Episode III in our multiprocessing miniseries.
Relaxed memory models.
What I really wanted here was an elephant with sunglasses relaxing
On a beach, but I couldn’t find one.
Sequential Consistency: A Memory Model

Sequential Consistency =
 arbitrary order-preserving interleaving
 of memory references of sequential programs

SC is easy to understand but architects and
 compiler writers want to violate it for performance

Mark Hill written a paper which essentially says “why break your back for
20%”. Actually people are out there breaking their backs for 1% in
architecture these days.
Architectural optimizations that are correct for uniprocessors often result in a new memory model for multiprocessors.

This means that we are relaxing the ordering or relaxing atomicity.
Relaxed Models

• What orderings among reads and writes performed by a single processor are preserved by the model?
  - $R \rightarrow R, R \rightarrow W, W \rightarrow W, W \rightarrow R$
  - dependence if they are to the same address

• If there is a dependence, then program semantics demand that operations be ordered

• If there is no dependence, the memory consistency model determines what orders must be preserved
  - Relaxed model may allow an operation executed later to complete first
Memory Fences & Weak Memory Models

Processors with relaxed or weak memory models need memory fence instructions to force serialization of memory accesses

- In SC there is an implicit fence at each memory operation

Processors with relaxed memory models:
- Sparc V8 (TSO, PSO): Membar
- PowerPC (WO): Sync, EIEIO

Memory fences are expensive operations, however, one pays for serialization only when it is required
Ensures that tail pointer is not updated before X has been stored.

Ensures that R is not loaded before x has been stored.
**Data-Race Free Programs**
(a.k.a. Properly Synchronized Programs)

<table>
<thead>
<tr>
<th>Process 1</th>
<th>Process 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>Acquire(mutex);</td>
<td>Acquire(mutex);</td>
</tr>
<tr>
<td>&lt; critical section &gt;</td>
<td>&lt; critical section &gt;</td>
</tr>
<tr>
<td>Release(mutex);</td>
<td>Release(mutex);</td>
</tr>
</tbody>
</table>

Synchronization variables (e.g., `mutex`) are separate from data variables
Accesses to writable shared data variables are protected in critical regions
⇒ no data races except for locks
*Formal definition is elusive*

In general, it cannot be proven if a program is data-race free.

Nondeterminator.
Fences in Data-Race Free Programs

Process 1
...
Acquire(mutex);
membar;
< critical section >
membar;
Release(mutex);

Process 2
...
Acquire(mutex);
membar;
< critical section >
membar;
Release(mutex);

Relaxed memory models allow reordering of instructions by the compiler or the processor as long as the reordering is not done across a fence

What about speculation and prefetching?

Processor should not speculate or prefetch across fences.
Total Store Order (TSO)
IBM370, DECVAX

- Eliminates the order $W(a) \rightarrow R(b) \quad a \neq b$
- Advantage?

SC
\[\begin{align*}
B &= \text{acquire (S)} \\
C &= \text{release (S)} \\
E &= \\
F &=
\end{align*}\]

TSO
\[\begin{align*}
B &= \text{acquire (S)} \\
C &= \text{release (S)} \\
E &= \\
F &=
\end{align*}\]

Allows the buffering of writes with bypassing by reads, which occurs whenever the processor allows a read to proceed before it guarantees that an earlier write by the processor has been seen by all the other processors.
**TSO vs. SC**

Initially $x = \text{old}$, $y = \text{old}$

Processor $P_1$  
$x = \text{new};$
$y_{\text{copy}} = y;$

Processor $P_2$  
$y = \text{new};$
$x_{\text{copy}} = x;$

*Under SC what values can $x_{\text{copy}}$ and $y_{\text{copy}}$ get?*

*Under TSO what values can $x_{\text{copy}}$ and $y_{\text{copy}}$ get?*

TSO both can get old values.

SC at least one has to get the value of new.
Partial Store Ordering (PSO)
SPARC

- Also eliminates the order $W(a) \rightarrow W(b) \quad a \neq b$
- Advantage?

TSO

\begin{align*}
A & \rightarrow B \\
& \xrightarrow{\text{acquire (S)}} C = D \\
& \xrightarrow{\text{release (S)}} E = F
\end{align*}

PSO

\begin{align*}
A & \rightarrow B \\
& \xrightarrow{\text{acquire (S)}} C = D \\
& \xrightarrow{\text{release (S)}} E = F
\end{align*}

Allows pipelining or overlapping of write operations, rather than forcing one operation to complete before another.
Weak Ordering
POWERPC

• Also eliminates the orders $R(a) \rightarrow R(b)$  $a \neq b$
and $R(a) \rightarrow W(b)$  $a \neq b$

• Need non-blocking reads to exploit relaxation

Non-blocking reads, doesn’t help too much.
Release Consistency
Alpha, MIPS

- Read/write that precedes acquire need not complete before acquire, and read/write that follows a release need not wait for the release.

Weakest of the memory models used these days.
**Release Consistency Example**

Initially \( \text{data} = \text{old} \)

Processor \( P_1 \)

\[
\text{data} = \text{new};
\]

flag = SET;

Processor \( P_2 \)

\[
\text{while}(\text{flag} \neq \text{SET}) \{ \}
\]

\[
\text{data}_\text{copy} = \text{data};
\]

*How do we ensure that \( \text{data}_\text{copy} \) is always set to \( \text{new} \)?
Weaker Memory Models

- Hard to understand and remember
- Unstable - Modèle de l’année

Weak ordering. Interaction with cache coherence. Weak atomicity.
Why SC is not the right model for compilers

• Intermediate representation is intrinsically a partial order (data-flow graph)
  ⇒ expose scope for instruction reordering to the underlying architecture

• Load/Store atomicity forces compilers to over-specify requirements for completion of operations
  ⇒ expose cache coherence actions
The CRF Model  
X. Shen, Arvind, L. Rudolph (1999)

Exposes

- *data caching via semantic caches*

  \[
  \text{Store}(a,v) \equiv \text{StoreL}(a,v); \text{Commit}(a) \\
  \text{Load}(a) \equiv \text{Reconcile}(a); \text{LoadL}(a)
  \]

- *instruction reordering* (controllable via Fence)
CRF: Load Local & Store Local

- **LoadL** reads from the sache if the address is cached
- **StoreL** writes into the sache and sets the state to **Dirty**
**CRF: Commit and Reconcile**

- **Commit** completes if the address is not cached in the Dirty state
- **Reconcile** completes if the address is not cached in Clean
CRF: Background Operations

- Cache (retrieve) a copy of an uncached address from memory
- Writeback a Dirty copy to memory and set its state Clean
- Purge a Clean copy
**CRF: Fences**

Instructions can be reordered except for

- Data dependence
- StoreL(a,v); Commit(a);
- Reconcile(a); LoadL(a);

\[
\text{Reconcile}(a_1); \\
\text{LoadL}(a_1); \\
\text{Fence}_{rr}(a_1, a_2) \\
\text{Reconcile}(a_2); \\
\text{LoadL}(a_2);
\]

\[
\text{Fence}_{wr}; \text{Fence}_{rw}; \text{Fence}_{ww};
\]
**Producer-Consumer Synchronization**

- Break down the synchronization equally between the producer and consumer
- Semantically, memory behaves as the *rendezvous* between processors
  \[\Rightarrow\text{no operation involves more than one cache}\]
A CRF protocol is automatically a protocol for any memory model whose programs can be translated into CRF programs.
Translating SC into CRF

Initially $a = 0$, $flag = 0$

**Processor 1**

- `Store(a,10);`
- `Store(flag,1);`

**Processor 2**

- `L: r_1 = Load(flag);`
- `Jz(r_1,L);`
- `r_2 = Load(a);`
Translating SC into CRF

Processor 1

Store(a,10);
Fence_{ww}(a, flag);
Store(flag,1);

Processor 2

L: \( r_1 = \text{Load(flag)}; \)
Jz(r_1,L);
Fence_{rr}(flag, a);
\( r_2 = \text{Load(a)}; \)

Weak ordering
Translating SC into CRF

Processor 1

StoreL(a, 10);
Commit(a);
Fencewu(a, flag);
StoreL(flag, 1);
Commit(flag);

Processor 2

L: Reconcile(flag);
r1 = LoadL(flag);
Jz(r1, L);
Fencewr(flag, a);
Reconcile(a);
r2 = LoadL(a);