1 Basic terminology

(For consistency I am using the terminology in Sebastian's note.)

Basic sets:

P	set of processors, is $\{1, \ldots, n\}$ for some number $n \in \aleph$
A	set of storage addresses
V	set of values
$i(a)$	initial value of location $a \in A$; ranges over V

Event structure:

E	set of events
$t(e) \in \{load, store, sync\}$	type of event $e \in E$
$p(e) \in P$	processor that executes event $e \in E$
$a(e) \in A$	storage address used by event $e \in E$
$v(e) \in V$	value loaded or stored by event $e \in E$

Derived notation:

$E(p)$	events executed by processor p	$= \{e \in E \mid p(e) = p\}$
L, S, F	load/store/sync (fence) events	
$L(p), S(p), F(p)$	stores executed by processor p	
$S(a)$	store events to address a	$= \{s \in S \mid a(s) = a\}$
L_f	local loads ordered by sync f	$= L \cap F(f)$

2 The PPC-Core model PPCw

In this model, the requirement "load must be performed wrt processor q" does not place any restriction on the behavior of q.

Write order:

$<_w$ is a total order over $S(a)$ for all $a \in A$

Process-local orders:

$<_p$ is a partial order over E that is total on $E(p)$, for each $p \in P$,

We define $V(p)$ the set of *events seen by processor* p as $\{e \mid \exists e' \in E : e <_p e' \vee e' <_p e\}$. (Note: for $E(p)$ containing at least 2 elements, $E(p) \subseteq V(p)$.)

The tuple $(E, <_w, <_1, \ldots, <_n)$ is a *valid execution* if the following axioms are satisfied.

Coherence. For any processor p, $\forall a \in A; s, s' \in S(a) : s <_p s' \Rightarrow s <_w s'$.

Value flow. Each load gets the value of the latest preceding store to the same address that the processor observes, or the initial value if there is no such store. Formally, for all processors p and all loads $l \in L(p)$:

(V1) $v(l) = i(a(l)) \vee \exists s \in S(a(l)) : s <_p l$

(V2) $\forall s \in S(a(l)) \cap V(p) :$
$(\exists s' \in S(a(l)) : s <_p s' <_p l) \vee v(l) = v(s) \vee l <_p s$

Fence Accumulation. Fences order stores so that they are globally visible.

For every store $s \in S(a)$, define *loads*(s), the set of loads which read from s by:

$$\textit{loads}(s) = \{l \in L(a(s)) \mid s <_{p(l)} l, \forall s' \in S(a(l)) : s' <_{p(l)} l \Rightarrow s' <_{p(l)} s \vee s' = s\}$$

We can now formalize the barrier requirements (Section B.3). For any fence f, and processor $p = p(f)$ both the following conditions must hold:

(F1) $\forall e \in E, e' \in E(p) : e <_p f <_p e' \Rightarrow \forall q \in P : e <_q e'$

(F2) $\forall e \in E, s \in S(p), l \in \textit{loads}(s) : e <_p f <_p s, l <_{p(l)} e' \Rightarrow \forall q \in P : e <_q e'$

2.1 Examples

Example 2.1 (Dekker: Permitted) *Consider the program:*

```
p=1           p=2
[1] x=1       [3] y=1
[1s]sync      [3s] sync
[2] _=y(0)    [4] _=x(0)
```

By program order and (F1):

$$[1] <_1 [1s], [1s] <_1 [2], [3] <_2 [3s], [3s] <_2 [4],$$
$$[3] <_1 [4], [1] <_2 [2]$$

Now $[3] \in V(1)$*, therefore (V2) forces* $[2] <_1 [3]$ *(in order for* $[2]$ *to return* 0*). Therefore* $[1] <_1 [3]$*. Unfortunately there is no way to turn this into* $[1] <_2 [3]$ *(this would be one way of getting a contradiction). The only way to do so would be by applying (F1) or (F2). But (F1) is not applicable to the pair* $([1], [3])$*, since* $[3]$ *is not in* $E(1)$*. (F2) is not applicable because there is no store in* $\{[2]\}$*, the* B*-set of the sync for* $p = 1$*.*

Therefore we have an execution:

$E =$ $\{[1], \ldots, [4]\}$

$<_w$ $\emptyset$

$<_1$ $[1] <_1 [1s], [1s] <_1 [2], [2] <_1 [3], [3] <_1 [4]$

$<_2$ $[3] <_2 [3s], [3s] <_2 [4], [4] <_2 [1], [1] <_2 [2]$

Note that $[1] <_1 [4]$ *does not cause a contradiction because the relevant condition (V2) applies only to loads* l *at processors* $p = p(l)$*. Effectively, this semantics places no restrictions on* $<_p$ *due to foreign loads (loads* l *s.t.* $p(l) \neq p$*).*

Example 2.2 (IRIW: Permitted) *Consider the program:*

```
p=1           p=2           p=3       p=4
[1a] _=x(1)   [3] _=y(1)    [1] x=1   [3] y=1
[1s]sync      [3s] sync
[2] _=y(0)    [4] _=x(0)
```

The analysis is as above.

Example 2.3 (Causal Consistency, CC: Forbidden) *Consider the program:*

```
p=1          p=2          p=3
[1a] _=x(1)  [3] _=y(1)   [1] x=1
[1s]sync     [3s] sync
[2] y=1      [4] _=x(0)
```

This is not an execution of PPCw.

(V2) forces $[1]$ *to be visible at* $p = 1$*, and* $[1] <_1 [1a]$*, and hence* $[1] <_1 [2]$*. (F1) causes* $[1] <_2 [2]$*. (V2) forces* $[2] <_2 [3]$*. Therefore it follows that* $[1] <_2 [4]$ *and hence* $[4]$ *must return* 1*.*

Example 2.4 (Longer Causal Consistency, CC: Forbidden) *Consider the program:*

```
p=1          p=2          p=3       p=4
[1a] _=x(1)  [3] _=y(1)   [1] x=1   [5] _=z(1)
[1s]sync     [3s] sync              [5s] sync
[2] y=1      [4] z=1                [6] _=x(0)
```

This is not an execution of PPCw. The same reasoning as in the previous example, carried through one more step.

(V2) forces $[1]$ *to be visible at* $p = 1$*, and* $[1] <_1 [1a]$*, and hence* $[1] <_1 [2]$*. (F1) causes* $[1] <_2 [2]$*. (V2) forces* $[2] <_2 [3]$*. Therefore it follows that* $[1] <_2 [3s] <_2 [4]$*. Now (F1) can be applied to transfer* $[1] <_2 [4]$ *to* $<_4$*. This forces* $[6]$ *to read* 1*.*

Example 2.5 (Direct Consistency, DC: Permitted) *Consider the program:*

```
p=1          p=2          p=3
[1a] _=x(1)  [3] y=1      [1] x=1
[1s]sync     [3s] sync
[2] _=y(0)   [4] _=x(0)
```

This is a variant of Dekker that is permitted because Dekker is:

$E =$ $\{[1], \ldots, [4]\}$
$<_w$ $\emptyset$
$<_1$ $[1] <_1 [1a] <_1 [1s] <_1 [2] <_1 [3] <_1 [4]$
$<_2$ $[3] <_2 [3s] <_2 [4] <_2 [1] <_2 [1a] <_2 [2]$

Example 2.6 (LF – Lock-free Stack/Queue/List:Forbidden) *Consider the program:*

```
p=1          p=2
[1] x=1      [3] _=y(1)
[1s] sync    [3s] sync
[2] y=1      [4] r2=x(0)
```

This example is a direct application of (F2) to $[1], [2], [3]$ *and* $[4]$ *yielding* $[1] <_2 [4]$*, hence* $[4]$ *must return* 1*.*

Example 2.7 (Snapshot: Permitted) *Consider the program:*

```
p=1          p=2          p=3      p=4
[1a] _=x(1) [3] _=x(0)   [1] x=1 [3] y=1
[1s] sync    [3s] sync
[2] _=y(0)   [4] _=y(1)
[2s] sync    [4s] sync
[2b] _=x(1) [4b] _=x(0)
```

This is permitted. We will get $[1] <_1 [1a]$*, and hence by (F1)* $[1] <_2 [2]$*, and by (V2),* $[4b] <_2 [1]$*. However, the* $[4] <_2 [2]$ *does not generate a contradiction because* $[2]$ *is a foreign load.*

3 The PPC-Core model PPCs

In this model, the requirement "load l must be performed wrt processor q" is interpreted as "load l must return the same value when it is performed wrt processor q."

This is formalized by requiring that (V1) and (V2) apply to all loads $l \in V(p)$ (and not just loads $l \in L(p)$).

Example 3.1 (Dekker: Forbidden) *Consider the program:*

```
p=1          p=2
[1] x=1      [3] y=1
[1s]sync     [3s] sync
[2] _=y(0)   [4] _=x(0)
```

By program order and (F1):

$$[1] <_1 [1s], [1s] <_1 [2], [3] <_2 [3s], [3s] <_2 [4],$$
$$[3] <_1 [4], [1] <_2 [2]$$

Now $[3] \in V(1)$*, therefore (V2) forces* $[2] <_1 [3]$ *(in order for* $[2]$ *to return* 0*). Therefore* $[1] <_1 [3]$*, and hence* $[1] <_1 [4]$*. But then the value returned by* $[4]$ *should be* 1*, not* 0*.*

So this is not an execution.

Example 3.2 (IRIW: Forbidden) *Consider the program:*

```
p=1          p=2          p=3      p=4
[1a] _=x(1) [3] _=y(1)   [1] x=1 [3] y=1
[1s]sync     [3s] sync
[2] _=y(0)   [4] _=x(0)
```

The analysis is as above.

CC, LCC, LF are forbidden by PPCw, and hence by PPCs.

Example 3.3 (Direct Consistency, DC: Forbidden) *Consider the program:*

```
p=1          p=2          p=3
[1a] _=x(1)  [3] y=1      [1] x=1
[1s]sync     [3s] sync
[2] _=y(0)   [4] _=x(0)
```

This is forbidden because Dekker is. The execution permitted by PPCw:

$E =$ $\{[1], \ldots, [4]\}$
$<_w$ $\emptyset$
$<_1$ $[1] <_1 [1a] <_1 [1s] <_1 [2] <_1 [3] <_1 [4]$
$<_2$ $[3] <_2 [3s] <_2 [4] <_2 [1] <_2 [1a] <_2 [2]$

is not an execution because $[4]$ *must return* 1 *in* $p = 1$.

Example 3.4 (Snapshot: Forbidden) *Consider the program:*

```
p=1          p=2          p=3      p=4
[1a] _=x(1)  [3] _=x(0)   [1] x=1  [3] y=1
[1s] sync    [3s] sync
[2] _=y(0)   [4] _=y(1)
[2s] sync    [4s] sync
[2b] _=x(1)  [4b] _=x(0)
```

This is forbidden. We will get $[1] <_1 [1a]$*, and hence by (F1)* $[1] <_2 [2]$*, and by (V2),* $[4b] <_2 [1]$*. Now* $[4] <_2 [2]$ *will force* $[2]$ *to read* 1*, and this is a contradiction.*

References

[WSM+05] Joe Wetzel, Ed Silha, Cathy May, Brad Frey, Junichi Furukawa, and Giles Frazier. Powerpc virutal environment architecture, book ii, version 2.02. Technical report, IBM Corporation, January 2005.

A Rationale and notes

We model the execution of PowerPC processors executing sequences of Load, Store and **hwsync** (fence, $L = 0$ ***sync***) instructions against memory that is Memory Coherence Required and is neither Write Through Required nor Caching Inhibited.

Why introduce $<_w$? We wish to model:

> Atomic stores to a given location are *coherent* if they are serialized in some order, and no processor or mechanism is able to observe any subset of those stores as occurring in a conflicting order. This serialization order is an abstract sequence of values; the physical storage location need not assume each of the values written to it.

Thus $<_w$ captures the total order on all writes to a location, even if some of the writes are not seen by any processor (other than the one that performed it). We do not require that $<_w$ is contained in $<_p$ to model the idea that not all writes are performed at every

processor. Similarly, $V(p)$ is not required to be E, since some events may never be performed with respect to p (e.g. because the processor that performed them did not use fences).

The requirement:

> The sequence of values loaded from the location by any processor during an interval of time forms a subsequence of the sequence of values that the location logically held during that interval.

is captured by the formal Coherence condition.

Why model $<_p$ as a partial order rather than a total order? We have sought to model a load being performed against another processor, not just a store. Hence $V(p)$ may include loads belonging to other processors. However, there is no language in the redbooks which requires these loads to be totally ordered at p.

We turn to the fence conditions. (F1) captures the dependency between the cumulative A set and those elements in B belonging to the process p. (F2) directly captures the cumulativity clause for B.

B Text from the PowerPC redbooks

B.1 Performed

From [WSM+05, p. 10], the definition of *performed*

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

B.2 Memory Coherence Required

From [WSM+05, p. 5,Sec 1.6.3]:

> An access to a Memory Coherence Required storage location is performed coherently, as follows.
>
> Memory coherence refers to the ordering of stores to a single location. Atomic stores to a given location are *coherent* if they are serialized in some order, and no processor or mechanism is able to observe any subset of those stores as occurring in a conflicting order. This serialization order is an abstract sequence of values; the physical storage location need not assume each of the values written to it. For example, a processor may update a location several times before the value is written to physical storage. The result of a store operation is not available to every processor or mechanism at the same instant, and it may be that a processor or mechanism observes only some of the values that are written to a location. However,

when a location is accessed atomically and coherently by all processor and mechanisms, the sequence of values loaded from the location by any processor or mechanism during any interval of time forms a subsequence of the sequence of values that the location logically held during that interval. That is, a processor or mechanism can never load a "newer" value first, and then, later, load an "older" value.

B.3 Barrier

From [WSM+05, p. 15,Sec 1.7.1]:

> When a processor (P1) executes a *Synchronize* or *eieio* instruction a *memory barrier* is created, which orders applicable storage accesses pairwise, as follows. Let A be a set of storage accesses that includes all storage accesses associated with instructions preceding the barrier-creating instruction, and let B be a set of storage accesses that includes all storage accesses associated with instructions following the barrier-creating instruction. For each applicable pair a_i, b_j of storage accesses such that a_i is in A and b_j is in B, the memory barrier ensures that a_i will be performed with respect to any processor or mechanism, to the extent required by the associated Memory Coherence Required attributes, before b_j is performed with respect to that processor mechanism.
>
> The ordering done by a memory barrier is said to be "cumulative" if it also orders storage accesses that are performed by processors and mechanisms other than P1, as follows:
>
> - A includes all applicable storage acccesses by any such processor or mechanism that have been performed with respect to P1 before the memory barrier is created.
> - B includes all applicable storage accesses by any such processor or mechanism that are performed after a Load instruction executed by that processor or mechanism has returned the value stored by a store that is in B.

B.4 Sync

From [WSM+05, p. 26,Sec 3.3.3]:

> The ***sync*** instruction creates a memory barrier (see Section 1.7.1). The set of storage accesses that is ordered by the memory barrier depends on the value of the L field.
>
> **L=0("heavyweight sync")** The memory barrier provides an ordering function for the storage accesses associated with all instructions that are executed by the processor executing the ***sync*** instruction. The applicable pairs are all pairs a_i, b_j in which b_j is a data access, except that if a_i is the storage access caused by an **icbi** instruction then b_j

may be performed with respect to the processor executing the ***sync*** instruction before a_i is performed with respect to that processor.

L=1 ("lightweight sync") The memory barrier provides an ordering function for the storage accesses caused by *Load*, *Store*, and **dcbz** instructions that are executed by the processor executing the**sync** instruction and for which the specified storage location is in storage that is Memory Coherence Required and is neither Write Through Required nor Caching Inhibited. The applicable pairs are all pairs a_i, b_j of such accesses except those in which a_i is an access caused by a Store or **dcbz** instruction and b_j is an access caused by a *Load* instruction.

The ordering done by the memory barriers is cumulative.

If L=0 (or L=2), the ***sync*** instruction has the following additional properties:

- Executing the ***sync*** instruction ensures that all instructions preceding the ***sync*** instruction have completed before the ***sync*** instruction completes, and that no subsequent instructions are initiated until after the ***sync*** instruction completes.
- The ***sync*** instruction is execution synchronizing (see Book III, *PowerPC Operating Environment Architecture*). However, address translation and reference and change recording (see Book III) associated with subsequent instructions may be performed before the ***sync*** instruction completes.
- The memory barrier provides the additional ordering function such that if a given instruction that is the result of a *Store* in set B is executed, all applicable storage accesses in set A have been perforemd with respect to the processor executing the instruction to the extend required by the associated memory coherence properties. The single exception is that any storage access in set A that is caused by an ***icbi*** instruction executed by the processor executing the ***sync*** instruction (P1) may not have been performed with respect to P1 (see the description of the ***icbi*** instruction on page 18).

  The cumulative properties of the barrier apply to the execution of the given instruction as they would to a Load that returned a value that was the result of a *Store* in set B.
- The ***sync*** instruction provides an ordering function for the operation caused by **dcbr** instruction with $TH_0 = 1$.

The value L=3 is reserved.

The ***sync*** instruction may complete before storage accesses associated with instructions preceding the ***sync*** instruction have been performed. The ***sync*** instruction may complete before operations caused by ***dcbt*** instructions with $TH_0 = 1$ preceding the ***sync*** instruction have been performed.