Barrier logic: a program logic for concurrency on PowerPC

Richard Bornat
Computer Laboratory, University of Cambridge, Cambridge, UK
School of Electrical Engineering and Computer Science, Queen Mary, University of London, UK
School of Engineering and Information Sciences, Middlesex University, London, UK
rb501@cam.ac.uk, richard@eecs.qmul.ac.uk, R.Bornat@mdx.ac.uk

April 5, 2012

Abstract

Attempt to distill my rambling ideas.

1 Introduction

Modern processors can be made to go much faster than memory: long gone are the days when an instruction and a memory cycle took about the same amount of time. They go so fast that they can't afford to wait for a memory access to complete before they move on to the next instruction, and in large clusters of processors it takes a significant time for memory effects to flow from one processor to another. The old shared-memory concurrency model, in which processors all had the same view of a central memory, no longer applies. In the PowerPC model of Sarkar et al. (2011) memory, memory caches and storage buffers are all abstracted away: there are individual writes, issued by individual threads,[1] and each read takes its value from some particular write. That means that different threads can have quite different views of the values of shared variables, which makes concurrent programming quite tricky. The PowerPC complicates the issue further by not always executing instructions in the original *program order*, sometimes even speculating execution before it's certain to be needed.

1.1 TIoSE dependency order

Out-of-order execution on Power is not a complete free-for-all. Execution is organised so that from the point of view of a single thread on a single processor, running without interference from outside, instructions seem to execute in program order. To preserve the illusion of sequential execution (henceforth TIoSE) requires some constraints on the order of execution:

d: if one instruction computes a value which is used by another, then they must stay in program order;
c: instructions which have an external effect (e.g. writes to memory) and which follow a conditional branch must stay in program order with the branch;
m: instructions which access the same memory location must stay in program order.

These constraints give designers lots of wriggle room: there's no constraint, for example, that reads or writes to different locations have to happen in program order, and it's the lack of such a constraint which is one of

[1] Throughout this paper 'thread' means hardware thread: effectively a single program running on a single processor

the differences between Power and the TSO mechanism of the Intel x86 (ref). Most strikingly the hardware can *speculate* execution of instructions, provided that speculation is abandoned if a later conditional branch decides against those executions, and that its results are visible only to instructions which follow in program order. But all that is hidden from the single-isolated-thread programmer. Note that TIoSE is the same as Alglave's 'uniproc' ordering (Alglave, 2010).

If the machine's dependency ordering doesn't give the result we require, special *barrier* instructions can be used to enforce extra ordering on instructions which read or write to memory:

sync separates reads and writes before the barrier from those after;
lwsync separates reads and writes before from writes after;
ctrlisync separates one or more reads before from reads after.

Barrier instructions aren't speculated (thank goodness and the sense of the hardware designers).

1.2 Communication through the memory

Memory is a whole different story. In the model of Sarkar et al. (2011), writes to memory take place in stages: first, the instruction is *executed* (possibly even speculated); then it is *committed*, assigning it a place in the execution order; and finally its value is *propagated* to other threads. Committal – not execution – is constrained by control dependency from prior conditional branches. There's a global *coherence order* of committed writes to a particular location, and successive reads from the same location in the same thread can't read out of coherence order. But it's tricky because even though two writes to different locations from a single thread might be committed in a particular order, they might not be propagated in that order to any other thread, and they might be propagated to different threads in different orders. On top of that, coherence order is in principle a partial order. The fact that there isn't an instant when all other threads see a write to memory is the major difference between Power and TSO (ref)

A write which has been executed but not yet committed will appear later in the coherence order than any already committed write, if it's ever committed. So it's safe for a read to take its value from a previous uncommitted write in its own thread (if it is speculated but abandoned then the read will be abandoned too). This makes local reasoning easier, but multiplies the 'different threads have different views' character of Power's relaxed memory.

Actually it's simpler to program than it sounds. The treatment of writes has no discernible effect on local memory, because TIoSE makes writes execute in (eventual) coherence order, and for writes to shared memory the notion of *stability* of an assertion can be called into play. Two of the barriers described above do more than simply introduce local dependencies. Sync and lwsync ensure that a thread's view of shared data, made up of the writes that it has made and the writes that it has read from, is propagated entirely to all other threads. They do this in slightly different ways, discussed below, but overall the effect is that after a sync or lwsync we can convert a locally-stable assertion into one that is universally stable, and the need to preserve such stability can even be a guide to the placing of barriers. There is also an *exclusive write* that makes particular use of the coherence order: see §2.6.

2 Barrier logic

My logic is a Hoare logic for relaxed memory machines. It deals with out-of-order execution and propagation effects with two techniques: dependency graphs and a special treatment of stable assertions.

As in RGSep (Vafeiadis and Parkinson, 2007; Vafeiadis, 2007) assertions make a distinction between local

memory (unboxed assertions) and shared memory (boxed assertions). Assertions describe a thread's view of data; $\mathbb{U}$ (§2.3.1) and $\mathbb{U}_w$ modalities (§2.4.1) and the $\mathbb{C}$ operator (§2.6) allow global remarks.

Invariants are universal assertions. It will not in general be the case that all threads see the same values in shared locations, just because of delays in committing writes and propagating their values, and they won't receive updates at the same time or in the same order. An invariant asserts that all threads see the same *relation* between the values they see in shared locations; that relation must be immune to whatever updates are thrown at a thread in whatever order, including of course its own. Invariants are therefore stated using the $\mathbb{U}$ modality.

Actions are, as in RGSep, the result of indivisible writes to memory.[2] We can't talk of atomic writes, because writing is such a complicated process, but single-word writes are indivisible in the sense that you can't half-write or half-read a value. It makes little sense to pretend that it is possible to write several words at once, though such a pretence might be a useful refinement tool (Dodds and Batty, ref?) and programmers can use locks to provide pseudo-atomicity. There is more to actions though: they typically have a *context*, either implicit or explicit. One might write

$$x = 0 \wedge y = 1 \rightsquigarrow x = 0 \wedge y = 2 \tag{1}$$

$$x = 0 : \quad y = 1 \rightsquigarrow y = 2 \tag{2}$$

to assert equivalently that when $x = 0$, y may change value from 1 to 2. If those are the only actions affecting y, they implicitly assert that it can change value only when $x = 0$. In a conventional shared-memory machine, it is enough if a thread executes $y := 2$ only when $x = 0 \wedge y = 1$ stably in the shared memory. On a relaxed-memory machine that isn't enough: the issuing thread has to be sure that when the update to y is received at some other thread, it is into a view which includes $x = 0 \wedge y = 1$. The need for actions to be *well received* implies some universality, and makes another job for $\mathbb{U}$, $\mathbb{U}_w$ and $\mathbb{C}$.

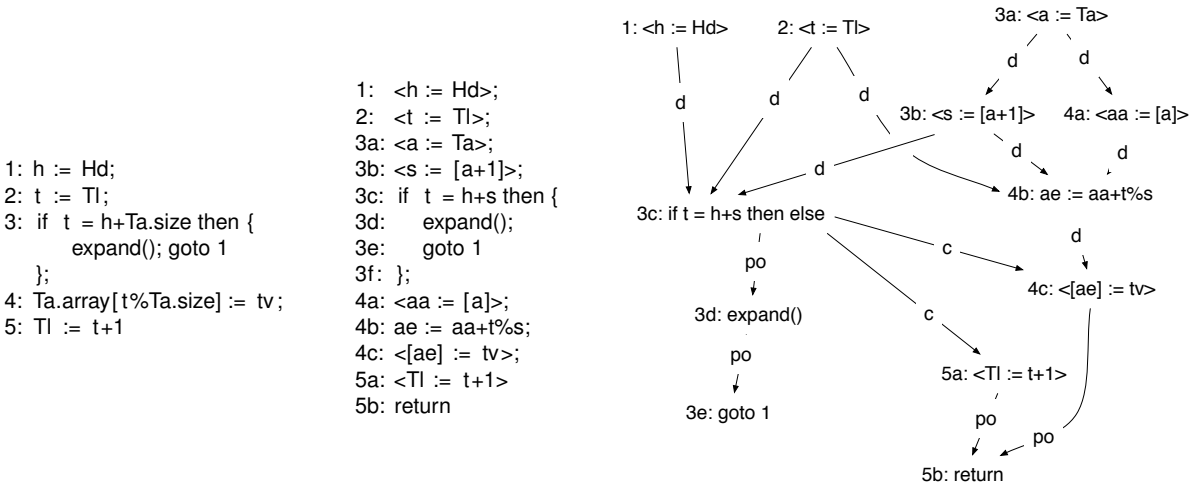
2.1 Dependency graphs

Because instructions are not necessarily executed in program order, it makes sense to reason about commands in dependency order. Because shared-memory accesses are the issue in concurrent algorithms, it is necessary first to render a program so that each command makes at most one access to shared data: to do that we have to do a little pseudo-compilation and invent names for intermediate values which we can think of as register or local variable names. The code in figure 1(a), for example, is pseudo-compiled to figure 1(b), and figure 1(c) shows the dependency graph imposed by TIoSE. Note that in the pseudo compilation I've angle-bracketed each command that makes an indivisible shared-memory read or write. Note also that the graph reveals quite a lot of potential out-of-sequence execution.

Some parts of the graph aren't linked by TIoSE dependencies, and I've preserved the original program order (po) edges. What the machine orders is *instruction executions*, so goto 1 on line 3e doesn't loop back to the top of the diagram: instead what descends from it is a copy of the whole graph, and we have to worry (see §2.2) about the interaction between those two instances. Return on line 5b is also po-linked: what follows in this algorithm will include further executions of this and other procedures. Expand() on line 3d is the dependency graph of another procedure, and we also have to consider dependencies of its commands.

Once we have the dependency graph we can decorate its arcs with state assertions. Each command may take as precondition the postcondition of any other command on which it is directly dependent. That even applies to writes, even though they might be speculated before the point at which a control dependency says they must be executed, because the rules of TIoSE say they can't have any *effect* until the control dependency is

[2] Hardware designers apparently talk of 'single copy atomicity' to describe indivisible writes. Or so I'm told.



(a) original code (b) pseudo-compiled (c) dependency graph

Figure 1: constructing a dependency graph (from (Bornat and Dodds, 2012))

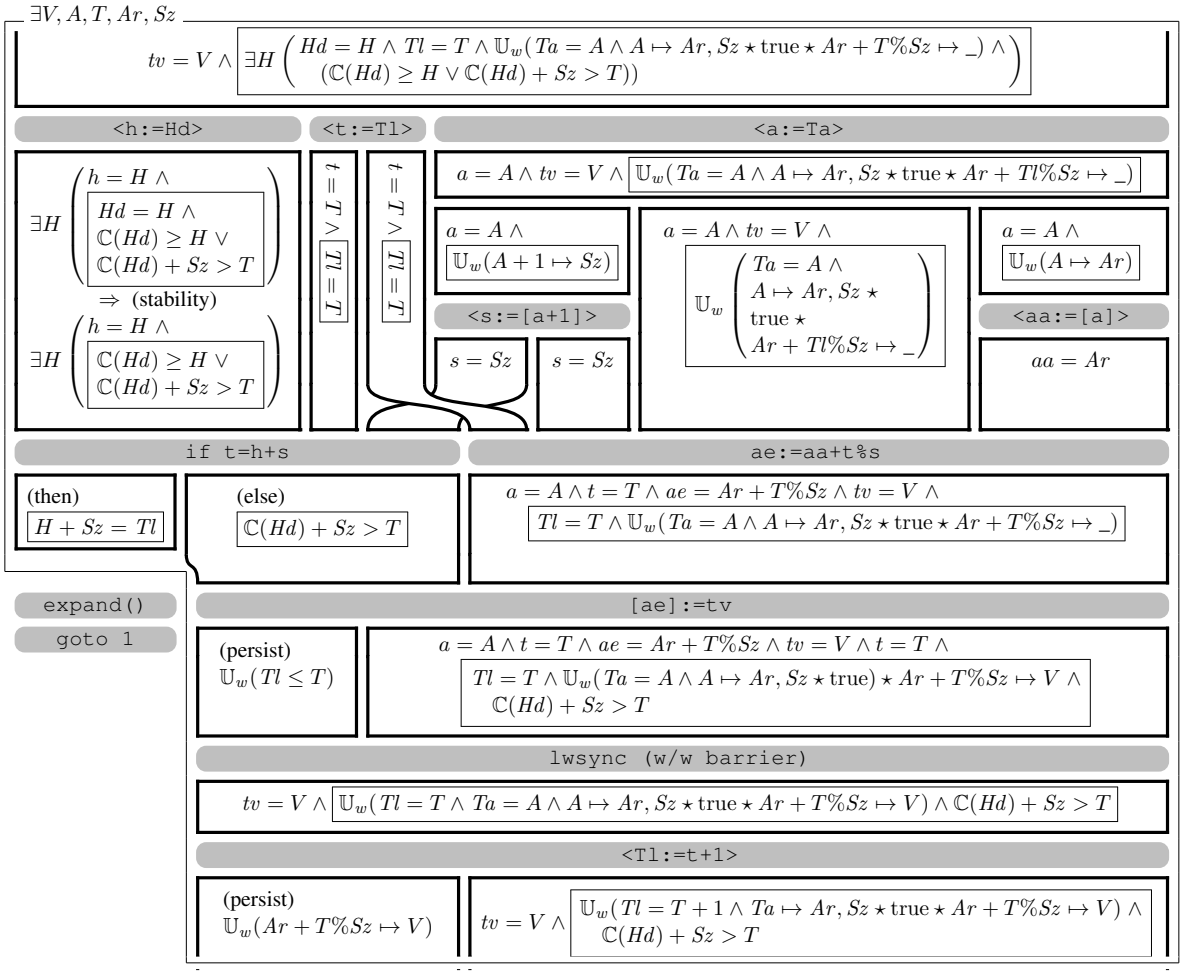


Figure 2: Ribbon proof of figure 1(c) once a barrier is inserted

resolved. In effect we are constructing a ribbon proof (Bean, 2006; Wickerson et al.) of the pseudo-compiled code. Given the invariants of the program, the actions of other threads and the descriptions of this thread's actions, it emerges that it doesn't have enough dependencies and we need to introduce more dependency by putting an lwsync barrier between lines 4c and 5a. The result is a completed ribbon proof like that in figure 2 (the details have to do with the particular invariants and interference of this example: what is important is that it is a decorated dependency graph). Note that there are two dependencies labelled 'persist', explained in §2.3.3.

For clarity and ease of calculation of the effect of a command, I use a version of Vafeiadis's *mid-stability*: the precondition of a command must be stable (proof against interference from other threads and, as we shall see, interference from concurrent writes in the same thread), but its postcondition is instantaneous and must be relaxed so as to be stable if it is to be the precondition of another command.

2.2 Stability

In an RGSep treatment of a concurrent program with a conventional shared memory, a *stable* assertion is about the state of shared memory at a particular program position in a particular thread, a local invariant because it is immune to interference from other threads. With a conventional shared memory, writes before that program point have been taken into account, and writes after it haven't happened yet, so the only actions that could negate a stable assertion would come from other threads.

In relaxed memory it isn't so clear. In figure 1(c), for example, the specification requires that $Tl = t$ holds stably from the instant after Tl is read on line 2 until the write to $Ta.array$ is completed on line 4c. But that can't be so, because on a different, independent, arc of the graph the same thread executes $Tl := t + 1$. $Tl = t$ isn't stable as a precondition of line 4c, which is what the specification requires. It's just that instability which demands an lwsync barrier between lines 4c and 5a.

To be stable under relaxed memory an assertion must be immune to the actions of other threads and also to those actions triggered by writes on concurrent arcs – those from which there is no dependency to or from the arc(s) on which the assertion sits but to or from which there is a program-order relationship. Instructions on concurrent arcs are those which would be executed before or after each other in a sequential execution. In figure 1(c), for example, when analysing stability of $Tl = t$ at line 4c, we don't have to consider what happens inside line 3d – no program-order relationship because what follows goto 1 is another instance of the graph – but we do have to worry about line 5a, which changes the value of Tl and is immediately after line 4c in program order. The specification requires that $Tl = t$ should be stable at line 4c; the structure of the diagram shows that it isn't; and so we need the lwsync.

2.3 Universal stability (the $\mathbb{U}$ modality)

Suppose that P is an assertion about shared memory. To claim that P is invariant is to assert $\mathbb{U}(P)$: P everywhere, always. To write P as a precondition on an arc of a dependency graph we assert that P is stable. What does it mean to assert that $\mathbb{U}(P)$ is stable, and what use is it?

To assert that $\mathbb{U}(P)$ is stable is to assert that P is stable and that P holds, also stably, in all other threads. So to negate P is to negate $\mathbb{U}(P)$. Simple as that.

2.3.1 Sync establishes $\mathbb{U}$

$\mathbb{U}(P)$ is established by the sync barrier from stable P, just because it transmits all the updates made or read from by a thread, which together amounted to P, to all other threads before the next read or write can start:

$$\{P\}\ \mathsf{sync}\ \{\mathbb{U}(P)\} \tag{3}$$
$$\{\mathbb{U}(P)\}\ \mathsf{sync}\ \{\mathbb{U}(P)\} \tag{4}$$
$$\{\mathbb{U}_w(P)\}\ \mathsf{sync}\ \{\mathbb{U}(P)\} \tag{5}$$

$\mathbb{U}(P)$ holds as long as P holds: since P is stable, that means until a write in the barring thread negates P. So there's a temporality involved: for $\mathbb{U}(P)$ to hold, it must have held continuously at least since the last sync. Just as with stability, if there is an assignment which negates P in one arc, $\mathbb{U}(P)$ can't hold in any concurrent arc. Note that since P must be stable at the sync, there can't be any arc concurrent to the sync that can negate P. The $\mathbb{U}_w$ modality, used in axiom (5), is explained in §2.4.1.

2.3.2 Coincidence and since

One use of $\mathbb{U}$ is to enable reads to mark a coincidence. If x and y are shared variables, j is local, and the value of y is subject to interference from other threads, we might reason as follows:

$$\begin{array}{l} \left\{\boxed{\mathbb{U}(x=0)\wedge y=\_}\right\} \\ \quad j := y \\ \left\{\exists J\left(j=J\wedge\boxed{\mathbb{U}(x=0)\wedge y=J}\right)\right\}\ \Rightarrow\ \text{(stability)} \\ \left\{\exists J\left(j=J\wedge\boxed{\left(\mathbb{U}(x=0)\,\mathsf{since}\,y=J\right)\wedge y=\_}\right)\right\} \end{array} \tag{6}$$

The temporal operator since allows us to convert a coincidence into a stable assertion. $A\,\mathsf{since}\,B$ holds if $A\wedge B$ holds, or if A holds and $A\,\mathsf{since}\,B$ held at the previous instant. So if A holds and is stable and we notice, even instantaneously, B, then we can assert stably $A\,\mathsf{since}\,B$ even though B is not stable. This, together with $\mathbb{U}$, seems to be key to understanding some concurrent algorithms, on Power or otherwise.

2.3.3 Writes and persistence

The other use of $\mathbb{U}$ is to ensure that writes are *well received* – that is, that they have the effect declared in the action description. It is simplest to consider actions in the form exemplified by action (2):

$$\alpha : \beta \rightsquigarrow \beta' \tag{7}$$

where α is the *context*, an assertion about values in shared data, β is a description of the value of a single location in shared data when the action happens, and β' is a description of that location after the action has happened. Clearly, an action affects a single location; we may add assertions about auxiliary locations to β and β', of course, without disturbing the nature of an action.

An action description requires that when the action happens in the view of a particular thread – when its view of the affected location changes value – $\alpha \star \beta$ must hold, and afterwards $\alpha \star \beta'$ (or $\alpha \wedge \beta$ before and $\alpha \wedge \beta'$ afterwards if the affected location is a shared variable). Consider, for example, action (2): change y from 1 to 2, but only in the context $x = 0$. We might reason in the acting thread as follows:

$$\{\mathbb{U}(x=0\wedge y=1)\}\ y := 2\ \{\mathbb{U}(x=0)\wedge y=2\} \tag{8}$$

The write will be executed when universally, and stably, $x = 0 \wedge y = 1$; afterwards universally $x = 0$ but locally $y = 2$ (and therefore not universally $y = 1$). We can be sure that no other thread will immediately see $y = 2$, unfortunately. On the other hand we can be sure that they will see the $y := 2$ write before they see any program-order later write to y executed by this thread, because of memory (m) dependency and coherence order. But we can't be sure in general when or if the update will be propagated to any particular other thread: writes don't have to be propagated to threads which don't read y, for example. Worse, if there is a dependency-order-later command $x := 1$,[3] destroying $\mathbb{U}(x = 0)$, we need to be sure that the $y = 2$ update arrives before the $x := 1$ update, and we can't have that assurance unless there is an intervening sync or lwsync.

What this amounts to is that the context α of an action description – $x = 0$ in the example of (8) – must *persist*, must continue to hold in all receiving threads at least until the action-update is received. Since those other threads can't negate α themselves (because we require that α is stable) we can ensure persistence by requiring that α is stable up to the next sync or lwsync. To assert that we need a new dependency, labelled '$(persist)$' which is labelled with the context α of an action description and which runs from the action-initiating assignment to the next executed sync or lwsync; then the stability rules ensure that the acting thread can't negate α before that point, which is the first point when we stop worrying about the order of propagation of assignments.

2.4 Delayed universal stability (the $\mathbb{U}_w$ modality)

Suppose that shared data consists of variables x and y, that initially $x = 0 \wedge y = 0$, and that we want $\mathbb{U}(x \leq y)$ to be invariant (just a stronger kind of stable assertion). The sequential program

$$y := 1; x := 1 \tag{9}$$

would preserve that invariant on a sequentially-consistent machine. But on Power there is no dependency between its two commands, still less any constraint on the propagation of their effects, so $x := 1$ might happen before $y := 1$ in the views of both the acting and the receiving thread. Lwsync is designed to order writes in program order, and in this example would ensure the behaviour we require. It produces a variant of the $\mathbb{U}$ modality which gives enough assurance for reasoning about the propagation order of writes.

2.4.1 Lwsync establishes $\mathbb{U}_w$

Lwsync forces all program-order-preceding writes to be committed before it is committed, which is in turn will be before any program-order succeeding writes are committed, and all the preceding writes plus any writes that have been read from are propagated to any particular other thread before any program-order-succeeding writes are propagated to it. Reads following the lwsync can't be satisfied before it is committed, but unlike in the case of sync, they needn't wait for the propagation of the barrier right round the machine. In effect it imposes the same effect on writes as sync, but doesn't allow reads to notice coincidence.

$$\{P\}\ \mathsf{lwsync}\ \{\mathbb{U}_w(P)\} \tag{10}$$

$$\{\mathbb{U}(P)\}\ \mathsf{lwsync}\ \{\mathbb{U}(P)\} \tag{11}$$

$$\{\mathbb{U}_w(P)\}\ \mathsf{lwsync}\ \{\mathbb{U}_w(P)\} \tag{12}$$

[3] It's tricky but it can happen: it needs a read from y and then an address-dependent write to x.

2.4.2 Lwsync has little effect on following reads

For stability, $\mathbb{U}_w$ behaves like $\mathbb{U}$: $\mathbb{U}_w(P) \Rightarrow P$, so it holds until a command in the barring thread negates P. So far as action-causing writes are concerned, it behaves like $\mathbb{U}$, and it demands the same persistence of the action context. So far as reads are concerned, we are able to assert $\mathbb{U}_w(P)$ since Q if a read starts with $\mathbb{U}_w(P)$ and notices Q, but there doesn't seem to be much force in that.

Lwsync does force all program-order-preceding reads to be committed, which implies satisfied, and likewise any program-order-preceding writes to be committed, before any program-order-succeeding reads are satisfied or writes are executed. So it does impose quite a lot of dependency between commands, which can be useful.

2.5 Ctrlisync

Sync is expensive because a lot of writes have to be propagated round the machine, followed by the sync itself, and the syncing thread has to wait for acknowledgement that all other threads have seen the barrier. Lwsync is cheaper because the lwsyncer doesn't wait for acknowledgement, and so it provides less information about what other threads must have already seen. Ctrlisync is cheaper still because it doesn't do anything about propagation.

Consider the following example program fragment, in which i is local (a register, we might suppose) and x and y are shared:

$$i := x; \text{if } i = i \text{ then skip fi; isync}; y := j \tag{13}$$

There's a data dependency between the first read and the conditional[4] but, if there were no isync, only a control (c) dependency between the branch and the second read, and control dependencies don't have much effect on reads. Putting an isync immediately after the branch means that those reads with a data dependency to the conditional must be satisfied before any read after it can be satisfied, or any write after it executed.

Ctrlisync is very cheap because it is entirely local, and all it does is put some lines into the dependency graph. It's not free because those lines deter speculation.

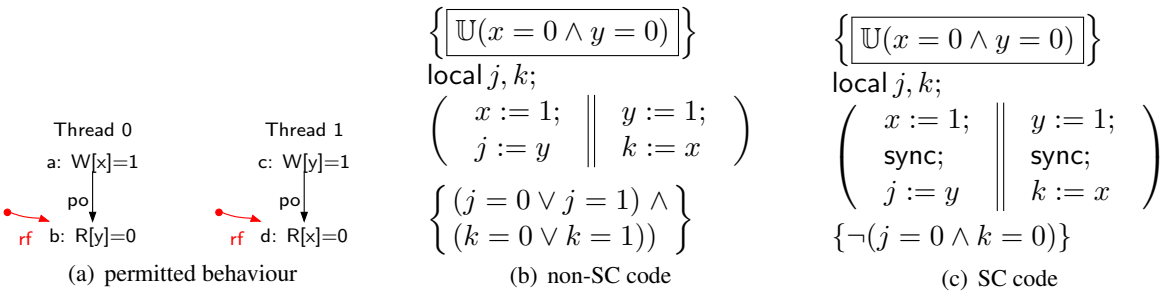
2.6 Exclusive read and write (the $\mathbb{C}$ operator)

Because there are no atomic memory actions, no step in the machine transition system which makes an update visible to all threads at the same time, Power can't have an equivalent of Intel's CAS, which atomically reads a location, compares that value with a test value, and if the comparison succeeds it writes, still in the same atomic action, a new value (some implementations return the value read, hence the name Compare And Swap, or CAS for short).

Instead of CAS Power has 'exclusive read' ($:=_* var$ or $:=_* [loc]$) and 'exclusive write' ($var_* :=$ or $[loc]_* :=$). Exclusive read marks the location it reads; a later exclusive write to the same location either succeeds and writes a value, if that location hasn't been written to since it was marked, or fails and writes nothing. The mechanism is both superior and inferior to CAS:

- superior because it doesn't suffer from the ABA problem, where a location is read, giving A, and a later CAS tests if it has the value A and if so overwrites it with C, but the CAS can't detect that the location has meantime had a different value B and changed back;

[4] I'm assuming a transparent transcription of pseudocode into machine code: that is, no fancy compilers that would eliminate such a vacuous test and unnecessary branch.



(a) permitted behaviour (b) non-SC code (c) SC code

Figure 3: The SB litmus test (Store-Buffering or Dekker's example)

- inferior because exclusive write can fail spuriously, for example if it has been marked again by another thread, but also in many other circumstances, some hard to control or to deal with properly in hardware.

Exclusive write is meant to provide the same sort of assurance that CAS gives, and so it has a complicated implementation. In essence it checks that the write it read from in the exclusive read is currently last in the global coherence order for writes to that location, and if so it tries to put its own write next in the global coherence order as the new last write. Being last in the global coherence order doesn't mean that other threads will have seen its value already or at all: what it does mean is that other exclusive writes have to take their turn.

The $\mathbb{C}$ operator allows us to assert something about values in a coherence order subject to exclusive writes. We might, for example, force by sync an update to a shared variable x in an algorithm in which exclusive writes are used by other threads to increase x. After the sync we have, stably, $\mathbb{C}(x) \geq x$ – the value we see will never be larger than the latest in the global coherence order for x – even though we couldn't say, stably, $\exists X(x = X \wedge \mathbb{U}(x \geq X))$ – because we might, temporarily, see the effect of one thread's exclusive write to x before it reaches all other threads. It's subtle, but that's because it's a subtle mechanism.

3 Example: the SB litmus test

Figure 3 shows a 'litmus test'[5] for the operation of Power. In the absence of any ordering, the machine is allowed to and capable of producing any combination of 0s and 1s in j and k: there are no dependencies in the code of figure 3(b) between writes and reads, so the reads can be satisfied before or after the writes and before or after any writes from the other thread have reached their own thread. Ctrlisync can't order a write-read pair, and although lwsync can, all it does is ensure that the read is satisfied after the write is committed, which doesn't help since it says nothing about when the write is propagated. In principle and in practice this code needs syncs, as in figure 3(c), to enforce sequentially consistent (SC) behaviour so that the result is produced by some interleaving of the commands of the two threads.

The explanation of how the code of figure 3(c) works uses notions of 'before' and 'after': if $j := y$ reads 0 then it must have happened before the write $y := 1$ was received, which must be before the sync of the other thread completed and therefore before $i := x$ started, but after its own sync had completed and therefore after $x := 1$ was propagated to that other thread; so it's impossible, if $j := y$ reads 0, for $i := x$ to read 0, and by a similar argument, vice-versa.

Of course the commands and the barriers don't really interleave, and we don't want to have to think about

[5] Who knows who invented litmus tests? Please tell me.

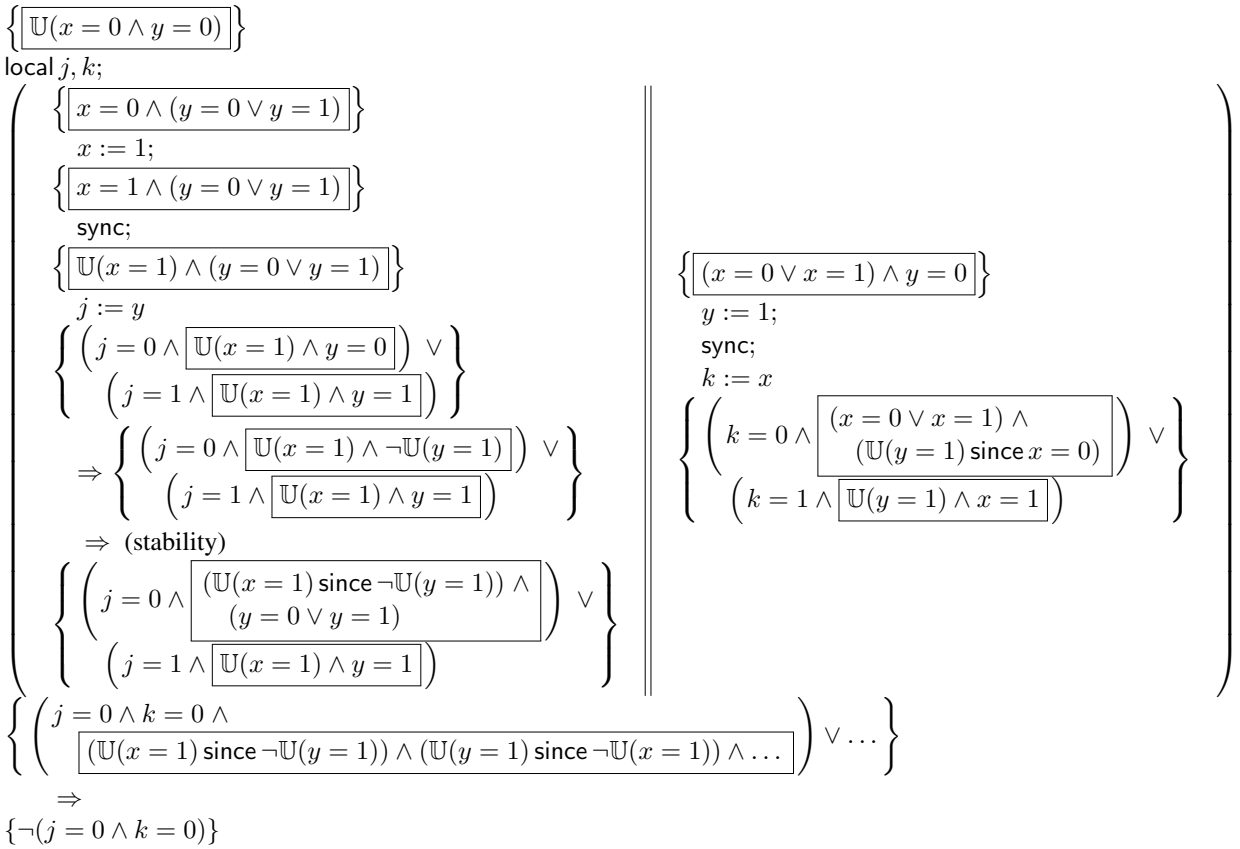


Figure 4: Why SB works with syncs

other thread's orderings when dealing with our own. The proof in figure 4 deals with the question by showing a clear temporal contradiction: it's impossible that the timeline over which $\mathbb{U}(x = 1)$ holds can be extended farther than the timeline over which $\mathbb{U}(y = 1)$ and at the same time the opposite.

4 No more examples?

The logic suggests that of four barriers required for Maged Michael's concurrent stack with hazard pointers (Michael, 2004, 2011) only two are required. It appears to show that in two of the IWSQ algorithms from (Michael et al., 2009) more barriers are required than are described (Bornat and Dodds, 2012; Bornat, 2012).

Each of these examples is too intricate and too large to include here. There isn't yet a soundness proof linking the logic to the model of (Sarkar et al., 2011), and even if there were we would need some more experimental evidence, from more litmus tests, to gain confidence that the model itself captures the machine's behaviour accurately in the areas that the logic patrols.

References

Jade Alglave. *A Shared Memory Poetics*. PhD thesis, Paris 7 – Denis Diderot, November 2010. 2

Jules Bean. *Ribbon Proofs — A Proof System for the Logic of Bunched Implications*. PhD thesis, Queen Mary, University of London, 2006. 5

Richard Bornat. Memory barriers for LIFO IWSQ on Power. 2012. 11

Richard Bornat and Mike Dodds. Abducing memory barriers. submitted to PODC 2012, 2012. 4, 11

Maged M. Michael. Hazard Pointers: Safe Memory Reclamation for Lock-Free Objects. *IEEE Trans. Parallel Distrib. Syst.*, 15(6):491–504, 2004. 11

Maged M. Michael. Four barriers for hazard pointers in concurrent stack. Private communication, 2011. 11

Maged M. Michael, Martin T. Vechev, and Vijay A. Saraswat. Idempotent work stealing. In *Proceedings of the 14th ACM SIGPLAN symposium on Principles and practice of parallel programming*, pages 45–54, 2009. 11

Susmit Sarkar, Peter Sewell, Jade Alglave, Luc Maranget, and Derek Williams. Understanding power multiprocessors. In *Proceedings of the 32nd ACM SIGPLAN conference on Programming language design and implementation*, pages 175–186, 2011. 1, 2, 11

Viktor Vafeiadis. *Modular fine-grained concurrency verification*. PhD thesis, University of Cambridge, 2007. 2

Viktor Vafeiadis and Matthew J. Parkinson. A Marriage of Rely/Guarantee and Separation Logic. In *CONCUR 2007 – Concurrency Theory*, volume 4037 of *LNCS*, pages 256–271, August 2007. 2

John Wickerson, Mike Dodds, and Matthew J. Parkinson. Ribbon proofs for separation logic. to appear, 2012 . 5