Relaxed memory models must be rigorous

Francesco Zappa Nardelli[1] Peter Sewell[2] Jaroslav Ševčík[3] Susmit Sarkar[2] Scott Owens[2]
Luc Maranget[1] Mark Batty[2] Jade Alglave[1]

[1]INRIA [2]University of Cambridge [3]University of Edinburgh

Abstract

Multiprocessors and high-level languages generally provide only *relaxed* (non-sequentially-consistent) memory models, to permit performance optimisations. One has to understand these models to program reliable concurrent systems — but they are typically ambiguous and incomplete informal-prose documents, sometimes give guarantees that are too weak to be useful, and are sometimes simply unsound. Based on our previous work, we review various problems with some current specifications, for x86 (Intel 64/IA32 and AMD64), and Power and ARM processors, and for the Java and C++ languages. We argue that such specifications should be rigorously defined and tested.

The Black Magic of Multiprocessor Programming

Parallelism is finally going mainstream, but, despite 40 years of research on concurrency, programming and reasoning about concurrent systems remains very challenging.

A key issue is that most programmers (and most researchers) assume that memory is sequentially consistent: that accesses by multiple threads to a shared memory occur in a global-time linear order. Real multiprocessors and compilers, however, incorporate many performance optimisations. These are typically unobservable by single-threaded programs, but some have observable consequences for the behaviour of concurrent code. For example, on standard Intel or AMD x86 processors, given two memory locations x and y (initially holding 0), if two processors proc:0 and proc:1 respectively write 1 to x and y and then read from y and x, as in the program below, it is possible for both to read 0 in the same execution.

iwp2.3.a/amd4	proc:0	proc:1
poi:0	MOV [x]←$1	MOV [y]←$1
poi:1	MOV EAX←[y]	MOV EBX←[x]
Allow: 0:EAX=0 ∧ 1:EBX=0		

One can view this as a visible consequence of *write buffering*: each processor effectively has a FIFO buffer of pending memory writes (to avoid the need to block while a write completes), so the reads from y and x can occur before the writes have propagated from the buffers to main memory. Such optimisations (of which this is a particularly simple example) destroy the illusion of sequential consistency, making it impossible (at this level of abstraction) to reason in terms of an intuitive notion of global time.

To describe what programmers can rely on, processor vendors document *architectures*. These are loose specifications, claimed to cover a range of past and future actual processors, which should reveal enough for effective programming, but without unduly constraining future processor designs. In practice, however, they are typically informal-prose documents, e.g. the Intel 64 and IA-32 Architectures SDM [Int09], the AMD64 Architecture Programmer's Manual [AMD07], or the Power ISA specification [Pow07] (SPARC and Itanium have somewhat clearer semi-formal memory models). Informal prose is a poor medium for loose specification of subtle properties, and, as we shall see, such documents are often ambiguous, sometimes incomplete (too weak to program above), and sometimes unsound (forbidding behaviour that the actual processors allow). Moreover, one cannot test programs above such a vague specification (one can only run programs on particular actual processors), and one cannot use them as criteria for testing processor implementations.

Further, different processor families (Intel 64/IA-32 and AMD64, PowerPC, SPARC, Alpha, Itanium, ARM, etc.) allow very different reorderings, and an algorithm that behaves correctly above one may be incorrect above another.

Using standard high-level languages to program multiprocessor systems does not give immunity to these problems. High-level language statements may require several accesses to main memory to complete, and the hardware can reorder these as above, with unexpected results. Even worse, compiler optimisations, which are semantically invisible for single-threaded programs, may become observable for multi-threaded code. There have been attempts at defininining memory models at the language level for Java [Pug00, JSR] and, more recently, C++ [BA08, cpp08]. Unfortunately, these also suffer from ambiguities and cannot be used for testing, and those for Java have been shown to be unsound.

A Short Tour of Some Real-World Memory Models

We review the specifications of the memory models of some recent, widely used, processors and high-level programming languages, based on our previous work [SSZN+09, AFI+09, OSS09, vA08]. As we shall see, for now, programming correct concurrent algorithms is really black magic.

Intel 64/IA32 and AMD64 There have been several versions of the Intel and AMD documentation, some differing radically; we contrast them with each other, and with our knowledge of the behaviour of the actual processors.

Pre-IWP (before Aug. 2007) Early revisions of the Intel SDM (e.g. rev-22, Nov. 2006) gave an informal-prose model called 'processor ordering', unsupported by any litmus-test examples. It is hard to give a precise interpretation of this description, as illustrated by the animated discussion between Linux kernel developers on how to correctly implement spinlocks [Lin99].

IWP/AMD64-3.14/x86-CC In August 2007, an Intel White Paper (IWP) [Int07] gave a somewhat more precise model, with 8 informal-prose principles supported by 10 litmus tests. This was incorporated, essentially unchanged, into later revisions of the Intel SDM (including rev.26–28), and AMD gave similar, though not identical, prose and tests [AMD07]. These are essentially causal-consistency models. They allow independent readers to see independent

writes (by different processors to different addresses) in different orders, as in the IRIW example below [BA08]),

proc:0	proc:1	proc:2	proc:3
MOV [x]←$1	MOV [y]←$1	MOV EAX←[x] MOV EBX←[y]	MOV ECX←[y] MOV EDX←[x]
Final: 2:EAX=1 ∧ 2:EBX=0 ∧ 3:ECX=1 ∧ 3:EDX=0			

but require that, in some sense, causality is respected: *"P5. In a multiprocessor system, memory ordering obeys causality (memory ordering respects transitive visibility)"*. These were the basis for our x86-CC model [SSZN$^+$09], for which a key issue was giving a reasonable interpretation to this "causality". Apart from that, the informal specifications were reasonably unambiguous — but they turned out to have two serious flaws.

First, they are arguably rather weak for programmers. In particular, they admit the IRIW behaviour above but, under reasonable assumptions on the strongest x86 memory barrier, MFENCE, adding MFENCEs would not suffice to recover sequential consistency [SSZN$^+$09, §2.12]. Here the specifications seem to be much looser than the behaviour of implemented processors: to the best of our knowledge, and following some testing, IRIW is not observable in practice. It appears that some JVM implementations depend on this fact, and would not be correct if one assumed only the IWP/AMD64-3.14/x86-CC architecture [Dic08].

Second, more seriously, they are unsound with respect to current processors. The following n6 example, due to Paul Loewenstein [Loe08], shows a behaviour that is observable (e.g. on an Intel Core 2 duo), but that is disallowed by x86-CC, and by any interpretation we can make of IWP and AMD64-3.14.

n6	proc:0	proc:1
poi:0	MOV [x]←$1	MOV [y]←$2
poi:1	MOV EAX←[x]	MOV [x]←$2
poi:2	MOV EBX←[y]	
Final: 0:EAX=1 ∧ 0:EBX=0 ∧ [x]=1		
cc : Forbid; tso : Allow		

To see why this may be allowed by multiprocessors with FIFO write buffers, suppose that first the proc:1 write of [y]=2 is buffered, then proc:0 buffers its write of [x]=1, reads [x]=1 from its own write buffer, and reads [y]=0 from main memory, then proc:1 buffers its [x]=2 write and flushes its buffered [y]=2 and [x]=2 writes to memory, then finally proc:0 flushes its [x]=1 write to memory.

Intel SDM rev-29 (Nov. 2008) and rev-30 (Mar. 2009) The most recent x86 vendor specifications, at the time of writing, are revisions 29 and 30 of the Intel SDM (these are essentially identical, and we are told that there will be a future revision of the AMD specification on similar lines). They are in a similar informal-prose style to previous versions, again supported by litmus tests, but are significantly different to IWP/AMD64-3.14/x86-CC. First, the IRIW final state above is forbidden [Example 7-7, rev-29], and the previous coherence condition: *"P6. In a multiprocessor system, stores to the same location have a total order"* has been replaced by: *"P9. Any two stores are seen in a consistent order by processors other than those performing the stores"*.

Second, the memory barrier instructions are now included, with *"P11. Reads cannot pass LFENCE and MFENCE instructions"* and *"P12. Writes cannot pass SFENCE and MFENCE instructions"*.

Third, same-processor writes are now explicitly ordered (we regarded this as implicit in the IWP *"P2. Stores are not reordered with other stores"*): *"P10. Writes by a single processor are observed in the same order by all processors"*.

This specification appears to deal with the unsoundness, admitting the n6 behaviour above, but, unfortunately, it is still problematic. The first issue is, again, how to interpret "causality" as used in P5. The second issue is one of weakness: the new P9 says nothing about observations of two stores by those two processors themselves (or by one of those processors and one other). This is illustrated by our n5 and n4 examples [OSS09]. These have final states that were not allowed in x86-CC, and we would be surprised if they were allowed by any reasonable implementation (they are not allowed in a pure write-buffer implementation). We have not observed them on actual processors, and programming above a model that permitted them would be problematic. However, rev-29 appears to allow them.

Following this, we proposed an *x86-TSO* model [OSS09], with equivalent operational and axiomatic definitions formalised in the HOL4 proof assistant [HOL]. Summarising the key litmus-test differences, we have:

	IWP/x86-CC	rev-29	x86-TSO	actual processors
IRIW	allowed	forbidden	forbidden	not observed
n6	forbidden	allowed	allowed	observed
n4/5	forbidden	allowed	forbidden	not observed

Power and ARM Power and ARM have weaker models than x86, but are similar to each other. The following discussion is based on the Power ISA Version 2.05 specification [Pow07] (applicable to POWER6 and POWER5 processors) and the ARM Architecture Reference Manual [ARM08] (applicable to ARMv7 processors).

A key concept in these informal-prose architecture specifications is that of accesses being "performed" (Power) or "observed" (ARM) with respect to processors. "Performed" is defined as follows [Pow07, p.408]:

> *A load or instruction fetch by a processor or mechanism (P1) is performed with respect to any processor or mechanism (P2) when the value to be returned by the load or instruction fetch can no longer be changed by a store by P2. A store by P1 is performed with respect to P2 when a load by P2 from the location accessed by the store will return the value stored (or a value stored subsequently).*

This is used in the informal semantics of barriers (`sync`, `lwsync`, `eieio`, `DMB`, `DSB`), and of dependencies, e.g. [Pow07, Book II,§1.7.1,p.413]:

> *If a* Load *instruction depends on the value returned by a preceding* Load *instruction, the corresponding storage accesses are performed in program order with respect to any processor or mechanism.*

Such a definition of "performed" does not lend itself to a direct formalisation. First, it implicitly refers to a notion of global time. That can be easily solved, as we are only concerned with whether one access is performed before or after another. Second, more seriously, it is subjunctive: the first clause refers to a hypothetical store by P2, and the second to a hypothetical load by P2. A memory model should define whether a particular execution is allowed, and it would be awkward in the extreme to define this in terms of executions modified by adding such hypothetical accesses.

Several initially-plausible interpretations turn out to be too weak or unsound. One could adopt view orders, per-processor orders capturing when events become visible to each processor in its local view of time, and consider an

access to be "performed" from the point where it appears in the view order. Intuitively, the visible events are those that influence the processor's behaviour, and we defined a preliminary such model [AFI+09] in which we do not include other processors' loads in view orders. Doing this naively gives too weak a semantics for barriers, though it can be improved by considering the stores that write to the memory location read by foreign loads. Alternatively, one can include other processor's read events. Again, doing so naively would be wrong: the Load/Load dependency text above suggests that the program order of two loads performed on the same processor is preserved even if they are not interleaved with a barrier instruction, but we have observed non-sequentially consistent behaviour in a variant of the IRIW example above if there is an address dependency between each pair of loads, but not if there is a barrier between each pair of loads. Adir et al. [AAS03] give a more complex model including some foreign loads in view orders, for an earlier PowerPC specification. However, none of these models accounts completely for the Power 2.05 barriers, capturing the rather subtle differences between `sync` and its lighter alternative `lwsync`, and identifying exactly what barriers are required to regain sequential consistency.

If it is this hard to give a consistent interpretation to the architecture documentation, one has to wonder whether correct low-level code could be written based on it, without additional knowledge of the processor implementations.

The Java Memory Model Java has integrated multithreading, and much effort has been devoted to the specification of the precise behaviour of concurrent Java programs. By the year 2000, the initial specification was shown to allow unexpected behaviours, prohibit common compiler optimisations, and was challenging to implement on top of a weakly-consistent multiprocessor [Pug00]. It was superseded around 2004 by the JSR-133 memory model [JSR]. JSR-133 attempts to solve a difficult challenge. It deals with the full complexity of the Java language, including object finalisation and final fields, and aims to satisfy several competing criteria: first, programs which are data-race free, should exhibit only sequentially consistent behaviour; second, arbitrary programs, possibly with data-races, should still satisfy some memory safety and security requirement (for instance "out of thin air" reads are forbidden, that is programs may not observe values not written anywhere); third, a range of common compiler optimisations should be sound.

The resulting model is quite intricate and, unfortunately, poorly understood. For instance it turned out that, despite the efforts, standard optimisations as common subexpression elimination were illegal in the model [CKS07, vA08]. Such optimisations were claimed to be permitted, and are implemented by typical compilers including Sun's reference HotSpot compiler. For example, consider the following transformation

x = y = 0	
r1=x	r2=y
y=r1	x=(r2==1)?y:1

$\longrightarrow$

x = y = 0	
r1=x	x=1
y=r1	r2=y

where `x` and `y` are shared variables and `r1` and `r2` are local variables. For the program on the left, the JMM forbids the outcome `r2` = 1. However, Sun's HotSpot compiler reuses the value of `y` in `r2` and rewrites `x=(r2==1)?y:1` $\longrightarrow$ `x=(r2==1)?r2:1`, which is equivalent to `x=1`. Swapping the independent statements `r2=y` and `x=1` yields the program on the right, which can result in `r2` = 1 even on a sequentially consistent architecture.

The following table (from Ševčík and Aspinall [vA08]) lists some basic transformations used by compiler optimisations, and shows their legality in sequential consistency and in the JSR-133 model:

Transformation	SC	JMM
Trace-preserving transformations	√	√
Reordering normal memory accesses	×	×
Redundant read after read elimination	√	×
Redundant read after write elimination	√	√
Irrelevant read elimination	√	√
Irrelevant read introduction	√	×
Redundant write before write elimination	√	√
Redundant write after read elimination	√	×
Roach-motel reordering	×	×
(for locks)	(√)	
External action reordering	×	×

The C++ Memory Model The C++ language was originally designed without thread support and relied on an external library of threading primitives. It is now clear that an optimising compiler designed independently of threading issues cannot guarantee correctness of the resulting code [Boe05], and an ongoing effort, currently nearing completion, attempts to explicitely provide semantics for threads in the next revision of the C++ standard [BA08, cpp08]. Partially following the Java experience, the revised standard gives semantics only to well-synchronised (data-race free) programs but does not attempt to provide safety guarantees about racy programs.

At a first glance the non-expert programmer's concurrency model [BA08] appears simpler and more comprehensible than the Java one. However, a close look reveals some ambiguities, and there is an extensive framework for low-level synchronisation with intricate semantics. Some technical ambiguities would be easily fixed: the specification refers to sets of memory actions, but in a literal reading of the text there is no way to distinguish memory actions that arise from the same program point and have the same operation and value. The model mentions several kinds of memory accesses, including normal reads and writes, lock/unlock of locks, and atomic accesses, but does not specify whether accesses of different kinds to the same location are permitted. Also, the model does not fully deal with loads from the initial state.

For the specialised "low-level atomics", intended for high-performance code, the semantics appear complex. Memory operations and fences are parameterised by one of six memory ordering constraints, with semantics for interactions between such operations specified informally in the standard [cpp08].

As for other existing models, ambiguities are hidden in the informal prose, but become manifest as soon as the model is formalised mathematically.

Towards Rigorous Memory Models

What, then, is the way forward? Existing real-world memory models cannot be completely trusted, and, although there exists an extensive literature on relaxed memory models, most of it does not address real processor semantics, or is not based on rigorous mathematical models. In this position paper we argue that a specification for a multiprocessor or programming-language memory model should satisfy several stringent criteria.

First, it should be *precise*. It should state unambigously what is and is not permitted. This must be with mathematical precision, not in informal prose — experience shows that

the latter, even when written with great care, is prone to ambiguity and omission. The mathematical definitions should be normative parts of the architecture or language standard. Ideally, these mathematical definitions should be expressed and mechanised in a proof assistant, supporting mechanical type-checking and independently machine-checked proofs of metatheoretic results.

Second, it should be *testable*. Given a program, it should be possible to compute (a) whether some particular candidate execution is admitted by the memory model, and (b) the set of all such admissible executions (up to a certain length). Ideally, the algorithms for these should be derived automatically from the statement of the model, or from some provably-equivalent alternative characterisation thereof, to reduce the possibility of translation error.

Third, it should be *accurate with respect to implementations* (of processors or compilers). Given the above, this can be tested empirically, as it is easy to run programs on real hardware or a real compiler-and-hardware combination. The model should allow all the behaviours observed in practice. This testing, of the behaviour of the real implementations against the published rigorous specification, should be part of the normal development process of the processors or compilers. In principle, accuracy can also be established by proof, showing that the semantics is an accurate abstraction of a microarchitectural model or compiler. That would be highly desirable, but very challenging: for modern multiprocessors and compilers, such a model would be very large, and typically also proprietary.

Fourth, it should be *loose enough for future implementations*: the range of permitted behaviour should be wide enough to allow reasonable developments in implementations. However, this point should not be over-emphasised at the expense of the others, as seems to have often happened in the past.

Fifth, it should be *strong enough for programmers*. A well-specified model should constrain the behaviours enough that reasonable programs can be shown (by informal reasoning, proof, or exhaustive symbolic emulation) to have their intended behaviour, without relying on any other knowledge of the implementations. A different and complementary approach is to formally prove metatheoretic results about the model, including data-race-freedom properties; these proofs are subtle and should be mechanised.

Sixth, it should be *integrated with the semantics of the rest of the system* (describing the behaviour of the processor instructions or of the phrases of the programming language). Memory models are typically presented in isolation, and this makes it all too easy to gloss over important details.

Lastly, it should be *accessible*, to concurrent programmers, hardware architects, language designers and implementors, and builders of verification tools, as the interface between these four groups. For that it should be expressed in straightforward logic, not some exotic specialised formalism, and should be extensively annotated so that it can be read by non-mathematicians. Having a precise mathematical specification will make it easier to write self-consistent tutorial documents. For processors, where possible, it seems desirable to have both an operational (or abstract machine) model and a provably equivalent axiomatic model; the former are often more comprehensible and the latter more useful for some metatheory, and an equivalence proof may detect errors and inconsistencies. However, operational models should not involve more microarchitectural detail than is necessary: it should be clearly understood that these are specifications of the programmer-visible behaviour, not descriptions of any actual microarchitecture.

We are not the first to make some of these points, but we aim, starting with our preliminary investigations on x86 [SSZN+09, OSS09], Power/ARM [AFI+09], and Java [vA08], towards satisfying all the criteria simultaneously. Such work seems to be a necessary precondition for exploiting concurrency efficiently and correctly.

References

[AAS03] A. Adir, H. Attiya, and G. Shurek. Information-flow models for shared memory with an application to the PowerPC architecture. *IEEE Trans. Parallel Distrib. Syst.*, 14(5):502–515, 2003.

[AFI+09] J. Alglave, A. Fox, S. Ishtiaq, M. Myreen, S. Sarkar, P. Sewell, and F. Zappa Nardelli. The semantics of Power and ARM multiprocessor machine code. In *Proc. DAMP 2009*, January 2009.

[AMD07] *AMD64 Architecture Programmer's Manual (3 vols).* Advanced Micro Devices, September 2007. rev. 3.14.

[ARM08] ARM. *ARM Architecture Reference Manual (ARMv7-A and ARMv7-R edition).* April 2008.

[BA08] H.-J. Boehm and S.V. Adve. Foundations of the C++ concurrency memory model. In *Proc. PLDI*, 2008.

[Boe05] H.-J. Boehm. Threads cannot be implemented as a library. In *Proc. PLDI*, 2005.

[CKS07] P. Cenciarelli, A. Knapp, and E. Sibilio. The Java memory model: Operationally, denotationally, axiomatically. In *Proc. ESOP*, 2007.

[cpp08] ISO/IEC 14882, programming languages - C++. WG21 n2800, October 2008.

[Dic08] D. Dice. Java memory model concerns on Intel and AMD systems. `http://blogs.sun.com/dave/entry/java_memory_model_concerns_on`, January 2008.

[HOL] The HOL 4 system. `http://hol.sourceforge.net/`.

[Int07] Intel. Intel 64 architecture memory ordering white paper, 2007. SKU 318147-001.

[Int09] *Intel 64 and IA-32 Architectures Software Developer's Manual (5 vols).* Intel Corporation, March 2009. rev. 30.

[JSR] JSR 133: Java memory model and thread specification revision. `http://jcp.org/en/jsr/detail?id=133`.

[Lin99] Linux kernel traffic, 1999. `http://www.kernel-traffic.org/kernel-traffic/kt19991220_47.txt`.

[Loe08] P. Loewenstein. Personal communication, Nov. 2008.

[OSS09] S. Owens, S. Sarkar, and P. Sewell. A better x86 memory model: x86-TSO (extended version). Technical Report UCAM-CL-TR-745, Univ. of Cambridge, 2009. Supporting material at `www.cl.cam.ac.uk/users/pes20/weakmemory/`.

[Pow07] *Power ISA Version 2.05.* October 2007.

[Pug00] W. Pugh. The Java memory model is fatally flawed. *Concurrency - Practice and Experience*, 12(6):445–455, 2000.

[SSZN+09] S. Sarkar, P. Sewell, F. Zappa Nardelli, S. Owens, T. Ridge, T. Braibant, M. Myreen, and J. Alglave. The semantics of x86-CC multiprocessor machine code. In *Proc. POPL 2009*, January 2009.

[vA08] J. Ševčík and D. Aspinall. On validity of program transformations in the Java memory model. In *ECOOP*, pages 27–51, 2008.