Parallel Computing Platforms:

Coherence, Ordering, & Synchronization

John Mellor-Crummey

Department of Computer Science
Rice University

johnmc@rice.edu

Topics for Today

- Cache coherence
  - —definition
  - —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
  - —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

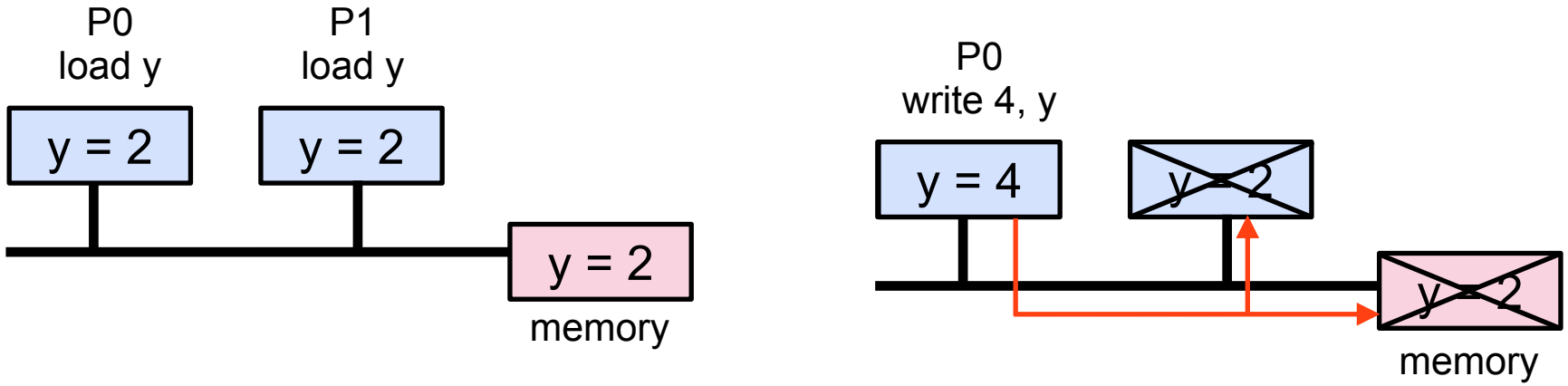
Approaches to Cache Coherence

- **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

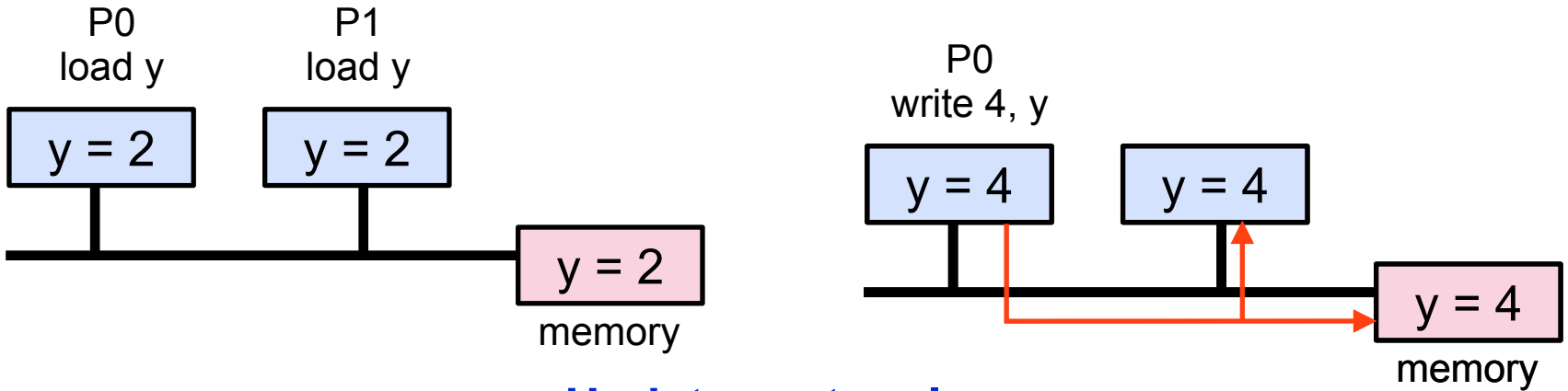
      Intel Scalable Cloud Computer (SCC), 2010

Cache Coherence Protocols

When changing a variable's value: invalidate or update all copies



Invalidate protocol



Update protocol

Update and Invalidate Protocols

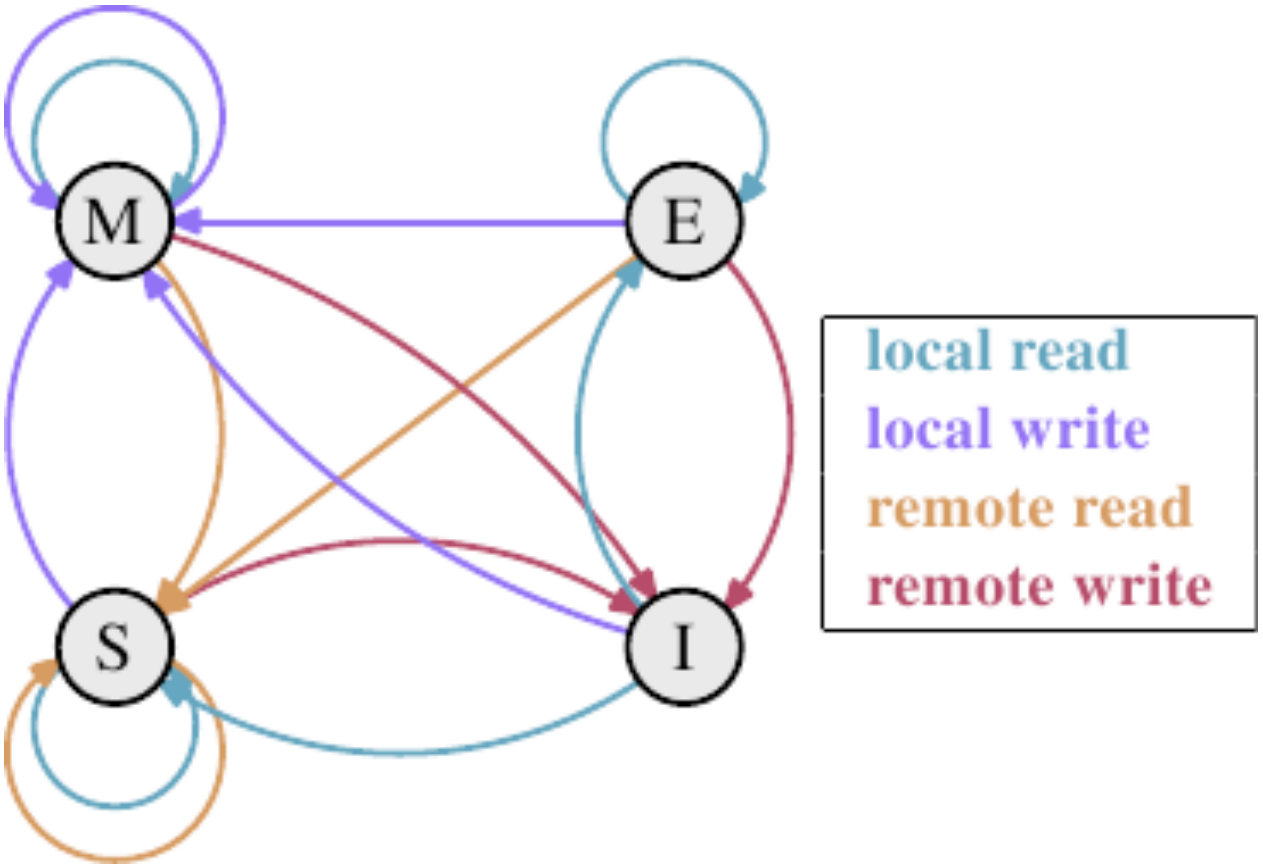
- Cost-benefit tradeoff depends upon traffic pattern
  - —invalidation is worse when
    - single producer of data and many consumers
  - —update is worse when
    - multiple writes by one CPU before data is read by another
    - a cache is filled with data that is not read again
      - e.g. leftovers after thread or process migration
- Both protocols suffer from false sharing overheads
  - —line accessed by multiple readers and writers
  - —each CPU accesses disjoint data
  - —false sharing overhead = coherence cost in this case
- Modern machines use invalidate protocols as the default

Using Invalidate Protocols

- Each copy of a data item 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

State Diagram for 4-state Invalidate Protocol

MESI states: Modified, Exclusive, Shared, and Invalid



	M	E	S	I
M				
E				
S				
I				

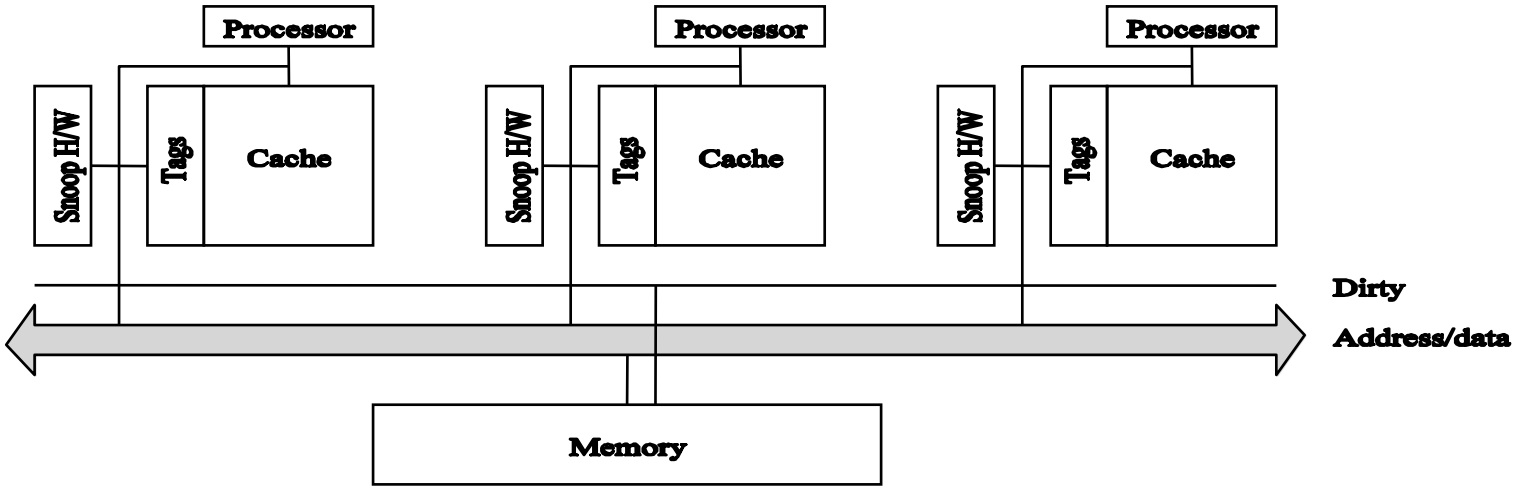
Figure credit: http://lwn.net/images/cpumemory/cpumemory.13.png

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

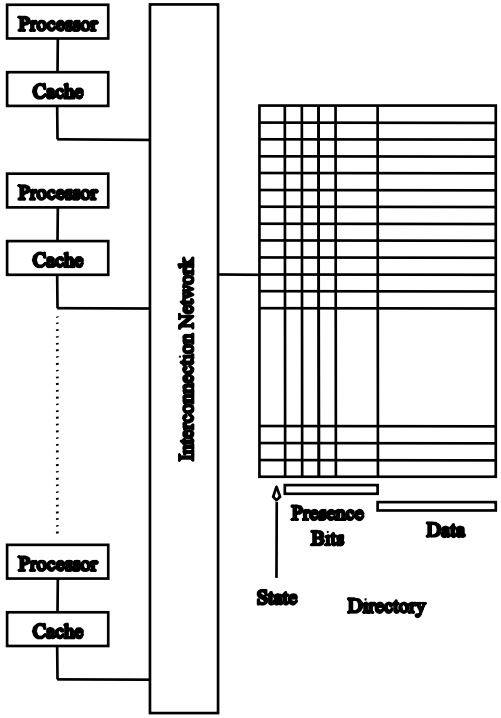
Operation of Snoopy Caches

- 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

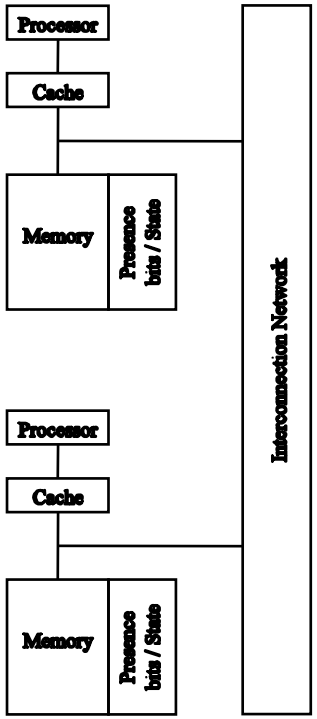
The Cost of Coherence

- **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

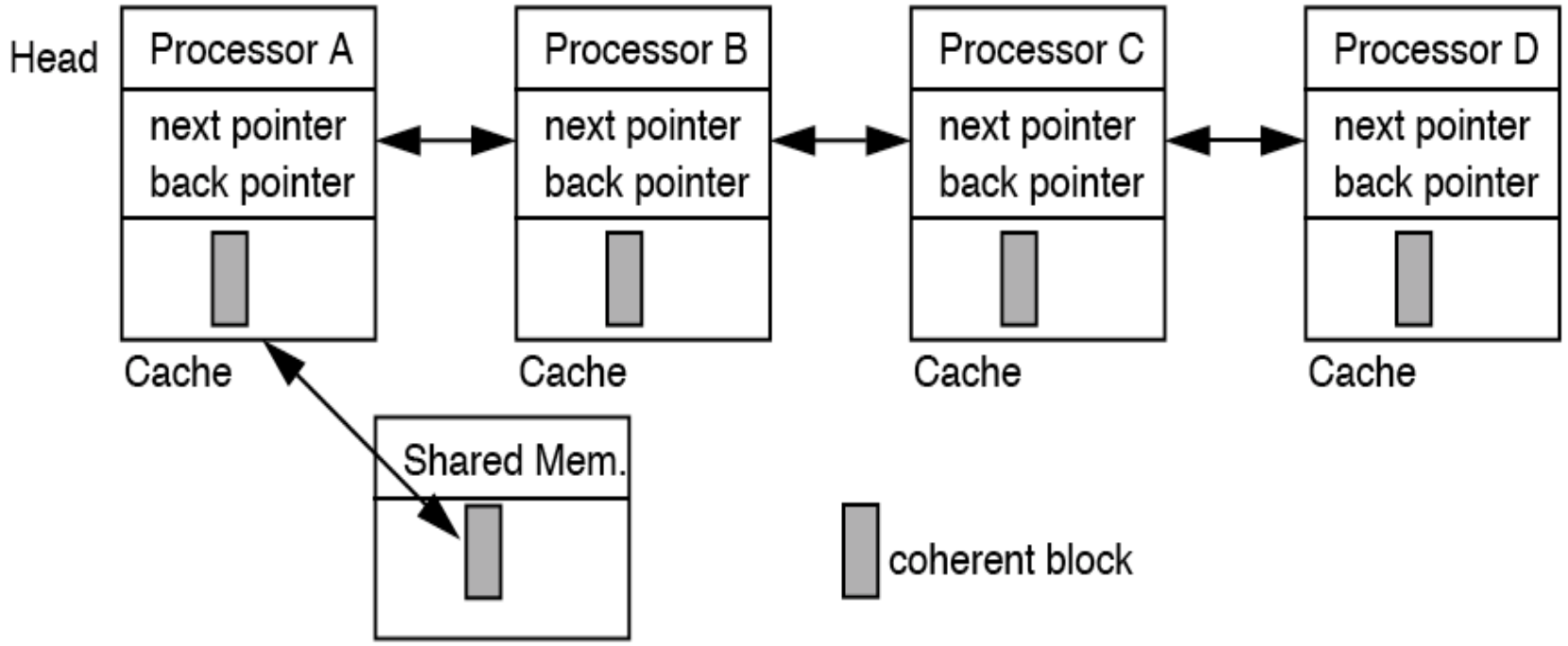
- **Bit vector**
  - —presence bit for each cache line along with its global state
- **Pointer set**
  - —limited set of pointers (node ids)
    - less overhead than full map
  - —issue: widespread sharing

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

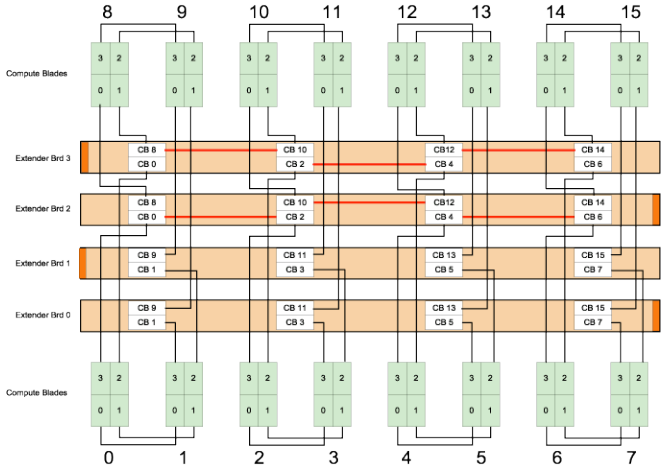
Figure credit: http://mufasa.informatik.uni-mannheim.de/Lectures/WS0506/RA2/script_pdf/sci.pdf

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

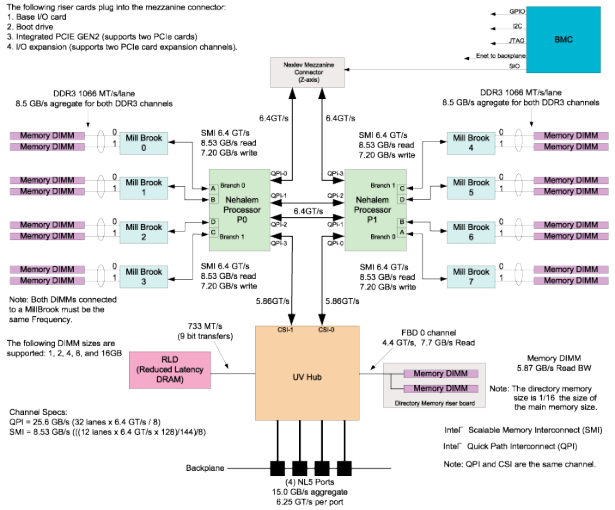


Figure credits: http://bit.ly/hLX85a

- **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**

SGI Altix UV (2010)

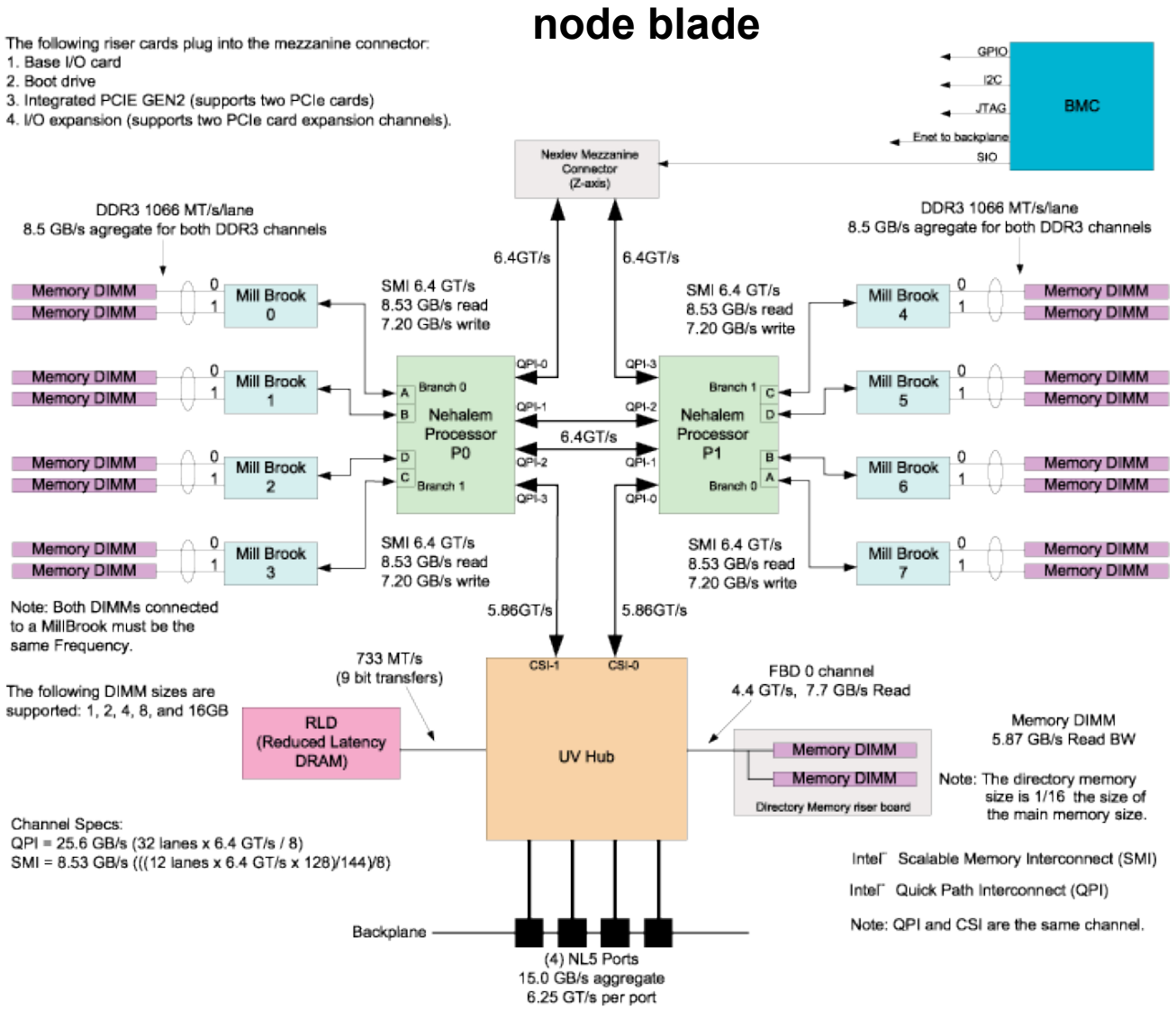
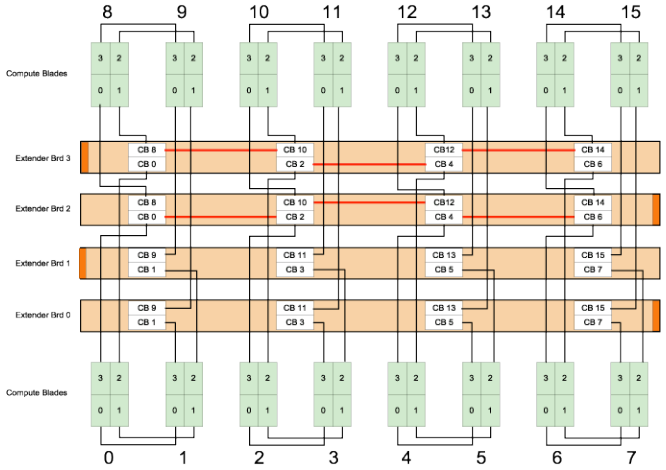


Figure credits: http://bit.ly/hLX85a

SGI Altix UV (2010)

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

rack unit (16 nodes)



node blade

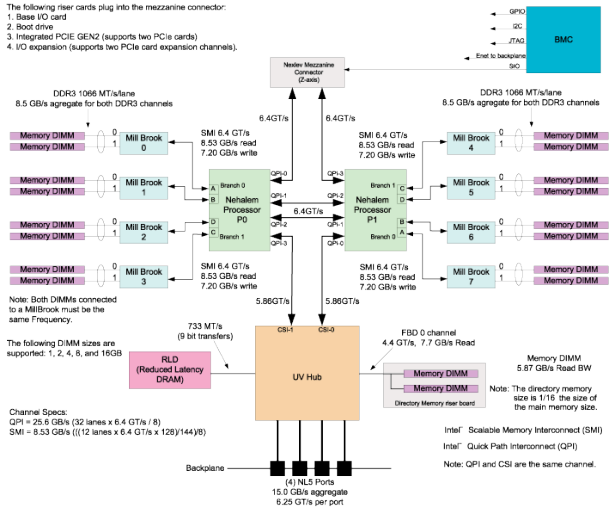


Figure credits: http://bit.ly/hLX85a

- **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**

SGI Altix UV (2010)

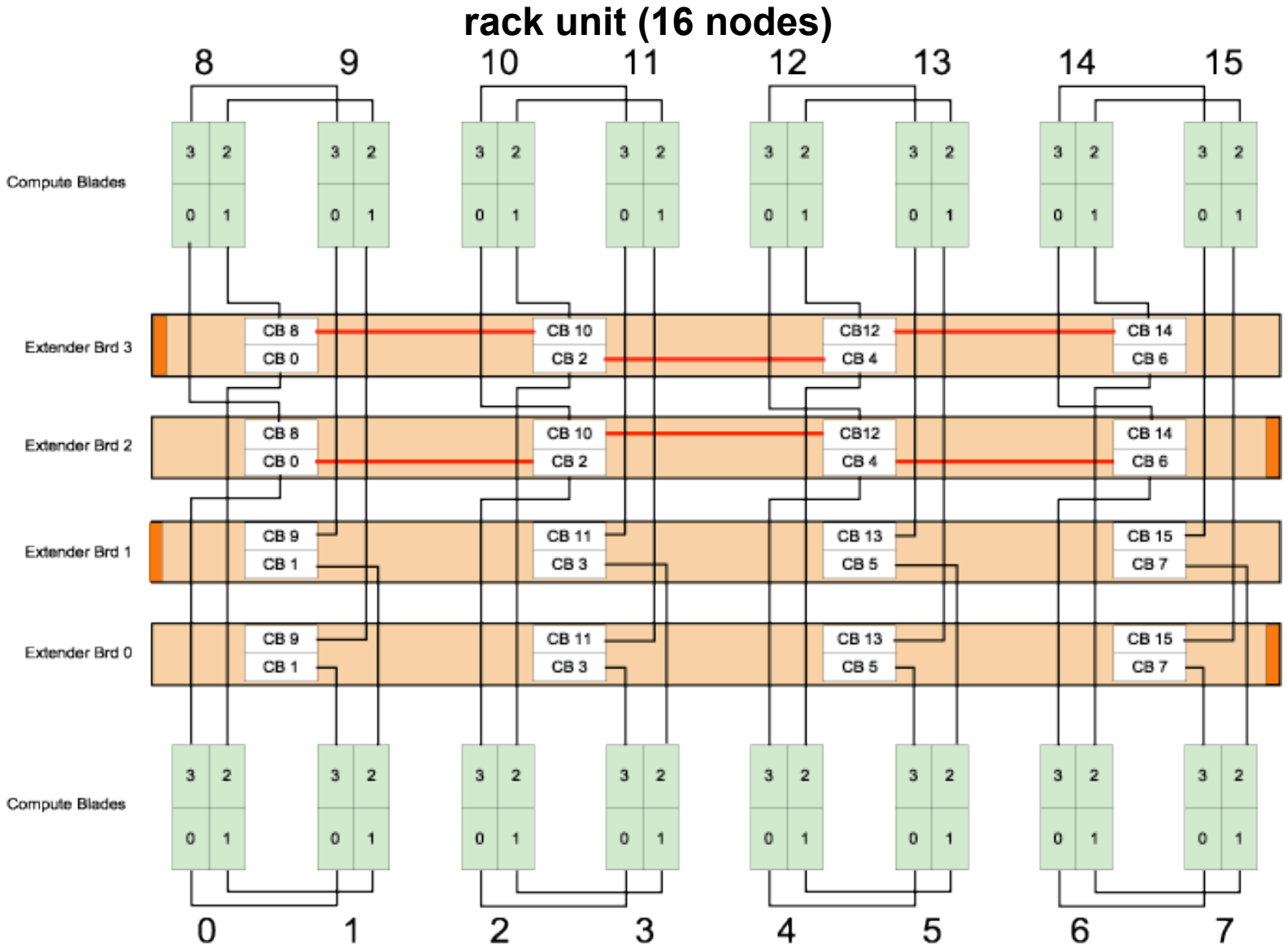
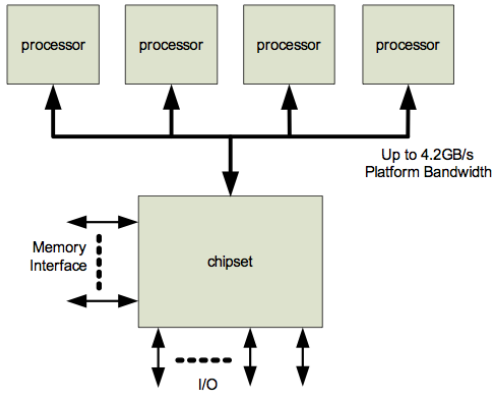
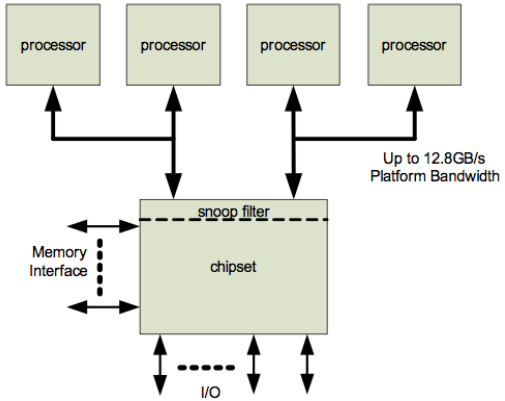


Figure credits: http://bit.ly/hLX85a

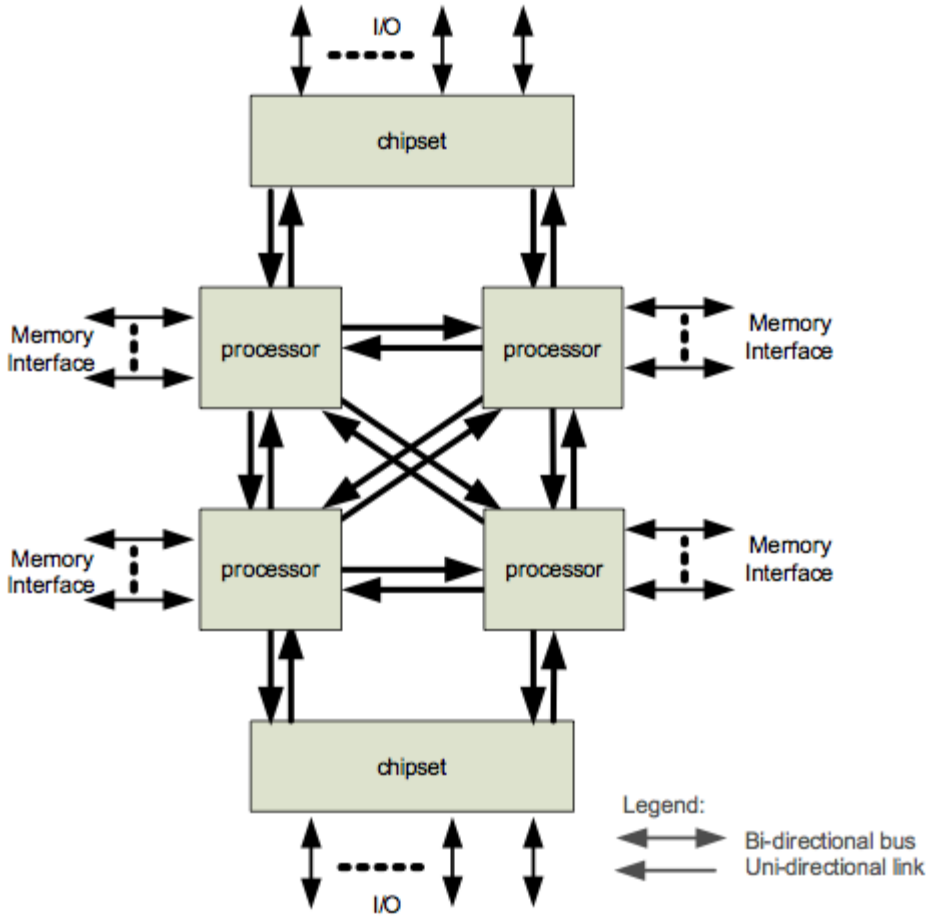
Evolution of Node Interconnects



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: Modified, Exclusive, Shared, Invalid and Forward
- 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 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)

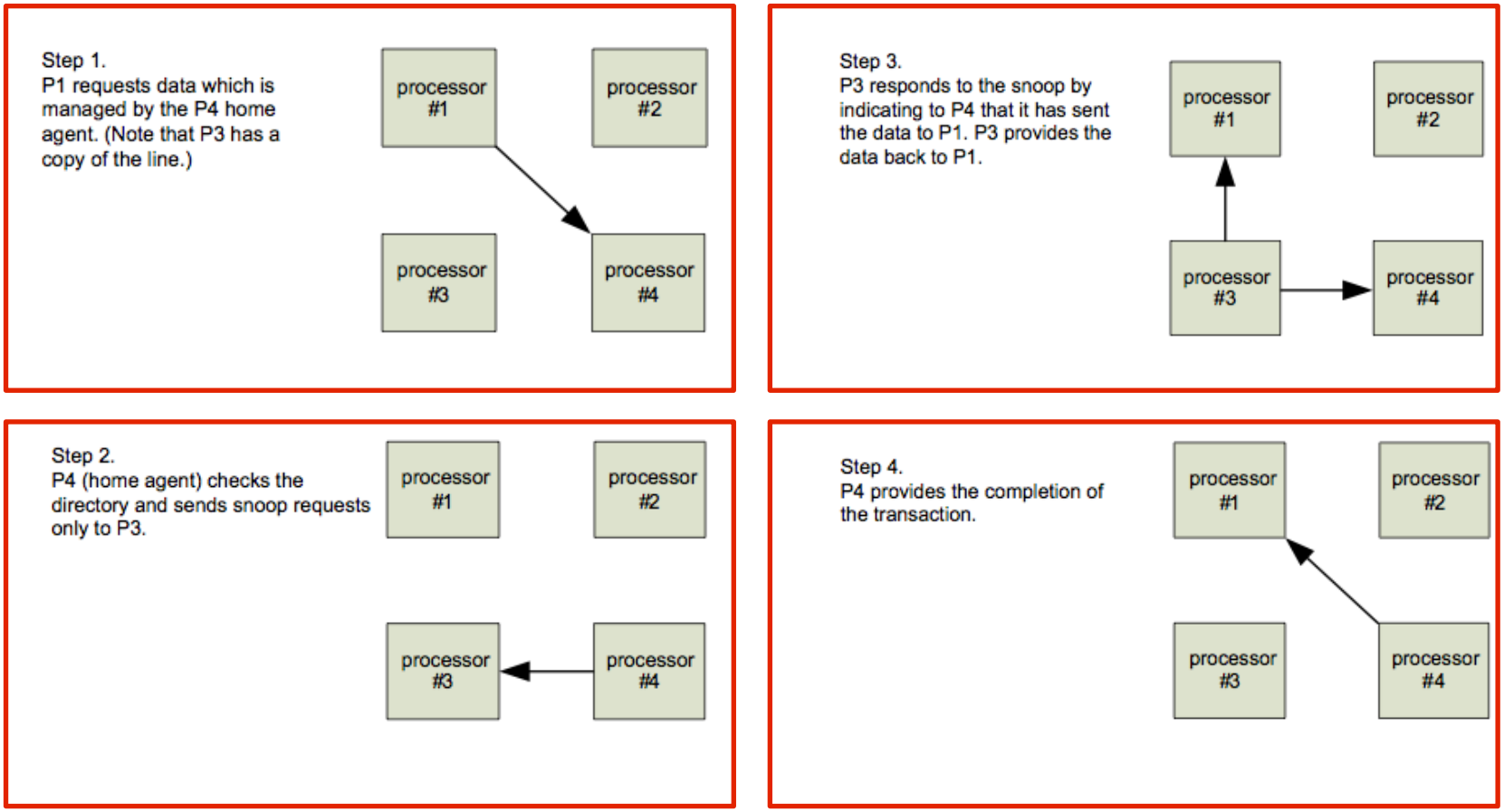
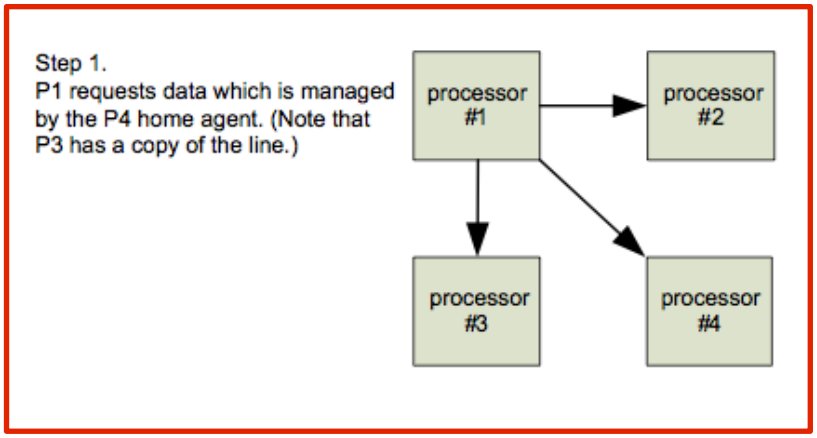


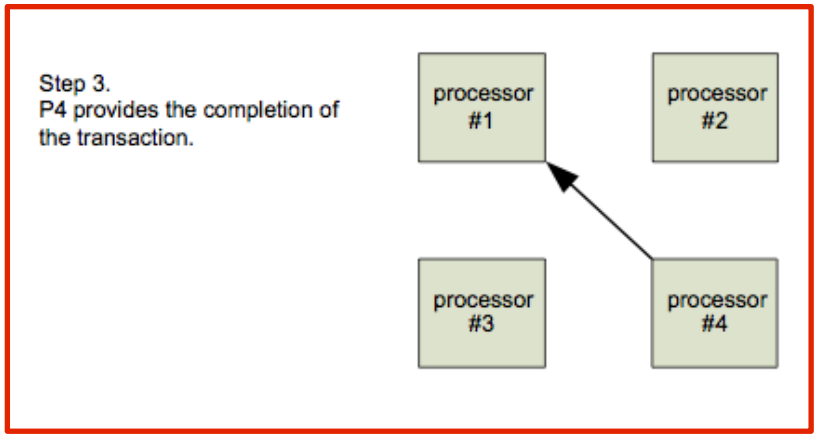
Figure credits: Introduction to Intel QuickPath Interconnect in Weaving High Performance Multiprocessor Fabric, Robert A. Maddox, Gurbir Singh, and Robert J. Safranek, Intel Press

Intel 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)





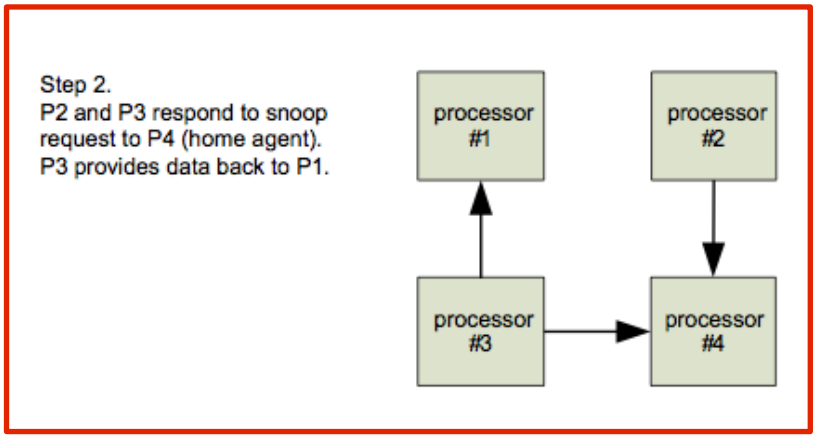


Figure credits: Introduction to Intel QuickPath Interconnect in Weaving High Performance Multiprocessor Fabric, Robert A. Maddox, Gurbir Singh, and Robert J. Safranek, Intel Press

AMD's HT Assist

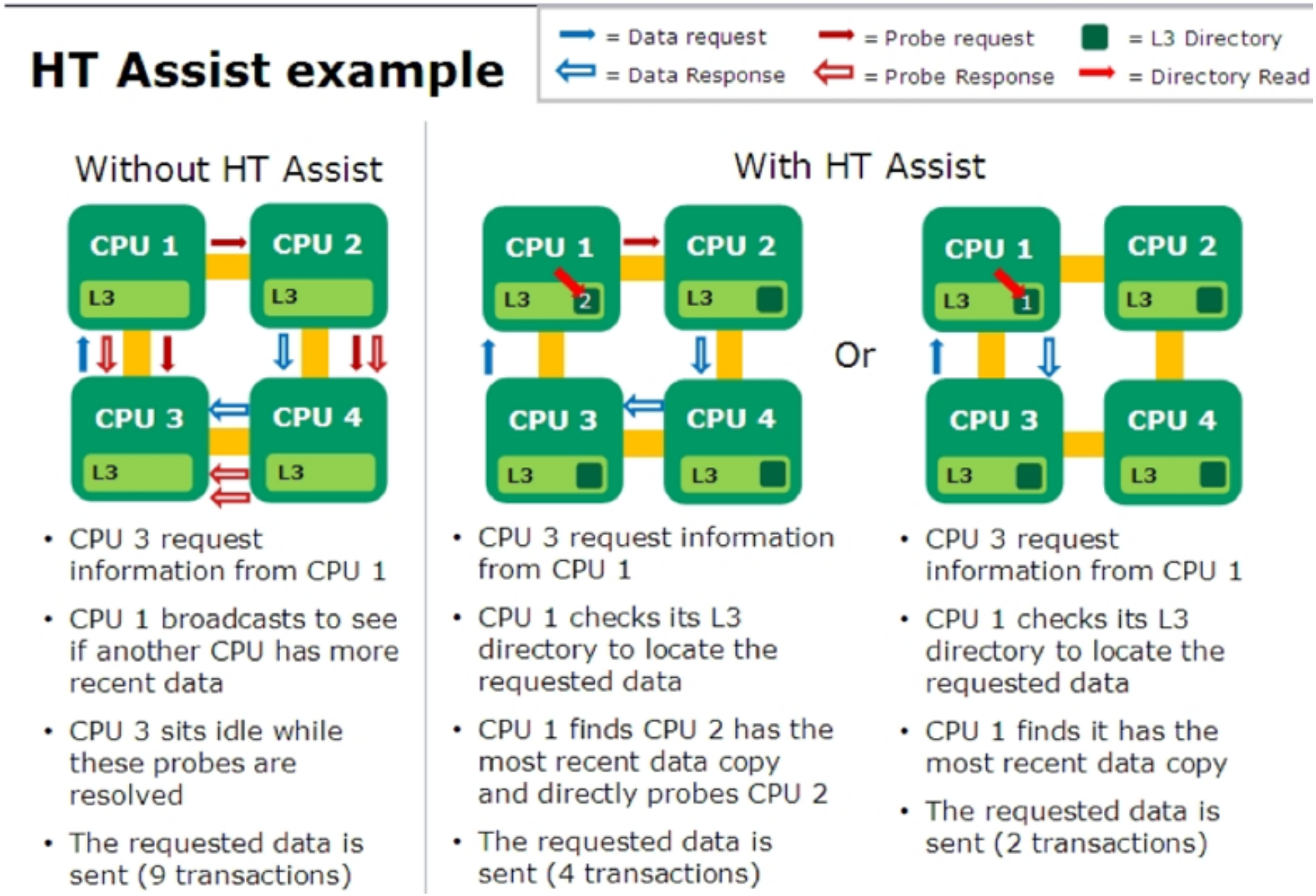


Figure source: http://blogs.zdnet.com/perlow/?p=10187

Memory Models and Weak Ordering

What is a Memory Model?

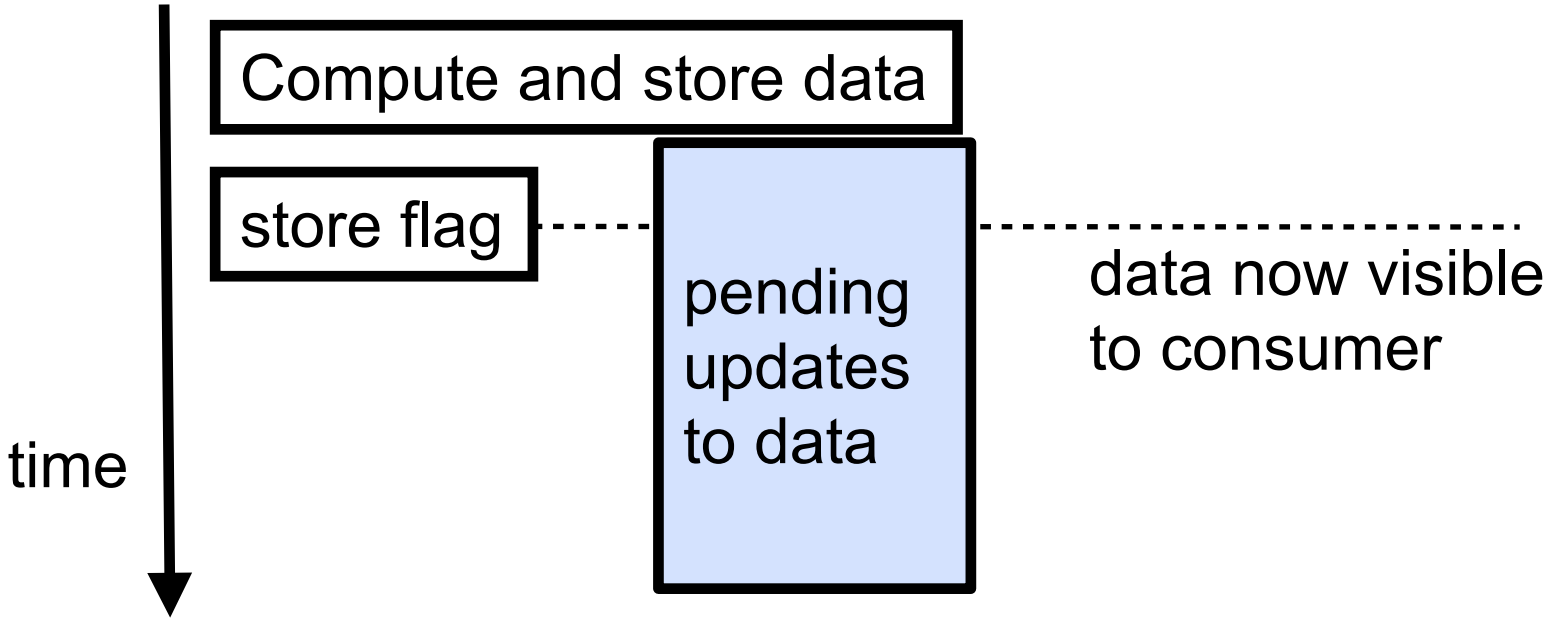
- A contract between a program and any hardware and software that reorders operations in a program execution
- In the context of parallelism, a memory model governs interactions between threads and shared memory
  - — atomicity, ordering, visibility
- Weak memory models: any load/store operation can 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: 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

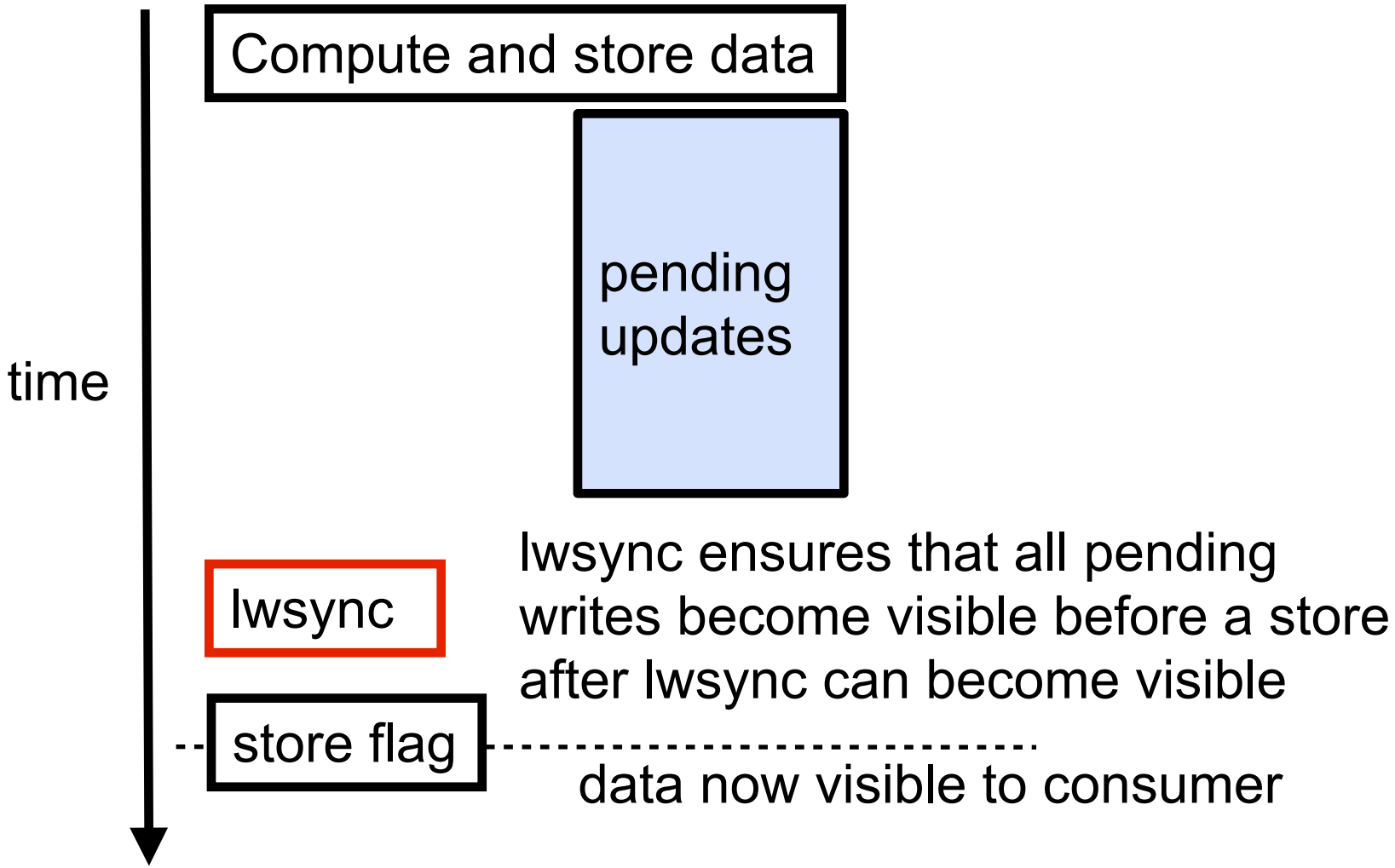
IBM Power Weak Memory Model: Producer

Incorrect way: without attention to weak ordering



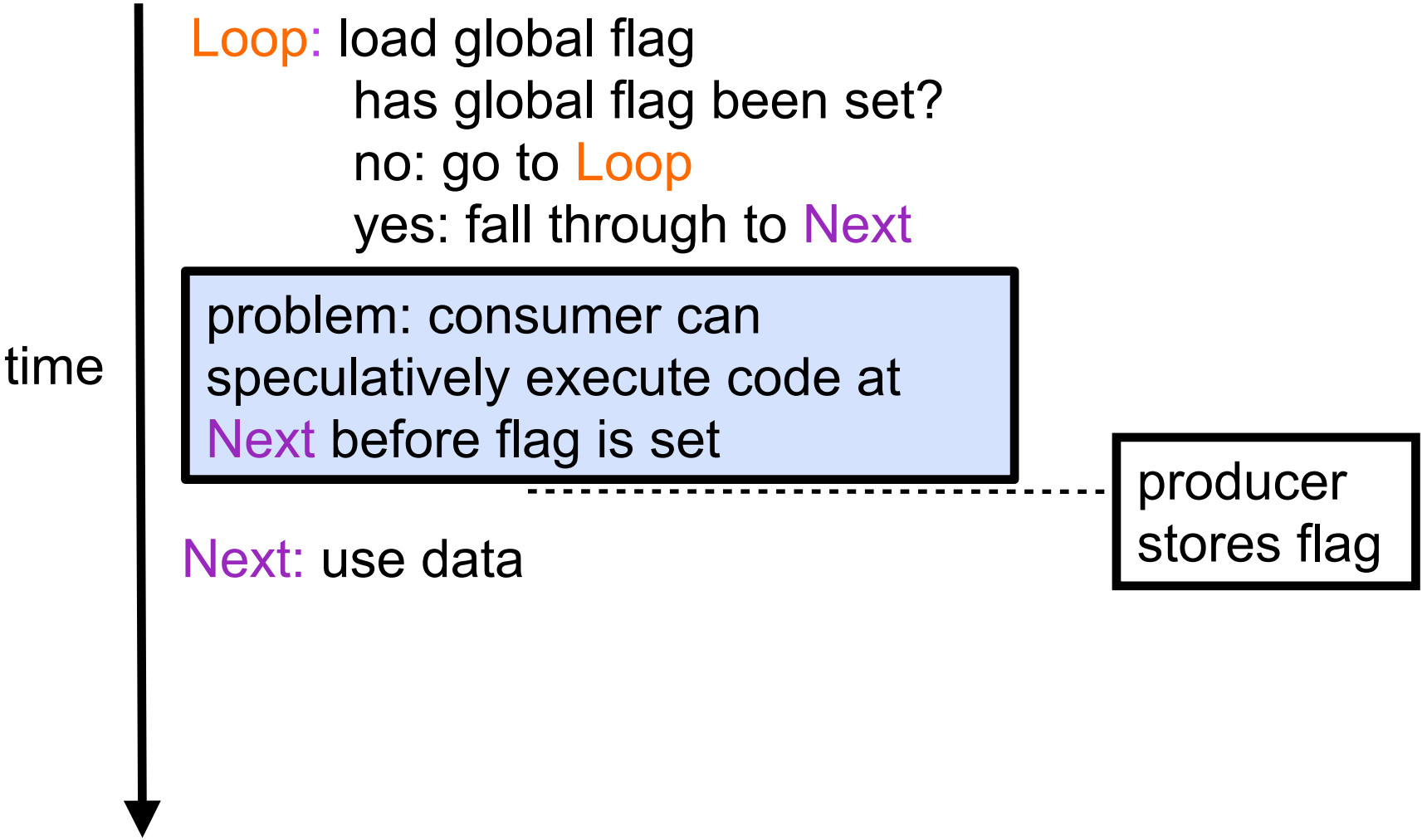
IBM Power Weak Memory Model: Producer

Correct way: ensure writes complete before setting flag



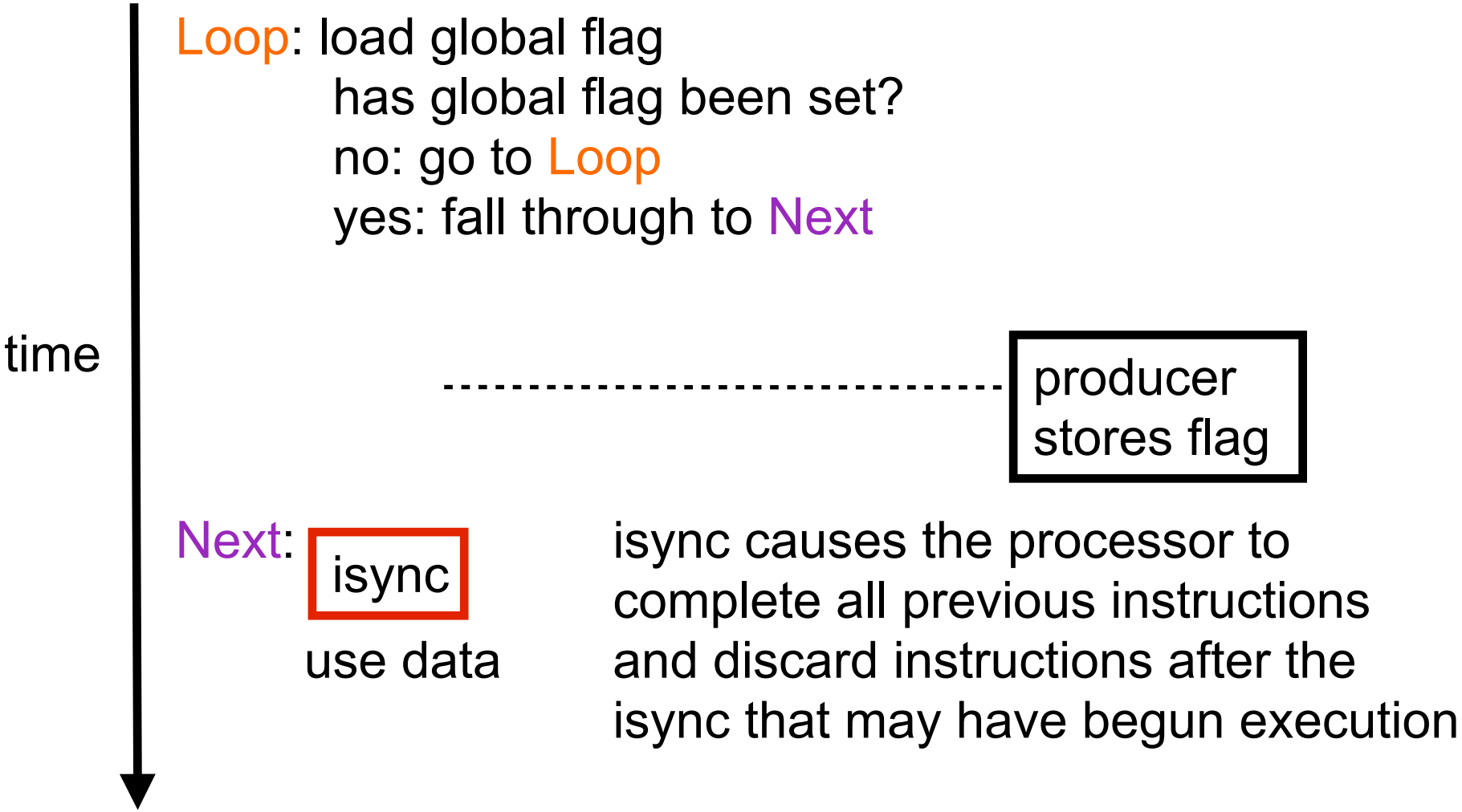
IBM Power Weak Memory Model: Consumer

Incorrect way: without attention to weak ordering



IBM Power Weak Memory Model: Consumer

Correct way: inhibit speculative reads until flag is set



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
    - task-group 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 dedicated environment if # threads <= # cores
  - —use blocking in shared 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:
    - writer needs intermediate value: use fetch_and_add
    - else 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

```
shared count : integer := P
shared sense : Boolean := true

processor private local_sense : Boolean := true

procedure central_barrier
   // each processor toggles its own sense
   local_sense := not local_sense
   if fetch_and_decrement (&count) = 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 an advanced link-list-based lock (local spinning)

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

- 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