlocked object is stored in exactly one shared memory location. Corollaries: Every interlocked read and write is atomic. An interlocked object is suitable for use with atomic operations including compare-and-swap (**a_cas**) and exchange (**a_swap**).

An *ordinary read or write* is a read or write of a non-interlocked shared memory location, and is generated from a single program read or write of a shared object. A *batch* of ordinary reads and writes is a sequence of ordinary reads and/or writes executed by the same observer with no intervening interlocked writes in program order. Every batch shall be finite, followed by either the next interlocked operation or the end of that observer's execution; in particular, a loop consisting only of ordinary operations is assumed to be finite (see Example 3.1.3).

We require that interlocked reads and writes behave as though each interlocked operation directly accesses main memory, and supports the requirements of P1:

> **Rule R3 (=P1): Interlocked reads and writes.** Interlocked reads and writes by the same observer cannot be reordered relative to one another and shall be performed in program order. An interlocked read additionally precedes all ordinary reads and writes by the same observer that follow it in program order ("acquire semantics"), and shall not be eliminated unless it is immediately followed by another interlocked read or write of the same memory location, or as described in R4. An interlocked write additionally follows all ordinary and interlocked reads and writes by the same observer that precede it in program order ("release semantics"), and shall not be eliminated unless it is immediately followed by another interlocked write to the same memory location, or as described in R4.

A *lock* is used to ensure mutual exclusion to a set of shared objects. In this paper, a lock refers to either a traditional lock acquired and released explicitly by the programmer, or to a system-generated lock surrounding critical sections that are acquired and released automatically in a transactional memory system (e.g., to implement begin, commit, retry, and rollback operations; see also Examples 3.5.1 and 3.5.2). A

[2] Examples: A thread is an observer. Any subset of the code in the same thread is an observer. The set of all fibers on a thread is an observer because the fibers are scheduled cooperatively (during any interval wherein the set of fibers sharing the thread does not change due to migration of a fiber from or to another thread). An individual fiber is an observer.

[3] The term "observer" does not imply that it does not perform writes — by definition, it can. We adopt this term from the domain of physics as a neutral term for generality, in order to avoid implying that it is necessarily a thread, a fiber, a process, or any other particular system-specific entity.

lock can be held by a single observer at a time; an observer *holds* a lock after acquiring it until releasing it. A lock *acquire* operation blocks indefinitely until the observer successfully acquires the lock, and a lock *try-acquire* operation returns without blocking indefinitely and reports whether or not the lock was successfully acquired. A lock can be *released* by the observer that acquired it, after which another observer can acquire the lock. Lock implementations are permitted to select among different semantics compatible with the foregoing; in particular, a given type of lock may or may not permit nested acquisition of the same lock by an observer who already holds it, and if so then a release may release only the last acquisition or all existing acquisitions. Per P2, we require:

> **Rule R4 (=P2): Interlocked locks.** Each lock is implemented using a distinct interlocked control variable. A lock acquire operation performs an interlocked read and then an interlocked write on the lock's control variable. A lock try-acquire operation performs an interlocked read, and if successful then also an interlocked write, on the lock's control variable. A lock release operation performs an interlocked write on the lock's control variable. A lock acquire or release can be eliminated only when a lock release is immediately followed by a lock acquire for the same lock; then the pair can be eliminated together.

Note that acquiring a lock is required to perform an interlocked write, not just a read, because the availability of a try-acquire operation means that a lock acquire on another thread is an observable event detectable by other threads (see Example 3.2.5 and [Boehm 2005a]).

The programmer cannot apply P1 and write the correct synchronization if he does not control all writes to shared variables. Therefore P1 implies that the system cannot invent writes to shared variables. Further, programming languages must also be able to create additional data, such as vptrs, that are associated with program-declared objects, but the programmer cannot perform correct locking if he is not able to see where all writes to the conceptual object (including additional hidden data) can occur. Therefore we require:

> **Rule R5 (=P1): Translating program writes.** Every ordinary or interlocked write shall correspond to a valid program write and write the value that would be written by that program write, such that the set of all such program writes is possible in some execution wherein all are performed in program order. A program write to a shared object **s** shall not result in performing ordinary or interlocked writes to any memory location holding a program object other than **s**. If the system creates a hidden object **h** associated with a specific a shared program object **s**, then **h** is part of **s**, a read (or write) of a memory location holding a part of **h** can be generated adjacent to a read (or write) of a memory location holding a part of **s**, and reads and writes of **h** must obey all rules pertaining to reads and writes of **s** (including volatility).

Note that the first sentence of R5 implies that, even in a program that contains a race, no observer shall read a value that was not written as part of some valid program write in a sequentially consistent execution. A program with races can still see unexpected values due to word tearing and similar effects, but it shall not see values of individual memory locations (which by definition are atomically updatable) that would not have been written by some observer in a sequentially consistent execution.

The second sentence of R5 implies that: (a) a program write to an object **a** may not create an ordinary or interlocked write to the bits of any other object **b** (see Example 3.1.2); and (b) an ordinary or interlocked read or write cannot be invented that does not occur as part of a valid program read or write (see Example 3.1.8, and see also Example 3.1.9).

The third sentence implies that: (c) **h** is interlocked iff **o** is interlocked; and (d) the system may not create a read or write of **h** where no program read or write of **o** appears. Once created, these reads and writes of **h** can be reordered subject to R3 and R4.

2.2.3 Causality

For the purpose of P3, an *event* of interest is an individual interlocked read or write, or a batch of ordinary reads and writes. An event **a** is *observed* by the observer that performs **a** immediately upon completion of **a**, and by a different observer when the value(s) written by **a** are available to be read by that observer. Note that in a correctly synchronized program all writes performed in the same event become visible atomically with respect to another observer.

We define a *causally-precedes* relation $\xrightarrow{c}$ to define a partial ordering of events according to which events could causally affect other events. The relation $\xrightarrow{c}$ on the events of a program execution is the smallest relation satisfying the following conditions: (1) For events **a** and **b** performed by the same observer, if **a** precedes **b** in program order then **a** $\xrightarrow{c}$ **b**. (2) For events **a** and **b**, if **b** observes **a** then **a** $\xrightarrow{c}$ **b**. (3) For events **a** and **b** that write different values to the same memory location **m**, and an observer **o** that observes both **a** and **b** and then in program order reads **m**, if **o** reads the value written by **b**, then **a** $\xrightarrow{c}$ **b**. (4) For events **a** and **b**, if some observer performs **a** and then observes **b**, then **a** $\xrightarrow{c}$ **b**. (5) If **a** $\xrightarrow{c}$ **b** and **b** $\xrightarrow{c}$ **c** then **a** $\xrightarrow{c}$ **c**.

Two events **a** and **b** are *causally related* iff **a** $\xrightarrow{c}$ **b** or **b** $\xrightarrow{c}$ **a**; otherwise, they are *causally unrelated* (alternatively, *concurrent*). Note that **a** $\not\xrightarrow{c}$ **a** for any event **a**. Therefore $\xrightarrow{c}$ is an irreflexive partial ordering on the set of events in the program.

Note: Other work defines relations that are closely related to *causally-precedes* as defined above. For example, [Lamport 1978], [Adve 1990], [Manson 2005], and [Arvind 2006] define similar happens-before relations for Lamport clocks, the DRF0 memory model, the happens-before relation for the Java memory model, and the is-before relation for serializability and store atomicity, respectively, and the specific wording above borrows heavily from that in [Lamport 1978]. See §5 of this paper for a discussion of differences with other formulations.

We can now adopt P3 and P4 directly as an additional rule that further constrains the reordering and visibility of events:

> **Rule R6 (=P3,P4): Causality.** An observer shall not observe an event before any other event that causally precedes it (its cause or potential cause). All observers shall observe causally related events in the order defined by $\xrightarrow{c}$. When an observer performs a read of a memory location, the result is the value written by the event most recently observed that performed a write to that location. Only in a race, an observer may observe a distorted batch whose writes appear to be performed in a different order, but not with different values, than in a sequentially consistent execution.

A *race* exists when, for any shared object **s**, there are two causally unrelated events **a** and **b** where **a** performs an ordinary write to **s** and **b** performs a read or write of **s**.

Finally, per P1.a, the only rule that places a requirement on the programmer is that the programmer eliminate races using locks (or, alternatively, by designating a shared object to be interlocked):

> **Rule R7 (=P1.a): Correct locking.** For every shared object **s**, if any observer can perform an ordinary write to any part of **s** and a different observer can perform an ordinary read or ordinary write of any part of **s**, then the program shall have one lock associated with that object and both observers shall perform their actions only while holding that lock.

Note that, because no other rule prevents it, by Rule 1 an implementation is permitted to freely apply local optimizations that reorder, create, and remove ordinary reads and writes performed by the same observer, subject only to the constraints that they not move ahead of an interlocked read, move after an interlocked write, or violate normal sequential data and control dependencies. Global knowledge of the whole program and other threads is not required to perform such optimizations.

2.2.4 Language Semantics

Programming languages do not always precisely define the exact ordering of memory operations on program variables. For example, this often arises when a single expression in the language automatically generates multiple calls to other functions. Where languages do permit latitude, the compiler must translate the program as conservatively as possible to avoid performing an interlocked read later, or an interlocked write earlier, than necessary. (See also Example 3.8.1.)

> **Rule R8: Conservative interpretation of language semantics.** Given a set M of memory operations performed by the same observer that corresponds to a particular program expression or statement, where the programming language permits latitude in compiler translation of the ordering of operations in M: The compiler shall translate the program so that every interlocked read in M precedes all possible ordinary reads and writes in M, and every interlocked write in M follows all possible ordinary reads and writes in M, to the extent permitted by language semantics.

3 Examples

In these examples, unless otherwise noted, all initial values are **0**, all variables whose names start with **r** are unshared (representing unshared memory locations, e.g., in local variables, registers, and caches), and all other variables are ordinary shared variables (not interlocked). Where possible, we mention the source where we first encountered the example.

3.1 Ordinary Reads and Writes

3.1.1 Basic Reordering

This example was supplied by Kang Su Gatlin.

Consider the following code, where initially **x = y = 0** and threads **T1** and **T2** are the only observers manipulating **x** and **y**:

```
// thread T1
x = 1;              // 1
y = 1;              // 2
```

```
// thread T2
if( y == 1 )        // 3
  --x;              // 4
```

This code contains a race because both **x** and **y** can be concurrently read and written and there is no synchronization. How the race can manifest for **y** is obvious; it can manifest for **x** because lines 1 and 2 can be reordered.

Incidentally, note that even if **x** and **y** have type **int**, the programmer cannot rely on program writes to actually be atomic (e.g., **int**s are not guaranteed to be aligned), and in general under this memory model atomicity is not an inherent property of any type, not even **char**, unless the variable is marked interlocked.

P1 tells the programmer how to remove the race. There are two ways, either of which is sufficient:

- **Use a lock:** If both code fragments are protected using the same traditional lock or protected in an **atomic { … }** block, there is no race because of mutual exclusion.
- **Make y interlocked:** If **y** is interlocked, then there is no race on **y** because it is atomically updatable, and there is no race on **x** because $\mathbf{1} \overset{c}{\hookrightarrow} \mathbf{2} \overset{c}{\hookrightarrow} \mathbf{4}$.

3.1.2 Masking and Object Layout

This example was supplied by Intel (see [Boehm 2006a]). Consider the following code, assuming 8-bit **char**s and that **S**'s members are laid out contiguously so that **sizeof(S) == 4**:

```
// program source
struct S {
  char a;
  int b : 9;    // note: bitfields
  int c : 7;
  char d;
};
S s;
s.b = 1;
```

Consider the transformation that reads **s** in a single operation, writes only to the bits corresponding to **b**, and writes **s** back:

```
// transformation
struct S {
  char a;
  int b : 9;
  int c : 7;
  char d;
};
S s;
char tmp[4];
memcpy( &tmp[0], &s, 4 );
... in tmp, write to only the bits corresponding to b ...
memcpy( &s, &tmp[0], 4 );
```

If **s** is not a shared object, then this transformation is legal. If **s** is a shared object, this transformation is illegal by R5 because it creates ordinary writes to **a** and **d** that are not present in the program source. (The creation of an ordinary write to the bits of **c** is valid because **b** and **c** are contiguous bitfields and are therefore the same object.)

3.1.3 Nonterminating Loops

This example was supplied by [Boehm 2006a]. Consider the following code, which contains no synchronization (locks or interlocked variables):

```
// program source
for( T* p = q; p != 0; p = p->next ) { ... }
x = 42;
```

Can any of the write to **x** be moved ahead of the loop? In particular, if the loop is potentially nonterminating, could an observer on another thread see a value for **x** even when the assignment to **x** could never be executed according to program order?

The answer is yes. All of the code is part of the same batch, and R6 permits the reordering of writes within a batch. R5 does not prohibit moving a valid write within a batch, and the write **x = 42;** must occur because the batch is required to be finite (if the loop is infinite then this code violates the requirement that a batch must be finite).

In particular, this choice makes it illegal for surrounding/calling code to take a lock protecting **x** if and only if the loop will terminate, as in the following example provided by Carol Eidt:

```
if( ConsultOracleWillLoopTerminate() ) { lock(); }       // take lock protecting x iff necessary?
for( ... ) { ... }
x = 1;
if( ConsultOracleWillLoopTerminate() ) { unlock(); }     // release lock protecting x iff necessary?
```

If any other observer reads or writes **x**, whether under a lock or not, then the above code contains a race because a write to **x** can occur without holding the lock.

3.1.4 Merging Successive Loops

This motivation for this example was provided by David Callahan. Consider the following loops, where there are no interlocked operations:

```
// program source
for( i = 0; i < max; ++i ) { c[i] = a[i] + b[i]; }
for( i = 0; i < max; ++i ) { d[i] = a[i] * b[i]; }
for( i = 0; i < max; ++i ) { e[i] = sqrt( a[i]*a[i] + b[i]*b[i] ); }
```

The question is, if the bodies are free of other side effects, can an optimizer merge the loops and transform this into the following (e.g., for better locality on the shared arrays **a** and **b**)?

```
// transformation
for( i = 0; i < max; ++i ) {
  c[i] = a[i] + b[i];
  d[i] = a[i] * b[i];
  e[i] = sqrt( a[i]*a[i] + b[i]*b[i] );
}
```

The answer is yes. All of the code is part of the same batch, and R6 permits the reordering of writes within a batch. R5 does not prohibit moving a valid write within a batch, and the writes must occur because the batch is required to be finite (if the loop is infinite then this code violates the requirement that a batch must be finite).

3.1.5 Inverting Nested Loops

Consider the following loops, where there are no interlocked operations:

```
// program source
for( j = 0; j < jmax; ++j ) {
  for( i = 0; i < imax; ++i ) {
    b[i] += a[i][j] * 2;
  }
}
```

The question is, if the bodies are free of other side effects, can an optimizer rearrange the loops and transform this into the following (e.g., for better locality on the shared arrays **a** and **b**)?

```
// transformation
for( i = 0; i < imax; ++i ) {
  for( j = 0; j < jmax; ++j ) {
    b[i] += a[i][j] * 2;
  }
}
```

The answer is yes. All of the code is part of the same batch, and R6 permits the reordering of writes within a batch. R5 does not prohibit moving a valid write within a batch, and the writes must occur because the batch is required to be finite (if the loop is infinite then this code violates the requirement that a batch must be finite).

3.1.6 Condition-Write

Consider the following code, where **x** is an ordinary shared variable, as usual with initial value **0**:

```
// program source (correct, no race)
if( cond )
  x = 42;
```

Assuming this code contains no interlocked operations, may this be transformed as follows (e.g., if the compiler or profile-guided optimizer determines that **cond** is expected to be true):

```
// transformation (incorrect, creates ordinary write that never occurred)
x = 42;
if( !cond )
  x = 0;
```

The answer is no. The transformation is disallowed by R5 for two reasons: (1) The write **x = 42;** does not correspond to a program write and so cannot be invented. (2) If **cond** is false the value **0** would be written, which is a valid program write in any sequentially consistent execution of the program code.

See also Example 3.1.7.

3.1.7 Write-Condition-Write

Consider the following code, where **x** is an ordinary shared variable:

```
// program source
x = 0;
if( cond )
  x = 42;
```

Assuming this code contains no interlocked operations, may this be transformed as follows (e.g., if the compiler or profile-guided optimizer determines that **cond** is expected to be true):

```
// transformation (incorrect, creates a value that never occurrs in the program)
x = 42;
if( !cond )
  x = 0;
```

The answer is no. The transformation is disallowed by R5 because if **cond** is false the value **42** would be written, and conversely if **cond** is true the value **0** would be written, neither of which is possible in any sequentially consistent execution of the program code.

See also Example 3.1.6.

3.1.8 Register Allocation Without Dirty Check

This example was supplied by Kevin Frei from actual code, and based on a similar example in [Boehm 2006a]. Consider the following code, where object **x** is protected by a lock:

```
// program source (correct, no race)
if( cond )
  lock();                    // more generally, "initialize resource"
for( ... )
  if( cond && other_cond ) {
```

```
    ++x;                  // more generally, "use resource"
  }
if( cond )
  unlock();               // more generally, "release resource"
```

This pattern arises in a function that optionally performs additional work (here, optional work that involves updating **x**), where the flag used to control whether the extra work should be done (here **cond**) is typically passed as a parameter to the function. In this case, the programmer knows the lock is only needed if the optional work will be done and **x** could be updated, so the lock is only taken if the optional additional work involving **x** will actually be performed.

If **x** is not a shared object, then this may be legally transformed as follows to enregister **x**:

```
// transformation (incorrect, creates potential race)
if( cond )
  lock();
r1 = x;
for( ... )
  if( cond && other_cond ) {
    ++r1;
  }
x = r1;
if( cond )
  unlock();
```

But if **x** is a shared object, this transformation is illegal by R5 because it can create an ordinary write that is not present in the program source, for example whenever **cond** is false.

Example 3.1.9 shows how to change this transformation to make it legal.

3.1.9 Register Allocation With Dirty Check

Consider again the original code in Example 3.1.8:

```
// program source (correct, no race)
if( cond )
  lock();
for( ... )
  if( cond && other_cond ) {
    ++x;                  // more generally, "use resource"
  }
if( cond )
  unlock();
```

If **x** is not a shared object, then this may be legally transformed as follows to enregister **x**:

```
// transformation (correct, no race)
if( cond )
  lock();
r1 = x;
bDirty = false;
for( ... )
  if( cond && other_cond ) {
```

```
    ++r1;
    bDirty = true;
  }
if( bDirty)
  x = r1;

if( cond )
  unlock();
```

If **x** is a shared object, this transformation does not violate R5 the way that Example 3.1.8 does, because here the transformed code writes the register back to **x** only if there is a program write to **x**. Therefore this transformation amounts to combining all the loop's ordinary writes to **x** and moving them after the loop, and it is legal if and only if that combination and motion is legal.

3.1.10 Eliminating a Redundant Read

Consider this example, where **x** is not interlocked:

```
// program source
x = 2;          // a
r1 = x;         // b
```

Is it legal to transform this as follows to eliminate the redundant read of **x**?

```
// transformation (correct, if x is not interlocked)
x = 2;          // a
r1 = 2;         // b'
```

This is legal, because it obeys R1. (Note that R5 only forbids the invention of reads and writes not visible in the source code, not their elision when doing so does not introduce new behaviors.) Even in a race, this local transformation only reduces the set of possible behaviors, by making **b'** be unable to see a racing update on another thread, which it cannot rely on seeing anyway. Once this transformation is performed, line **b'** could further be reordered ahead of line **a**.

Note that if **x** were interlocked, this transformation would be disallowed by R3, which does not permit this elision of an interlocked read.

3.2 Interlocked Reads and Writes

3.2.1 Interlocked Read, Interlocked Write

Consider this example, where **x** and **y** are interlocked:

```
r1 = x;     // interlocked read
y = r2;     // interlocked write
```

By either R3 or R4, these operations may not be reordered.

3.2.2 Interlocked Write, Interlocked Read

Consider this example, where **x** and **y** are interlocked:

```
x = r1;     // interlocked write
r2 = y;     // interlocked read
```

These operations may be reordered. Neither R3 nor R4 restricts this transformation.

3.2.3 Lock Coarsening

Consider this example, where **a_lock** is a lock and **x** and **y** are shared:

```
// program source
a_lock.lock();
x = 42;
a_lock.unlock();          // 1
y = 53;
a_lock.lock();            // 2
x = 64;
a_lock.unlock();
```

First, we can move the assignment to **y** into either critical section, which leaves lines 1 and 2 adjacent. Second, by R4, a lock release immediately followed by a lock acquire of the same lock can be elided as a pair, and so lines 1 and 2 can be elided. Third, we can then eliminate the redundant assignment to **x**. The result is that the following is a legal transformation:

```
// transformation (correct)
a_lock.lock();
y = 53;
x = 64;
a_lock.unlock();
```

3.2.4 Locks As Barriers

Consider this example, supplied by Hans Boehm, where **x** and **y** may or may not be interlocked, but if not interlocked assume they are atomically updated:

```
// thread T1
x = 1;                    // a
lock(l1); unlock(l1);
lock(l1); unlock(l1);
r1 = y;                   // b
```

```
// thread T2
y = 1;                    // c
lock(l2); unlock(l2);
lock(l2); unlock(l2);
r2 = x;                   // d
```

The question is: Can **r1 == r2 == 0**?

If **x** and **y** are interlocked, the answer is no, because in that case if **r1 == 0** then **b** $\xrightarrow{c}$ **c** and so **a** $\xrightarrow{c}$ **d** and so **r2 == 1**.

If **x** and **y** are not interlocked, the program is incorrectly synchronized and contains a race, violating R7, and the answer is yes. First, note that the successive locks in each thread can be combined by R4 (see also Example 3.2.3). If the locks are not combined, no local reordering of lines **a** and **b** or lines **c** and **d** is possible, but if the locks are combined, we have:

```
// thread T1 (valid transformation)
x = 1;                    // a
lock(l1);
        unlock(l1);
r1 = y;                   // b
```

```
// thread T2 (valid transformation)
y = 1;                    // c
lock(l2);
        unlock(l2);
r2 = x;                   // d
```

Now in each thread the commented lines can move into the critical section and then be reordered.

```
// thread T1 (valid if x and y not
interlocked)
lock(l1);
r1 = y;            // b
x = 1;             // a
       unlock(l1);
```

```
// thread T2 (valid if x and y not
interlocked)
lock(l2);
r2 = x;            // d
y = 1;             // c
       unlock(l2);
```

3.2.5 Lock Acquire As Publishing Events

The following example is adapted from the example for Theorem 6.1 in [Boehm 2005a]. This code demonstrates why lock acquisition can be a "publishing" event if there is a **try_lock** operation that can make lock acquisition observable on another thread. Here **v1** is noninterlocked:

```
// thread T1
v1 = 1;            // a
lock(l1);          // b1=read, b2=write
```

```
// thread T2
while( try_lock(l1) ) { // c1=read, c2=write
  unlock( l1 );
}
// here try_lock failed, so now T1 holds l1
r2 = v1;           // d
```

This code does not contain a race, and the result is **r2 == 1**. Event **b2** performs an interlocked write, and each execution of event **c1** performs an interlocked read (after the last of which T2 also performs an interlocked write **c2**), of the same lock control variable. T2 waits to observe event **b2**, and after the last execution of event **c1** observes event **b2**, it performs event **c2**. So $\mathbf{a} \overset{c}{\hookrightarrow} \mathbf{c1}_{\mathbf{all}} \overset{c}{\hookrightarrow} \mathbf{c2} \overset{c}{\hookrightarrow} \mathbf{d}$, and in turn $\mathbf{a} \overset{c}{\hookrightarrow} \mathbf{d}$ implies **r2 == 1**.

In pthreads, this code likewise does not contain a race. As Boehm notes, speaking of pthreads rules: "Although few would defend this as good, or even reasonable, programming style, it is data-race-free. T2 can only read **v1** after the loop terminates. This can only happen once T1 acquires the lock."

In the Java memory model, this program *does* contain a race; see also §5.2.

3.3 Publishing Idioms

These examples are variants of the general case where one observer creates (or in isolation mutates) shared objects and then makes them visible to the rest of the system with an atomic operation, which in this memory model means an interlocked write.

3.3.1 Create and Publish New Object

Consider the following code, where **p** is an interlocked pointer to an **ImmutableObject**:

```
// thread T1 (publisher)
p = new ImmutableObject();
```

```
// threads T2..n (readers)
DoSomethingWith( p );
```

This program is correct and race-free because **p** is interlocked and after construction ***p** is shared but immutable. Note that R8 requires that in line 1 the write to **p** must occur last even if the language allows flexibility in the ordering of line 1's subactions. (See also Example 3.8.1.) Therefore readers of a non-null **p** see the fully constructed object. (If the object is mutable, further locking may be required, but the code above is sufficient for this example of constructing an object that is thereafter immutable.)

3.3.2 Create and Publish Queue Items

This example is taken from [Adve 1995] Figure 1. Consider the following code, where thread T1 builds up a singly-linked list of tasks and then publishes the list via an interlocked **head** pointer, and other threads wait for the publishing to be complete and then each take one queue item from the queue (using a lock to serialize the readers with respect to each other). Initially all pointers are null and all integers are **0**, and **head** is the publishing variable:

```
// thread T1 (publisher)
while( there are more tasks ) {
  task = GetFromFreeList();    // read task
  task->data = ...;            // set values
  ... insert task in queue ...
}
head = head of task queue;
```

```
// threads T2..n (readers)
while( myTask == null ) {
  lock_list();
  if( head != null ) {
    myTask = head;             // take task
    head = head->next;         // remove it
  }
  unlock_list();
}
... = myTask->data;
```

This program is correct and race-free. Because **head** is interlocked, all the work in T1 must be visible to any other thread that sees a non-null value of **head**. After T1 publishes the list, it is protected by a lock.

3.3.3 Internally Versioned Objects Using Immutable Slices

Consider the following **Versioned** class whose instances are safe to use without locking because state is never updated in place, but rather internal state is maintained in immutable slices accessed via an interlocked **pState** pointer:

```
// program source (correct, no race)
class Versioned {
private:
  State *interlocked pState;   // pointer to current immutable "slice"/"version" of this object's state
  ...
  void EveryReader() {         // every reader method of this class must follow the pattern that
    State* pOld = pState;      // "taking a local copy of the state pointer" must come first
    ... use pOld, not pState...  // and then only pOld is used to access the object's state
  }
  void EveryMutator() {        // every mutator method of this class must follow the pattern that
    while( true ) {
      State* pOld = pState;    // like every method it first takes a copy of the state pointer
      State* pNew = new State; // and then creates a new State with new values, and then
      ... set values of *pNew from values of *pOld and other sources, but not pState ...
      if( a_cas( &pState, pOld, pNew ) ) {
        break;                 // finally overwrites pState to publish the new state
      } else {
        ... undo work and delete pNew ...
      }
    }
  }
}
```

This program is correct and race-free. Because **pState** is interlocked, all the work to initialize a new slice must be visible to any other thread that sees the result of the new pointer stored with **a_cas**.

Note that the above code elides the details of memory management to free old slices when they are no longer referenced by any readers.

3.3.4 Double-Checked Locking (DCL)

Consider the classic Double-Checked Locking pattern, where the first thread to call **GetPointer** lazily initializes the singleton **T** object pointed to by the interlocked pointer **p**:

```
// program source (correct, no race)
T* GetPointer() {
  if( p == 0 ) {                    // 1: interlocked read (p)
    p_lock.lock();                  // 2: interlocked read (p_lock.var)
    if( p == 0 ) {                  // 3: interlocked read (p)
      p = new T;                    // 4: ordinary reads/writes + interlocked write (p)
    }
    p_lock.unlock();                // 5: interlocked write (p_lock.var)
  }

  return p;                         // 6: interlocked read
}
```

This code is correct and race-free:

- By R3 and R4, lines 1, 2, and 3 cannot be reordered and must precede 4, 5, and 6.
- By R8, in line 4 the ordinary reads/writes are performed first (and may be reordered with respect to each other) before the interlocked write to **p**. This is necessary to ensure that another thread executing lines 1 and 6 will not see a partly-constructed object.
- By R4, line 4 must precede lines 5 and 6.

Note that lines 5 and 6 can be reordered. (See also Example 3.2.2.)

See also Example 3.8.2 for an alternative equivalent to DCL for initialization that does not require traditional locks.

3.4 Causality

3.4.1 Canonical Example

This example comes from many sources, including [Adve 1995] and Hans Boehm. Consider the following code, notably where each thread runs on a different processor or core. In this example, **x** and **y** are interlocked, and initially **x = y = 0**:

```
// thread T1
x = 1;          // event a
```

```
// thread T2
if( x == 1 )   // observe a
  y = 1;       // event b
```

```
// thread T3
if( y == 1 )           // observe b
  assert( x == 1 );    // observe a
```

The assertion is required to succeed under this memory model because $\mathbf{a} \xrightarrow{c} \mathbf{b}$ and so T3 cannot observe **b** (**y == 1**) without also observing **a** (**x == 1**).

3.4.2 Initialization (I)

In [Boehm 2006c], Hans Boehm provided the following example, where **p** and **q** are interlocked and initially **p = q = null**:

```
// thread T1
construct X;            // a
p = reference to X;     // b
```

```
// thread T2
r2 = p;                 // c
q = r2;                 // d
```

```
// thread T3
r3 = q;                 // e
if( r3 != null ) {
  q.foo();              // f
}
```

If T3 sees **r3 != null**, then **q** must refer to a fully-constructed **X** object. Here **r3 != null** implies **d** $\overset{c}{\hookrightarrow}$ **e**, **r2 != null**, and **c** $\overset{c}{\hookrightarrow}$ **d**, therefore **a** $\overset{c}{\hookrightarrow}$ **b** $\overset{c}{\hookrightarrow}$ **e** $\overset{c}{\hookrightarrow}$ **f**. By R3, all ordinary writes performed by **X**'s constructor (which by R5 include compiler-generated writes to set up the vtable, the vptr member, and initonly or literal members), must be visible to **f**. For example, if **X** is a type with immutable instances like **System::String**, T3 must not be able to observe the string's value changing asynchronously. See also Example 3.8.1.

3.4.3 Initialization (II)

Similarly to Example 3.4.2, consider this code (adapted from [Boehm 2006c]), where **p_initialized** and **q_initialized** are interlocked:

```
// thread T1
p = new X;              // a
p_initialized = true;   // b
```

```
// thread T2
while( !p_initialized )// c
  { ; }
q = new Y( p );         // d
q_initialized = true;   // e
```

```
// thread T3
while( !q_initialized )// f
  { ; }
access *p via *q        // g
```

If T3 sees **q_initialized == true**, then **q** must refer to a fully-constructed **Y** object which in turn refers to a fully-constructed **X** object. Here **q_initialized == true** in line **f** implies **e** $\overset{c}{\hookrightarrow}$ **f**, and since also by construction **b** $\overset{c}{\hookrightarrow}$ **d**, therefore **a** $\overset{c}{\hookrightarrow}$ **b** $\overset{c}{\hookrightarrow}$ **f** $\overset{c}{\hookrightarrow}$ **g**. By R3, all ordinary writes performed by **X**'s and **Y**'s constructors (which by R5 include compiler-generated writes to set up the vtable, the vptr member, and initonly or literal members), must be visible to **g**.

3.4.4 Hand-Rolled Locks

Boehm provides the following example, where initially **x = y = lck = 0**, and **lck** is interlocked:

```
// thread T1
x = 17;
lck = 1;        // a
```

```
// thread T2
while( lck == 0 ) { ; }
r1 = x;
y = r1;
lck = 2;        // b
```

```
// thread T3
while( lck < 2 ) { ; }
r2 = y;         // c
```

By R6, **a** $\overset{c}{\hookrightarrow}$ **b** $\overset{c}{\hookrightarrow}$ **c**, and so the result is that **r1 == r2 == 17**.

3.5 Transactional Memory

3.5.1 Optimistic Versioning (I)

This example is adapted from [Harris 2006], as sample code that could be found in a software transactional memory (STM) system. Consider the following code, where **w** is an interlocked write-control variable storing a version number or write-lock flag, **w** protects object **x**, multiple readers can execute concurrently and commit as long as no writers are in progress (**w == WRITELOCK**) or completed since (**w** was incremented), and threads **T1** and **T2** are the only observers manipulating **w** and **x**:

```
// thread T1 (reader)
do {
  w1 = w;                  // 1: read version #
  if( w1 != WRITELOCK ) {
    local = x;             // 2: read from x
    ... other work ...
  }
}
while( w1 == WRITELOCK ||
        !a_cas( &w, w1, w1 ) ); // 3: check v#
```

```
// thread T2 (writer)
while(
  (w2 = a_swap( &w, WRITELOCK ))  // 1: r+w
  == WRITELOCK
) { ; } // spin
x = ... ;                 // 2: write to x
... other work ...
w = w2 + 1;               // 3: write new v#
```

This code is correct and race-free because T1(1-3) must be performed in that order on T1, and T2(1-3) must be performed in that order on T2:

- Because T1(1) has acquire semantics, T1(2) and T1(3) correctly cannot move ahead of T1(1).
- Because T1(3) has release semantics, T1(3) correctly cannot move ahead of T1(1) or T1(2).
- Because T1(3) has acquire semantics, it ensures that T1(3)'s check will detect any in-progress or completed writes during the execution of T1's loop body.
- Because T2(1) has acquire semantics (actually a full fence thanks to **a_swap**), T2(2) and T2(3) correctly cannot move ahead of T2(1).
- Because T2(3) has release semantics, T2(3) correctly cannot move ahead of T2(1) or T2(2).

3.5.2 Optimistic Versioning (II)

This example is adapted from [Harris 2006], as sample code that could be found in a software transactional memory (STM) system. Consider the following program code, where none of the variables are interlocked:

```
// program code
...
int x = g_x;
int y = g_y;
...
```

In the above code, the two assignments can be reordered.

An STM implementation may transform the above program code as follows to add instrumentation:

```
// STM transformation — from [Harris 2006]
...
OpenForRead(&g_x, ...);     // 1: performs an interlocked read of some g_x.tmw
int x = g_x;                // 2
OpenForRead(&g_y, ...);     // 3: performs an interlocked read of some g_y.tmw
int y = g_y;                // 4
...
```

The requirements here are that (a) 1 must precede 2, and (b) 3 must precede 4. To ensure this ordering, it is sufficient to make **OpenForRead** contain a read of an interlocked variable associated with the particular memory location passed to the function.

Note that this guarantee is more restrictive than strictly necessary to achieve the desired semantics fro this example, in that 1 does not need to precede 3 or 4. This memory model does not provide a direct way to express the less restrictive ordering that would permit 3 and/or 4 to be reordered before 1, but this memory model does allow looser models to be implemented at higher levels that would permit such reorderings. For further discussion, see §4.3.

3.5.3 Atomic Block Coarsening

Consider the following example, provided by Tim Harris [Harris 2006a], where initially **x = y = 0** and x and y are noninterlocked:

```
// thread T1
atomic {
  x = 1;              // a
}
atomic {
  y = 2;              // b
}
```

```
// thread T2
atomic {
  r1 = y;             // c
}
atomic {
  r2 = x;             // d
}
```

In all cases, if **r1 == 2** then **r2 == 1**. Having **r1 == 2** and **r2 == 0** is not a valid result.

The only legal transformation in this example is to merge consecutive **atomic** blocks (permitted by R4, if the implementation uses the same control variable for the consecutive blocks; see also Example 3.2.3). Consider each case:

- If merging is not performed, **r1 == 2** only if **b** $\overset{c}{\rightarrow}$ **c**, in which case **a** $\overset{c}{\rightarrow}$ **d** and **r2 == 1**.
- If the merging of either or both sets of atomic blocks is performed, then lines **a** and **b** can be reordered, and/or **c** and **d** can be reordered, within their respective merged atomic blocks, but this does not affect the result because **a** and **b** are now performed atomically before or after **c** and **d**, and again **r1 == 2** implies **r2 == 1**.

The following examples demonstrate variants of the above.

3.5.4 Partially Synchronized Program (I)

Consider the following example, proposed by Tim Harris [Harris 2006a] as a variant of Example 3.5.3, where again initially **x = y = 0** and **x** and **y** are noninterlocked:

```
// thread T1
atomic {
  x = 1;              // a
}
y = 2;                // b
```

```
// thread T2
r1 = y;               // c
atomic {
  r2 = x;             // d
}
```

In all cases, **r2** is either **0** or **1**. However, this program violates R7 because it contains a race on **y**, and so **r1** can contain any value.

The following doesn't change the answer, but for completeness we note that the only legal transformations are: (1) line **b** could move into T1's atomic block, and possibly move ahead of **a** within the block; and/or (2) line **c** could move into T2's atomic block, and possibly move after **d** within the block. If only (1) or only (2), then **r1** can still contain any value. (Having both (1) and (2) would reduce the set of possible values, but the programmer cannot rely on these transformations happening.)

3.5.5 Partially Synchronized Program (II)

Consider the following example, proposed by Tim Harris [Harris 2006a] as a variant of Example 3.5.3, where again initially **x = y = 0** and **x** and **y** are noninterlocked:

```
// thread T1
y = 2;            // part of an event a
atomic {
  x = 1;          // b
}
```

```
// thread T2
atomic {
  r2 = x;         // c
}
r1 = y;           // part of an event d
```

The question is: If **r2 ==1**, are we guaranteed that **r1 == 2**? The answer is yes, because if **r2 ==1** then **c** observed **d**, so **a** $\xrightarrow{c}$ **b** $\xrightarrow{c}$ **c** $\xrightarrow{c}$ **d**, so in line **d r1 == 2**. Note that there is no race, and it does not matter whether or not an optimizer chooses to move the write and/or read of **y** into the atomic block(s).

3.5.6 Intervening Atomic Block

Consider the following example, proposed by Tim Harris [Harris 2006a] as a variant of Example 3.5.3, where again initially **x = y = 0** and **x** and **y** are noninterlocked:

```
// thread T1
x = 1;            // a
atomic { }
y = 2;            // b
```

```
// thread T2
atomic {
  r1 = y;         // c
}
atomic {
  r2 = x;         // d
}
```

This program violates R7 because it contains races on both **x** and **y**, and so **r1** and **r2** can contain any values.

The following doesn't change the answer, but for completeness we note that the only legal transformations are that: (1) line **a** could move into the atomic block which would remove the race on **x**; and/or (2) line **b** could move into the atomic block which would remove the race on **y**. If both (1) and (2), then line **b** could additionally be moved ahead of line **a**, both remaining inside the atomic block. (Any of these transformations would reduce the set of possible values, but the programmer cannot rely on these transformations happening.)

3.6 Arvind's Examples

3.6.1 [Arvind 2006a] Figure 3

This example is adapted from [Arvind 2006a] Figure 3, and by R7 we make **x** and **y** interlocked instead of writing explicit fences as in the original example:

```
// thread T1
x = 1;      // a
y = 2;      // b
r1 = y;     // c == 3
```

```
// thread T2
y = 3;      // d
x = 4;      // e
r2 = x;     // f ?= 1
```

Note that, in this example, each thread's reads and writes must be performed in program order because of the interlocked semantics and data dependencies.

The question is, it is possible to have **r1 == 3** and **r2 == 1**? The answer is no, because this result would require two observers to disagree on the order of causally related events, which violates R6. The contradiction is that **r1 == 3** implies **b** $\xrightarrow{c}$ **d**, whereas **r2 == 1** implies **d** $\xrightarrow{c}$ **b**. Expanding slightly:

- If **r1 == 3**, then line **c** observed **d**, and so **b** $\xrightarrow{c}$ **d**.
- If **r2 == 1**, then line **f** observed **a**, and so **e** $\xrightarrow{c}$ **a**, and so **d** $\xrightarrow{c}$ **e** $\xrightarrow{c}$ **a** $\xrightarrow{c}$ **b**.

3.6.2 [Arvind 2006a] Figure 4

This example is adapted from [Arvind 2006a] Figure 4, and by R7 we make **x** and **y** interlocked instead of writing explicit fences as in the original example:

```
// thread T1
x = 1;     // a
x = 2;     // b
r1 = y;    // c == 3
```

```
// thread T2
y = 3;     // d
y = 5;     // e
r2 = x;    // f ?= 1
```

Note that, in this example, each thread's reads and writes must be performed in program order because of the interlocked semantics and data dependencies.

The question is, it is possible to have **r1 == 3** and **r2 == 1**? The answer is no, because this result would require two observers to disagree on the order of causally related events, which violates R6. The contradiction is that **r1 == 3** implies $\mathbf{b} \overset{c}{\rightarrow} \mathbf{e}$, whereas **r2 == 1** implies $\mathbf{e} \overset{c}{\rightarrow} \mathbf{b}$.

Expanding slightly:

- If **r1 == 3**, then line **c** observed **d** but not **e**, and so $\mathbf{b} \overset{c}{\rightarrow} \mathbf{c} \overset{c}{\rightarrow} \mathbf{e}$.
- If **r2 == 1**, then line **f** observed **a** but not **b**, and so $\mathbf{e} \overset{c}{\rightarrow} \mathbf{a} \overset{c}{\rightarrow} \mathbf{b}$.

3.6.3 [Arvind 2006a] Figure 5

This example is adapted from [Arvind 2006a] Figure 5, and by R7 we make **x** and **y** interlocked instead of writing explicit fences as in the original example:

```
// thread T1
x = 1;     // a
r1 = y;    // b == 2
r2 = y;    // c == 4
```

```
// thread T2
y = 2;     // d
```

```
// thread T3
y = 4;     // e
x = 8;     // f
r4 = x;    // g ?= 1
```

Note that, in this example, each thread's reads and writes must be performed in program order because of the interlocked semantics and data dependencies.

The question is, if **r1 == 2** and **r2 == 4**, is it possible to have **r4 == 1**? The answer is no, because this result would require two observers to disagree on the order of causally related events, which violates R6. The contradiction is that having both **r1 == 2** and **r2 == 4** implies $\mathbf{a} \overset{c}{\rightarrow} \mathbf{h}$, whereas **r4 == 1** implies $\mathbf{h} \overset{c}{\rightarrow} \mathbf{a}$.

Expanding slightly:

- If **r1 == 2** and **r2 == 4**, then $\mathbf{d} \overset{c}{\rightarrow} \mathbf{b} \overset{c}{\rightarrow} \mathbf{e} \overset{c}{\rightarrow} \mathbf{c}$, and in turn $\mathbf{b} \overset{c}{\rightarrow} \mathbf{e}$ implies $\mathbf{a} \overset{c}{\rightarrow} \mathbf{f}$.
- If **r4 == 1**, then $\mathbf{f} \overset{c}{\rightarrow} \mathbf{a}$.

3.6.4 [Arvind 2006a] Figure 7

This example is adapted from [Arvind 2006a] Figure 7, and by R7 we make **x** and **y** interlocked instead of writing explicit fences as in the original example:

```
// thread T1
x = 1;     // a
y = 3;     // b
r1 = y;    // c
```

```
// thread T2
y = 4;     // d
r2 = x;    // e
```

```
// thread T3
x = 2;     // f
```

Note that, in this example, each thread's reads and writes must be performed in program order because of the interlocked semantics and data dependencies.

The question is, if **r1 == 4** and **r2 == 2**, what if anything can we say about the relationship between events **a** and **f**? If **r1 == 4**, then **b** $\overset{c}{\hookrightarrow}$ **d**, and so **a** $\overset{c}{\hookrightarrow}$ **e**. If also **r2 == 2**, then **a** $\overset{c}{\hookrightarrow}$ **f** $\overset{c}{\hookrightarrow}$ **e**. Therefore, if **r1 == 4** and **r2 == 2**, then **a** $\overset{c}{\hookrightarrow}$ **f**.

3.6.5 [Arvind 2006a] Figure 8: Speculative Execution

This example is adapted from [Arvind 2006a] Figure 8, and by R7 we make **w**, **x**, **y**, and **z** interlocked instead of writing explicit fences as in the original example (note that this affects the answer to the question posed in the original and considered below). Note that **w**, **x**, and **z** are pointers containing the address of another memory location, and unary * denotes deference:

```
// thread T1
x = w;      // a
y = 2;      // b
y = 4;      // c
x = z;      // d
```

```
// thread T2
r1 = y;     // e = 2
r6 = x;     // f
*r6 = 7;    // g
r8 = y;     // h
```

The first question is: If **r1 == 2**, can **h** observe either **b** or **c** (**r8 == 2** or **4**)? The answer is yes. If **r1 == 2**, then **b** $\overset{c}{\hookrightarrow}$ **e** $\overset{c}{\hookrightarrow}$ **c**. Three is no causal ordering between **c** and **h**, so **r8 == 2** and **r8 == 4** are legal outcomes.

The second question is: Can line **g** be reordered after line **h**? (Clearly line **g** cannot be reordered to precede line **f**, because of the data dependency.) The answer does not depend on the memory model, but only on local sequential data and control flow rules: Lines **g** and **h** can be reordered iff **r6** does not contain the address of **y**. As noted in [Arvind 2006a], this restricts speculative execution. If line **h** is executed speculatively as written before line **f**, then the speculation will have to be thrown away if it is discovered that **r6** contains the address of **y**. On the other hand, if line **h** is speculatively executed as **r8 = 7**, then the speculation will have to be thrown away if it is discovered that **r6** does not contain the address of **y**.

3.7 [JSR-133 2004]'s Examples

3.7.1 [JSR-133 2004] Figure 6

This example is adapted from [JSR-133 2004] Figure 6, and **x** and **y** are ordinary shared variables:

```
// thread T1
r1 = x;             // a
if( r1 != 0 )
  y = 1;            // b
```

```
// thread T2
r2 = y;             // c
if( r2 != 0 )
  x = 1;            // d
```

By R5 and R7, this code is correctly synchronized and the result is **r1 == r2 == 0**. R5 does not permit either thread's reads and writes of **x** and **y** to be reordered, because there is no sequentially consistent execution where line **b** or line **d** will be executed.

3.7.2 [JSR-133 2004] Figure 7

This example is adapted from [JSR-133 2004] Figure 7, and **x** and **y** are ordinary shared variables:

```
// thread T1
r1 = x;             // a
y = r1;             // b
```

```
// thread T2
r2 = y;             // c
x = r2;             // d
```

By R7, this code is not correctly synchronized. Even though there is a race, if **x** and **y** each occupies a single memory location (and therefore each read and write is atomic) then we can make the statement that the result is **r1 == r2 == x == y == 0** because there is no sequentially consistent execution where any variable could have a nonzero value.

3.7.3 [JSR-133 2004] Figure 8

This example is adapted from [JSR-133 2004] Figure 8, and **x** and **y** are ordinary shared variables:

```
// thread T1
r1 = x;              // a
r2 = x;              // b
if( r1 == r2 )
  y = 2;             // c
```

```
// thread T2
r3 = y;
x = r3;
```

By R7, this code is not correctly synchronized. Given that there is a race, the question is: Is **r1 == r2 == r3 == 2** possible? The answer is yes. As described in [JSR-133 2004], one valid transformation is to remove the redundant read of **x** in line **a**:

```
// thread T1 (valid transformation)
r1 = x;              // a
r2 = r1;             // b'
if( r1 == r2 )
  y = 2;             // c
```

```
// thread T2
r3 = y;              // d
x = r3;              // e
```

After this, the condition is always true and can be eliminated, and line **c** can be moved ahead of lines **a** and **b'**.

3.7.4 [JSR-133 2004] Figure 12

This example is adapted from [JSR-133 2004] Figure 12, and **x** is an ordinary shared variable:

```
// thread T1
r1 = x;              // a
x = 1;               // b
```

```
// thread T2
r2 = x;              // c
x = 2;               // d
```

By R7, this code is not correctly synchronized. Given that there is a race, the question is: Is **r1 == 2** and **r2 == 1** possible? The answer is yes. [JSR-133 2004] permits this, saying that "the behavior **r1 == 2** and **r2 == 1** might be allowed by a processor architecture that performs the writes early, but in a way that they were not visible to local reads that came before them in program order. This behavior, while surprising, is allowed by the Java memory model." No rule in this memory model prohibits such an implementation.

3.7.5 [JSR-133 2004] Figure 14

This example is adapted from [JSR-133 2004] Figure 14, and **x** and **y** are ordinary shared variables:

```
// thread T1
r1 = x;              // a
if( r1 == 1 )
  y = 1;             // b
```

```
// thread T2
r2 = y;              // c
if( r2 == 1 )
  x = 1;             // d
else
  x = 1;             // e
```

By R7, this code is not correctly synchronized. Given that there is a race, the question is: Is **r1 == r2 == 1** possible? The answer is yes. The reason is that T2's assignment to **x** will be performed regardless of the value of **r2**, and so lines **d** and **e** can be merged and moved before the conditional test (which can then be eliminated because nothing remains in either branch), and then before line **c**.

3.8 Selected Language Semantics

todo: this section under development, quite a bit more needs to come here

3.8.1 new

Consider the following C++ statement that contains a new-expression, where **p** is interlocked:

```
// program code
p = new T();
```

Conceptually, the compiler actually allocates raw memory, constructs the object, and stores the pointer into **p** — in some order. The following is a translation that conforms to ISO C++ rules and to R8:

```
// transformation (valid)
void *__temp = /* T */ ::operator new( sizeof(T) );     // allocate raw memory
new (__temp) T();                                        // call constructor
p = __temp;                                              // copy pointer
```

The following translation also conforms to ISO C++ rules, but is invalid according to this memory model:

```
// transformation (not valid)
p = (T*) /* T */ ::operator new( sizeof(T) );            // allocate raw memory
new ((void*)p) T();                                      // call constructor
```

Even in the absence of C++ language rules, the latter translation is invalid because it violates R8.

It also invalid by C++ language rules. Because there is a sequence point at the end of the constructor call, the compiler must first translate it into a constructor call followed by the assignment to **p**, and then cannot reorder the write to **p** upwards because it is an interlocked store.

3.8.2 Shared Function Static Objects (C++)

In C++, a **static** local object is shared across all executions of the function, but is not initialized until its first use:

```
void f() {
  static X x;          // dynamically initialized

  ...
}
```

To implement the language's required semantics correctly, the C++ compiler must ensure that initialization of **x** is race-free (unless it can prove that **f** can never be called concurrently by two different observers).

One option is to have the compiler generate code like that for Double-Checked Locking to protect **x**'s initialization (see Example 3.3.4).

A second option is to generate code similar to the following:

```
void f() {
  static X x;                                   // statically uninitialized
  static interlocked char flag = 0;             // statically initialized to 0

  if( flag != DONE ) {                          // (for efficiency)
    if( a_cas( &flag, 0, CONSTRUCTING ) ) {     // if I get to be the one constructing
      new (&x) X;                               // then construct
      flag = DONE;
    } else {
      while( flag == CONSTRUCTING )
        ; // spin
    }
  }
  ...
}
```

In either case, **x** is guaranteed to be initialized without a race. (If the program later uses **x** in a way that could cause a race, it must correctly synchronize access to **x**.)

3.8.3 Initonly and Final Fields

todo

4 Discussion

4.1 Compatibility

For backward compatibility, the/an old memory model can be explicitly requested by the developer, or used automatically by default for code that can be recompiled dynamically (e.g., JIT compilation) and that was originally developed under a previous memory model.

In our next tool chain release that implements this memory model by default:

- Compilers will add a tag to every binary/assembly produced using the new memory model.
- A developer can opt out of the new model and select the old model via some syntax (e.g., **#pragma**) to be defined by individual languages.
- Any JIT-like compiler will check the tag, and if the new memory model does not apply to the code being compiled it will disable optimizations as needed to comply with the older memory model.

todo: barriers around calls across new/old code? barrier on thread create? destroy?

4.2 Guarantees In the Presence of Races

Some safety guarantees should be provided even in the presence of program races, notably where needed to strengthen runtime system integrity (e.g., memory safety) and language feature semantics (e.g., initialization of initonly/final fields should be made safe without external explicit synchronization; see §3.8.3).

For the programmer's own invariants, however, what guarantees should hold even in programs with races? The potential answers range widely, and this is perhaps the area of most debate. From most to least restrictive, the major options include the following, where "transformation" includes the reordering, elision, and/or invention of memory operations. Note that these deal only with ordinary reads and writes of shared variables, and deals only with additional guarantees (we always assume ordinary sequential dependencies are satisfied):

1. **Allow no transformations, require full sequential consistency?** This option would seriously inhibit optimizations, especially compiler code motion and memory latency hiding.
2. **Allow transforming reads, but not writes?** It is conjectured that allowing read reordering while prohibiting write reordering would enable most of the desirable optimizations. Prohibiting write reordering is also conjectured to improve debuggability of races and eliminate some classes of invalid state, by reducing the set of possible surprising behaviors in a race. Chris Brumme [Brumme 2006] in particular makes a persuasive argument that, because races cannot in general be prevented or diagnosed with perfect accuracy even at test time, performing writes in program order can significantly help programmers to figure out what is going wrong when debugging a race.
3. **Allow transforming both reads and writes, but every write to a memory location must write a value that would be written in a sequentially consistent execution?** This allows latitude for most local optimizations, while prohibiting the creation of "impossible" values in individual (atomically updated) memory locations; see Examples 3.1.3 through 3.1.7.
4. **Allow all transformations.** This would follow the philosophy of permitting full local optimizations and relying on the programmer to always correctly synchronize his program so that the optimizations cannot be detected.

The Whidbey managed memory model chooses approximately #2. [Hogg 2005] (See also §5.3.) The Java memory model chooses approximately #3. [Manson 2005]

This paper chooses #3, and the rest of this section makes an argument for this choice. For the programmer's own invariants, we believe that only a few useful guarantees are possible in the presence of races. Although enforcing strict sequential consistency could make races somewhat easier to reason about during debugging, which is attractive, we believe that this path is probably unfruitful for the following reasons:

- **The stronger guarantees, even #1 (SC), don't matter unless there is a race.** The surprising values can only be observed in a race condition, and so the extra guarantees don't matter for a correctly synchronized program.
- **The stronger guarantees, even #1 (SC), don't help much when there is a race.** In general, in a race a program can observe the same kinds of surprising values anyway. For example, even under #1 (full SC), in a race even a plain **int** variable can be observed with "impossible" values (e.g., due to word tearing), and in general nearly any invariant that involves multiple variables (e.g., the state of an object, which depends on the values of its member variables) is liable to be broken in a race when the program fails to perform correct synchronization.

There does not appear to be a significant practical difference between: (a) a corrupted object containing an invalid combination of bits because of a program race, even in a sequentially consistent execution; and (b) a corrupted object containing a different invalid combination of bits because of a program race and other effects such as write reordering. Once an object is in such a state, it is not possible in general to safely use the object, not even to safely destroy or finalize it.

So our position is not that we choose not to make guarantees for programs with races, but rather that few useful guarantees are possible, and that trying to provide guarantees for a program with races at best gives the programmer a false sense of security.

In contrast, consider choice #2 above: The managed memory model follows #2 and attempts to reduce invalid values even in races by prohibiting write reordering, and the managed environment aggressively aligns some fundamental types (including **int**) to guarantee that simple reads and writes are atomic by default on popular hardware platforms. For example, the following code will behave in a sequentially consistent manner on .NET even if **x** is a plain **int** without any synchronization (not even **volatile**), and **x** will end up being either **-1** or **1**:

// thread T1	// thread T2
x = -1;	x = 1;

However, even with prohibiting all write reordering (per #1) plus strong alignment for **x**, this seems to be only a partial illusion of safety. Even slight code changes will break this sequentially consistent façade and allow "impossible" values, for example by: (a) changing the type of **x** to be **Double** or **Decimal** which are too large to be updated atomically; or (b) changing T1's code to **x--;** which is not atomic (note that although code like **x--** could be made atomic using a compare-and-swap technique, doing so is impractically expensive). We wonder whether choosing #2 would have a net effect of improving or worsening the problem; on the one hand, #2 stands improve the programmer's ability to debug detected races; on the other hand, it could degrade the ability to discover races, providing a false sense of security by masking some kinds of latent races in some circumstances.

There has been much debate about the actual performance value of relaxed memory models. [Adve 1995, Hill 1998, Adve 2000, Hill 2003, JSR-133 2004]). The academic literature typically focuses on hardware optimizations, not software (compiler) optimizations. This is unfortunate, because routine compiler optimizations are known to have significant benefits up to order-of-magnitude improvements, whereas in hardware it is argued that techniques like scouting and other speculative execution have closed the gap between SC and relaxed models to 20% or less. [Hill 1998, Hill 2003] We assert that memory models that allow both read and write reordering are essential in order to take advantage of common techniques like register allocation and common subexpression evaluation that are known to be important and useful compiler optimization techniques.

Consider this code adapted from [Adve 2000], where initially **x = y = flag = 0** and **flag** is interlocked:

```
// processor P1
for( ... ) {
  ...
  x++;
  ...
  y += ...;
  ...
}
flag = 1;
```

```
// processor P2
while( flag != 1 ) ;
...
r1 = x;
r2 = y;
```

First, this memory model permits reordering ordinary writes. Compilers can therefore apply common optimizations like register allocation and CSE to shared variables like **x** and **y**. Without such optimizations, loops like P1's can be significantly slower (e.g., a June 2006 internal mail thread reported a 400% performance difference for just such a loop, where **x** had type **int** and **y** had type **float** [Clrperfe 2006]).

Second, in P2's frame of reference, this memory model allows P1's writes to **x** and **y** to be postponed until as late as P2's reads of **x** and **y**. Adve observes that hardware implementations can exploit this latitude with "lazy invalidations [and] lazy release consistency on software DSMs." [Adve 2000]

4.3 Finer Granularity

This memory model uses the conventional notion of interlocked reads and writes having acquire and release semantics. This is known to be somewhat coarse-grained, but we use it because it is difficult to get much finer-grained without seriously complicating the model. This model permits languages to define additional fine-grained semantics that will be preserved by this model.

In particular, when a program performs an interlocked write (e.g., lock release) to publish a set of ordinary writes or to exit a critical section, the interlocked write is often publishing or protecting some, but not all, of the reads and writes in the preceding batch (see Example 3.5.2). But it is not known exactly which reads and writes the programmer intended to protect, and so this model therefore prevents any memory operation from moving past an interlocked write, in case that access was part of what was to be published or protected.

By knowing exactly which ordinary reads and writes are associated with a given interlocked variable, we could enable optimizations to move unrelated ordinary reads and writes across the interlocked write without affecting program semantics.

Although this memory model does not require a way to associate a given ordinary read or write with a given interlocked variable, it does allow languages and tools to let such relationships to be declared (e.g., by the programmer in programming model extensions) and/or deduced (e.g., through whole program analysis), and then to make use of the looser semantics in optimizations at higher levels (e.g., compiler optimizations). Optimizations at lower levels that are unaware of the looser semantics will apply the stricter semantics in this memory model. This correctly preserves the finer-grained semantics as long as they are strictly looser than the guarantees of this model, and so any looser models built on top of this memory model must not add any additional guarantees not present in this model (unless it implements them in terms of the guarantees of this model, e.g., by generating appropriate use of interlocked reads and writes).

5 Related Work

There are three main pieces of commercial software existing practice that this proposal should consider or coordinate with. In chronological order, they are.

- **Java 5 memory model (2004):** Before Java 5, Java's memory model was known to be deficient in a number of ways. [Pugh 2000] Java 5 then specified a new memory model that provided more consistent guarantees to programmers. [JSR-133 2004]
- **Visual Studio 2005 managed memory model (2005):** During the VS 2005 product cycle, the Phoenix and CLR teams specified a CLR memory model for managed code. [Hogg 2005; Morrison 2005; Morrison 2005a] (Note that Ecma/ISO CLI also specifies a memory model; this paper will not consider that model because it is known to be looser than what CLI implementations actually implement and therefore untestable. It also arguably places an unreasonable burden of responsibility on programmers. [Brumme 2003])
- **ISO C++ memory model (under development, ETA 2007):** The ISO C++ standards committee is now working to define an international standard for a cross-platform native memory model. [C++MM 2006] This work has gained momentum during 2006, and is expected to be finalized in 2007.

We also note similarities between this model and the following academic work in particular:

- **Lamport's happens-before relation (1978):** For message-passing systems, and used to implement Lamport clocks. [Lamport 1978]
- **Adve and Hill's DRF0 memory model (1990):** The model in this paper was independently derived, and is similar to DRF0. [Adve 1990]
- **Gharachorloo's RC memory model (1990):** Release consistency. [Gharachorloo 1990]

This section considers some of the above, and discusses how this paper's goals and choices differ from the above designs and provides a rationale for those choices.

5.1 Lamport Happens-Before [Lamport 1978]

Applying Lamport's formulation directly to memory operations considers an individual ordinary read (message send) or ordinary write (message receive) to be an event, in that the write sends information that can propagate and be subsequently read by another process (observer):

> *A single process is defined to be a set of events with an a priori total ordering. … We assume that sending or receiving a message is an event in a process. …*
>
> *The relation '→' on the set of events of a system is the smallest relation satisfying the following three conditions: (1) If **a** and **b** are events in the same process, and **a** comes before **b**, then **a** → **b**. (2) If **a** is the sending of a message by one process and **b** is the receipt of the same message by another process, then **a** → **b**. (3) If **a** → **b** and **b** → **c** then **a** → **c**.* — [Lamport 1978]

This formulation can be directly applied to specify a memory model, but it is not sufficient to guarantee causality (Principle P3 = Rule R6) without one additional guarantee, described below.

Consider Figure 2, an interaction diagram showing three processors P1-P3 where time increases upward. Two writes **a** and **b** are performed by processors P1 and P2, respectively. Each dashed arrow begins at a write performed by one processor, and points to when the write becomes observable by another specific target processor.

In particular, if P2 observes **a** at $\mathbf{a}_2$ and then performs **b**, is it possible for processor P3 to observe **b** at $\mathbf{b}_3$ before it is able to observe **a** at $\mathbf{a}_3$?

According to this memory model, if **a** and **b** are events and $\mathbf{a} \stackrel{c}{\hookrightarrow} \mathbf{b}$, then the red edge is illegal by R6 because P3 cannot observe **b** before being able to observe **a**. (Imagine that P1, P2, and P3 are physical observers who observe events through telescopes. It is not possible for a light signal to travel from P1 to P2 to P3 in less time than it can travel directly from P1 to P3.)

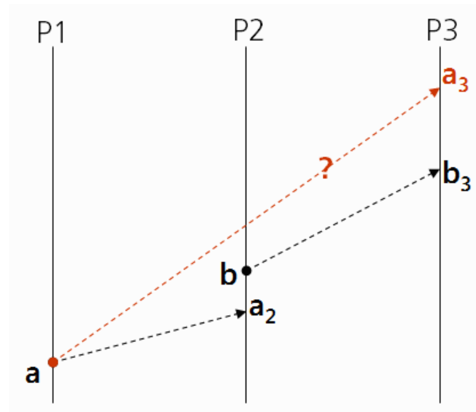

Figure 2: A "faster-than-light" causality violation

According to the [Lamport 1978] rules, each individual read and write is considered to be a distinct event, and we see that the red edge is legal because $\mathbf{a} \to \mathbf{a}_2 \to \mathbf{b} \to \mathbf{b}_2 \to \mathbf{a}_3$ and $\mathbf{a} \to \mathbf{a}_3$ are both legal paths in the happens-before graph. The problem is that reads like $\mathbf{a}_2$ and $\mathbf{a}_3$ that are *observations of the same write event* become decoupled and treated independently, so that the above rules are insufficient to govern the ordering in which dependent writes performed by two different observers become visible to third parties.

What is needed is an additional requirement that a message not travel "faster than light." For example:

> **(4)** If **a** and **b** are the sendings of two messages by two different processes, **a′** and **b′** are the receipt of **a** and **b** by a third process, and $\mathbf{a} \to \mathbf{b}$, then $\mathbf{a}' \to \mathbf{b}'$.

With this additional rule, and interpreting "event" as defined in this paper (an interlocked read or write, or a batch of ordinary reads and writes), we believe the Lamport happens-before relation $\to$ is closer to causally-precedes $\stackrel{c}{\hookrightarrow}$ for the purpose of specifying Rule R6 and preserving causality.

5.2 Java 5 Memory Model [JSR-133 2004]

The Java 5 memory model (henceforth Java model) has many strengths. We feel there are two main weaknesses in this model. The first is that it is complex and hard to understand.

The second is that it is unclear and inconsistent about causality, a notion that is central but is not well defined or enforced in the Java model. The paper frequently falls back on case-by-case analysis of code examples that it interprets as apparently violating causality and surprising programmers, and then somewhat arbitrarily declares some to be illegal and others to be legal (the latter several times accompanied by handwringing that it's unfortunate that the cases are surprising to programmers but that allowing them is necessary to enable important optimizations).

We strongly agree that the theme of causality is important, but the reason the Java model doesn't answer these questions well is because its notion of causality not well-defined. In particular, this paper's model: (a) defines the unit of "an event" to be an interlocked operation *or a batch of ordinary operations between interlocked operations*; (b) rigorously defines causality; and then (c) rigorously guarantees causality for those units of work *which allows full local optimizations that do not violate acquire/release boundaries*.

Under this memory model, all of the causality "problem examples" in [JSR-133 2004] come out the same way they do in the Java model, but with a much stronger rationale and without special fudging or arbitrary case-by-case rules. We believe this paper gives a more powerful definition and a better model to achieve what both papers agree are the right answers for these examples. See §3.7 for detailed examples.

Finally, a few of the other examples differ, in particular the lock-based code in Example 3.2.5 which is a race under the Java model and not under this memory model.

5.3 Visual Studio 2005 Managed Memory Model [Hogg 2005, Morrison 2005a]

The Whidbey managed memory model (henceforth "managed model") was designed to target currently shipping IA32- and IA64-compatible hardware. Therefore, in addition to its explicit rules, it also includes implicit rules based on assumptions that happen to be true on that hardware. In particular, the managed model assumes that every shared write (whether ordinary or interlocked) will become visible to all other processors at the same time.

The managed model also defines the following explicit rules:

> *Rule-1. Shared-writes have release semantics*
>
> *Rule-2. May coalesce adjacent shared-reads or shared-writes*
>
> *Rule-3. Interlocked accesses have acquire/release semantics; adjacent merging is not allowed*
>
> *Rule-4. Cannot introduce or remove non-adjacent shared-reads; ditto for shared-writes*
>
> *Notes: …*
>
> - *Note that all of the reads and writes discussed in this spec are assumed atomic.*
>
> — [Hogg 2005]

This memory model differs mainly in its treatment of ordinary reads and writes. We allow much greater latitude for the reordering/creation/elision of ordinary reads and writes, and permit a strict superset of the transformations permitted under the managed model. Specifically:

- Rule-1 is not required in this paper's model. (Rule-1 is discussed in further detail below.)
- Rule-2 agrees with this paper, and is covered by R1, R3, and R4.
- Rule-3 mostly agrees with this paper, and is covered by R3 and R4. However, R3, R4, and R4 do permit some elision/merging of interlocked operations.
- Rule-4 mostly agrees with this paper, and is covered mainly by R5.
- The atomicity note above is covered for interlocked objects by Rule R2.

Rule-1 does not exist in this memory model. Note that Rule-1 could be restated as "writes cannot be reordered." The rule is stated in terms of release semantics because, on current Intel platforms, emitting every write as a **st.rel** is observed to be sufficient to both perform each processor's stores in order *and* to make them visible in that order to other processors; that is, the execution environment is processor consistent (PC) so that writes performed in order will be observed in order by even ordinary reads because all writes are assumed to be visible atomically at the same time to all other processors. (A general acquire/release model would additionally need all reads to have acquire semantics, and then Rule-1 would have to be formulated differently because that would additionally prohibit read reordering which the managed model does not want to prohibit.)

Rule-1 was adopted in part to make certain classes of existing bugs be legal, by assuming that all writes might be releases. One motivation for Rule-1 was backward compatibility with existing code that will be recompiled in the field with a new JIT compiler, because it would be impossible in general to require all shipped code to first be fixed (to use locks or **interlocked**) before it is recompiled. As noted in the managed model's specification:

> *In some cases, the original code is technically wrong – it doesn't follow CLR rules for use of interlocked (as specified in the ECMA/ISO spec; Partition I, Section 12). In other cases it assumes that execution will slavishly follow source code, with no optimization being performed by the JIT. Put another way, if the author had made correct use of* ***lock****s and* ***interlocked*** *references, it would have worked correctly on all platforms – past, present and future.*
>
> *Going forward, Microsoft might simply state that such code is wrong, and must be fixed. However, the CLR team feels this creates an unacceptable user experience. (It's slightly worse than this: some of our own .NET Framework library code, already shipped, contains these defects. A customer might see breakages if he simply re-ran existing code on a multi-processor Itanium).* — [Hogg 2005]

But this compatibility goal is inconsistent with the rationale for Snippet-4 in [Hogg 2005], which states that changes to shipped code are nevertheless required:

> *The CLR team shall check that any spin-locks in managed library code are written correctly to keep working, by ensuring that reads on* ***g*** *are marked as ordered (ie, having acquire semantics). The alternative, of having JITs treat every shared-read as ordered is estimated as too costly to runtime performance of managed code.* — [Hogg 2005]

Rule-1 prevents some optimizations that may be desirable, including some kinds of common subexpression elimination and register allocation. For example, Rule-1 prevents any optimization of loops like **for(i=0; i<1000000; i++) { count++; count2++; }** where **count** and **count2** might be shared. Recent internal mail threads have complained about 400% performance differences between managed and native code in such examples [Clrperfe 2006], although that appears to be a worst case because the loop is not doing any other work which would reduce or swamp this overhead. The managed model paper itself notes this for Snippet-9:

> *"The two shared-writes are not adjacent, and so cannot be coalesced by Rule-2. Moreover, the JIT cannot advance [3] above [2] in an attempt to make them adjacent – that is disallowed by Rule-1. Not allowing the JIT to perform this optimization is unfortunate. However, in general, we cannot be sure that another thread is spinning on* ***g2*** *– when set, it signals that* ***g1*** *can be accessed."* — [Hogg 2005]

In that example, the shared variables **g1** and **g2** are neither protected by a lock or marked interlocked. The problem arises that, because the managed model essentially treats every shared variable as a potential flag (but does so incompletely; see below) it cannot optimize the vast majority that are not. In this paper's model, **g2** would be declared interlocked if it were such a flag, and the optimization would be allowed in the majority of cases where it is not.

Finally, note that as of this writing Rule-1 is not enforced consistently in our JIT compilers (notably JIT64), which appears to perform such optimizations anyway in violation of the managed model.

This paper does not currently adopt Rule-1, mainly because Rule-1 is not necessary to achieve sequential consistency in race-free programs, and prevents compiler optimizations that could benefit from moving ordinary writes. However, if preventing store ordering is considered important (see §4.2), then such a rule should be adopted (but it should probably not be specified in terms of **st.rel** semantics).

6 References

[Adve 1990] S. Adve and M. Hill. "Weak Ordering—A New Definition" (*Proceedings of the 27th Annual International Symposium on Computer Architecture*, p. 2-14, May 1990).

[Adve 1998] S. Adve and M. Hill. "Retrospective: Weak Ordering—A New Definition" (*25 Years of the International Symposia on Computer Architecture (selected papers)*, p. 63-66, Barcelona, Spain, 1998).

[Adve 2000] S. Adve. "Memory Consistency Models" (slides from 2000 Java Workshop talk).

[Arvind 2006] Arvind. "Memory Model = Instruction Reordering + Store Atomicity" (Microsoft Research Lecture, May 15, 2006).

[Arvind 2006a] Arvind and J-W. Maessen. "Memory Model = Instruction Reordering + Store Atomicity" (*33rd Annual International Symposium on Computer Architecture*, Boston, MA USA, June 17-21, 2006).

[Boehm 2005] H. Boehm et al. "Memory Model for Multithreaded C++: Issues" (ISO C++ committee paper ISO/IEC JTC1/SC22/WG21 N1777, March 2005).

[Boehm 2005a] H. Boehm. "Reordering Constraints for Pthread-Style Locks" (HP Technical Report HPL-2005-217, November 2005).

[Boehm 2006] H. Boehm. "A Memory Model for C++: Strawman Proposal" (ISO C++ committee paper ISO/IEC JTC1/SC22/WG21 N1942, February 2006).

[Boehm 2006a] H. Boehm. "Memory Model Overview" (ISO C++ committee paper ISO/IEC JTC1/SC22/WG21 N2010, April 2006).

[Boehm 2006b] H. Boehm and N. Maclaren. "Should **volatile** Acquire Atomicity and Thread Visibility Semantics?" (ISO C++ committee paper ISO/IEC JTC1/SC22/WG21 N2016, April 2006).

[Boehm 2006c] H. Boehm, private communication.

[Brumme 2003] C. Brumme. "Memory model" (Blog article, May 2003).

[Brumme 2006] C. Brumme, private communication.

[C++MM 2006] H. Boehm et al. "Threads and Memory Model for C++" (ISO C++ subgroup working site).

[Clrperfe 2006] "Case SRX060601603077: C++ vs C# performance" (*CLR Performance Discussions* discussion thread, June 2006).

[Dijkstra 1968] E. W. Dijkstra. "Cooperating Sequential Processes" (*Programming Languages* (F. Genuys, Ed.), Academic Press, 1968).

[Fraser 2004] K. Fraser and T. Harris. "Concurrent Programming Without Locks" (University of Cambridge Computer Laboratory, 2004).

[Gharachorloo 1990] K. Gharachorloo, D. Lenoski, J. Laudon, P. Gibbons, A. Gupta, and J. Hennessy. "Memory Consistency and Event Ordering in Scalable Shared-Memory Multiprocessors" (*Proceedings of the 17th Annual International Symposium on Computer Architecture*, p. 15-26, Seattle, Washington, United States, 1990).

[Gharachorloo 1991] K. Gharachorloo, A. Gupta, and J. Hennessy. "Performance Evaluation of Memory Consistency Models for Shared-Memory Multiprocessors" (*Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, p. 245-257, April 1991).

[Gharachorloo 1991a] K. Gharachorloo and P. Gibbons. "Detecting Violations of Sequential Consistency" (*Proceedings of the 3rd ACM Symposium on Parallel Algorithms and Architectures*, p. 316-326, 1991, Hilton Head, South Carolina, United States).

[Gharachorloo 1992] K. Gharachorloo, A. Gupta, and J. Hennessy. "Hiding Memory Latency Using Dynamic Scheduling in Shared-Memory Multiprocessors" (*Proceedings of the 19th Annual International Symposium on Computer Architecture*, p. 22-33, May 1992).

[Gharachorloo 1995] K. Gharachorloo. "Memory Consistency Models for Shared-Memory Multiprocessors" (PhD thesis, Stanford University, 1995).

[Gharachorloo 1998] K. Gharachorloo. "Retrospective: Memory Consistency and Event Ordering in Scalable Shared-Memory Multiprocessors" (*25 Years of the International Symposia on Computer Architecture (selected papers)*, p. 67-70, Barcelona, Spain, 1998).

[Harris 2006] T. Harris, M. Plesko, A. Shinnar, D. Tarditi. "Optimizing Memory Transactions" (PLDI'06, Ottawa, Ontario, Canada, June 2006)

[Harris 2006a] T. Harris, private communication.

[Herlihy 1990] M. Herlihy and J. M. Wing. "Linearizability: A Correctness Condition for Concurrent Objects " (*ACM Transactions on Programming Languages and Systems*, 12(3):463-492, 1990).

[Herlihy 1991] M. Herlihy. "Wait-Free Synchronization" (*ACM Transactions on Programming Languages and Systems*, 13(1):124-149, 1991).

[Herlihy 1993] M. Herlihy. "A Methodology for Implementing Highly Concurrent Data Objects" (*ACM Transactions on Programming Languages and Systems*, 15(5):745-770, 1993).

[Herlihy 2003] M. Herlihy, V. Luchangco, and M. Moir. "Obstruction-Free Synchronization: Double-Ended Queues as an Example" (*Proceedings of the 23rd International Conference on Distributed Computing Systems*, IEEE Computer Society, 2003).

[Hill 1998] M. Hill. "Multiprocessors Should Support Simple Memory Consistency Models" (*IEEE Computer*, August 1998).

[Hill 2003] M. Hill. "Revisiting 'Multiprocessors Should Support Simple Memory Consistency Models'" (retrospective of [Hill 1998], Dagstuhl, 2003).

[Hoare 1978] C. A. R. Hoare. "Communicating Sequential Processes" (*Communications of the ACM*, 21(8), August 1978).

[Hogg 2005] J. Hogg et al. "CLR Memory Model" (Microsoft internal Whidbey spec, 2005).

[Intel TBB] Intel. Reference for Threading Building Blocks, version 1.0, April 2006.

[JSR-133 2004] J. Manson, W. Pugh, and S. Adve. "JSR-133: Java™ Memory Model and Thread Specification" (*Java Community Process*, 2004).

[JSR-133 FAQ 2004] J. Manson and B. Goetz. "JSR 133 (Java Memory Model) FAQ" (February 2004).

[Knuth 1989] D. Knuth. "The Errors of TeX" (*Software—Practice & Experience*, 19(7), July 1989).

[Lamport 1978] L. Lamport. "Time, Clocks, and the Ordering of Events in a Distributed System." (*Communications of the ACM*, 21(7):558-565), July 1978).

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

[Lamport 1997] L. Lamport. “How to Make a Correct Multiprocess Program Execute Correctly on a Multiprocessor.” (*IEEE Transactions on Computers*, 46(7):779-782, 1997).

[Lea 2005] D. Lea. “The JSR-133 Cookbook for Compiler Writers” (2005).

[Magruder 2006] M. Magruder, private communication.

[Manson 2005] J. Manson, W. Pugh, and S. Adve. “The Java Memory Model” (*Proceedings of the 32nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, p.378-391, January 12-14, 2005, Long Beach, California, USA).

[Morrison 2005] V. Morrison. “What Every Dev Must Know About Multithreaded Apps” (*MSDN Magazine*, August 2005).

[Morrison 2005a] V. Morrison. “Understand the Impact of Low-Lock Techniques in Multithreaded Apps” (MSDN Magazine, October 2005).

[Morrison 2005b] J. Morrison. “Effects of Compiler and Processor Reordering on Lock-Free Operating System and Application Code” (Windows Reliability, Microsoft internal).

[N1942] H. Boehm et al. “A Memory Model for C++: Strawman Proposal” (ISO/IEC JTC1/SC22/WG21 N1942, February 2006).

[Pugh 2000] W. Pugh. “The Java Memory Model is Fatally Flawed” (*Concurrency—Practice and Experience*, 12(1):1-11, 2000).

[Sutter 2005] H. Sutter. “The Free Lunch Is Over: A Fundamental Turn Toward Concurrency in Software” (*Dr. Dobb's Journal*, 30(3), March 2005).

[Sutter 2005a] H. Sutter and J. Larus. “Software and the Concurrency Revolution” (*ACM Queue*, September 2005).

[Win32prg 2006] “InterlockedIncrement()” (*Windows 32-bit Sys Prgrmmg Questions* discussion thread, June 2006).