# A Proposal for a UPC Memory Consistency Model, v1.1

Lawrence Berkeley National Lab Tech Report LBNL-DRAFT [*]

Katherine Yelick    Dan Bonachea
University of California, Berkeley

Charles Wallace
Michigan Technological University

August 22, 2004

{yelick,bonachea}@cs.berkeley.edu, wallace@mtu.edu [1]

## 1 Introduction

The memory consistency model in a language defines the order in which the results of write operations may be observed through read operations. The behavior of a UPC program may depend on the timing of accesses to shared variables, so a program defines a set of possible executions, rather than a single execution. The memory consistency model constrains the set of possible executions for a given program; the user may then rely on properties that are true of all of those executions.

The memory consistency model is defined in terms of the read and write operations issued by each thread in naïve translation of the code, i.e., without any code transformations by the compiler – with each thread issuing operations as defined by the abstract machine defined in ISO C 5.1.2.3. A UPC compiler or runtime system may perform various code transformations to improve performance, so long as they are not visible to the programmer – i.e. provided the set of externally-visible behaviors (the input/output dynamics and volatile behavior defined in ISO C 5.1.2.3) from any execution of the transformed program are identical to those of the original program executing on the abstract machine and adhering to the consistency model defined in this document.

[*] This document is a draft. The archival version which should be cited in publication is: "A Proposal for a UPC Memory Consistency Model, v1.0", Lawrence Berkeley National Lab Tech Report LBNL-54983

[1] This work is supported in part by the DOE Office of Science and the Hewlett-Packard Company

## 1.1 Executive Summary

*This section will appear in the main body of the language specification, replacing the entire current contents of section 5.1.2.3 and making reference to rest of this document, which will appear as appendix XXX in the language specification.*

The complete and definitive memory consistency model for the language is formally presented in Appendix XXX. However, the following informal summary and rules of thumb may suffice to provide a casual understanding for novice users.

All shared accesses are classified as being either strict or relaxed, as described in sections 6.4.2 and 6.6.1. Accesses to shared objects via pointers-to-local behave as relaxed shared accesses with respect to memory consistency. Most synchronization-related language operations and library functions (notably *upc_fence*, *upc_notify*, *upc_wait*, and *upc_lock*/*upc_unlock*) imply the consistency effects of a strict access.

In general, any sequence of purely relaxed shared accesses issued by a given thread in an execution may appear to be arbitrarily reordered relative to program order by the implementation, and different threads need not agree upon the order in which such accesses appeared to have taken place. The only exception to the previous statement is that two relaxed accesses issued by a given thread to the same memory location where at least one is a write will always appear to all threads to have executed in program order. Consequently, relaxed shared accesses should never be used to perform deterministic inter-thread synchronization - synchronization should be performed using language/library operations whenever possible, or otherwise using only strict shared reads and strict shared writes.

Strict accesses always appear (to all threads) to have executed in program order with respect to other strict accesses, and in a given execution all threads observe the effects of strict accesses in a manner consistent with a single, global total order over the strict operations. Consequently, an execution of a program whose only accesses to shared objects are strict is guaranteed to behave in a sequentially consistent manner.

When a thread's program order dictates a set of relaxed operations followed by a strict operation, all threads will observe the effects of the prior relaxed operations made by the issuing thread (in some order) before observing the strict operation. Similarly, when a thread's program order dictates a strict access followed by a set of relaxed accesses, the strict access will be observed by all threads before the any subsequent relaxed accesses by the issuing thread. Consequently, strict operations can be used to synchronize the execution of different threads, and to prevent the apparent reordering of surrounding relaxed operations across a strict operation.

Most programs will use the strict synchronization facilities provided by the language and library (e.g., barriers, locks, etc) to synchronize threads and prevent non-determinism arising from data races. A data race may occur whenever two or more relaxed operations from different threads access the same location with no intervening strict synchronization, and at least one such access is a write. Programs which produce executions that are always free of data races (as formally defined in Appendix XXX), are guaranteed to behave in a sequentially consistent manner.

# 2 Formal Semantics of Read and Write Operations

## 2.1 Definitions

A UPC program execution is specified by a program text and a number of threads, $T$. An *execution* is a set of operations $O$, each operation being an instance of some instruction in the program text. The set of operations issued by a thread $t$ is denoted $O_t$. The program executes memory operations on a set of variables (or locations) $L$. The set $V$ is the set of possible values that can be stored in the program variables. [2]

A *memory operation* in such an execution is given by a location $l \in L$ to be written or read and a value $v \in V$, which is the value to be written or the value returned by the read. A memory operation $m$ in a UPC program has one of the following forms:

- a strict shared read, denoted SR(l,v)
- a strict shared write, denoted SW(l,v)
- a relaxed shared read, denoted RR(l,v)
- a relaxed shared write, denoted RW(l,v)
- a local read, denoted LR(l,v)
- a local write, denoted LW(l,v)

(Here shared vs local is determined by the sharing type qualification on the expression used to perform the access, and for shared accesses, strict vs relaxed is determined as described in UPC Spec 6.4.2). In addition, each memory operation $m$ is associated with exactly one of the $T$ threads, denoted $\mathit{Thread}(m)$, and we define the accessor $\mathit{Location}(m)$ to return the location $l$ accessed by $m$.

Given a UPC program execution with $T$ threads, let $M \subseteq O$ be the set of memory operations in the execution and $M_t$ be the set of memory operations issued by a given thread $t$. Each operation in $M$ is one of the above six types, so the set $M$ is partitioned into the following six disjoint subsets:

- $SR(M)$ is the set of strict shared reads in $M$
- $SW(M)$ is the set of strict shared writes in $M$
- $RR(M)$ is the set of relaxed shared reads in $M$
- $RW(M)$ is the set of relaxed shared writes in $M$
- $LR(M)$ is the set of local reads in $M$
- $LW(M)$ is the set of local writes in $M$

We denote the set of all writes in $M$ as $W(M) = SW(M) \cup RW(M) \cup LW(M)$ and the set of all strict accesses as $\mathit{Strict}(M) = SR(M) \cup SW(M)$.

[2] This is the point that we could add an atomicity constraint on what types of values are the fundamental unit of a read or write, possibly using something like ISO C's sig_atomic_t. There are actually two separate issues here, namely atomicity and clobbering a.k.a. word tearing.

## 2.2 Memory Access Model

Let $\mathit{StrictPairs}(M)$, $\mathit{StrictOnThreads}(M)$, and $\mathit{AllStrict}(M)$ be unordered pairs of memory operations defined as:

- $\mathit{StrictPairs}(M) \overset{def}{=} \{(m_1, m_2) |\ m_1 \neq m_2\ \wedge\ m_1 \in \mathit{Strict}(M)\ \wedge\ m_2 \in \mathit{Strict}(M)\}$
- $\mathit{StrictOnThreads}(M) \overset{def}{=} \{(m_1, m_2) |\ m_1 \neq m_2\ \wedge\ \mathit{Thread}(m_1) = \mathit{Thread}(m_2)$ $\wedge\ (m_1 \in \mathit{Strict}(M)\ \vee\ m_2 \in \mathit{Strict}(M))\}$
- $\mathit{AllStrict}(M) \overset{def}{=} \mathit{StrictPairs}(M) \cup \mathit{StrictOnThreads}(M)$

Thus, $\mathit{StrictPairs}(M)$ is the set of all pairs of strict memory accesses, including those between threads, and $\mathit{StrictOnThreads}(M)$ is the set of all pairs of memory accesses from the same thread in which at least one is strict. $\mathit{AllStrict}(M)$ is their union, which intuitively is the set of operation pairs on which all threads must agree upon an ordering (i.e., all threads must agree on the directionality of each pair), although what that order is may depend on the resolution of race conditions at runtime. We later define an *ordering* of $\mathit{AllStrict}(M)$ – a set of ordered pairs that contains all pairs in $\mathit{AllStrict}(M)$ but with an orientation for each pair.

UPC programs must preserve the serial dependencies within each thread, defined by the set of ordered pairs $\mathit{DependOnThreads}(M_t)$:

$\mathit{Conflicting}(M) \overset{def}{=} \{(m_1, m_2) |\ \mathit{Location}(m_1) = \mathit{Location}(m_2)\ \wedge\ (m_1 \in W(M)\ \vee\ m_2 \in W(M))\ \}$

$\mathit{DependOnThreads}(M) \overset{def}{=} \{\langle m_1, m_2 \rangle |\ m_1 \neq m_2\ \wedge\ \mathit{Thread}(m_1) = \mathit{Thread}(m_2)\ \wedge\ \mathit{Precedes}(m_1, m_2)$ $\wedge\ (\ (m_1, m_2) \in \mathit{Conflicting}(M)\ \vee\ (m_1, m_2) \in \mathit{StrictOnThreads}(M)\ )\}$

$\mathit{DependOnThreads}(M_t)$ establishes an ordering between operations issued by a given thread $t$ that involve a data dependence (i.e., those operations in $\mathit{Conflicting}(M_t)$) – this ordering is the one maintained by serial compilers and hardware. $\mathit{DependOnThreads}(M_t)$ additionally establishes an ordering between operations appearing in $\mathit{StrictOnThreads}(M_t)$. In both cases, the ordering imposed is the one dictated by $\mathit{Precedes}(m_1, m_2)$, which intuitively is an ordering relationship defined by serial program order. [3] It's important to note that $\mathit{DependOnThreads}(M_t)$ intentionally avoids introducing ordering constraints between non-conflicting, non-strict operations executed by a single thread (i.e., it does not impose ordering between a thread's relaxed/local operations to independent memory locations, or between relaxed/local reads to any location). As we shall later see, this allows implementations to freely reorder any consecutive relaxed/local operations issued by a single thread, except for pairs of operations accessing the same location where at least one is a write; by design this is exactly the condition that is enforced by serial compilers/hardware to maintain sequential data dependences – requiring any stronger ordering property would complicate implementations and likely degrade the performance of relaxed/local accesses. The reason this flexibility must be directly exposed in the model (rather than lumped together with other code transformation optimizations generally permitted by UPC Spec 5.1.2.3) is because the results of this reordering may be "visible" to

[3] DOB: We still need to fill in an appropriate formal definition for $\mathit{Precedes}(m_1, m_2)$, which will probably be derived from the relative location of $m_1$ and $m_2$ in the execution trace of the program executing on the abstract machine. Chuck has previously argued that program order depends on the consistency model and defining the latter in terms of the former leads to a circular definition, so we should provide some justification about why we believe this is a valid approach. It may be useful to note that we don't need to construct the entire set of valid serial execution traces in order to specify $\mathit{Precedes}(m_1, m_2)$. So for example, we needn't try to decide whether or not a given statement in the source program which is control-dependent on some read could possibly be dynamically executed (making such a determination in general requires consulting the memory model, leading to circularity) – the only functionality that $\mathit{Precedes}(m_1, m_2)$ requires is the ability to look at two dynamic operations that WERE executed by a single thread (and can be mapped back to the statement which generated them in the source program), and state which of the two operations must come first in a valid serial execution on the abstract machine.

other threads in the UPC program (as we'll see in later examples) and therefore could impact the program's "input/output dynamics".

A UPC program execution on $T$ threads with memory accesses $M$ is considered *UPC consistent* if there exists a partial order $<_{Strict}$ that provides an orientation for each pair in $AllStrict(M)$ and for each thread $t$, there exists a total order $<_t$ on $O_t \cup W(M) \cup SR(M)$ (i.e. all operations issued by thread $t$ and all writes and strict reads issued by any thread) such that:

1. $<_t$ defines a correct serial execution. [4] In particular:
   - Each read operation returns the value of the "most recent" preceding write to the same location, where "most recent" is defined by $<_t$. If there is no prior write of the location in question, the read returns the initial value of the referenced object as defined by ISO C 6.7.8 and 7.2.0.3 [5]
   - The order of operations in $O_t$ is consistent with the ordering dependencies in $DependOnThreads(M_t)$.

2. $<_t$ is consistent with $<_{Strict}$. In particular, this implies that all threads agree on a total order over the strict operations ($Strict(M)$), and the relative ordering of all pairs of operations issued by a single thread where at least one is strict ($StrictOnThreads(M)$).

For a UPC consistent execution, we say that the set of $<_t$ orderings that satisfy the above constraints are the *enabling orderings* for the execution. There must be at least one ordering from each thread in this set.

[4] Note these definitions of $DependOnThreads(M)$ and $<_t$ provide well-defined consistency semantics for local accesses, essentially making them behave as relaxed accesses. Some further thought may be required to determine whether this is the right decision. Defining the interaction between shared and local accesses has been neglected in earlier versions of the memory model, but we feel this is an important issue to tackle in order to have a complete memory model.

[5] i.e., the initial value of an object declared with an initializer is the value given by the initializer. Objects with static storage duration lacking an initializer have an initial value of zero. Objects with automatic storage duration lacking an initializer have an indeterminate (but fixed) initial value. The initial value for a dynamically allocated object is described by the memory allocation function used to create the object.

# 3 Alternate Semantics Under Consideration

There have been several discussions about what semantics we want for UPC to allow both optimized implementations and a reasonable semantic model for programmers. Here are some alternatives that we may consider.

## 3.1 Maintaining Local Serial Order

The first alternative is a stronger semantics that replaces clause 1 with a stronger (and simpler) rule that the order of operations in $O_t$ is consistent with the program text for $t$. In other words, the operations ordered by $<_t$ behave like the sequential program for $t$ when augmented by the shared writes from other threads.

This definition is attractive because of its conceptual simplicity, but it prevents certain optimizations that we believe are both useful and not too surprising in terms of their semantic implications. Consider the following execution, in which both variables are initially 0:

| | | |
|---|---|---|
| $T0$: | SW(x,1); | SW(y,1) |
| $T1$: | RR(y,1); | RR(x,0) |

The problem is that $T1$ observes an updated value for $y$, but still the old value for $x$. With the revised definition, $T1$ must observe $T0$'s writes in order, because they are both strict, and must also observe its own reads in order. So the execution would not be allowed, which has serious performance implications. For example, $x$ and $y$ cannot be prefetched, if there is a possibility that the actual reads would happen out of order. And if there were a third read in $T1$'s stream $RR(x,0)$ before the first read, then a compiler allocation of $x$ into a register could not leave it there past the read of $y$, even though both operations are relaxed.

## 3.2 Adding Directionality to Reads/Writes

A somewhat weaker, but possibly still acceptable semantics makes the strict read and write operations less symmetric. The original definition can be modified by replacing the definition of $StrictOnThreads(M)$ with the set of unordered pairs:

- $StrictOnThreads(M) \overset{def}{=} \{(m_1, m_2) |\ m_1 \neq m_2\ \wedge\ Thread(m_1) = Thread(m_2)$ $\wedge\ Precedes(m_1, m_2) \wedge\ (\ m_1 \in SR(M)\ \vee\ m_2 \in SW(M)\ \vee\ (m_1, m_2) \in StrictPairs(M))\}$

In this case a strict read operation will prevent reads and writes from moving earlier in the instruction stream; they cannot move past the strict read. A strict write forces all pending reads and writes from that thread to complete before the strict write is performed. In an implementation with software caches, write buffers, or even register value re-use, this will force values back to memory before the strict write operation. In the parlance of release consistency, this essentially assigns *acquire* semantics to strict reads and *release* semantics to strict writes, whereas under the non-directional semantics all strict operations essentially imply both a *release* and an *acquire*.

This definition is similar to some hardware instruction sets, which separate read and write fence operations. A read fence typically prevents subsequent reads from being issued before all prior reads have completed, and a write fence typically forces all prior writes to complete before subsequent writes are issued. There are two important differences in this alternate specification. The first is that these fence-like operations are associated with an actual memory operation, which may not be useful. In practice either the language or the programmer may define a kind of dummy location for read and write fences to get the hardware-style behavior from these memory operations.

The second difference is that a strict read restricts writes as well as reads in the UPC model, so it may be stronger than we wish. A strict read in the instruction stream will indicate that neither reads nor writes can be moved from after the strict read to before it. Similarly, a strict write limits any read or write before the strict write from being moved after it.

(Note: This gives asymmetry in the direction of movement of other operations. If one views the instruction stream as executing from top to bottom, then a strict read prevents movement upward, and a strict write prevents movements downward. Without adding a separate read fence and write fence, this seems like the best we can do, although it's unclear whether there is good evidence to suggest that a read operation is the right place to hang a read fence, or that a write operation is the right place to hang a write fence.)

These asymmetric semantics have a few potentially surprising implications. Here are two executions that are legal under these weaker semantics that are not legal under the vanilla, symmetric semantics (all variables have an initial value of zero):

| | | | | |
|---|---|---|---|---|
| $T0$: | RW(y,1); | SR(x,0) | | |
| $T1$: | | | SW(x,1); | RR(y,0) |

This example demonstrates that under the asymmetric semantics, relaxed operations on different threads are not relatively ordered by a strict write-after-read conflict (an anti-dependence). In contrast, under both semantics an analogous strict read-after-write conflict (true dependence) does order relaxed operations on different threads. Incidentally, this example still works if either (but not both) of the relaxed operations are made strict.

| | | |
|---|---|---|
| $T0$: | SW(x,1); | RR(y,0) |
| $T1$: | SW(y,1); | RR(x,0) |

In this example, all threads agree on the order in which the strict writes took place (as required by both semantics), but one or both of the relaxed reads has been moved before the strict write, taking advantage of the asymmetric semantics and producing a non-intuitive result. This example also still works if either (but not both) of the relaxed operations are made strict.

## 3.3 Resolving Write Conflicts

One of the oddities of both the original spec and this new definition is that they allow write races (i.e. relaxed writes from different threads to the same location with no other synchronization) to be resolved differently by different threads, and never require that they resolve this difference (i.e. lack of coherence). If $T0$ executes $RW(x, 1)$ and in parallel $T1$ executes $RW(x, 2)$, then some threads may observe through relaxed reads that $x$ has been set to 1 and other may observe it has been set to 2. One might expect that at some point, e.g., after barriers, all threads would agree on the values stored in each location (restore coherence), but this is not required by the current semantics (nor is it clear exactly what the right requirement should be).

Perhaps more troubling is the fact that a strict read of the location in question (at any point after the write race, perhaps in the far distant future) will force all threads to agree on the original result of the race (both for the result of the strict read and any intervening relaxed reads). For example, this is a legal execution:

| | | | | |
|---|---|---|---|---|
| $T0$: | RW(x,1); | upc_notify; upc_wait; | RR(x,1) | |
| $T1$: | RW(x,2); | upc_notify; upc_wait; | RR(x,2) | |

But the following is not a legal execution (the only change being the new strict read):

| | | | | |
|---|---|---|---|---|
| $T0$: | RW(x,1); | upc_notify; upc_wait; | RR(x,1); | SR(x,1) |
| $T1$: | RW(x,2); | upc_notify; upc_wait; | RR(x,2) | |

All threads must agree on the value returned by the strict read, and in this case it prevents constructing a legal $<_1$. In essense, the strict read forces the earlier relaxed reads to agree on the result of the race, even though the strict read may occur much later in the execution (with no intervening writes to x).

One place we expect this particular design decision to matter is for implementations employing software caching with an update protocol - requiring coherence at all program points seems likely to make such an implementation strategy prohibitively expensive (although requiring coherence only after barriers seems reasonable). This may be fixable in the definition of the synchronization primitives, which are given below, but it is also possible some more mechanisms will be required in the basic semantics.

# 4 Consistency Semantics of Language and Library Operations

## 4.1 Consistency Semantics of Synchronization Operations

UPC has several synchronization operations that can be used to strengthen the consistency requirements of a program. Some of these involve no explicit synchronization variable or object, so we define these in terms of a fresh variable $l_{synch} \in L$ that does not appear elsewhere in the program. Given this machinery, the memory consistency semantics of the synchronization operations are defined in terms of equivalent memory operations:[6]

- A *upc_fence* operation is equivalent to a strict write followed by a strict read, $SW(l_{synch}, 0)SR(l_{synch}, 0)$.
- A *upc_notify* operation implies a strict write, $SW(l_{synch}, 0)$.
- A *upc_wait* implies a strict read, $SR(l_{synch}, 0)$.
- A *upc_lock* operation of the form *upc_lock*($l_{lock}$) (or a successful *upc_lock_attempt*($l_{lock}$)) implies a strict read $SR(l_{lock}, 0)$, where $l_{lock}$ is a unique location associated with each upc_lock_t object in the execution.
- An *upc_unlock* operation of the form *upc_unlock*($l_{lock}$) implies a strict write, $SW(l_{lock}, 0)$, where $l_{lock}$ is a unique location associated with each upc_lock_t object in the execution.

These represent a slight relaxation to the current language semantics, which state that *upc_lock*, *upc_unlock*, *upc_notify* and *upc_wait* all imply a full *upc_fence*. This relaxation is most relevant if we adopt the asymmetric ordering semantics described in section 3.2, because it permits more aggressive movement of memory operations past synchronization operations.

[6] Note: These definitions do not give the synchronization operations their synchronizing effects – they only define the memory model behavior.

## 4.2 Consistency Semantics of Library Operations

Many of the functions in the UPC standard library can be used to access and modify data in shared objects, either non-collectively (e.g., *upc_mem*{*put*, *get*, *cpy*}) or collectively (e.g., *upc_all_broadcast*, etc). It is important to define the consistency semantics of the accesses to shared objects which are implied to take place within the implementation of these library functions, because they may interact with concurrent explicit reads and writes of the same shared objects. For example, an application which calls a function such as *upc_memcpy* may need to know whether surrounding explicit relaxed operations on non-conflicting shared objects could possibly be reordered relative to the accesses that take place inside the library call. This is a subtle but unavoidable aspect to the library interface which needs to be explicitly defined to ensure that applications can be written with portably deterministic behavior across implementations.

The following sections define the consistency semantics of shared accesses implied by UPC standard library functions, in the absence of any explicit consistency specification for the given function (which would always take precedence in the case of conflict).

### 4.2.1 Consistency Semantics of Non-Collective Library Operations

- For *non-collective* library functions (e.g., *upc_mem*{*put*, *get*, *cpy*}), any implied data accesses to shared objects behave as a set of relaxed shared reads and relaxed shared writes of unspecified size and ordering, issued by the calling thread. No strict operations or fences are implied by a non-collective library function call, unless explicitly noted otherwise.

**Example:**

```
#include <upc_relaxed.h>

shared int x, y;       // initial values are zero
shared [] int z[2];    // initial values are zero
int init_z[2] = { -3, -4 };
...
if (MYTHREAD == 0) {
    x = 1;

    upc_memput(z, init_z, 2*sizeof(int));

    y = 2;
} else {
    #pragma upc strict
    int local_y = y;
    int local_z1 = z[1];
    int local_z0 = z[0];
    int local_x = x;
    ...
}
```

In this example, all of the writes to shared objects are relaxed (including the accesses implied by the library call), and thread 0 executes no strict operations or fences which would inhibit reordering. Therefore, other threads which are concurrently performing strict shared reads of the shared objects ($x, y, z[0]$ and $z[1]$) may observe the updates occuring in any arbitrary order that need not correspond to thread 0's program order. For example, thread 1 may legally end up with $local\_y == 2$, $local\_z1 == -4$, $local\_z0 == 0$ and

$local\_x == 0$, or any other permutation of old and new values for the result of the strict shared reads. Furthermore, because the shared writes implied by library call have unspecified size, thread 1 may even read intermediate values into $local\_z0$ and $local\_z1$ which correspond to neither the initial nor the final values for those shared objects.[7] Finally, note that all of this remains true even if $z$ had instead been declared as:

```
shared strict [] int z[2];
```

because the consistency qualification used on the shared object declarator is irrelevant to the operation of the library call, whose implied shared accesses are specified to always behave as relaxed shared accesses.

If $upc\_fence$ operations were inserted in the blank lines immediately preceding and following the *upc_memput* operation in the example above, then $<_{Strict}$ implies that all reading threads would be guaranteed to observe the shared writes according to thread 0's program order. Specifically, any thread reading a non-initial value into *local_y* would be guaranteed to read the final values for all the other shared reads, and any thread reading the initial zero value into *local_x* would be guaranteed to also have read the initial zero values for all the other shared reads.[8] Explicit use of *upc_fence* immediately preceding and following non-collective library calls operating on shared objects is the recommended method for ensuring ordering with respect to surrounding relaxed operations issued by the calling thread, in the rare cases where such ordering guarantees are required for program correctness.

### 4.2.2 Consistency Semantics of Collective Library Operations

- For *collective* functions in the UPC standard library, any implied data accesses to shared objects behave as a set of relaxed shared reads and relaxed shared writes of unspecified size and ordering, issued by one or more unspecified threads (unless explicitly noted otherwise).
- For *collective* functions in the UPC standard library that take a $upc\_flag\_t$ argument (e.g., *upc_all_broadcast*), one or more $upc\_fence$ operations may be implied upon entry and/or exit to the library call, based on the flags selected in the value of the $upc\_flag\_t$ argument, as follows:
  - `UPC_IN_ALLSYNC` and `UPC_IN_MYSYNC` imply a $upc\_fence$ operation on each calling thread, immediately upon entry to the library function call.
  - `UPC_OUT_ALLSYNC` and `UPC_OUT_MYSYNC` imply a $upc\_fence$ operation on each calling thread, immediately before return from the library function call.
  - No fence operations are implied by `UPC_IN_NOSYNC` or `UPC_OUT_NOSYNC`.

The $upc\_fence$ operations implied by the rules above are sufficient to ensure the results one would naturally expect in the presence of relaxed shared reads and writes issued immediately preceding or following an `ALLSYNC` or `MYSYNC` collective library call that accesses the same shared objects. Without such fences, nothing would prevent prior or subsequent relaxed shared operations explicitly issued by the calling thread from being reordered relative to some of the accesses implied by the library call (which might not be issued by the current thread), potentially leading to very surprising and unintuitive results. The `NOSYNC` flag provides no synchronization guarantees between the execution stream of the calling thread and the shared accesses implied by the collective library call, therefore no additional fence operations are required. [9]

[7] This issue is a consequence of the byte-oriented nature of the shared data movement functions (which we assume in the absense of further specification) and will remain regardless of how we resolve the related but orthogonal issue of write atomicity.

[8] However, for threads reading the initial value into *local_y* and the final value into *local_x*, the writes to $z[0]$ and $z[1]$ could still appear to have been arbitrarily reordered or segmented, leading to indeterminate values in *local_z*0 and *local_z*1.

[9] Any deterministic program which makes use of `NOSYNC` collective data movement functions is likely to be synchronizing access to shared objects via other means – for example, through the use of explicit *upc_barrier* or `ALLSYNC`/`MYSYNC` collective calls that already provide sufficient synchronization and fences.

# 5 Properties Implied by the Specification

The memory model definition is fairly subtle in some points, but most programmers need not worry about these details. There are some simple properties that are helpful in understanding the semantics. The first is:

- A UPC program which accesses shared objects using only strict operations [10] will be sequentially consistent.

This property is trivially true due to the global total order that $<_{Strict}$ imposes over strict operations (which is respected in every thread's $<_t$), but is not very useful in practice – because a UPC program written entirely with strict accesses is likely to be quite slow. However, it may be a useful debugging tool because even in the presence of data races, a fully strict program is guaranteed to only produce behaviors possible under sequential consistency (which is the easiest memory model to understand and the one which naïve programmers typically assume).

Of more interest is that programs free of race conditions will also be sequentially consistent. This requires a more formal definition of race condition, because programmers may believe their program is properly synchronized using memory operations when it is not.

We define a set $PotentialRaces(M)$ as unordered pairs $(m_1, m_2)$:

- $PotentialRaces(M) \overset{def}{=} \{(m_1, m_2) | \ Location(m_1) = Location(m_2) \ \wedge \ Thread(m_1) \neq Thread(m_2) \ \wedge \ (m_1 \in W(M) \ \vee \ m_2 \in W(M))\}$

An execution is race-free if every $(m_1, m_2) \in PotentialRaces(M)$ is ordered by $<_{Strict}$. i.e., an execution is race-free if and only if:

- $\forall (m_1, m_2) \in PotentialRaces(M) : m_1 <_{Strict} m_2 \ \vee \ m_2 <_{Strict} m_1$.

Note this implies that all threads $t$ and all enabling orderings $<_t$ agree upon the ordering of each $(m_1, m_2) \in PotentialRaces(M)$ (so there is no race).

These definitions allow us to state a very useful property of UPC programs:

- A program that produces only race-free executions will be sequentially consistent.

Note that UPC locks and barriers constrain $PotentialRaces$ as one would expect, because these language-level synchronization ops imply strict operations which introduce orderings in $<_{Strict}$ for the operations in question.

[10] i.e., no relaxed shared accesses, and accesses to shared objects via pointers-to-local

# 6 Examples

In the figures below, each execution is shown by the linear graph which is the *Precedes*$(M)$ program order for each thread, generated by an execution of the source program on the abstract machine. Pairs of memory operations that are ordered by the global ordering over memory operations in *AllStrict*$(M)$ (i.e. $m_1 <_{Strict} m_2$) are shown here as $m_1 \Rightarrow m_2$. All threads must agree on the relative ordering imposed by these edges in their $<_t$ orderings. Pairs ordered by a thread $t$ as in $m_1 <_t m_2$ are represented by $m_1 \rightarrow m_2$. Arcs that are implied by transitivity are omitted. Assume all variables are initialized to 0.

1. **Legal behavior** that would not be legal under sequential consistency. There are only relaxed operations, so the threads need not observe the program order by other threads. Because all operations are relaxed, there are no $\Rightarrow$ orderings between operations.

   $T0$: RR(x,1); RW(x,2)
   $T1$: RR(x,2); RW(x,1)

   $<_0$:

   $$\begin{tikzcd}
   RR(x,1) \arrow[r] & RW(x,2) \\
   & RW(x,1) \arrow[ul]
   \end{tikzcd}$$

   $T0$ observes $T1$'s write happening before its own read.

   $<_1$:

   $$\begin{tikzcd}
   & RW(x,2) \arrow[dl] \\
   RR(x,2) \arrow[r] & RW(x,1)
   \end{tikzcd}$$

   $T1$ must observe its own program order for conflicting operations, but it sees $T0$'s write as the first operation.

   Note that relaxed reads issued by thread $t$ only appear in the $<_t$ of that thread.

2. **Illegal behavior**, which is the same as the previous example, but with all accesses marked strict. All edges in the graph below must therefore be $\Rightarrow$ edges. This also implies the program order edges must be observed and the two threads must agree on the order of the races. The use of unique values in the writes for this example forces an orientation of the cross-thread edges, so an acyclic $<_{Strict}$ cannot be defined that satisfies the write-to-read data flow requirements for a legal $<_t$.

   $T0$: SR(x,1); SW(x,2)
   $T1$: SR(x,2); SW(x,1)

   $<_{Strict}$:

   $$\begin{tikzcd}
   SR(x,1) \arrow[r, Rightarrow] & SW(x,2) \arrow[dl, Rightarrow] \\
   SR(x,2) \arrow[r, Rightarrow] & SW(x,1) \arrow[ul, Rightarrow]
   \end{tikzcd}$$

   All of the edges above are required, but this is not a legal $<_{Strict}$, since it contains a cycle.

3. **Legal behavior** that would, as in the first example, not be legal if all of the accesses were strict. Again one thread may observe the other's operations happening out of program order. This is the pattern of memory operations that one might see with a spinlock, where $y$ is the lock protecting the variable $x$. The implication is that UPC programmers should not build synchronization out of relaxed operations.

   $T0$: RW(x,1); RW(y,1)
   $T1$: RR(y,1); RR(x,0)

   $<_0$:

$$\begin{tikzcd} RW(x,1) \arrow[r] & RW(y,1) \end{tikzcd}$$

   $T0$ observes only its own writes.
   The writes are non-conflicting, so either ordering constitutes a legal $<_0$.

   $<_1$:

$$\begin{tikzcd} RW(x,1) & RW(y,1) \arrow[dl] \\ RR(y,1) \arrow[r] & RR(x,0) \arrow[ul] \end{tikzcd}$$

   To satisfy write-to-read data flow in $<_1$, RW(x,1) must follow RR(x,0) and RR(y,1) must follow RW(y,1). There are three other legal $<_1$ orderings which satisfy these constraints.

4. **Legal behavior** that would not be legal under sequential consistency. This example is similar to the previous ones, but involves a read-after-write on each processor. Neither thread sees the update by the other, but in the $<_t$ orderings, each thread conceptually observes the other threads operations happening out of order.

   $T0$: RW(x,1); RR(y,0)
   $T1$: RW(y,1); RR(x,0)

   $<_0$:

$$\begin{tikzcd} RW(x,1) \arrow[r] & RR(y,0) \arrow[dl] \\ RW(y,1) & \end{tikzcd}$$

   The only constraint on $<_0$ is RW(y,1) must follow RR(y,0). Several other legal $<_0$ orderings are possible.

   $<_1$:

$$\begin{tikzcd} RW(x,1) & \\ RW(y,1) \arrow[r] & RR(x,0) \arrow[ul] \end{tikzcd}$$

   Analogous situation with a write-after-read, this time on x. Several other legal $<_1$ orderings are possible.

5. **Illegal behavior**, since with strict accesses, one of the two writes must "win" the race condition. Each thread observes the other thread's write happening after its own write, which creates a cycle when we attempt to construct $<_{Strict}$.

   $T0$: SW(x,2); SR(x,1)
   $T1$: SW(x,1); SR(x,2)

$<_{Strict}$:

$$\begin{tikzcd}
SW(x,2) \arrow[r, Rightarrow] \arrow[d, Leftrightarrow] & SR(x,1) \\
SW(x,1) \arrow[r, Rightarrow] & SR(x,2)
\end{tikzcd}$$

6. **Legal behavior**, where a thread observes its own reads occurring out-of-order. Reordering of reads is commonplace in serial compilers/hardware, but in this case an intervening modification by a different thread makes this reordering visible. Strengthening the model to prohibit such reordering of conflicting relaxed reads would impose serious restrictions on the implementation of relaxed reads that would likely degrade performance - for example, under such a model an optimizer could not reorder the reads in this example (or allow them to proceed as concurrent non-blocking operations if they might be reordered in the network) unless it could statically prove the reads were non-conflicting or no other thread was writing the location.

| | | | |
|---|---|---|---|
| $T0$: | RW(x,1); | SW(y,1); | RW(x,2) |
| $T1$: | RR(x,2); | RR(x,1) | |

$<_{Strict}$:

$$\begin{tikzcd}
RW(x,1) \arrow[r, Rightarrow] & SW(y,1) \arrow[r, Rightarrow] & RW(x,2)
\end{tikzcd}$$

*DependOnThreads*$(M_0)$ implies this is the only legal $<_{Strict}$ ordering over *StrictOnThreads*$(M)$

$<_0$:

$$\begin{tikzcd}
RW(x,1) \arrow[r, Rightarrow] \arrow[r, bend left] & SW(y,1) \arrow[r, Rightarrow] \arrow[r, bend left] & RW(x,2)
\end{tikzcd}$$

$<_0$ conforms to $<_{Strict}$

$<_1$:

$$\begin{tikzcd}
RW(x,1) \arrow[r, Rightarrow] \arrow[dr] & SW(y,1) \arrow[r, Rightarrow] \arrow[r, bend left] & RW(x,2) \arrow[dll] \\
RR(x,2) & RR(x,1) \arrow[u] &
\end{tikzcd}$$

$<_1$ conforms to $<_{Strict}$. T1's operations on x do not conflict because they are both reads, and hence may appear relatively reordered in $<_1$. One other $<_1$ ordering is possible.

7. **Illegal behavior**, similar to the previous example, but in this case the addition of a relaxed write on thread 1 introduces dependencies in *DependOnThreads*$(M_1)$, such that (all else being equal) T1's second read may only legally return the value 3. If T1's write were to any location other than x, the behavior shown would be legal.

| | | | |
|---|---|---|---|
| $T0$: | RW(x,1); | SW(y,1); | RW(x,2) |
| $T1$: | RR(x,2); | RW(x,3); | RR(x,1) |

$<_{Strict}$:

$$\begin{tikzcd}
RW(x,1) \arrow[r, Rightarrow] & SW(y,1) \arrow[r, Rightarrow] & RW(x,2)
\end{tikzcd}$$

*DependOnThreads*$(M_0)$ implies this is the only legal $<_{Strict}$ ordering over *StrictOnThreads*$(M)$

$<_0$:

$$\begin{tikzcd}
RW(x,1) \arrow[r, Rightarrow] \arrow[r, bend left] & SW(y,1) \arrow[r, Rightarrow] \arrow[r, bend left] & RW(x,2) \arrow[dll] \\
RW(x,3) & &
\end{tikzcd}$$

$<_0$ conforms to $<_{Strict}$. Other orderings are possible.

$<_1$:

$$\begin{tikzcd}
RW(x,1) \arrow[r, Rightarrow] \arrow[r, bend left] & SW(y,1) \arrow[r, Rightarrow] \arrow[r, bend left] & RW(x,2) \arrow[dll] \\
RR(x,2) \arrow[r] & RW(x,3) \arrow[r] & RR(x,?)
\end{tikzcd}$$

This is the only $<_1$ that conforms to $<_{Strict}$ and $DependOnThreads(M_1)$. The second read of x cannot return 1 - it must be 3.

8. **Illegal behavior** Demonstrating why strict reads appear in every $<_t$, rather than just for the thread that issued them. If the strict reads were absent from $<_0$, this behavior would be legal.

| | | |
|---|---|---|
| $T0$: | RW(x,1); | RW(x,2) |
| $T1$: | SR(x,2); | SR(x,1) |

$<_{Strict}$:

$$\begin{tikzcd}
SR(x,2) \arrow[r, Rightarrow] & SR(x,1)
\end{tikzcd}$$

$DependOnThreads(M_1)$ implies this is the only legal $<_{Strict}$ ordering over $StrictOnThreads(M)$

$<_0$:

$$\begin{tikzcd}
RW(x,1) \arrow[r] & RW(x,2) \arrow[dl] \\
SR(x,2) \arrow[r, Rightarrow] \arrow[r, bend left] & SR(x,?)
\end{tikzcd}$$

This is the only $<_0$ that conforms to $<_{Strict}$ and $DependOnThreads(M_0)$. The second read of x cannot return 1 - it must be 2.

9. **Legal behavior** Similar to the previous example, but the writes are no longer conflicting, and therefore not ordered by $DependOnThreads(M_0)$.

| | | |
|---|---|---|
| $T0$: | RW(x,1); | RW(y,1) |
| $T1$: | SR(y,1); | SR(x,0) |

$<_{Strict}$:

$$\begin{tikzcd}
SR(y,1) \arrow[r, Rightarrow] & SR(x,0)
\end{tikzcd}$$

$DependOnThreads(M_1)$ implies this is the only legal $<_{Strict}$ ordering over $StrictOnThreads(M)$

$<_0, <_1$:

$$\begin{tikzcd}
RW(x,1) & RW(y,1) \arrow[dl] \\
SR(y,1) \arrow[r, Rightarrow] \arrow[r, bend left] & SR(x,0) \arrow[ul]
\end{tikzcd}$$

The writes are non-conflicting, therefore not ordered by $DependOnThreads(M_0)$.

10. **Legal behavior** Another example of a thread observing its own relaxed reads out of order, regardless of location accessed.

| | | | |
|---|---|---|---|
| $T0$: | RW(x,1); | SW(y,1) | |
| $T1$: | RR(y,1); | RR(x,1); | RR(x,0) |

$<_{Strict}$:

$$\begin{tikzcd} RW(x,1) \arrow[r, Rightarrow] & SW(y,1) \end{tikzcd}$$

$DependOnThreads(M_0)$ implies this is the only legal $<_{Strict}$ ordering over $StrictOnThreads(M)$

$<_0$:

$$\begin{tikzcd} RW(x,1) \arrow[r, Rightarrow] \arrow[r, bend left] & SW(y,1) \end{tikzcd}$$

Relaxed reads from thread 1 do not appear in $<_0$

$<_1$:

$$\begin{tikzcd} RW(x,1) \arrow[r, Rightarrow] \arrow[r, bend left] & SW(y,1) \arrow[dl] & \\ RR(y,1) \arrow[r] & RR(x,1) & RR(x,0) \arrow[ull] \end{tikzcd}$$

Relaxed reads have been reordered. Other $<_1$ orders are possible.

11. **Illegal behavior** Demonstrating that a barrier synchronization orders relaxed operations as one would expect.

| | | | | |
|---|---|---|---|---|
| $T0$: | RW(x,1); | upc_notify; | upc_wait | |
| $T1$: | | upc_notify; | upc_wait; | RR(x,0) |

$<_{Strict}$:

$$\begin{tikzcd} RW(x,1) \arrow[r, Rightarrow] & \begin{array}{c}\text{upc\_notify}\\(=SW*)\end{array} \arrow[r, Rightarrow] \arrow[d, Rightarrow] \arrow[dr, Rightarrow] & \begin{array}{c}\text{upc\_wait}\\(=SR*)\end{array} & \\ & \begin{array}{c}\text{upc\_notify}\\(=SW*)\end{array} \arrow[r, Rightarrow] \arrow[ur, Rightarrow] & \begin{array}{c}\text{upc\_wait}\\(=SR*)\end{array} \arrow[r, Rightarrow] \arrow[u, Rightarrow] & RR(x,0) \end{tikzcd}$$

$DependOnThreads(M)$ and the synchronization semantics of barrier imply that $<_{Strict}$ must respect all the edges shown (except the edge between the upc_wait's and the edge between the upc_notify's, both of which can point either way).

There is no legal $<_1$ which respects $<_{Strict}$ – write-to-read data flow along the chain $RW(x,1) \Rightarrow upc\_notify \Rightarrow upc\_wait \Rightarrow RR(x,0)$ implies the read must return 1 (i.e., because $RW(x,1) <_{Strict} RR(x,0)$ and there are no intervening writes of x).

12. **Illegal behavior** $<_{Strict}$ is an ordering over the pairs in $AllStrict(M)$, which includes an edge between two upc_notify operations. Every $<_t$ must conform to a single $<_{Strict}$ ordering – all threads agree on a single total order over $SR(M) \cup SW(M)$ in general, and in particular they all agree on the order of upc_notify operations. Therefore, at least one of the read operations must return 1.

| | | | | |
|---|---|---|---|---|
| $T0$: | RW(x,1); | upc_notify; | RR(y,0); | (upc_wait not shown) |
| $T1$: | RW(y,1); | upc_notify; | RR(x,0); | (upc_wait not shown) |

$<_{Strict}$:

$$\begin{tikzcd} RW(x,1) \arrow[r, Rightarrow] & \begin{array}{c}\text{upc\_notify}\\(=SW*)\end{array} \arrow[r, Rightarrow] \arrow[d, Rightarrow] & RR(y,0) \\ RW(y,1) \arrow[r, Rightarrow] & \begin{array}{c}\text{upc\_notify}\\(=SW*)\end{array} \arrow[r, Rightarrow] & RR(x,0) \end{tikzcd}$$

$DependOnThreads(M_0)$ implies these edges in $StrictOnThreads(M)$ must be respected by $<_{Strict}$ (except the edge between the upc_notify's which can point either way).

$<_0$:

$$\begin{tikzcd}
RW(x,1) \arrow[r, Rightarrow] \arrow[r, bend left] & \begin{array}{c}\text{upc\_notify}\\(=SW*)\end{array} \arrow[r, Rightarrow] \arrow[r, bend left] \arrow[d, Rightarrow] & RR(y,0) \arrow[dll] \\
RW(y,1) \arrow[r, Rightarrow] \arrow[r, bend left] & \begin{array}{c}\text{upc\_notify}\\(=SW*)\end{array} &
\end{tikzcd}$$

$<_1$:

$$\begin{tikzcd}
RW(x,1) \arrow[r, Rightarrow] & \begin{array}{c}\text{upc\_notify}\\(=SW*)\end{array} \arrow[d, Rightarrow] & \\
RW(y,1) \arrow[r, Rightarrow] & \begin{array}{c}\text{upc\_notify}\\(=SW*)\end{array} \arrow[r, Rightarrow] & RR(x,0)
\end{tikzcd}$$

Read cannot return 0.

There is no legal $<_1$ which respects $<_{Strict}$ – write-to-read data flow along the chain $RW(x,1) \Rightarrow upc\_notify \Rightarrow upc\_notify \Rightarrow RR(x,0)$ implies the read must return 1 (i.e., because $RW(x,1) <_{Strict} RR(x,0)$ and there are no intervening writes of x). Reversing the edge between the upc_notify's in $<_{Strict}$ causes the same problem for y in $<_0$.

Note that under the alternate asymmetric semantics proposed in section 3.2, this behavior would be legal (because one or both of the relaxed reads could be moved earlier than the upc_notify's). [11]

[11] CW: The individual upc_notify's in a single collective synchronization operation are totally ordered. I think this is undesirable, as it enforces synchronization "too early". Consider the following example:

$T0$: RW(x,1); upc_notify; RW(x,2); RR(x,3)
$T1$: RW(x,3); upc_notify; RW(x,4); RR(x,1)

I think this should be allowed, since upc_notify by itself doesn't imply any synchronization; there's no need for T0 to be aware of T1's write, and vice versa.. But if the upc_notify's are ordered, one of the two reads will be disallowed. (DOB: again, this is not a problem under the alternate asymmetric semantics.)

# 7 Miscellaneous Open Issues

1. **Write atomicity, clobbering, tearing and overlap**
   We currently say nothing about the atomicity of writes (e.g. in a multi-byte write, can other threads read a partially-written value, and if so what are the possible values). A related effect is write "tearing", where two writes in a race condition could each cause some of the bytes to be modified. We also say nothing about write clobbering (e.g. when different threads write to disjoint, but adjacent bytes in a shared array, can the writes "clobber" each other). These issues arise from inescapable properties of modern architectures, and we need to say something about them. Finally, we need to address the behavior of overlapping accesses such as read/writes of a struct field vs. read/writes of the entire struct - the current definition of $Conflicting()$ doesn't really accomodate such accesses, and we need a way to explain how data from these "subset" accesses flow from writes to reads in $<_t$.

   We should investigate how these issues are handled in other related memory models (eg Java memory model). One idea for addressing the clobbering issue is to require that implementations provide one integer and one floating point type that are adequately padded to guarantee that accesses to adjacent values of that type in an array will never lead to clobbering. Eg:

   ```
   typedef struct {
     int val;
     char __pad[...]; // implementation-specific amount of padding
   } upc_noclobber_int_t;

   typedef struct {
     double val;
     char __pad[...]; // implementation-specific amount of padding
   } upc_noclobber_double_t;
   ```

2. **Provide an equivalent operational semantics**
   It would be instructive and useful to develop an equivalent formulation of this declarative memory consistency model, defined in an operational manner (perhaps using abstract state machines). Despite the well-known problems with using an operational approach to formal specification (for a particularly horrendous example, see Java's memory model), they are occasionally useful for certain purposes. Also, the exercise of constructing such a model and formally proving its equality with the declarative model should provide deeper insights into both formulations.

3. **Add more rules of thumb to Section 4.2.2**
   Although precision and correctness are the primary goals of this spec (and consistency models are notoriously subtle), we'd very much like to have a model which is understandable to our users. Towards this end, the implications section attempts to explain (in English!) properties that users can rely on, although it wouldn't hurt to add a few more "rules of thumb" that arise from the formal semantics (e.g., "don't use relaxed ops to perform synchronization").

   Interestingly enough, although not recommended it is possible to build race free synchronization using only relaxed ops and fence - for example:

   ```
     Thread 0: RW(data1,v1); RW(data2,v2); fence; RW(flag,1);
     Thread 1: while (RR(flag)==0) (poll); fence; RR(data1,v1); RR(data2,v1)
   ```

   Thread 1 is guaranteed to always see v1 and v2 properly written – we've essentially built a pairwise synchronization by combining relaxed read/writes with fence to make them behave somewhat like strict read/writes.

4. **Causal consistency**

The current model (and to our knowledge, all previously proposed UPC memory models) suffer from a lack of causal consistency.

One example (x and y are shared variables initialized to 0):

$T0$: if (x != 0) y = 1;
$T1$: if (y != 0) x = 1;

Under sequential consistency (e.g., where both x and y are strict) the writes will never execute, and therefore the program is trivially free of data races (i.e., "correctly synchronized"). However, when x and y are relaxed, it is possible that both writes will occur - specifically, if we start from the assumption that one write will occur, then we can construct a legal ordering where the write causes itself to occur (a causality loop). Under our model, this essentially means the program is not free of data races when the variables are relaxed, although a casual inspection may lead one to believe otherwise. For an in-depth discussion, see Manson & Pugh's "Multithreaded Semantics for Java, Revised".

Another example of the current model's lack of causal consistency is the "out-of-thin-air" example. Consider the following code, where x and y are shared variables (initially both zero), r1/r2 are registers, and all accesses are relaxed:

$T0$: r1 = x; y = r1;
$T1$: r2 = y; x = r2;

The problem is nothing in the current model forbids an execution such as the following, which causes an arbitrary value (e.g., 42) to appear "out-of-thin-air":

$T0$: RR(x,42); RW(y,42)
$T1$: RR(y,42); RW(x,42)

$<_0$:

$$\begin{tikzcd}
RR(x,42) \arrow[r] & RW(y,42) \\
 & RW(x,42) \arrow[ul]
\end{tikzcd}$$

Each thread observes the other's write preceding its own read.

$<_1$:

$$\begin{tikzcd}
 & RW(y,42) \arrow[dl] \\
RR(y,42) \arrow[r] & RW(x,42)
\end{tikzcd}$$

One interpretation is that T0 "guesses" the value of the read into r1 will be 42 and writes that value into Y. T1 then reads that value from Y and sets X to 42. T0 then checks that its prediction that r1 = 42 is true by doing the real read of X, which at this point is indeed 42.

Another way to understand this problem is that our spec currently allows some code transformations based on assumptions which only become true after the transformation. In other words, T0 arbitrarily decides the read will return 42, and starting from the basis of that assumption one can "cause" it to become true - a causality loop.

The new Java memory model solution to these types of problems is to introduce a causal consistency requirement, in order to forbid these degenerate interpretations. The advantage is it makes the spec more explicit about what transformations are legal/forbidden, the disadvantage is that it adds plenty of spec complexity. Users probably will never care (or never even think about such an example) but people writing aggressive optimizations might need to refer to a causal consistency property to decide whether a given transformation is legal.

5. **Lock fairness**
   Neither the memory consistency model or the language specification currently say anything about upc_lock_t fairness. Specifically, this means that an implementation would be within its rights to transform a program like:

   ```
   shared strict int flag = 0;
   if (MYTHREAD == 0) {
     while (1) {
       upc_lock(&mylock);
       if (flag == 1) break;
       upc_unlock(&mylock);
     }
   } else {
     upc_lock(&mylock);
     flag = 1;
     upc_unlock(&mylock);
   }
   ```

   into the following program, which is likely to deadlock:

   ```
   shared strict int flag = 0;
   if (MYTHREAD == 0) {
     upc_lock(&mylock);
     while (1) {
       if (flag == 1) break;
     }
     upc_unlock(&mylock);
   } else {
     upc_lock(&mylock);
     flag = 1;
     upc_unlock(&mylock);
   }
   ```

   This is related to the question of progress, about which the spec currently provides no guarantees.

6. **Add** *Precedes*() **definition**
   As noted in section 2.2, we need to add a definition of *Precedes*(). The intuition is clear and we have some initial ideas on how this formalism might look, but haven't settled on the final solution yet.