ECE 757 Review: Parallel Processors

Lecture notes based in part on slides created by John Shen, Mark Hill, David Wood, Guri Sohi, Jim Smith, Erika Gunadi, Mitch Hayenga, Vignyan Reddy, Dibakar Gope

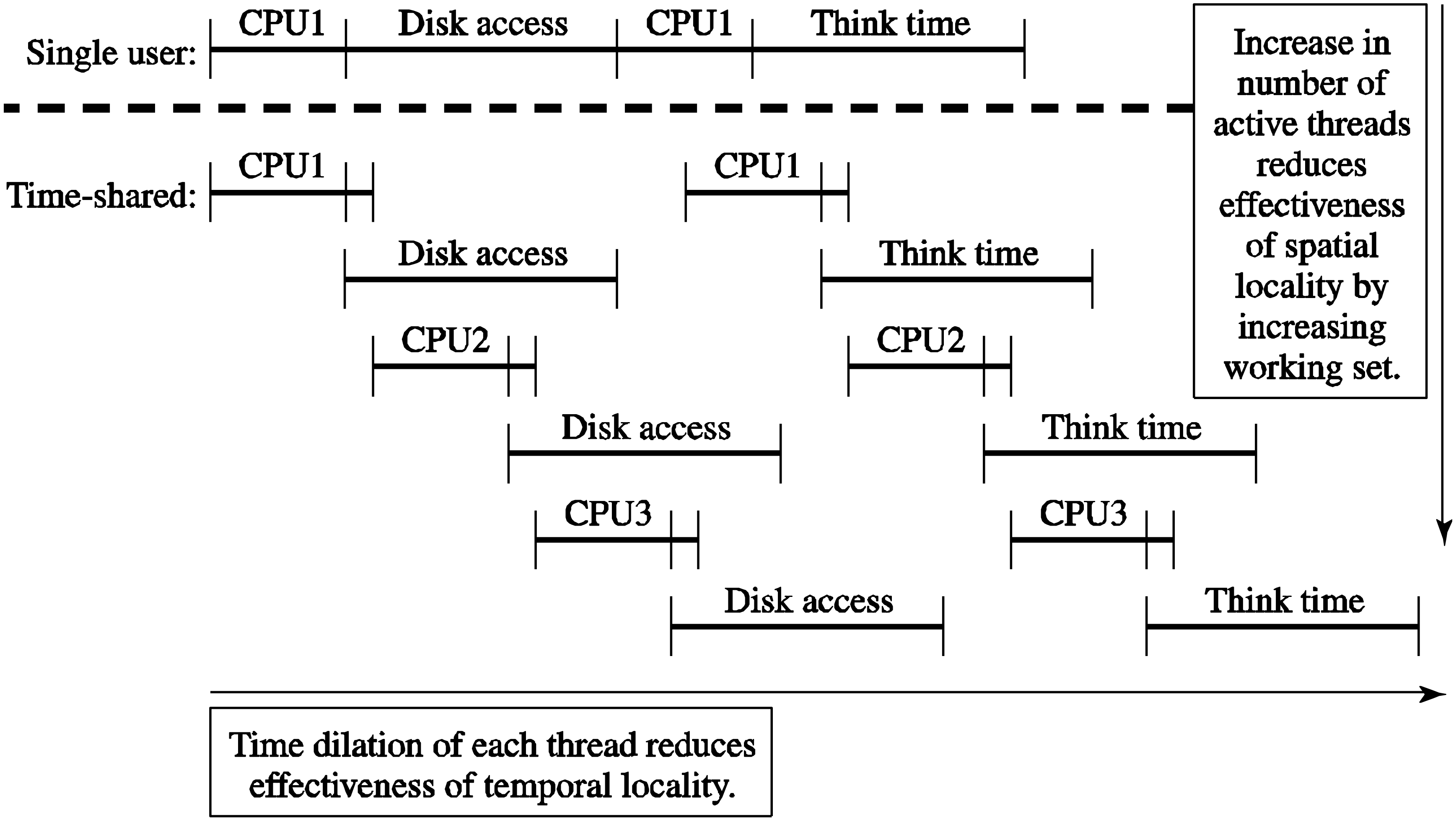
Parallel Processors

- Thread-level parallelism
- Synchronization
- Coherence
- Consistency
- Multithreading
- Multicore interconnects

Thread-level Parallelism

- Instruction-level parallelism
  - Reaps performance by finding independent work in a single thread
- Thread-level parallelism
  - Reaps performance by finding independent work across multiple threads
- Historically, requires explicitly parallel workloads
  - Originate from mainframe time-sharing workloads
  - Even then, CPU speed >> I/O speed
  - Had to overlap I/O latency with “something else” for the CPU to do
  - Hence, operating system would schedule other tasks/processes/threads that were “time-sharing” the CPU

Thread-level Parallelism

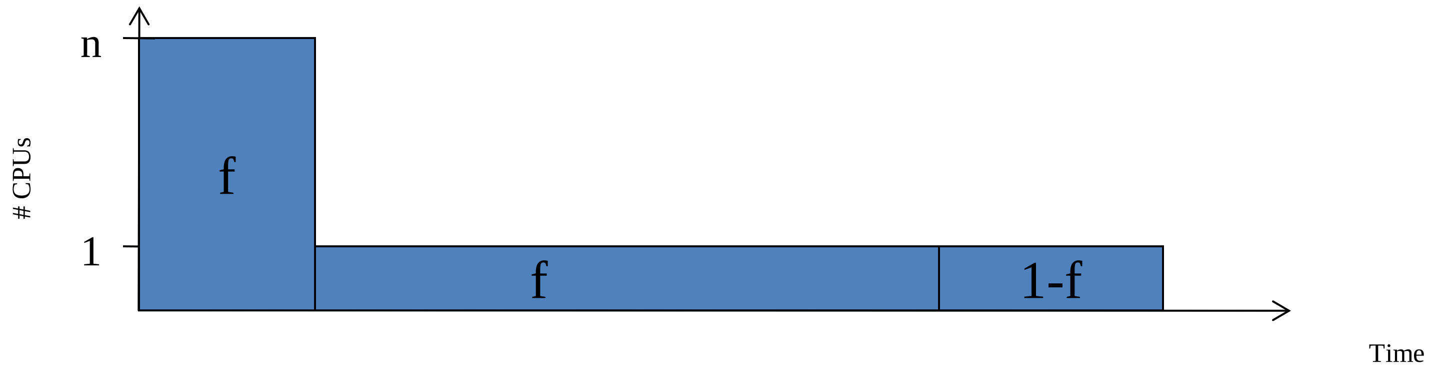

Single user:
CPU1
Disk access
CPU1
Think time
Time-shared:
CPU1
CPU1
Disk access
Think time
CPU2
CPU2
Disk access
Think time
CPU3
CPU3
Disk access
Think time
Increase in number of active threads reduces effectiveness of spatial locality by increasing working set.
Time dilation of each thread reduces effectiveness of temporal locality.

- Reduces effectiveness of temporal and spatial locality

Thread-level Parallelism

- Initially motivated by time-sharing of single CPU
  - OS, applications written to be multithreaded
- Quickly led to adoption of multiple CPUs in a single system
  - Enabled scalable product line from entry-level single-CPU systems to high-end multiple-CPU systems
  - Same applications, OS, run seamlessly
  - Adding CPUs increases throughput (performance)
- More recently:
  - Multiple threads per processor core
    - Coarse-grained multithreading (aka “switch-on-event”)
    - Fine-grained multithreading
    - Simultaneous multithreading
  - Multiple processor cores per die
    - Chip multiprocessors (CMP)
    - Chip multithreading (CMT)

Amdahl's Law



f – fraction that can run in parallel

1-f – fraction that must run serially

$$Speedup = \frac{1}{(1-f)+\frac{f}{n}}$$

$$\lim_{n\to\infty}\frac{1}{1-f+\frac{f}{n}}=\frac{1}{1-f}$$

Thread-level Parallelism

- Parallelism limited by sharing
  - Amdahl's law:
    - Access to shared state must be serialized
    - Serial portion limits parallel speedup
  - Many important applications share (lots of) state
    - Relational databases (transaction processing): GBs of shared state
  - Even completely independent processes "share" virtualized hardware through O/S, hence must synchronize access
- Access to shared state/shared variables
  - Must occur in a predictable, repeatable manner
  - Otherwise, chaos results
- Architecture must provide primitives for serializing access to shared state

Synchronization

Thread 0	Thread 1
	load r1, A
	addi r1, r1, 3
load r1, A	
addi r1, r1, 1	
store r1, A	
	store r1, A

(a)

Thread 0	Thread 1
load r1, A	
addi r1, r1, 1	
store r1, A	
	load r1, A
	addi r1, rl, 3
	store r1, A

(b)

Thread 0	Thread 1
	load r1, A
	addi r1, r1, 3
	store r1, A
load r1, A	
addi r1, r1, 1	
store r1, A	

(c)

Thread 0	Thread 1
load r1, A	
addi r1, r1, 1	
	load r1, A
	addi r1, rl, 3
	store r1, A
store r1, A	

(d)

Some Synchronization Primitives

Primitive	Semantic	Comments
Fetch-and-add	Atomic load/add/store operation	Permits atomic increment, can be used to synthesize locks for mutual exclusion
Compare-and-swap	Atomic load/compare/conditional store	Stores only if load returns an expected value
Load-linked/store-conditional	Atomic load/conditional store	Stores only if load/store pair is atomic; that is, there is no intervening store

- Only one is necessary
  – Others can be synthesized

Synchronization Examples

```
Thread 0        Thread 1         Thread 0          Thread 1           Thread 0          Thread 1
--------        --------         --------          --------           --------          --------
fetchadd A, 1   fetchadd A, 3    spin:             spin:              spin:             spin:
                                   cmpswp AL, 1      cmpswp AL, 1       ll r1, A          ll r1, A
                                   bfail spin        bfail spin         addi r1, r1, 1    addi r1, r1, 3
                                   load r1, A        load r1, A         stc r1, A         stc r1, A
                                   addi r1, rl, 1    addi r1, rl, 3     bfail spin        bfail spin
                                   store r1, A       store r1, A
                                   store 0, AL       store 0, AL

             (a)                             (b)                                  (c)
```

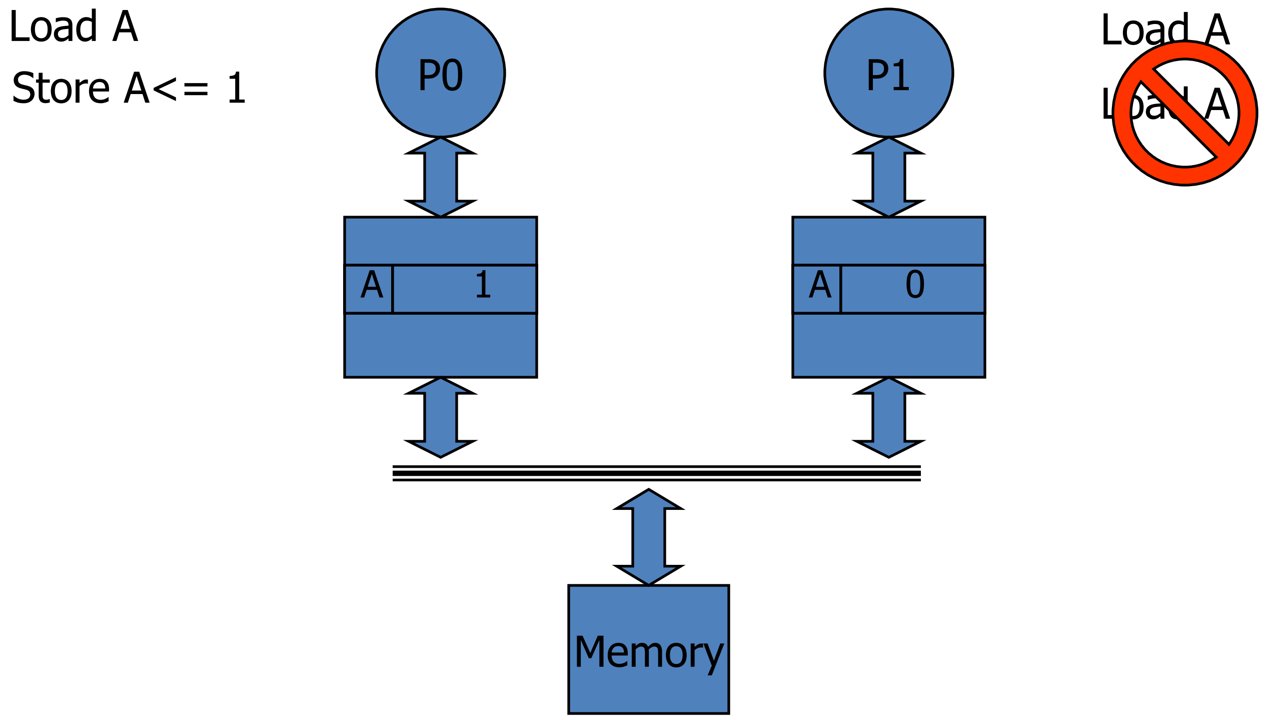
- All three guarantee same semantic:
  - Initial value of A: 0
  - Final value of A: 4
- b uses additional lock variable AL to protect *critical section* with a *spin lock*
  - This is the most common synchronization method in modern multithreaded applications

Multicore Designs

- Belong to: shared-memory symmetric multiprocessors
  - Many other types of parallel processor systems have been proposed and built
  - Key attributes are:
    - Shared memory: all physical memory is accessible to all CPUs
    - Symmetric processors: all CPUs are alike
  - Other parallel processors may:
    - Share some memory, share disks, share nothing
    - May have asymmetric processing units or noncoherent caches

- Shared memory in the presence of caches
  - Need caches to reduce latency per reference
  - Need caches to increase available bandwidth per core
  - **But, using caches induces the *cache coherence* problem**
  - **Furthermore, how do we *interleave* references from cores?**

Cache Coherence Problem



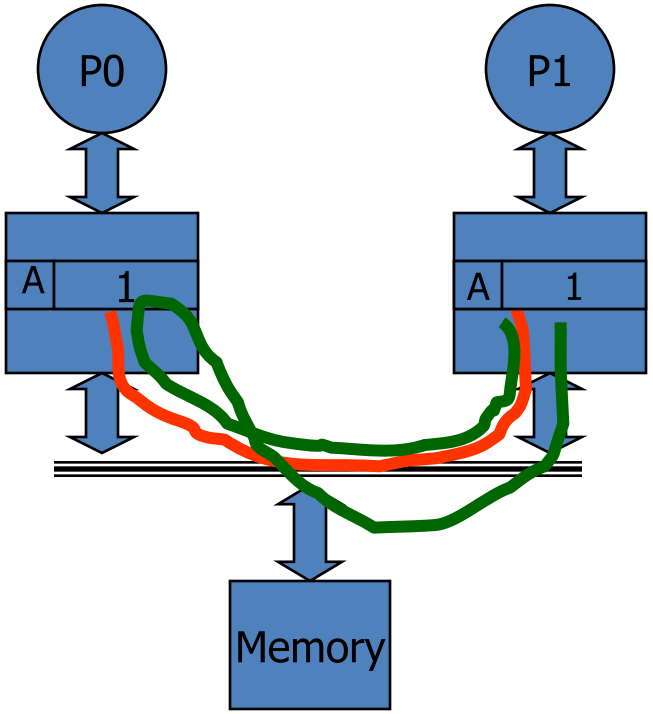
Cache Coherence Problem

Load A

Store A<= 1

Load A

Load A



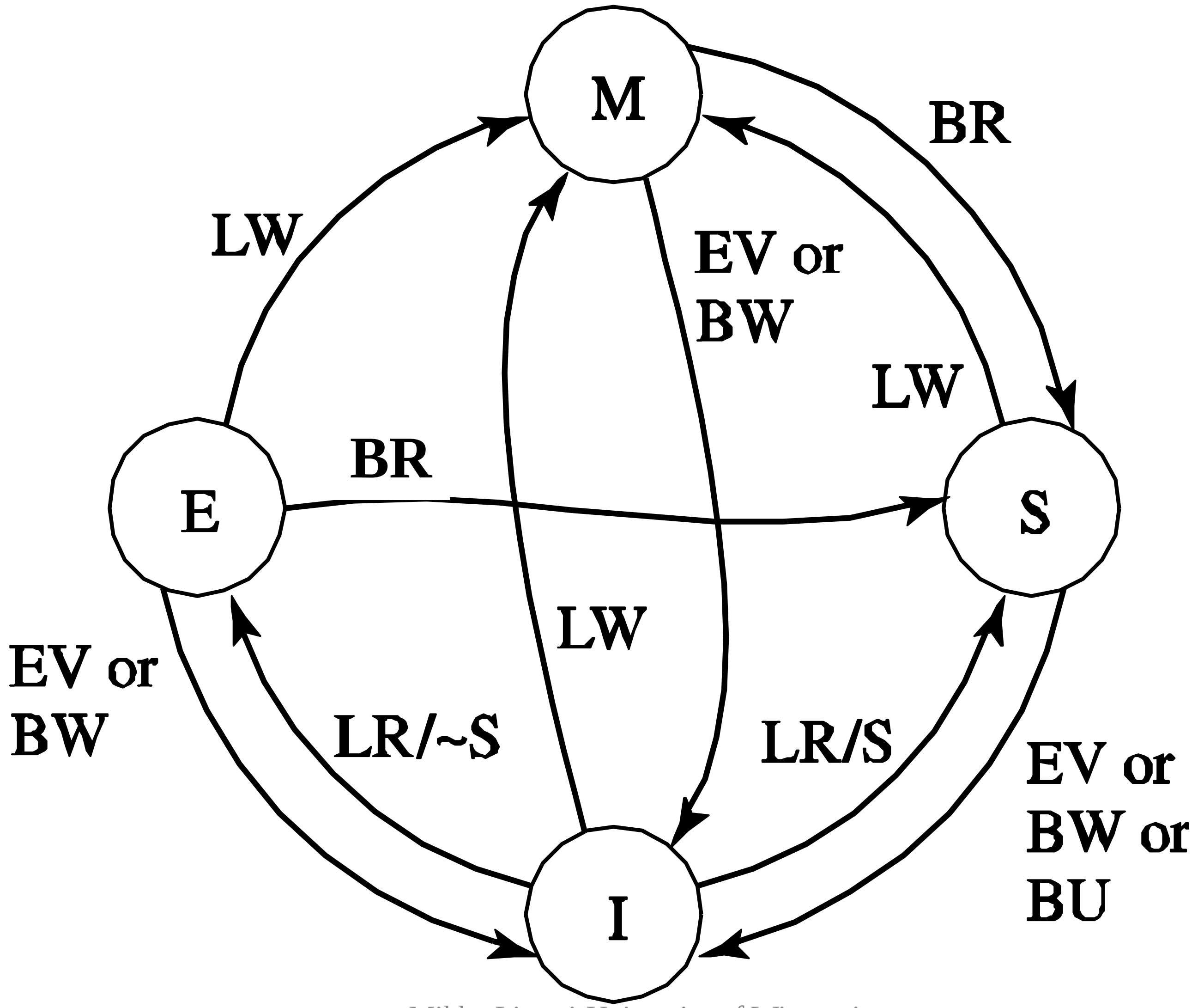
Invalidate Protocol

- Basic idea: maintain **single writer** property
  - Only one processor has write permission at any point in time
- Write handling
  - On write, invalidate all other copies of data
  - Make data private to the writer
  - Allow writes to occur until data is requested
  - Supply modified data to requestor directly or through memory
- Minimal set of states per cache line:
  - Invalid (not present)
  - Modified (private to this cache)
- State transitions:
  - Local read or write: I->M, fetch modified
  - Remote read or write: M->I, transmit data (directly or through memory)
  - Writeback: M->I, write data to memory

Invalidate Protocol Optimizations

- Observation: data can be *read-shared*
  - Add S (shared) state to protocol: MSI
- State transitions:
  - Local read: I->S, fetch shared
  - Local write: I->M, fetch modified; S->M, invalidate other copies
  - Remote read: M->S, supply data
  - Remote write: M->I, supply data; S->I, invalidate local copy
- Observation: data can be write-private (e.g. stack frame)
  - Avoid invalidate messages in that case
  - Add E (exclusive) state to protocol: MESI
- State transitions:
  - Local read: I->E if only copy, I->S if other copies exist
  - Local write: E->M silently, S->M, invalidate other copies

Sample Invalidate Protocol (MESI)

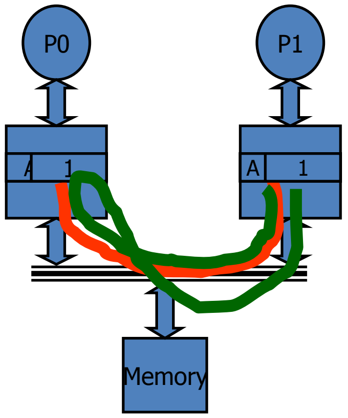


Sample Invalidate Protocol (MESI)

Current State s	Event and Local Coherence Controller Responses and Actions (s' refers to next state)					
	Local Read (LR)	**Local Write (LW)**	**Local Eviction (EV)**	**Bus Read (BR)**	**Bus Write (BW)**	**Bus Upgrade (BU)**
Invalid (I)	Issue bus read if no sharers then s' = E else s' = S	Issue bus write s' = M	s' = I	Do nothing	Do nothing	Do nothing
Shared (S)	Do nothing	Issue bus upgrade s' = M	s' = I	Respond shared	s' = I	s' = I
Exclusive (E)	Do nothing	s' = M	s' = I	Respond shared s' = S	s' = I	Error
Modified (M)	Do nothing	Do nothing	Write data back; s' = I	Respond dirty; Write data back; s' = S	Respond dirty; Write data back; s' = I	Error

Snoopy Cache Coherence

- Origins in shared-memory-bus systems
- All CPUs could observe all other CPUs requests on the bus; hence "snooping"
  - Bus Read, Bus Write, Bus Upgrade
- React appropriately to snooped commands
  - Invalidate shared copies
  - Provide up-to-date copies of dirty lines
    - Flush (writeback) to memory, or
    - Direct intervention (*modified intervention* or *dirty miss*)



Directory Cache Coherence

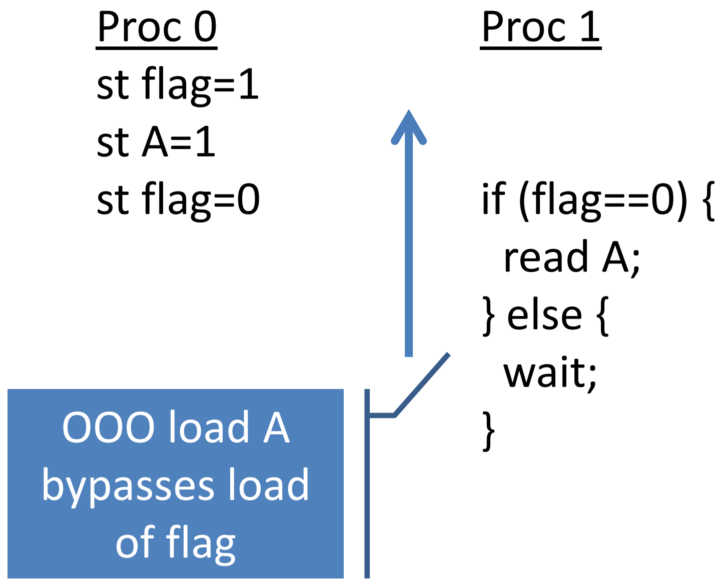
- Directory implementation
  - Extra bits stored in memory (directory) record MSI state of line
  - Memory controller maintains coherence based on the current state
  - Other CPUs' commands are not snooped, instead:
    - Directory forwards relevant commands
  - Ideal filtering: only observe commands that you need to observe
  - Meanwhile, bandwidth at directory scales by adding memory controllers as you increase size of the system

    Leads to very scalable designs (100s to 1000s of CPUs)

Another Problem: Memory Ordering

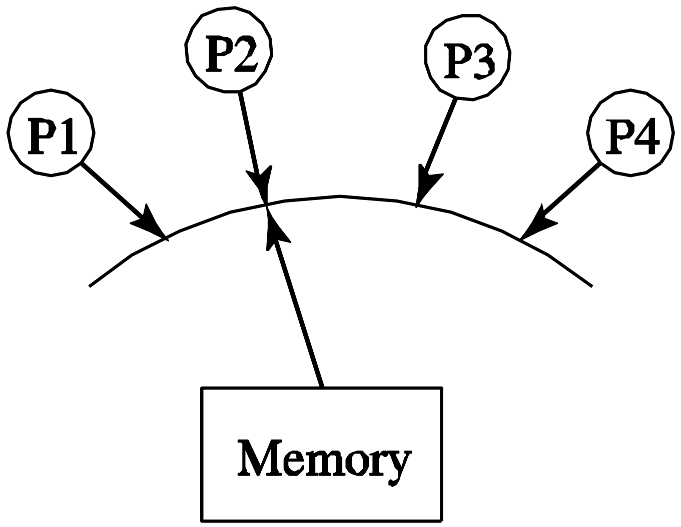
- Producer-consumer pattern:
  - Update control block, then set flag to tell others you are done with your update
  - Proc1 reorders load of A ahead of load of flag, reads stale copy of A but still sees that flag is clear
- Unexpected outcome
  - Does not match programmer’s expectations
  - Just one example of many subtle cases
- ISA specifies rules for what is allowed:

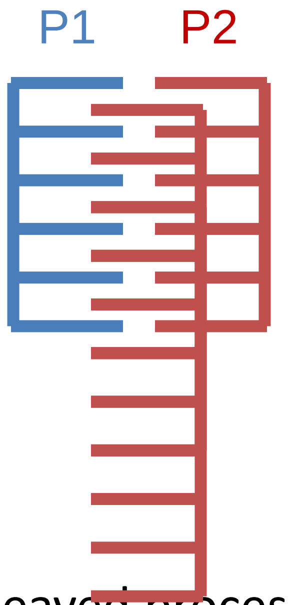
  memory consistency model



```
Proc 0            Proc 1
st flag=1
st A=1
st flag=0         if (flag==0) {
                    read A;
                  } else {
                    wait;
                  }
```

Sequential Consistency [Lamport 1979]





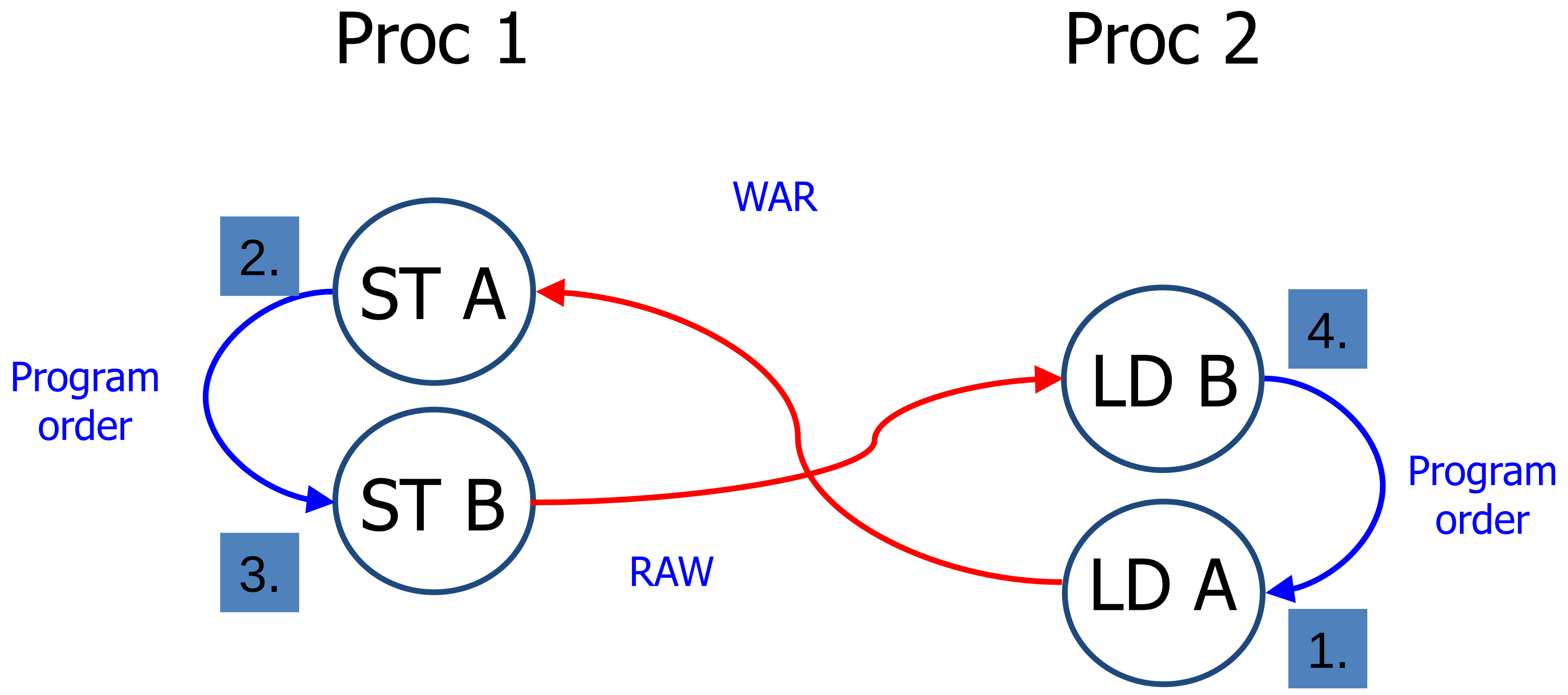
- Processors treated as if they are interleaved processes on a single time-shared CPU

- All references must fit into a total global order or interleaving that does not violate any CPUs program order
  - Otherwise sequential consistency not maintained

Constraint graph

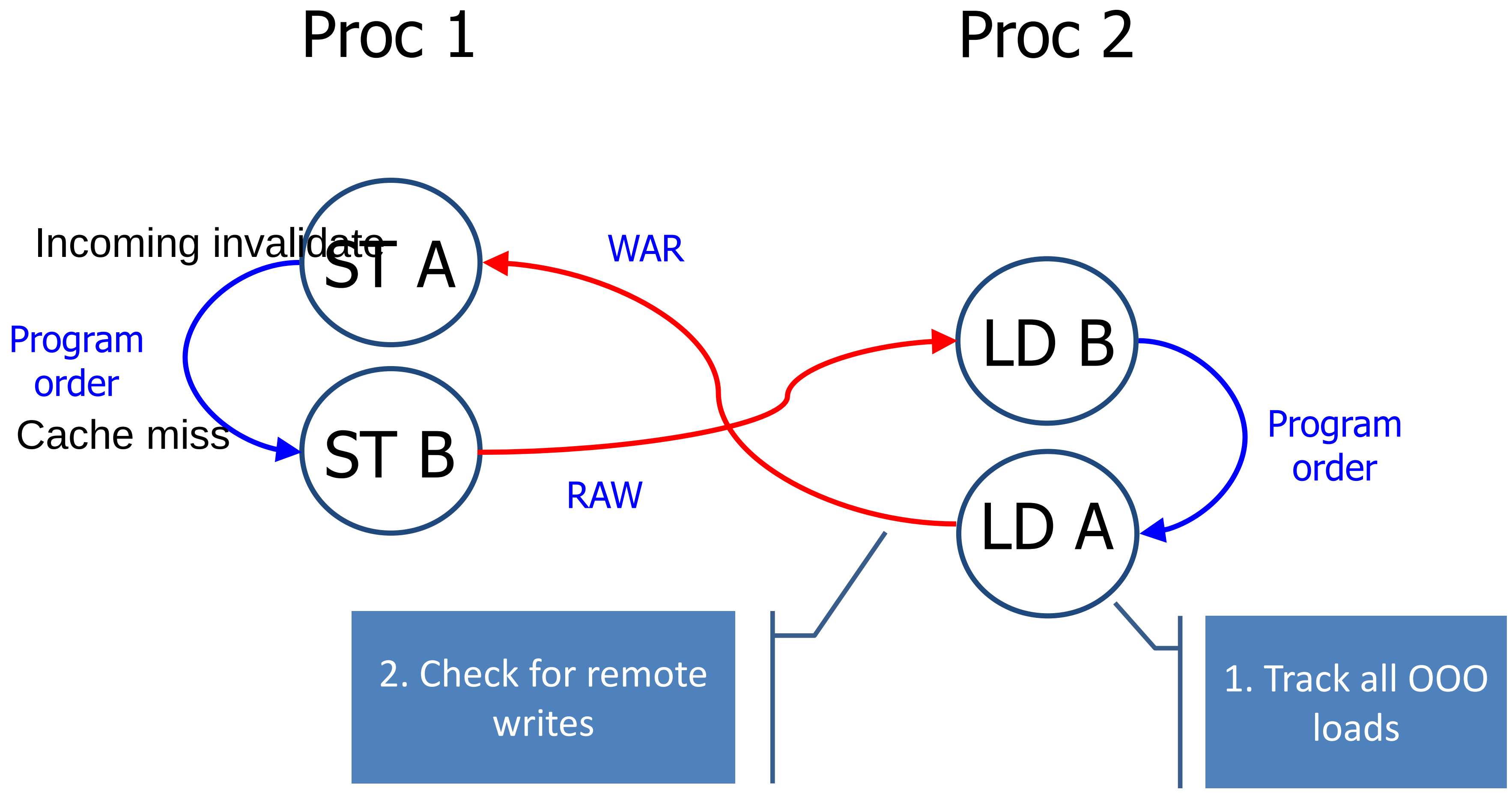
- Reasoning about memory consistency [Landin, ISCA-18]
- Directed graph represents a multithreaded execution
  - Nodes represent dynamic instruction instances
  - Edges represent their transitive orders (program order, RAW, WAW, WAR).
- If the constraint graph is acyclic, then the execution is correct
  - Cycle implies A must occur before B and B must occur before A => **contradiction**

Constraint graph example - SC



Cycle indicates that execution is incorrect

Anatomy of a cycle

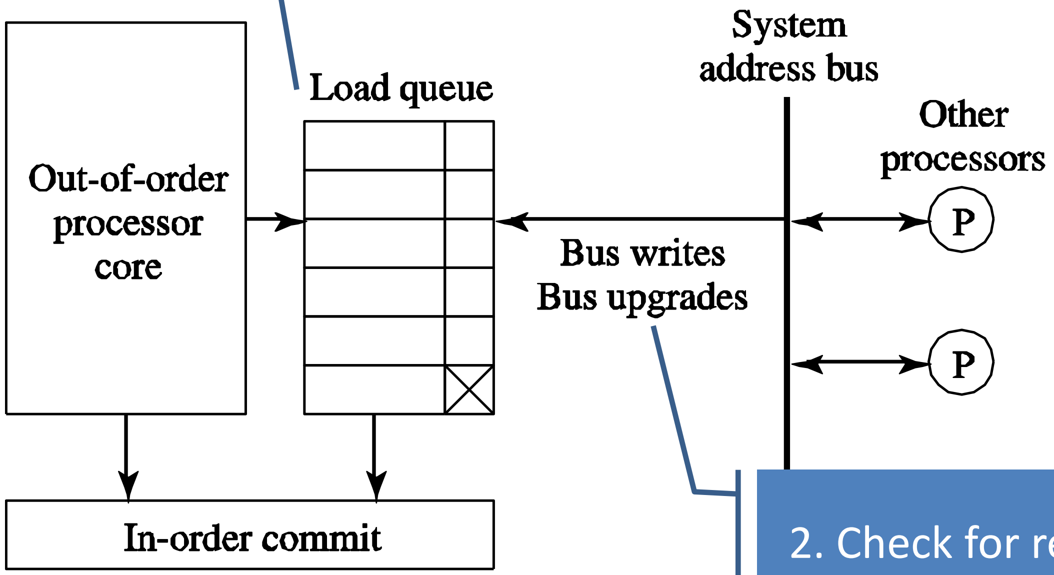


2. Check for remote writes

1. Track all OOO loads

High-Performance Sequential Consistency

1. Track all OOO loads



2. Check for remote writes

- Load queue records all speculative loads
- Bus writes/upgrades are checked against LQ
- Any matching load gets marked for replay
- At commit, loads are checked and replayed if necessary
  - Results in machine flush, since load-dependent ops must also replay
- Practically, conflicts are rare, so expensive flush is OK

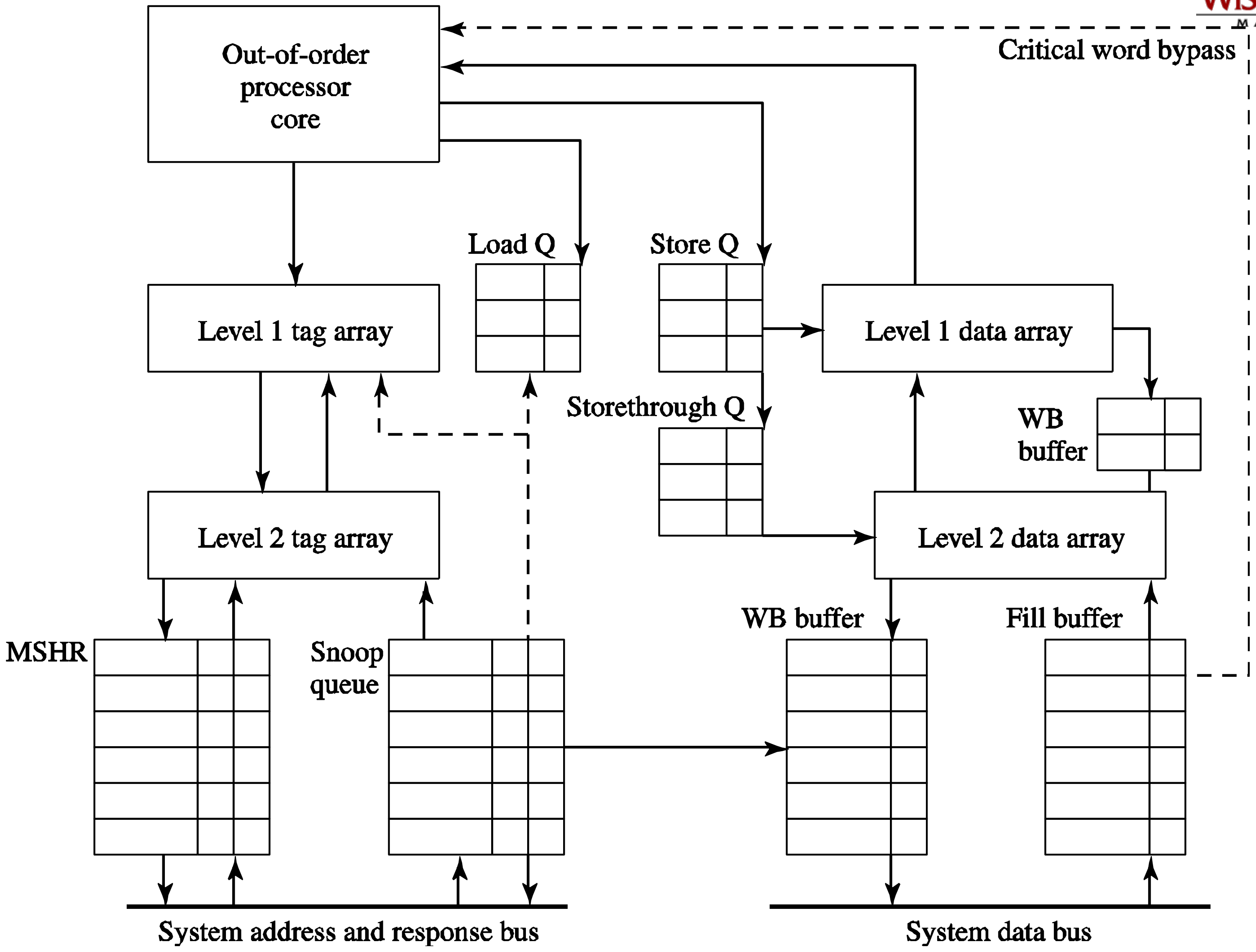
Recapping

- Multicore processors need shared memory
- Must use caches to provide latency/bandwidth
- Cache memories must:
  - Provide coherent view of memory

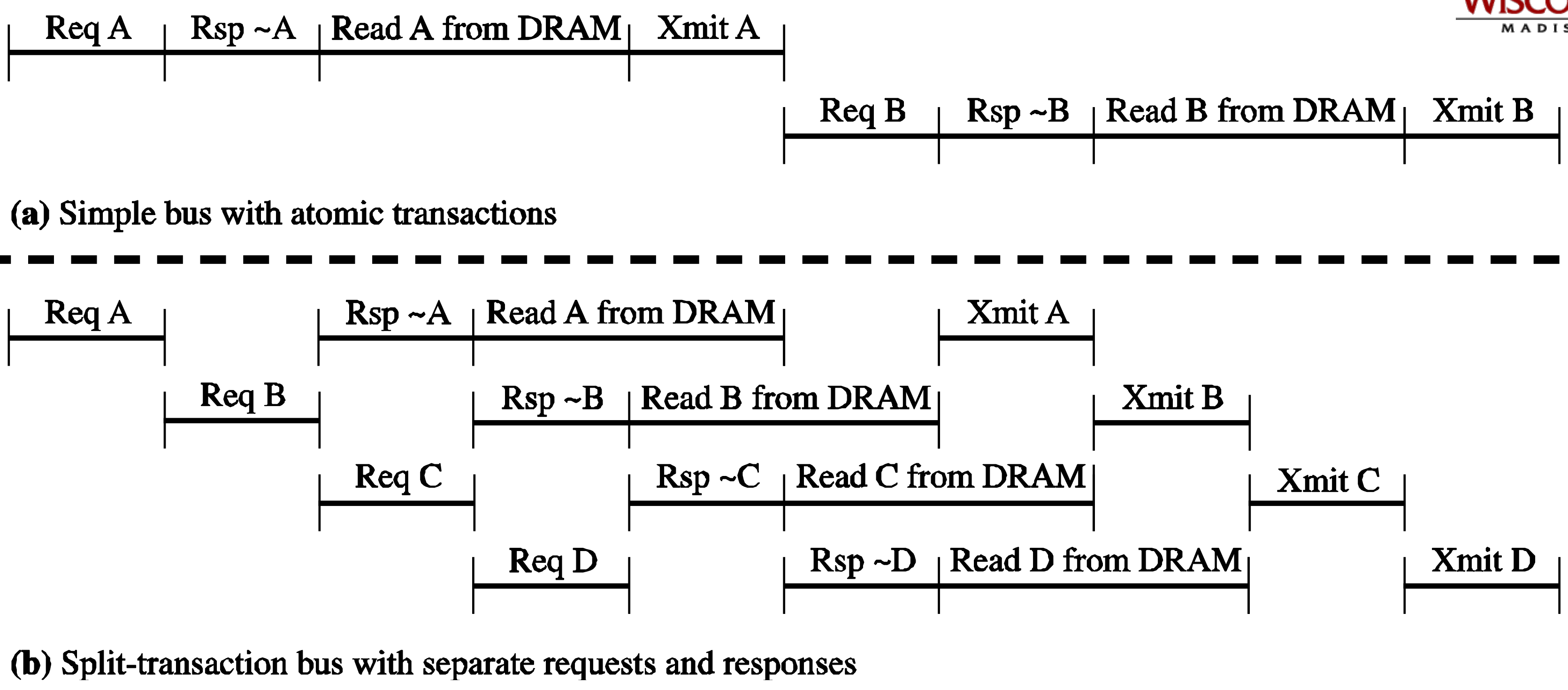
  $\rightarrow$**must solve cache coherence problem**
- Cores and caches must:
  - Properly order interleaved memory references

  $\rightarrow$ **must implement memory consistency correctly**

Coherent Memory Interface



Split Transaction Bus



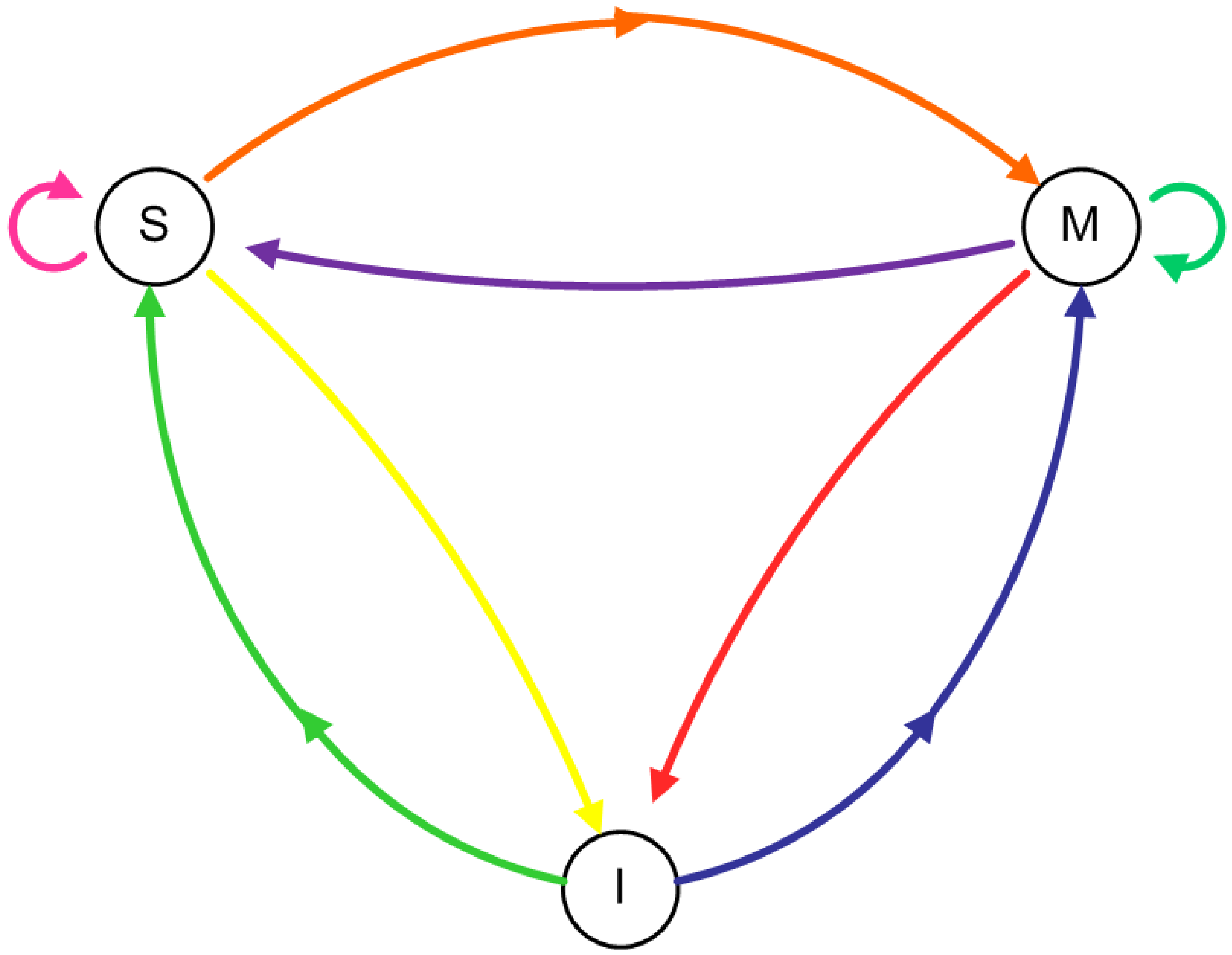
(a) Simple bus with atomic transactions

(b) Split-transaction bus with separate requests and responses

- “Packet switched” vs. “circuit switched”
- Release bus after request issued
- Allow multiple concurrent requests to overlap memory latency
- Complicates control, arbitration, and coherence protocol
  – *Transient* states for pending blocks (e.g. “req. issued but not completed”)

Example: MSI (SGI-Origin-like, directory, invalidate)

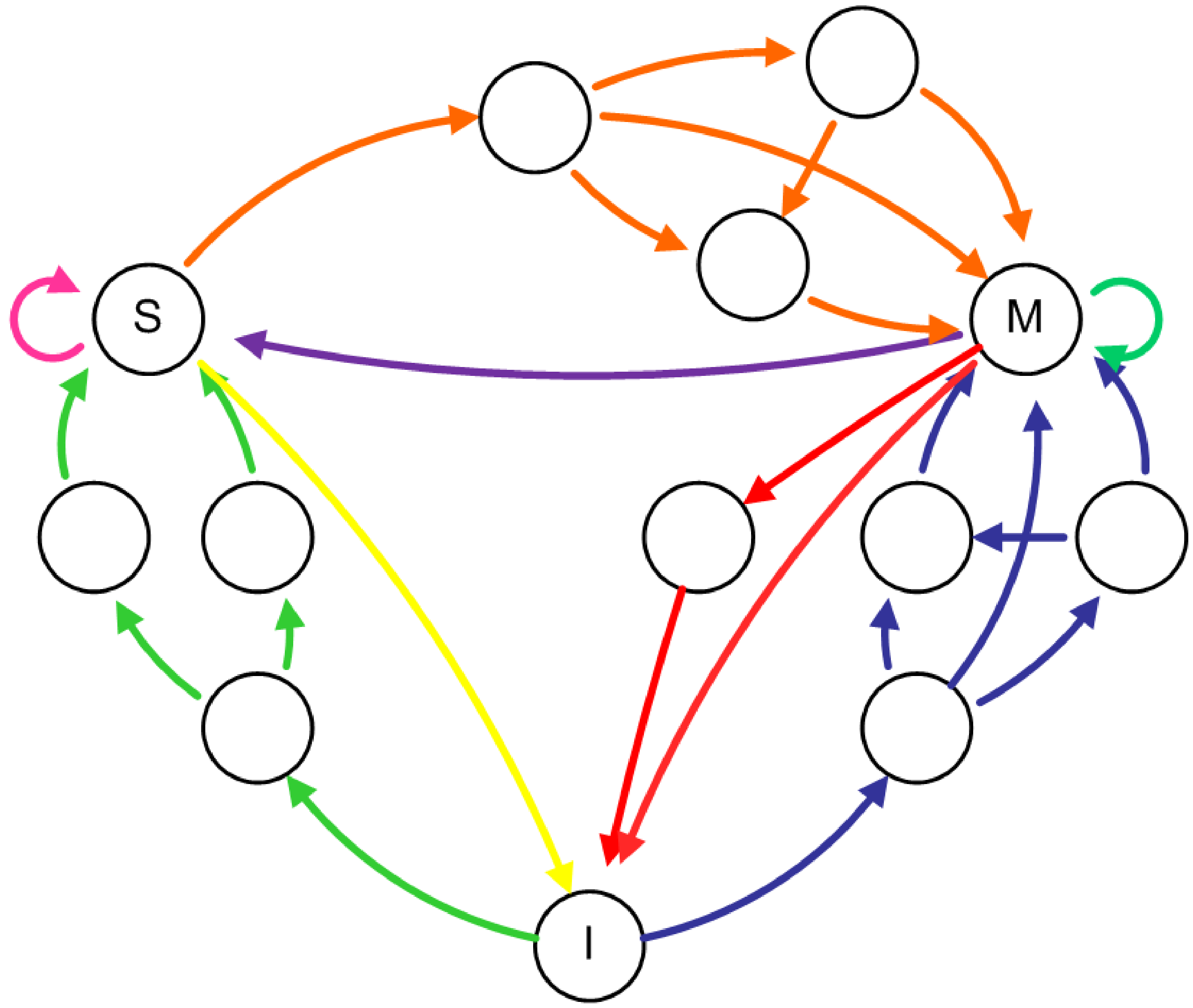
High Level



Example: MSI (SGI-Origin-like, directory, invalidate)

High Level

Busy States

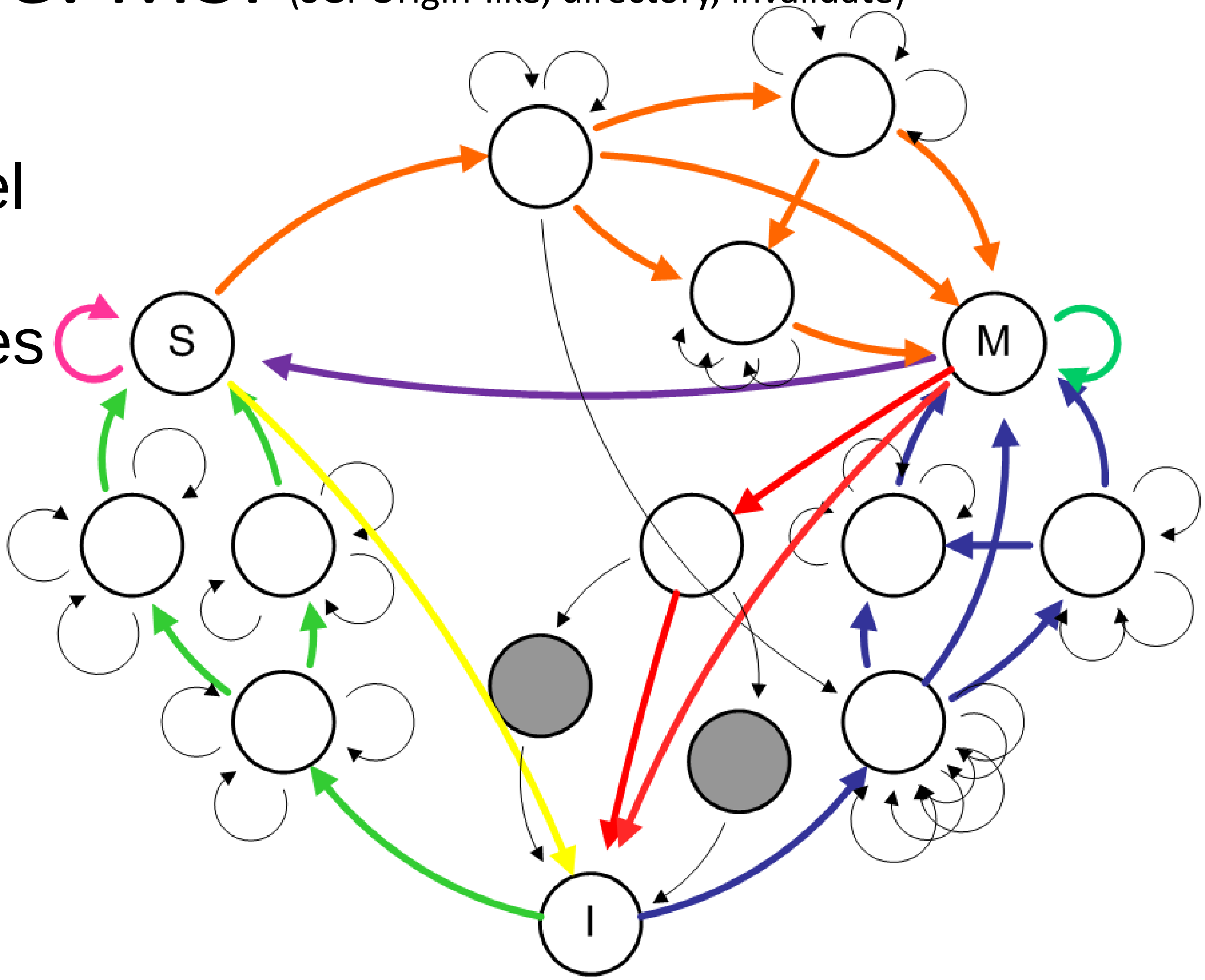


Example: MSI (SGI-Origin-like, directory, invalidate)

High Level

Busy States

Races



Multithreaded Cores

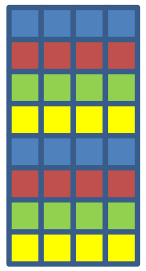
- Basic idea:
  - CPU resources are expensive and should not be idle
- 1960's: Virtual memory and multiprogramming
  - Virtual memory/multiprogramming invented to tolerate latency to secondary storage (disk/tape/etc.)
  - Processor-disk speed mismatch:
    - microseconds to tens of milliseconds (1:10000 or more)
  - OS context switch used to bring in other useful work while waiting for page fault or explicit read/write
  - Cost of context switch must be much less than I/O latency (easy)

Multithreaded Cores

- 1990's: Memory wall and multithreading
  - Processor-DRAM speed mismatch:
    - nanosecond to fractions of a microsecond (1:500)
  - H/W task switch used to bring in other useful work while waiting for cache miss
  - Cost of context switch must be much less than cache miss latency

- Very attractive for applications with abundant thread-level parallelism
  - Commercial multi-user workloads

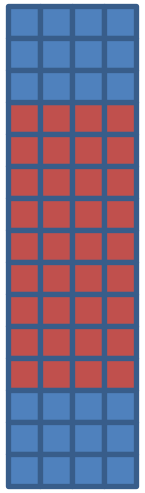
Approaches to Multithreading



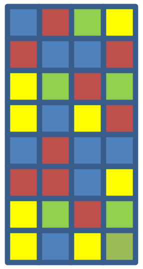
- Fine-grain multithreading
  - Switch contexts at fixed fine-grain interval (e.g. every cycle)
  - Need enough thread contexts to cover stalls
  - Example: Tera MTA, 128 contexts, no data caches
- Benefits:
  - Conceptually simple, high throughput, deterministic behavior
- Drawback:
  - Very poor single-thread performance

Approaches to Multithreading

- Coarse-grain multithreading
  - Switch contexts on long-latency events (e.g. cache misses)
  - Need a handful of contexts (2-4) for most benefit
- Example: IBM RS64-IV (Northstar), 2 contexts
- Benefits:
  - Simple, improved throughput (~30%), low cost
  - Thread priorities mostly avoid single-thread slowdown
- Drawback:
  - Nondeterministic, conflicts in shared caches



Approaches to Multithreading



- Simultaneous multithreading
  - Multiple concurrent active threads (no notion of thread switching)
  - Need a handful of contexts for most benefit (2-8)
- Example: Intel Pentium 4/Nehalem/Sandybridge, IBM Power 5/6/7, Alpha EV8/21464
- Benefits:
  - Natural fit for OOO superscalar
  - Improved throughput
  - Low incremental cost
- Drawbacks:
  - Additional complexity over OOO superscalar
  - Cache conflicts

Approaches to Multithreading

- Chip Multiprocessors (CMP)
- Very popular these days

Processor	Cores/ chip	Multi- threaded?	Resources shared
IBM Power 4	2	No	L2/L3, system interface
IBM Power 7	8	Yes (4T)	Core, L2/L3, DRAM, system interface
Sun Ultrasparc	2	No	System interface
Sun Niagara	8	Yes (4T)	Everything
Intel Pentium D	2	Yes (2T)	Core, nothing else
Intel Core i7	4	Yes	L3, DRAM, system interface
AMD Opteron	2, 4, 6, 12	No	System interface (socket), L3

Approaches to Multithreading

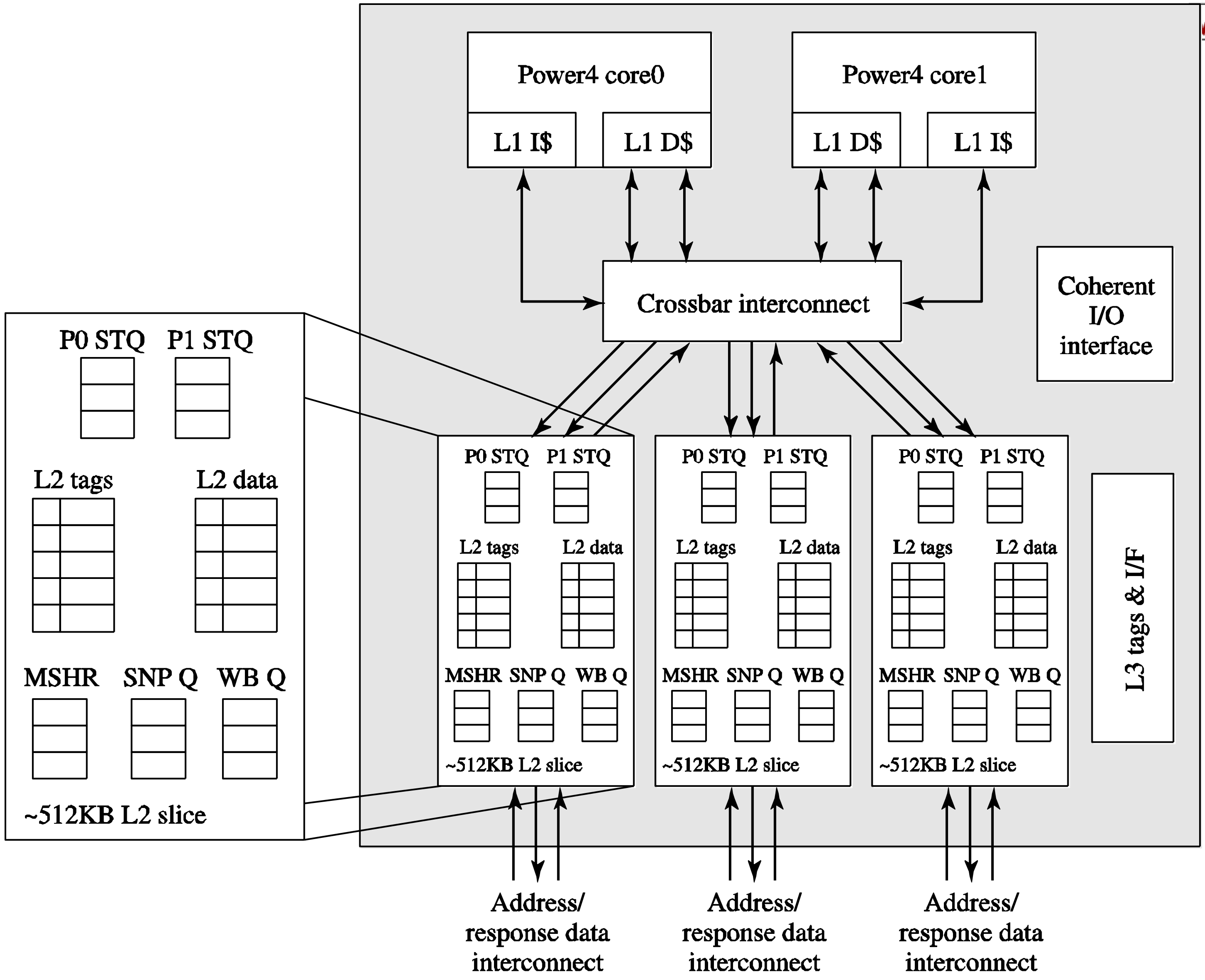
- Chip Multithreading (CMT)
  - Similar to CMP
- Share something in the core:
  - Expensive resource, e.g. floating-point unit (FPU)
  - Also share L2, system interconnect (memory and I/O bus)
- Examples:
  - Sun Niagara, 8 cores per die, one FPU
  - AMD Bulldozer: one FP cluster for every two INT clusters
- Benefits:
  - Same as CMP
  - Further: amortize cost of expensive resource over multiple cores
- Drawbacks:
  - Shared resource may become bottleneck
  - 2nd generation (Niagara 2) does **not** share FPU

Multithreaded/Multicore Processors

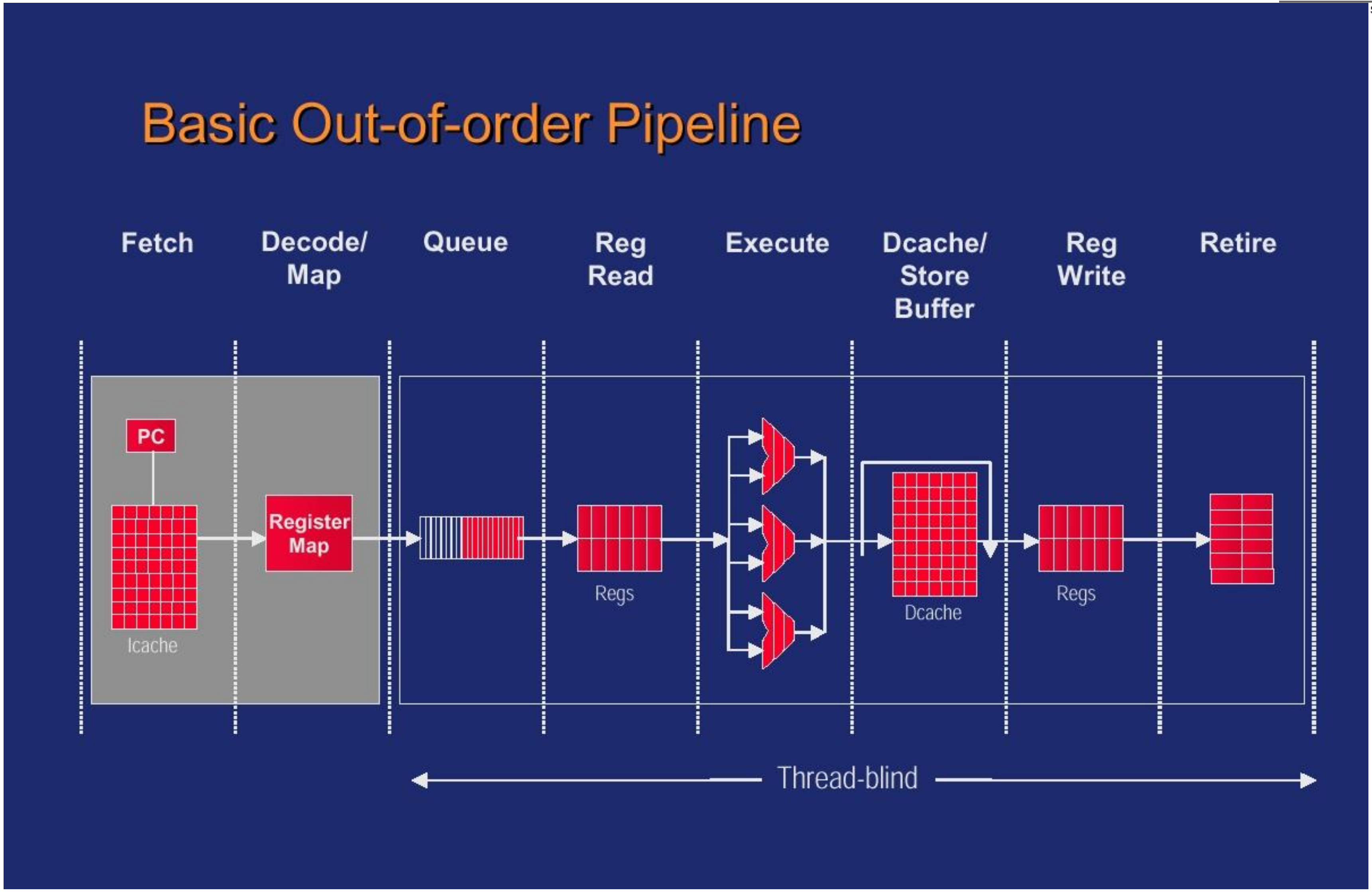
MT Approach	Resources shared between threads	Context Switch Mechanism
None	Everything	Explicit operating system context switch
Fine-grained	Everything but register file and control logic/state	Switch every cycle
Coarse-grained	Everything but I-fetch buffers, register file and con trol logic/state	Switch on pipeline stall
SMT	Everything but instruction fetch buffers, return address stack, architected register file, control logic/state, reorder buffer, store queue, etc.	All contexts concurrently active; no switching
CMT	Various core components (e.g. FPU), secondary cache, system interconnect	All contexts concurrently active; no switching
CMP	Secondary cache, system interconnect	All contexts concurrently active; no switching

- Many approaches for executing multiple threads on a single die
  - Mix-and-match: IBM Power7 CMP+SMT

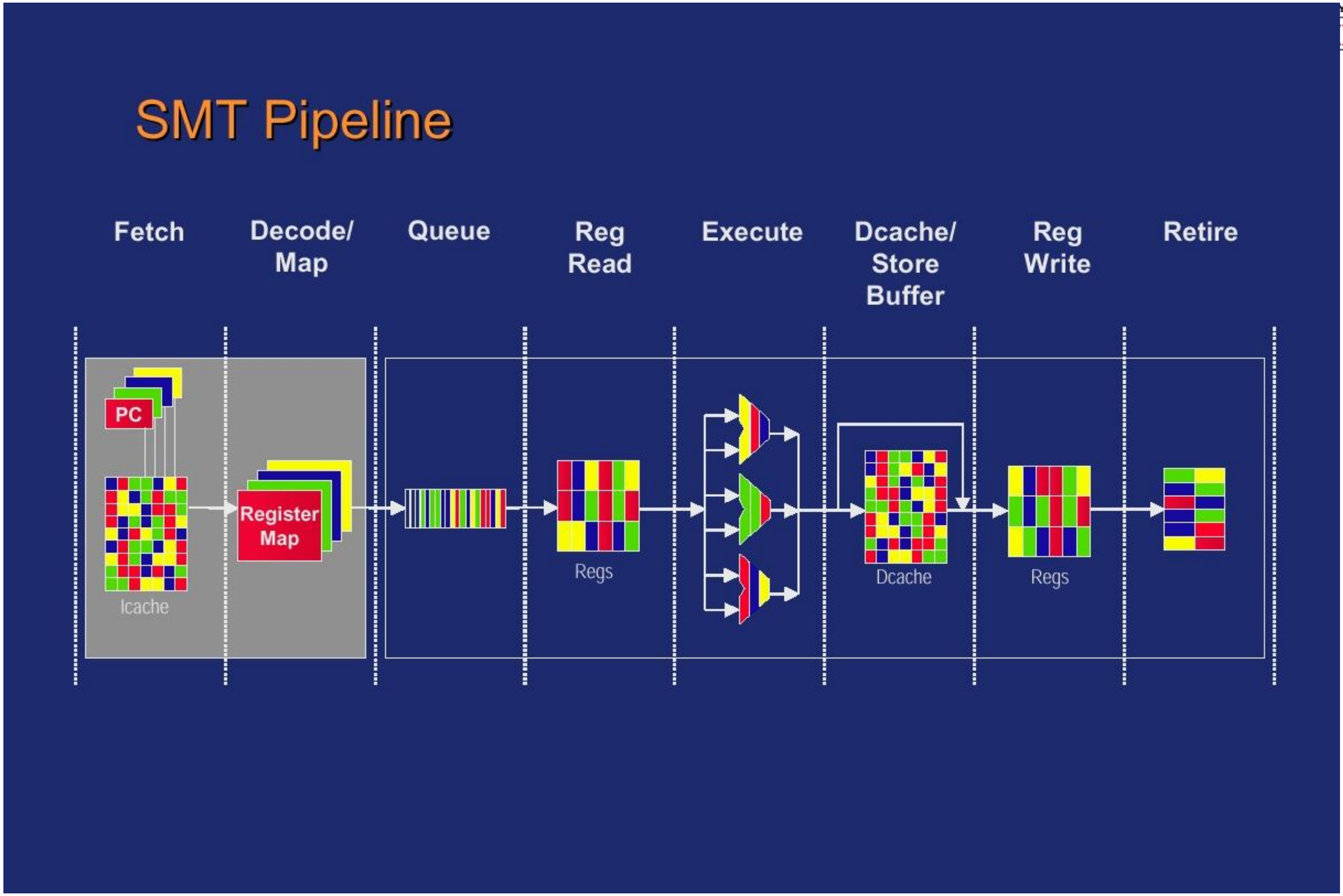
IBM Power4: Example CMP



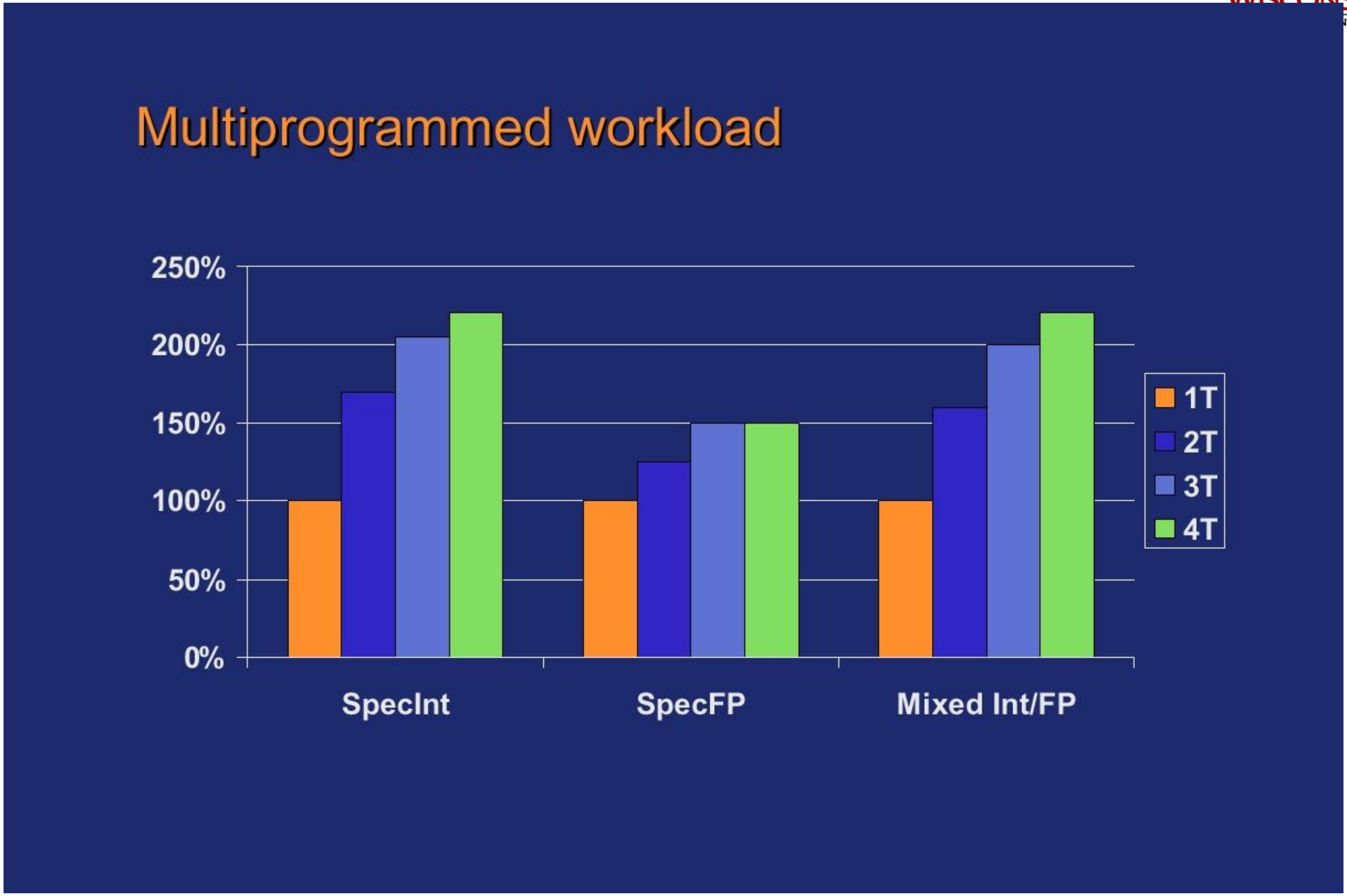
SMT Microarchitecture (from Emer, PACT '01)



SMT Microarchitecture (from Emer, PACT '01)



SMT Performance (from Emer, PACT '01)



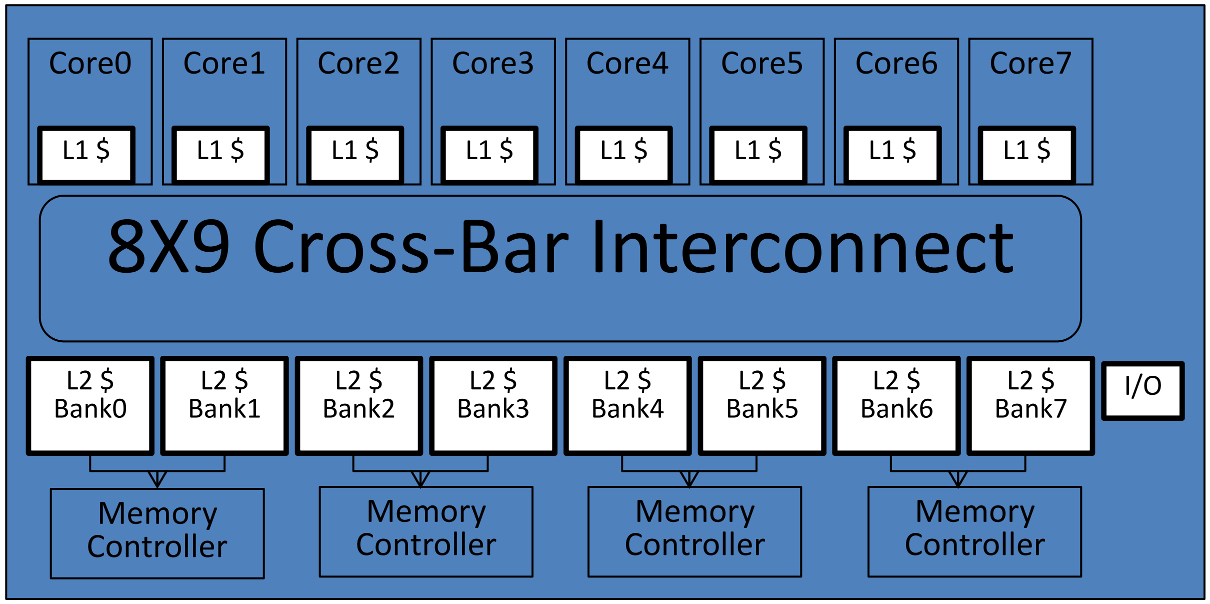
SMT Summary

- Goal: increase throughput
  - Not latency
- Utilize execution resources by sharing among multiple threads
- Usually some hybrid of fine-grained and SMT
  - Front-end is FG, core is SMT, back-end is FG
- Resource sharing
  - I$, D$, ALU, decode, rename, commit – shared
  - IQ, ROB, LQ, SQ – partitioned vs. shared

Multicore Interconnects

- Bus/crossbar - dismiss as short-term solutions?
- Point-to-point links, many possible topographies
  - 2D (suitable for planar realization)
    - Ring
    - Mesh
    - 2D torus
  - 3D - may become more interesting with 3D packaging (chip stacks)
    - Hypercube
    - 3D Mesh
    - 3D torus

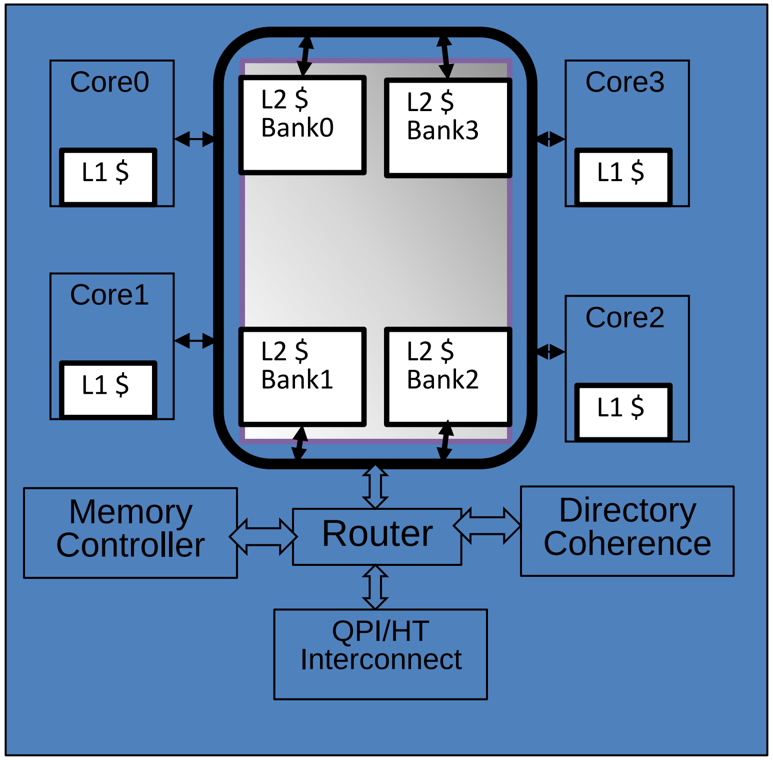
Cross-bar (e.g. IBM Power4/5/6/7)



On-Chip Bus/Crossbar

- Used widely (Power4/5/6/7 Piranha, Niagara, etc.)
  - Assumed not scalable
  - Is this really true, given on-chip characteristics?
  - May scale "far enough" : watch out for arguments at the limit
    - e.g. swizzle-switch makes x-bar scalable enough [UMich]
- Simple, straightforward, nice ordering properties
  - Wiring can be a nightmare (for crossbar)
  - Bus bandwidth is weak (even multiple busses)
  - Compare DEC Piranha 8-lane bus (32GB/s) to Power4 crossbar (100+GB/s)
  - Workload demands: commercial vs. scientific

On-Chip Ring (e.g. Intel)



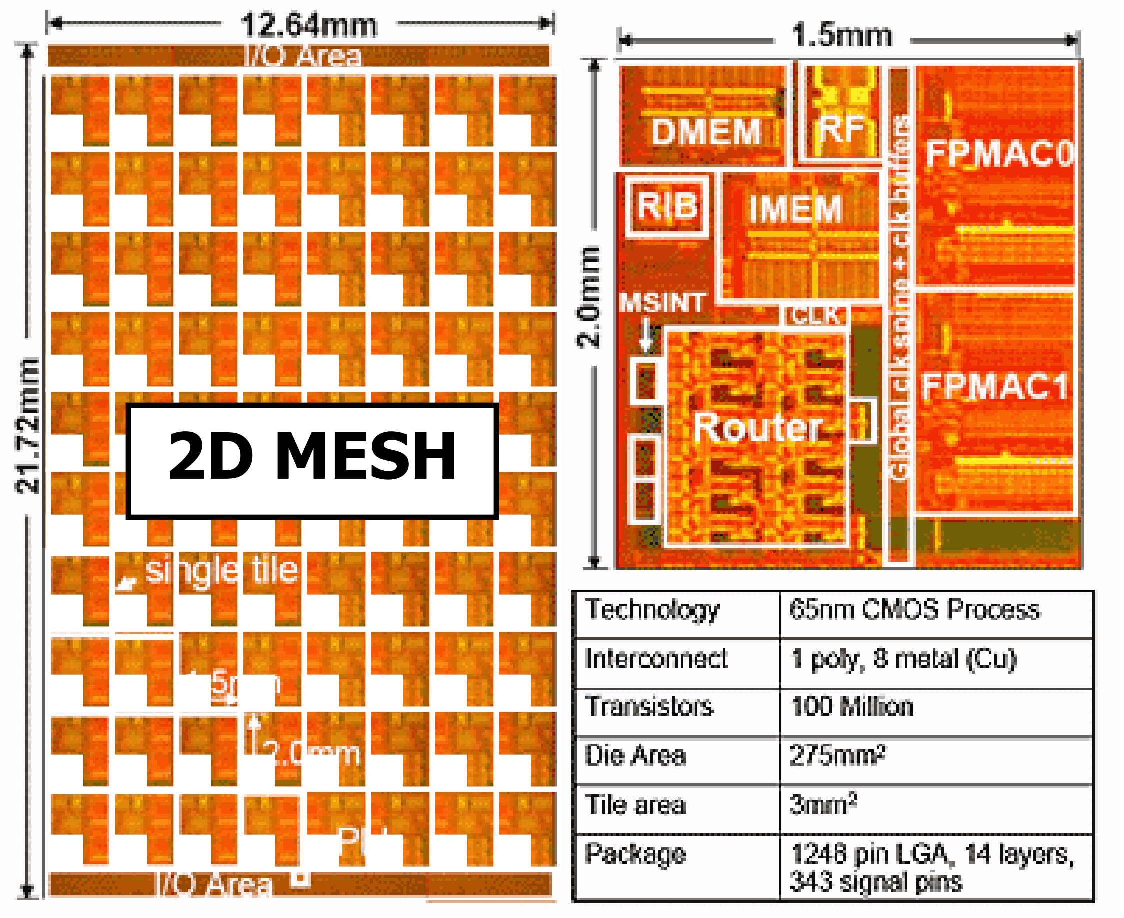
On-Chip Ring

- Point-to-point ring interconnect
  - Simple, easy
  - Nice ordering properties (unidirectional)
  - Every request a broadcast (all nodes can snoop)
  - Scales poorly: *O(n)* latency, fixed bandwidth
- Optical ring (nanophotonic)
  - HP Labs Corona project
  - Much lower latency (speed of light)
  - Still fixed bandwidth (but lots of it)

On-Chip Mesh

- Widely assumed in academic literature
- Tilera [Wentzlaff], Intel 80-core prototype
- Not symmetric, so have to watch out for load imbalance on inner nodes/links
  - 2D torus: wraparound links to create symmetry
    - Not obviously planar
    - Can be laid out in 2D but longer wires, more intersecting links
- Latency, bandwidth scale well
- Lots of recent research in the literature

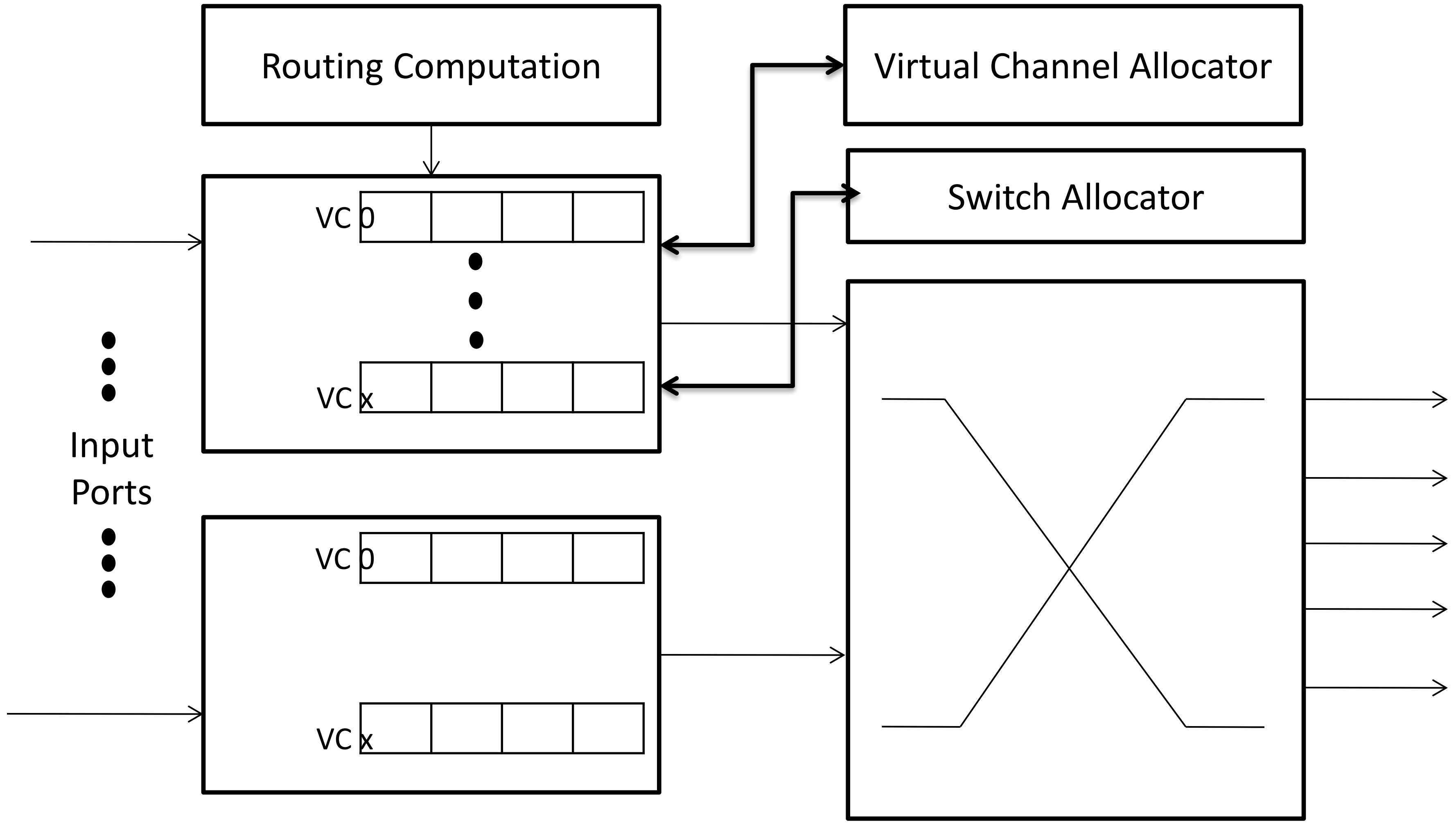
2D Mesh Example



Technology	65nm CMOS Process
Interconnect	1 poly, 8 metal (Cu)
Transistors	100 Million
Die Area	275mm²
Tile area	3mm²
Package	1248 pin LGA, 14 layers, 343 signal pins

- Intel Polaris
  - 80 core prototype
- Academic Research ex:
  - MIT Raw, TRIPs
    - 2-D Mesh Topology
    - Scalar Operand Networks

Virtual Channel Router



Baseline Router Pipeline

BW	RC	VA	SA	ST	LT

- Canonical 5-stage (+link) pipeline
  - BW: Buffer Write
  - RC: Routing computation
  - VA: Virtual Channel Allocation
  - SA: Switch Allocation
  - ST: Switch Traversal
  - LT: Link Traversal

On-chip Routers

Virtual Channel Router Pipeline Evolution

BW	RC	VA	SA	ST	LT
BW NRC	VA	SA	ST	LT	
BW NRC	VA SA	ST	LT		
VA NRC SA	ST	LT			

- 5-stages excessive for 1-cycle LT
- Collapsed into fewer and fewer pipestages
  - Speculation rampant

On-Chip Interconnects

- More coverage in ECE/CS 757 (usually)
- Synthesis lecture:
  - Natalie Enright Jerger & Li-Shiuan Peh, “On-Chip Networks”, Synthesis Lectures on Computer Architecture
  - http://www.morganclaypool.com/doi/abs/10.2200/S00209ED1V01Y200907CAC008

Lecture Summary

- ECE 757 Topics reviewed (briefly):
  - Thread-level parallelism
  - Synchronization
  - Coherence
  - Consistency
  - Multithreading
  - Multicore interconnects
- <u>Many</u> others not covered