Testing (and defining) Weak Memory Models

Luc Maranget
Inria Paris-Rocquencourt

The Sequentially Consistent Model (SC)

Definition by L. Lamport:

"...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".

(One may add "stores take effect immediately".)

Interleaving semantics: This is "interleaving semantics" as "some sequential order" results from interleaving "the order specified by the program of all individual processors".

A first, one expect shared memory multiprocessors to behave that way, which of course they don’t.

A simple shared memory computer

Threads execute programs as usual: instructions are executed completely and atomically (memory stores in particular).

Axiomatic or post-mortem semantics — Events

The effects of "operations executed by the processors" are represented by events. We define memory events (a):d[ℓ]v:

- Unique label typically (a), (b), etc.
- Direction d, that is read (R) or write (W)
- Memory location ℓ, typically x, y, etc.
- Value v, typically 0, 1 etc.
- Originating thread: T0, T1 (often omitted)

The program order →po ("order specified by program") is a linear order amongst the events originating from the same thread.

Relation →po represents the sequential execution of events by one thread that follows the uniprocessor model: the usual processor execution model, where instructions are executed by following the order given in program.
Example of program order

Despite its name, program order is a dynamic notion.

/* x, t and y are (shared) memory locations, t = { 2, 3, } */
int r1, r2 = 0 ; // non-shared locations (e.g. registers)
x = 1 ;
for (int k = 0 ; k < 2 ; k++) { r1 = t[k] ; r2 += r1 ; }
y = r2 ;

Events and program order :

Notice: program order (and events) may depend on the values of
the reads, i.e. on values written by other threads (if . . . ).
In simple examples, program order is given by program text.

Relating writes and reads

We define a first “communication relation” between events with the
same location.

Definition (Read-from →rf)
Relates write events to read events that read the stored value (implicit
initial writes).
1. Existence and unicity:
   \( \forall r, \exists! w, w \rightarrowrf r \)
2. Same location, same value:
   \( \text{loc}(w) = \text{loc}(r) \land \text{val}(w) = \text{val}(r) \).

A definition of SC

Definition (SC 1)
An execution is SC when there exists a total order on events <, such
that:
1. Order < is compatible with program order:
   \( e_1 \rightarrowpo e_2 \implies e_1 < e_2 \).
2. A Read \( r \) reads from the recent write before \( r \) in <.
   \( \rightarrowrf \text{Def} \{ (w, r) | w = \max(w', \text{loc}(w') = \text{loc}(r) \land w' < r) \} \).
A question on SC

Program:

<table>
<thead>
<tr>
<th></th>
<th>$T_0$</th>
<th>$T_1$</th>
</tr>
</thead>
<tbody>
<tr>
<td>(a)</td>
<td>$x \leftarrow 1$</td>
<td>(c) $y \leftarrow 2$</td>
</tr>
<tr>
<td>(b)</td>
<td>$y \leftarrow 1$</td>
<td>(d) $r_0 \leftarrow x$</td>
</tr>
</tbody>
</table>

Observed? $y=2$; $r_0=0$

How can we know? Let us enumerate all interleavings.

- $a, b, c, d$  
  $y=2$; $r_0=1$;
- $a, c, b, d$  
  $y=1$; $r_0=1$;
- $a, c, d, b$  
  $y=1$; $r_0=1$;
- $c, d, a, b$  
  $y=1$; $r_0=1$;
- $c, a, b, d$  
  $y=1$; $r_0=1$;
- $c, a, d, b$  
  $y=1$; $r_0=1$;

**Conclusion:** No SC execution would ever yield the output “$y=2$; $r_0=0$”.

Modern machines are not SC, how can we know?

```c
void *T0(void *p) {
    ctx_t *p = _p;
    common_t *q = p->common;
    q->x = 1;
    q->y = 1;
    return NULL;
}
void *T1(void *p) {
    ctx_t *p = _p;
    common_t *q = p->common;
    q->y = 2;
    int r0 = q->x;
    q->r0 = r0;
    return NULL;
}
```

```c
for ( ; ; ) {
    // Initialise
    common_t c; c.x = c.y = 0; ctx_t a0,a1;
    // Run
    a0.id = 0; a0.common = &c; create_thread(&th0,T0,&a0);
    a1.id = 1; a1.common = &c; create_thread(&th1,T1,&a1);
    join_thread(&th0);
    join_thread(&th1);
    // Collect results
    ... c.y ... c.r0
}
```

Naive testing, graphically

Let us run test $R$ on this machine (demo/01/naive_r.out)

Minimizing the impact of thread creation on cost

We perform size tests per thread creation, in arrays.

```c
void *T0(void *p) {
    ctx_t *p = _p;
    common_t *q = p->common;
    for (int k = 0 ; k < q->size ; k++) {
        q->x[k] = 1;
        q->y[k] = 1;
    }
    return NULL;
}
void *T1(void *p) {
    ctx_t *p = _p;
    common_t *q = p->common;
    for (int k = 0 ; k < q->size ; k++) {
        q->y[k] = 2;
        int r0 = q->x[k];
        q->r0[k] = r0;
    }
    return NULL;
}
```
Graphically

Let us synchronise iterations

```c
void *T0(void *p) {
    ctx_t *p = _p;
    common_t *q = p->common;
    for (int k = 0; k < q->size; k++) {
        wait_partner(&q->sync[k], k, 0);
        q->x[k] = 1;
        q->y[k] = 1;
    }
    return NULL;
}

void *T1(void *p) {
    ctx_t *p = _p;
    common_t *q = p->common;
    for (int k = 0; k < q->size; k++) {
        wait_partner(&q->sync[k], k, 1);
        q->y[k] = 2;
        int r0 = q->x[k];
        q->r0[k] = r0;
    }
    return NULL;
}
```

```c
inline static void wait_partner(volatile int *p, int k, int id) {
    if (k % 2 == id) {
        *p = 1; __sync_fetch_and_add(&p, 1);
    } else {
        while (*p == 0);
    }
}
```

Let us run test R on this machine (demo/01/loop_r.out)

Running n test instances at once

Why not run \( n = \lfloor a/2 \rfloor \) instances of R on a a-core machine?

- Much more outcomes on many-core machines per test run
- Important when one pays resources by chunks of say 32 cores.
- Makes noise and favors outcome variability.
The litmus tool

- Tests are written in assembler, for precision.

  X86 R
  {
  
  P0 | P1 ;
  MOV [x],$1 | MOV [y],$2 ;
  MOV [y],$1 | MOV EAX,[x] ;
  exists (y=2 /\ 1:EAX=0)

- Tests are compiled to C programs (with inline assembler):

  ```
  file.litmus
  litmus
  file.c
  gcc -pthread
  ```

  ```
  utils.c
  ```

  ```
  file.exe
  ```

  Demo in demo/02:

  ```
  % litmus -mach ./x86.cfg -o run R.litmus
  % cd run
  % make
  % ./R.exe -v -v
  ```

Building significant tests

Perfect! We know how to run tests on hardware, but what tests do we run?

We study relaxed memory models, that is relaxed w.r.t SC.

Hence, we focus on programs that have non-SC behaviours.

The question is: how do we generate such programs.

Let us study SC in detail first.

Some litmus settings

- Synchronisation
  - Loose synchronisation (we saw it).
  - Exact synchronisation with polling synchronisation barriers and time base.
  - Other, less useful, modes: POSIX thread barrier based and no synchronisation at all.

- Affinity: force threads to run on designated cores, or let the OS scheduler perform its job.

- Prefetching of flushing cache lines.
  - Automatic, depending on test.
  - Random.
  - Complete or none.

- Various scanning order of location arrays.
  - Random (by the means of a shuffled array of pointer).
  - Linear with a stride.

A complete testing campaign usually involves trying many settings (for instance, testing all strides from 1 to cache line size).

Back to our non-SC example

<table>
<thead>
<tr>
<th></th>
<th>( T_0 )</th>
<th>( T_1 )</th>
</tr>
</thead>
<tbody>
<tr>
<td>( a ) x ← 1</td>
<td>( c ) y ← 2</td>
<td></td>
</tr>
<tr>
<td>( b ) y ← 1</td>
<td>( d ) r0 ← x</td>
<td></td>
</tr>
</tbody>
</table>

Observed? \( y=2; r0=0 \)

All interleavings.

\[
\begin{aligned}
  a, b, c, d & \quad y=2; r0=1; \\
  a, c, b, d & \quad y=1; r0=1; \\
  a, c, d, b & \quad y=1; r0=1; \\
  c, d, a, b & \quad y=1; r0=1; \\
  c, a, b, d & \quad y=1; r0=1; \\
  c, a, d, b & \quad y=1; r0=1; \\
\end{aligned}
\]

We observe if \( b < c \) then \( y=2 \), if \( d < a \) then \( r0=0 \).
Let us be a bit more clever

<table>
<thead>
<tr>
<th>$T_0$</th>
<th>$T_1$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$(a) x ← 1$</td>
<td>$(c) y ← 1$</td>
</tr>
<tr>
<td>$(b) y ← 1$</td>
<td>$(d) r_0 ← x$</td>
</tr>
</tbody>
</table>

Collecting constraints on the scheduling order $\prec$:

We respect program order, thus $a < b$ and $c < d$.
We observe $r_0 = 0$, thus $d < a$.
We observe $y = 2$, thus $b < c$.

Hence we have a cycle in $\prec$, which prevents it from being an order!

$a < b < c < d < a \cdots$

Conclusion: No SC execution would ever yield the output "$y = 2; r_0 = 0;\$".

---

Example of $\xrightarrow{\text{co}}$

<table>
<thead>
<tr>
<th>$T_0$</th>
<th>$T_1$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$(a) x ← 1$</td>
<td>$(c) y ← 2$</td>
</tr>
<tr>
<td>$(b) y ← 2$</td>
<td>$(d) x ← 2$</td>
</tr>
</tbody>
</table>

Observe: $x; y$;

Example of $\xrightarrow{\text{co}}$

2+2W

<table>
<thead>
<tr>
<th>$T_0$</th>
<th>$T_1$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$(a) x ← 1$</td>
<td>$(c) y ← 1$</td>
</tr>
<tr>
<td>$(b) y ← 1$</td>
<td>$(d) x ← 2$</td>
</tr>
</tbody>
</table>

Observe: $x; y$;

---

Systematic approach

For a particular (partial) execution candidate (that is, for a set of events and a $\xrightarrow{\text{po}}$ relation) we assume two additional relations:

- **Read-from ($\xrightarrow{\text{rf}}$)**: Relates write events to read events that read the stored value (implicit initial writes).
  \[
  \forall r, \exists ! w, \ x \xrightarrow{\text{rf}} r
  \]

- **Coherence ($\xrightarrow{\text{co}}$)**: Relates write events to the same location.
  For any location $\ell$, the restriction of $\xrightarrow{\text{co}}$ to write events to location $\ell$ ($W_\ell$) is a total order.

Notice: To me, the very existence of $\xrightarrow{\text{co}}$ stems from the existence of a shared, coherent, memory — Given location $\ell$, there is exactly one memory cell whose location is $\ell$.

---

One more relation: $\xrightarrow{\text{fr}}$

The new relation $\xrightarrow{\text{fr}}$ (from read) relates reads to “younger writes” (younger w.r.t. $\xrightarrow{\text{co}}$).

\[ r \xrightarrow{\text{fr}} w \ \overset{\text{Def}}{=} w' \xrightarrow{\text{rf}} r \wedge w' \xrightarrow{\text{co}} w \]

This amounts to place a read into the coherence order of its location.

Given

\[
\begin{align*}
W_0 & \xrightarrow{\text{co}} W_1 & \ldots & \xrightarrow{\text{co}} W_n \\
\xrightarrow{\text{rf}} & \quad & \xrightarrow{\text{fr}}
\end{align*}
\]

We have

\[
\begin{align*}
W_0 & \xrightarrow{\text{co}} W_1 & \ldots & \xrightarrow{\text{co}} W_n \\
\xrightarrow{\text{rf}} & \quad & \xrightarrow{\text{fr}}
\end{align*}
\]
Playing with $\xrightarrow{fr}$

Particular, easy, case: a read from the initial state is in $\xrightarrow{fr}$ with writes by the program.

<table>
<thead>
<tr>
<th>MP</th>
<th>SB</th>
</tr>
</thead>
<tbody>
<tr>
<td>$T_0$</td>
<td>$T_1$</td>
</tr>
<tr>
<td>$(a) x \leftarrow 1$</td>
<td>$(c) r_0 \leftarrow y$</td>
</tr>
<tr>
<td>$(b) y \leftarrow 1$</td>
<td>$(d) r_1 \leftarrow x$</td>
</tr>
</tbody>
</table>

Observed? $r_0=1; r_1=0$

<table>
<thead>
<tr>
<th>a: Wx=1</th>
<th>b: Wy=1</th>
<th>c: Ry=1</th>
<th>d: Rx=0</th>
</tr>
</thead>
<tbody>
<tr>
<td>po</td>
<td>po</td>
<td>rf</td>
<td>rf</td>
</tr>
</tbody>
</table>

Observed? $r_0=0; r_1=0$

<table>
<thead>
<tr>
<th>a: Wx=1</th>
<th>b: Wy=1</th>
<th>c: Wy=1</th>
<th>d: Rx=0</th>
</tr>
</thead>
<tbody>
<tr>
<td>po</td>
<td>po</td>
<td>co</td>
<td>po</td>
</tr>
</tbody>
</table>

Introducing herd a memory model simulator

A model sc.cat:

```
% cat sc.cat
"Sequential consistency"
let com = rf | co | fr
acyclic (po | com) as hb
```

Running R on SC (demo in demo/02):

Test R Allowed
States 3
1:EAX=0; y=1;
1:EAX=1; y=1;
1:EAX=1; y=2;
No
Witnesses
Positive: 0 Negative: 3
Condition exists $(y=2 \land \neg 1:EAX=0)$
Observation R Never 0 3

**Notice:** Outcome 1:EAX=0; y=2; is forbidden by SC.

Second definition of SC

**Definition (SC 2)**

An execution is SC when:

$$\text{Acyclic}\left( \xrightarrow{rf} \cup \xrightarrow{co} \cup \xrightarrow{fr} \cup \xrightarrow{po} \right)$$

And of course:

**Theorem**

*The two definitions of SC are equivalent.*

Herd structure

- Generate all candidate executions, *i.e.* all possible $\xrightarrow{po}$, $\xrightarrow{rf}$ and $\xrightarrow{co}$ ($\xrightarrow{fr}$ deduced):

<table>
<thead>
<tr>
<th>a: Wx=1</th>
<th>c: Wy=2</th>
</tr>
</thead>
<tbody>
<tr>
<td>po</td>
<td>po</td>
</tr>
<tr>
<td>rf</td>
<td>po</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>a: Wx=1</th>
<th>c: Wy=2</th>
</tr>
</thead>
<tbody>
<tr>
<td>po</td>
<td>po</td>
</tr>
<tr>
<td>rfco</td>
<td>po</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>a: Wx=1</th>
<th>c: Wy=2</th>
</tr>
</thead>
<tbody>
<tr>
<td>po</td>
<td>po</td>
</tr>
<tr>
<td>co</td>
<td>po</td>
</tr>
</tbody>
</table>

- Apply model checks to each candidate execution.
Violations of SC

A cycle of $\text{po} \rightarrow \text{rf} \rightarrow \text{co} \rightarrow \text{fr} \rightarrow \text{des}$ describes a violation of SC.

From such a cycle, one may easily generate programs that potentially violate SC, and run them on an actual machine.

However, the cycle does not describe:

- How many threads are involved.
- How many memory locations are involved.

We now aim at:

- Extract a subset of significant cycles.
- Generate one program out of one cycle.

Simplifying cycles: all $\text{com}$ steps are external

Given a cycle, we consider all $\text{com}$ and $\hat{\text{com}}$ steps are external, that is source and target events are from pairwise distinct thread.

Given $e_1 \xrightarrow{\text{com}} e_2$, s.t. $e_1$ and $e_2$ are from the same thread:

- Either $e_1 \xrightarrow{\text{po}} e_2$ and we consider this $\text{po}$ step in the cycle, in place of the $\text{com}$ step.
- Or $e_2 \xrightarrow{\text{po}} e_1$ and we have a very simple cycle $e_2 \xrightarrow{\text{po}} e_1 \xrightarrow{\text{com}} e_2$.

Such cycles are "violations of coherence" (more on them later).

Notice: The same reasoning applies individual $\text{com}$ steps in composite $\hat{\text{com}}$.

Simplifying cycles — Locations

Cycle: $W \xrightarrow{\text{po}} W \xrightarrow{rf} R \xrightarrow{po} R \xrightarrow{fr} W \xrightarrow{po} W \xrightarrow{rf} R \xrightarrow{po} R \xrightarrow{fr}$

- One interpretation (four locations):
  a: $R_x=1$  c: $R_y=1$  e: $R_z=1$  g: $R_a=1$
  po  rf  po  rf  po
  b: $W_y=1$  d: $W_x=1$  f: $W_a=1$  h: $W_x=1$

- Another interpretation (two locations):
  a: $R_x=2$  c: $R_y=1$  e: $R_x=1$  g: $R_y=2$
  po  rf  po  rf  po
  b: $W_y=1$  d: $W_x=1$  f: $W_y=2$  h: $W_x=2$
The second interpretation is not “minimal”

Reminding the interpretation with two locations:

\[
\begin{align*}
  & a: Rx=2 & c: Ry=1 & e: Rx=1 & g: Ry=2 \\
  & \quad po & \quad po & \quad po & \quad po \\
  & b: Wy=1 & d: Wx=1 & f: Wy=2 & h: Wx=2
\end{align*}
\]

But, coherence \(\text{co}\) totally orders write events to a given location.

Let us choose: \(Wx1 \xrightarrow{\text{co}} Wx2:\)

\[
\begin{align*}
  & a: Rx=2 & c: Ry=1 & e: Rx=1 & g: Ry=2 \\
  & \quad po & \quad po & \quad po & \quad po \\
  & b: Wy=1 & d: Wx=1 & f: Wy=2 & h: Wx=2
\end{align*}
\]

We have a smaller cycle: \(d \xrightarrow{\text{co}} h \xrightarrow{\text{rf}} a \xrightarrow{\text{po}} b \xrightarrow{\text{rf}} c \xrightarrow{\text{po}} d\).

Choosing \(Wx2 \xrightarrow{\text{co}} Wx1\) would yield another smaller cycle.

Generally: do not repeat locations in cycles.

\[\text{... Simplifying cycles}\]

In a non SC execution we find:

\[\text{\textbullet ~ A violation of coherence, that is a cycle } e_1 \xrightarrow{\text{po}} e_2 \xrightarrow{\text{com}} e_1.\]

\[\text{\textbullet ~ Or a critical cycle that is:}\]

\[\text{\textbullet ~ The cycle alternates } \xrightarrow{\text{po}} \text{ steps and external } \xrightarrow{\text{com}} \text{ steps, with at least four steps.}\]

\[\text{\textbullet ~ The cycle passes through a given thread at most once.}\]

\[\text{\textbullet ~ All } \xrightarrow{\text{com}} \text{ steps have pairwise different locations.}\]

\[\text{\textbullet ~ The source and target of one given } \xrightarrow{\text{po}} \text{ steps have different locations.}\]

\[\text{Notice: ~ For a more formal presentation see D. Shasha and M. Snir Toplas 88 article, which introduced critical cycles.}\]

\[\text{Violations of coherence}\]

There are five such cycles, which can occur as the following executions:

\[
x \xrightarrow{\text{po}} \text{ conflicts with } \xrightarrow{\text{co}}, \xrightarrow{\text{rf}}, \xrightarrow{\text{fr}}, \xrightarrow{\text{po}}, \xrightarrow{\text{po}}, \xrightarrow{\text{rf}}, \xrightarrow{\text{fr}}, \xrightarrow{\text{po}}, \xrightarrow{\text{po}}, \xrightarrow{\text{rf}}, \xrightarrow{\text{fr}}.
\]

\[\text{CoWW}\]

\[\text{CoRW1}\]

\[\text{CoWR}\]

\[\text{CoRW2}\]

\[\text{CoRR}\]

\[\text{Notice: ~ For a more formal presentation see D. Shasha and M. Snir Toplas 88 article, which introduced critical cycles.}\]
Building another text, non-trivial coherence

Test R

\[
\begin{align*}
\text{fre} & \rightarrow (a):W[x]1 \quad \text{po} \rightarrow (b):W[y]1 \quad \text{com} \rightarrow (c):W[y]2 \quad \text{po} \rightarrow (x):R[x]0 \\
\end{align*}
\]

Here, coherence order is \(0 \rightarrow 1 \rightarrow 2\), it suffices to read the final value.

Longer coherence orders command other techniques, for instance adding an observer thread.

A first tool: diyone

Generating WRC:

\[
\begin{align*}
\% \text{ diyone -arch X86 Rfe Pod** Rfe Pod** Fre X86 A} \\
\text{PO} & | \text{P1} | \text{P2} ; \\
\text{MOV [x]},1 & | \text{MOV EAX,[x]} | \text{MOV EAX,[y]} ; \\
\text{MOV [y]},1 & | \text{MOV EBX,[x]} ; \\
\text{exists (1:EAX=1} & | 2:EAX=1 \\
\text{2:EBX=0)} \\
\text{Doing the same for ARM is as simple as:} \\
\% \text{ diyone -arch ARM Rfe Pod** Rfe Pod** Fre ARM A} \\
\{ & \%x0=x; \%x1=x; \%y1=y; \%y2=y; \%x2=x; \\
\text{PO} & | \text{P1} | \text{P2} ; \\
\text{MOV R0,#1} & | \text{LDR R0,\%x1} | \text{LDR R0,\%y2} ; \\
\text{STR R0,\%x0} & | \text{MOV R1,#1} | \text{LDR R1,\%x2} ; \\
\text{STR R1,\%y1} & | \text{STR R1,\%y1} ; \\
\text{exists (1:R0=1} & | 2:R0=1 \\
\text{2:R1=0) }
\end{align*}
\]
Tool diyone, generating R

% diyone -arch X86 PodWW Wse PodWR Fre
X86 A
"PodWW Wse PodWR Fre"
{ }

P0 | P1 ;
MOV [x],$1 | MOV [y],$2 ;
MOV [y],$1 | MOV EAX,[x] ;
exists (y=2 /\ 1:EAX=0)

Notice: We wrote PodWW, PodWR. The vocabulary of Candidate Relaxations is quite rich:

▶ Internal communications Rfi, Fri, Wsi.
▶ po→ edges with identical target and source locations: PosRR, etc.
▶ Dependencies (DpAddrdW, etc.). fences (MFencedWR, etc.)

Generating two-threads SC violations

The tool diy generates cycles (and tests) from a vocabulary of CR. It can be configured for the two threads case as follows:

-arch X86  # target architecture
-safe Pod**,Rfe,Fre,Wse # vocabulary
-nprocs 2  # 2 procs
-size 4  # max size of cycle (2 X nprocs)
-num false # for naming tests

Demo in demo/03.

% diy -conf 2.conf
Generator produced 6 tests
% ls
2+2W.litmus 2.conf @all LB.litmus
MP.litmus R.litmus SB.litmus S.litmus
% diy -conf 4.conf
Generator produced 68 tests...

Application, testing non-SC executions for two threads

Cycle → execution → program + final condition.

All (critical) cycles for two threads: six cycles.

2+2W  po → co → po → co
LB  po → rf → po → rf
MP  po → rf → po → fr
R  po → co → po → fr
S  po → fr → po → co
SB  po → fr → po → fr

Save of coherence violation, any non-SC execution on two threads includes one of the above six cycles.

Hence, testing the six tests built from the six cycles gives reasonable coverage of possible SC violation on two threads. (Notice: coherence violations neglected).

Demo 03 continued, running the tests

Compiling:
% litmus -mach ./x86.cfg src/@all -o run
% make -C run -j 4

Running:
% cd run
% sh run.sh > X.00

Analysis:
% grep Observation X.00
Observation R Sometimes 79 1999921
Observation MP Never 0 2000000
Observation 2+2W Never 0 2000000
Observation S Never 0 2000000
Observation SB Sometimes 1194 1998806
Observation LB Never 0 2000000
Results for running the six tests on this machine

<table>
<thead>
<tr>
<th>Test</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>a: Wx=1 c: Wy=2</td>
<td>R: No</td>
</tr>
<tr>
<td>b: Wx=1 d: Rx=0</td>
<td>S: No</td>
</tr>
<tr>
<td>a: Wx=2 c: Ry=1</td>
<td>SB: No</td>
</tr>
<tr>
<td>b: Ry=0 d: Rx=0</td>
<td></td>
</tr>
<tr>
<td>a: Wx=1 c: Wx=1</td>
<td></td>
</tr>
<tr>
<td>b: Ry=0 d: Rx=0</td>
<td></td>
</tr>
<tr>
<td>a: Wx=2 c: Wx=2</td>
<td></td>
</tr>
<tr>
<td>b: Ry=0 d: Rx=0</td>
<td></td>
</tr>
<tr>
<td>2+2W: No</td>
<td></td>
</tr>
</tbody>
</table>

TSO — The Model of X86 machines

The write buffer explains how "reads can pass over writes".

Axiomatic TSO

- Remember SC:
  \[
  \text{Acyclic} \left( \text{rf} \cup \text{co} \cup \text{fr} \cup \text{po} \right)
  \]

- A model for herd, our generic simulator:
  let \( \text{ppo} = \text{po} \) # \( \text{ppo} \) stands for 'preserved program-order'
  let \( \text{com-hb} = \text{fr} \cup \text{rf} \cup \text{co} \) # All communications create order
  acyclic (ppo | com-hb)

- In TSO:
  - Write-to-read does not create order:
    let \( \text{ppo} = \text{RM(po)} \cup \text{WW(po)} \) # WR(po) omitted
  - Local reads do not create order:
    let \( \text{com-hb} = \text{rf} \cup \text{fr} \cup \text{co} \) # rfi omitted

- TSO “happens-before” (hb) check:
  acyclic (ppo | com-hb | mfence) as hb

Notice: Relations are between the points in time where a load binds its value and where a written value reaches memory.

Results for running the six test on the model
Internal \( \rightarrow_{\text{rf}} \) (\( \rightarrow_{\text{rfi}} \)) are not in HB

<table>
<thead>
<tr>
<th>SB+rfi-pos</th>
<th>( T_0 )</th>
<th>( T_1 )</th>
</tr>
</thead>
<tbody>
<tr>
<td>(a) x ← 1</td>
<td>(d) y ← 1</td>
<td></td>
</tr>
<tr>
<td>(b) r0 ← x</td>
<td>(e) r2 ← y</td>
<td></td>
</tr>
<tr>
<td>(c) r1 ← y</td>
<td>(f) r3 ← x</td>
<td></td>
</tr>
</tbody>
</table>

Observed? \( r0=1; r1=0; r2=1; r3=0; \)

\[ \text{SB+rfi-pos: Ok} \]

We are not done yet...

Our TSO model:

\[
\text{let ppo = RM(po) | WW(po) # WR(po) omitted} \\
\text{let com-hb = rfe | fr | co # rfi omitted} \\
\text{acyclic (ppo | com-hb)} \\
\text{show ppo | com-hb as hb}
\]

Allows two violations of coherence:

<table>
<thead>
<tr>
<th>a: Rx=1</th>
<th>a: Wx=1</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \rightarrow_{\text{po}} )</td>
<td>( \rightarrow_{\text{po}} )</td>
</tr>
<tr>
<td>b: Wx=1</td>
<td>rf: Rx=0</td>
</tr>
</tbody>
</table>

CoRW1

\[
\rightarrow_{\text{rf}} \text{ not in }^{\text{hb}} \quad W \rightarrow_{\text{po}} R \text{ not in }^{\text{hb}}
\]

Those behaviours must be rejected by our TSO model.

Correct axiomatic TSO

We add a specific uniproc check to rule out coherence violations:

\[
\text{Irreflexive } (\text{po-loc}; \text{com}^+) \text{ as uniproc}
\]

Where \( \rightarrow_{\text{po-loc}} \) is \( \rightarrow \) between accesses to the same memory location.

irreflexive (po-loc; com+) as uniproc ... .

In the TSO case we can “optimise”:

irreflexive rf;RW(po-loc)
irreflexive fr;WR(po-loc)

because the other coherence violations are rejected by the HB check.
A word on UNIPROC

From cycle analysis, we have the attractive definition (since relying on
local action of the core and on the existence of coherence orders):

**Definition (Uniproc 1)**

Program order \( \overset{po}{\rightarrow} \) does not contradict communication \( \overset{com^+}{\rightarrow} \).

There is another definition “SC per location”’. (Jason F. Cantin, Mikko
H. Lipasti, James E. Smith ACM Symposium on Parallel Algorithms and
Architectures 2004).

**Definition (Uniproc 2)**

Relation \( \overset{po-loc}{\rightarrow} \cup \overset{com}{\rightarrow} \) is acyclic.

Definitions are equivalent.

It suffices to show that the existence of a cycle in \( \overset{po-loc}{\rightarrow} \cup \overset{com}{\rightarrow} \) implies the
existence of a coherence violation (i.e. a cycle \( e_1 \overset{po}{\rightarrow} e_2 \overset{com}{\rightarrow} e_1 \)).

---

**Equivalence of the two UNIPROC definitions**

Proof is easy from “Identical locations” lemma.

Consider a cycle in \( \overset{po}{\rightarrow} \cup \overset{com}{\rightarrow} \).

- If there exists a \( e_1 \overset{po}{\rightarrow} e_2 \) step s.t. \( e_2 \overset{com}{\rightarrow} e_1 \), then we are done.
- Otherwise, for each \( e_1 \overset{po}{\rightarrow} e_2 \) step:
  - Either, \( r_1 \overset{po}{\rightarrow} r_2 \), with \( w \overset{rf}{\rightarrow} r_1 \) and \( w \overset{rf}{\rightarrow} r_2 \). We short-circuit the
    \( \overset{po}{\rightarrow} \) step, replacing \( w \overset{rf}{\rightarrow} r_1 \overset{po}{\rightarrow} r_2 \) by \( w \overset{rf}{\rightarrow} r_2 \).
  - Or, \( e_1 \overset{com}{\rightarrow} e_2 \). We replace the \( \overset{po}{\rightarrow} \) step by \( \overset{com}{\rightarrow} \) steps.

As a result we have a cycle in \( \overset{com}{\rightarrow} \), which is impossible.

---

**Consequence of \( \overset{co}{\rightarrow} \) ordering writes**

**Lemma (Identical locations)**

Let \( e_1, e_2 \) be two different events with the same location,

1. either \( e_1 \overset{com}{\rightarrow} e_2 \),
2. or \( e_2 \overset{com}{\rightarrow} e_1 \),
3. or \( w \overset{rf}{\rightarrow} e_1 \) and \( w \overset{rf}{\rightarrow} e_2 \).

Case analysis:

- \( w_1, w_2 \), then either \( w_2 \overset{co}{\rightarrow} w_1 \) or \( w_2 \overset{co}{\rightarrow} w_1 \) (total order).
- \( r_1, r_2 \), let \( r_1 \overset{rf}{\rightarrow} r_1 \) and \( w \overset{rf}{\rightarrow} r_2 \). Then, either \( w_1 = w_2 \) and we are
  in case 3; or (for instance) \( w_1 \overset{co}{\rightarrow} w_2 \) and we have \( w_1 \overset{fr}{\rightarrow} w_2 \overset{fr}{\rightarrow} r_2 \).
- \( r_1, w_2 \), let \( w_1 \overset{rf}{\rightarrow} r_1 \). Then, either \( w_1 = w_2 \) and \( w_2 \overset{rf}{\rightarrow} r_1 \); or
  \( w_1 \overset{co}{\rightarrow} w_2 \) and \( r_1 \overset{fr}{\rightarrow} w_2 \); or \( w_2 \overset{co}{\rightarrow} w_1 \) and \( w_2 \overset{co}{\rightarrow} r_1 \).

**Corollary:** \( \overset{com}{\rightarrow} \) is acyclic.

---

**A relaxed shared memory computer**

---

More or less visible to user code:

- **Cores:**
  - Out of order execution
  - Branch speculation
- **Memory**
  - Physically distributed
  - Caches
Situation of (our) ARM/Power models

- **Architecture public reference**: Informal, cannot clearly explain how fences restore SC for instance.

- **Simple, global-time model**: (CAV'10) too relaxed. It remains useful as it supports simple reasoning on SC-violations (CAV'11).

- **Operational model**: (PLDI’11) more precise, developed with IBM experts. It is quite complex, and the simulator is very slow.

- **Multi-event axiomatic model**: (CAV’12) more precise (equivalent to PLDI’11), uses several events per access.

- **Single-event axiomatic model**: (...) more precise (proved to be more relaxed than PLDI’11, experimentally equivalent). A more simple axiomatic model.

Joint work with (in order of appearance) Jade Alglave, Susmit Sarkar, Peter Sewell, Derek Williams, Kayvan Memarian, Scott Owens, Mark Batty, Sela Mador-Haim, Rajeev Alur, Milo M. K. Martin and Michael Tautschnig.

Some issues for ARM/Power

- No simple preserved-program-order. More precisely, po \( \rightarrow \) will now account for core constraints, such as dependencies.

- Communication relations alone do not define happen-before steps.

- A variety of memory fences: lightweight (Power lwsync) and full (Power sync).

An experiment on ARM/Power

Consider test MP:

<table>
<thead>
<tr>
<th></th>
<th>( T_0 )</th>
<th>( T_1 )</th>
</tr>
</thead>
<tbody>
<tr>
<td>((a)) x ← 1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>((b)) y ← 1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>((c)) r0 ← y</td>
<td></td>
<td></td>
</tr>
<tr>
<td>((d)) r1 ← x</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Observed? \( r_0 = 1; r_1 = 0 \)

We know that the test is Ok (observed, valid) on ARM/Power, what does it take (amongst fences, dependencies,) to make the test No (unobserved, invalid)?

- Fences: dsb, dab, isb (ARM); sync, lwsync, isync (Power).
- Dependencies: address, data, control, control+isb/isync.

Two-threads SC violation for ARM

Generating tests is as simple as:

```
% diy -conf 2.conf -arch ARM
```

With the same configuration file 2.conf as for X86. Then, compile (in two steps, generate C locally, compile it on target machine), run and...

Observation R Sometimes 5722 1994278
Observation MP Sometimes 3571 1996429
Observation 2+2W Sometimes 17439 1982561
Observation S Sometimes 7270 1992730
Observation SB Sometimes 9788 1990212
Observation LB Sometimes 4782 1995218

All Non-SC behaviours observed!

No hope to define po \( \rightarrow \) as simply as for TSO.
Dependencies (Power)

Address dependency:
\[
\begin{align*}
  r1 & \leftarrow x \\
  r2 & \leftarrow t[r1]
\end{align*}
\]

lwz r1,0(r8) \ # r8 contains the address of ‘x’
slwi r7,r1,2 \ # sizeof(int) = 4
lwzx r2,r7,r9 \ # r9 contains the address of ‘t’

Data dependency:
\[
\begin{align*}
  r1 & \leftarrow x \\
  y & \leftarrow r1+1
\end{align*}
\]

lwz r1,0(r8) \ # r8 contains the address of ‘x’
addi r2,r1,1
stw r2,0(r9) \ # r9 contains the address of ‘y’

Control dependency: (+isync)
\[
\begin{align*}
  r1 & \leftarrow x \\
  \text{if } r1=0 \text{ then} \\
  \text{(isync)} \\
  y & \leftarrow 1
\end{align*}
\]
lwz r1,0(r8)
cmpi r1,0
bne L1
li r2,1
stw r2,0(r9)
L1:

Generating tests (ARM), yet another tool: diycross

Generating tests with diycross (demo in demo/04):
\%
diycross -arch ARM
PodWW,DMBdWW,DSBdWW,ISBdWW
Rfe
PodRR,DpCtrl1dR,DpCtrlIsbdR,DpAddrdR,DMBdRR,DSBdRR,ISBdRR
Fre
Generator produced 28 tests

- One generates MP as diyone PodWW Rfe PodRR Fre
- diycross \( r_1, \ldots, r_{N_1}, \ldots, r_{N_M} \), generates the \( N_1 \times \cdots \times N_M \) cycles \( r_{k_1} \cdots r_{k_1} \cdots r_{k_M} \) by cross-producting the given CR list arguments.

This generates some variations in the MP family.

We then compile and run, and...

Optimal fencing for the 6 two-threads tests (Power)

Optimal fencing/dependencies for MP

![Optimal fencing/dependencies for MP](image)

- a: Wx=1 \arrow{c}{co_r} \ c: Wy=2
- b: Wy=1 \arrow{d}{addr} \ d: Rx=0
- S+lsync+addr
- R+synchs
- a: Wx=1 \arrow{rff}{frf} \ c: Wy=1
- b: Ry=0 \arrow{d}{addr} \ d: Rx=0
- SB+synchs
- MP+lsync+addr
- a: Rx=1 \arrow{rff}{frf} \ c: Ry=1
- b: Wy=1 \arrow{d}{addr} \ d: Wx=1
- LB+addr
- 2+2W+lsyncs

- a: Wx=2 \arrow{rff}{frf} \ c: Ry=1
- b: Wy=1 \arrow{d}{addr} \ d: Wx=1
- 62/74
Some observations

In the previous slide we considered increasing power (and cost):

\[ \text{addr} < \text{lwsync} < \text{sync} \]

Then:

- Dependencies (address) are sufficient to restore order from reads to writes and reads in two-threads examples (but...)
- Fences restore order from writes to write and reads.
- Full fence (\text{sync}) is required from write to read.
- When to use the lightweight fence between writes is complex:

\[ 2+2 \text{W+lwsyncs} \text{ vs. } R+\text{lwsync+sync} \]

<table>
<thead>
<tr>
<th>\text{lb+datas}</th>
<th>\text{T}_0</th>
<th>\text{T}_1</th>
</tr>
</thead>
<tbody>
<tr>
<td>(a) \text{r0} \leftarrow \text{x}</td>
<td>(c) \text{r1} \leftarrow \text{y}</td>
<td></td>
</tr>
<tr>
<td>(b) \text{y} \leftarrow \text{r0}</td>
<td>(d) \text{x} \leftarrow \text{r1}</td>
<td></td>
</tr>
</tbody>
</table>

Observes? \text{r0}=42; \text{r1}=42.

- a: \text{Rx}=42
- b: \text{Wy}=42
- c: \text{Ry}=42
- d: \text{Wx}=42

No

Dependencies from reads not always enough!

Consider test \text{WRC+data+addr}:

<table>
<thead>
<tr>
<th>\text{WRC}</th>
<th>\text{T}_0</th>
<th>\text{T}_1</th>
<th>\text{T}_2</th>
</tr>
</thead>
<tbody>
<tr>
<td>(a) \text{x} \leftarrow 1</td>
<td>(b) \text{r0} \leftarrow \text{x}</td>
<td>(d) \text{r1} \leftarrow \text{y}</td>
<td></td>
</tr>
<tr>
<td>(c) \text{y} \leftarrow 1</td>
<td>(e) \text{r2} \leftarrow \text{x}</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Observed? \text{r0}=1; \text{r1}=1; \text{r2}=0

- a: \text{Wx}=1
- b: \text{Rx}=1
- c: \text{Wy}=1
- d: \text{Ry}=1
- e: \text{Rx}=0

\text{WRC+data+addr}

Behaviour observed on Power 6 and 7 (not on ARM, but documentation allows it).

Stores are not “multi-copy atomic” \text{T}_0 and \text{T}_1 share a private buffer/cache/memory (e.g. a cache in SMT context). \text{T}_2 “does not see” the store by \text{T}_0, when \text{T}_1 does.

Restoring SC for \text{WRC}

Use a lightweight fence on \text{T}_1:

<table>
<thead>
<tr>
<th>\text{T}_0</th>
<th>\text{T}_1</th>
<th>\text{T}_2</th>
</tr>
</thead>
</table>

\[ a: \text{Wx}=1 \quad b: \text{Rx}=1 \quad d: \text{Ry}=1 \]

<table>
<thead>
<tr>
<th>\text{lwsync}</th>
<th>\text{rfr}</th>
<th>\text{data}</th>
<th>\text{addr}</th>
</tr>
</thead>
<tbody>
<tr>
<td>c: \text{Wy}=1</td>
<td>e: \text{Rx}=0</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

\text{WRC+lwsync+addr}

Observation: The fence orders the writes \text{a} (by \text{T}_0) and \text{c} (by \text{T}_1) for any observer (here \text{T}_2).
Another case of unsufficient dependencies

Consider test IRIW+addr:

<table>
<thead>
<tr>
<th></th>
<th>T₀</th>
<th>T₁</th>
<th>T₂</th>
<th>T₃</th>
</tr>
</thead>
<tbody>
<tr>
<td>(a)</td>
<td>x ← 1</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>(b)</td>
<td>r₀ ← x</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>(c)</td>
<td>r₁ ← y</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>(d)</td>
<td>y ← 1</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>(e)</td>
<td>r₂ ← y</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>(f)</td>
<td>r₃ ← x</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Observed? r₀=1; r₁=0; r₂=1; r₃=0;

a: Wₓ=1  b: Rₓ=1  d: Wᵧ=1  e: Rᵧ=1

Restoring SC for IRIW

Use a full fence on T₁ and T₂:

a: Wₓ=1  b: Rₓ=1  d: Wᵧ=1  e: Rᵧ=1

Propagation: Full fences order all communications.

IRIW+syncs

Stores are not “multi-copy atomic”: T₀ and T₁ have a private buffer/cache/memory, T₂ and T₃ also have one.

Relation summary

Communication relations:

- **Read-from:** w ⇝ₕ r, with loc(w) = loc(r), val(w) = val(r).
- **Coherence:** w ⇝ₚ r’, with loc(w) = loc(r’) = x. Total order for given x: hence “coherence orders”.
- **We deduce from-read:** r ⇝ₕ w, i.e w’ ⇝ₜ r and w’ ⇝ₚ w.
- **We distinguish internal (same proc, rfi ⇝ₜ, coi ⇝ₚ, fri ⇝ₜ) and external (different procs, rfe ⇝ₜ, coe ⇝ₚ, fre ⇝ₜ) communications.**

“Execution” relations

- **Program order:** e₁ ⇝ₕ e₂, with proc(e₁) = proc(e₂).
- **Same location program order:** e₁ ⇝ₚ e₂.
- **Preserved program order:** e₁ ⇝ₚ e₂, with ppo ⇝ₚ. Computed from other relations.
- **Fences:** effective strong and lightweight fences in between events strong and light. Effective means that for instance w ¹wsync r does not implies w ¹light r.

A model in four checks

UNIPROC

acyclic poloc | com as uniproc

HB

let fence = strong | light
let hb = ppo | fence | rfe
acyclic hb

OBSERVATION We now define the effect of fences (any fence) for ordering writes:

let propbase = (WW(fence)|(rfe;RW(fence)));hb*
irreflexive fre;propbase as observation

PROPAGATION Strong fences wait for all communications.

let propstrong = com*; propbase*; strong; hb*
let prop = WW(propbase)|(com*;propbase*;strong;hb*)
acyclic co | prop as propagation
ARM/Power preserved program order

Rather complex, results from a two events per access analysis (cf. CAV’12).

(* Utilities *)
let dd = addr | data let rdw = po-loc & (fre;rfe) let detour = po-loc & (coe ; rfe) let addrpo = addr;po

(* Initial value *)
let ci0 = ctrlisync | detour let ii0 = dd | rfi | rdw let cc0 = dd | po-loc | ctrl | addrpo let ic0 = 0

(* Fixpoint from i -> c in instructions and transitivity *)
let rec ci = ci0 | (ci;ii) | (cc;ci) and ii = ii0 | ci | (ic;ci) | (ii;ii) and cc = cc0 | ci | (ci;ic) | (cc;cc) and ic = ic0 | ii | cc | (ic;cc) | (ii ; ic)

let ppo = RW(ic) | RR(ii)

Can be limited to dependencies...

How good is our model?

Is it sound?
- A proof: any behaviour allowed is also allowed by the operational model of PLDI’11.
- Experiments
  - Soundness w.r.t. hardware (ARM being a bit problematic because of acknowledged read-after-read hazard).
  - Experimental equivalence with our previous models, saved from current debate on some subtle semantical point for lwsync.

In any case:
- Simulation is fast (∗1000 w.r.t. PLDI’11) (∗10 w.r.t. CAV’12).
- The existence of four checks UNIPROC, HB OBSERVATION and PROPAGATION stand on firm bases.
- The semantics of strong fences also does.
- The model and simulator (i.e. herd) are flexible, one easily change a few relations (e.g. ppo→, or the semantics of weak fences).

A test of coherence violation

Our setting also finds bugs...

The following execution:

\[
\begin{align*}
 a: & Wz=1 \\
 b: & Wy=1 \\
 c: & Wy=2 \\
 d: & Wz=1 \\
 e: & Rz=1 \\
 f: & Wx=1 \\
\end{align*}
\]

is observed on all (tested) ARM machines. It features a **CoRR-style coherence violation** (i.e. \( \xrightarrow{\text{po}} \)) contradicts \( \xrightarrow{\text{fr}}; \xrightarrow{\text{fr}} \)).