

# A Tutorial Introduction to the ARM and POWER Relaxed Memory Models

Luc Maranget  
INRIA

Susmit Sarkar  
University of Cambridge

Peter Sewell  
University of Cambridge

October 10, 2012      Revision: 120

## Abstract

ARM and IBM POWER multiprocessors have highly *relaxed* memory models: they make use of a range of hardware optimisations that do not affect the observable behaviour of sequential code but which are exposed to concurrent programmers, and concurrent code may not execute in the way one intends unless sufficient synchronisation, in the form of barriers, dependencies, and load-reserve/store-conditional pairs, is present. In this tutorial we explain some of the main issues that programmers should be aware of, by example. The material is based on extensive experimental testing, discussion with some of the designers, and formal models that aim to capture the architectural intent (though we do not speak for the vendors). To keep this tutorial as accessible as possible, we refer to our previous work for those details.

## Contents

<b>1</b>	<b>Introduction</b>	<b>3</b>
1.1	Scope . . . . .	3
1.2	Organisation . . . . .	3
<b>2</b>	<b>From Sequential Consistency to Relaxed Memory Models</b>	<b>4</b>
<b>3</b>	<b>Introducing Litmus Tests, and Simple Message Passing (MP)</b>	<b>8</b>
3.1	Message Passing Attempts without Barriers or Dependencies . . . . .	8
3.2	Enforcing Order with Strong (dmb/sync) Barriers . . . . .	11
3.3	Enforcing Order with the POWER lwsync Barrier . . . . .	12
3.4	Observed Behaviour . . . . .	13
<b>4</b>	<b>Enforcing Order with Dependencies</b>	<b>13</b>
4.1	Address Dependencies . . . . .	13
4.2	Control Dependencies . . . . .	14
4.3	Control-isb/isync Dependencies . . . . .	15
4.4	Control Dependencies from a Read to a Write . . . . .	15
4.5	Data Dependencies from a Read to a Write . . . . .	16
4.6	Summary of Dependencies . . . . .	17
4.7	Observed Behaviour . . . . .	17
<b>5</b>	<b>Iterated Message Passing on more than two threads and Cumulativity (WRC and ISA2)</b>	<b>18</b>
5.1	Cumulative Barriers for WRC . . . . .	19
5.2	Cumulative Barriers for ISA2 . . . . .	19
5.3	Observed Behaviour . . . . .	20
<b>6</b>	<b>Store-buffering (SB) or Dekker's Examples</b>	<b>20</b>
6.1	Extending SB to more threads: IRIW and RWC . . . . .	21
6.2	SB Variations with Writes: R and 2+2W . . . . .	22
6.3	Observed Behaviour . . . . .	23

<b>7</b>	<b>Load-Buffering (LB) Examples</b>	<b>23</b>
7.1	Observed Behaviour . . . . .	24
<b>8</b>	<b>Coherence (CoRR1, CoWW, CoRW1, CoWR, CoRW)</b>	<b>25</b>
<b>9</b>	<b>Periodic Tables of Litmus Tests</b>	<b>26</b>
9.1	Litmus Test Families . . . . .	26
9.2	Minimal Strengthenings for a Family . . . . .	26
9.3	4-edge 2-thread Tests and RF-Extensions . . . . .	27
9.4	6-edge 3-thread Tests . . . . .	30
9.5	Test Family Coverage . . . . .	31
9.6	Coherence . . . . .	32
<b>10</b>	<b>Preserved Program Order (PPO) Variations</b>	<b>32</b>
10.1	No Write-to-write Dependency from rf;addr (MP+nondep+dmb/sync) . . . . .	32
10.2	No Ordering from Register Shadowing (MP+dmb/sync+rs, LB+rs) . . . . .	33
10.3	Preserved Program Order, Speculation, and Write Forwarding (PPOCA and PPOAA) . . . . .	34
10.4	Aggressively Out-of-order Reads (RSW and RDW) . . . . .	34
10.5	Might-access-same-address . . . . .	35
10.6	Observable Read-request Buffering . . . . .	36
<b>11</b>	<b>Coherence and lwsync (Z6.3+lwsync+lwsync+addr)</b>	<b>37</b>
<b>12</b>	<b>Unobservable Interconnect Topology (IRIW+addrs-twice)</b>	<b>37</b>
<b>13</b>	<b>Load-reserve/Store-conditional</b>	<b>38</b>
13.1	Load-reserves and Store-conditionals Stay in Order . . . . .	39
13.2	Load-reserves and Store-conditionals Not Ordered with Normal Accesses . . . . .	40
<b>14</b>	<b>Analysis of Peterson's algorithm on POWER</b>	<b>41</b>
<b>15</b>	<b>Sources of Tests, and Correspondences among Them</b>	<b>44</b>
15.1	Boehm and Adve examples . . . . .	44
15.2	ARM Cookbook examples . . . . .	45
15.3	Power2.06 ISA examples . . . . .	45
15.4	Adir et al. examples . . . . .	45
15.5	Adve and Gharachorloo examples . . . . .	46
<b>16</b>	<b>Related Work</b>	<b>46</b>
<b>17</b>	<b>On-line material</b>	<b>47</b>
	<b>References</b>	<b>48</b>

# 1 Introduction

ARM and IBM POWER multiprocessors have highly *relaxed* memory models: they make use of a range of hardware optimisations that do not affect the observable behaviour of sequential code but which are exposed to concurrent programmers, and concurrent code may not execute in the way one intends unless sufficient synchronisation, in the form of barriers, dependencies, and load-reserve/store-conditional pairs, is present. In this tutorial we explain some of the main issues that programmers should be aware of, by example. The material is based on extensive experimental testing, discussion with some of the designers, and formal models that aim to capture the architectural intent (though we do not speak for the vendors). To keep this tutorial as accessible as possible, we refer to our previous work for those details.

We emphasise that our focus is on low-level concurrent code: implementations of synchronisation libraries, concurrent datastructures, etc. For simple higher-level code that follows a correct locking discipline and that is race-free apart from the lock implementations, most of what we describe should not be a concern. Even for low-level code, while some of our examples crop up in practical programming idioms, some (to be best of our knowledge) do not, and are included simply to provide a reasonably complete picture of the possible behaviour of the machines.

## 1.1 Scope

The ARM and IBM POWER architectures differ in many respects, but they have similar (though not identical) relaxed memory models. Here, we aim to cover the memory models for the fragments of the instruction sets required for typical low-level concurrent algorithms in main memory, as they might appear in user or OS kernel code. We include memory reads and writes, register-to-register operations, branches, and the various kinds of dependency between instructions. The two architectures each have a variety of special instructions that enforce particular ordering properties. First, there are *memory barriers*. For POWER we cover the `sync` (also known as `hwsync` or `sync 0`), `lwsync`, and `eieio` barriers, while for ARM we cover the DMB barrier, which is analogous to the POWER `sync`. Second, there are the POWER `isync` instruction and analogous ARM `ISB` instruction. Third, there are POWER load-reserve/store-conditional pairs `laxx/stcx` and the ARM load-exclusive/store-exclusive pairs `LDREX/STREX`. We do not deal with mixed-size accesses or with explicit manipulation of page tables, cache hints, self-modifying code, or interrupts. For ARM, we assume that all observers are in the “same required shareability domain”.

## 1.2 Organisation

We structure the explanation around a series of examples (often known as *litmus tests*): very small concurrent programs, accessing just a few shared variables, that illustrate the main relaxed-memory phenomena that one should be aware of. Most are taken from a systematic study of the interesting small examples, covering all possible patterns of communication and synchronisation up to a certain size, and we pull these together into a ‘periodic table’ of examples in Section 9. To let one see the communication and synchronisation patterns as clearly as possible, the examples are abstract rather than taken from production code, but we talk briefly about the possible use cases in which each might arise (and about possible microarchitectural explanations of their behaviour). After reading this tutorial, one should be able to look at some production concurrent code and analyse it in terms of the communication and synchronisation patterns it uses. Sections 10–12 illustrate a variety of more subtle phenomena, then Section 13 discusses load-exclusive/store-exclusive (ARM) and load-reserve/store-conditional (POWER) instructions. Section 14 analyses an example algorithm, a simplified version of Peterson’s algorithm for mutual exclusion, in terms of litmus test patterns. Section 15 relates the litmus tests we use to some that have appeared in the literature and Section 16 reviews some of the related work.

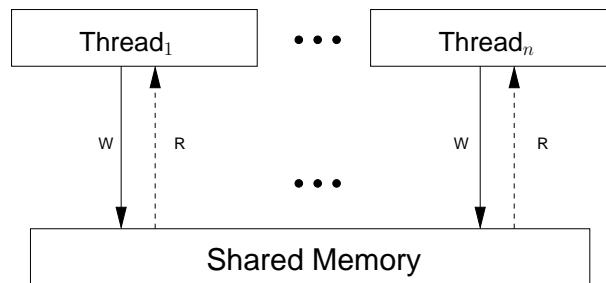
Finally Section 17 describes supporting material that is available on-line. Various papers describe an operational abstract-machine model for POWER, explain the behaviour of some litmus tests in terms of that model, give a correctness proof for an implementation of the C/C++ concurrency model of the C11 and C++11 revised standards [BA08, BOS<sup>+</sup>11, Bec11, ISO11] above POWER processors, and give an axiomatic model for POWER. Our `ppcmmem` tool, available via a web interface, lets one interactively explore the behaviour of a POWER or ARM litmus test with respect to our operational model; our `litmus` tool takes a litmus test and constructs a test harness (as a C program with embedded assembly) to experimentally test its observable behaviours; and our `diy` tool generating litmus tests from concise specifications. There is also an on-line summary of tests and experimental results.

## 2 From Sequential Consistency to Relaxed Memory Models

One might expect multiprocessors to have *sequentially consistent* (SC) shared memory, in which, as articulated by Lamport [Lam79]:

*“the result of any execution is the same as if the operations of all the processors were executed in some sequential order, and the operations of each individual processor appear in this sequence in the order specified by its program”.*

An SC machine can be modelled as in the diagram below:



Here there are a number of hardware threads, each executing code as specified by the program, which access a single shared memory (by writing and reading the values it holds at each address). Such a machine has two key properties:

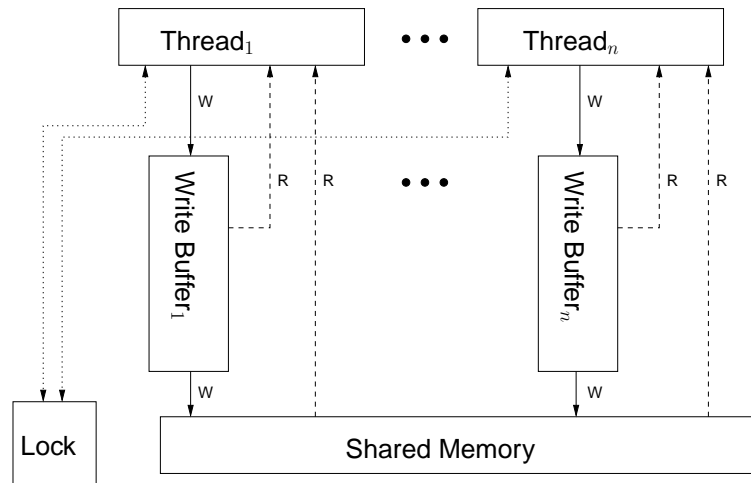
1. There is no *local reordering*: each hardware thread executes instructions in the order specified by the program, completing each instruction (including any reads or writes to the shared memory) before starting the next.
2. Each write becomes visible to all threads (including the thread doing the write) at the same time.

However, most multiprocessors are not sequentially consistent, including the current ARM, POWER, x86, Itanium, and SPARC architectures, and others dating back at least as far as the 1972 IBM System 370/158MP. Instead they have various *relaxed* or *weak* memory models: they guarantee only weaker properties, to allow a range of microarchitectural optimisations in the processor implementations that provide better performance, better energy usage, or simpler hardware. These optimisations are typically not observable to single-threaded code, or by programs that obey a conventional locking discipline and are (except within the lock implementations) race-free, but general concurrent code can observe non-SC behaviours.

The details vary between architectures, and even between different processor implementations of the same architecture. In broad terms x86 and SPARC are similar, with relatively strong models based on the *Total Store Ordering* (TSO) model [Spa92, OSS09, SSO<sup>+</sup>10] that we recall below. ARM and POWER are much weaker than TSO (though broadly similar to each other), as we shall describe in this tutorial. Itanium [Int02] is also much weaker than TSO, but in rather different ways to ARM and POWER; we do not cover it here.

**TSO** An x86-TSO or SPARC TSO machine can be described by the diagram below [SSO<sup>+</sup>10, OSS09, SSZN<sup>+</sup>09, Spa92]. Here each hardware thread has a FIFO write buffer of pending memory writes (thus avoiding the need to block a thread while a write completes). Moreover, a read in TSO is required to read from the most recent write to the

same address, if there is one, in the local store buffer.



In addition, many x86 instructions involve multiple memory accesses, e.g. an x86 increment `INC`. By default, these are not guaranteed atomic (so two parallel increments of an initially 0 location might result in it holding 1), but there are atomic variants of them: `LOCK;INC` atomically performs a read, a write of the incremented value, *and* a flush of the local write buffer, effectively locking the memory for the duration. Compare-and-swap instructions (`CMPXCHG`) are atomic in the same way, and memory fences (`MFENCE`) simply flush the local write buffer. SPARC is similar, though with a smaller repertoire of atomic instructions rather than a general `LOCK` prefix.

Returning to the two properties above, in TSO a thread can see its own writes before they become visible to other threads (by reading them from its write buffer), but any write becomes visible to all *other* threads simultaneously: TSO is a *multiple-copy atomic* model, in the terminology of Collier [Col92]. One can also see the possibility of reading from the local write buffer as allowing a specific kind of local reordering. A program that writes one location  $x$  then reads another location  $y$  might execute by adding the write to  $x$  to the thread's buffer, then reading  $y$  from memory, before finally making the write to  $x$  visible to other threads by flushing it from the buffer. In this case the thread reads the value of  $y$  that was in the memory *before* the new write of  $x$  hits memory.

**ARM and POWER** ARM and IBM POWER have adopted considerably more relaxed memory models. For several reasons (including performance, power efficiency, hardware complexity, and historical choices), **they allow a wider range of hardware optimisations to be observable to the programmer.** This allows a wide range of relaxed behaviour by default, so the architectures also provide mechanisms, in the form of various memory barriers and dependency guarantees, for the programmer to enforce stronger ordering (and to pay the cost thereof) only where it is required. In the absence of such:

1. The hardware threads can each perform reads and writes out-of-order, or even speculatively (before preceding conditional branches have been resolved). In contrast to TSO, where there is no local reordering except of reads after writes to different addresses, here any local reordering is allowed unless specified otherwise.
2. The memory system (perhaps involving a hierarchy of buffers and a complex interconnect) does not guarantee that a write becomes visible to all other hardware threads at the same time point; these architectures are not multiple-copy atomic.

**Implementation and Architecture** To explain the ARM and POWER allowed behaviour in more detail, we have to clearly distinguish between several views of a multiprocessor. A specific processor implementation, such as the IBM POWER 7, or the NVIDIA Tegra 2 (an SoC containing a dual-core ARM Cortex-A9 CPU), will have a specific *microarchitecture* and detailed design that incorporates many optimisations. We are not concerned here with the internal structure of the designs, as they are too complex to provide a good programming model and are typically commercially confidential.

Instead, we are concerned with the *programmer-observable* behaviour of implementations: the set of all behaviour that a programmer might see by executing multithreaded programs on some particular implementation and examining the results (moreover, here we are concerned only with correctness properties, not with performance behaviour).

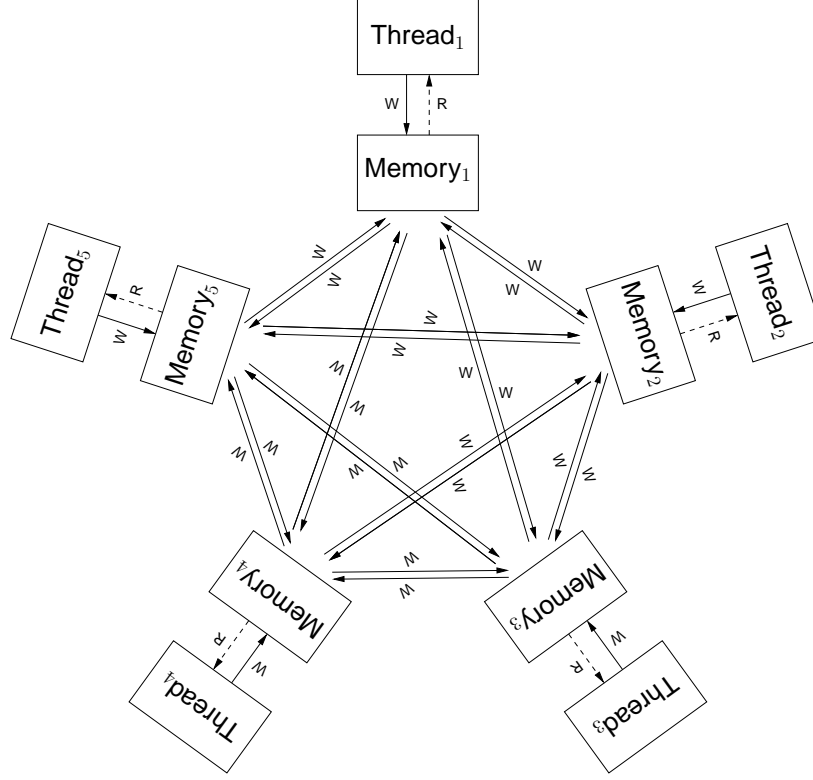
Even this is usually too specific to work with: each processor family comprises many implementations which (as we shall see) can have significantly different programmer-observable behaviour, and one typically wants software that will work correctly on all of them, not just on one specific processor implementation. An *architecture* specifies a range of behaviour that programmers should be able to depend on, for any processor implementation that conforms to the architecture. An architecture may be a significantly looser specification than the programmer-observable behaviour of a specific processor implementation, to accommodate the variations among current, past, and future implementations. (Of course, this looseness makes software development challenging: in principle one might think that one should “program to the architecture”, but normal software development relies on testing software running on specific processor implementations, which may not exhibit some architecturally allowed behaviour that other implementations do or will exhibit.)

Processor vendors produce architecture specifications, such as the Power ISA Version 2.06 [Pow09], and the ARM Architecture Reference Manual (ARMv7-A and ARMv7-R edition) [ARM08a]. These use prose and pseudocode to describe a range of observable behaviour, and are generally rather precise about sequential behaviour but less clear that one might hope when it comes to concurrent behaviour and relaxed-memory phenomena — it is very hard to produce prose that unambiguously and completely captures these subtleties.

Instead, we and others advocate the use of *mathematically rigorous* architecture definitions. These are often most accessibly presented as *operational models*, such as the TSO machine illustrated above. This has an ‘abstract microarchitectural’ flavour: it is an *abstract machine*, with some machine state (comprising the states of the shared memory, the FIFO write buffers, and the threads) and a definition of the transitions that the machine can take. It specifies a range of observable behaviour for a concurrent program implicitly, as all the behaviour that that abstract machine could exhibit when running that program; by claiming that an abstract machine is a sound architectural model for a range of processor implementations, we mean that for any program the set of observable behaviours of the abstract machine running that program includes any behaviour that might be exhibited by any of those implementations running the program. The internal structure of the abstract machine, on the other hand, might be very different to the microarchitecture and detailed design of the processor implementations. Indeed, modern x86 processors typically will have a complex cache hierarchy, out-of-order execution, and so on, quite different from the simple FIFO-write-buffer structure of the abstract machine. The significance of the abstract machine model is that those are not observable to the programmer (except via performance properties).

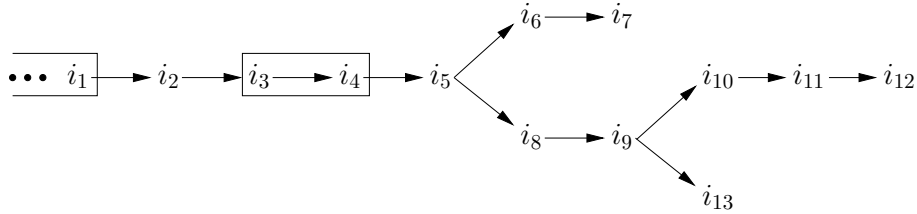
In ongoing work, we are developing mathematically rigorous operational-model architecture specifications for ARM and POWER [SSA<sup>+</sup>11, SMO<sup>+</sup>12]. In this tutorial we focus on explaining what observable behaviour is permitted for ARM and POWER by example, so we will not give those models in detail, but to develop some intuition for how concurrent programs might behave it is useful to first introduce some of the basic concepts used by the model. We emphasise again that these are not to be confused with the actual microarchitecture of current implementations.

**ARM and POWER Abstract Machine Concepts** To explain the behaviour of a non-multiple-copy-atomic machine, it is sometimes helpful to think of each thread as effectively having its own copy of memory, specifying abstractly what value that thread would normally read for any address, as in the diagram below. A write by one thread may *propagate* to other threads in any order, and the propagations of writes to different addresses can be interleaved arbitrarily, unless they are constrained by barriers or coherence. As we shall see later, one can also think of barriers (the ARM DMB and POWER sync and lwsync) as propagating from the hardware thread that executed them to each of the other threads.



We speak of the collection of all the memories and their interconnect (i.e., everything except the threads) as the *storage subsystem*.

For the thread-local out-of-order (and speculative) execution, in general we can think of each thread, at any point in time, as having a tree of the *committed* and *in-flight* instruction instances. Newly fetched instructions become in-flight, and later, subject to appropriate preconditions, can be committed. For example, below we show a set of instruction instances  $\{i_1, \dots, i_{13}\}$  with the program-order-successor relation among them. Three of those ( $\{i_1, i_3, i_4\}$ , boxed) have been committed; the remainder are in-flight.



Instruction instances  $i_5$  and  $i_9$  are branches for which the thread has fetched multiple possible successors; here just two, but a branch with a computed address might in principle fetch many possible successors. A typical implementation might well explore at most one speculative path at a time. Note that the committed instances are not necessarily contiguous: here  $i_3$  and  $i_4$  have been committed even though  $i_2$  has not, which can only happen if they are sufficiently independent. When a branch is committed then any un-taken alternative paths are discarded, and instructions that follow (in program order) an uncommitted branch cannot be committed until that branch is, so the tree must be linear before any committed (boxed) instructions.

For a read instruction, as soon as an address for the read is known, the read might be *satisfied*, binding its value to one received from the local memory (or in some cases forwarded from earlier in the thread). That value could immediately be used by later instructions in the thread that depend on it, but it and they are subject to being restarted or (if this is a speculative path) aborted until the read is *committed*.

For a write instruction, the key points are when the address and value become determined. After that (subject to other conditions) the write can be *committed*, sent to the local memory; this is not subject to restart or abort. After that, the write might *propagate* to other threads, becoming readable by them.



Barriers are similar in that they get *committed* at a thread and sent to the local part of the storage subsystem, before perhaps *propagating* to other threads. The constraints on how writes and barriers can propagate are intertwined, as we shall see.

**Aside: other notions of atomicity** We introduced *multiple-copy atomicity* above, but some caution is needed, as there are many different senses of “atomic” in use. Two other important notions of atomicity are as follows.

A memory read or write by an instruction is *access-atomic* (or *single-copy atomic*, in the terminology of Collier [Col92])—though note that single-copy atomic is not the opposite of multiple-copy atomic) if it gives rise to a single access to the memory. Typically an architecture will specify that certain sizes of reads and writes, subject to some alignment constraints, (such as 1, 2, 4, 8, and 16-byte accesses with those alignments), are access-atomic, while other sizes and non-aligned accesses may be split into several distinct subaccesses. For example, two writes of the same size to the same address are access-atomic iff the result is guaranteed to be either one or the other value, not a combination of their bits. Of course, in a machine which is not multiple-copy atomic, even if a write instruction is access-atomic, the write may become visible to different threads at different times (and if a write is not access-atomic, the individual subaccesses may become visible to different threads at different times, perhaps in different orders).

An instruction that involves more than one memory access, such as an increment that does a read and a write to the same location, or a load-multiple that reads several words, is *instruction-atomic* if its accesses are indivisible in time, with no other intervening access by other threads to their locations. For example, increment is instruction-atomic iff two concurrent increments to the same location that is initially 0 are guaranteed to result in the location containing 2, not 1. On x86 INC is not instruction-atomic whereas LOCK;INC is. On POWER an lmw load-multiple instruction is not instruction-atomic.

Yet another usage is the C11 and C++11 atomic types and operations. These have various properties, including analogues of access- and instruction-atomicity, that we will not discuss here; see [BA08, BOS<sup>+</sup>11, Bec11, ISO11] for details.

## 3 Introducing Litmus Tests, and Simple Message Passing (MP)

### 3.1 Message Passing Attempts without Barriers or Dependencies

**3.1.1 The Message Passing (MP) Example** A simple example illustrating some ways in which ARM and POWER are relaxed is the classic *message passing* (MP) example below, with two threads (Thread 0 and Thread 1) and two shared variables (x and y). This is a simple low-level concurrency programming idiom, in which one thread (Thread 0) writes some data x, and then sets a flag y to indicate that the data is ready to be read, while another thread (Thread 1) busy-waits reading the flag y until it sees it set, and then reads the data x into a local variable or processor register r2. The desired behaviour is that after the reading thread has seen the flag set, its subsequent read of the data x will see the value from the writing thread, not the initial state (or some other previous value). In pseudocode:

MP-loop		Pseudocode
Thread 0	Thread 1	
x=1 // write data	while (y==0) { }	// busy-wait for flag
y=1 // write flag	r2=x	// read data
Initial state: x=0 $\wedge$ y=0		
Forbidden?: Thread 1 register r2 = 0		

The test specifies the initial state of registers and memory (x=0 and y=0; henceforth we assume these are zero if not given explicitly) and a constraint on the final state, e.g. that Thread 1’s register r2 is 0. Here x (or [x] in assembly tests) is the value of memory location x; later we write 1:r2 for the value of register r2 on hardware thread 1. If one reached that final state, with r2=0, then the Thread 1 read of x would have to have read x=0 from the initial state despite the Thread 1 while loop having successfully exit on reading from the Thread 0 write of y=1, program-order-after its write of x=1.

We can simplify the example without really affecting what is going on by looking at just a single test of the flag: instead of looking at all executions of the MP-loop busy-waiting loop, we can restrict our attention to just the executions of the MP program below in which the Thread 1 read of y sees the value 1 written by Thread 0 (we are effectively considering just the executions of MP-loop in which the while loop test succeeds the first time). In other



words, the desired behaviour is that *if* the read of  $y$  saw 1 then the read of  $x$  must not have seen 0. Or, equivalently, the desired behaviour is that final outcomes in which  $r1=1$  and  $r2=0$  should be forbidden.

MP		Pseudocode	
Thread 0		Thread 1	
x=1		r1=y	
y=1		r2=x	
Initial state: $x=0 \wedge y=0$			
Forbidden?: $1:r1=1 \wedge 1:r2=0$			

A *litmus test* such as this comprises a small multithreaded program, with a defined initial state and with a constraint on their final state that picks out the potential executions of interest. Given that, for any architecture we can ask whether such an execution is *allowed* or *forbidden*; we can also run the test (in a test harness [AMSS11a]) on particular processor implementations to see whether it is *observed* or *not observed*.

Throughout this document we use the term “thread” to refer to hardware threads on SMT machines and processors on non-SMT machines. Assuming a correctly implemented scheduler (with appropriate barriers at context switches) it should be sound to think of software threads in the same way.

**3.1.2 Observed Behaviour** In a sequentially consistent model, that final outcome of  $r1=1 \wedge r2=0$  is indeed forbidden, as there is no interleaving of the reads and writes (in which each read reads the value of the most recent write to the same address) which permits it. To check this, one can just enumerate the six possible interleavings that respect the program order of each thread:

Interleaving	Final register state
$x=1; y=1; r1=y; r2=x$	$r1=1 \wedge r2=1$
$x=1; r1=y; y=1; r2=x$	$r1=0 \wedge r2=1$
$x=1; r1=y; r2=x; y=1$	$r1=0 \wedge r2=1$
$r1=y; r2=x; x=1; y=1$	$r1=0 \wedge r2=0$
$r1=y; x=1; r2=x; y=1$	$r1=0 \wedge r2=1$
$r1=y; x=1; y=1; r2=x$	$r1=0 \wedge r2=1$

On x86-TSO or SPARC TSO that final outcome of  $r1=1 \wedge r2=0$  is also forbidden, as the two writes flow through a FIFO buffer into the shared memory before becoming visible to the reading thread. But on ARM and POWER, this final outcome is *allowed* in the architecture, and it is commonly *observable* on current processor implementations. Thread 1 can see the flag  $y$  set to 1, and program-order-subsequently see the data  $x$  still 0. The table below gives some sample experimental data, running this test on various processor implementations using a test harness produced by our litmus tool [AMSS11a]. Each entry gives a ratio  $m/n$ , where  $m$  is the number of times that the final outcome of  $r1=1 \wedge r2=0$  was observed in  $n$  trials.

		POWER			ARM			
	Kind	PowerG5	Power6	Power7	Tegra2	Tegra3	APQ8060	A5X
MP	Allow	10M/4.9G	6.5M/29G	1.7G/167G	40M/3.8G	138k/16M	61k/552M	437k/185M

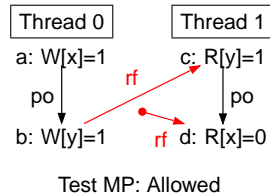
Here we just show the frequency of the outcome identified by the final state constraint, but many other outcomes (all the sequentially consistent outcomes listed above), are also allowed and observable.

Care is needed in interpreting such results, of course: the specific numbers can be highly dependent on the test harness; such testing, of highly nondeterministic systems, is not guaranteed to produce all the executions that an implementation might produce; and the architectures are intentionally looser in some respects than current implementations, so (as we will see later) some behaviour may be architecturally allowed even though it is never observable in current processors. Moreover, there might be differences between our architectural models, the vendor’s architectural intent, and the vendor’s architecture manuals (the ARM ARM [ARM08a] and POWER ISA [Pow09]). And of course, while our models are based in part on extensive discussion with ARM and IBM architects and designers, we do not speak for either vendor. All that said, we have reasonable confidence in our models, and we have found our testing process to be reasonably discriminating. Wherever we mark a test execution as allowed or forbidden, we believe that that does match the architectural intention, and, unless otherwise stated, everything marked as allowed is observable on some implementation of one or other architecture, and (modulo processor errata, which we do not discuss here) everything marked as forbidden has not been observable. We give some summary test data to illustrate this in each section.

**3.1.3 Explaining the Behaviour** To explain *why* some particular relaxed behaviour is allowed by an architecture, or why it may be observable on a processor implementation of that architecture, one might refer to the processor-vendor architecture text, or to the microarchitectural details of the implementation, or to a formal model that attempts to capture one or the other, or to an intuitive description of such a model. All have their uses and disadvantages. Reference to vendor architecture texts has the advantage of generality (as these aim to apply to all processor implementations of that architecture) and of the authority of the vendor. But they are often less clear than one might hope when it comes to relaxed-memory behaviour. A full microarchitectural explanation is in principle completely precise, but would require detailed knowledge of a processor implementation, which is typically commercially confidential, and it would also only apply to that implementation. For the third option, our formal models [OSS09, SSO<sup>+</sup>10, SSA<sup>+</sup>11, SMO<sup>+</sup>12] aim to capture the vendor’s architectural intent, and be consistent with the observed behaviour, for x86 and POWER. They are expressed in rigorous mathematics, and so are completely precise, and that mathematics can also be used to automatically generate testing and exploration tools, as for our `ppcmem` tool. But that mathematical detail can make them less accessible to a broad audience than one would like. Accordingly, in this tutorial we take advantage of the fact that those formal models are in an “abstract microarchitecture” style: to build intuition for the observable behaviour of the formal architectural model (and hence of the processor implementations and vendor architecture), we explain the behaviour of tests using the style and terminology of the abstract machines we introduced in Section 2. To avoid overwhelming detail, we do not describe the formal models completely; for that, one should refer to the papers cited above.

To explain the observed behaviour for MP on ARM and POWER, there are three distinct possibilities: the writes on Thread 0 are to distinct addresses and so can be committed out of order on Thread 0; and/or they might also be propagated to Thread 1 in either order; and/or the reads on Thread 1 (likewise to distinct addresses) can be satisfied out of order, binding the values they read in the opposite order to program order (the reads might also be committed in-order or out of order, but that has no effect on the outcome here). Any one of these microarchitectural features in isolation is sufficient to explain the non-sequentially-consistent behaviour above.

**3.1.4 Test Execution Diagrams** In most of the examples we will use, the final state constraint uniquely as a single candidate execution, implicitly specifying which write any read reads from, how any control-flow choices are resolved, and how, for each location, the writes to that location are ordered among themselves (their *coherence orders* that we discuss in Section 8). It is helpful to think of this execution diagrammatically, abstracting from the code. For example, we can depict the above MP test by the execution diagram below.



The diagram has a node for each memory access. The nodes are labelled (a, b, c, d), so that we can refer to them, and each specifies whether it is a write (W) or read (R), its location (here x or y), and the value read or written in this execution (here 0 or 1). The events are laid out in one column per thread, and there is a *program order* (po) edge between the two events of each thread (here this happens to be the same as the syntactic order of the two instructions in the source code, but in general it represents a choice of a control-flow path for that thread, unfolding any branches). The *reads-from* (rf) edge from b to c indicates that read c reads from the write b, and the reads-from edge from a red dot to read d indicates that the latter reads from the initial state. We will introduce other kinds of nodes and edges as we use them. Sometimes we mark diagrams to indicate that the execution is allowed or forbidden (in our models and our best understanding of the vendors’ architectural intent).

**3.1.5 Pseudocode vs Real code** In general, relaxed-memory behaviour can be introduced both by the hardware, from microarchitectural optimisations, and by the compiler, from compiler optimisations. For example, a compiler might well do common subexpression elimination (CSE), thereby reordering accesses to different locations within each thread’s code. In this tutorial, we are talking *only* about how the hardware executes machine-code assembly instructions, and the pseudocode we give must be understood as a description of the actual assembly instructions being executed, not as a program in C, Java, or another high-level language. We therefore take the definitive versions

of the test to be the POWER and ARM assembly code, as on the right below, not the pseudocode above and on the left. The assembly code can be harder to read for those unfamiliar with it, but as we usually use execution diagrams as above, this is not often a problem.

MP		Pseudocode		MP		ARM		MP		POWER	
Thread 0		Thread 1		Thread 0		Thread 1		Thread 0		Thread 1	
x=1		r1=y		MOV R0,#1		LDR R0,[R3]		li r1,1		lwz r1,0(r2)	
y=1		r2=x		STR R0,[R2]		LDR R1,[R2]		stw r1,0(r2)		lwz r3,0(r4)	
Initial state: $x=0 \wedge y=0$				MOV R1,#1				li r3,1			
Allowed: $1:r1=1 \wedge 1:r2=0$				STR R1,[R3]				stw r3,0(r4)			
				Initial state: $0:R2=x \wedge 0:R3=y \wedge 1:R2=x$				Initial state: $0:r2=x \wedge 0:r4=y \wedge 1:r2=y$			
				$\wedge 1:R3=y$				$\wedge 1:r4=x$			
				Allowed: $1:R0=1 \wedge 1:R1=0$				Allowed: $1:r1=1 \wedge 1:r3=0$			

**3.1.6 Undefined behaviour and data races** By focussing on the execution of machine code, we can also sidestep the fact that in some high-level languages certain programs have undefined behaviour. For example, in C11/C++11, unless `y` were declared to be *atomic*, the code above would give rise to a data race, making any program that executed it undefined, and if it was declared atomic the compiler might introduce various assembly-language fences (depending on the memory-order parameters of the atomic accesses). At the machine-code level, all programs have well-defined (albeit typically nondeterministic) behaviour even if they have races.

**3.1.7 Real usage of the MP idiom** In the MP test as shown above, the data `x` is just a single memory location, but in real usage one might have multi-word data. For most or all of the MP variations that we explore later, that should make no difference.

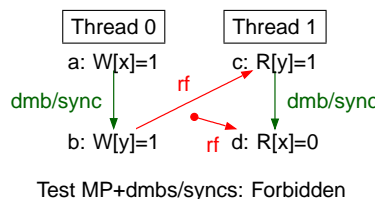
**3.1.8 Running the example in ppcmem** To interactively explore the behaviour of this example using our `ppcmem` tool, go to <http://www.cl.cam.ac.uk/~pes20/ppcmem>, click on *Change to ARM model* if desired, click on *Select POWER/ARM Test* and select MP from the menu, and click on *Interactive*. The screen will show the state of our model (we do not give all the details here, but they are described in our PLDI 2011 and PLDI 2012 papers [SSA<sup>+</sup>11, SMO<sup>+</sup>12]) running that test, with the green underlined options the possible model transitions; one can click on those to explore particular possible behaviours. Alternatively, there is a direct link to run `ppcmem` on each POWER test via the *Tests and Test Results* link at <http://www.cl.cam.ac.uk/~pes20/ppc-supplemental>.

## 3.2 Enforcing Order with Strong (dmb/sync) Barriers

To regain order, the programmer must defend against all of the above out-of-order possibilities. A strong memory barrier (or *fence*) instruction inserted between the two writes on Thread 0, and between the two reads on Thread 1, suffices. On POWER this would be the `sync` instruction (also written as `hwsync`), and on ARM it would be `DMB`. The resulting litmus tests are given below.

MP+dmb/syncs		Pseudocode		MP+dmb		ARM		MP+syncs		POWER	
Thread 0		Thread 1		Thread 0		Thread 1		Thread 0		Thread 1	
x=1		r1=y		MOV R0,#1		LDR R0,[R3]		li r1,1		lwz r1,0(r2)	
dmb/sync		dmb/sync		STR R0,[R2]		DMB		stw r1,0(r2)		sync	
y=1		r2=x		MOV R1,#1		LDR R1,[R2]		li r3,1		lwz r3,0(r4)	
Initial state: $x=0 \wedge y=0$				STR R1,[R3]				stw r3,0(r4)			
Forbidden: $1:r1=1 \wedge 1:r2=0$				Initial state: $0:R2=x \wedge 0:R3=y \wedge 1:R2=x$				Initial state: $0:r2=x \wedge 0:r4=y \wedge 1:r2=y$			
				$\wedge 1:R3=y$				$\wedge 1:r4=x$			
				Forbidden: $1:R0=1 \wedge 1:R1=0$				Forbidden: $1:r1=1 \wedge 1:r3=0$			

We illustrate the execution of interest as below, with green `dmb/sync` arrows to indicate memory accesses separated by a `sync` or a `DMB` instruction.



Microarchitecturally, and in our models, the `sync` or `DMB` on Thread 0 keeps the two writes in order, both locally and in the order they propagate to other threads in the system. Meanwhile, the `dmb/sync` on Thread 1 forces the two reads to be satisfied, binding their values, in program order, by stopping the second read from being satisfied before the first one is. Note that, as we shall see below, this is much stronger than necessary, with various dependencies or lighter flavours of barriers being sufficient in this case.

The `dmb/sync` barriers are potentially expensive, but satisfy the property that if inserted between every pair of memory accesses, they restore sequentially consistent behaviour. Looking at the four cases of a pair of a read or write before and after a barrier in more detail, we have:

**RR** For two reads separated by a `dmb/sync`, the barrier will ensure that they are satisfied in program order, and also will ensure that they are committed in program order.

**RW** For a read before a write, separated by a `dmb/sync`, the barrier will ensure that the read is satisfied (and also committed) before the write can be committed, and hence before the write can be propagated and thereby become visible to any other thread.

**WW** For a write before a write, separated by a `dmb/sync`, the barrier will ensure that the first write is committed and has propagated to all other threads before the second write is committed, and hence before the second write can propagate to any other thread.

**WR** For a write before a read, separated by a `dmb/sync`, the barrier will ensure that the write is committed and has propagated to all other threads before the read is satisfied.

We emphasise that these descriptions are all *as far as the programmer's model is concerned*. An actual hardware implementation might be more aggressive, e.g. with some speculative execution of instructions that follow a barrier, or a microarchitectural structure that allows more write propagation, so long as the programmer cannot detect it.

### 3.3 Enforcing Order with the POWER `lwsync` Barrier

On POWER, there is a ‘lightweight sync’ `lwsync` instruction, which is weaker and potentially faster than the ‘heavyweight sync’ or `sync` instruction, and for this message-passing example the `lwsync` suffices, on both the writer and reader side of the test. Looking again at the four cases above:

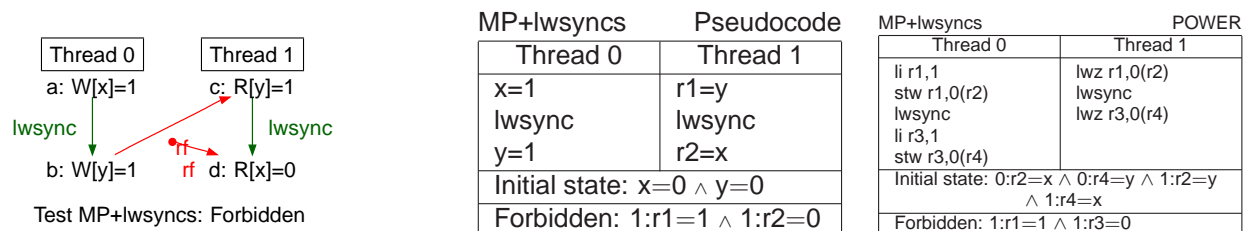
**RR** For two reads separated by an `lwsync`, just like `sync`, the barrier will ensure that they are satisfied in program order, and also will ensure that they are committed in program order.

**RW** For a read before a write, separated by a `lwsync`, just like `sync`, the barrier will ensure that the read is satisfied (and also committed) before the write can be committed, and hence before the write can be propagated and thereby become visible to any other thread.

**WW** For a write before a write, separated by a `lwsync`, the barrier will ensure that for any particular other thread, the first write<sup>1</sup> propagates to that thread before the second does.

**WR** For a write before a read, separated by a `lwsync`, the barrier will ensure that the write is committed before the read is satisfied, but lets the read be satisfied before the write has been propagated to any other thread.

In this message-passing example, we just need the WW and RR cases; an `lwsync` on the writing thread keeps the two writes in order (their commit and propagation) as far as the reading thread is concerned, and an `lwsync` on the reading thread ensures that the reads are satisfied in program order.



<sup>1</sup>Or a coherence successor thereof; see Section 8.

We show a case where the weakness of `lwsync` really matters in test `SB+lwsyncs`, in Section 6. ARM does not have an analogue of `lwsync`.

### 3.4 Observed Behaviour

Below we show experimental data for these tests: for `MP+dmbs` and `MP+syncs` on ARM and POWER, and for `MP+lwsyncs` just on POWER.

		POWER			ARM			
	Kind	PowerG5	Power6	Power7	Tegra2	Tegra3	APQ8060	A5X
MP	Allow	10M/4.9G	6.5M/29G	1.7G/167G	40M/3.8G	138k/16M	61k/552M	437k/185M
MP+dmbs/syncs	Forbid	0/6.9G	0/40G	0/252G	0/24G	0/39G	0/26G	0/2.2G
MP+lwsyncs	Forbid	0/6.9G	0/40G	0/220G	—	—	—	—

Here the allowed result for MP is observable on all platforms, while the forbidden results for the variants with barriers are not observable on any platform.

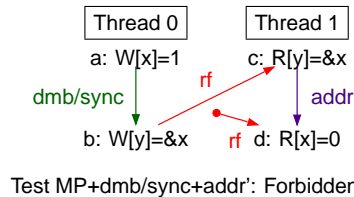
## 4 Enforcing Order with Dependencies

In fact, on the read side of the message-passing example, the `sync`, `lwsync`, and `DMB` memory barriers used above are stronger than necessary: one can enforce enough ordering to prohibit the undesired outcome just by relying on various kinds of *dependency* in the code. In this section we explain what those are and what their force is. In later sections we use dependencies in examples that illustrate some other relaxed-memory properties of the machines. For POWER, in all the examples of this section one could replace the `sync` on the writing thread with `lwsync` without affecting the results.

### 4.1 Address Dependencies

The simplest kind of dependency is an *address dependency*. There is an address dependency from a read instruction to a program-order-later read or write instruction when the value read by the first is used to compute the address used for the second. In the variation of MP below, instead of writing a flag value of 1, the writer Thread 0 writes the address of location `x`, and the reader Thread 1 uses that address for its second read. That dependency is enough to keep the two reads satisfied in program order on Thread 1: the second read cannot get started until its address is (perhaps speculatively) known, so the second read cannot be satisfied until the first read is satisfied (in other words, the ARM and POWER architectures do not allow *value speculation* of addresses). Combining that with the `dmb/sync` on Thread 0 (which keeps the write to `x` and the write to `y` in order as far as any other thread is concerned) is enough to prevent Thread 1 reading 0 from `x` if it has read `&x` from `y`.

MP+dmb/sync+addr'	Pseudocode
Thread 0	Thread 1
<code>x=1</code>	<code>r1=y</code>
<code>dmb/sync</code>	
<code>y=&amp;x</code>	<code>r2=*r1</code>
Initial state: <code>x=0 ∧ y=0</code>	
Forbidden: <code>1:r1=&amp;x ∧ 1:r2=0</code>	



Note that there is a slight mismatch here between the C-like syntax of our pseudocode, in which `x` is a C variable and `&x` its address, and the notation of our assembly examples, in which `x` is a location.

**4.1.1 Compound Data** To see that this message-passing-with-dependency idiom can still work correctly if the data (the value stored at `x`) were multi-word, note that all the writes to the parts of the data would precede the `dmb/sync` on Thread 0, while all the reads of the parts of the data should each be address-dependent on the value read from `y` on Thread 1, by some offset calculation from that value.

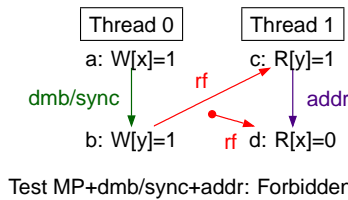
**4.1.2 C11 Consume** This preserved address dependency is what is made available in the C/C++11 memory models by their *read-consume* atomic operations: if the read of *y* were tagged as a read-consume at the C/C++ level, then compilers are required to respect the dependency to the second Thread 1 read (or to implement something stronger, such as an *lwsync/sync/dmb* barrier between them.

**4.1.3 Address Dependencies from Reads to Writes** An address dependency from a read to a write has a similar effect to one from a read to a read, preventing the write getting started until the read has been satisfied and a value for the read is known; we illustrate this with the test **S** in Section 4.4 below.

**4.1.4 Artificial Dependencies** Above we said that “there is an address dependency from a read instruction to a program-order-later read or write instruction when the value read by the first is *used to compute* the address used for the second”, and that computation may be via any data-flow path through registers and arithmetic or logical operations (though not via memory) — even if the value of the address used cannot be affected by the value read. In the **MP+dmb/sync+addr** variant below, the value read is exclusive-or’d with itself and then added to the (constant) address of *x* to calculate the address to be used for the second Thread 1 read. The result of the exclusive-or will always be zero, and so the address used for the second read will always be equal to that of *x*, but the two reads are still kept in order. Adding such an *artificial* dependency (sometimes these are known, perhaps confusingly, as *false* or *fake* dependencies) can be a useful programming idiom, to enforce some ordering from a read to a later read (or write) at low run-time cost.

MP+dmb/sync+addr	Pseudocode	MP+dmb+addr	ARM	MP+sync+addr	POWER
Thread 0	Thread 1	Thread 0	Thread 1	Thread 0	Thread 1
x=1 dmb/sync y=1	r1=y r3=(r1 xor r1) r2=*(&x + r3)	MOV R0,#1 STR R0,[R2] DMB MOV R1,#1 STR R1,[R3]	LDR R0,[R4] EOR R1,R0,R0 LDR R2,[R1,R3]	li r1,1 stw r1,0(r2) sync li r3,1 stw r3,0(r4)	lwz r1,0(r2) xor r3,r1,r1 lwzx r4,r3,r5
Initial state: $x=0 \wedge y=0$		Initial state: $0:R2=x \wedge 0:R3=y \wedge 1:R3=x \wedge 1:R4=y$		Initial state: $0:r2=x \wedge 0:r4=y \wedge 1:r2=y \wedge 1:r5=x$	
Forbidden: $1:r1=1 \wedge 1:r2=0$		Forbidden: $1:R0=1 \wedge 1:R2=0$		Forbidden: $1:r1=1 \wedge 1:r4=0$	

As artificial address dependencies behave just like natural ones, we draw them in the same way, with **addr** edges, abstracting from the details of exactly what address computation is done:



## 4.2 Control Dependencies

A rather different kind of dependency is a *control dependency*, from a read to a program-order-later read or write instruction, where the value read by the first read is used to compute the condition of a conditional branch that is program-order-before the second read or write.

A control dependency from a read to a read has little force, as we see in the **MP+dmb+ctrl** and **MP+sync+ctrl** examples below: ARM and POWER processors can speculatively execute past the conditional branch (perhaps following one path based on a branch prediction, or in principle following both paths simultaneously), and so satisfy the second read before satisfying the first read. The first read, the branch, and the second read might then all be committed (with those values) in program order.



MP+dmb/sync+ctrl Pseudocode

Thread 0	Thread 1
x=1 dmb/sync y=1	r1=y if (r1 == r1) {} r2=x
Initial state: $x=0 \wedge y=0$	
Allowed: $1:r1=1 \wedge 1:r2=0$	

MP+dmb+ctrl

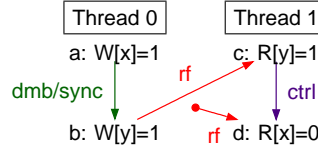
Thread 0	Thread 1
MOV R0,#1 STR R0,[R2] DMB MOV R1,#1 STR R1,[R3]	LDR R0,[R3] CMP R0,R0 BNE LC00 LC00: LDR R1,[R2]
Initial state: $0:R2=x \wedge 0:R3=y \wedge 1:R2=x \wedge 1:R3=y$	
Allowed: $1:R0=1 \wedge 1:R1=0$	

ARM

MP+sync+ctrl

Thread 0	Thread 1
li r1,1 stw r1,0(r2) sync li r3,1 stw r3,0(r4)	lwz r1,0(r2) cmpw r1,r1 beq LC00 LC00: lwz r3,0(r4)
Initial state: $0:r2=x \wedge 0:r4=y \wedge 1:r2=y \wedge 1:r4=x$	
Allowed: $1:r1=1 \wedge 1:r3=0$	

POWER



Test MP+dmb/sync+ctrl: Allowed

For compactness, we show examples in which the branch is just to the next instruction, an instance of which will be executed whether or not the branch is taken. This makes no difference: the behaviour would be just the same if the branch were to a different location and the second read were only executed in one case.

The value of the branch condition in the examples is also unaffected by the value read, as it is just based on an equality comparison of a register with itself. Just as for the artificial address dependencies described above, this also makes no difference: the existence of the control dependency simply relies on the fact that the value read is used in the computation of the condition, not on whether the value of the condition would be changed if a different value were read. There are therefore many ways of writing a pseudocode example that are essentially equivalent to that above, e.g. by putting the  $r2=x$  inside the conditional, or, for an example without a race on  $x$ , putting the read of  $y$  in a loop such as  $\text{do } \{r1=y;\} \text{ while } (r1 == 0)$ .

### 4.3 Control-isb/isync Dependencies

To give a read-to-read control dependency some force, one can add an ISB (ARM) or isync (POWER) instruction between the conditional branch and the second read, as in the examples below. This prevents the second read from being satisfied until the conditional branch is committed, which cannot happen until the value of the first read is fixed (i.e., until that read is satisfied and committed); the two reads are thus kept in order and the specified outcome of the test is now forbidden.

MP+dmb/sync+ctrlisb/ctrlisync

Thread 0	Thread 1
x=1 dmb/sync y=1	r1=y if (r1 == r1) {} isb/isync r2=x
Initial state: $x=0 \wedge y=0$	
Forbidden: $1:r1=1 \wedge 1:r2=0$	

MP+dmb+ctrlisb

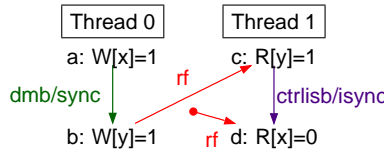
Thread 0	Thread 1
MOV R0,#1 STR R0,[R2] DMB MOV R1,#1 STR R1,[R3]	LDR R0,[R3] CMP R0,R0 BNE LC00 LC00: ISB LDR R1,[R2]
Initial state: $0:R2=x \wedge 0:R3=y \wedge 1:R2=x \wedge 1:R3=y$	
Forbidden: $1:R0=1 \wedge 1:R1=0$	

ARM

MP+sync+ctrlisync

Thread 0	Thread 1
li r1,1 stw r1,0(r2) sync li r3,1 stw r3,0(r4)	lwz r1,0(r2) cmpw r1,r1 beq LC00 LC00: isync lwz r3,0(r4)
Initial state: $0:r2=x \wedge 0:r4=y \wedge 1:r2=y \wedge 1:r4=x$	
Forbidden: $1:r1=1 \wedge 1:r3=0$	

POWER



Test MP+dmb/sync+ctrlisb/ctrlisync: Forbidden

### 4.4 Control Dependencies from a Read to a Write

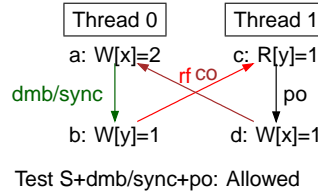
In contrast to control dependencies (without an isb/isync) from a read to a read, a control dependency from a read to a *write* does have some force: the write cannot be seen by any other thread until the branch is committed, and hence



until the value of the first read is fixed. To illustrate this we use a variation of the MP test family, known (for historical reasons) as S, in which the second read on Thread 1 (of x) is replaced by a write of x. Instead of asking whether the Thread 1 read of x is guaranteed to see the value written by Thread 0, we now ask whether the Thread 1 write of x is guaranteed to be *coherence-after* the Thread 0 write of x (i.e., whether a third thread, that reads x twice, is guaranteed not to see those two writes in the opposite order; we return to coherence in Section 8). Without a control dependency on Thread 1, that is not guaranteed; the execution below is allowed.

S+dmb/sync+po	Pseudocode	S+dmb+po	ARM	S+sync+po	POWER
Thread 0	Thread 1	Thread 0	Thread 1	Thread 0	Thread 1
x=2 dmb/sync y=1	r1=y x=1	MOV R0,#2 STR R0,[R2] DMB MOV R1,#1 STR R1,[R3]	LDR R0,[R3] MOV R1,#1 STR R1,[R2]	li r1,2 stw r1,0(r2) sync li r3,1 stw r3,0(r4)	lwz r1,0(r2) li r3,1 stw r3,0(r4)
Initial state: $x=0 \wedge y=0$		Initial state: $0:R2=x \wedge 0:R3=y \wedge 1:R2=x \wedge 1:R3=y$		Initial state: $0:r2=x \wedge 0:r4=y \wedge 1:r2=y \wedge 1:r4=x$	
Forbidden: $1:r1=1 \wedge x=2$		Allowed: $[x]=2 \wedge 1:R0=1$		Allowed: $[x]=2 \wedge 1:r1=1$	

We draw such coherence conditions with a CO edge, between two writes to the same address:



If we add a read-to-write control dependency on Thread 1, that final outcome becomes forbidden:

S+dmb/sync+ctrl	Pseudocode	S+dmb+ctrl	ARM	S+sync+ctrl	POWER
Thread 0	Thread 1	Thread 0	Thread 1	Thread 0	Thread 1
x=2 dmb/sync y=1	r1=y if (r1==r1) { } x=1	MOV R0,#2 STR R0,[R2] DMB MOV R1,#1 STR R1,[R3]	LDR R0,[R3] CMP R0,R0 BNE LC00 LC00: MOV R1,#1 STR R1,[R2]	li r1,2 stw r1,0(r2) sync li r3,1 stw r3,0(r4)	lwz r1,0(r2) cmpw r1,r1 beq LC00 LC00: li r3,1 stw r3,0(r4)
Initial state: $x=0 \wedge y=0$		Initial state: $0:R2=x \wedge 0:R3=y \wedge 1:R2=x \wedge 1:R3=y$		Initial state: $0:r2=x \wedge 0:r4=y \wedge 1:r2=y \wedge 1:r4=x$	
Forbidden: $1:r1=1 \wedge x=2$		Forbidden: $[x]=2 \wedge 1:R0=1$		Forbidden: $[x]=2 \wedge 1:r1=1$	

Test S+dmb/sync+ctrl: Forbidden

## 4.5 Data Dependencies from a Read to a Write

Our final kind of dependency is a *data dependency*, from a read to a program-order-later write where the value read is used to compute the value written. These have a broadly similar effect to address, control, or control-isb/isync dependencies from reads to writes: they prevent the write being committed (and hence being propagated and becoming visible to other threads) until the value of the read is fixed when the read is committed. Accordingly, the S+dmb/sync+data variant below of the above S+dmb/sync+ctrl test is also forbidden.

#### S+dmb/sync+data Pseudocode

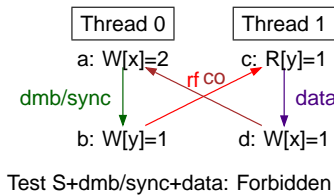
Thread 0	Thread 1
x=2 dmb/sync y=1	r1=y r3 = (r1 xor r1) x = 1 + r3 r2=x
Initial state: $x=0 \wedge y=0$	
Forbidden: $1:r1=0 \wedge x=1$	

#### S+dmb+data ARM

Thread 0	Thread 1
MOV R0,#2 STR R0,[R2] DMB MOV R1,#1 STR R1,[R3]	LDR R0,[R3] EOR R1,R0,R0 ADD R1,R1,#1 STR R1,[R2]
Initial state: $0:R2=x \wedge 0:R3=y \wedge 1:R2=x \wedge 1:R3=y$	
Forbidden: $[x]=2 \wedge 1:R0=1$	

#### S+sync+data POWER

Thread 0	Thread 1
li r1,2 stw r1,0(r2) sync li r3,1 stw r3,0(r4)	lwz r1,0(r2) xor r3,r1,r1 addi r3,r3,1 stw r3,0(r4)
Initial state: $0:r2=x \wedge 0:r4=y \wedge 1:r2=y \wedge 1:r4=x$	
Forbidden: $[x]=2 \wedge 1:r1=1$	



## 4.6 Summary of Dependencies

To summarise, we have:

**RR and RW address** an address dependency from a read to a program-order-later read or write where the value read by the first is used to compute the address of the second;

**RR and RW control** a control dependency from a read to a program-order-later read or write where the value read by the first is used to compute the condition of a conditional branch that is program-order-before the second;

**RR and RW control-isb/isync** a control-isb or control-isync dependency from a read to a program-order-later read or write where the value read by the first is used to compute the condition of a conditional branch that is program-order-before an isb/isync instruction before the second; and

**RW data** a data dependency from a read to a program-order-later write where the value read by the first is used to compute the value written by the second.

There are no dependencies from writes (to either reads or writes).

In each case, the use of the value read can be via any dataflow chain of register-to-register operations, and it does not matter whether it is artificial (or fake/false) or not: there is still a dependency even if the value read cannot affect the actual value used as address, data, or condition.

From one read to another, an address or control-isb/isync dependency will prevent the second read being satisfied before the first is, while a plain control dependency will not.

From a read to a write, an address, control (and so also a control-isb/isync) or data dependency will prevent the write being visible to any other thread before the value of the read is fixed.

We return in Section 10 to some more subtle properties of dependencies.

As we shall see, dependencies are strictly weaker than the DMB, sync, and lwsync barriers: replacing a dependency by one of those barriers will never permit more behaviour (and so should always be a safe program transformation), whereas the converse is not true. Dependencies only have thread-local effects, whereas DMB, sync, and lwsync have stronger ‘cumulatively’ properties that we introduce in the next section.

## 4.7 Observed Behaviour

Below we summarise the results of hardware experiments on dependencies.

	Kind	POWER			ARM			
		PowerG5	Power6	Power7	Tegra2	Tegra3	APQ8060	A5X
MP	Allow	10M/4.9G	6.5M/29G	1.7G/167G	40M/3.8G	138k/16M	61k/552M	437k/185M
MP+dmb/sync+po	Allow	670k/2.4G	0/26G <sup>U</sup>	13M/39G	3.1M/3.9G	50/28M	69k/743M	249k/195M
MP+dmb/sync+addr	Forbid	0/6.9G	0/40G	0/252G	0/29G	0/39G	0/26G	0/2.2G
MP+dmb/sync+ctrl	Allow	363k/5.5G	0/43G <sup>U</sup>	27M/167G	5.7M/3.9G	1.5k/53M	556/748M	1.5M/207M
MP+dmb/sync+ctrlsib/isync	Forbid	0/6.9G	0/40G	0/252G	0/29G	0/39G	0/26G	0/2.2G
S+dmb/sync+po	Allow	0/2.4G <sup>U</sup>	0/18G <sup>U</sup>	0/35G <sup>U</sup>	271k/4.0G	84/58M	357/1.8G	211k/202M
S+dmb/sync+ctrl	Forbid	0/2.1G	0/14G	0/29G	0/24G	0/39G	0/26G	0/2.2G
S+dmb/sync+data	Forbid	0/2.1G	0/14G	0/29G	0/24G	0/39G	0/26G	0/2.2G

The experimental data shows that the forbidden results are all non-observable. Some of the allowed results, on the other hand, are not observable on some implementations, as highlighted in blue and tagged with a superscript **U** (allowed-Unseen): for **MP+sync+po** and **MP+sync+ctrl** POWER 6 does not exhibit the allowed behaviour (in this sense it has a more in-order pipeline than either POWER G5 or POWER 7), and for **S+sync+po** none of these POWER implementations do. It appears that these implementations do not commit writes when there is an outstanding program-order-earlier read, even to a different address; though of course other and future implementations may differ.

These are all cases where the particular implementations are tighter than the architectural intent, and the fact that this can and does change from one processor generation to another reinforces the fact that programmers aiming to write portable code must be concerned with the architectural specification, not just their current implementation.

## 5 Iterated Message Passing on more than two threads and Cumulativity (WRC and ISA2)

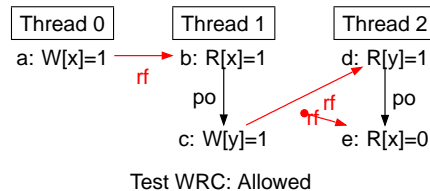
Up to this point, all our examples have used only two threads. Generalising to three or four threads reveals a new phenomenon: on POWER and ARM, two threads can observe writes to different locations in different orders, even in the absence of any thread-local reordering. In other words, the architectures are not *multiple-copy atomic* [Col92]. To see this, consider first a three-thread variant of MP in which the first write has been pulled off to another thread, with Thread 1 busy-waiting to see it before doing its own write:

WRC-loop		Pseudocode
Thread 0	Thread 1	Thread 2
x=1	while (x==0) {} y=1	while (y==0) {} r3=x
Initial state: x=0 ∧ y=0		
Forbidden?: 2:r3=0		

This test was known as WRC, for ‘write-to-read causality’ in Boehm and Adve [BA08].

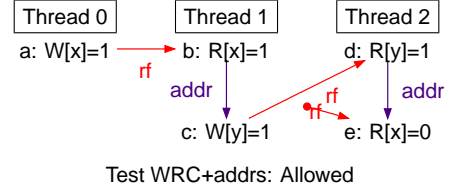
As before, we simplify the example without really affecting what is going on by removing the loops, replacing them by a final-state constraint that restricts attention to the executions in which Thread 1 reads x=1 and Thread 2 reads y=1. The question is whether such an execution can also see x=0 (instead of reading from the Thread 0 write of x=1).

WRC		Pseudocode
Thread 0	Thread 1	Thread 2
x=1	r1=x y=1	r2=y r3=x
Initial state: x=0 ∧ y=0		
Allowed: 1:r1=1 ∧ 2:r2=1 ∧ 2:r3=0		



Without any dependencies or barriers, this is trivially allowed: the Thread 1 read and write are to different addresses and can be reordered with each other, and likewise the Thread 2 reads can be satisfied out of program order. Adding artificial dependencies to prevent those reorderings gives us the **WRC+adrs** test below.

WRC+addrs		Pseudocode
Thread 0	Thread 1	Thread 2
x=1	r1=x *(&y+r1-r1) = 1	r2=y r3 = *(&x + r2 - r2)
Initial state: x=0 $\wedge$ y=0		
Allowed: 1:r1=1 $\wedge$ 2:r2=1 $\wedge$ 2:r3=0		

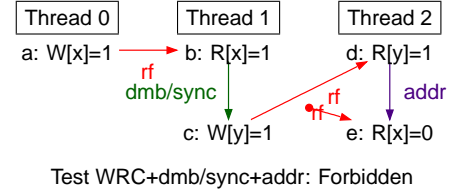


On a multiple-copy-atomic architecture this would be forbidden, but on ARM and POWER it is allowed. Thread 2 has to do its reads in program order, but the fact that Thread 1 sees the Thread 0 write of x=1 before starting its write of y=1 does not prevent those writes propagating to Thread 2 in the opposite order, allowing it to read y=1 and then read x=0. We have observed this on POWER implementations, and expect it to be observable on some ARM processors with more than two hardware threads, but we have not yet observed it on the only such machine that we have access to at present (Tegra3).

## 5.1 Cumulative Barriers for WRC

To prevent the unintended outcome of WRC, one can strengthen the Thread 1 address dependency above, replacing it by a DMB or sync barrier (on POWER the weaker lwsync barrier also suffices).

WRC+dmbsync+addr		Pseudocode
Thread 0	Thread 1	Thread 2
x=1	r1=x dmb/sync y=1	r2=y r3 = *(&x + r2 - r2)
Initial state: x=0 $\wedge$ y=0		
Forbidden: 1:r1=1 $\wedge$ 2:r2=1 $\wedge$ 2:r3=0		



In MP+syncs we saw that a DMB or sync barrier keeps any write done by a thread before the barrier in order with respect to any write done by the same thread after the barrier, as far as any other thread is concerned.

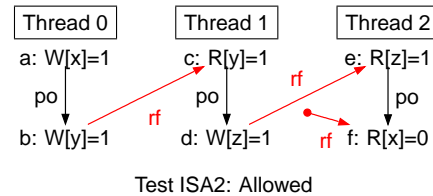
Here the Thread 1 DMB or sync barrier also keeps any write that Thread 1 has read from (before the barrier) in order with respect to any write that Thread 1 does after the barrier, as far as any other thread (e.g., Thread 2) is concerned. More generally, the barrier ensures that any write that has propagated to Thread 1 before the barrier is propagated to any other thread before the Thread 1 writes after the barrier can propagate to that other thread. This *cumulative* property is essential for iterated message-passing examples.

As minor variations, one could also weaken the Thread 1 barrier to a POWER lwsync, giving the test WRC+lwsync+addr, or strengthen the Thread 2 address dependency to another DMB or sync barrier, giving the test WRC+dmbs or WRC+syncs; these all have the same possible outcomes as the test above.

## 5.2 Cumulative Barriers for ISA2

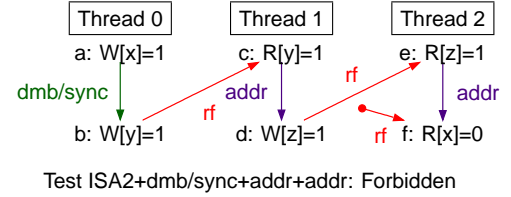
The WRC test extends the message-passing (MP) example on the left, and WRC+dmbsync+addr shows one aspect of cumulative barriers there. The ISA2 example below shows another aspect of cumulatity. The example (a simplified version of [Pow09, §1.7.1, Example 2], replacing loops by a final state constraint as usual) extends the MP example on the right, interposing a Thread 1 write and Thread 2 read of a third shared variable, z, before the final read of x. One could think of this as Thread 0 writing some possibly compound data into x, then setting a flag y; Thread 1 waiting for that flag then writing another flag z, and Thread 2 waiting for that flag before reading the data; as usual, one would like to prevent the possibility that that reads the initial state value for the data (or for part of it).

ISA2		Pseudocode
Thread 0	Thread 1	Thread 2
x=1 y=1	r1=y z=1	r2=z r3=x
Initial state: x=0 $\wedge$ y=0 $\wedge$ z=0		
Allowed: 1:r1=1 $\wedge$ 2:r2=1 $\wedge$ 2:r3=0		



To make this work (i.e., to forbid the stated final state), it suffices to have a DMB or sync barrier (or POWER lwsync) on Thread 0 and preserved dependencies on Threads 1 and 2 (an address, data or control dependency between the Thread 1 read/write pair, and an address or control-isb/isync dependency between the Thread 2 read/read pair). Those dependencies could be replaced by DMB/sync/lwsync.

ISA2+dmbsync+addr+addr		Pseudocode
Thread 0	Thread 1	Thread 2
x=1 dmbsync y=1	r1=y *(&z+r1-r1)=1	r2=z r3 = *(&x +r2-r2)
Initial state: $x=0 \wedge y=0 \wedge z=0$		
Forbidden: $1:r1=1 \wedge 2:r2=1 \wedge 2:r3=0$		



Here one can think of the Thread 0 barrier as ensuring that the Thread 0 write of  $x=1$  propagates to Thread 1 before the barrier does, which in turn is before the Thread 0  $y=1$  propagates to Thread 1, which is before Thread 1 does its write of  $z=1$ . Cumulativity, applied to the  $x=1$  write before the barrier before the  $z=1$  write (all propagated to or done by Thread 1) then keeps the  $x=1$  and  $z=1$  writes in order as far as all other threads are concerned, and specifically as far as Thread 2 is concerned. As usual, the dependencies just prevent local reordering which otherwise would make the unintended result trivially possible.

### 5.3 Observed Behaviour

Below we summarise the results of hardware experiments for these cumulativity tests.

		POWER			ARM
	Kind	PowerG5	Power6	Power7	Tegra3
WRC	Allow	44k/2.7G	1.2M/13G	25M/104G	8.6k/8.2M
WRC+addrs	Allow	0/2.4G <sup>u</sup>	225k/4.3G	104k/25G	0/20G <sup>u</sup>
WRC+dmbsync+addr	Forbid	0/3.5G	0/21G	0/158G	0/20G
WRC+lwsync+addr	Forbid	0/3.5G	0/21G	0/138G	—
ISA2	Allow	3/91M	73/30M	1.0k/3.8M	6.7k/2.0M
ISA2+dmbsync+addr+addr	Forbid	0/2.3G	0/12G	0/55G	0/20G
ISA2+lwsync+addr+addr	Forbid	0/2.3G	0/12G	0/55G	—

These tests involve three hardware threads, while the Tegra2, APQ8060, and A5X implementations that we have access to support only two hardware threads. Accordingly, for ARM we give results only for Tegra3. As before, there is no ARM analogue of the lwsync variant.

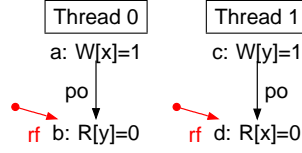
The results confirm that the forbidden results are not observable. For WRC+addrs, POWER G5 and ARM Tegra3 do not exhibit the architecturally-allowed possibility, while POWER 6 and POWER 7 do.

## 6 Store-buffering (SB) or Dekker's Examples

We now turn to a rather different two-thread example, which is a pattern that arises at the heart of some mutual exclusion algorithms. It is sometimes referred to as the *store-buffering* example (SB), as this is more-or-less the only relaxed-memory behaviour observable in the TSO model of x86 or Sparc with their FIFO (and forwardable) store buffers, and sometimes referred to as Dekker's example, as it appears in his mutual exclusion algorithm.

The two-thread version of the example has two shared locations, just like MP, but now each thread writes one location then reads from the other. The question is whether they can both (in the same execution) read from the initial state.

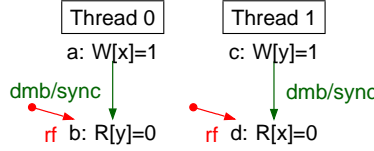
SB		Pseudocode		SB	ARM	SB	POWER
Thread 0		Thread 1		Thread 0		Thread 1	
x=1		y=1		MOV R0,#1		li r1,1	
r1=y		r2=x		STR R0,[R2]		stw r1,0(r2)	
Initial state: $x=0 \wedge y=0$		Initial state: $0:R2=x \wedge 0:R3=y \wedge 1:R2=x \wedge 1:R3=y$		LDR R1,[R3]		lwz r3,0(r4)	
Allowed: $0:r1=0 \wedge 1:r2=0$		Allowed: $0:R1=0 \wedge 1:R1=0$		Initial state: $0:r2=x \wedge 0:r4=y \wedge 1:r2=y \wedge 1:r4=x$		Allowed: $0:r3=0 \wedge 1:r3=0$	



Test SB: Allowed

Without any barriers or dependencies, that outcome is allowed, and, as there are no dependencies from writes, the only possible strengthening of the code is to insert barriers. Adding a DMB or sync on both threads suffices to rule out the unintended outcome:

SB+dmbs/syncs		Pseudocode		SB+dmbs	ARM	SB+syncs	POWER
Thread 0		Thread 1		Thread 0		Thread 1	
x=1		y=1		MOV R0,#1		li r1,1	
dmb/sync		dmb/sync		STR R0,[R2]		stw r1,0(r2)	
r1=y		r2=x		DMB		sync	
Initial state: $x=0 \wedge y=0$		Initial state: $0:R2=x \wedge 0:R3=y \wedge 1:R2=x \wedge 1:R3=y$		LDR R1,[R3]		lwz r3,0(r4)	
Forbidden: $0:r1=0 \wedge 1:r2=0$		Forbidden: $0:R1=0 \wedge 1:R1=0$		Initial state: $0:r2=x \wedge 0:r4=y \wedge 1:r2=y \wedge 1:r4=x$		Forbidden: $0:r3=0 \wedge 1:r3=0$	



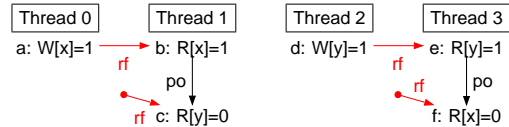
Test SB+dmbs/syncs: Forbidden

Here the dmb or sync barriers ensure that the program-order-previous writes must have propagated to all threads before the reads are satisfied, ruling out the given execution. On POWER, it does not suffice here to use lwsync barriers (or one lwsync and one sync barrier): the POWER lwsync does *not* ensure that writes before the barrier have propagated to any other thread before subsequent actions, though it does keep writes before and after an lwsync in order as far as all threads are concerned.

## 6.1 Extending SB to more threads: IRIW and RWC

Just as we extended the MP example by pulling out the first write to a new thread, to give the WRC example, we can extend SB by pulling out one or both writes to new threads. Pulling out both gives the Independent Reads of Independent Writes (IRIW) example below (so named by Lea). Threads 0 and 2 write to x and y respectively; Thread 1 reads x then y; and Thread 3 reads y then x.

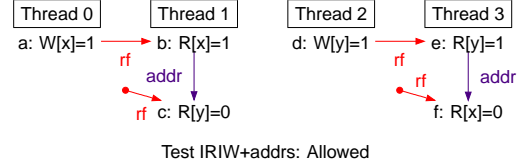
IRIW		Pseudocode	
Thread 0	Thread 1	Thread 2	Thread 3
x=1	r1=x r2=y	y=1	r3=y r4=x
Initial state: $x=0 \wedge y=0$			
Allowed: $1:r1=1 \wedge 1:r2=0 \wedge 3:r3=1 \wedge 3:r4=0$			



Test IRIW: Allowed

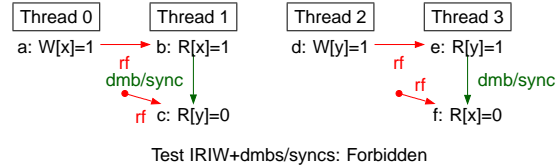
This gives us a striking illustration of the fact that writes can be propagated to different threads in different orders: in IRIW+adds below (where we add dependencies to the reading threads to rule out the trivial executions in which the reads are locally reordered), Thread 1 sees the write to x but not that to y, while Thread 3 sees the write to y but not that to x.

IRIW+addrs		Pseudocode	
Thread 0	Thread 1	Thread 2	Thread 3
x=1	r1=x r2=*&y+r1-r1	y=1	r3=y r4=*&x+r3-r3
Initial state: $x=0 \wedge y=0 \wedge z=0$			
Allowed: $1:r1=1 \wedge 1:r2=0 \wedge 3:r3=1 \wedge 3:r4=0$			



To rule out this behaviour one needs a DMB or sync on both of the reading threads (lwsyncs do not suffice here), just as for the SB test:

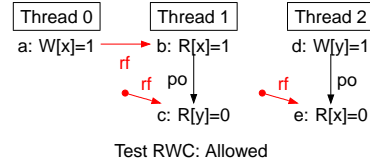
IRIW+dmbs/syncs		Pseudocode	
Thread 0	Thread 1	Thread 2	Thread 3
x=1	r1=x dmb/sync r2=y	y=1	r3=y dmb/sync r4=x
Initial state: $x=0 \wedge y=0$			
Allowed: $1:r1=1 \wedge 1:r2=0 \wedge 3:r3=1 \wedge 3:r4=0$			



We are not aware of any case where IRIW arises as a natural programming idiom (we would be glad to hear of any such), but it is a concern when one is implementing a high-level language memory model, perhaps with sequentially consistent behaviour for volatiles or atomics, above highly relaxed models such as ARM and POWER.

Pulling just one of the SB writes out to a new thread gives the RWC (for ‘read-to-write causality’) example of Boehm and Adve [BA08]:

RWC		Pseudocode
Thread 0	Thread 1	Thread 2
x=1	r1=x r2=y	y=1 r4=x
Initial state: $x=0 \wedge y=0 \wedge z=0$		
Allowed: $1:r1=1 \wedge 1:r2=0 \wedge 2:r4=0$		

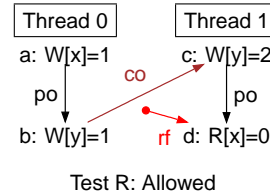


and that also needs two DMBs or syncs.

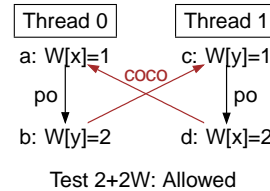
## 6.2 SB Variations with Writes: R and 2+2W

Two different variations of SB are obtained by replacing one or both of the reads by writes, analogous to the way we obtained S from MP earlier, with coherence edges in place of the reads from the initial state. We call these test families R and 2+2W respectively. Just as for IRIW, they are principally of interest when implementing a high-level language model (that has to support arbitrary high-level language programs) above ARM or POWER; we are not yet aware of cases where they arise in natural programming idioms.

R		Pseudocode
Thread 0	Thread 1	
x=1 y=1	y=2 r1=x	
Initial state: $x=0 \wedge y=0$		
Allowed: $y=2 \wedge 1:r1=0$		



2+2W		Pseudocode
Thread 0	Thread 1	
x=1 y=2	y=1 x=2	
Initial state: $x=0 \wedge y=0$		
Allowed: $x=1 \wedge y=1$		



Just as for SB, R needs two DMBs or syncs to rule out the specified behaviour. 2+2W needs two DMBs on ARM but on POWER two lwsyncs suffices.



### 6.3 Observed Behaviour

		POWER			ARM			
	Kind	PowerG5	Power6	Power7	Tegra2	Tegra3	APQ8060	A5X
SB	Allow	102M/4.9G	1.9G/26G	11G/167G	430M/3.8G	1.0M/16M	16M/240M	8.1M/185M
SB+dmbs/syncs	Forbid	0/6.9G	0/40G	0/252G	0/24G	0/39G	0/26G	0/2.2G
SB+lwsyncs	Allow	7.0M/4.8G	10G/26G	1.0G/162G	—	—	—	—
IRIW	Allow	220k/2.6G	1.3M/13G	16M/83G	—	835/8.3M	—	—
IRIW+addrs	Allow	0/3.5G <sup>U</sup>	1.2M/14G	344k/107G	—	0/20G <sup>U</sup>	—	—
IRIW+dmbs/syncs	Forbid	0/3.5G	0/20G	0/126G	—	0/20G	—	—
IRIW+lwsyncs	Allow	0/3.6G <sup>U</sup>	568k/13G	429k/87G	—	—	—	—
RWC	Allow	883k/1.2G	7.4M/4.2G	118M/24G	—	90k/8.2M	—	—
RWC+dmbs/syncs	Forbid	0/2.3G	0/12G	0/55G	—	0/20G	—	—
S	Allow	250/2.3G	129k/8.3G	1.7M/14G	16M/3.8G	107k/16M	16k/550M	4.5M/185M
S+dmbs/syncs	Forbid	0/2.1G	0/14G	0/29G	0/24G	0/39G	0/26G	0/2.2G
S+lwsyncs	Forbid	0/2.1G	0/14G	0/29G	—	—	—	—
R	Allow	45M/1.9G	263M/7.3G	47M/4.5G	207M/3.8G	441k/16M	1.6M/240M	6.9M/185M
R+dmbs/syncs	Forbid	0/2.0G	0/13G	0/27G	0/24G	0/39G	0/26G	0/2.2G
R+lwsync+sync	Allow	0/2.3G <sup>U</sup>	0/17G <sup>U</sup>	0/33G <sup>U</sup>	—	—	—	—
2+2W	Allow	2.1M/6.3G	251M/33G	29G/894G	114M/4.4G	484k/31M	10k/580M	11M/365M
2+2W+dmbs/syncs	Forbid	0/6.3G	0/43G	0/943G	0/29G	0/44G	0/26G	0/2.5G
2+2W+lwsyncs	Forbid	0/6.2G	0/43G	0/911G	—	—	—	—

Tegra3 is the only four-hardware-thread implementation we currently have access to, so we show IRIW and RWC results only for that, and there are no ARM analogues of the lwsync tests.

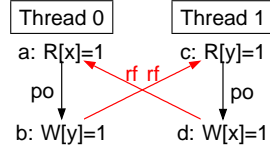
As one would hope, the forbidden behaviours are all non-observable. In some cases the allowed behaviours are not exhibited by particular implementations: just as for *WRC+addrs*, the *IRIW+addrs* test is not observable on POWER G5 or on ARM Tegra3, and *IRIW+lwsyncs* is also not observable on POWER G5 (together with the previous data, this suggests that POWER G5 and POWER 6 are incomparable: neither is strictly weaker or stronger than the other).

The *R+lwsync+sync* test is not observable on any of these POWER implementations, which is particularly interesting for the implementation of higher-level language models such as the C/C++11 model. As we explain elsewhere [BMO<sup>+</sup>12], an early proposal for an implementation of the C/C++11 concurrency primitives on POWER implicitly assumed that the *R+lwsync+sync* is forbidden, using an *lwsync* at a certain point in the implementation in a place that would be sound for the POWER implementations we have tested to date but that which would not be sound with respect to the architectural intent. The proposal has since been updated.

## 7 Load-Buffering (LB) Examples

Dual to store-buffering is the *load-buffering* (LB) example below, in which two threads first read from two shared locations respectively and then write to the other locations. The outcome in which the reads both read from the write of the other thread is architecturally allowed on ARM and POWER, and it is observable on current ARM processors; we have not observed it on POWER G5, POWER 6, or POWER 7.

LB		Pseudocode		ARM		POWER	
Thread 0	Thread 1	Thread 0	Thread 1	Thread 0	Thread 1	Thread 0	Thread 1
r1=x y=1	r2=y x=1	LDR R0,[R2] MOV R1,#1 STR R1,[R3]	LDR R0,[R3] MOV R1,#1 STR R1,[R2]			lwz r1,0(r2) li r3,1 stw r3,0(r4)	lwz r1,0(r2) li r3,1 stw r3,0(r4)
Initial state: x=0 ∧ y=0		Initial state: 0:R2=x ∧ 0:R3=y ∧ 1:R2=x ∧ 1:R3=y		Initial state: 0:R2=x ∧ 0:R3=y ∧ 1:R2=y ∧ 1:R4=x		Initial state: 0:r2=x ∧ 0:r4=y ∧ 1:r2=y ∧ 1:r4=x	
Allowed: r1=1 ∧ r2=1		Allowed: 0:R0=1 ∧ 1:R0=1		Allowed: 0:R0=1 ∧ 1:R0=1		Allowed: 0:r1=1 ∧ 1:r1=1	



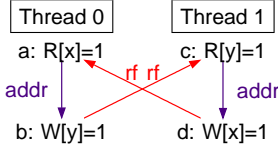
Test LB: Allowed

To forbid that outcome it suffices to add any read-to-write dependency, or a DMB, sync, or lwsync barrier, to both threads, as in the LB+adrs variant below:

LB+adrs	Pseudocode
Thread 0	Thread 1
r1=x	r2=y
*(&y+r1-r1)=1	*(&x+r2-r2)=1
Initial state: $x=0 \wedge y=0$	
Forbidden: $r1=1 \wedge r2=1$	

LB+adrs	ARM
Thread 0	Thread 1
LDR R0,[R3]	LDR R0,[R4]
EOR R1,R0,R0	EOR R1,R0,R0
MOV R2,#1	MOV R2,#1
STR R2,[R1,R4]	STR R2,[R1,R3]
Initial state: $0:R3=x \wedge 0:R4=y \wedge 1:R3=x \wedge 1:R4=y$	
Forbidden: $0:R0=1 \wedge 1:R0=1$	

LB+adrs	POWER
Thread 0	Thread 1
lwz r1,0(r2)	lwz r1,0(r2)
xor r3,r1,r1	xor r3,r1,r1
li r4,1	li r4,1
stwx r4,r3,r5	stwx r4,r3,r5
Initial state: $0:r2=x \wedge 0:r5=y \wedge 1:r2=y \wedge 1:r5=x$	
Forbidden: $0:r1=1 \wedge 1:r1=1$	



Test LB+adrs: Forbidden

or in the LB+datas and LB+ctrls variants below:

LB+datas	Pseudocode
Thread 0	Thread 1
r1=x	r2=y
y=r1	x=r2
Initial state: $x=0 \wedge y=0$	
Forbidden: $r1=m \wedge r2=n$ for any $m, n \neq 0$	

LB+ctrls	Pseudocode
Thread 0	Thread 1
r1=x	r2=y
if (r1==1) y=1	if (r2==1) x=1
Initial state: $x=0 \wedge y=0$	
Forbidden: $r1=m \wedge r2=n$ for any $m, n \neq 0$	

All of these ensure that both writes cannot be committed (and thence propagated and become visible to the other thread) until their program-order-preceding reads have been satisfied and committed.

## 7.1 Observed Behaviour

		POWER			ARM			
	Kind	PowerG5	Power6	Power7	Tegra2	Tegra3	APQ8060	A5X
LB	Allow	0/7.4G <sup>u</sup>	0/43G <sup>u</sup>	0/258G <sup>u</sup>	1.5M/3.9G	124k/16M	58/1.6G	1.3M/185M
LB+adrs	Forbid	0/6.9G	0/40G	0/216G	0/24G	0/39G	0/26G	0/2.2G
LB+datas	Forbid	0/6.9G	0/40G	0/252G	0/16G	0/23G	0/18G	0/2.2G
LB+ctrls	Forbid	0/4.5G	0/16G	0/88G	0/8.1G	0/7.5G	0/1.6G	0/2.2G

Here we see another case where some implementations are stronger than the architectural intent: these POWER implementations do not exhibit LB, while the ARM implementations do. This suggests that these POWER implementations do not commit a write until program-order-previous reads have bound their values and committed (as we saw in for S+sync+po in Section 4.7), while in the ARM case a program-order-previous read-request may still be outstanding when a write is committed.

Comfortingly, neither architecture permits values to be synthesised out of thin air, as LB+datas illustrates; this has been a key concern in the design of high-level-language models for Java and C/C++11.

## 8 Coherence (CoRR1, CoWW, CoRW1, CoWR, CoRW)

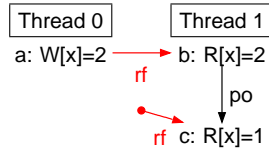
As we have seen, ARM and POWER are far from sequentially consistent: one cannot assume that in any execution of a multithreaded program there is some sequential order of all the read and write operations of the threads, consistent with the program order of each thread, in which each read reads the value of the most recent write. However, if one restricts attention to just the reads and writes of a *single location* in an ARM or POWER execution, it is true that all threads must share a consistent view of those reads and writes. Effectively, in any execution, for each location, there is a single linear order of all writes to that location which must be respected by all threads. This property is known as *coherence*; we explore and make precise what we mean by ‘respected by all threads’ with the examples below.

Our first test, CoRR1, is shown below in three forms, as usual: a readable C-like pseudocode on the left, using thread-local variables *r1* and *r2* and a shared variable *x* (initially 1), and the definitive ARM and POWER versions (the versions we test), in assembly language on the right. If one reached the specified final state, with *r1*=2 and *r2*=1, then the second Thread 1 read of *x* would have to have been from the initial state despite the first Thread 1 read of *x* seeing the value 2 from Thread 0’s write of *x*=2. That write must be coherence-after the initial state, so this would be a violation of coherence, and that execution is forbidden in both ARM and POWER. As usual, several other executions of the same code are allowed: both Thread 1 reads could read 1, or both could read 2, or the first could read 1 and the second 2. Those are just sequentially consistent interleavings of the code, not exposing any of the relaxed aspects of the architectures.

CoRR1 Pseudocode	
Thread 0	Thread 1
<i>x</i> =2	<i>r1</i> = <i>x</i> <i>r2</i> = <i>x</i>
Initial state: <i>x</i> =1	
Forbidden: 1: <i>r1</i> =2 $\wedge$ 1: <i>r2</i> =1	

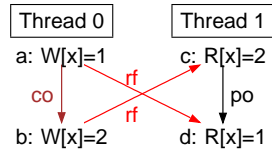
CoRR1 ARM	
Thread 0	Thread 1
STR R2,[R5]	LDR R1,[R5] LDR R2,[R5]
Initial state: 0: <i>R2</i> =2 $\wedge$ 0: <i>R5</i> = <i>x</i> $\wedge$ 1: <i>R5</i> = <i>x</i> $\wedge$ [ <i>x</i> ]=1	
Forbidden: 1: <i>R1</i> =2 $\wedge$ 1: <i>R2</i> =1	

CoRR1 POWER	
Thread 0	Thread 1
stw r2,0(r5)	lwz r1,0(r5) lwz r2,0(r5)
Initial state: 0: <i>r2</i> =2 $\wedge$ 0: <i>r5</i> = <i>x</i> $\wedge$ 1: <i>r5</i> = <i>x</i> $\wedge$ [ <i>x</i> ]=1	
Forbidden: 1: <i>r1</i> =2 $\wedge$ 1: <i>r2</i> =1	

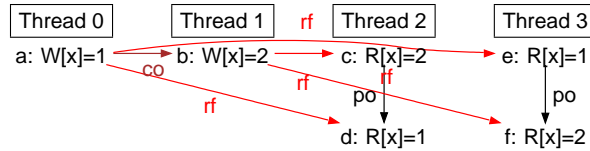


Test CoRR1: Forbidden

Two minor variations of the test can be instructive. In CoRR0 on the left below, the above initial-state write of *x* is done by Thread 0, and the forbidden outcome is simply that Thread 1 sees the two program-ordered writes by Thread 0 in the opposite order. In CoRR2 on the right below (like IRIW but with a single shared location, not two), the two writes of *x*=1 and *x*=2 are by different threads (different both from each other and from the reading threads), so they are not a priori ordered either way, but it is still true that the two reading threads have to see them in the same order as each other: it is forbidden for Thread 2 to see 2 then 1 in the same execution as Thread 3 sees 1 then 2.



Test CoRR0: Forbidden

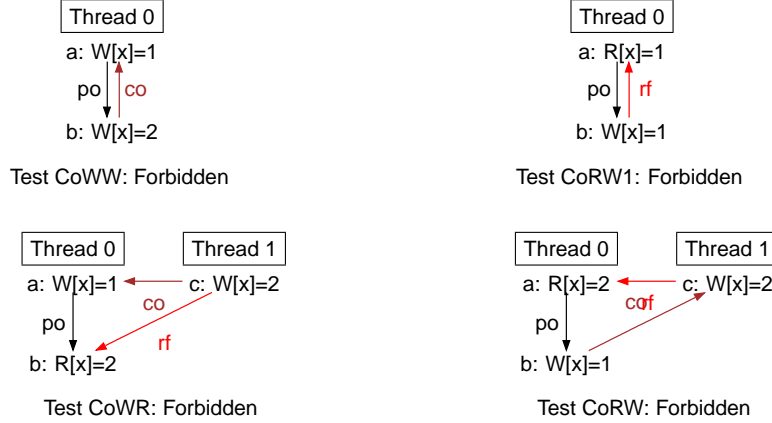


Test CoRR2: Forbidden

We express coherence with **co** edges between writes to the same address; in a complete execution, for each address those edges must form a total linear order. All writes are implicitly coherence-after the initial-state write to their address. The diagrams above illustrate executions in which write *a* is coherence-before write *b*.

The coherence order for a location can be observed experimentally in two ways. In simple cases where there are at most two writes to any location, as here, one can read the final state (after all threads have completed, with suitable barriers). In general one can have an additional thread (for each location) that does a sequence of reads, seeing each successive value in order. For example, a **co** edge from a write *a* of *x*=*v*<sub>1</sub> to a write *b* of *x*=*v*<sub>2</sub> means that no thread should observe *x* as taking value *v*<sub>2</sub> *then* value *v*<sub>1</sub>.

Test CoRR1 above showed that a pair of reads by a thread cannot read contrary to the coherence order, but there are several other cases that need to be covered to ensure that the coherence order is respected by all threads. Test CoWW below shows that the coherence order must respect program order for a pair of writes by a single thread. Test CoRW1 shows that a read cannot read from a write that program-order follows it. Test CoWR shows that a read cannot read from a write that is coherence-hidden by another write that precedes the read on its own thread. Test CoRW shows that a write cannot coherence-order-precede a write that a program-order-preceding read read from.

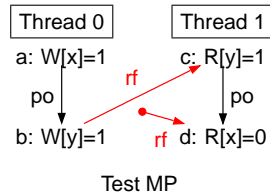


## 9 Periodic Tables of Litmus Tests

After seeing all the litmus tests that we have used to illustrate various relaxed-memory phenomena, one might ask whether they are *complete* in any sense. We now show that they can be treated systematically, organising them into “periodic tables” of families of tests with similar behaviour, and giving a sense in which this covers all “small” tests of some particular kinds. The tables do not include all interesting tests; we return to some others in later sections.

### 9.1 Litmus Test Families

First, we define a *family* of litmus tests to be the common shape of a group of tests, as specified by the read and write events, with the events of each thread related by program-order po edges, the write events to each location related by coherence co edges, and writes related to any reads that read from them by reads-from rf edges. For example, the MP test:



also defines a family of related tests, all of the same shape, obtained by replacing the program-order edges by dependencies or barriers. The terminology we have been using already suggests this, for example with MP+dmb+addr denoting the MP variation with a dmb barrier on Thread 0 and an address dependency (perhaps artificial) on Thread 1.

### 9.2 Minimal Strengthenings for a Family

For any family of tests, one can ask what are the minimal strengthenings of its program-order (po) edges required to forbid the specified execution. To make this precise, recall that the read-to-read dependencies that prevent local reordering are address and control-isb (ARM) or control-isync (POWER) dependencies (a control dependency to a read does not prevent the read being satisfied speculatively, there cannot be a data dependency to a read, and a lone isb or isync has no effect in this context), while the read-to-write dependencies that prevent local reordering are address, data, control, or control-isb/isync dependencies. We define notation:

$$\begin{aligned} \text{RRdep} &::= \text{addr} \mid \text{ctrlisb/ctrlisync} \\ \text{RWdep} &::= \text{addr} \mid \text{data} \mid \text{ctrl} \mid \text{ctrlisb/ctrlisync} \end{aligned}$$

To a first approximation, the different kinds of `RRdep` and `RWdep` dependencies behave similarly to each other (but see Section 10.5 for a subtle exception). The ARM `dmb` barrier and the POWER `lwsync` and `sync` barriers are both stronger than those dependencies, giving an order:

$$\text{po} < \{\text{RRdep}, \text{RWdep}\} < \text{lwsync} < \text{dmb/sync}$$

In other words, it should always be safe (giving the same or fewer allowed behaviours) to replace a plain program-order edge by an `RRdep` or `RWdep` dependency, or to replace one of those by an `lwsync` barrier, or to replace any of those by a `dmb` or `sync` barrier. As we saw before, a read-to-read control dependency has no force, and (for normal memory accesses) nor does an `isb/isync` without an associated control dependency, so replacing a program-order edge by one of those should have no effect.

Now consider all the POWER MP variations obtained from the basic test shape of the family by replacing its program-order (po) edges by `isync`, `addr`, `ctrlisync`, `lwsync`, or `sync` (ARM would be similar except without `lwsync` and with `dmb` in place of `sync`). The diagram in Fig. 1 shows each variation, in green if it forbids the undesirable outcome, or in red if that outcome is permitted; the arrows show the above order. Tests that have ‘similar’ dependencies and therefore should behave similarly are grouped in blue boxes. The minimal green tests are `MP+lwsync+addr` and `MP+lwsync+ctrlisync`, showing that on POWER to rule out the undesirable behaviour of MP one needs at least an `lwsync` on Thread 0 and an address or control-isync dependency on Thread 1. All tests above those two are also green, showing that in this case our ordering is meaningful.

### 9.3 4-edge 2-thread Tests and RF-Extensions

We now look at our set of test families more systematically, giving the minimal strengthenings for each, as shown in the table below. We have mentioned several families so far, of which MP, S, SB, R, 2+2W, and LB all have two threads, two shared locations, and two reads or writes on each thread. Additionally, we have seen a few three- or four-thread variations of those: `RWC`, `WRC`, `ISA2`, and `IRIW`. We have also seen several coherence tests, but those are of a slightly different character; their specified executions are forbidden without needing any dependencies or barriers, so there is less interest in exploring variations.

Looking at the table, in the left column we see those two-thread tests, grouped by the number of reads-from (rf) edges they have. In the first block, MP and S are similar: in S the MP read `d` from the initial state (coherence-before the write `a` to `x`) is replaced by a write `d` that is coherence-before `a`. They are similar also in what has to be done to prevent the undesirable outcome: MP needs at least `lwsync/dmb` and a read-to-read dependency, while S needs at least `lwsync/dmb` and a read-to-write dependency.

Next we have the three tests with no rf edges: SB and its variations R and 2+2W, which replace one or both (respectively) of the final initial-state reads by writes to coherence predecessors of the writes to `x`. In contrast to the first group, SB and R need two `syncs` or two `dmbs`; `lwsync` does not suffice here. However, `lwsync` does suffice for the last variation 2+2W, of four writes.

Finally there is the LB family, with two rf edges. Here simple dependencies suffice.

Moving to the right, the second and third columns are the tests obtainable from the left column by “pulling out” one or two initial writes to new threads. There are several exotic variations here, most of which are not (as far as we know) natural use-cases, but they include the `WRC`, `IRIW`, and `RWC` families discussed in the literature [BA08]. Notably, they need just the same strengthenings as their base tests in the first column: `lwsync/dmb` and a dependency in the first block, `syncs/dmbs` for the extensions of SB and R, and `lwsyncs/dmbs` for the extensions of 2+2W; this is the cumulativity properties of the barriers at work.

The MP family can usefully be varied in another dimension by considering a *sequence* of dependency or other edges between the reads, shown schematically on the diagram as the PPO (preserved program order) series; we come back to some of these in Section 10.

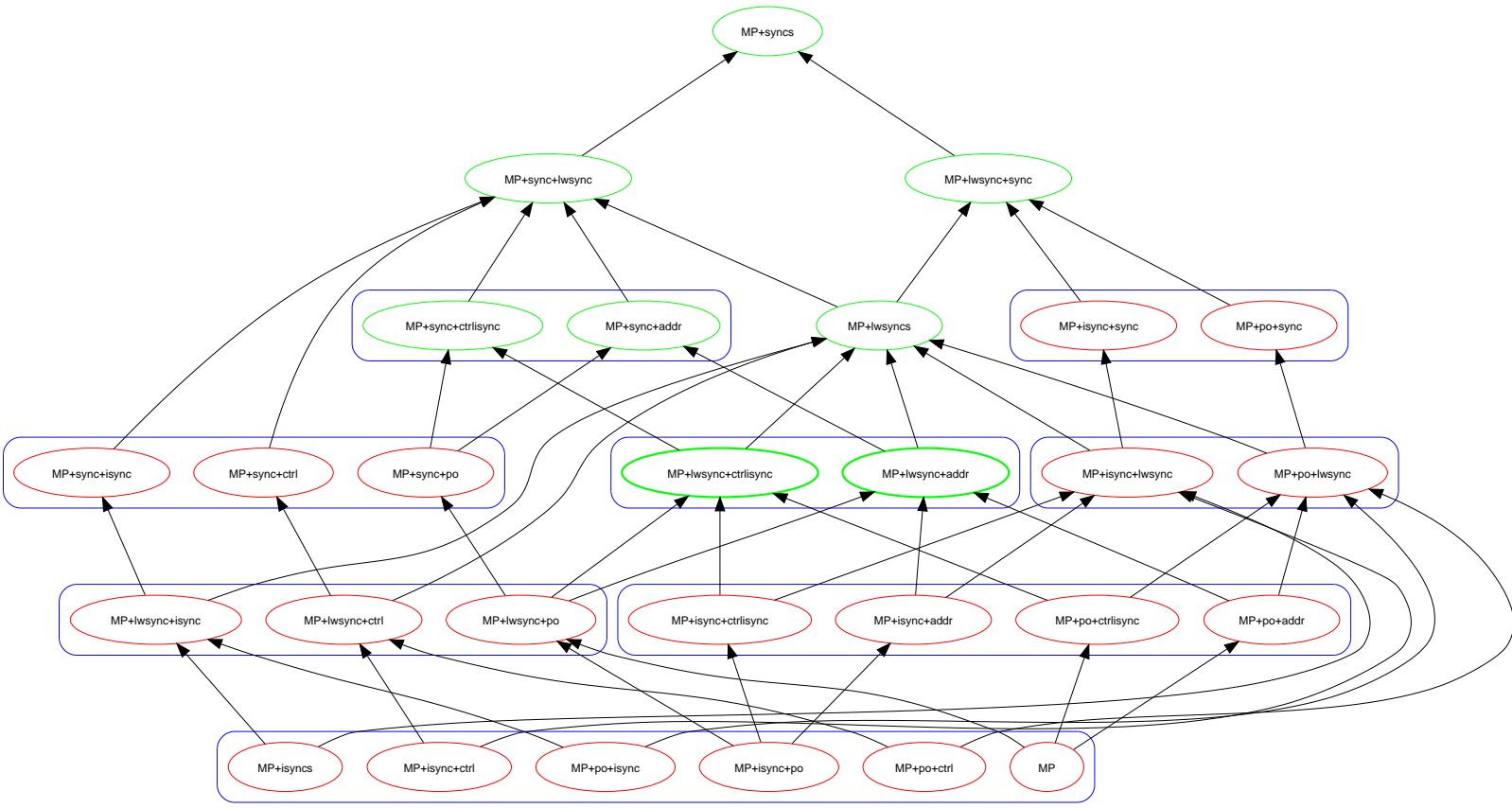


Figure 1: Variations of the MP test

# 4-edge 2-thread tests

# 5-edge extensions along one rf edge

<p><b>One rf</b></p> <p><b>MP: rf,fr</b> needs lwsync+RRdep or dmb+RRdep</p> <p>Test MP</p>	<p><b>Two rf</b></p> <p><b>WRC: rf,rf,fr</b> needs lwsync+RRdep or dmb+RRdep</p> <p>Test WRC</p>	<p><b>Preserved read-read program order</b></p> <p><b>PPO: barrier,rf,intra-thread*,fr</b></p> <p>PPO variations</p>
<p><b>No rf</b></p> <p><b>SB: fr,fr</b> needs sync+sync or dmb+dmb</p> <p>Test SB</p>	<p><b>One rf</b></p> <p><b>RWC: rf,fr,fr</b> needs sync+sync or dmb+dmb</p> <p>Test RWC</p>	<p><b>6-edge extensions along two rf edges</b></p> <p><b>IRIW: rf,fr,rf,fr</b> needs sync+sync or dmb+dmb</p> <p>Test IRIW</p>
<p><b>R: co,fr</b> needs sync+sync or dmb+dmb</p> <p>Test R</p>	<p><b>WRW+WR: rf,co,fr</b> needs sync+sync or dmb+dmb</p> <p>Test WRW+WR</p>	<p><b>IRRWIW: rf,fr,rf,co</b> needs sync+sync or dmb+dmb</p> <p>Test IRRWIW</p>
<p><b>2+2W: co,co</b> needs lwsync+lwsync or dmb+dmb</p> <p>Test 2+2W</p>	<p><b>WRW+2W: rf,co,co</b> needs lwsync+lwsync or dmb+dmb</p> <p>Test WRW+2W</p>	<p><b>IRWIW: rf,co,rf,co</b> needs lwsync+lwsync or dmb+dmb</p> <p>Test IRWIW</p>
<p><b>Two rf</b></p> <p><b>LB: rf,rf</b> needs RWdep+RWdep</p> <p>Test LB</p>	<p><b>Key</b></p> <p>Edges:</p> <ul style="list-style-type: none"> <li>po program order</li> <li>rf reads-from</li> <li>co coherence order</li> <li>fr from-reads: read from coherence predecessor, or from the initial state</li> </ul> <p>Read-read and read-write dependencies:</p> <p>RRdep ::= addr   ctrl-isb/isync</p> <p>RWdep ::= addr   data   ctrl   ctrl-isb/isync</p> <p>po &lt; {RRdep,RWdep} &lt; lwsync &lt; dmb/sync</p>	



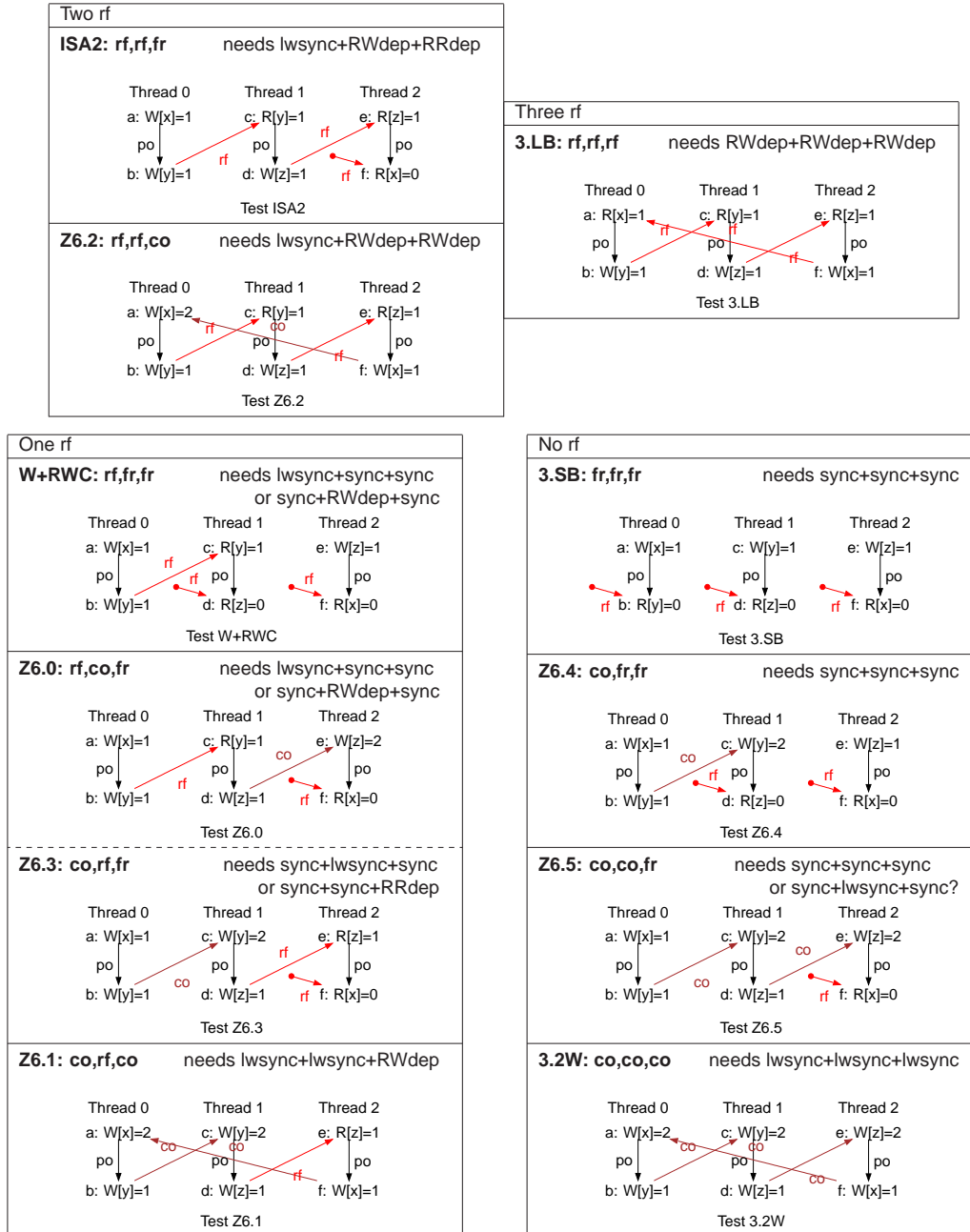
## 9.4 6-edge 3-thread Tests

Moving to tests with three threads, three shared locations, and two reads or writes in each thread, the tables below show our 11 families. Of interest here are:

- ISA2: the generalisation of message-passing to three threads we saw in Section 5, which needs a barrier only on the first thread;
- 3.SB, 3.2W, and 3.LB, the generalisations of SB, 2+2W, and LB to three threads, which need just the same as the two-thread variants; and
- Z6.3, which shows the lack of transitivity of coherence and lwsync barriers on POWER; we return to this in Section 11.

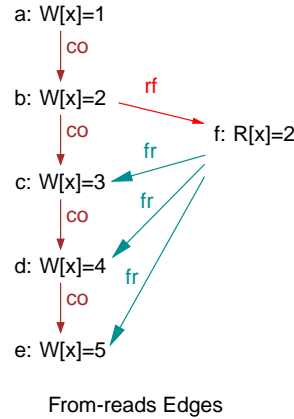
We expect that ISA2 is common in practice, but would be glad to hear of any use cases of the other families.

6-edge 3-thread tests



## 9.5 Test Family Coverage

The obvious question is whether those families give a set of tests that is complete in any sense? One could try to answer it by enumerating all multithreaded assembly programs up to some size (e.g. some bound on the number of threads and the number of instructions per thread), but that quickly gives an intractable number of tests, very many of which would be uninformative. A better approach would be to enumerate all families up to a certain size (e.g. up to four threads and some number of read and write actions per thread). However, simply enumerating families still includes many uninformative tests, where the execution in question is allowed in a sequentially consistent model. Instead, therefore, we consider the families generated by the *critical cycles* of [SS88, Alg10, AMSS10]. To do this, we first need the concept of a *from-reads* edge, introduced (as reads-before edges) by Ahamad *et al.* [ABJ<sup>+</sup>93] and (as some edges in their access graphs) by Landin *et al.* [LHH91]. Given a candidate execution, with its reads-from relation (from each write to all the reads that read-from that write) and its coherence relation (the union of some linear order over the writes to each address), we define its from-reads relation to have an edge from each read to all the coherence-successors of the write it reads from (or all the writes to the same address, if it reads from the initial state). For example, consider the candidate execution below, with 5 writes (perhaps by various threads), of 1, . . . , 5, to x, in that coherence order, and with a read that reads from the write b. The coherence-successors of b are writes c, d, and e, so we construct a from-reads edge from b to each of those.



We can replace the reads-from edges from the initial state (to some read) by the from-reads edges from that read to the write(s) to the same address, without any loss of information. For example, for MP and SB, we have:

	drawn with reads-from (rf) from initial state	drawn with from-reads (fr)
MP	<p>Thread 0      Thread 1</p> <p>a: W[x]=1      c: R[y]=1</p> <p>po ↓      rf ↓      po ↓</p> <p>b: W[y]=1      d: R[x]=0</p> <p>Test MP: Allowed</p>	<p>Thread 0      Thread 1</p> <p>a: W[x]=1      c: R[y]=1</p> <p>po ↓      fr rf ↓      po ↓</p> <p>b: W[y]=1      d: R[x]=0</p> <p>Test MP: Allowed</p>
SB	<p>Thread 0      Thread 1</p> <p>a: W[x]=1      c: W[y]=1</p> <p>po ↓      po ↓</p> <p>rf b: R[y]=0      rf d: R[x]=0</p> <p>Test SB: Allowed</p>	<p>Thread 0      Thread 1</p> <p>a: W[x]=1      c: W[y]=1</p> <p>po ↓      fr fr ↓      po ↓</p> <p>b: R[y]=1      d: R[x]=0</p> <p>Test SB: Allowed</p>

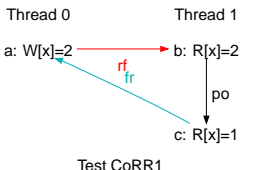
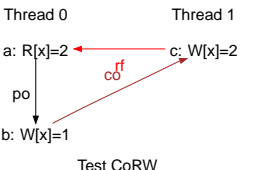
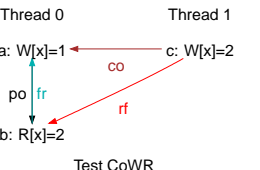
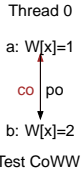
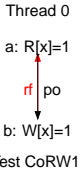
Note that the diagrams on the right have cycles in the union of rf, co, fr, and po, and indeed such cycles are exactly the violations of sequential consistency, as shown by [SS88, Alg10, AMSS10] (see also Theorem 5 in [LHH91]), so by enumerating such cycles we can produce exactly the test families of interest — the potential non-SC executions.

The families presented up to now cover all critical cycles up to six edges, where critical cycles are defined as: (1) **po** edges alternate with basic communication sequences defined as **rf**, **fr**, **co**, **co** followed by **rf**, or **fr** followed by **rf**; (2) communication edges **rf**, **fr** and **co** are between distinct threads; (3) **po** edges are between distinct locations; and (4) no thread holds more than two events, and when a thread holds two events those are related by a **po** edge. Following [SS88, Alg10, AMSS10], any non-SC axiomatic candidate execution (*i.e.* a set of events with a cyclic union of relations **po**, **rf**, **co** and **fr**), includes at least one critical cycle (or violates coherence, see below). Hence, critical cycles describe violations of SC (up to coherence), and our coverage is of such violations up to six edges.

The `diy` tool of Alglave and Maranget (<http://diy.inria.fr>) lets one generate litmus tests from particular cycles, and also lets one enumerate families (and the members of a family) by describing sets of cycles; most of the tests we show were generated in this fashion.

## 9.6 Coherence

Reducing violations of SC to critical cycles assumes a coherent architecture, which POWER and ARM architectures are. Coherence is related to the very existence of a shared memory: namely there is observably only one instance of a given memory location, or that writes to a given location are linearly ordered (that is, **co** exists), with all observations made in the system compatible with that linear ordering. These conditions can be summarised as “per-location sequential consistency” [CLS03]. More formally, one can show that forbidding the five tests we gave is equivalent to the acyclicity of the union of **rf**, **co**, **fr**, and the **po** edges restricted to events to the same location, which is a precise sense in which coherence is per-location sequential consistency; this is the *uniproc* condition of [Alg10, AMSS10].

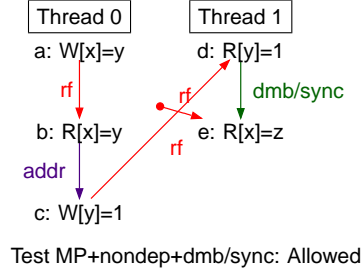
Coherence tests		
<b>CoRR1: rf,po,fr</b> forbidden  <p>Test CoRR1</p>	<b>CoRW: rf,po,co</b> forbidden  <p>Test CoRW</p>	<b>CoWR: co,fr</b> forbidden  <p>Test CoWR</p>
<b>CoWW: po,co</b> forbidden  <p>Test CoWW</p>	<b>CoRW1: po,rf</b> forbidden  <p>Test CoRW1</p>	

## 10 Preserved Program Order (PPO) Variations

We now explore some further variations of the message-passing (MP) test that illustrate some more subtle points about the circumstances in which the architectures do (and, more importantly, do not) respect program order.

### 10.1 No Write-to-write Dependency from **rf;addr** (MP+nondep+dmb/sync)

In Section 4 we saw dependencies from reads to reads and writes, but no ‘dependency’ from a write. One might think that if one writes a value to a location  $x$ , then reads it back on the same thread, then has a data or address dependency to a write of a different location  $y$ , then those two writes would be held in order as far as any other thread is concerned. That is *not* the case, as the **MP+nondep+addr** example below shows: even though the two writes might have to commit in program order, in the absence of any barriers (and because they are to different addresses) they can still propagate to other threads in arbitrary orders.



MP+nondep+sync	Pseudocode
Thread 0	Thread 1
x=&y r0 = x *r0 = 1	r1=y dmb/sync r2=x
Initial state: x=&z ∧ y=0	
Allowed: 0:r0=&y ∧ 1:r1=1 ∧ 1:r2=&z	

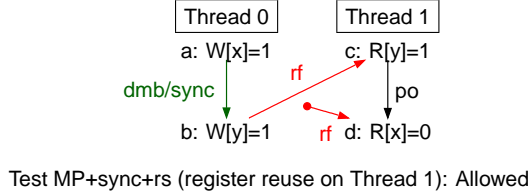
This is observable on POWER and ARM:

	Kind	POWER			ARM			
		PowerG5	Power6	Power7	Tegra2	Tegra3	APQ8060	A5X
MP+nondep+dmb/sync	Allow	12/3.7G	157k/20G	2.9G/860G	2.2M/3.8G	1.1k/90M	6.9k/640M	9.1k/185M

## 10.2 No Ordering from Register Shadowing (MP+dmb/sync+rs, LB+rs)

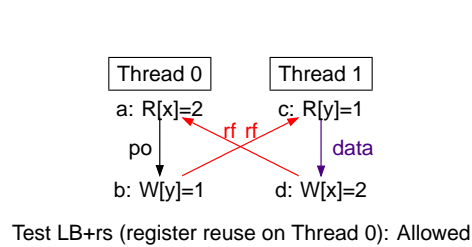
Another conceivable source of ordering which is not respected by the architectures is re-use of the same processor register: the hardware implementations typically have more ‘shadow’ registers than the architected general-purpose registers that can be referred to by machine-code instructions, and the allocation of hardware registers to architected registers is done on-the-fly. This register renaming is observable to the programmer, as the following two examples show.

First, we have a variant of MP that exhibits observable register shadowing: the two uses of r3 on Thread 1 do not prevent the second read being satisfied out-of-order, if the reads are into shadow registers (specifically, the first two uses of r3 on Thread 1 might involve one shadow register while the third usage might involve another). The reuse of a register is not represented in our diagrams, so we note it in the caption; the details can only be seen in the pseudocode or assembly versions of the test.



MP+dmb/sync+rs	Pseudocode
Thread 0	Thread 1
x=1 dmb/sync y=1	r3=y r1=r3 r3 = x
Allowed: 1:r1=1 ∧ 1:r3=0	

Along the same lines, we have a variant of LB (taken from Adir et al. [AAS03]) in which the reuse of register r1 on Thread 0 does not keep the read of x and the write of y in order.



LB+rs	Pseudocode
Thread 0	Thread 1
r1=x r2=r1 r1=1 y=r1	r3=y r3=r3+1 x=r3
Allowed: 0:r1=1 ∧ 0:r2=2 ∧ 1:r3=2 ∧ y=1 ∧ x=2	

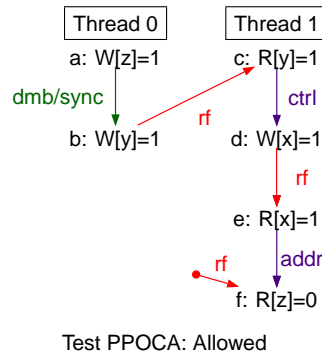
In current implementations, the MP+sync+rs behaviour is observable on both ARM and POWER, while the LB+rs behaviour is only observable on ARM, as the table below shows. The latter is simply because the base LB behaviour is only observable on ARM (it appears that current POWER implementations do not commit writes in the presence of outstanding uncommitted reads). Nonetheless, both behaviours are architecturally permitted in both architectures.

	Kind	POWER			ARM			
		PowerG5	Power6	Power7	Tegra2	Tegra3	APQ8060	A5X
LB+rs	Allow	0/3.7G <sup>u</sup>	0/26G <sup>u</sup>	0/898G <sup>u</sup>	101k/3.9G	6.4k/89M	0/26G <sup>u</sup>	60k/201M
MP+dmb/sync+rs	Allow	1.8k/3.0G	0/41G <sup>u</sup>	29M/146G	9.0M/3.9G	1.2k/19M	11k/753M	549k/201M

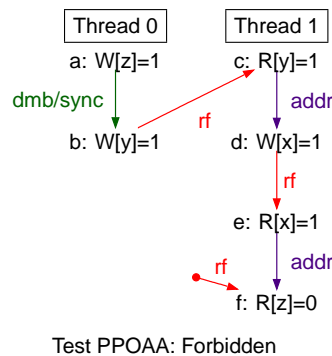
### 10.3 Preserved Program Order, Speculation, and Write Forwarding (PPOCA and PPOAA)

The POWER architecture states that that writes are not performed speculatively, but we see here that, while speculative writes are never visible to other threads, they can be forwarded locally to program-order-later reads on the same thread; this forwarding is observable to the programmer.

In the PPOCA variant of MP below, *f* is address-dependent on *e*, which reads from the write *d*, which is control-dependent on *c*. One might expect that chain to prevent read *f* binding its value before *c* does, but in fact in some implementations *f* can bind out-of-order, as shown — the write *d* can be forwarded directly to *e* within the thread, before the write is committed to the storage subsystem, while *d*, *e*, and *f* are all still speculative (before the branch of the control dependency on *c* is resolved).



Replacing the control dependency with a data dependency (test PPOAA, below) removes that possibility, forbidding the given result on current hardware, as far as our experimental results show, and in our model.

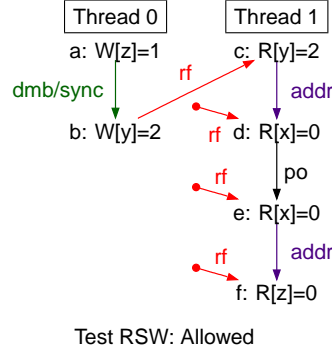


		POWER			ARM			
	Kind	PowerG5	Power6	Power7	Tegra2	Tegra3	APQ8060	A5X
PPOCA	Allow	1.1k/3.4G	0/49G <sup>U</sup>	175k/157G	0/24G <sup>U</sup>	0/39G <sup>U</sup>	233/743M	0/2.2G <sup>U</sup>
PPOAA	Forbid	0/3.4G	0/46G	0/209G	0/24G	0/39G	0/26G	0/2.2G

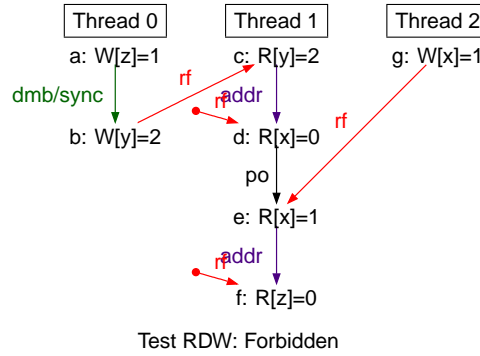
### 10.4 Aggressively Out-of-order Reads (RSW and RDW)

Given the discussion of coherence in Section 8, one might expect two reads from the same address to have to be satisfied in program order. That is usually the case, but in the special case where the two reads happen to read from the same write (not merely that they read the same value), it is not.

In the reads-from-same-writes (RSW) variant of MP below, the two reads of *x*, *d* and *e*, happen to read from the same write (the initial state). In this case, despite the fact that *d* and *e* are reading from the same address, the *e*/*f* pair can satisfy their reads out-of-order, before the *c*/*d* pair, permitting the outcome shown. The address of *e* is known, so it can be satisfied early, while the address of *d* is not known until its address dependency on *c* is resolved.



In contrast, in an execution of the same code in which **d** and **e** read from different writes to **x** (test **RDW** below), with another write to **x** by another thread, that is forbidden — in the model, the commit of the first read (**d**) would force a restart of the second (**e**), together with its dependencies (including **f**), if **e** had initially read from a different write to **d**. In actual implementations the restart might be earlier, when an invalidate is processed, but will have the same observable effect.

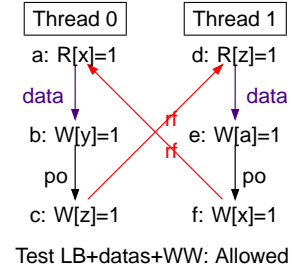
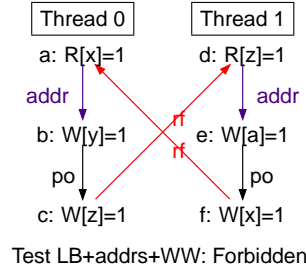


		POWER			ARM			
	Kind	PowerG5	Power6	Power7	Tegra2	Tegra3	APQ8060	A5X
RSW	Allow	1.3k/3.4G	0/33G <sup>u</sup>	33M/144G	0/24G <sup>u</sup>	0/39G <sup>u</sup>	0/26G <sup>u</sup>	0/2.2G <sup>u</sup>
RDW	Forbid	0/1.7G	0/17G	0/125G	—	0/20G	—	—
RDWI	Allow	5.2k/3.0G	0/12G <sup>u</sup>	1.3M/43G	0/24G <sup>u</sup>	0/39G <sup>u</sup>	0/26G <sup>u</sup>	0/2.2G <sup>u</sup>

Test **RDWI** is a two-thread variant of **RDW** in which the write **g:W[x]=1** is on Thread 1, between **d** and **e**. One notices that **RSW** (and **RDWI**) stands unobserved on **ARM**, while observed on **POWER**.

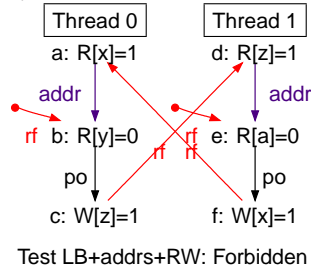
## 10.5 Might-access-same-address

In the examples we have seen so far, address and data dependencies to a write have the same effect, preventing the write being visible to other threads before the instructions that provide the dependent value are committed. However, there can be a second-order effect that distinguishes between them: the fact that there is an address dependency to a write might mean that another program-order-later write cannot proceed until it is known that the first write is not to the same address, whereas the existence of a data dependency to a write has no such effect on program-order-later writes that are statically known to be to different addresses. This can be seen in the two variations of the **LB** test below. In both, there are extra writes, to two different addresses, inserted in the middle of each thread. On the left, those writes are address-dependent on the first reads, and so before those reads are satisfied, the middle writes are not known to be to different addresses to the last writes on each thread. On the right, the middle writes are merely data-dependent on the first reads, so they are statically known to be to different addresses to the last writes on each thread.



The first is not observable on any of the ARM implementations we have tested (Tegra 2, Tegra 3, APQ8060, A5X), while the second is observable on all of them except APQ8060. For POWER, recall that we have not observed the basic LB behaviour on any current implementation, and these variations are also, unsurprisingly, not observable.

Replacing the intervening writes by reads gives the test below, which has the same observable behaviour as LB+addr+WW.



The operational model we gave in PLDI 2011 [SSA<sup>+</sup>11] matches these observations precisely, giving the ‘forbidden’ or ‘allowed’ status as shown for each test. But whether an architectural model should allow or forbid the two it forbids may be debatable.

		POWER			ARM			
	Kind	PowerG5	Power6	Power7	Tegra2	Tegra3	APQ8060	A5X
LB+addr+WW	Forbid	0/30G	0/8.7G	0/208G	0/16G	0/23G	0/18G	0/2.1G
LB+datas+WW	Allow	0/30G <sup>u</sup>	0/9.2G <sup>u</sup>	0/208G <sup>u</sup>	15k/6.3G	224/854M	0/18G <sup>u</sup>	23/1.9G
LB+addr+RW	Forbid	0/3.6G	0/6.0G	0/128G	0/13G	0/23G	0/16G	—

## 10.6 Observable Read-request Buffering

Our final example is a case where our PLDI 2011 [SSA<sup>+</sup>11] model, there tested against POWER, is not sound with respect to behaviour observable on ARM (specifically, the APQ8060), and that behaviour is architecturally intended to be permitted for ARM.

The test is another variation of message passing (MP), with a strong dmb barrier on the writing side. On the reading side, the read of y is followed by a write (necessarily of a coherence-later value) back to y, followed by a read of that value, and finally a control-isb dependency to the ultimate read of x.

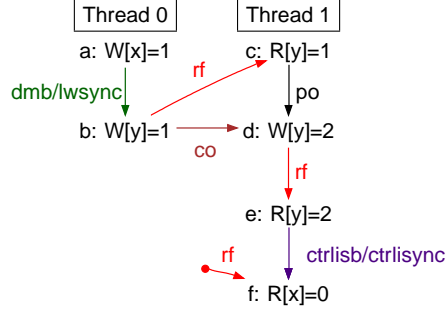
The control-isb means that the read f of x cannot be satisfied until the read e of y=2 is committed, and that read cannot be committed before the write d that it reads from is committed.

In our PLDI 2011 model, to maintain coherence, that write d cannot be committed before program-order-previous reads and writes that might be to the same address are committed, which blocks the whole chain, ensuring that f is satisfied after c.

To see how legitimate hardware might be doing the contrary, suppose that the read request for c is buffered. It can proceed with the write d to the same address, letting that write be read from and e and f continue, so long as the hardware can guarantee that the read request will eventually be satisfied by a coherence predecessor of the write d. If read requests and writes are buffered in the same FIFO-per-location buffer, that will happen naturally.

This can be accommodated in a variant of the PLDI 2011 model by allowing writes to commit in slightly more liberal circumstances.



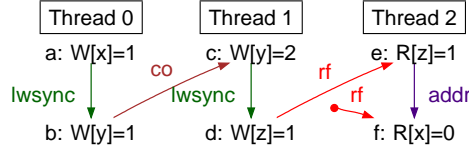


Test MP+dmb/lwsync+fri-rfi-ctrlisb/ctrlisync: Forbidden

		POWER			ARM			
	Kind	PowerG5	Power6	Power7	Tegra2	Tegra3	APQ8060	A5X
MP+dmb/lwsync+fri-rfi-ctrlisb/isync	Allow	0/26G <sup>U</sup>	0/6.6G <sup>U</sup>	0/80G <sup>U</sup>	0/26G <sup>U</sup>	0/39G <sup>U</sup>	7/1.6G	0/1.9G <sup>U</sup>

## 11 Coherence and lwsync (Z6.3+lwsync+lwsync+addr)

This POWER example (known as blw-w-006 in our earlier work) shows that one cannot assume that the transitive closure of lwsync and coherence edges guarantees ordering of write pairs, which is a challenge for over-simplified models. In our abstract machine, the fact that the storage subsystem commits to **b** being before **c** in the coherence order has no effect on the order in which writes **a** and **d** propagate to Thread 2. Thread 1 does not read from either Thread 0 write, so they need not be sent to Thread 1, so no cumulativity is in play. In other words, coherence edges do not bring writes into the “Group A” of a POWER barrier.



Test Z6.3+lwsync+lwsync+addr: Allowed

In some implementations, and in our model, replacing both lwsyncs by syncs forbids this behaviour. In the model, it would require a cycle in abstract-machine execution time, from the point at which **a** propagates to its last thread, to the Thread 0 sync ack, to the **b** write accept, to **c** propagating to Thread 0, to **c** propagating to its last thread, to the Thread 1 sync ack, to the **d** write accept, to **d** propagating to Thread 2, to **e** being satisfied, to **f** being satisfied, to **a** propagating to Thread 2, to **a** propagating to its last thread.

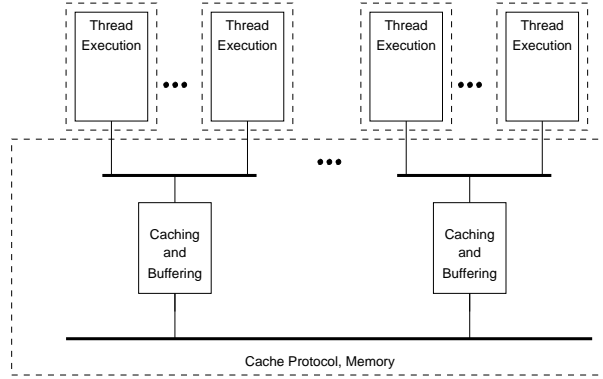
ARM does not have an analogue of lwsync, so there is no analogue of this example there.

	Kind	PowerG5	Power6	Power7
Z6.3+lwsync+lwsync+addr	Allow	0/658M <sup>U</sup>	4.7k/1.8G	29k/4.0G
Z6.3+sync+sync+addr	Forbid	0/648M	0/3.7G	0/5.0G
W+RWC+lwsync+addr+sync	—	0/658M	2.3k/1.8G	45k/4.0G

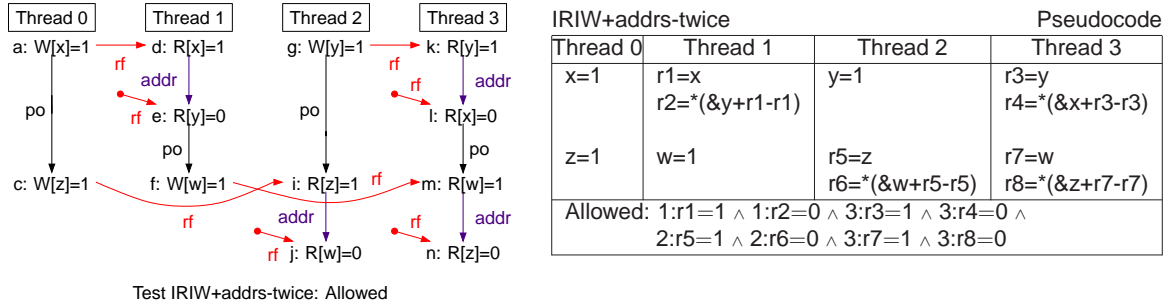
## 12 Unobservable Interconnect Topology (IRIW+addrs-twice)

A straightforward microarchitectural explanation for the behaviour of IRIW+addrs we saw in Section 6.1 would be that there is a storage hierarchy in which Threads 0 and 1 are “neighbours”, able to see each other’s writes before the other threads do, and similarly Threads 2 and 3 are “neighbours”. For example, one might have an interconnect

topology as shown below.



If that were the only reason why IRIW+adrs were allowed, then one could only observe the specified behaviour for some specific assignment of the threads of the test to the hardware threads of the implementation (some specific choice of thread affinity). That would mean that two consecutive instances of IRIW+adrs as shown below, with different assignments of test threads to hardware threads, could never be observed.



In fact, however, on some current POWER machines the IRIW+adrs-twice behaviour *is* observable (microarchitecturally, while they do have a storage hierarchy [LSF<sup>+</sup>07, KSSF10], the cache protocol behaviour alone suffices to give the observed behaviour, and threads can also be reassigned by the hypervisor in some circumstances). Moreover, it is desirable for the architectures not to require that there be a single topology fixed before a program starts executing: as far as correctness goes, the hardware threads should all be interchangeable. If programmers learn the interconnect topology, by a test like IRIW+adrs-twice or otherwise, and use that to make choices within their code, they should not expect consistent and predictable behaviour.

	Kind	PowerG5	Power6	Power7
IRIW+adrs-twice	Allow	0/290M <sup>U</sup>	0/2.9G <sup>U</sup>	5/29G

## 13 Load-reserve/Store-conditional

Load-reserve/store-conditional primitives were introduced by Jensen *et al.* [JHB87] as a RISC-architecture alternative to the compare-and-swap (CAS) instruction; they have been used on the PowerPC architecture since 1992 and are also present in ARM, MIPS, and Alpha. They are also known as load-linked/store-conditional (LL/SC), or, on ARM, load-exclusive/store-exclusive. They provide a simple form of optimistic concurrency (very roughly, optimistic transactions on single locations).

Herlihy [Her93] uses load-reserve/store-conditional to implement various wait-free and lock-free algorithms, noting that (as for CAS, but unlike test-and-set and fetch-and-add) it is *universal* in terms of consensus number, and moreover that load-reserve/store-conditional is practically superior to CAS in that it defends against the ABA problem.

We will illustrate the properties of load-reserve/store-conditional by the sequence below, which implements an atomic add operation. The first sequence is in pseudocode, followed by ARM assembly and POWER assembly.

Atomic Add (Fetch and Add)	Pseudocode	Atomic Add	ARM	Atomic Add	POWER
do { r = load-reserve x; r = r + v; } while (!store-conditional (r,x));		1:LDREX R0, [R3] ADD R0, R0, R4 STREX R1, R0, [R3] TEQ R1, #0 BNE 1b		1:lwax r0,0,r2 add r0,r1,r0 stwcx. r0,0,r2 bne- 1b	

Let us understand the code above by going through the components. The load-reserve does a load from some memory address, and establishes a *reservation* for the loading thread to that address. A subsequent store-conditional to the same address will either succeed or fail. Moreover, the store-conditional sets a flag so that later instructions can determine whether or not it succeeded; load-reserve/store-conditional pairs are often repeated until success. Note that other operations are permitted between the load-reserve and store-conditional, including memory reads and writes, though, unlike transactions, nothing is rolled back if the store-conditional fails.

So when can a store-conditional succeed, and when must it fail? Load-reserve and store-conditional are typically used in tandem as above. The key property they must jointly ensure is that, if the store-conditional succeeds, the corresponding store must be immediately after the store read-from by the load-reserve. Recalling coherence, the key condition is that the store of the successful store-conditional must immediately follow (in the coherence order) the store read-from by the immediately previous load-reserve. Furthermore, that situation should not be subject to change as the system evolves (no other write should be able to sneak in between). One subtlety is that POWER allows stores from the same thread (as the load-reserve and the store-conditional) to appear in coherence order in between the two stores above. This can only happen by program-order intervening stores to the same location between the load-reserve and the store-conditional.

The store-conditional can succeed if this coherence condition is possible, and must fail if it no longer is (for example, if another write to the same address gets propagated to that thread in between the write read from and that of the store-conditional, which means that third write must become coherence-between the two). Note also that this is merely a condition for possible success, and it is possible for the store-conditional to fail spuriously, thus making any strong guarantee of forward progress or fairness theoretically impossible, though in practice this may not be a concern.

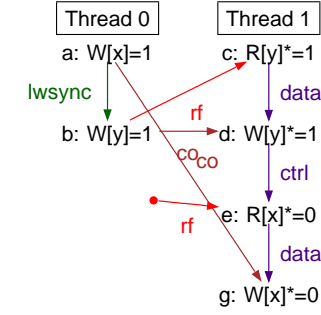
Various kinds of atomic operations can be built out of load-reserve/store-conditional pairs. For the purpose of examples, we will use two extreme forms shown below: one where the value loaded is immediately stored back (fetch and no-op), which implements an atomic load, and another which ignores the value loaded and stores a pre-determined value (store-atomic).

Fetch and No-Op	Pseudocode	Fetch-and-no-op	ARM	Fetch-and-no-op	POWER
do { r = load-reserve x; } while (!store-conditional (r,x));		1:LDREX R0, [R3] STREX R1, R0, [R3] TEQ R1, #0 BNE 1b		1:lwax r0,0,r2 stwcx. r0,0,r2 bne- 1b	
Store-atomic	Pseudocode	Store-Atomic	ARM	Store-Atomic	POWER
do { r = load-reserve x; } while (!store-conditional (v,x));		1:LDREX R0, [R3] STREX R1, R2, [R3] TEQ R1, #0 BNE 1b		1:lwax r0,0,r2 stwcx. r1,0,r2 bne- 1b	

In diagrams below, we show a load-reserve by a marked read (R\*), and a successful store-conditional by a marked write (W\*).

### 13.1 Load-reserves and Store-conditionals Stay in Order

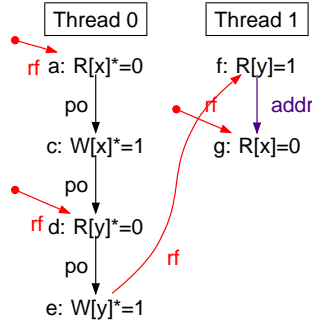
Load-reserves and store-conditionals from the same thread stay in order, that is, the load-reserve is satisfied by loading its value, and the store-conditional succeeds or fails, strictly according to program order. This means that having two fetch-and-no-op on the reader side of the MP example, together with a lwsync between the writes, makes the non-SC behaviour forbidden:



Test MP+lwsync+poaa: Forbidden

MP+lwsync+poaa		Pseudocode	
Thread 0		Thread 1	
x=1 lwsync y=1		r1=fetch-and-no-op (y) r2 = fetch-and-no-op (x)	
Initial state: $x=0 \wedge y=0$			
Forbidden: $1:r1=1 \wedge 1:r2=0$			

The program-order edge between the two atomic sequences on the reading thread here acts almost like a dependency. Indeed, we can go further, and if *all* memory accesses are replaced by atomic sequences (loads by fetch-and-no-op, stores by store-atomics), then we will only have SC behaviour. This property is less useful that it sounds, however, since the presence of any non-atomic sequence on any thread could permit observable non-SC behaviour, as we see below. Further, even though store-atomics succeed or fail in order, if they do succeed, their underlying stores can be propagated to other threads in any order (thus, there is no barrier-like effect here). This leads to the variation of Message Passing with the writes being replaced by store-atomics and the reads being ordered by dependency being allowed on POWER.

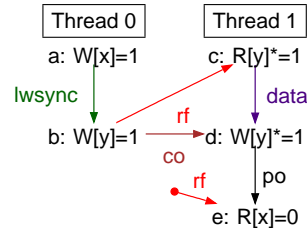


Test MP+poaa+addr: Allowed

MP+poaa+addr		Pseudocode	
Thread 0		Thread 1	
store-atomic(1,x) store-atomic (1,y)		r1=y r3=(r1 xor r1) r2=* (&x + r3)	
Initial state: $x=0 \wedge y=0$			
Allowed: $1:r1=1 \wedge 1:r2=0$			

## 13.2 Load-reserves and Store-conditionals Not Ordered with Normal Accesses

There is no special ordering requirements between load-reserve/store-conditionals and normal loads/stores, on the same or different threads. A normal load can be satisfied at any time with respect to load-reserves and store-conditionals on the same thread, ignoring program-order (the usual coherence restrictions on accesses to the same location do still apply). Similarly, a normal store and a store from a store-conditional from the same thread can propagate to other threads in any order whatsoever, as long as they are to different locations. All this means that, for example, the Message Passing test with a lwsync between the writes and just one of the reads replaced by a fetch-and-no-op still permits the non-SC behaviour.



Test MP+lwsync+poap: Allowed

MP+lwsync+poap		Pseudocode	
Thread 0		Thread 1	
x=1 lwsync y=1		r1=fetch-and-no-op (y) r2=x	
Initial state: $x=0 \wedge y=0$			
Forbidden: $1:r1=1 \wedge 1:r2=0$			

	Kind	PowerG5	Power6	Power7
MP+lwsync+poaa	Forbid	0/302M	0/6.3G	0/6.1G
MP+poaa+addr	Allow	0/302M <sup>u</sup>	27k/1.1G	56/1.4G
MP+lwsync+poap	Allow	362/20M	0/6.3G <sup>u</sup>	15k/19M

## 14 Analysis of Peterson’s algorithm on POWER

The following pseudocode is a simplification of Peterson’s algorithm for mutual exclusion [Pet81]. The presented code focusses on mutual exclusion by presenting only the “lock” fragment — Thread 0 (resp. Thread 1) would perform unlocking by writing 0 to the flag variable `f0` (resp. `f1`); and by simplifying this lock fragment. Indeed, in the actual algorithm the final `if` conditional is replaced by a `while` loop whose condition is the negation of the presented final condition. For instance Thread 0 code of the actual lock fragment could end as “`while (f1 == 1 && vict == 0) ;`”.

PET	Pseudocode
Thread 0	Thread 1
<code>f0=1</code> // write flag	<code>f1=1</code> // write flag
<code>vict=0</code> // let other pass	<code>vict=1</code> // let other pass
<code>if (f1==0    vict==1) crit0=1 ;</code>	<code>if (f0==0    vict==0) crit1=1 ;</code>
Initial state: <code>crit0=0 ∧ crit1=0</code>	
Forbidden?: <code>crit0=1 ∧ crit1=1</code>	

The above final condition `crit0=1 ∧ crit1=1` expresses mutual exclusion: if both `crit0` and `crit1` hold the value 1 at the end of test, then we witness a failure of mutual exclusion.

Due to standard short-circuiting compilation of the boolean connector `||`, if `crit0` holds the value 1, then either:

- (1) Thread 0 has read the value 0 from the location `f1`, or
- (2) Thread 0 has read the value 1 from the location `f1` and then the value 1 from the location `vict`.

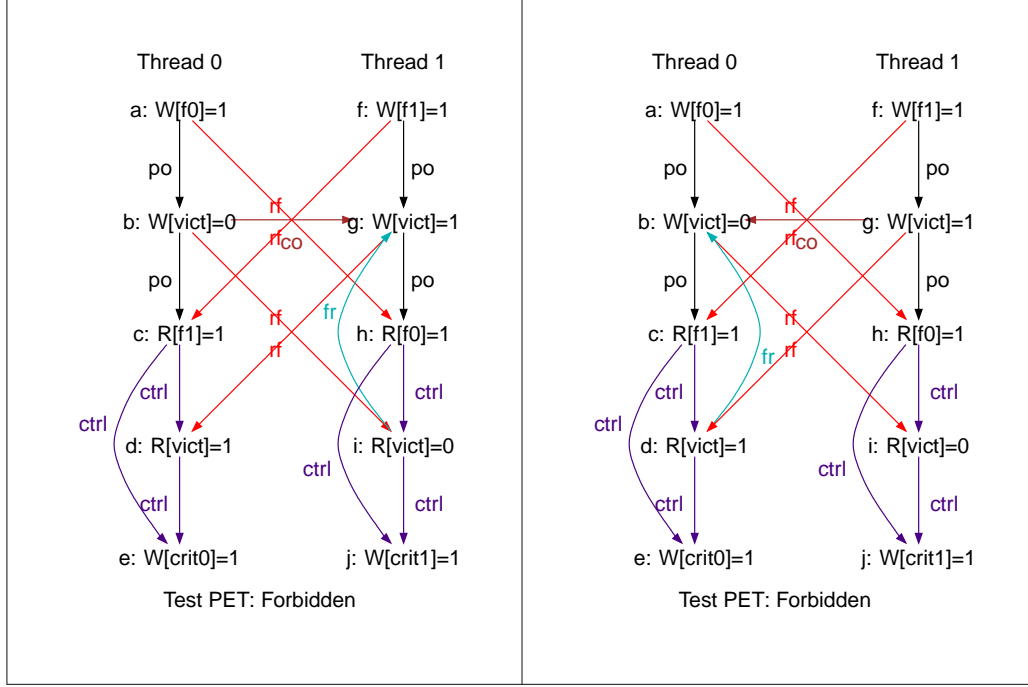
Similarly, if `crit1` holds the value 1, then either:

- (3) Thread 1 has read the value 0 from the location `f0`, or
- (4) Thread 1 has read the value 1 from the location `f0` and then the value 0 from the location `vict`.

Usually, a proof of correctness of Peterson’s algorithm checks that any two conjunction of conditions (1) or (2) on the one hand, and of (3) or (4) on the other hand leads to a contradiction. Such “contradictions” can be interpreted as violations of sequential consistency, as we discuss.

### Guarantee of mutual exclusion, by uniproc

We first consider the case where Peterson’s algorithm refines a trivial mutual exclusion algorithm that would consider flags `f0` and `f1` only. More precisely, if both threads read value 1 in the other thread’s flag, then the winner is selected by having each thread to read `vict`, considering that `vict` has a settled value that designates the loser in the competition for mutual exclusion. And indeed, on POWER we cannot have (2) and (4) simultaneously, by the uniproc condition (see Section 9.6). The following diagrams depict (candidate) executions that originate from the hypothesis (2) and (4):

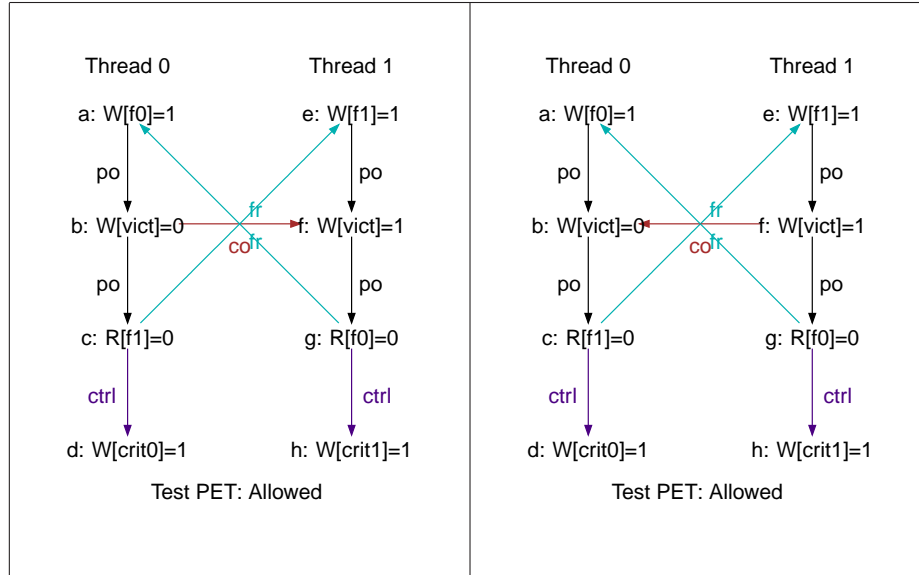


In all executions we have  $a \xrightarrow{rf} h$  and  $f \xrightarrow{rf} c$  (i.e. each thread reads the value stored in the appropriate flag by the other thread), this commands the reading of `vict` by both threads. The remaining arrows then depend on the choice of a coherence order for the writes  $b$  and  $g$  to location `vict` ( $b \xrightarrow{co} g$  for the diagram on the left,  $g \xrightarrow{co} b$  for the diagram on the right). We only pictured the situation where Thread 1 reads the value 0 stored by Thread 0 in `vict`; Thread 1 can also read the initial value of `vict`, resulting in a similar analysis, which we omit for brevity.

Then, in all executions, we have a violation of uniproc, as characterised by test **CoWR**: some `fr` edge contradicts program order from one memory access to `vict` to another — e.g. in the first diagram we have  $i \xrightarrow{fr} g \xrightarrow{po} i$ . As a consequence, the pictured executions are forbidden by the POWER architecture, as they are by any coherent architecture. More generally, the arbitration protocol by the means of the `vict` shared location introduced by Peterson does work on any coherent architecture.

### Failure of mutual exclusion, SB style

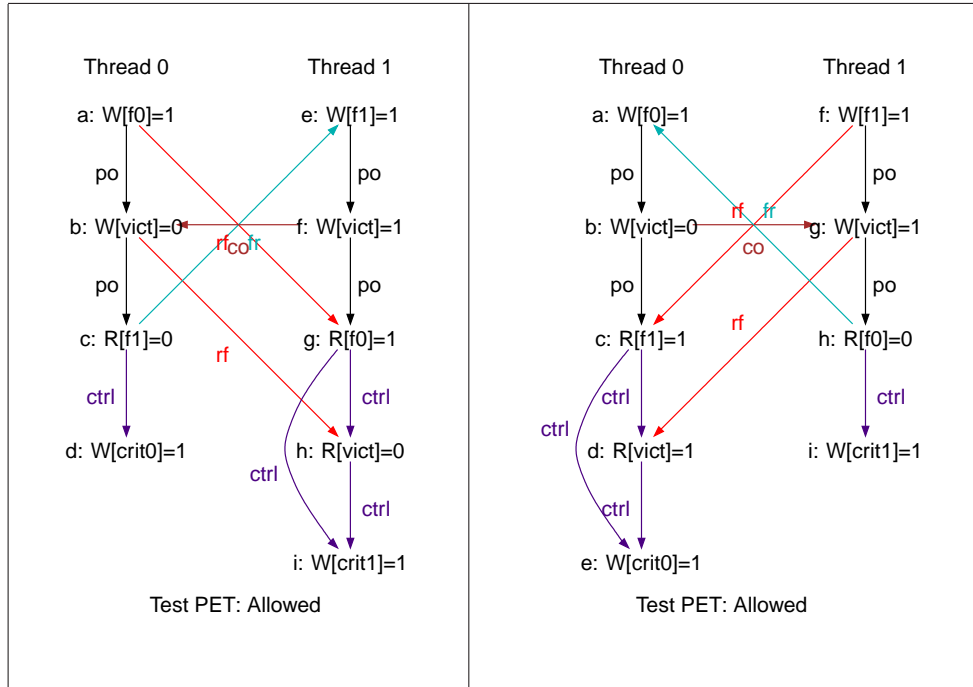
We now consider the case where (1) and (3) simultaneously hold. That is, Thread 0 reads value 0 from `f1` while Thread 1 reads value 0 from `f0`. The following diagrams depict the resulting (candidate) executions:



There are two executions and not one because of the changing coherence edge between the two writes  $b$  and  $f$  to location  $vict$ . This coherence edge is irrelevant to the pictured violations of sequential consistency: namely, there is cycle  $a \xrightarrow{po} c \xrightarrow{fr} e \xrightarrow{po} g \xrightarrow{fr} a$ . This cycle is characteristic of the SB (or Dekker) test, see the complete SB diagram in Section 9.5. As a consequence the pictured executions are allowed by the POWER architecture, as test SB is.

### Failure of mutual exclusion, R style

We now consider the cases where (1) and (4) simultaneously hold. That is, Thread 0 reads value 0 from  $f1$ , while Thread 1 reads value 1 from  $f0$  but is granted right to mutual exclusion by reading 0 from  $vict$ . This situation is depicted by the leftmost of the following two diagrams:



First notice that we also picture the symmetrical case where (2) and (3) hold on the right.



Now we turn back to the leftmost diagram and argue that the pictured execution originates from the hypothesis (1) and (4). Namely, by (1) (*i.e.* Thread 0 reads 0 from f1) Thread 0 reads the initial value of f1 and we thus have  $c \xrightarrow{\text{fr}} e$ . Furthermore by (3) (*i.e.* Thread 1 reads 1 from f1 and then 0 from vict), we first have  $a \xrightarrow{\text{rf}} g$  and then  $b \xrightarrow{\text{rf}} h$ . The first rf arrow is immediate, as the write  $a$  is the only write of 1 to f0 in the whole program. The second rf arrow  $b \xrightarrow{\text{rf}} h$  deserves a detailed argument:  $h$  reading value 0 could be from vict initial state, but this would violate the uniproc condition, because Thread 1 writes to vict before reading from it. Moreover from  $b \xrightarrow{\text{rf}} h$  we can deduce  $f \xrightarrow{\text{co}} b$ . Otherwise, we would have  $b \xrightarrow{\text{co}} f$  (as the coherence order is a total order on writes to the given location vict), and thus we would again witness a CoWR violation of coherence.

Once arrows are settled we easily see the cycle  $b \xrightarrow{\text{po}} c \xrightarrow{\text{fr}} e \xrightarrow{\text{po}} f \xrightarrow{\text{co}} b$  in the leftmost diagram, and, symetrically,  $g \xrightarrow{\text{po}} g \xrightarrow{\text{fr}} a \xrightarrow{\text{po}} b \xrightarrow{\text{co}} g$  in the rightmost diagram. Those cycles are of the R style, they are allowed and observed on POWER.

## Ensuring mutual exclusion with fences

To restore mutual exclusion it suffices to forbid the SB and R style violations of sequential consistency described in the previous two sections. As tests SB+syncs and R+syncs are forbidden on POWER, it suffices to insert two sync fences in each thread code, one after the the store to the flag f0 or f1, and one after the store to vict. The resulting program PET+syncs is shown below:

PET+syncs		Pseudocode	
Thread 0		Thread 1	
f0=1	// write flag	f1=1	// write flag
sync		sync	
vict=0	// let other pass	vict=1	// let other pass
sync		sync	
if (f1==0    vict==1) crit0=1 ;		if (f0==0    vict==0) crit1=1 ;	
Initial state: crit0=0 $\wedge$ crit1=0			
Forbidden: crit0=1 $\wedge$ crit1=1			

The following observations confirm our analysis:

		POWER			ARM		
	Kind	PowerG5	Power6	Power7	Tegra2	Tegra3	APQ8060
PET	Allow	4.7M/160M	66k/26M	475M/32G	2.3M/206M	505/10k	38k/100M
PET-UNI	Forbid	0/160M	0/26M	0/32G	0/206M	0/10k	0/100M
PET-SB	Allow	4.3M/160M	64k/26M	471M/32G	2.2M/206M	438/10k	38k/100M
PET-R	Allow	446k/160M	2.4k/26M	4.6M/32G	55k/206M	67/10k	503/100M
PET+dmbs/syncs	Forbid	0/160M	0/3.0G	0/32G	0/12G	0/16G	0/16G

One observes that the SB and R styles of mutual exclusion failure are observed, that the UNI style is not observed and that no failure of mutual exclusion is observed once fences are added to the test PET.

## 15 Sources of Tests, and Correspondences among Them

Several of our tests are taken or adapted from the literature, as we describe here.

### 15.1 Boehm and Adve examples

Boehm and Adve [BA08] give four main tests in C-like psuedo-code using a fence statement. Taking that to be a POWER sync or ARM DMB barrier, they correspond with the tests here as follows.

**IRIW** [BA08, Fig. 4]: IRIW+dmbs/syncs

**WRC** [BA08, Fig. 5]: WRC+dmbs/syncs

**RWC** [BA08, Fig. 6]: RWC+dmbs/syncs

**CC** [BA08, Fig. 7]: we do not discuss

Note that we use an unadorned IRIW, WRC or RWC to refer to versions of these tests without any dependency constraints or barriers, writing e.g. IRIW+dmbs/syncs for versions of IRIW with two POWER sync barriers or ARM DMB barriers.

## 15.2 ARM Cookbook examples

The correspondence between our tests and the examples from Section 6 of the ARM Barrier Litmus Tests and Cookbook document [ARM08b] is as follows. We label the latter ARMC6.1, etc., after their section numbers. The code of our tests is not identical to that of the Cookbook: it differs in the choice of registers, values, etc., and we avoid loops to simplify our automated checking. Apart from these minor issues, the correspondence is reasonably exact.

**ARMC6.1** *Simple Weakly Consistent Ordering Example* is SB.

**ARMC6.2.1** *Weakly-Ordered Message Passing problem* is MP.

**ARMC6.2.1.1** *Resolving by the addition of barriers* is MP+dmbs.

**ARMC6.2.1.2** *Resolving by the use of barriers and address dependency*. The main example is MP+dmb+addr.

**ARMC6.2.2** *Message passing with multiple observers* is a variant of ARMC6.2.1, but with two reading processors, seeing the writes in opposite orders. This shows that two of the possible outcomes of the unadorned MB are simultaneously possible in the same execution.

**ARMC6.2.2.1** *Resolving by the addition of barriers* adds a DMB to the writer and dependencies to the readers, giving a variant of ARMC6.2.1.2 but with two reading processors.

**ARMC6.3** *Address Dependency with object construction*. This is essentially another variant of MP+dmb+addr.

**ARMC6.4** *Causal consistency issues with Multiple observers*. The first example is a variant of WRC+po+addr, without a dependency on the middle processor. The second example adds a DMB to that processor, giving an analogue of WRC+dmb+addr.

**ARMC6.5** *Multiple observers of writes to multiple locations* The first example is IRIW. The second is IRIW+dmbs.

**ARMC6.6** *Posting a Store before polling for acknowledgement*. This is an example in which a DMB barrier after a write is used to ensure a progress property. Our impression is that more recent versions of the architecture make this barrier unnecessary.

**ARMC6.7** *WFE and WFI and Barriers*. We do not consider interrupts here.

## 15.3 Power2.06 ISA examples

The Power2.06 ISA [Pow09] Book II Chapter 1 has just two examples, in §1.7.1, intended to illustrate A- and B-cumulativity. The first is a variant of WRC+syncs. The second is an iterated message-passing example; our ISA2+sync+data+addr is a similar but loop-free test.

## 15.4 Adir et al. examples

The correspondence between our tests and those of Adir, Attiya, and Shurek [AAS03] (which we label AdirNNN) is as follows.

**Adir1** is the MP+syncs example.

**Adir1v1** removes one sync.

**Adir1v2** replaces the second sync by a load/load dependency, as in MP+sync+addr.

**Adir1v3** replaces the first sync by a “store/store” dependency, or in our terms a write then read from the same address followed by a load/store dependency, as in **MP+nondep+sync**.

**Adir1v4** replaces the second sync by a re-use of the same register.

**Adir2** is **WRC+syncs**, except that the final state is allowed. This example predated the introduction of cumulativity to PowerPC barriers.

**Adir3** is an example showing that store buffering is visible, more elaborate than **SB**.

**Adir4** is a message-passing example with a **sync** between the stores and a control dependency between the loads, showing that control dependencies between loads are not respected, as in **MP+sync+ctrl**.

**Adir5** is an example showing that control dependencies from loads to stores are respected.

**Adir6** shows that the existence of multiple copies of registers is visible to the programmer; here this is the **LB+rs** test.

**Adir7** illustrates artificial (or “false”) load/load dependencies, i.e. where the value of the first load does not in fact affect the address of the second, showing that they are respected.

**Adir8** (not included here) shows an example of behaviour which in some models is forbidden by a cycle through from-reads and sync edges.

## 15.5 Adve and Gharachorloo examples

From [AG96]:

**Fig. 4(a), Fig. 5(a)** **SB**

**Fig. 4(b), Fig. 10(b)** **WRC**

**Fig. 5(b,c)** **MP**

**Fig. 6** involves four writes on two processors (loosely analogous but not identical to the **2+2W+syncs** example), used in a discussion of write atomicity

**Fig. 10(a)** is an extension of the **SB** example with an additional read/write pair between the instructions of each processor

## 16 Related Work

There has been extensive previous work on relaxed memory models, of which we recall here just some of that on models for the major current processor families that do not have sequentially consistent behaviour: Sparc, x86, Itanium, ARM, and POWER. Early work by Collier [Col92] developed models based on empirical testing for the multiprocessors of the day. For Sparc, the vendor documentation has a clear Total Store Ordering (TSO) model [SFC91, Spa92]. It also introduces PSO and RMO models, but these are not used in practice. For x86, the vendor intentions were until recently quite unclear, as was the behaviour of processor implementations. The work by Sarkar, Owens, et al. [SSZN<sup>+</sup>09, OSS09, SSO<sup>+</sup>10] suggests that for normal user- or system-code they are also TSO. This is in a similar spirit to the work we describe here, with a mechanised semantics that is tested against empirical observation. Itanium provides a much weaker model than TSO, but one which is more precisely defined by the vendor than x86 [Int02]; it has also been formalised in TLA [JLM<sup>+</sup>03] and in higher-order logic [YGLS03].

For POWER, there have been several previous models, but none are satisfactory for reasoning about realistic concurrent code. In part this is because the architecture has changed over time: the **lwsync** barrier has been added, and barriers are now cumulative. Corella, Stone and Barton [CSB93] gave an early axiomatic model for PowerPC, but, as Adir et al. note [AAS03], this model is flawed (it permits the non-SC final state of the **MP+syncs** example we show in §3). Stone and Fitzgerald later gave a prose description of PowerPC memory order, largely in terms of the microarchitecture of the time [SF95]. Gharachorloo [Gha95] gives a variety of models for different architectures in a general framework, but the model for the PowerPC is described as “*approximate*”; it is apparently based on Corella

et al. [CSB93] and on May et al. [MSSW94]. Adve and Gharachorloo [AG96] make clear that PowerPC is very relaxed, but do not discuss the intricacies of dependency-induced ordering, or the more modern barriers. Adir, Attiya, and Shurek give a detailed axiomatic model [AAS03], in terms of a view order for each thread. The model was “*developed through an iterative process of successive refinements, numerous discussions with the PowerPC architects, and analysis of examples and counterexamples*”, and its consequences for a number of litmus tests (some of which we use here) are described in detail. These facts inspire some confidence, but it is not easy to understand the force of the axioms, and it describes *non-cumulative* barriers, following the pre-PPC 1.09 PowerPC architecture; current processors appear to be quite different. More recently, Chong and Ishtiaq give a preliminary model for ARM [CIO8], which has a very similar architected memory model to POWER. In our initial work in this area [AFI<sup>+</sup>09], we gave an axiomatic model based on a reading of the Power ISA 2.05 and ARM ARM specifications, with experimental results for a few tests (described as work in progress); this seems to be correct for some aspects but to give an unusably weak semantics to barriers. More recently still, Alglave et al. gave a rather different axiomatic model [AMSS10], further developed in Alglave’s thesis [Alg10] as an instance of a general framework; it models the non-multiple-copy-atomic nature of POWER (with examples such as IRIW+addr correctly allowed) in a simple global-time setting. The axiomatic model is sound with respect to our experimental tests, and on that basis can be used for reasoning, but it is weaker than the observed behaviour or architectural intent for some important examples. Moreover, it was based principally on black-box testing and its relationship to the actual processor implementations is less clear than that for the operational model of [SSA<sup>+</sup>11, SMO<sup>+</sup>12], which are more firmly grounded on microarchitectural and architectural discussion. In more detail, the axiomatic model is weaker than one might want for lwsync and for cumulativity: it allows MP+lwsync+addr and ISA2+sync+data+addr, which are not observed and which are intended to be architecturally forbidden. It also forbids the R+lwsync+sync variant of R which is not observed but architecturally intended to be allowed.

We mention also Lea’s *JSR-133 Cookbook for Compiler Writers* [Lea], which gives informal (and approximate) models for several multiprocessors, and which highlights the need for clear models.

## 17 On-line material

Various supporting material is available on-line at <http://www.cl.cam.ac.uk/~pes20/ppc-supplemental>:

- There are papers describing an operational abstract-machine model for POWER and its extension for load-reserve/store-conditional and eieio instructions:
  - Understanding POWER multiprocessors. Susmit Sarkar, Peter Sewell, Jade Alglave, Luc Maranget, and Derek Williams. In *Proc. PLDI*, 2011. [SSA<sup>+</sup>11]
  - Synchronising C/C++ and POWER. Susmit Sarkar, Kayvan Memarian, Scott Owens, Mark Batty, Peter Sewell, Luc Maranget, Jade Alglave, and Derek Williams. In *Proc. PLDI*, 2012. [SMO<sup>+</sup>12]

These also describe how that model explains the behaviour of some of the tests we discuss here, and summarise some of our experimental data in support of the model. The following paper, together with the PLDI 2012 paper above, describes a correctness proof for an implementation of the C/C++ concurrency model of the C11 and C++11 revised standards [BA08, BOS<sup>+</sup>11, Bec11, ISO11] above POWER processors.

- Clarifying and Compiling C/C++ Concurrency: from C++11 to POWER. Mark Batty, Kayvan Memarian, Scott Owens, Susmit Sarkar, and Peter Sewell. In *Proc. POPL*, 2012. [BMO<sup>+</sup>12]

The following paper gives an axiomatic model for POWER (without load-reserve/store-conditional), equivalent to the abstract-machine model above.

- An Axiomatic Memory Model for POWER Multiprocessors. Sela Mador-Haim, Luc Maranget, Susmit Sarkar, Kayvan Memarian, Jade Alglave, Scott Owens, Rajeev Alur, Milo M.K. Martin, Peter Sewell, and Derek Williams. In *Proc. CAV*, 2012 [MHMS<sup>+</sup>12].
- Our ppcmem tool lets one interactively explore the behaviour of a POWER or ARM litmus test with respect to our model; this is available via a web interface at <http://www.cl.cam.ac.uk/~pes20/ppcmem>. The use of ppcmem was described in a Linux Weekly News (LWN) article by McKenney [McK11].

Note that at the time of writing the `ppcmem` tool is based on the model presented in those papers which was developed principally for POWER. The ARM mode of `ppcmem` uses the same model instantiated to a small fragment of the ARM instruction set. We believe this to be correct in most instances but there are cases, most notably the `MP+dmb+fri-rfi-ctrlisb` test we describe in Section 10.5, where an ARM test has observable and architecturally allowed behaviour that that model forbids. Work on a revised model for ARM is in progress.

- Our `litmus` tool takes a litmus test and constructs a test harness (as a C program with embedded assembly) to experimentally test its observable behaviours. This is downloadable from <http://diy.inria.fr>, which also includes our `diy` tool for generating litmus tests from concise specifications. The `litmus` tool is described in this paper:
  - Litmus: running tests against hardware. Jade Alglave, Luc Maranget, Susmit Sarkar, and Peter Sewell. In *Proc. TACAS*, 2011. [AMSS11b]
- A summary of tests and of experimental results.

**Acknowledgements** We acknowledge funding from EPSRC grants EP/F036345, EP/H005633, and EP/H027351, and from ANR project WMC (ANR-11-JS02-011).

## 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.
- [ABJ<sup>+</sup>93] Mustaque Ahamad, Rida A. Bazzi, Ranjit John, Prince Kohli, and Gil Neiger. The power of processor consistency. In *SPAA*, pages 251–260, 1993.
- [AFI<sup>+</sup>09] J. Alglave, A. Fox, S. Ishtiaq, M. O. Myreen, S. Sarkar, P. Sewell, and F. Zappa Nardelli. The semantics of Power and ARM multiprocessor machine code. In *Proc. DAMP 2009*, January 2009.
- [AG96] S. V. Adve and K. Gharachorloo. Shared memory consistency models: A tutorial. *IEEE Computer*, 29(12):66–76, 1996.
- [Alg10] Jade Alglave. *A Shared Memory Poetics*. PhD thesis, Université Paris 7 – Denis Diderot, November 2010.
- [AMSS10] J. Alglave, L. Maranget, S. Sarkar, and P. Sewell. Fences in weak memory models. In *Proc. CAV*, 2010.
- [AMSS11a] J. Alglave, L. Maranget, S. Sarkar, and P. Sewell. Litmus: Running tests against hardware. In *Proc. TACAS*, 2011.
- [AMSS11b] Jade Alglave, Luc Maranget, Susmit Sarkar, and Peter Sewell. Litmus: running tests against hardware. In *Proc. TACAS: the 17th international conference on Tools and Algorithms for the Construction and Analysis of Systems, LNCS 6605, TACAS’11/ETAPS’11*, pages 41–44. Springer-Verlag, 2011.
- [ARM08a] ARM. *ARM Architecture Reference Manual (ARMv7-A and ARMv7-R edition)*. April 2008.
- [ARM08b] ARM. ARM Barrier Litmus Tests and Cookbook, October 2008. PRD03-GENC-007826 2.0.
- [BA08] H.-J. Boehm and S. Adve. Foundations of the C++ concurrency memory model. In *Proc. PLDI*, 2008.
- [Bec11] P. Becker, editor. *Programming Languages — C++*. 2011. ISO/IEC 14882:2011. A non-final but recent version is available at <http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2011/n3242.pdf>.
- [BMO<sup>+</sup>12] Mark Batty, Kayvan Memarian, Scott Owens, Susmit Sarkar, and Peter Sewell. Clarifying and Compiling C/C++ Concurrency: from C++11 to POWER. In *Proceedings of POPL 2012: The 39th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (Philadelphia)*, 2012.

- [BOS<sup>+</sup>11] M. Batty, S. Owens, S. Sarkar, P. Sewell, and T. Weber. Mathematizing C++ concurrency. In *Proc. POPL*, 2011.
- [CI08] N. Chong and S. Ishtiaq. Reasoning about the ARM weakly consistent memory model. In *MSPC*, 2008.
- [CLS03] Jason F. Cantin, Mikko H. Lipasti, and James E. Smith. The complexity of verifying memory coherence. In *SPAA*, pages 254–255, 2003.
- [Col92] W.W. Collier. *Reasoning about parallel architectures*. Prentice-Hall, Inc., 1992.
- [CSB93] F. Corella, J. M. Stone, and C. M. Barton. A formal specification of the PowerPC shared memory architecture. Technical Report RC18638, IBM, 1993.
- [Gha95] K. Gharachorloo. Memory consistency models for shared-memory multiprocessors. *WRL Research Report*, 95(9), 1995.
- [Her93] M. Herlihy. A methodology for implementing highly concurrent data objects. *TOPLAS*, 15(5):745–770, Nov 1993.
- [Int02] Intel. A formal specification of Intel Itanium processor family memory ordering, 2002. `developer.intel.com/design/itanium/downloads/251429.htm`.
- [ISO11] *Programming Languages — C*. 2011. ISO/IEC 9899:2011. A non-final but recent version is available at <http://www.open-std.org/jtc1/sc22/wg14/docs/n1539.pdf>.
- [JHB87] E. H. Jensen, G. W. Hagensen, and J. M. Broughton. A new approach to exclusive data access in shared memory multiprocessors. (Technical Report UCRL-97663), Nov 1987.
- [JLM<sup>+</sup>03] R. Joshi, L. Lamport, J. Matthews, S. Tasiran, M. Tuttle, and Y. Yu. Checking cache-coherence protocols with TLA+. *Form. Methods Syst. Des.*, 22:125–131, March 2003.
- [KSSF10] R. Kalla, B. Sinharoy, W. J. Starke, and M. Floyd. Power7: IBM’s next-generation server processor. *IEEE Micro*, 30:7–15, March 2010.
- [Lam79] L. Lamport. How to make a multiprocessor computer that correctly executes multiprocess programs. *IEEE Trans. Comput.*, C-28(9):690–691, 1979.
- [Lea] D. Lea. The JSR-133 cookbook for compiler writers. <http://gee.cs.oswego.edu/dl/jmm/cookbook.html>.
- [LHH91] Anders Landin, Erik Hagersten, and Seif Haridi. Race-free interconnection networks and multiprocessor consistency. *SIGARCH Comput. Archit. News*, 19(3):106–115, 1991.
- [LSF<sup>+</sup>07] H. Q. Le, W. J. Starke, J. S. Fields, F. P. O’Connell, D. Q. Nguyen, B. J. Ronchetti, W. Sauer, E. M. Schwarz, and M. T. Vaden. IBM POWER6 microarchitecture. *IBM Journal of Research and Development*, 51(6):639–662, 2007.
- [McK11] Paul McKenney. Validating memory barriers and atomic instructions. Linux Weekly News <http://lwn.net/Articles/470681/>, December 2011.
- [MHMS<sup>+</sup>12] Sela Mador-Haim, Luc Maranget, Susmit Sarkar, Kayvan Memarian, Jade Alglave, Scott Owens, Raveej Alur, Milo M. K. Martin, Peter Sewell, and Derek Williams. An axiomatic memory model for power multiprocessors. In *Proc. CAV, 24th International Conference on Computer Aided Verification, LNCS 7358*, pages 495–512, 2012.
- [MSSW94] C. May, E. Silha, R. Simpson, and H. Warren, editors. *The PowerPC architecture: a specification for a new family of RISC processors*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1994.
- [OSS09] S. Owens, S. Sarkar, and P. Sewell. A better x86 memory model: x86-TSO. In *Proc. TPHOLs*, pages 391–407, 2009.



- [Pet81] G.L. Peterson. Myths about the mutual exclusion problem. *Information Processing Letters*, 12(3):115–116, 1981.
- [Pow09] *Power ISA<sup>TM</sup> Version 2.06*. IBM, 2009.
- [SF95] J. M. Stone and R. P. Fitzgerald. Storage in the PowerPC. *IEEE Micro*, 15:50–58, April 1995.
- [SFC91] P. S. Sindhu, J.-M. Frailong, and M. Cekanov. Formal specification of memory models. In *Scalable Shared Memory Multiprocessors*, pages 25–42. Kluwer, 1991.
- [SMO<sup>+</sup>12] Susmit Sarkar, Kayvan Memarian, Scott Owens, Mark Batty, Peter Sewell, Luc Maranget, Jade Alglave, and Derek Williams. Synchronising C/C++ and POWER. In *Proceedings of PLDI, the 33rd ACM SIGPLAN conference on Programming Language Design and Implementation (Beijing)*, 2012.
- [Spa92] *The SPARC Architecture Manual, V. 8*. SPARC International, Inc., 1992. Revision SAV080SI9308. <http://www.sparc.org/standards/V8.pdf>.
- [SS88] Dennis Shasha and Marc Snir. Efficient and correct execution of parallel programs that share memory. *ACM Trans. Program. Lang. Syst.*, 10(2):282–312, 1988.
- [SSA<sup>+</sup>11] Susmit Sarkar, Peter Sewell, Jade Alglave, Luc Maranget, and Derek Williams. Understanding POWER multiprocessors. In *Proc. PLDI*, 2011.
- [SSO<sup>+</sup>10] P. Sewell, S. Sarkar, S. Owens, F. Zappa Nardelli, and M. O. Myreen. x86-TSO: A rigorous and usable programmer’s model for x86 multiprocessors. *Communications of the ACM*, 53(7):89–97, July 2010.
- [SSZN<sup>+</sup>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.
- [YGLS03] Y. Yang, G. Gopalakrishnan, G. Lindstrom, and K. Slind. Analyzing the Intel Itanium memory ordering rules using logic programming and SAT. In *Proc. CHARME, LNCS 2860*, 2003.