



TECHNISCHE
UNIVERSITÄT
DRESDEN

Faculty of Computer Science Institute for System Architecture, Operating Systems Group

Distributed Operating Systems

Memory Consistency

Marcus Völp

(slides Julian Stecklina, Marcus Völp)

SS2014

global variables: int i;
 int k;

i = 1;
if (**i > 1**) k = 3;

||

i = i + 1;
if (k == 0) k = 4;

Cache Coherency: Consistency of reads/writes
to the **same** memory location.


Ordering and consistency of reads/writes to
different memory locations?
=> Memory Consistency

Memory Consistency Model

defines how loads and stores to different memory locations become visible with respect to each other.

Different memory consistency models exist

- Sequential Consistency (MIPS R10K)
- IBM 370 (z-Series)
- Total Store Order (Sparc v8)
- Processor Consistency (Intel x86)
- Partial Store Order (Sparc v8)
- Weak Ordering
- Release Consistency
- Dependent Load Ordering (Alpha)



more complex,
more performance

“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. A multiprocessor satisfying this condition will be called **sequentially consistent.**”
[Lamport 1979]

Three major ingredients:

- Atomicity: “one operation at a time”
 a, b are atomic if $A \parallel B = A;B$ or $B;A$
- Issue: “order specified by its program”
(program order)
- Visibility: “some sequential order”
the same for all processors

Violating Atomicity

[A] [B] Memory
u, v Registers

CPU 0

```
[A] = 0xdeadbeaf; (a1)
```

CPU 1

```
u = [A]; (a2)
```

```
if (u == 0x0000beaf)  
    print ("mmh, delicious")
```

a1.low, a2.low, a2.high, a1.high
disallowed in sequentially consistent systems

Violating Program Order

CPU 0

```
[Buffer] = 0xbeaf;      (a1)
[Flag]   = 1; /* full */ (b1)
```

CPU 1

```
u = [Flag]; (a2)

if (u == 1)
    /* use buffer */ (b2)
```

b1, a2, b2, a2
disallowed in sequentially consistent systems

Violating Same Total Visibility Order

CPU 0

[Data] = 0xbeaf; (a1)

CPU 1

[Data] = 0xdead; (a2)

CPU 2

v = [Data]; (a3)

CPU 3

v = [Data]; (a4)

print ("He said" v)

print ("He said" v)

a1, a2, ..., a3, a4

a2, a1, ..., a3, a4

disallowed in sequentially consistent systems

But: (assuming [A] ... [Z] is “normal” memory)

CPU 0

```
lock()  
  [A] = 1;  
  [B] = 1;  
  [C] = 1;  
  ...  
  [Z] = 1;  
unlock()
```

CPU 1

```
lock()  
  u = [A];  
unlock()
```


CPU0

[A] = 1; (a1)

[B] = 1; (b1)

CPU1

u = [B]; (a2)

v = [A]; (b2)

$(u,v) = (1,1)$

- Sequentially consistent: a1, b1, a2, b2

$(u,v) = (1,0)$

- Not sequentially consistent: a2, b2, a1, b1
- Violates program order for CPU0/1
- No visibility order possible that is seq. consistent!

CPU0

[A] = 1; (a1)

u = [B]; (b1)

CPU1

[B] = 1; (a2)

v = [A]; (b2)

(u,v) = (1,1)

- Sequentially consistent: a1, a2, b1, b2

(u,v) = (0,0)

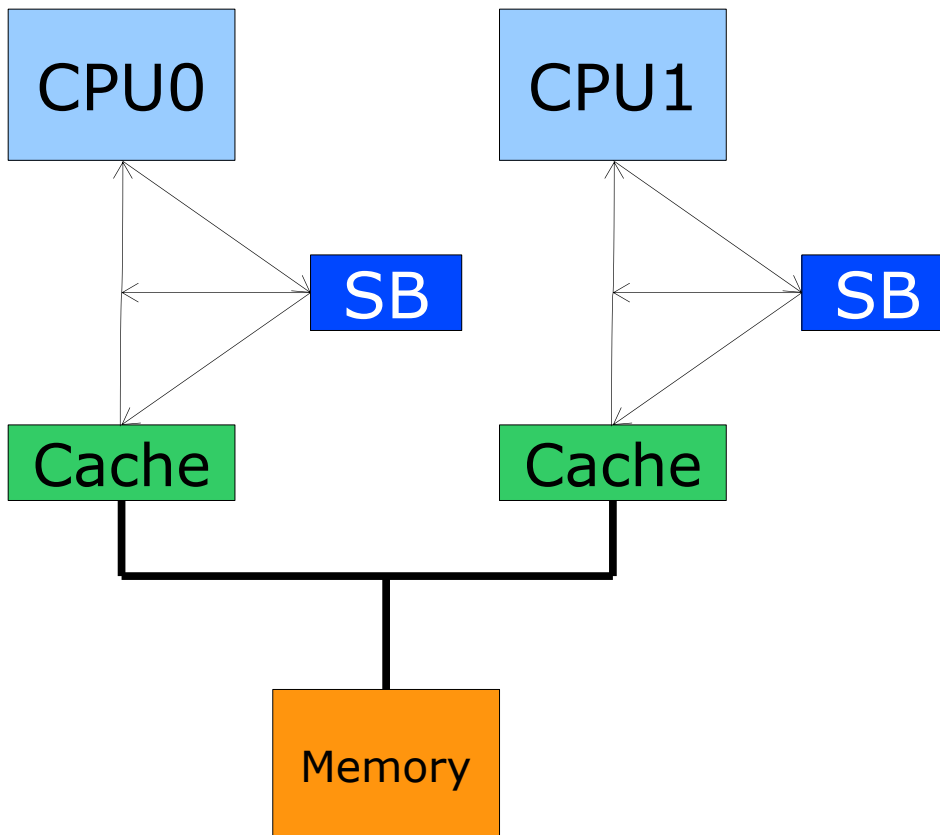
- Not sequentially consistent: b1, b2, a1, a2
- Violates program order for CPU0/1
- No visibility order possible that is seq. Consistent!

Memory consistency models describe which operations can be reordered in the visibility order of memory operations and which are maintained.

In-order memory operations in SC:

- Read→Read
- Read→Write
- Write→Read
- Write→Write

Weaker models relax some or all of these orderings.



SB optimizes writes to memory and/or caches to optimize interconnect accesses.

CPU can continue before write is completed.

Store forwarding allows reads from local CPU to see pending writes in the SB.

SB invisible to remote CPUs.

FIFO vs. non-FIFO. Writes can be combined, may reorder writes on some architectures.

Relaxing Write→Read (later reads can bypass earlier writes)

- Write followed by a read can execute out-of-order
- Typical hardware usage: Store Buffer
 - Writes must wait for cache line ownership
 - Reads can bypass writes in the buffer
 - Hides write latency

Relaxing Write→Write (later writes can bypass earlier writes)

- Write followed by a write can execute out-of-order
- Typical hardware usage: Coalescing store buffer

- In-order memory operations:
 - Read→Read
 - Read→Write
 - Write→Write
- Out-of-order memory operations:
 - Write-to-Read (later reads can bypass earlier writes)
 - Unless both to same location
 - Write-to-Read to same location must execute in-order
 - No forwarding from the store buffer

- In-order memory operations:
 - Read-to-Read
 - Read-to-Write
 - Write-to-Write
- Out-of-order memory operations:
 - Write-to-Read (later reads can bypass earlier writes)
 - Forwarding of pending writes in the store buffer to successive reads to the same location
 - Writes become visible to writing processor first
 - Store buffer is FIFO

CPU0 $[A] = 1; (a1)$ $u = [A]; (b1)$ $w = [B]; (c1)$ **CPU1** $[B] = 1; (a2)$ $v = [B]; (b2)$ $x = [A]; (c2)$

- $(u,v,w,x) = (1,1,0,0)$
 - Not possible with SC and z Series
 - Possible with TSO
 - $b1, b2, c1, c2, a1, a2$
 - $b1$ reads $[A]$ from write buffer
 - $b2$ reads $[B]$ from write buffer

- Similar to Total Store Order (TSO)
- Additionally supports multiple cached memory copies
 - Relaxed atomicity for write operations
 - Each write broken into suboperations to update cached copies of other CPUs
 - Non-unique write order: **per-CPU visibility order**
- Additional coherency requirement
 - All write suboperations to the same location complete in the same order across all memory copies (or in other words: each processor sees writes to the same location in the same order)

CPU0 $[A] = 1; (a1)$ **CPU1** $u = [A]; (a2)$ $[B] = 1; (b2)$ **CPU2** $v = [B]; (a3)$ $w = [A]; (b3)$

- $(u,v,w) = (1,1,0)$
 - Not possible with SC, z Series, TSO
 - Possible with Processor Consistency (PC)
 - CPU0 sets [A], sends update to other CPUs
 - CPU1 gets update, sets [B], sends update
 - CPU2 sees update from CPU1, but hasn't seen update from CPU0 yet
 - Single memory bus enforces single visibility order
 - Multiple visibility orders with different topologies

CPU0

```
[A] = 1;
```

CPU1

```
while ([A] == 0);  
[B] = 1;
```

CPU2

```
while ([B] == 0);  
print [A];
```

Write Atomicity

All cores see writes at the same time (and the same order).

Relaxing write atomicity

- CPU0 writes [A]; sends update to CPU1/2
- CPU1 receives; writes [B]; sends update to CPU2
- CPU2 receives update from CPU1, prints [A] = 0
- CPU2 receives update from CPU0

Not sequentially consistent!

- In-order memory operations:
 - Read→Read
 - Read→Write
- Out-of-order memory operations:
 - Write→Read (later reads can bypass earlier writes)
 - Forwarding of pending writes to successive reads to the same location
 - Write→Write (later writes can bypass earlier writes)
 - Unless both are to the same location
 - Breaks naive producer-consumer code
- Write atomicity is maintained → single visibility order

CPU0`[A] = 1; (a1)``[B] = 1; (b1)``[Flag] = 1; (c1)`**CPU1**`while ([Flag] == 0); (a2)``u = [A]; (b2)``v = [B]; (c2)`

- $(u,v) = (0,0)$ or $(0,1)$ or $(1,0)$
 - Not possible with SC, z Series, TSO, PC
 - Possible with PSO
 - $c1, a2, b2, c2, a1, b1$
 - Store Barrier (STBAR) before $c1$ ensures sequentially consistent result $(u,v) = (1,1)$

- In addition to previous relaxations:
 - Read→Read (later reads can bypass earlier reads)
 - Read followed by read can execute out-of-order
 - Read→Write (later writes can bypass earlier reads)
 - Read followed by a write can execute out-of-order
- Examples
 - Weak Ordering (WO)
 - Release Consistency (RC)
 - DEC Alpha
 - SPARC V9 Relaxed Memory Model (RMO)
 - PowerPC
 - Itanium (IA-64)

- Conceptually similar to Processor Consistency
 - Including coherency requirement
- Classifies memory operations into
 - Data operations
 - Synchronization operations
- Reordering of operations between synchronization operations typically does not affect correctness of a program
- Program order only maintained at synchronization points
 - Between synchronization operations

- Distinguishes memory operations as
 - Ordinary (data)
 - Special
 - Sync (synchronization)
 - Nsync (asynchronous data)
- Sync operations classified as
 - Acquire
 - Read operation for gaining access to a shared resource
 - e.g., spinning on a flag to be set, reading a pointer
 - Release
 - Write operation for granting permission to a shared resource
 - e.g., setting a synchronization flag

- RC_{SC}
 - Sequential consistency between special operations
 - Program order enforced between:
 - acquire \rightarrow all
 - all \rightarrow release
 - special \rightarrow special
- RC_{PC}
 - Processor consistency between special operations
 - Program order enforced between:
 - acquire \rightarrow all
 - all \rightarrow release
 - special \rightarrow special, **except** release followed by acquire

$A = 1, B = 0, P = \&A$

CPU0

$[B] = 1;$ (a1)

Store barrier

$[P] = \&B;$ (b1)

CPU1

$u = [P]$ (a2)

$v = [u];$ (b2)

Load depends previous load for address generation. Alpha may reorder loads due to speculation. Allows:

$(u,v) = (\&B, 0)$

- Even with barrier between a1,b1!
- Visibility order: a1,b1,b2,a2

Most (all?) processors except Alpha disallow dependend load/store reordering.

- IA32
 - lfence, sfence, mfence (load, store, memory fence)
- Alpha
 - mb (memory barrier), wmb (write memory barrier)
- SPARC (PSO)
 - stbar (store barrier)
- SPARC (RMO)
 - membar (4-bit encoding for r-r, r-w, w-r, w-w)
- PowerPC
 - sync (similar to Alpha mb, except r-r), lwsync
 - eieio (enforce in-order execution of I/O)

Compilers reorder memory accesses for performance.
Effects are equivalent to reordering by hardware.

```
Flag0 = true;          ld r1 ← flag1
while (flag1) {        st flag0 ← true
  ...                  loop: cmp r1,0
  ...                  ...
}                      ld r1 ← flag1
```

Is this a legal optimization?

Single threaded: Yes
Can't perceive difference

Multithreaded: **NO!**

Standardized memory models for HLL:

- C / C++ 2011
- Java

Basic model: Sequentially Consistency for data-race free programs (SC-DRF)

A data race free program will execute sequentially consistent.

Data Race (informal)

Multiple threads access a memory location without synchronization, one of them is a writer.

```
a = b = 0;
```

Thread 1

```
mtx_lock(l);
```

```
a = 1;
```

```
b = 1;
```

```
mtx_unlock(l);
```

Thread 2

```
x = a;
```

```
y = b;
```

Not Data Race Free:

- a,b accessed without synchronization
- $(x,y) = (0,0) (1,0) (0,1) (1,1)$ all legal!
- Need to add synchronization to Thread 2

With synchronization yields either $(0,0)$ or $(1,1)$:

- Data race free, sequentially consistent!

- Mutexes may cause scalability issues
- C++ 11 offers rich set of atomic memory operations (`std::atomic`)
 - Implements RC_{SC} :
 - Atomic reads acquire
 - Atomic stores release
 - Can use weaker ordering if desired
 - Compare-and-Swap
 - Add/Sub/And/Or/Xor/...
- Does the right thing on all platforms
 - Adds appropriate memory barriers
 - Uses locked instructions as necessary
 - May use locks on certain platforms!

- A Primer on Memory Consistency and Cache Coherence
Sorin, Hill, Wood; 2011
- [atomic<> Weapons](#): The C++ Memory Model and Modern Hardware (Video)
Sutter; 2013
- [Shared memory consistency models: a tutorial](#)
Adve, Gharachorloo; 1996
- [IA Memory Model](#)
Richard Hudson; Google Tech Talk 2008
- [Memory Ordering in Modern Microprocessors](#)
McKenney; Linux Journal 2005
- How to Make a Multiprocessor Computer That Correctly Executes Multiprocess Programs
Lampert, 1979
- [PowerPC Storage Model](#)

Scheduler-Conscious Synchronization

Leonidas Kontothanassis, Robert Wisniewski, Michael Scott

Scalable Reader- Writer Synchronization for Shared-Memory Multiprocessors

John M. Mellor-Crummey, Michael L. Scott

Algorithms for Scalable Synchronization on Shared-Memory Multiprocessors

John Mellor-Crummey, Michael Scott

Concurrent Update on Multiprogrammed Shared Memory Multiprocessors

Maged M. Michael, Michael L. Scott

Scalable Queue-Based Spin Locks with Timeout

Michael L. Scott and William N. Scherer III

Reactive Synchronization Algorithms for Multiprocessors

B. Lim, A. Agarwal

Lock Free Data Structures

John D. Valois (PhD Thesis)

Reduction: A Method for Proving Properties of Parallel Programs

R. Lipton - *Communications of the ACM* 1975

Decoupling Contention Management from Scheduling (ASPLOS 2010)

F.R. Johnson, R. Stoica, A. Ailamaki, T. Mowry

Corey: An Operating System for Many Cores (OSDI 2008)

Silas Boyd-Wickizer, Haibo Chen, Rong Chen, Yandong Mao,
Frans Kaashoek, Robert Morris, Aleksey Pesterev, Lex Stein,
Ming Wu, Yuehua Dai, Yang Zhang, Zheng Zhang