EECS 570
Lecture 13
Memory Consistency

Winter 2020
Prof. Satish Narayanasamy

http://www.eecs.umich.edu/courses/eecs570/

Memory Consistency
Memory Consistency Model

A memory (consistency) model specifies the order in which memory accesses performed by one thread become visible to other threads in the program.

It is a contract between the hardware of a shared memory multiprocessor and the successive programming abstractions (instruction set architecture, programming languages) built on top of it.

• Loosely, the memory model specifies:
  ◆ the set of legal values a load operation can return
  ◆ the set of legal final memory states for a program
Who cares about memory models?

- Programmers want:
  - A framework for writing correct parallel programs
  - Simple reasoning - “principle of least astonishment”
  - The ability to express as much concurrency as possible

- Compiler/Language designers want:
  - To allow as many compiler optimizations as possible
  - To allow as much implementation flexibility as possible
  - To leave the behavior of “bad” programs undefined

- Hardware/System designers want:
  - To allow as many HW optimizations as possible
  - To minimize hardware requirements / overhead
  - Implementation simplicity (for verification)

We will consider all three perspectives
Uniprocessor memory model

- Loads return value of nearest preceding matched store in <p
  - Need to make partial overlaps work
  - Probably need some special cases for I/O
  - Otherwise, any sort of reordering goes!

- Programmer’s perspective:
  - Generally, no way to tell what order things actually happened

- Compiler’s perspective
  - “as-if” rule – any optimization is legal as long as it produces the same output “as-if” executed in program order
  - No “random” changes to memory values (except volatile)

- HW perspective
  - Out-of-order, store buffers, speculation all ok
  - Order only needed per-address, enforced via LSQ
Language-Level
DRF-0 Vs SC
Memory Model
Program Order

A ; B

a thread

Execute A and then B
**Atomic Shared Memory**

Memory is a map from address to values with reads/writes taking effect immediately.
Intuitive Concurrency Semantics

Shared Memory

Atomic

Program order

Threads

Memory model that guarantees this is called sequential consistency
Sequential Consistency

X* x = null;
bool flag = false;

// Producer Thread
A: x = new X();
B: flag = true;

// Consumer Thread
C: while(!flag);
D: x->f++;

sequential consistency (SC)
[Lamport 1979]
memory operations appear to occur in some global order consistent with the program order
Intuitive reasoning fails in C++/Java

X* x = null;
bool flag = false;

// Producer Thread
A: x = new X();
B: flag = true;

// Consumer Thread
C: while(!flag);
D: x->f++;

In C++ model this can crash!
Intuitive reasoning fails in C++/Java

```cpp
X* x = null;
bool flag = false;

// Producer
A: x = new X();
B: flag = true;

// Consumer
C: while(!flag);
D: x->f++;
```

B doesn’t depend on A. It might be faster to reorder them!
Why are accesses reordered?

- Programming Language
  - Compiler
    - sequentially valid optimizations can reorder memory accesses
      - e.g. common subexpression elimination, register promotion, instruction scheduling
- Hardware
  - sequentially valid optimizations can reorder memory accesses
    - e.g. out-of-order execution, store buffers

Data-Race-Free-0 Model
- Java Memory Model
- C++ Memory Model
A program has a **data race** if it has an execution in which two **conflicting accesses** to memory are simultaneously ready to execute.

```plaintext
// Thread t // Thread u
A: x = new Data(); C: while(!flag);
B: flag = true; D: x->f++;
```

**access the same memory location at least one is a write**
Useful Data Races

- Data races are essential for implementing shared-memory synchronization

```c
AcquireLock()
{
    while (lock == 1) {}  
    t = CAS (lock, 0, 1);  
    if (!t) retry;  
}
```

```c
ReleaseLock()
{
    lock = 0;  
}
```
A program is **data-race-free** if all data races are appropriately annotated (**volatile/atomic**).

**DRFO**

[Adve & Hill 1990]
SC behavior for data-race-free programs, weak or no semantics otherwise

**Java Memory Model (JMM)**
[Manson et al. 2005]

**C++0x Memory Model**
[Boehm & Adve 2008]
DRF0-compliant Program

\[ X^* x = \text{null}; \]
\[ \text{atomic bool flag} = \text{false}; \]

A: \( x = \text{new X}(); \)
B: \( \text{flag} = \text{true}; \)
C: \( \text{while}(!\text{flag}); \)
D: \( x->f++; \)

- DRF0 guarantees SC
  .... only if data-race-free (all \textit{unsafe} accesses are annotated)
- What if there is one data-race?
  .... all bets are off (e.g., compiler can output an empty binary!)
Data-Races are Common

• Unintentional data-races
  □ Easy to accidentally introduce a data race
    ▪ forget to grab a lock
    ▪ grab the wrong lock
    ▪ forget an atomic annotation
    ▪ ...

• Intentional data-races
  □ 100s of “benign” data-races in legacy code
Data Races with no Race Condition (assuming SC)

- Single writer multiple readers

```cpp
// Thread t
A:  time++;

// Thread u
B:  l = time;
```
Data Races with no Race Condition (assuming SC)

• Lazy initialization

```c
// Thread t
if( p == 0 )
    p = init();

// Thread u
if( p == 0 )
    p = init();
```
Intentional Data Races

• ~97% of data races are not errors under SC
  □ Experience from one Microsoft internal data-race detection study [Narayanasamy et al. PLDI’07]

• The main reason to annotate data races is to protect against compiler/hardware optimizations
Data Race Detection is Not a Solution

- Current static data-race detectors are not sound and precise
  - typically only handle locks, conservative due to aliasing, ...

- Dynamic analysis is costly
  - DRFx: throw exception on a data-race [Marino’10]
  - Either slow (8x) or requires complex hardware

- Legacy issues
Deficiencies of DRFO

- Weak or no semantics for data-racy programs
- No easy way to identify & reject racy programs

Problematic for debuggability

Analogous to unsafe languages: relying on programmer infallibility

Optimization + data race = jump to arbitrary code! [Boehm et al., PLDI 2008]

Cost of complexity [Ševčík & Aspinall, ECOOP 2008]

Compiler correctness: Java must maintain safety at the cost of complexity

Debuggability
Languages, compilers, processors are adopting DRF0

Not a strong foundation to build our future systems
Language-level SC: A Safety-First Approach

Program order and shared memory are important abstractions

Modern languages should protect them

All programs, buggy or otherwise, should have SC semantics
What is the Cost of SC?

SC prevents essentially all compiler and hardware optimizations.

And thus SC is impractical.
Sequential Consistency
Review: Coherence

A Memory System is Coherent if

- can serialize all operations to that location such that,
- operations performed by any processor appear in program order (<p)
- value returned by a read is value written by last store to that location

There is broad consensus that coherence is a good idea.

But, that is not enough for consistency...
Coherence vs. Consistency

\[ \begin{align*}
A &= 0 & \text{flag} &= 0 \\
\text{Processor 0} & & \text{Processor 1} \\
A &= 1; & \text{while (!flag);} & \text{// spin} \\
\text{flag} &= 1; & \text{print A;} \\
\end{align*} \]

- **Intuition says**: P1 prints A=1

- **Coherence says**: absolutely nothing
  - P1 can see P0’s write of `flag` before write of `A`!!! How?
    - P0 has a coalescing store buffer that reorders writes
    - Or out-of-order execution
    - Or compiler re-orders instructions

- Imagine trying to figure out why this code sometimes “works” and sometimes doesn’t

- **Real systems** act in this strange manner
  - What is allowed is defined as part of the ISA of the processor
Caches make things more mystifying

A=0  B=0

\begin{align*}
\text{P1} & : & A=1; \\
\text{P2} & : & \text{while } (A==0); \quad \text{while } (B==0); \\
\text{P3} & : & B = 1; \quad \text{print } A;
\end{align*}

- **Intuition says**: P3 prints A=1
  - But, with caches:
    - A=0 initially cached at P3 in shared state
    - Update for A arrives at P2; sends out B=1
    - Update for B arrives at P3
    - P3 prints A=0 before invalidation from P1 arrives

- Many past commercial systems allow this behavior
  - Key issue here: **store atomicity**
    - Do new values reach all nodes at the same time?
Coherence vs. Consistency

Coherence is not sufficient to guarantee a memory model.

Coherence concerns only one memory location.
Consistency concerns apparent ordering for all locations.

Coherency = SC for accesses to one location.

- Guarantees a total order for all accesses to a location that is consistent with the program order.
  - Value returned by a read is value written by last store to that location.
Tools to reason about memory models

- Time? Generally impractical, but may be useful for some systems and use cases (e.g., Lamport clocks)

- (Partially) ordered sets
  - $A \rightarrow B \land B \rightarrow C \Rightarrow A \rightarrow C$ (transitive)
  - $A \rightarrow A$ (reflexive)
  - $A \rightarrow B \land B \rightarrow A \Rightarrow A = B$ (antisymmetric)

- Some important (partial) orders
  - Program order ($<p$) – per-thread order in inst. sequence
  - Memory order ($<M$) – order memory ops are performed
When is a mem. op. “performed”?

- Nuanced definitions due to [Scheurich, Dubois 1987]
  - A Load by $P_i$ is **performed with respect to $P_k$** when new stores to same address by $P_k$ can not affect the value returned by the load.
  - A Store by $P_i$ is **performed with respect to $P_k$** when a load issued by $P_k$ to the same address returns the value defined by this (or a subsequent) store.
  - An access is **performed** when it is performed with respect to all processors.
  - A Load by $P_i$ is **globally performed** if it is performed and if the store that is the source of the new value has been performed.
SC: Hardware

- Formal Requirements:
  - Before LOAD is performed w.r.t. any other processor, all prior LOADs must be globally performed and all prior STOREs must be performed.
  - Before STORE is performed w.r.t. any other processor, all prior LOADs globally performed and all previous STORE be performed.
  - Every CPU issues memory ops in program order

- In simple words:
  SC: Perform memory operations in program order
Sequential Consistency (SC)

processors appear to perform memory ops in program order

P1 P2 P3

Memory

switch randomly set after each memory op provides total order among all operations
Sufficient Conditions for SC

“A multiprocessor is sequentially consistent if 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”

-Lamport, 1979

Every proc. “performs” memory ops in program order

One implementation:

Memory ops happen (start and end) atomically

- Each processor core waits for a memory access to complete before issuing next memory op

Easily implemented with a shared bus
Dekker’s Algorithm

- Mutually exclusive access to a critical region
  - Works as advertised under sequential consistency

```c
/* initial A = B = 0 */

P1
A = 1;
if (B != 0) goto retry;
/* enter critical section*/
P2
B = 1;
if (A != 0) goto retry;
/* enter critical section*/
```
Problems with SC Memory Model

• Difficult to implement efficiently in hardware
  - Straight-forward implementations:
    - No concurrency among memory access
    - Strict ordering of memory accesses at each node
    - Essentially precludes out-of-order CPUs

• Unnecessarily restrictive
  - Most parallel programs won’t notice out-of-order accesses

• Conflicts with latency hiding techniques
E.g., Add a Store Buffer

- Allow reads to bypass incomplete writes
  - Reads search store buffer for matching values
  - Hides all latency of store misses in uniprocessors, but...
Dekker's Algorithm w/ Store Buffer

P1
A = 1;
if (B != 0) goto retry;
/* enter critical section*/

P2
B = 1;
if (A != 0) goto retry;
/* enter critical section*/
Naïve SC Processor Design

Requirement: Perform memory operations in program order

Assume

- coherence
- store atomicity
- + memory ordering restrictions

Memory ordering restrictions

- Processor core waits for store to complete, before issuing next memory op
- Processor core waits for load to complete, before issuing next op
Store Atomicity

- Store atomicity – property of a memory model stating the existence of a total order of all state-changing memory ops.

What does this mean?

All nodes will agree on the order that writes happen

\[
\begin{align*}
&\begin{array}{ll}
& A=0 \quad B=0 \\
P1 & A=1; \\
P2 & B = 1; \\
P3 & Ld B \rightarrow r1; \\
P4 & Ld A \rightarrow r1; \\
& Ld A \rightarrow r2; \\
& Ld B \rightarrow r2;
\end{array}
\end{align*}
\]

Under store-atomicity, what results are (im-)possible?
Implementing Store Atomicity

• On a bus...
  □ Trivial (mostly); store is globally performed when it reaches the bus

• With invalidation-based directory coherence...
  □ Writer cannot reveal new value till all invalidations are ack’d

• With update-based coherence...
  □ Hard to achieve... updates must be ordered across all nodes

• With multiprocessors & shared caches
  □ Cores that share a cache must not see one another’s writes! (ugly!)
SC: Programmer’s Perspective

• Generally the least astonishing alternative
  - Looks a lot like a multitasking uniprocessor
  - Memory behaves as intuition would suggest
  - Causality is maintained (SC implies store atomicity)

• But, still plenty of rope to hang yourself
  - Any memory access is potentially a synchronization
  - Arbitrary wild memory races are legal
  - There is still weirdness with C/C++ bit fields
  - ...thus, PL still exploring alternative paradigms (e.g., TM)

• And it’s probably overkill
  - Most programmers use libraries for sync...
  - ...hence, they don’t actually need SC guarantees
SC: Compiler’s Perspective

• Disaster! Nearly all optimizations initially appear illegal!
  □ Anything could be a sync ⇒ no mem ops may be reordered
  □ Effectively disallows:
    ● Loop invariant code motion
    ● Common sub-expression elimination
    ● Register allocation
    ● ...

• Not quite that bad...
  □ C/C++ specify order only across sequence points (statements)
    ● Operations within an expression may be reordered

• Conflict analysis can improve things [Shasa & Snir’88]
  □ Static analysis identifies conflicting (racing) accesses
  □ Can determine the minimal set of delays to enforce SC
  □ But, needs perfect whole-program analysis
Fixing SC Performance

• Option 1: Change the memory model
  □ Weak/Relaxed Consistency
  □ Programmer specifies when order matters
    ○ Other access happen concurrently/out-of-order
  + Simple hardware can yield high performance
    - Programmer must reason under counter-intuitive rules

• Option 2: Speculatively ignore ordering rules
  □ In-window Speculation & InvisiFence
  □ Order matters only if re-orderings are observed
    ○ Ignore the rules and hope no-one notices
    ○ Works because data races are rare
  + Performance of relaxed consistency with simple programming model
    - More sophisticated HW; speculation can lead to pathological behavior

One of the most esoteric (but important) topics in multiprocessors
We will study it in-depth after winter break
SC Preserving Compiler
SC-Preserving Definition

- A SC-preserving compiler ensures that every SC-behavior of the binary is a SC-behavior of the source
- Guarantees end-to-end SC when the binary runs on SC HW
An SC-Preserving C Compiler

modified LLVM [Lattner & Adve 2004] to be SC-preserving

- obvious idea: restrict optimizations so they never reorder shared accesses
- simple, small modifications to the base compiler
- slowdown on x86: average of 3.8%
  - PARSEC, SPLASH-2, SPEC CINT2006
Many optimizations preserve SC

- for(i=0;i<3;i++)
  X++;

- foo();
- bar();
- baz();

- t=X*4;
- t=X<<2;

- loop unrolling
- X++;X++;X++

- function inlining
- foo();
- X++;
- bar();

- arithmetic reassociation
- bar(){X++;}

- arithmetic simplification
- t=X<<2;

- unreachable code elim.
- dead argument elim.
- loop rotation
- loop unswitching

- virtual to physical register allocation

- correlated val prop

- scalar replication
- tail call elim

- allocating locals to virtual registers
Optimizations That Break SC

- Example: Common Subexpression Elimination (CSE)

L1: \( t = X \times 5; \)
L2: \( u = Y; \)
L3: \( v = X \times 5; \)

\[
\text{L1: } t = X \times 5; \\
\text{L2: } u = Y; \\
\text{L3: } v = X \times 5;
\]

\[
\text{L1: } t = X \times 5; \\
\text{L2: } u = Y; \\
\text{L3: } v = t;
\]

t, u, v are local variables
X, Y are possibly shared
Common Subexpression Elimination is not SC-Preserving

Init: \( X = Y = 0 \);

\[
\begin{aligned}
L1: & \quad t = X \times 5; \\
L2: & \quad u = Y; \\
L3: & \quad v = X \times 5; \\
M1: & \quad X = 1; \\
M2: & \quad Y = 1;
\end{aligned}
\]

\( u = 1 \Rightarrow v = 5 \)

Init: \( X = Y = 0 \);

\[
\begin{aligned}
L1: & \quad t = X \times 5; \\
L2: & \quad u = Y; \\
L3: & \quad v = t; \\
M1: & \quad X = 1; \\
M2: & \quad Y = 1;
\end{aligned}
\]

possibly \( u = 1 \land v = 0 \)
Implementing CSE in a SC-Preserving Compiler

Enable this transformation when
- X is a *safe* variable, or
- Y is a *safe* variable

Identifying *safe* variables:
- Compiler generated temporaries
- Stack allocated variables whose address is not taken

More *safe* variables?
A SC-preserving LLVM for C programs

• Enable transformations on *safe* variables

• Enable transformations involving a single shared variable
  - e.g. \( t = X; u = X; v = X; \rightarrow t = X; u = t; v = t; \)

• Enable trace-preserving optimizations
  - These do not change the order of memory operations
  - e.g. loop unrolling, procedure inlining, control-flow simplification, dead-code elimination,…

• Modified each of ~70 passes in LLVM to be SC-preserving
Experiments using LLVM

- baseline
  - stock LLVM compiler with standard optimizations (-O3)
- no optimizations
  - disable all LLVM optimization passes
- naïve SC-preserving
  - disable LLVM passes that possibly reorder memory accesses
- SC-preserving
  - use modified LLVM passes that avoid reordering shared memory accesses

- ran compiled programs on 8-core Intel Xeon
Parallel Benchmarks

Slowdown over LLVM –O3

-20.00% 0.00% 20.00% 40.00% 60.00% 80.00% 100.00%

llvm-noopt
llvm+trace-preserving
SC-preserving

Naïve
SC-preserving

No opts.

Parallel Benchmarks

Slowdown over LLVM –O3

-20.00% 0.00% 20.00% 40.00% 60.00% 80.00% 100.00%

llvm-noopt
llvm+trace-preserving
SC-preserving

Naïve
SC-preserving

No opts.
SPEC Integer 2006

Slowdown over LLVM –O3

- No optimization
- Naïve SC-preserving
- SC-preserving

Bar chart showing slowdown percentages over LLVM –O3 for various benchmarks.
How Far Can SC-Preserving Compiler Go?

```c
float s, *x, *y;
int i;
s=0;
for( i=0; i<n; i++ ){
    s += (x[i]-y[i])
        * (x[i]-y[i]);
}
```

```c
float s, *x, *y;
int i;
s=0;
for( i=0; i<n; i++ ){
    s += (*x + i*sizeof(float)) -
        *(y + i*sizeof(float)))
        * (*x + i*sizeof(float)) -
        *(y + i*sizeof(float));
}
```

```c
float s, *x, *y;
float *px, *py, *e;
s=0; py=y; e = &x[n]
for( px=x; px<e; px++, py++){
    s += (*px-*py)
        * (*px-*py);
}
```

```c
float s, *x, *y;
float *px, *py, *e, t;
s=0; py=y; e = &x[n]
for( px=x; px<e; px++, py++){
    t = (*px-*py);
    s += t*t;
}
```
Many “Can’t-Live-Without” Optimizations are Eager-Load Optimizations

- Eagerly perform loads or use values from previous loads or stores

**Common Subexpression Elimination**

Original Code:

- L1: \( t = X \times 5 \)
- L2: \( u = Y \)
- L3: \( v = X \times 5 \)

Optimized Code:

- L1: \( t = X \times 5 \)
- L2: \( u = Y \)
- L3: \( v = t \)

**Constant/copy Propagation**

Original Code:

- L1: \( X = 2 \)
- L2: \( u = Y \)
- L3: \( v = X \times 5 \)

Optimized Code:

- L1: \( X = 2 \)
- L2: \( u = Y \)
- L3: \( v = 10 \)

**Loop-invariant Code Motion**

Original Code:

- L1:
- L2: \( \text{for(...)\ for(...)} \)
- L3: \( t = X \times 5 \)

Optimized Code:

- L1: \( u = X \times 5 \)
- L2: \( \text{for(...)\ for(...)} \)
- L3: \( t = u \)
Performance overhead

Allowing eager-load optimizations alone reduces max overhead to 6%
Speculatively Performing Eager-Load Optimizations

- On `monitor.load`, hardware starts tracking coherence messages on X’s cache line.
- The interference check fails if X’s cache line has been downgraded since the monitor.load.
- In our implementation, a single instruction checks interference on up to 32 tags.
Conclusion on SC-Preserving Compiler

• Efficient SC-preserving compiler is feasible with careful engineering

• Hardware support can enable eager-load optimizations without violating SC
Relaxed Consistency
Review: Problems with SC

• Difficult to implement efficiently in hardware
  □ Straight-forward implementations:
    ○ No concurrency among memory access
    ○ Strict ordering of memory accesses at each node
    ○ Essentially precludes out-of-order CPUs

• Unnecessarily restrictive
  □ Most parallel programs won’t notice out-of-order accesses

• Conflicts with latency hiding techniques
Execution in strict SC

- Miss on Wr A stalls all unrelated accesses

Memory accesses issue one-at-a-time
Relaxing Write-to-Read Order

- Motivation: Post-retirement store buffers

- Allow reads to bypass incomplete writes
  - Reads search store buffer for matching values
  - Hides all latency of store misses in uniprocessors

- Writes are still ordered w.r.t. other writes
- Reads are still ordered w.r.t. other reads
Dekker's Algorithm w/ Store Buffer

**P1**
- A = 1;
- if (B != 0) goto retry;
- /* enter critical section*/

**P2**
- B=1;
- if (A != 0) goto retry;
- /* enter critical section*/
Processor Consistency (PC)

- Formal requirements [Goodman 1989]:
  - Before LOAD is performed w.r.t. any other processor, all prior LOADs must be performed
  - Before STORE is performed w.r.t. any other processor, all prior mem ops must be performed

- Does not require store atomicity
- This is basically what x86 and VAX do
- Also roughly what IBM’s System-370 did (1960’s)
  - Oddly, on 370, loads must stall if they hit in the SB
Sun’s “Total Store Order” (TSO)

• Formal requirements [v8 architecture manual]:
  - **Order** - There exists a partial memory order (<M) and it is *total* for all operations with store semantics
  - **Atomicity** - Atomic read-modify-write-writes do not allow a store between the read and write parts
  - **Termination** - All stores are performed in finite time
  - **Value** – Loads return the most recent value w.r.t. memory order (<M) and w.r.t. local program order (<p)
  - **LoadOP** – Loads are blocking
  - **StoreStore** – Stores are ordered

• This is the same as PC, except that it requires store atomicity
PC/TSO: Programmer’s Perspective

• Can occasionally lead to astonishing behavior changes
  □ E.g., Dekker’s algorithm doesn’t work
  □ ISAs provide an STBAR (store barrier) to manually force order
    ❍ Semantics – store buffer must be empty before memory
      operations after STBAR may be executed
  □ Can also enforce order by replacing stores with RMWs

• But, the key case, where sync is done with locks, simply works
  □ Lock acquires are RMW operations ⇒
    they force order for preceding/succeeding loads/stores
    ❍ Load semantics of RMW imply load-load orderings
    ❍ Ditto for store semantics
  □ Lock release is a store operation ⇒
    it must be performed after critical section is done
PC/TSO: Compiler’s Perspective

• Compiler may now hoist loads across stores
  □ Still can’t reorder stores or move loads across loads
  □ Not clear how helpful this is in practice...
  □ ...Recent results from Prof. Satish’s group:
    ❏ 5-30% perf. gap vs. compiler that preserves SC [PLDI’11]

• No new crazy memory barriers to emit
  □ Library/OS writers need to use a little bit of caution;
    use RMWs instead of loads/stores for synchronization
  □ TSO-unsafe code is rare enough that it can be the
    programmer’s problem
• No need to invoke “undefined behavior” to avoid onerous
  implementation requirements
PC/TSO: HW Perspective

- Allows a FIFO-ordered, non-coalescing store buffer
  - Typically maintains stores at word-granularity
  - Loads search buffer for matching store(s)
    - Some ISAs must deal with merging partial load matches
  - Coalescing only allowed among adjacent stores to same block
  - Must force buffer to drain on RMW and STBAR
  - Often, this is implemented in same HW structure as (speculative) store queue

- Can hide store latency!
  - But, store buffer may need to be quite big
    - Stores that will be cache hits remain buffered behind misses
  - Associative search limits scalability
    - E.g., certainly no more than 64 entries
Execution in PC / TSO

- Stores misses do not stall retirement
  - St A, St C, Ld D misses overlapped
  - Ld B retired without waiting for St A to fill
  - St C consumes space in store buffer even though it will hit

**PC/TSO hides store miss latency**
Relaxing Write-to-Write Order

- Motivation: Coalescing store buffers & early drain

- Allows writes to coalesce in SB & drain early
Sun’s “Partial Store Order” (PSO)

- Formal requirements [v8 architecture manual]:
  - **Order** - There exists a partial memory order (<M) and it is total for all operations with store semantics
  - **Atomicity** - Atomic read-modify-write-writes do not allow a store between the read and write parts
  - **Termination** - All stores are performed in finite time
  - **Value** – Loads return the most recent value w.r.t. memory order (<M) and w.r.t. local program order (<p)
  - **LoadOP** – Loads are blocking
  - **StoreStore** – Stores are ordered only if they are separated by a *membar* (memory barrier) instruction
  - **StoreStoreEq** – Stores to the same address are ordered
PSO: Compiler/HW Perspective

- Allows an unordered, coalescing post-retirement store buffer
  - Can now use a cache-block-grain set-associative structure
  - Store misses leave store buffer upon cache fill
  - Loads search buffer for matching store(s)
  - Must force buffer to drain on RMW and STBAR

- Much more efficient store buffer

- But, still doesn’t allow out-of-order loads
  - No OoO execution (without speculation)
  - Compiler’s hands are still tied
Relaxing all Order

- Motivation: Out-of-order execution & multiple load misses

- Now Ld E can complete even though earlier insn aren’t done
Two ISA Approaches to enforce order

• Approach 1: Using explicit “fence” (aka memory barrier)
  ▶ Sun’s Relaxed Memory Order (RMO), Alpha, PowerPC, ARM

  Ld, St, ...
  \[\text{L L S S}\]
  Fence ↓↓↓↓ Enforces order if bit is set
  \[\text{L S L S}\]
  Ld, St, ...

• Approach 2: Annotate loads/stores that do synchronization
  ▶ Weak Ordering, Release Consistency (RC)
  ▶ Data-Race-Free-0 (DRF0) – prog. language-level model

  Load.acquire Lock1
  ...
  Store.release Lock1
More definitions... Dependence Order

• A refinement of program order (<\(p\))

• Dependence order (<\(d\)) captures the minimal subset of (<\(p\)) that guarantees self-consistent execution traces. \(X < p Y\) implies \(X < d Y\) if at least one of the following is true:
  - The execution of \(Y\) is conditional on \(X\) and \(S(Y)\) (\(Y\) is a store)
  - \(Y\) reads a register that is written by \(X\)
  - \(X\) and \(Y\) access the same memory location and \(S(X)\) and \(L(Y)\)

• Dependence order captures what an out-of-order core is allowed to do (ignoring exceptions)
Sun’s “Relaxed Memory Order” (RMO)

- Formal requirements [v9 architecture manual]:
  - \( X <d Y \land L(X) \Rightarrow X <M Y \)
    - RMO will maintain dependence order if preceding insn. is a load
  - \( M(X,Y) \Rightarrow X <M Y \)
    - MEMBAR instructions order memory operations
  - \( X_a <p Y_a \land S(Y) \Rightarrow X <M Y \)
    - Stores to the same address are performed in program order
Execution in RMO w/ fence

- “Fence” indicates ordering affects correctness
  - Retirement stalls at Fence
  - Typically, accesses after fence don’t issue
RMO: Programmer's Perspective

- Programmer must specify MEMBARS wherever they are needed
  - This is hard; Specifying minimum barriers is harder
  - E.g., lock and unlock (from v9 ref. manual; w/o branch delays)

```assembly
LockWithLDSTUB(lock)
retry:  ldstub [lock],%10
tst %10
be out
loop:   l Dub [lock],%10
tst %10
bne loop
ba,a retry
out:    membar #LoadLoad | #LoadStore

UnLockWithLDSTUB(lock)
membar #StoreStore !RMO and PSO only
membar #LoadStore !RMO only
stub %g0,[lock]
```
RMO: Compiler’s Perspective

- Sweet, sweet freedom!
  - Compiler may freely re-order between fences
  - Also across fences if the fence allows it (partial fences)
  - Note: still can’t reorder stores to same address
  - Programmer’s problem if the fences are wrong...
RMO: HW Perspective

• Unordered, coalescing post-retirement store buffer
  □ Just like PSO

• Out-of-order execution!
  □ Need a standard uniprocessor store queue
  □ Fence instruction implementation
    ○ Easy – stall at issue
    ○ Hard – retire to store buffer and track precise constraints
Weak Ordering (WO)

- Loads and stores can be labeled as “sync”
  - No reordering allowed across sync instructions
Release Consistency (RC)

- Specialize loads and stores as “acquires” and “releases”
  - Load at top of critical section is an acquire
  - Store at bottom of critical section is a release
  - Allows reordering into critical sections, but not out

Weak Consistency (WCsc)
WO/RC: Programmer Perspective

- A new way of thinking: programmer-centric memory models
  - If you annotate syncs correctly, your program will behave like SC
  - E.g., Data-Race-Free-0 (DRF0)
    - Accesses are either normal or sync
    - Sync accesses are sequentially consistent, and order normal accesses
    - Data races among normal accesses are prohibited
    - DRF0 programs appear as if they ran under SC

- Lot’s of ongoing work on race detection
WO/RC: Compiler’s Perspective

- DRF0 is foundation of C++ and Java memory models
  - Compiler may freely re-order between syncs
  - But, it may not introduce any new data races
    - Can’t do speculative loads (can disallow optimizations)

- Why are races so evil?
  - They break reasonable optimization:
    ```
    unsigned int i = x;  // x is shared variable
    if (i < 2) {          // opt: don’t copy x, use by ref
      foo: ...           // suppose x changes here...
      switch (i) {
        case 0: ... break;
        case 1: ... break;
        default: ... break;  // opt: range inference tells
        }                    // compiler this case is
      }                      // impossible, drop the check
    ```