Parallel Computing Platforms:

Coherence, Ordering, & Synchronization

John Mellor-Crummey

Department of Computer Science
Rice University

johnmc@rice.edu

Topics

- Cache coherence
  - —update vs. invalidate
  - —snoopy vs. directory
  - —protocol examples
- Memory models and weak ordering
- Shared-memory synchronization
  - —approaches
  - —primitives
  - —operations
    - – initialize, signal, acknowledge, reinitialize
  - —techniques
    - – sense switching
    - – paired data structures
    - – avoid interconnect traffic due to spin waiting (local spinning)

Cache Coherence

- **Shared address space machines**
  - —must coordinate access to data that might have multiple copies
    - – copies in caches
  - —multiple copies can easily become inconsistent
    - – processor writes, I/O writes
  - —coordination must provide some guarantees about the semantics
- **Sequential consistency**
  - —all data accesses appear to have been executed
    - – atomically
    - – in some sequential order

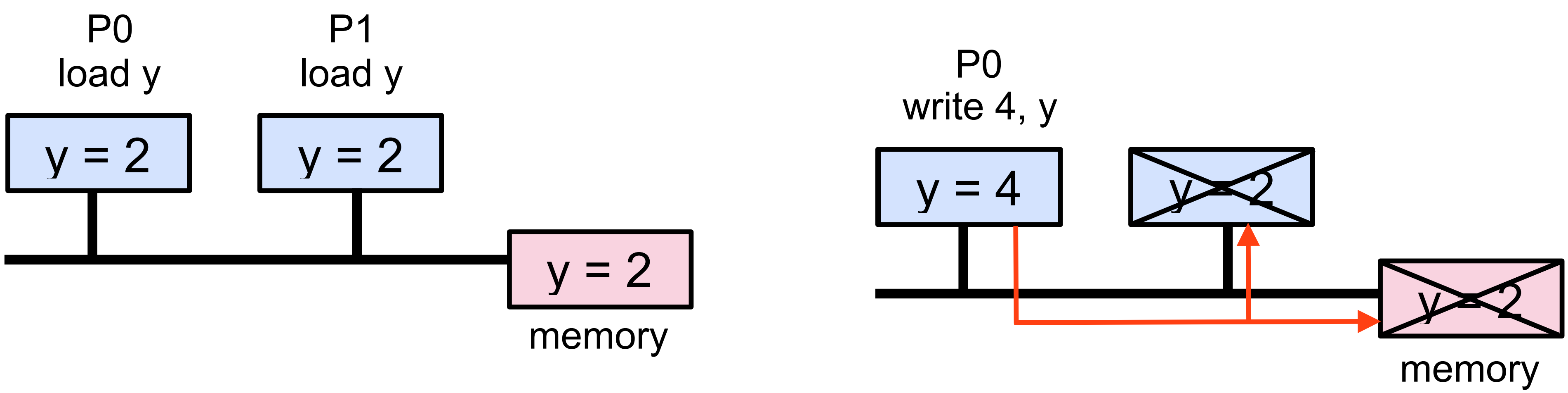
      consistent with the order of operations in individual threads
  - —corollary
    - – each variable must appear to have only a single value at a time

Approaches to Cache Coherence

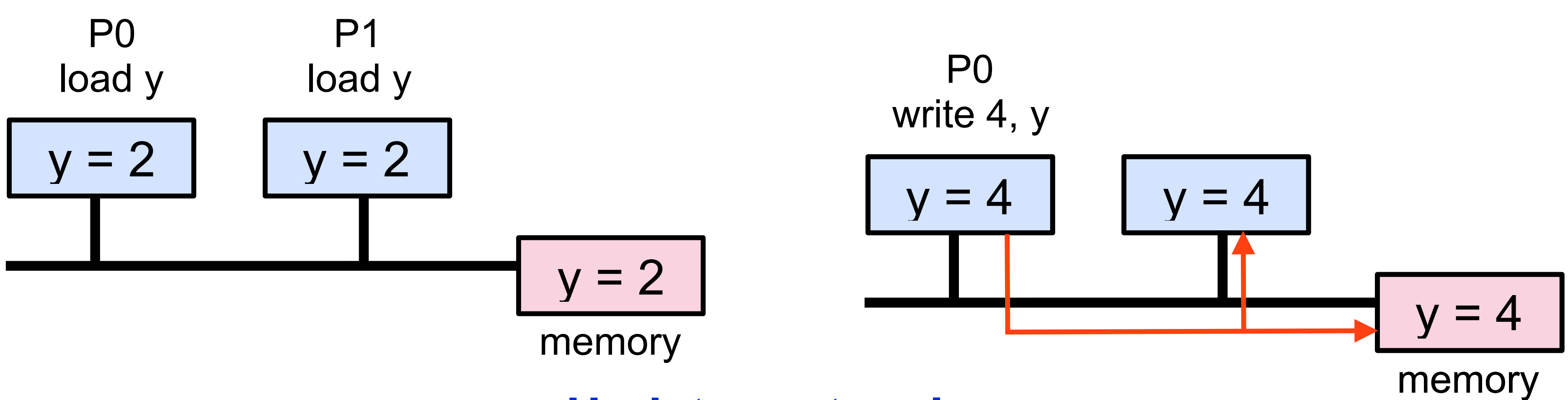
- **Hardware**
  - —caches implement coherence protocols to ensure that data appears globally consistent
  - —typical in systems today
- **Software**
  - —relies on compiler and/or runtime support
    - – may or may not have help from the hardware
  - —must be conservative to be safe
    - – assume the worst about potential memory aliases
  - —of increasing interest
    - – concerns about cost of coherence in joules
    - – scales well for microprocessors based on “tiled” designs
      - Intel Scalable Cloud Computer (SCC), 2010

Cache Coherence Protocols

When changing a variable's value: <u>invalidate</u> or <u>update</u> all copies



Invalidate protocol



Update protocol

Update and Invalidate Protocols

- Cost-benefit tradeoff depends upon traffic pattern
  - —invalidation is worse when
    - – single producer of data and many consumers
  - —update is worse when
    - – multiple writes by one CPU before data is read by another
    - – a cache is filled with data that is not read again

      e.g., leftovers after thread or process migration
- Data organized in cache lines
  - —e.g. 64B on recent Intel processors
- Both protocols suffer from false sharing overheads
  - —line accessed by multiple readers and writers
  - —cores accessing disjoint data
  - —false sharing overhead = coherence cost in this case
- Modern machines use invalidate protocols as the default

Using Invalidate Protocols

- **Each cache line is associated with a state**
- **Example set of states: modified, exclusive, shared, or invalid**
  - —modified: only one copy exists
    - a write need not generate any invalidates
  - —exclusive: only one copy exists
    - a write need not generate any invalidates
  - —shared: multiple valid copies of the data item
    - a write needs to generate an invalidate
  - —invalid: data copy is invalid
    - a read generates a data request and updates the state

Diagram for 4-state MESI Protocol

MESI states: Modified, Exclusive, Shared, and Invalid

- **PrRd: processor read**
- **PrW: processor write**
- **BusRd(S): bus read (shared)**
- **BusRd(~~S~~): bus read (not shared)**
- **BusRdX: bus read exclusive - cause others to invalidate**
- **Flush: save a line to memory**
- **Flush': cache to cache transfer, one cache saves data to memory**

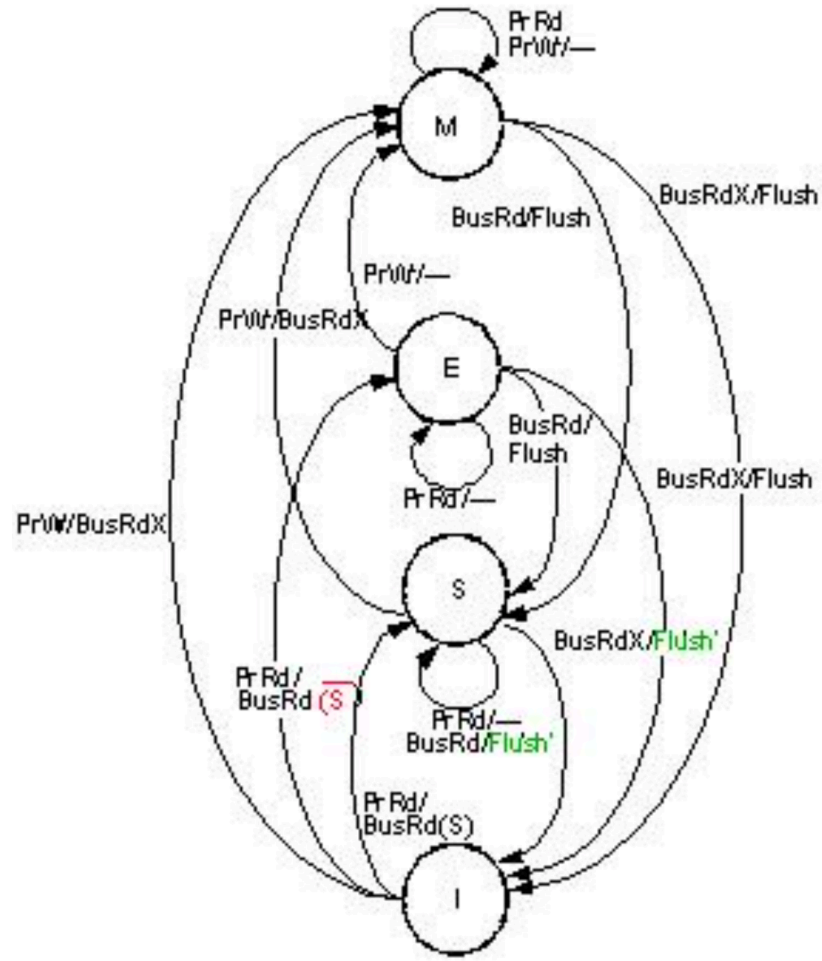


Figure credit: David Culler, UC Berkeley. CS 258, Spring 99, Lecture 7, Slide 15
https://people.eecs.berkeley.edu/~culler/cs258-s99/slides/lec07/sld015.htm

MESI Implications for Multiple Caches

MESI states: Modified, Exclusive, Shared, and Invalid

Permissible state pairs for a pair of caches

	M	E	S	I
M				
E				
S				
I				

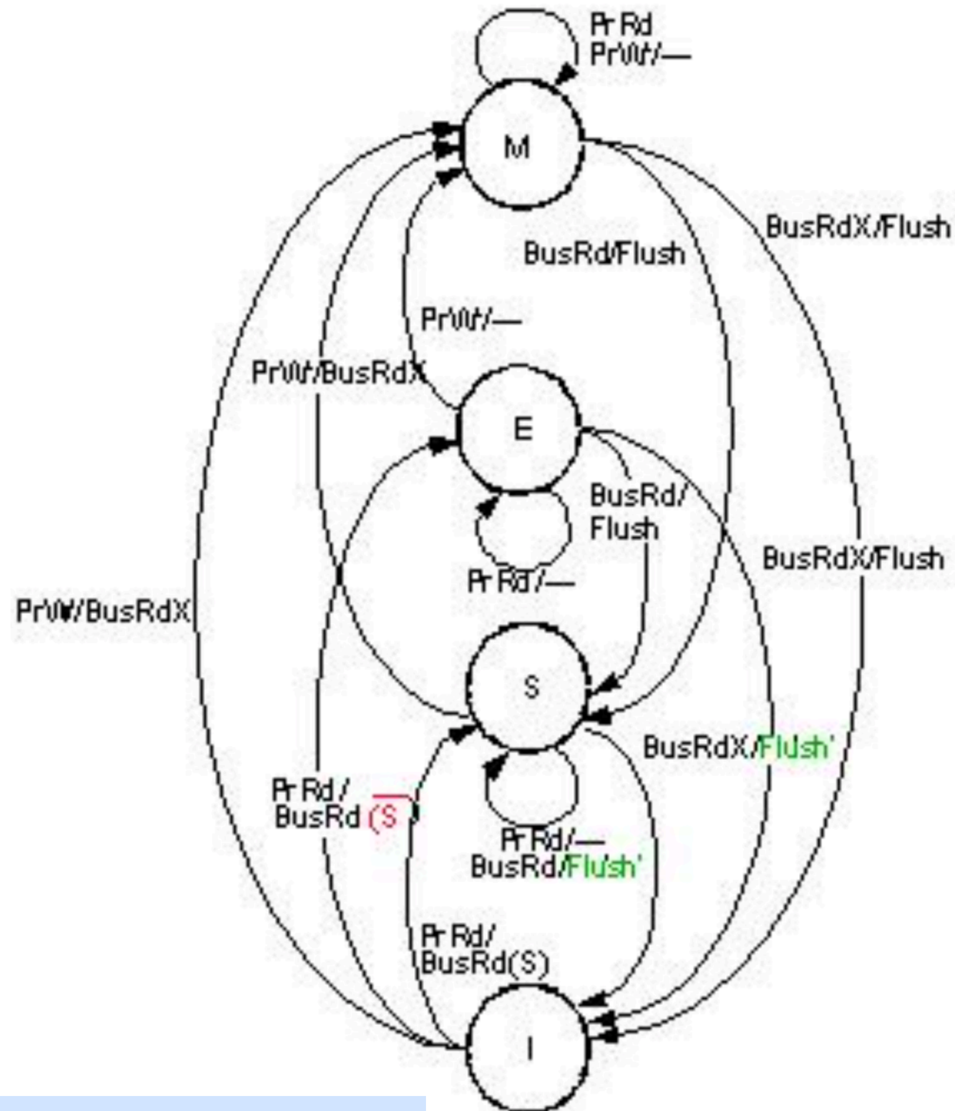


Figure credit: David Culler, UC Berkeley. CS 258, Spring 99, Lecture 7, Slide 15
https://people.eecs.berkeley.edu/~culler/cs258-s99/slides/lec07/sld015.htm

Contemporary use of Update Protocols

Finally, the barrier synchronization register (BSR) facility originally implemented in POWER5 and POWER6 processors has been virtualized in the POWER7 processor [2]. Within each system, multiple megabytes of main storage may be classified as BSR storage and assigned to tasks by the virtual memory manager. The BSR facility enables low-latency synchronization for parallel tasks. Writes to BSR storage are instantaneously broadcast to all readers, allowing a designated master thread to orchestrate the activities of workers threads in a low-latency fine-grained fashion. This capability is particularly valuable for improving parallel speedups in certain HPC environments.

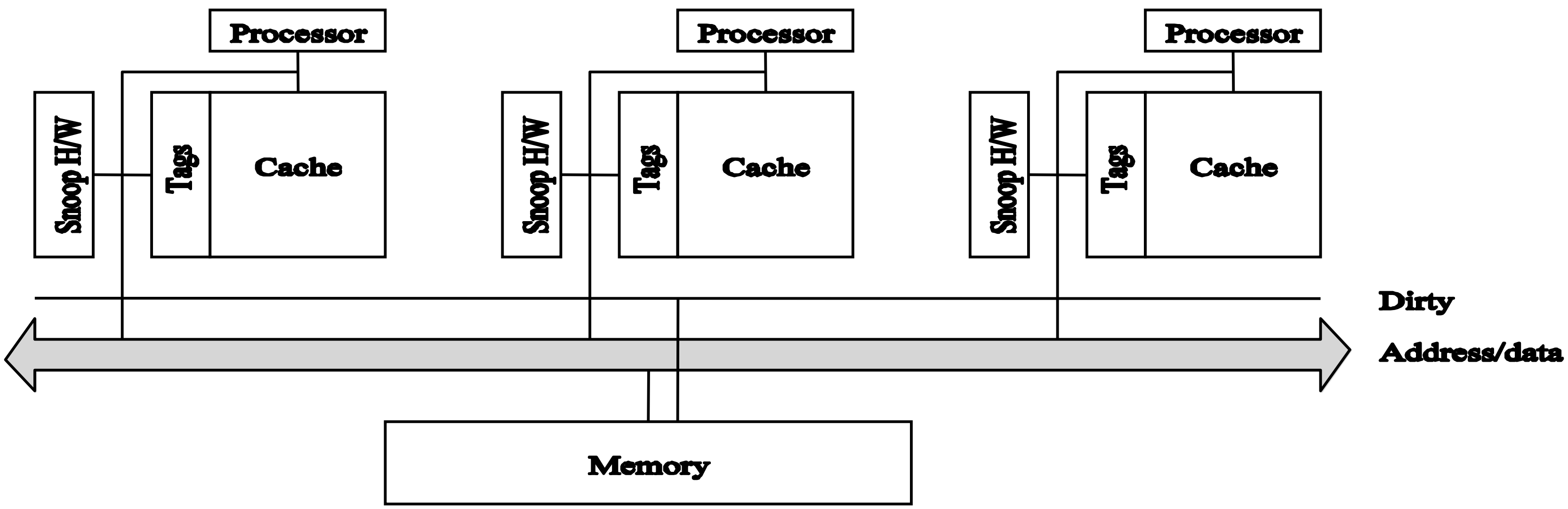
B. Sinharoy.et al. [IBM POWER7 multicore server processor.](http://dx.doi.org/10.1147/JRD.2011.2127330) IBM Journal of Research and Development 55(3), May-June 2011, 1:1-1:29. http://dx.doi.org/10.1147/JRD.2011.2127330

Snoopy Cache Systems

How are invalidates sent to the right processors?

Snoopy cache systems

- Broadcast all invalidates and read requests
- Snoopy cache listens and performs appropriate coherence operations locally

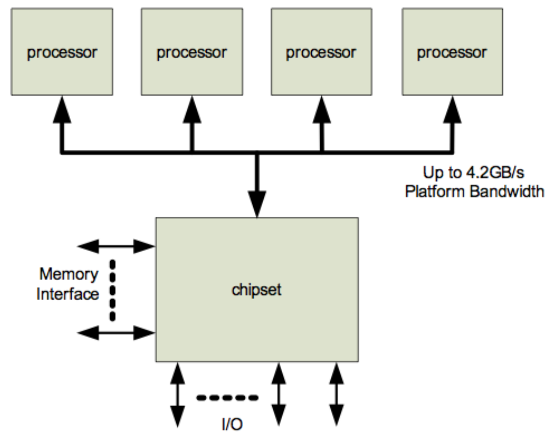


Simple bus-based snoopy cache coherence

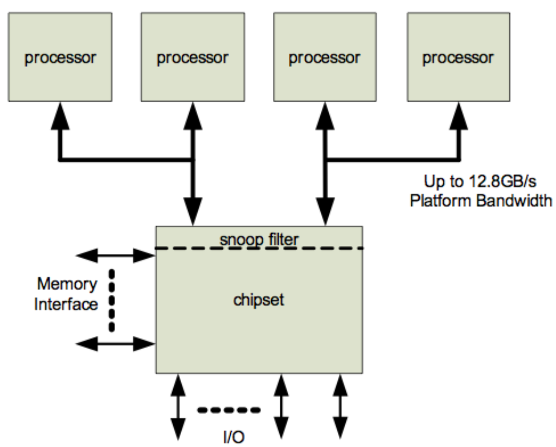
Operation of Snoopy Caches

- **Once a datum is tagged modified or exclusive**
  - —all subsequent operations can be performed locally in cache
  - —no external traffic needed
- **If a data item is read by a number of processors**
  - —transitions to the shared state in all caches
  - —all subsequent read operations become local
- **If multiple processors read and update data**
  - —generate coherence requests on the bus
  - —bus is bandwidth limited: imposes a limit on updates per second

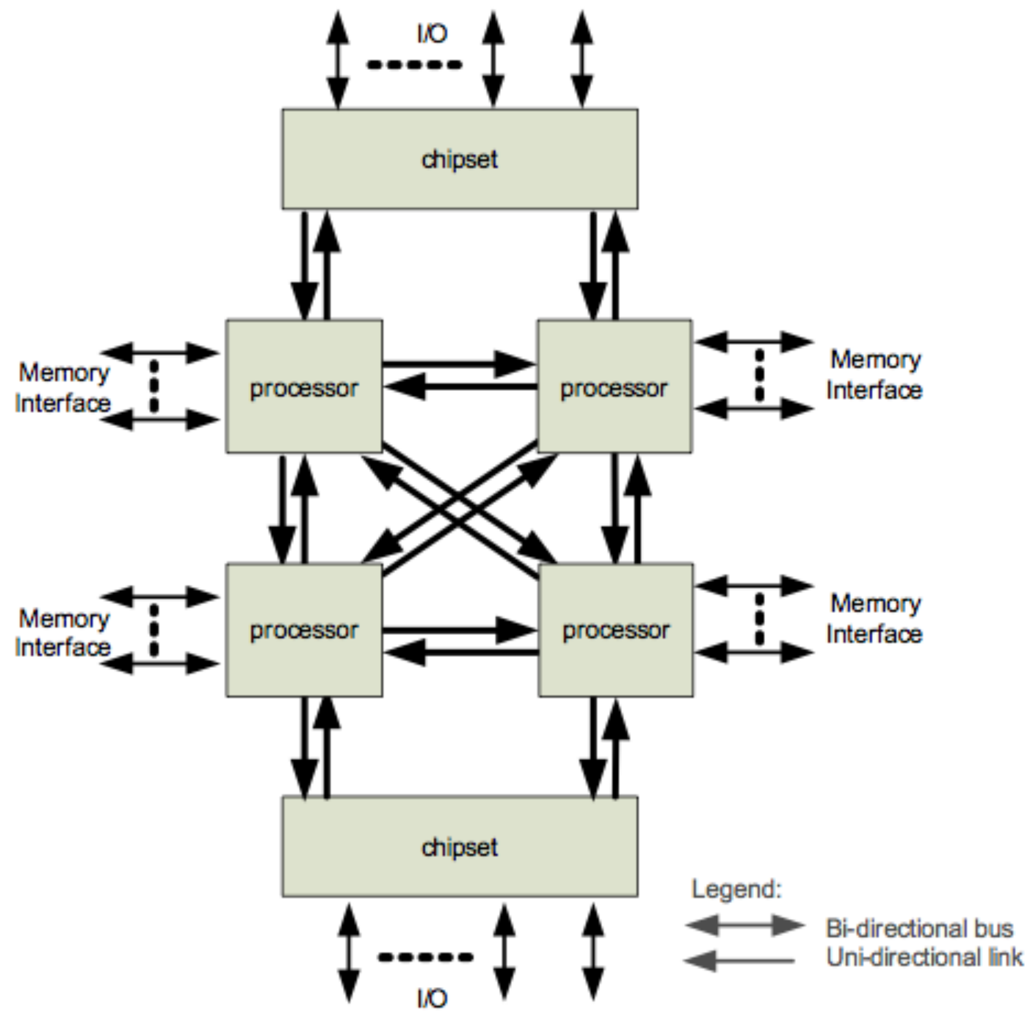
Evolution of Node Interconnects



Shared front-side bus
(through 2004)



Dual independent buses
(circa 2005)



Intel Quickpath interconnect
(2009 - present)

Figure credits: Introduction to Intel QuickPath Interconnect in Weaving High Performance Multiprocessor Fabric, Robert A. Maddox, Gurbir Singh, and Robert J. Safranek, Intel Press

Intel MESIF Protocol (2005)

- MESIF: **M**odified, **E**xclusive, **S**hared, **I**nvalid and **F**orward
- If a cache line is shared
  - —one shared copy of the cache line is in the F state
  - —remaining copies of the cache line are in the S state
- Forward (F) state designates a single copy of data from which further copies can be made
  - —cache line in the F state will respond to a request for a copy of the cache line
  - —consider how one embodiment of the protocol responds to a read
    - – newly created copy is placed in the F state
    - – cache line previously in the F state is put in the S or the I state

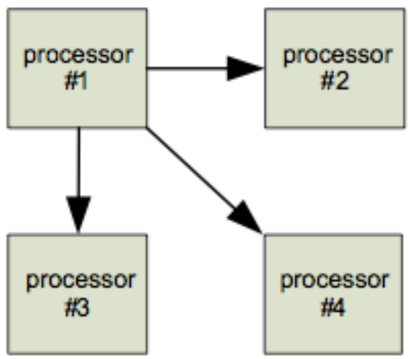
H. Hum et al. US Patent 6,922,756. July 2005. http://bit.ly/gQNkRR

Intel QuickPath Source Snoop

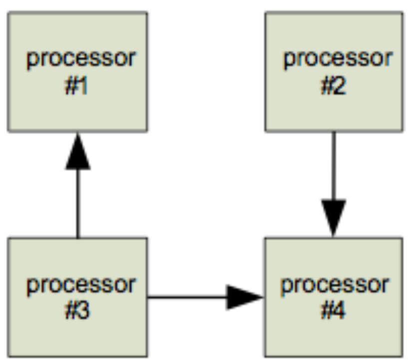
Labels: P1 is the requesting caching agent
P2 and P3 are peer caching agents
P4 is the home agent for the line
Precondition: P3 has a copy of the line in either M, E or F-state

MESIF protocol (Intel): Modified (M), Exclusive (E), Shared (S), Invalid (I) and Forward (F)

Step 1.
P1 requests data which is managed by the P4 home agent. (Note that P3 has a copy of the line.)



Step 2.
P2 and P3 respond to snoop request to P4 (home agent).
P3 provides data back to P1.



Step 3.
P4 provides the completion of the transaction.

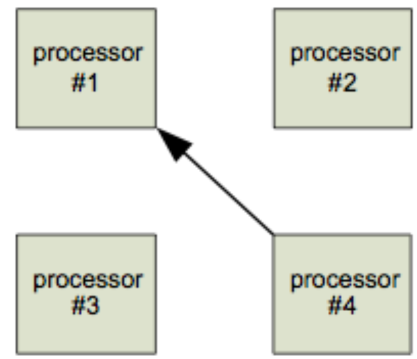


Figure credits: Introduction to Intel QuickPath Interconnect in Weaving High Performance Multiprocessor Fabric, Robert A. Maddox, Gurbir Singh, and Robert J. Safranek, Intel Press

Beyond MESI and MESIF: Power7 Cache States

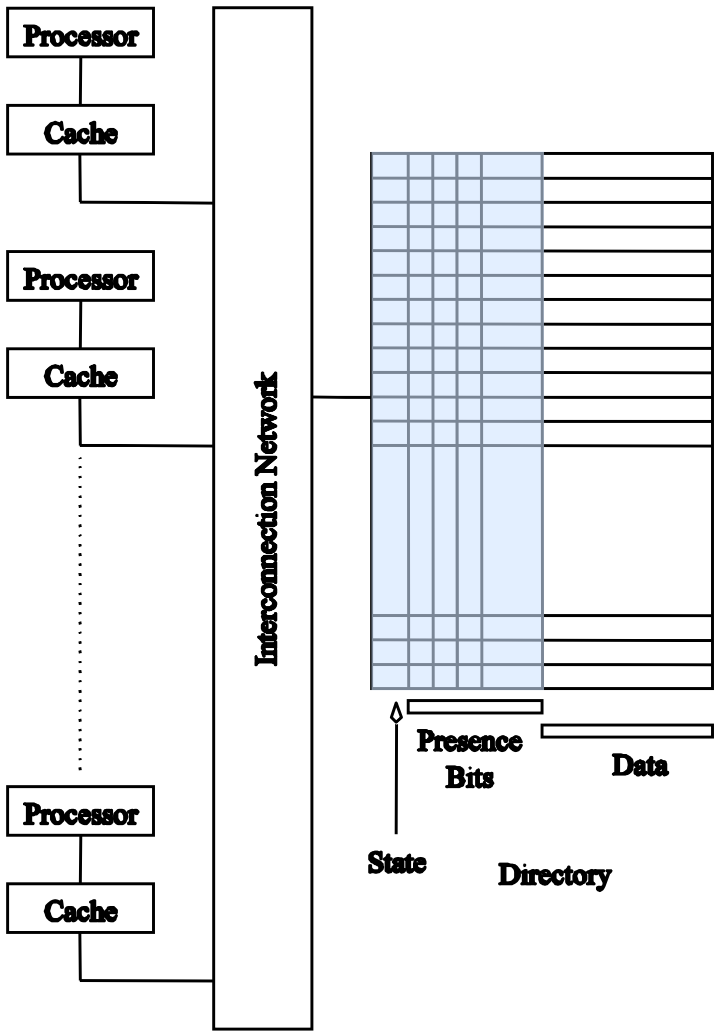
State	*Description*	*Authority*	*Sharers and scope*	*Source data*	*Data cast-out*	*Scope cast-out*
I	Invalid	None	N/A	N/A	N/A	None
ID	Deleted, do not allocate	None	N/A	N/A	N/A	None
S	Shared	Read	Yes, scope unknown	No	No	None
SL	Shared, local data source	Read	Yes, scope unknown	At request	No	None
T	Formerly MU, now shared	Update	Yes, probably global	If notified	Yes	Required, global
TE	Formerly ME, now shared	Update	Yes, probably global	If notified	No	Required, global
M	Modified, avoid sharing	Update	No	At request	Yes	Optional, local
ME	Exclusive	Update	No	At request	No	None
MU	Modified, bias toward sharing	Update	No	At request	Yes	Optional, local
IG	Invalid, cached scope-state	None	N/A, probably global copies	N/A	N/A	Required, global
IN	Invalid, scope predictor	None	N/A, probably local copies	N/A	N/A	None
TN	Formerly MU, now shared	Update	Yes, local	If notified	Yes	Optional, local
TEN	Formerly ME, now shared	Update	Yes, local	If notified	No	None

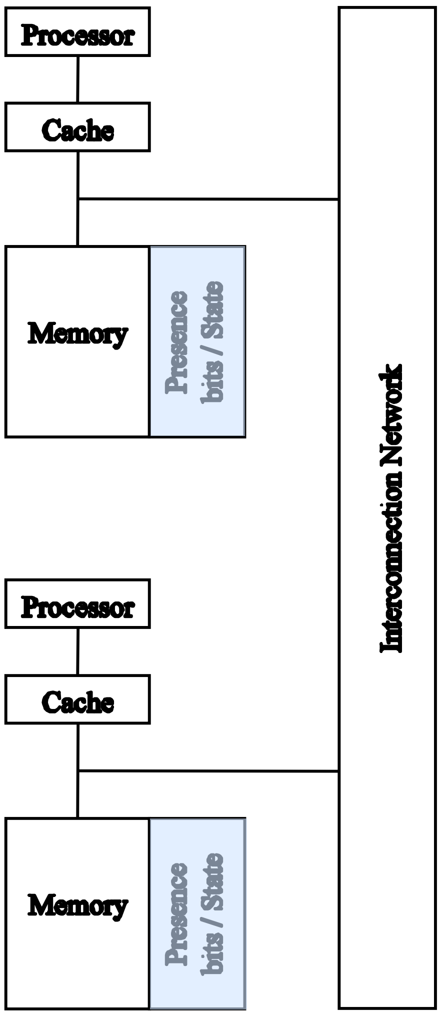
B. Sinharoy.et al. IBM POWER7 multicore server processor. IBM Journal of Research and Development 55(3), May-June 2011, 1:1-1:29. http://dx.doi.org/10.1147/JRD.2011.2127330

The Cost of Coherence

- **Snoopy caches**
  - —each coherence operation is sent to all processors
  - —hurts scalability
- **Why not send coherence requests to only those processors that need to be notified?**

Directory-based Schemes





Centralized directory

Distributed directory

Some Directory Implementation Alternatives

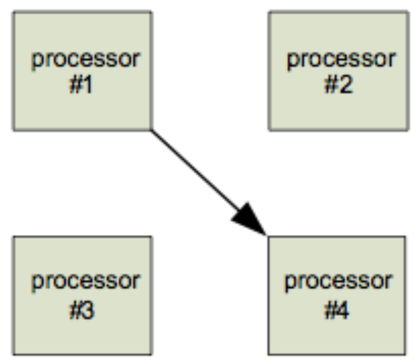
- Bit vector
  - —presence bit for each cache line along with its global state
- Pointer set
  - —limited set of pointers (node ids)
    - – less overhead than full map
  - —issue: widespread sharing

Intel QuickPath Home Snoop

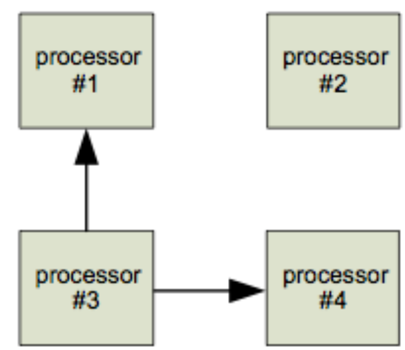
Labels:	P1 is the requesting caching agent P2 and P3 are peer caching agents P4 is the home agent for the line
Precondition:	P3 has a copy of the line in either M, E or F-state

MESIF protocol (Intel): Modified (M), Exclusive (E), Shared (S), Invalid (I) and Forward (F)

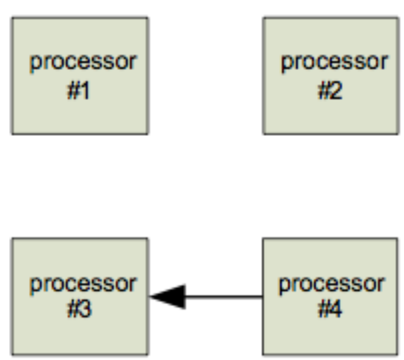
Step 1.
P1 requests data which is managed by the P4 home agent. (Note that P3 has a copy of the line.)



Step 3.
P3 responds to the snoop by indicating to P4 that it has sent the data to P1. P3 provides the data back to P1.



Step 2.
P4 (home agent) checks the directory and sends snoop requests only to P3.



Step 4.
P4 provides the completion of the transaction.

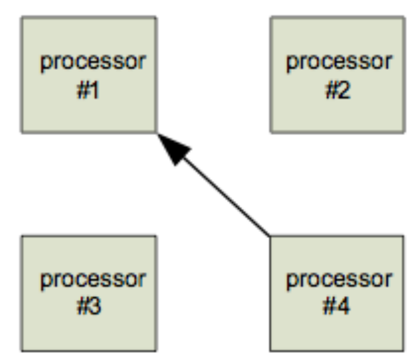


Figure credits: Introduction to Intel QuickPath Interconnect in Weaving High Performance Multiprocessor Fabric, Robert A. Maddox, Gurbir Singh, and Robert J. Safranek, Intel Press

AMD’s HT Assist

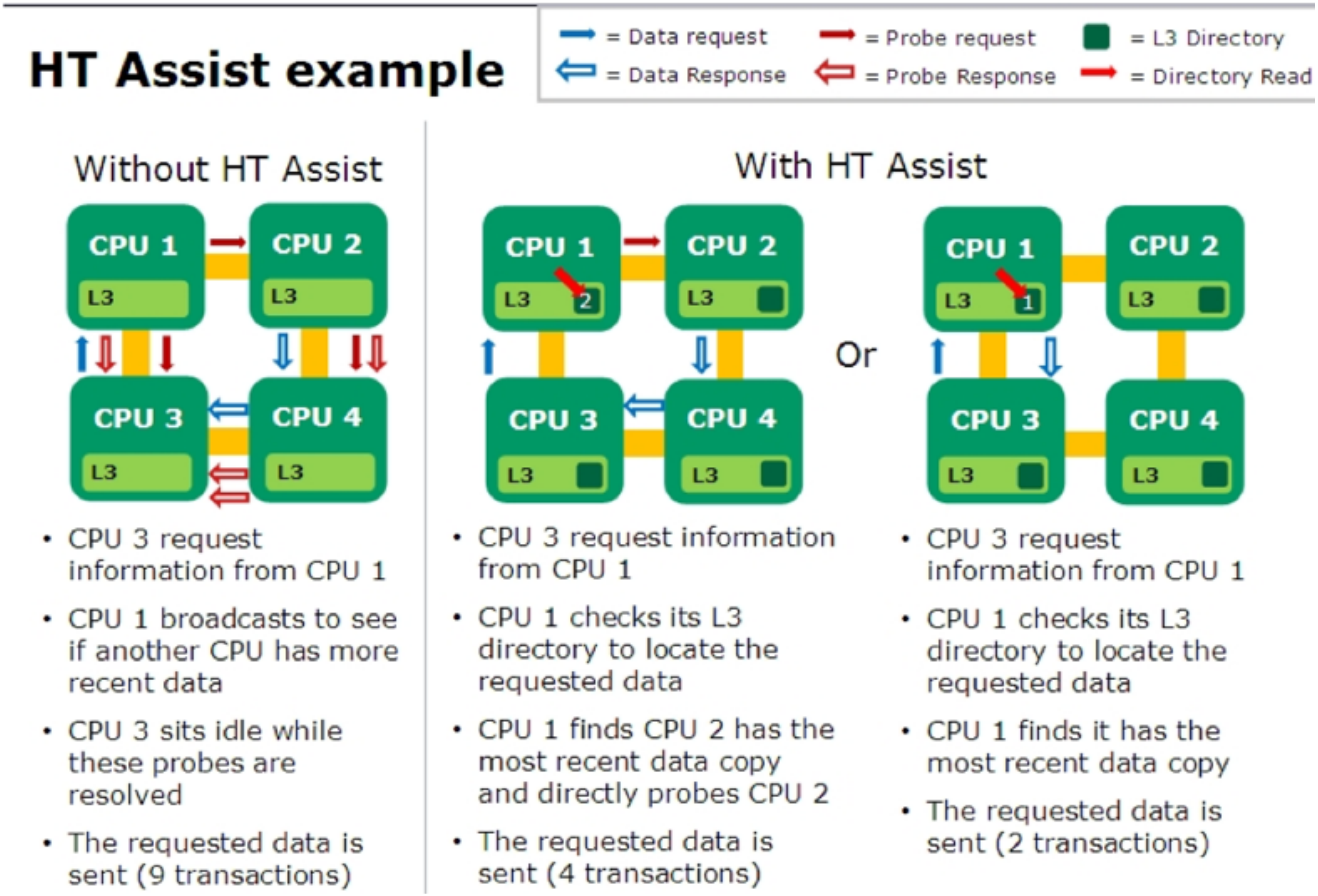


Figure source: http://blogs.zdnet.com/perlow/?p=10187

Coherence on Intel Platforms Today

- **Ultra Path Interconnect**
  - —point-to-point interconnect replaced QuickPath in Xeon Skylake-SP platforms in 2017
  - —only supports directory-based coherency using a home snoop coherency protocol
- **Improvements beyond Quickpath**
  - —power efficiency: adds a new low-power state
  - —transfer efficiency: new packetization format
  - —scalability: protocol layer does not require preallocation of resources
- **Combined caching and home agent**
  - —manages of coherency across multiple processors
  - —per core logic for handling snoops from local and remote processor cores

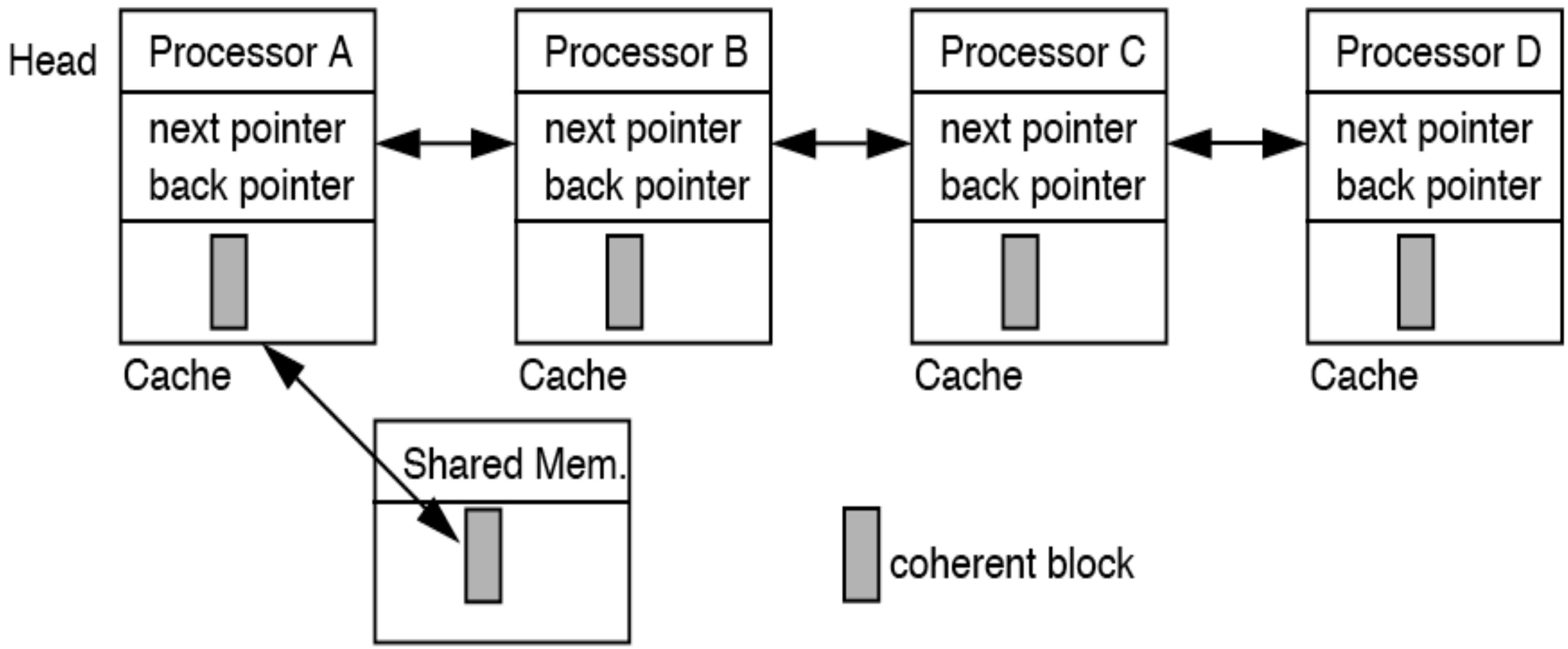
Paul Alcorn. Intel Xeon Platinum 8176 Scalable Processor Review. July 11, 2017. https://www.tomshardware.com/uk/reviews/intel-xeon-platinum-8176-scalable-cpu,5120-4.html

Performance of Directory-based Schemes

- Bits to store the directory may add significant overhead
  —think about scaling to many processors
    - data bits per cache block vs. presence bits per cache block
- Underlying network must carry all coherence requests
- Directory becomes a point of contention
  —distributed directory schemes are necessary for scalability

Scalable Coherent Interface

Linked-list based distributed directory scheme



ANSI/IEEE Std1596-1992

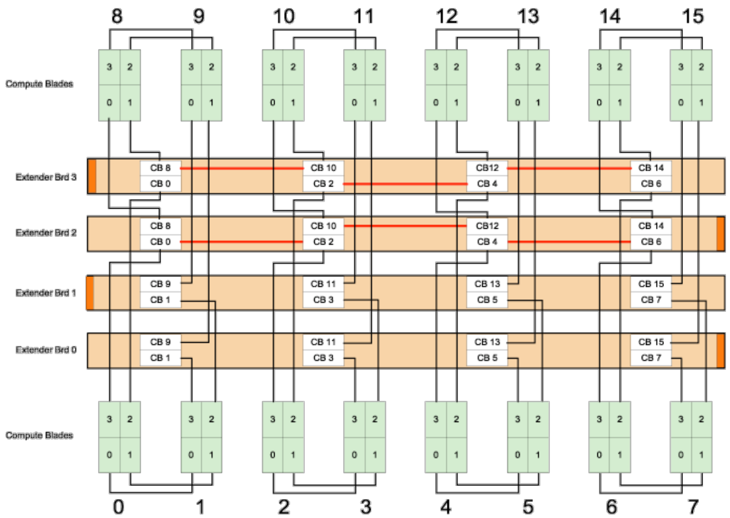
Figure credit: http://mufasa.informatik.uni-mannheim.de/Lectures/WS0506/RA2/script_pdf/sci.pdf

Product: Dophin Interconnect Solutions D333 PMC 64/66 SCI ADAPTER (http://bit.ly/gH2i7o)

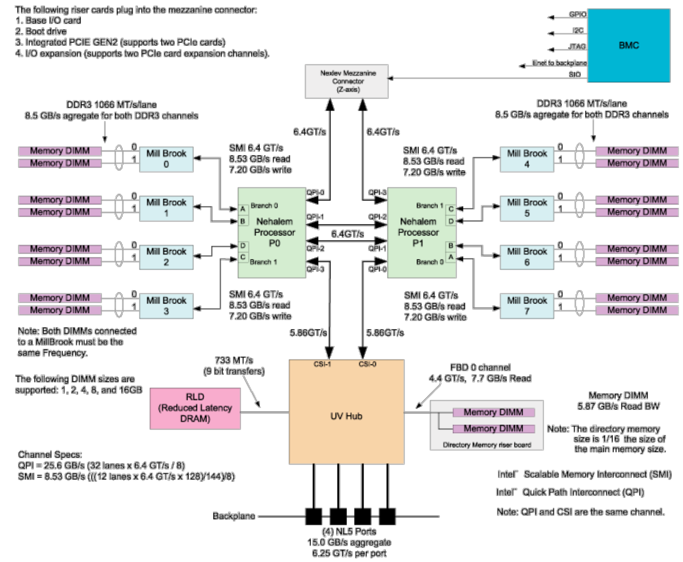
SGI Altix UV (2010)

- System overview: 32-2048 cores; cache coherent single system image

rack unit (16 nodes)



node blade



- Coherence
  - — directory-based coherence
    - – each 128B cache line has an entry in a directory
    - – directories distributed among the compute/memory blade nodes, like the data homes
    - – directory size = 1/16 main memory
    - – line states in a directory
      - unowned: when a line is not cached
      - exclusive: when only one processor has a copy
      - shared: when more than one processor has a copy
    - – bit vector indicates which nodes may contain a copy
  - — invalidation-based protocol: write invalidates copies & acquires exclusive ownership

Figure credits: http://bit.ly/hLX85a

SGI Altix UV (2010)

node blade

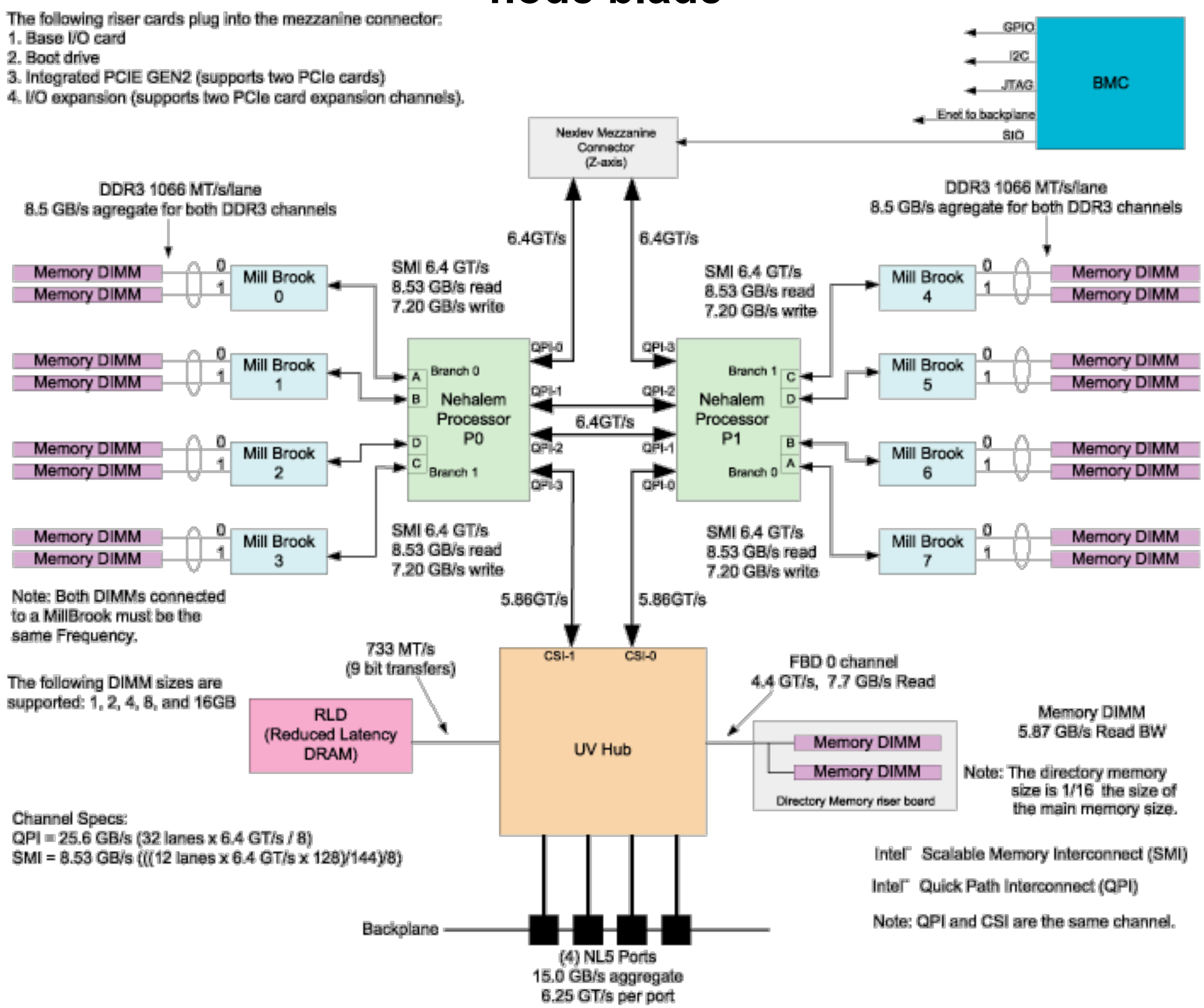
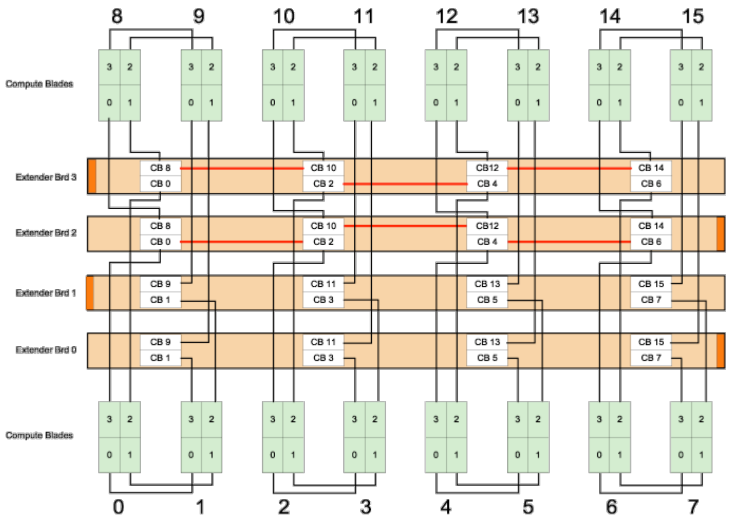


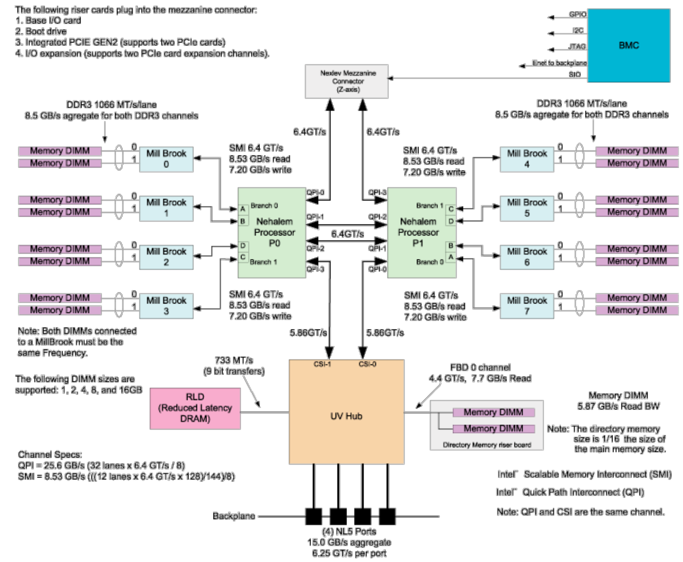
Figure credits: http://bit.ly/hLX85a

SGI Altix UV (2010)

- System overview: 32-2048 cores; cache coherent single system image

rack unit (16 nodes) **node blade**





- Coherence
  - — directory-based coherence
    - – each 128B cache line has an entry in a directory
    - – directories distributed among the compute/memory blade nodes, like the data homes
    - – directory size = 1/16 main memory
    - – line states in a directory
      - unowned: when a line is not cached
      - exclusive: when only one processor has a copy
      - shared: when more than one processor has a copy
    - – bit vector indicates which caches may contain a copy
  - — invalidation-based protocol: write invalidates copies & acquires exclusive ownership

Figure credits: http://bit.ly/hLX85a

SGI Altix UV (2010)

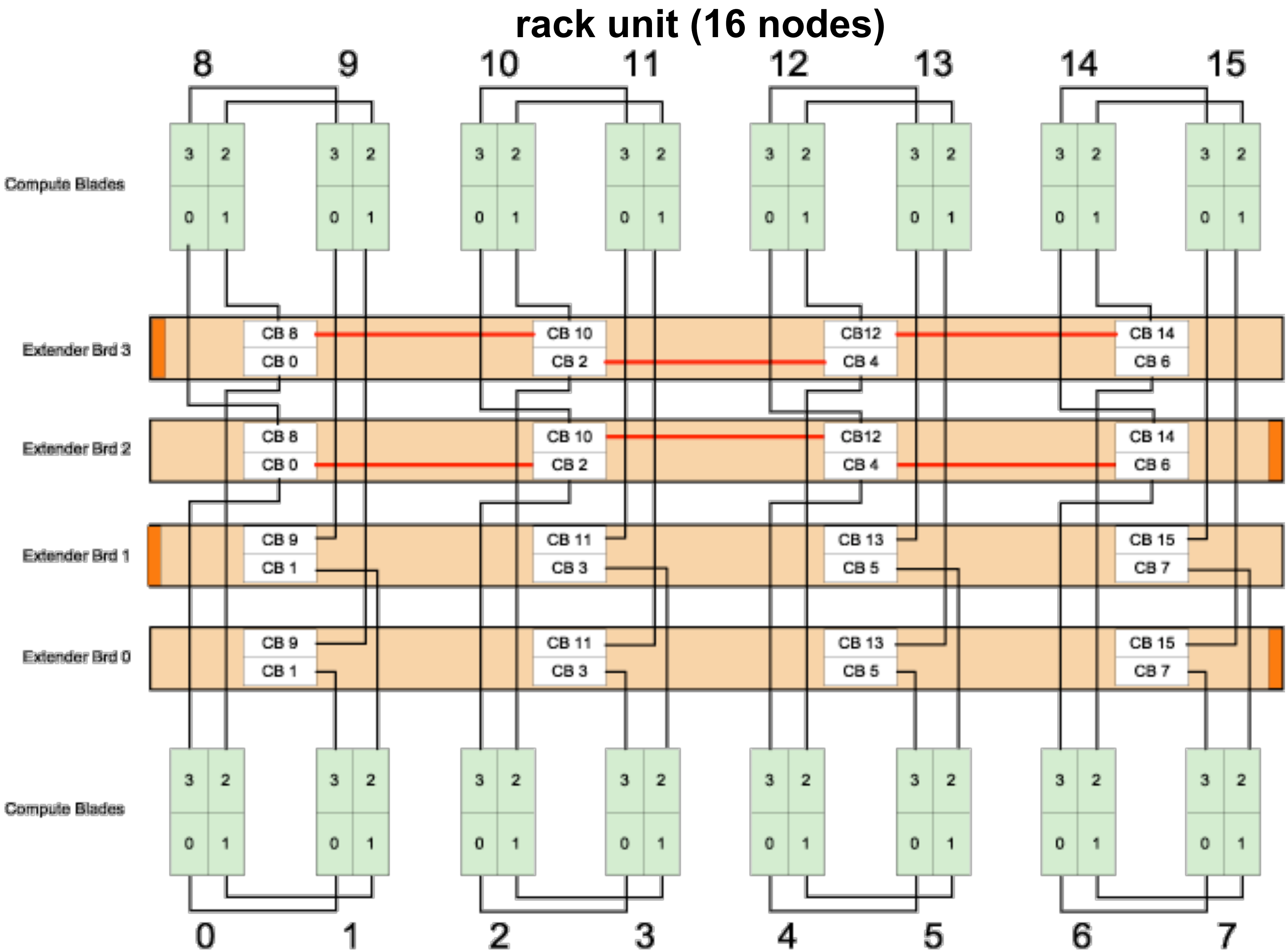


Figure credits: http://bit.ly/hLX85a

Memory Models and Weak Ordering

Sarita V. Adve and Hans-J. Boehm. 2010. Memory models: a case for rethinking parallel languages and hardware. CACM 53, 8 (August 2010), 90–101. DOI:https://doi.org/10.1145/1787234.1787255

What is a Memory Model?

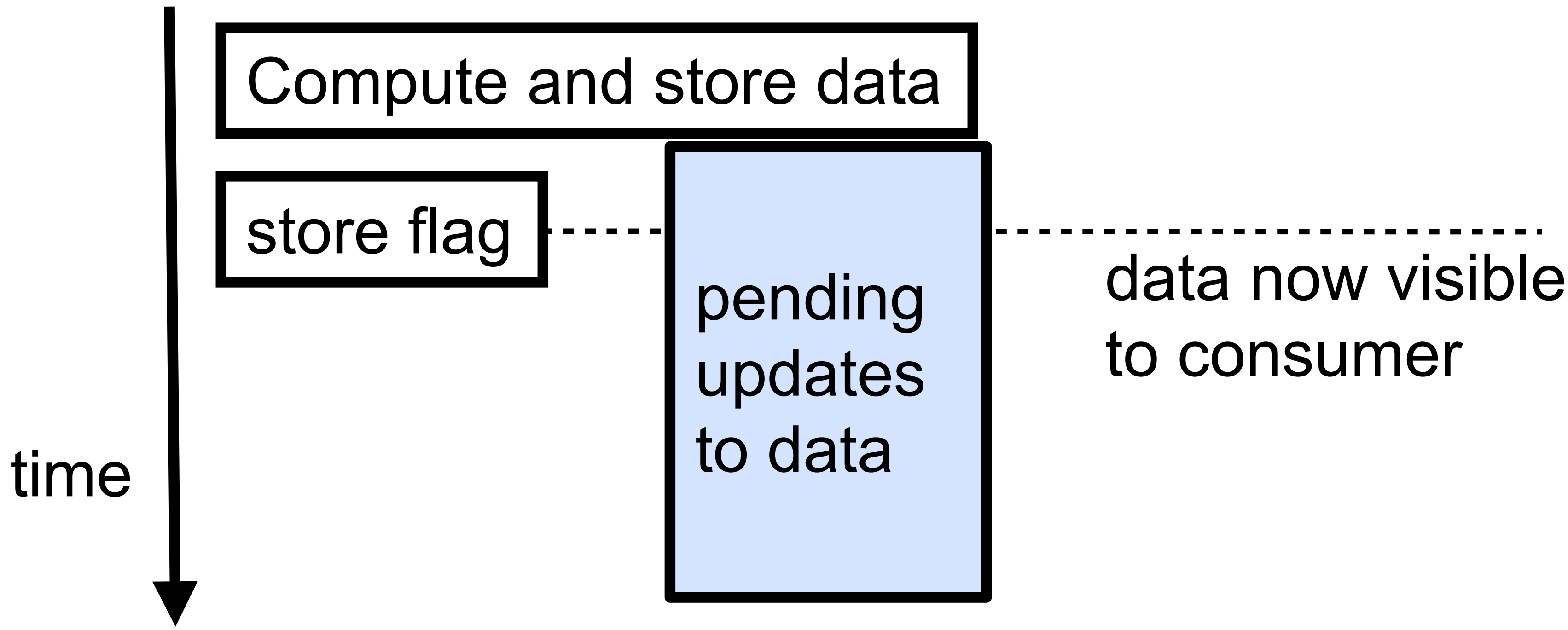
- A contract between a program and any hardware and software that reorders operations in a program execution
- In the context of parallelism, a memory model governs interactions between threads and shared memory
  - — atomicity, ordering, visibility
- Weak memory models: any load/store operation can be reordered with another, as long as the reordering doesn't affect single thread execution
  - —read/write, read/read, write/read, write/write
- Why weak memory models? performance!
  - —reordering of accesses by compiler, e.g., register allocation
  - —reordering by hardware
    - OOO execution: many operations in flight at once
    - write buffers, non-blocking caches, …
    - don't wait for operations to globally complete before continuing

Producer/Consumer Synchronization

- Example: using a global flag for synchronization between producer and consumer threads
  - —producer indicates that it is done with data by setting a flag
  - —consumer waits until flag is set before reading data
- Getting it right
  - —producer must not set flag until updates to data are visible to consumer
  - —both the producer and consumer must act to control weak ordering

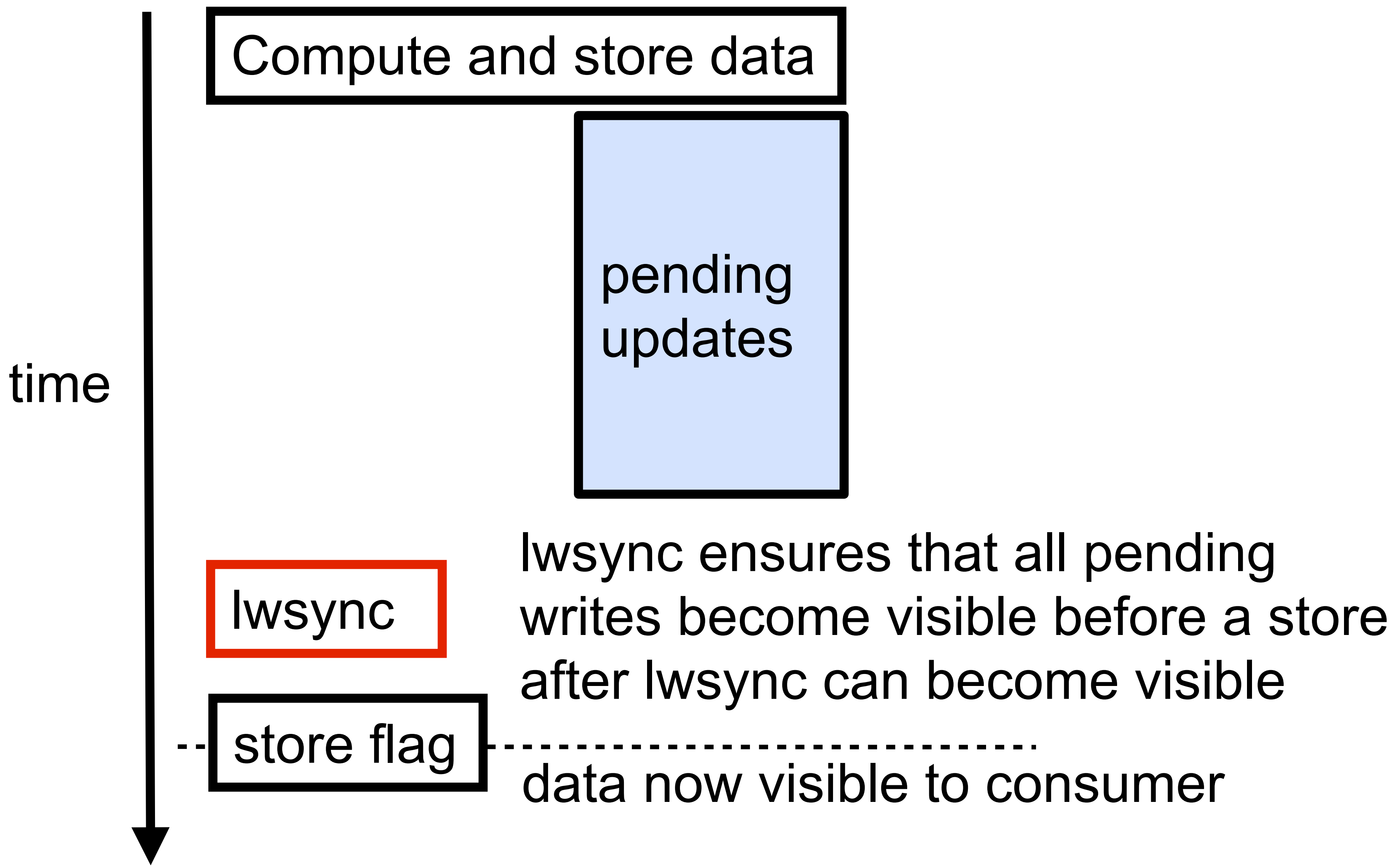
IBM Power Weak Memory Model: Producer

Incorrect way: without attention to weak ordering



IBM Power Weak Memory Model: Producer

Correct way: ensure writes complete before setting flag



IBM Power Weak Memory Model: Consumer

Incorrect way: without attention to weak ordering

Loop: load global flag
has global flag been set?
no: go to Loop
yes: fall through to Next

time

problem: consumer can speculatively execute code at Next before flag is set

producer stores flag

Next: use data

IBM Power Weak Memory Model: Consumer

Correct way: inhibit speculative reads until flag is set

Loop: load global flag
has global flag been set?
no: go to Loop
yes: fall through to Next

time

producer stores flag

Next: isync
use data

isync causes the processor to complete all previous instructions and discard instructions after the isync that may have begun execution

Java Memory Model

Why have a Memory Model for Java?

- Java supports threads that shared memory
- Must have a memory model to define program semantics
  - —determines the transformations the compiler can make
  - —specifies ordering guarantees that a compiler must preserve regardless of the underlying architecture

Sequential Consistency Revisited

- **Sequential consistency**
  - **—all data accesses appear to have been executed**
    - **atomically**
    - **in some sequential order**

      consistent with the order of operations in individual threads
  - **—corollary**
    - **each variable must appear to have only a single value at a time**

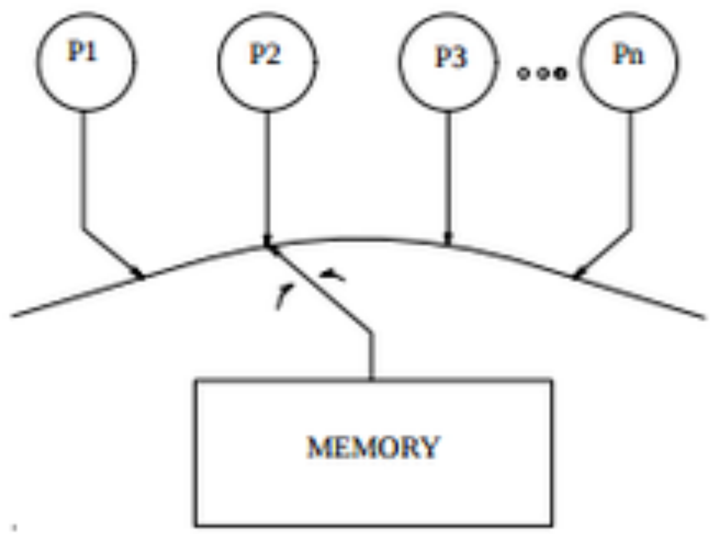


Figure credit: Sarita V. Adve, Kourosh Gharachorloo, Memory Consistency Models for Shared-Memory Multiprocessors. Computer Science Department, Stanford University Technical Report CSL-TR-95-685. December 1995.

Why Not Sequential Consistency for Java?

Precludes many optimizations important for performance

- **HW optimizations: store buffers, speculation, …**
- **Compiler optimizations**
  - **—register allocation**
  - **—common sub-expression elimination**
  - **—loop interchange or blocking**

all have the effect of reordering or eliminating memory operations

‘Out-of-thin-air’ Problem

- **Assume an incorrectly synchronized program**
- **After execution, could r1 == r2 == 42?**

Initially, x == y == 0

Thread 1	Thread 2
`r1 = x;`	`r2 = y;`
`y = r1;`	`x = r2;`

What if:

1. **thread 1 speculatively writes 42 to y**
2. **thread 2 reads 42 for y**
3. **thread 2 writes 42 for x**
4. **thread 1 reads 42 for x**
5. **thread 1 validates its write speculation for y**

Analysis of ‘Out-of-thin-air’ Problem

- **Should we disallow this ‘optimization’?**
- **Why not let this error be undefined?**
- **Consider the Java class loader**
  - —cornerstone of the Java virtual machine
  - —describes behavior of converting a named class into the bits responsible for implementing that class
- **Suppose ‘42’ was `&loadClass`?**
  - —unintentional errors => violate safety
  - —intentional errors => security risk

Lazy Initialization

```
class Foo {
    private Helper helper;
    public Helper getHelper() {
        if (helper == null) {
            helper = new Helper();
        }
        return helper;
    }
}
```

Clearly is not thread safe

Ensuring Thread Safety?

Two things to consider

- **synchronization**
  - **if used correctly, can provide mutual exclusion to shared data**
- **data visibility**
  - **writing a value to a variable from a thread doesn't mean it will be immediately visible in a different thread**

Mechanisms in Java

- **Synchronization**
  - —synchronized keyword for methods and blocks
    - permits one thread to enter at any given time
      - reentrant: thread can call a synch method within a synch method
    - synchronized block specifies object providing the lock
  - —explicit Lock: finer control
- **Data Visibility**
  - —final variable
    - can only be initialized only once
      - initializer or assignment statement
    - final modifier applied to a field or variable only determines the properties of the value, not the referenced object
      - public final Point p;
        - after p is assigned, p.x and p.y can be still be assigned
  - —volatile variable
    - never cached: all reads and writes go straight to memory
    - a write to a volatile variable v synchronizes-with all subsequent reads of v by any thread

Approach 1: Synchronized Method

- **Idea: guarantee thread safety by mutual exclusion using a synchronized method to control access to helper**

```
// extend to multithread -threaded version,
add synchronized on method
class Foo {
   private Helper helper;
   public synchronized Helper getHelper(){
       if (helper == null) {
           helper = new Helper();
       }
       return helper;
   }
 }
```

critical section highlighted in blue

Approach 2: Double-checked Locking (DCL)

- **Idea: synchronize initialization, but not access**
- **Why? improve performance**

one possible execution sequence

```java
class Foo {
    private Helper helper;
    public Helper getHelper() {
        if (helper == null) {
            synchronized(this) {
                if (helper == null) {
                    helper = new Helper();
                }
            }
        }
        return helper;
    }
}
```

it seems to work…

Approach 2: Double-checked Locking (DCL)

- **Idea: synchronize initialization, but not access**
- **Why? improve performance**

how about this sequence?

```
class Foo {
   private Helper helper;
   public Helper getHelper() {
       if (helper == null) {
           synchronized(this) {
               if (helper == null) {
                   helper = new Helper();
               }
           }
       }
       return helper;
   }
}
```

Problem:

compiler or hardware could reorder the writes initializing helper and its fields

some fields might be initialized after the write to helper becomes visible

Approach 3: DCL + volatile

volatile ensures that the actions that happen before the write to helper in the code must, when the program executes, actually happen before the write to helper

```
class Foo {
    private volatile Helper helper;
    public Helper getHelper() {
        if (helper == null) {
            synchronized(this) {
                if (helper == null) {
                    helper = new Helper();
                }
            }
        }
        return helper;
    }
}
```

Approach 4: DCL + volatile + caching

- Local variable ‘result’ reduces access to volatile variable ‘helper’. after ‘helper’ has been initialized, (most of the time), the volatile field is only accessed once (due to "return result;" instead of "return helper”)
- Can improve the method's overall performance by as much as 25 percent.

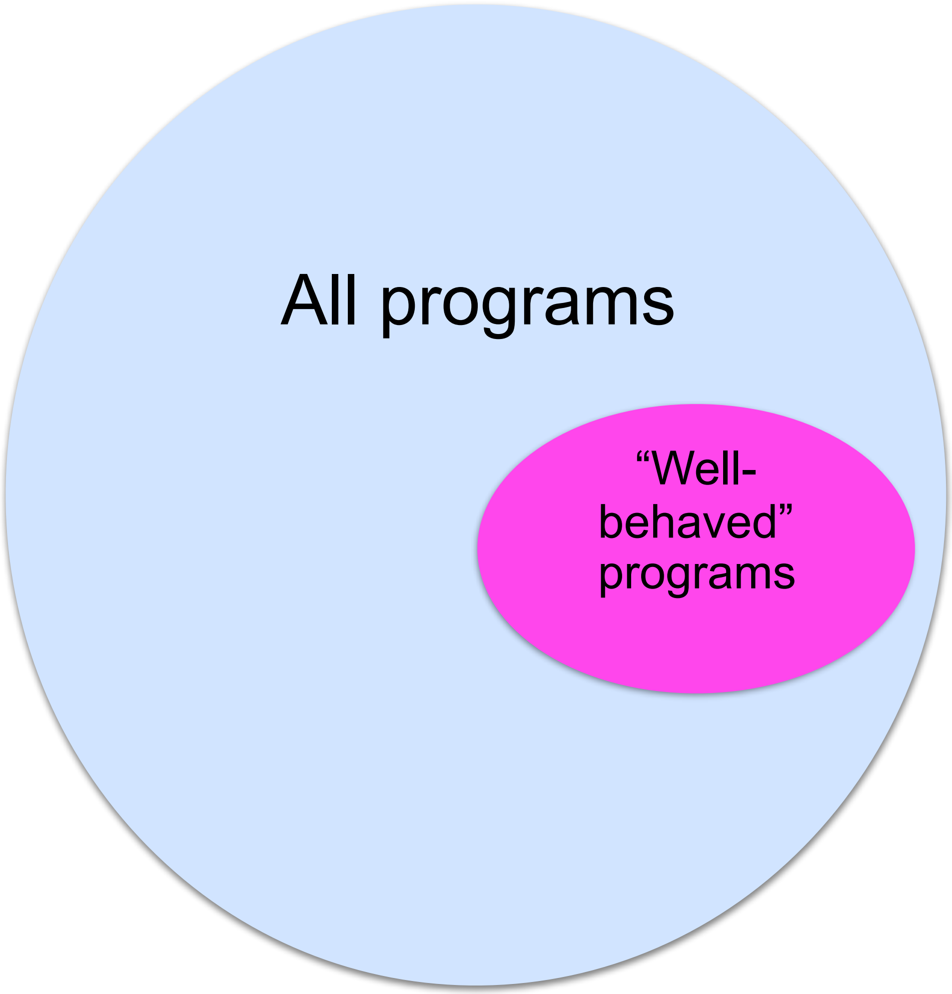
```
class Foo {
   private volatile Helper helper;
   public Helper getHelper() {
       Helper result = helper;
       if (result == null) {
           synchronized(this) {
               result = helper;
               if (result == null) {
                   helper = result =
                       new Helper();
                }
            }
        }
        return result;
    }
}
```

Terminology

- **Data race**
  - —two concurrent accesses to the same shared variable are said to be conflicting if at least one access is a write
- **Correctly synchronized**
  - —a program is said to be correctly synchronized or data-race-free iff all sequentially consistent executions of the program are free of data races

Java Memory Model

- **Goal**
  - —sufficiently easy to understand and use
  - —permit important optimizations used by compilers and hardware
- **Guarantees**
  - —“Well-Behaved” programs observe sequentially consistency
  - —“Incorrect” programs
    - – may contain data races
    - – still, no out of thin air result



Shared Memory Synchronization

Goal: Coordinate Shared-memory Computation

- Coordinate sharing among all threads
  - support mutually exclusive access to shared data
  - ensure threads advance through computation phases together
- Coordinate pairwise sharing
  - e.g. producer-consumer sharing
- Synchronization in prior lectures
  - locks
    - e.g. pthread_mutex_lock/unlock, omp_set_lock/unset_lock
  - barriers
    - team barrier implicit at end of OpenMP parallel loops
      - no thread can execute code following a parallel loop until all iterations have finished (unless nowait specified)

Approaches: Spinning vs. Blocking

- **Blocking**
  - —what: suspend execution until a resource is available
  - —advantage: frees up a processor for useful work
    - – important when # threads > # cores
  - —disadvantage: longer latency (context switch at a minimum)
  - —examples: pthread_mutex_lock/unlock/trylock
- **Spinning**
  - —what: repeatedly test a condition until it becomes true
  - —advantage: low latency
  - —disadvantage: ties up a processor core
    - – may displace useful computation
  - —examples: pthread_spin_lock/unlock/trylock
- **Rule of thumb**
  - —use spinning in a <u>dedicated</u> environment if # threads <= # cores
  - —use blocking in <u>shared</u> environment or if # threads > # cores

Primitives for Shared-memory Synchronization

- **Normal instructions**
  - —load
  - —store
- **What are their uses?**
  - —load: test a variable value
  - —store: useful when there is a single writer
    - – e.g., setting a boolean flag
- **Limitations**
  - —multiple writers of a variable yield unpredictable values
- **Solution: atomic operations (next slide)**

Atomic Primitives for Synchronization

Atomic read-modify-write primitives

- **test_and_set(Word &M)**
  - writes a 1 into M
  - returns M's previous value
- **swap(Word &M, Word V)**
  - replaces the contents of M with V
  - returns M's previous value
- **fetch_and_Φ(Word &M, Word V)**
  - Φ can be ADD, OR, XOR, ...
  - replaces the value of M with Φ(old value, V)
  - returns M's previous value
- **compare_and_swap(Word &M, Word oldV, Word newV)**
  - if (M == oldV) M $\leftarrow$ newV
  - returns TRUE if store was performed
  - universal primitive

See http://gcc.gnu.org/onlinedocs/gcc-4.1.0/gcc/Atomic-Builtins.html for use in practice

A Simple Lock with Test & Set

```
type Lock = (unlocked, locked)

procedure acquire_lock(Lock *L)
  loop
    // NOTE: test and set returns old value
    if test_and_set(L) == unlocked
       return

procedure release_lock(Lock *L)
  *L = unlocked
```

Synchronization

- **Initialize**
  - —prepare state of sync variable for first use
- **Signal**
  - —notify one or more threads with a sync variable state change
- **Acknowledge**
  - —optional handshake to prevent unbounded signaling
- **Reinitialize**
  - —adjust state of sync variable

Building Blocks

- **Single use flag variable**
  - initialized to false at program launch
  - producer sets a flag to true
  - consumer eventually notices
- **Counter**
  - initialized to zero
  - single writer: increment with non-atomic add
  - multiple writers:
    - if writer needs intermediate value, use fetch_and_add
    - otherwise, use atomic add
- **Pointers**
  - initialize to null
  - update with atomic_swap or compare_and_swap
    - retrieve old value; (conditionally for CAS) store new value

Considerations

- **Reinitialization can be tricky**
  - —techniques: sense switching, paired data structure
- **Interconnect traffic and contention can degrade performance**
  - —be careful with spin waiting on variables
  - —advanced technique: local spinning

Technique: Sense Switching

- Problem: reinitialization of a flag is often problematic
  - —can the reinitialization race with a flag inspection?
- Approach: don’t reinitialize, sense switch!
  - —in even synchronization rounds, wait for a flag to become true
  - —in odd synchronization rounds, wait for a flag to become false

Exercise: Design a Simple Barrier

- Each processor indicates its arrival at the barrier
  - —updates shared state
- Busy-waits on shared state to determine when all have arrived
- Once all have arrived, each processor is allowed to continue

Sense-reversing Centralized Barrier

```
integer count = P
bool sense = true

thread_local bool local_sense = true

void central_barrier() {
  // each processor toggles its own sense
  local_sense = not local_sense
  if (fetch_and_add(&count,-1) == 1)
    count = P
    sense = local_sense // last processor toggles global sense
  else
    repeat until sense == local_sense
}
```

Technique: Paired Data Structure

- **Use alternating sets of variables to avoid overlapping updates**
- **Motivating example**
  - —“dissemination barrier” uses only flag setting to achieve a barrier
    - – each processor has log P flags
    - – synchronization proceeds in log P rounds
      - each round
        - set a flag for another processor
        - spin until your flag is set
  - —one could use sense switching for next barrier phase
  - —but can’t keep adjacent barrier phases from interfering
    - – one thread may be stall while spinning in phase k
    - – what if another thread then flips the sense for phase k+1
- **Solve the problem with a paired data structure: separate flags for odd and even phases**

Spin Waiting and Interconnect Traffic

Considerations

- How many data transfers over the interconnect will occur?
  - —is the machine cache coherent?
  - —what coherence protocol is used?

- Let's first consider coherence on quad-processor nodes
  - —cache coherence protocols
    - – Intel: home snoop, source snoop (2009)
    - – AMD: HT Assist (2009)

Avoid Spin Waiting over the Interconnect

- How?
  - —don’t have multiple threads spin wait on a shared variable that will change multiple times per synchronization operation
- For instance
  - —avoid spin waiting on
    - – a barrier count that others are adjusting with atomic_add
      - use a barrier flag instead
    - – a lock variable that others will toggle with test and set
      - use a link-list-based lock (local spinning)
        - e.g. MCS lock

Producer Consumer Synchronization

- **Data structure**
  - —int64 produced, consumed;
- **Operations**
  - —producer
    - – produced = produced + 1;
  - —consumer spins
    - – while (produced < consumed);
    - – consumed ++;
- **Bounded signaling**
  - —producer can spin
    - – while consumed + SLACK < produced

References - I

- Adapted from slides "Parallel Programming Platforms" by Ananth Grama
- Based on Chapter 2 of "Introduction to Parallel Computing" by Ananth Grama, Anshul Gupta, George Karypis, and Vipin Kumar. Addison Wesley, 2003
- Josep Torrellas. "Cache Coherence," Slides for TAMU CPSC 564 Lecture, 2003. http://bit.ly/fWENU2
- J. Mellor-Crummey, M. L. Scott: Synchronization without Contention. ASPLOS, 269-278, 1991.
- J. Mellor-Crummey and M. Scott. Algorithms for scalable synchronization on shared-memory multiprocessors. ACM Transactions on Computer Systems, 9(1):21-65, Feb. 1991.
- H. Hum et al. US Patent 6,922,756. July 2005. http://bit.ly/gQNkRR
- SGI Altix UV 1000 System User's Guide, Chapter 3. http://bit.ly/hLX85a
- PowerPC storage model and AIX programming. http://www.ibm.com/developerworks/systems/articles/powerpc.html

References - II

- C++ Memory Model and Atomics
  - https://channel9.msdn.com/Shows/Going+Deep/Cpp-and-Beyond-2012-Herb-Sutter-atomic-Weapons-1-of-2
  - https://channel9.msdn.com/Shows/Going+Deep/Cpp-and-Beyond-2012-Herb-Sutter-atomic-Weapons-2-of-2