Executing Multiple Threads

ECE/CS 752 Fall 2017

Prof. Mikko H. Lipasti
University of Wisconsin-Madison

Readings

- Read on your own:
  - Shen & Lipasti Chapter 11
  - G. S. Sohi, S. E. Breach and T.N. Vijaykumar. Multiscalar Processors, Proc. 22nd Annual International Symposium on Computer Architecture, June 1995.
  - Dean M. Tullsen, Susan J. Eggers, Joel S. Emer, Henry M. Levy, Jack L. Lo, and Rebecca L. Stamm. Exploiting Choice: Instruction Fetch and Issue on an Implementable Simultaneous Multithreading Processor, Proc. 23rd Annual International Symposium on Computer Architecture, May 1996 (B5)
- To be discussed in class:
  - Review #6 due 11/17/2017: Y.-H. Chen, J. Emer, V. Sze, "Eyeriss: A Spatial Architecture for Energy-Efficient Dataflow for Convolutional Neural Networks," International Symposium on Computer Architecture (ISCA), pp. 367-379, June 2016. Online PDF

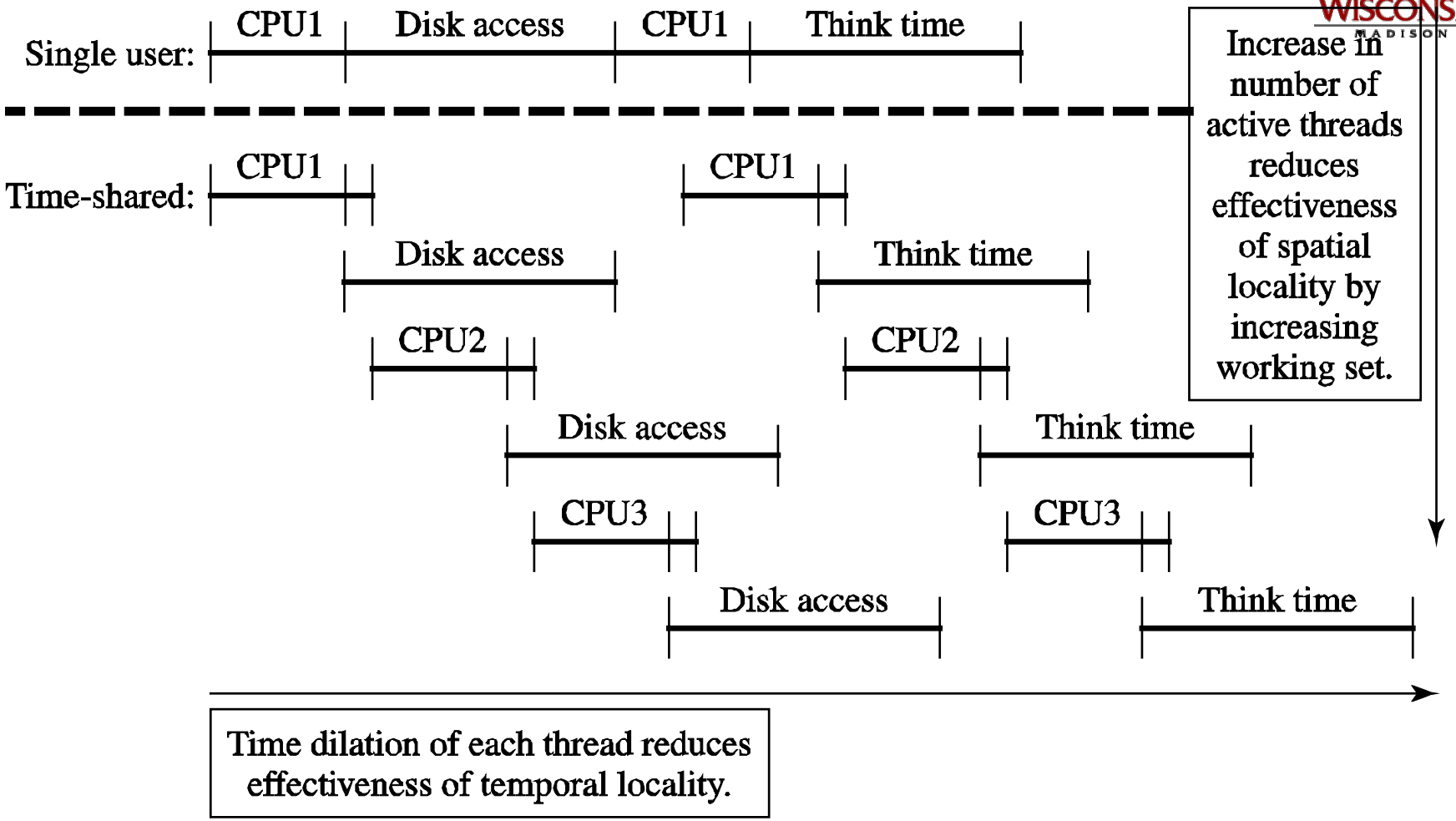
Executing Multiple Threads

- Thread-level parallelism
- Synchronization
- Multiprocessors
- Explicit multithreading
- Data parallel architectures
- Multicore interconnects
- Implicit multithreading: Multiscalar
- Niagara case study

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



- 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)

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
fetchadd A, 1	fetchadd A, 3

(a)

Thread 0	Thread 1
spin: cmpswp AL, 1 bfail spin load r1, A addi r1, rl, 1 store r1, A store 0, AL	spin: cmpswp AL, 1 bfail spin load r1, A addi r1, rl, 3 store r1, A store 0, AL

(b)

Thread 0	Thread 1
spin: ll r1, A addi r1, r1, 1 stc r1, A bfail spin	spin: ll r1, A addi r1, r1, 3 stc r1, A bfail spin

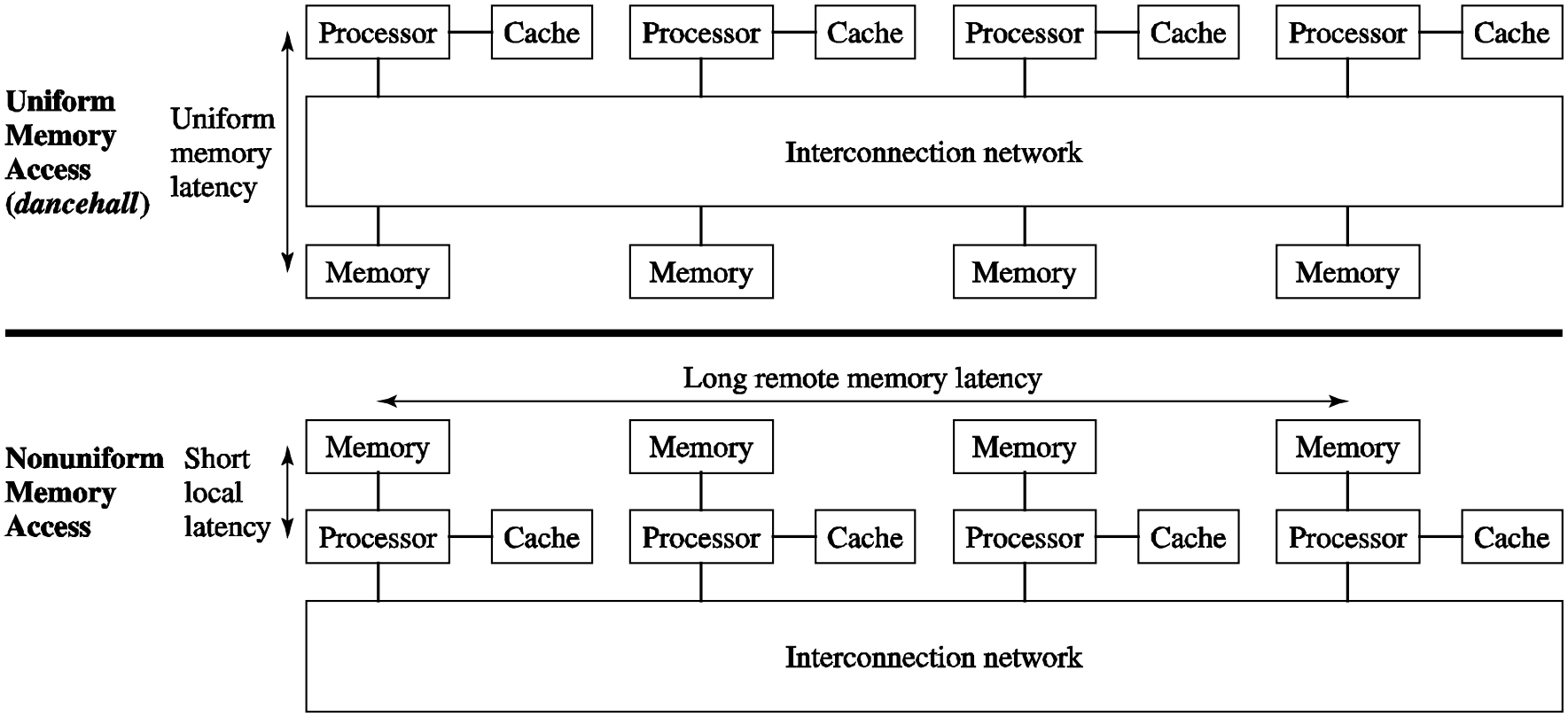
(c)

- 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

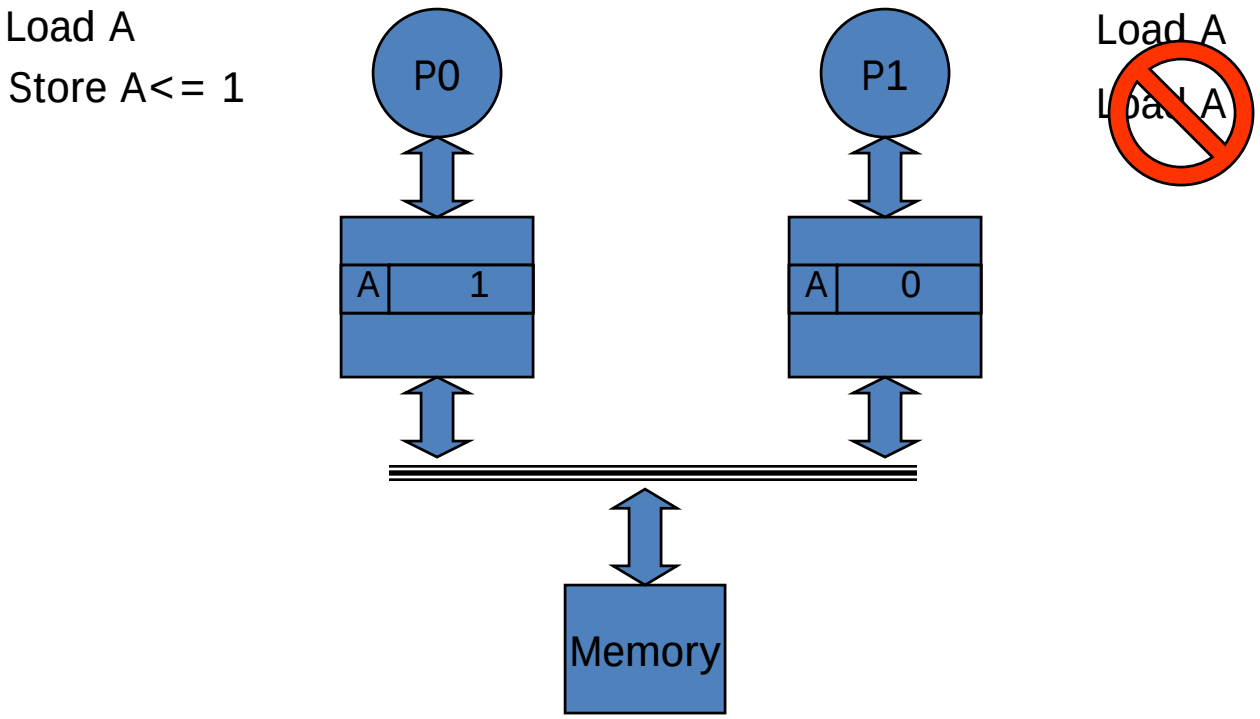
Multiprocessor Systems

- Focus on 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
    - Have asymmetric processing units
- Shared memory idealisms
  - Fully shared memory: *usually nonuniform latency*
  - Unit latency: *approximate with caches*
  - Lack of contention: *approximate with caches*
  - Instantaneous propagation of writes: *coherence required*

UMA vs. NUMA


Uniform Memory Access (dancehall)
Uniform memory latency
Processor
Cache
Interconnection network
Memory
Long remote memory latency
Nonuniform Memory Access
Short local latency

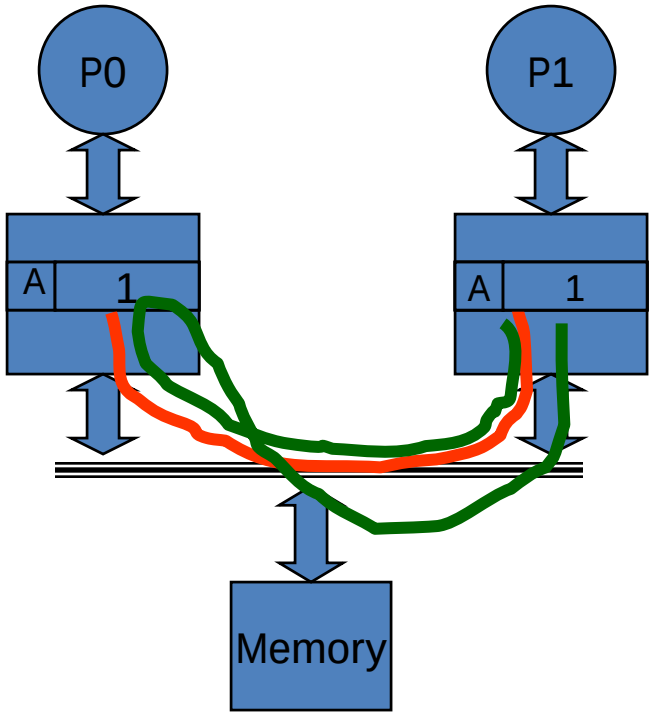
Cache Coherence Problem



Cache Coherence Problem

Load A

Store A<= 1



Load A

Load A

Invalidate Protocol

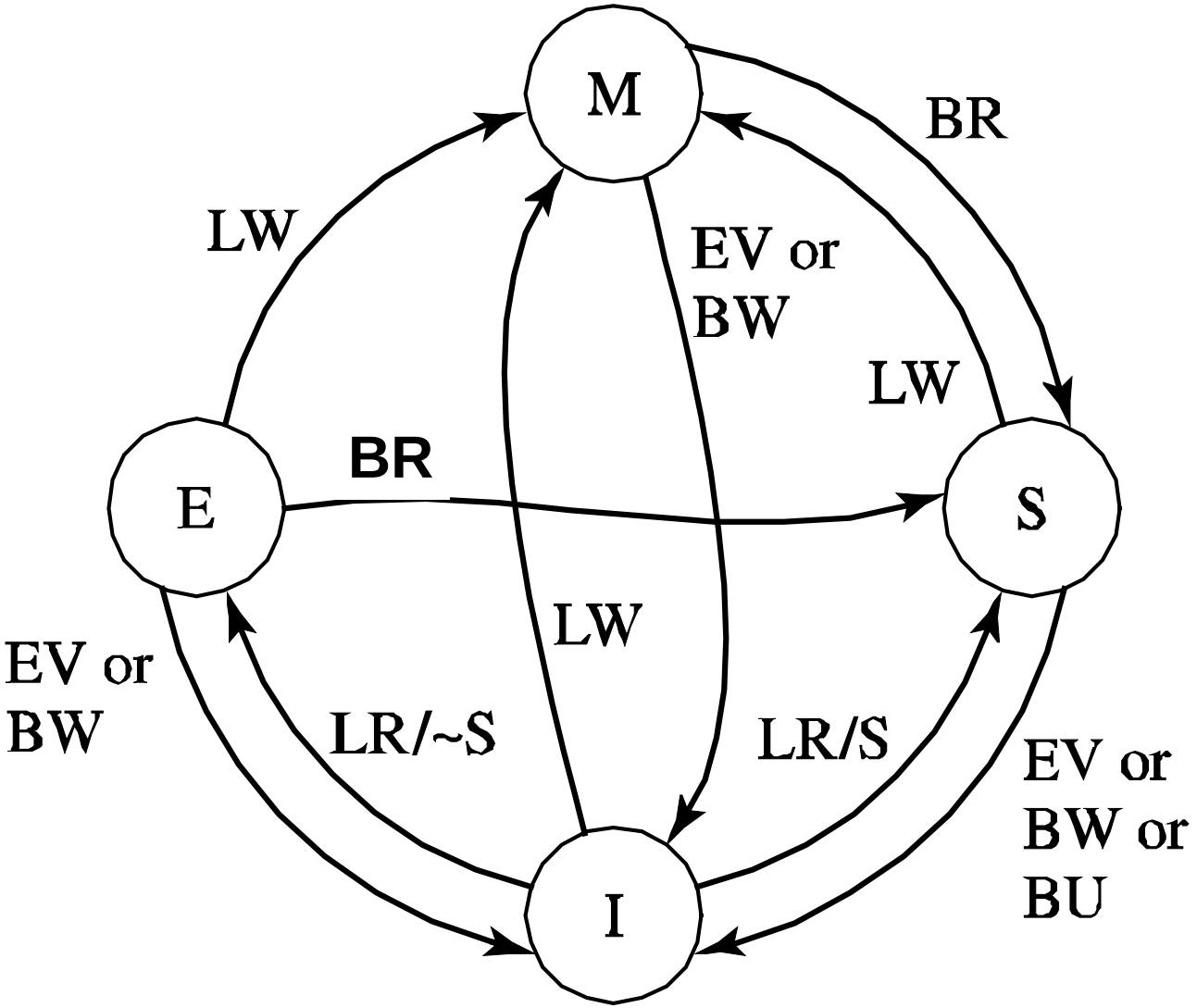
THE UNIVERSITY of WISCONSIN MADISON

- 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

Implementing Cache Coherence

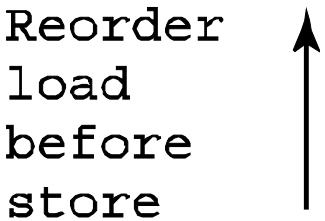
THE UNIVERSITY of WISCONSIN MADISON

- Snooping implementation
  - 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*)
- Snooping suffers from:
  - Scalability: shared busses not practical
  - Ordering of requests without a shared bus
  - Lots of prior work on scaling snoop-based systems

Alternative to Snooping

- 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)
- Directory shortcomings
  - Indirection through directory has latency penalty
  - Directory overhead for **all** memory, not just what is cached
  - If shared line is dirty in other CPU's cache, directory must forward request, adding latency
  - This can severely impact performance of applications with heavy sharing (e.g. relational databases)

Memory Consistency

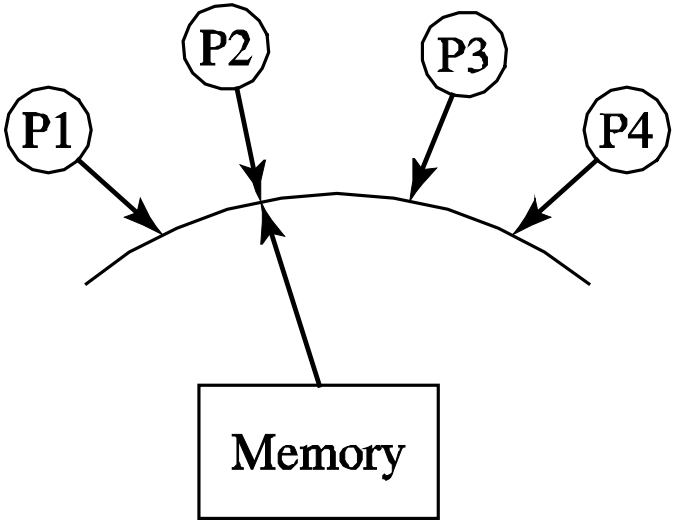



```
Proc0                          Proc1

st A=1                         st B=1
if (load B==0) {               if (load A==0) {
  ...critical section            ...critical section
}                              }
```

- How are memory references from different processors interleaved?
- If this is not well-specified, synchronization becomes difficult or even impossible
  - ISA must specify consistency model
- Common example using Dekker's algorithm for synchronization
  - If load reordered ahead of store (as we assume for a baseline OOO CPU)
  - Both Proc0 and Proc1 enter critical section, since both observe that other's lock variable (A/B) is not set
- If consistency model allows loads to execute ahead of stores, Dekker's algorithm no longer works
  - Common ISAs allow this: IA-32, PowerPC, SPARC, Alpha

Sequential Consistency [Lamport 1979]

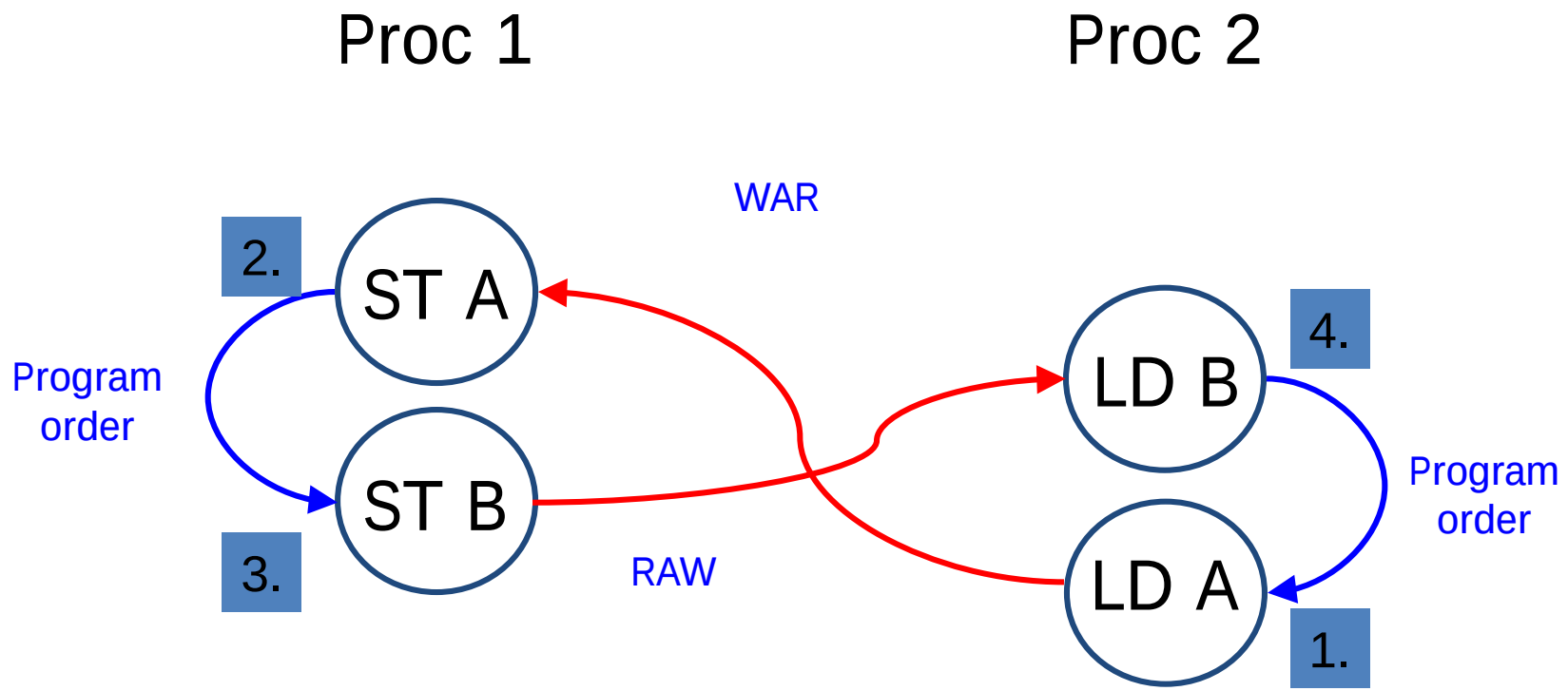


- Processors treated as if they are interleaved processes on a single time-shared CPU
- All references must fit into a total global order or interleaving that does not violate any CPUs program order
  - Otherwise sequential consistency not maintained
- Now Dekker's algorithm will work
- Appears to preclude any OOO memory references
  - Hence precludes any real benefit from OOO CPUs

High-Performance Sequential Consistency

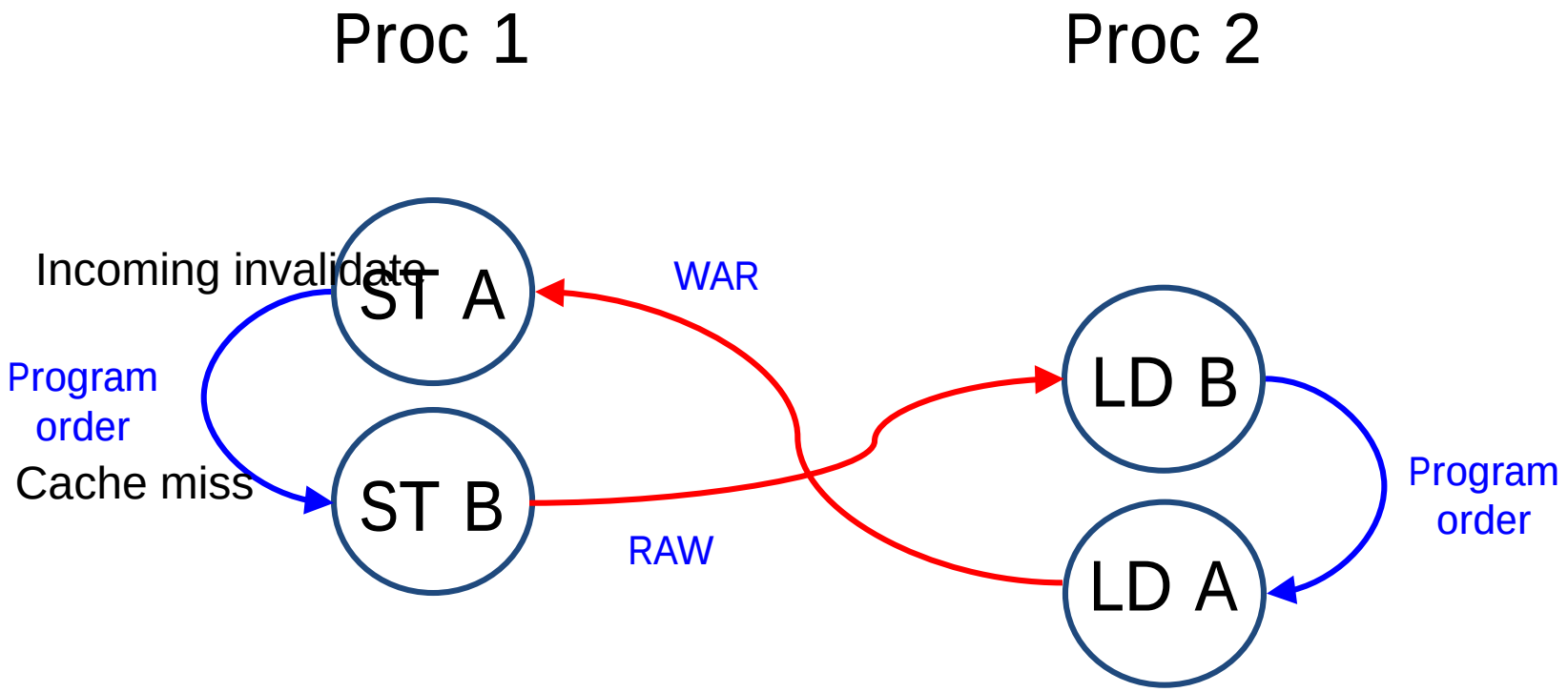
- Coherent caches isolate CPUs if no sharing is occurring
  - Absence of coherence activity means CPU is free to reorder references
- Still have to order references with respect to misses and other coherence activity (snoops)
- Key: use speculation
  - Reorder references speculatively
  - Track which addresses were touched speculatively
  - Force replay (in order execution) of such references that collide with coherence activity (snoops)

Constraint graph example - SC

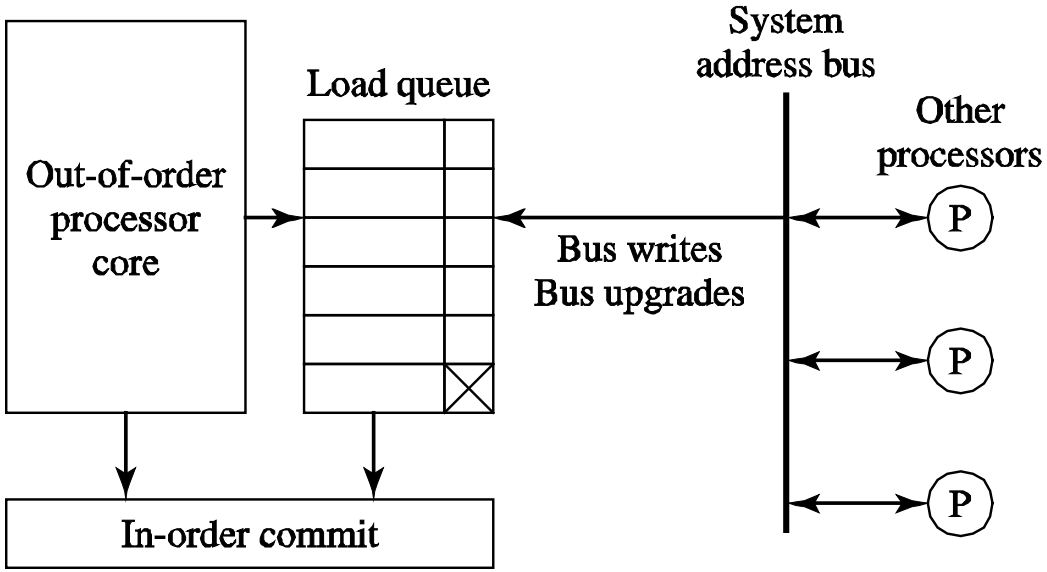


Cycle indicates that execution is incorrect

Anatomy of a cycle



High-Performance Sequential Consistency

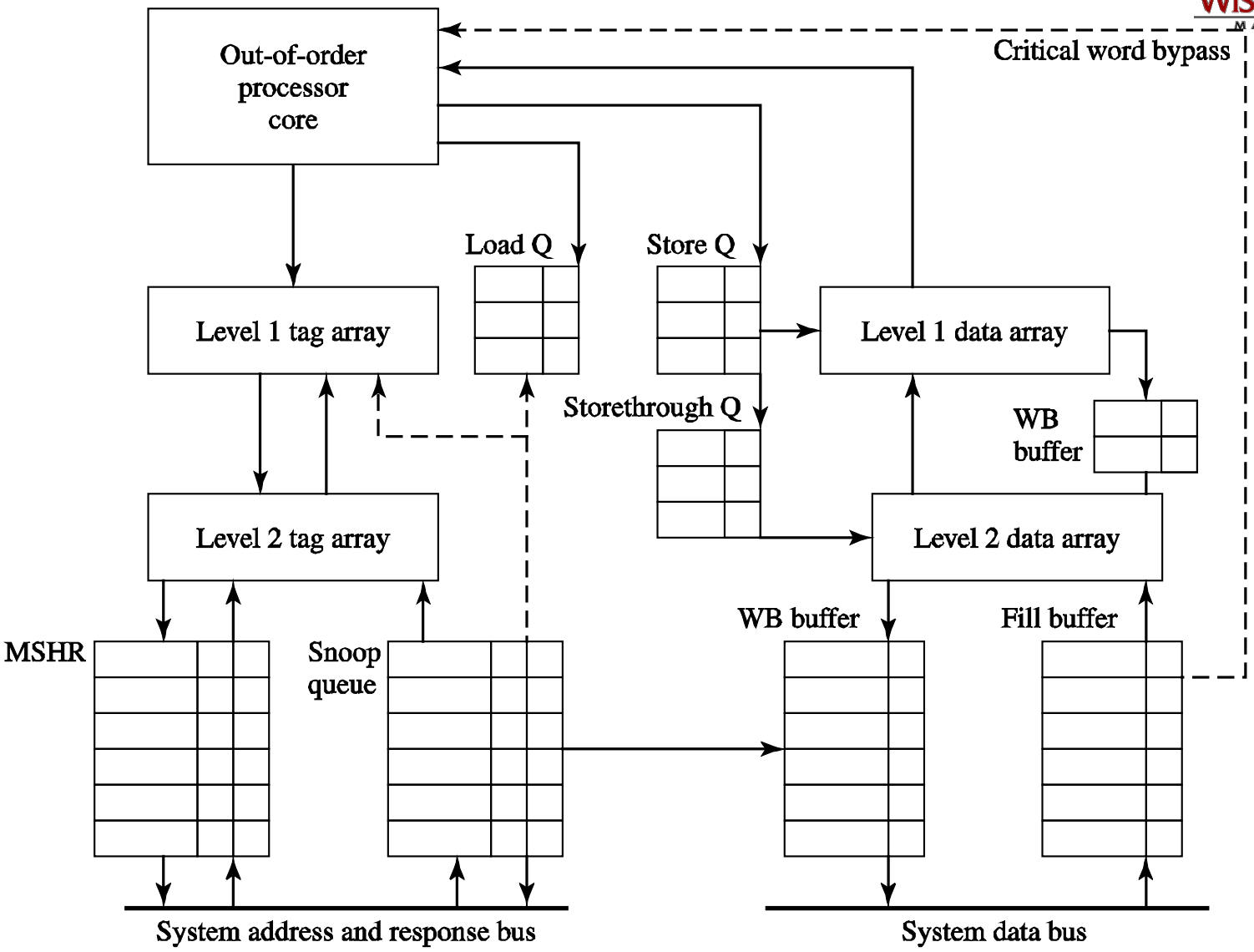


- Load queue records all speculative loads
- Bus writes/upgrades are checked against LQ
- Any matching load gets marked for replay
- At commit, loads are checked and replayed if necessary
  - Results in machine flush, since load-dependent ops must also replay
- Practically, conflicts are rare, so expensive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

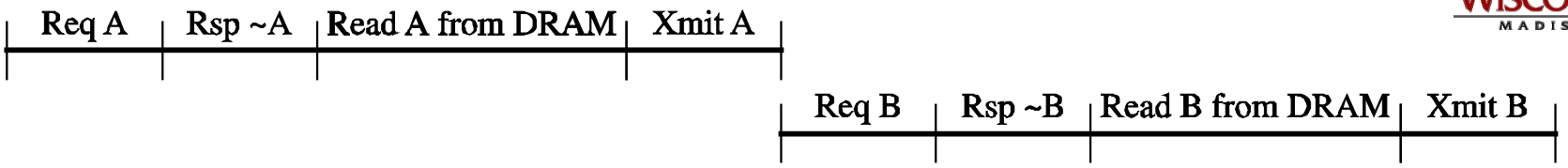
Relaxed Consistency Models

- Key insight: only synchronizing references need ordering
- Hence, relax memory for all other references
  - Enable high-performance OOO implementation
- Require programmer to **label** synchronization references
  - Hardware must carefully order these labeled references
  - All other references can be performed out of order
- Labeling schemes:
  - Explicit synchronization ops (acquire/release)
  - Memory fence or memory barrier ops:
    - All preceding ops must finish before following ones begin
- Often: fence ops cause pipeline drain in modern OOO machine
- More: ECE/CS 757

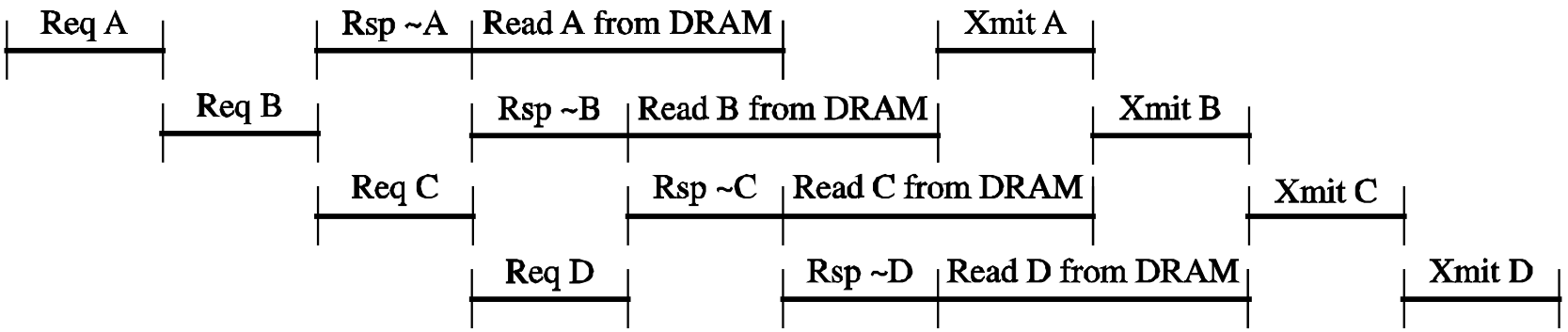
Coherent Memory Interface



Split Transaction Bus



(a) Simple bus with atomic transactions

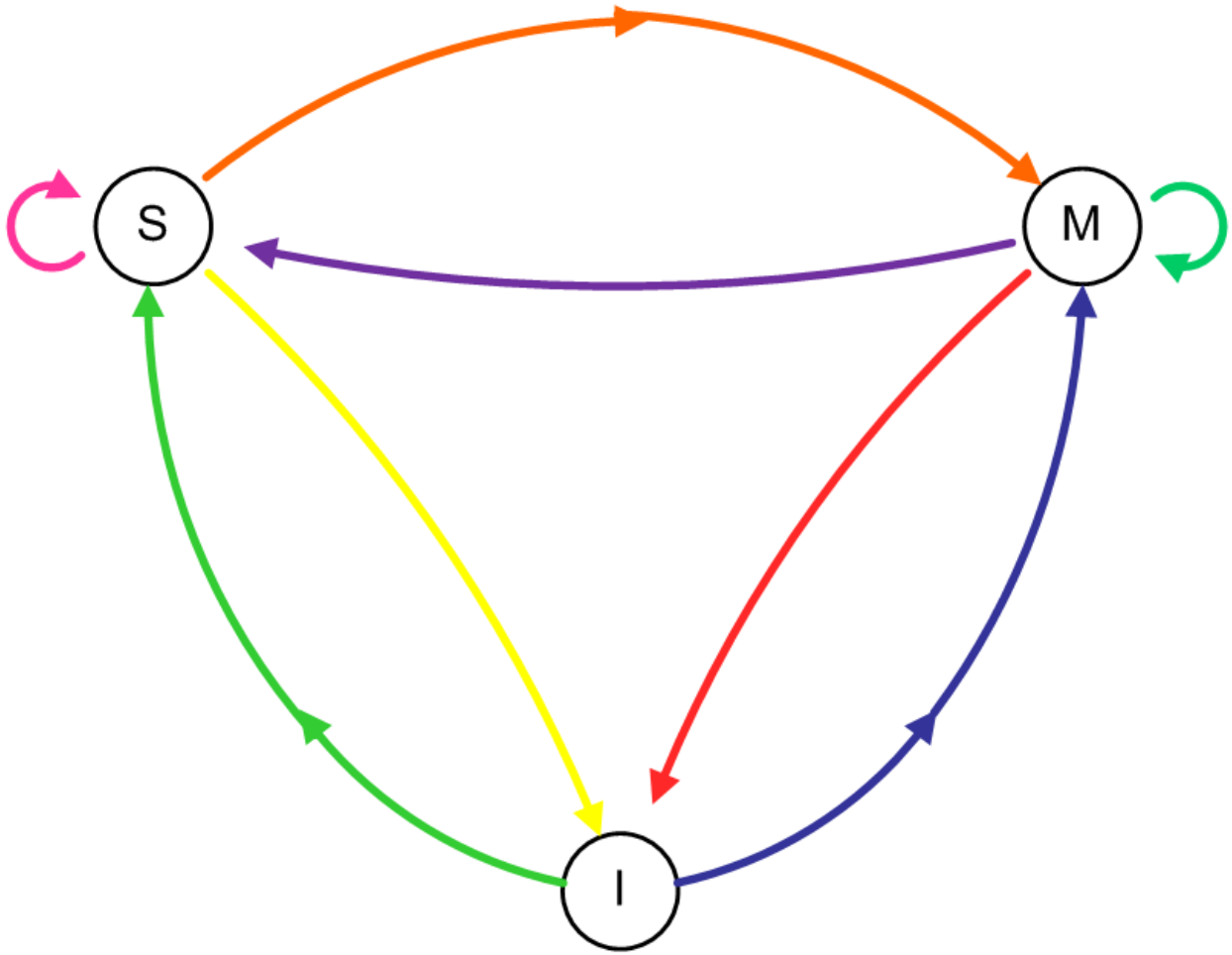


(b) Split-transaction bus with separate requests and responses

- "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)

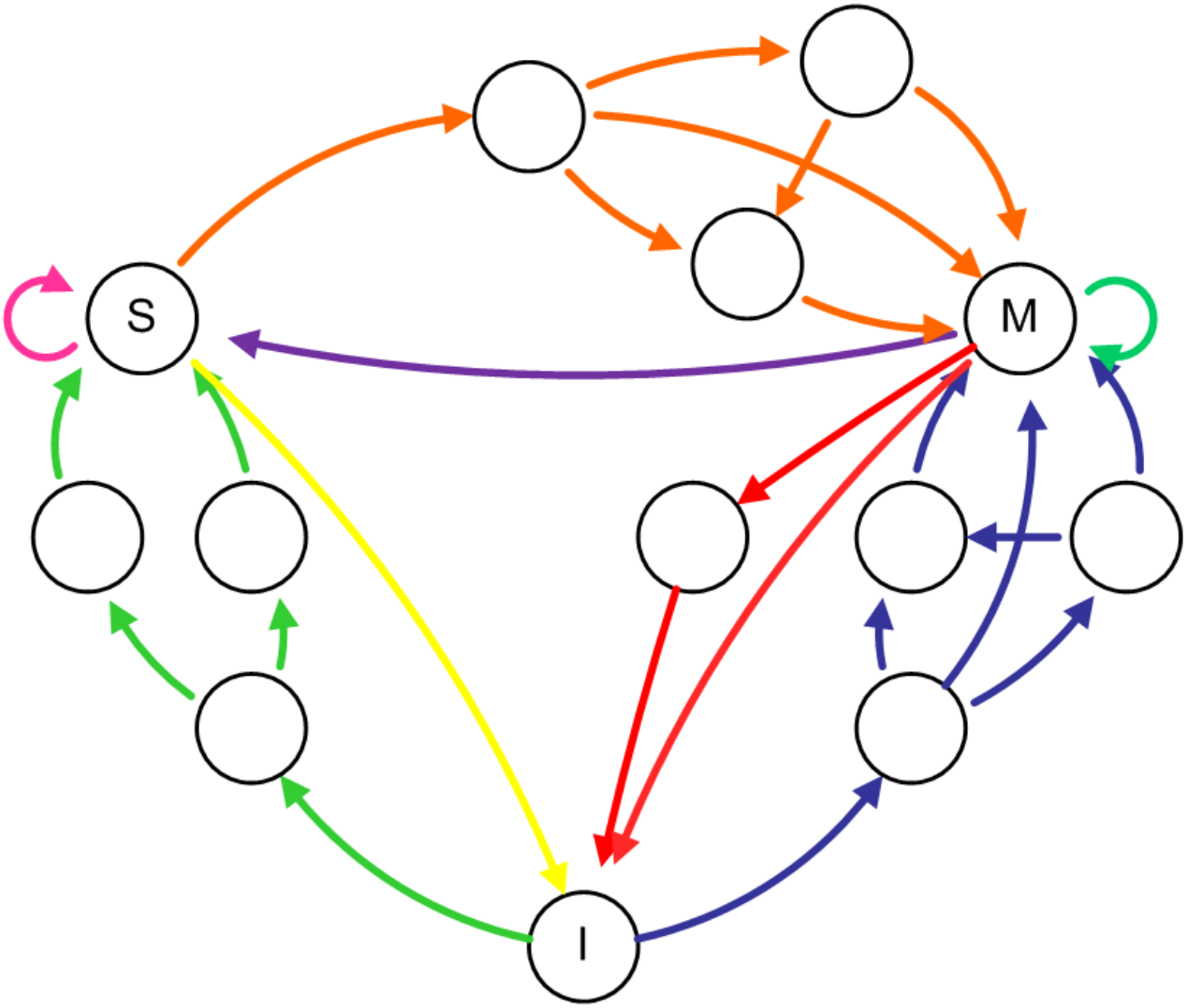
High Level



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

High Level

Busy States

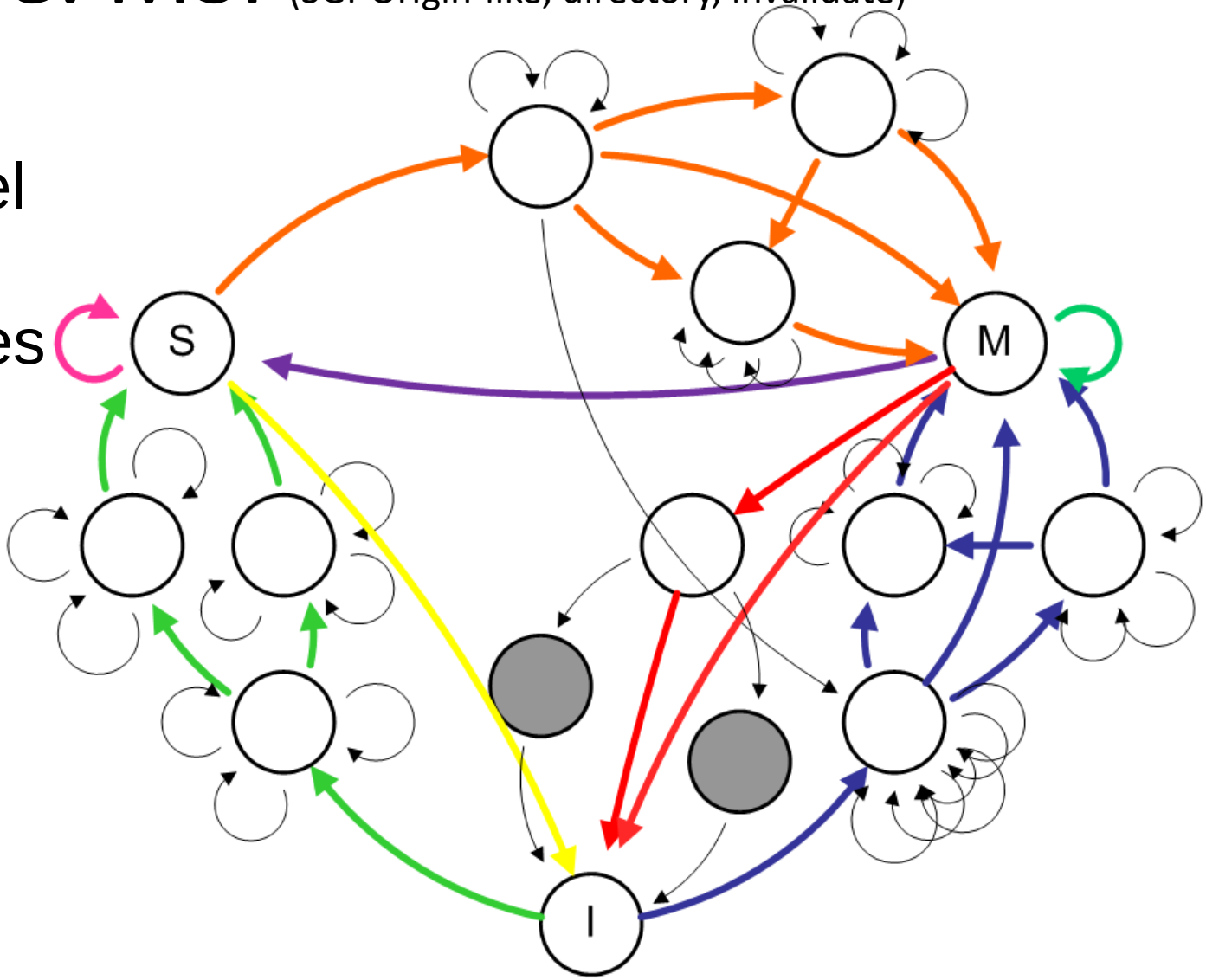


Example: MSI (SGI-Origin-like, directory, invalidate)

High Level

Busy States

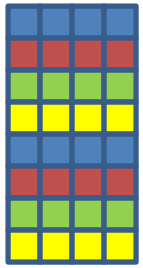
Races



Multithreaded Cores

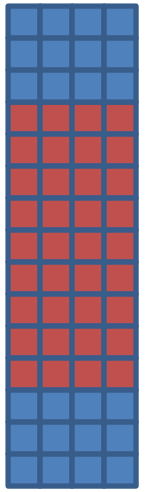
- 1990's: Memory wall and multithreading
  - Processor-DRAM speed mismatch:
    - nanosecond to fractions of a microsecond (1:500)
  - H/W task switch used to bring in other useful work while waiting for cache miss
  - Cost of context switch must be much less than cache miss latency
- Very attractive for applications with abundant thread-level parallelism
  - Commercial multi-user workloads

Approaches to Multithreading



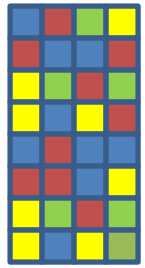
- Fine-grain multithreading
  - Switch contexts at fixed fine-grain interval (e.g. every cycle)
  - Need enough thread contexts to cover stalls
  - Example: Tera MTA, 128 contexts, no data caches
- Benefits:
  - Conceptually simple, high throughput, deterministic behavior
- Drawback:
  - Very poor single-thread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

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 (2T)	L3, DRAM, system interface
AMD Opteron	2, 4, 6, 12	No	System interface (socket), L3

Approaches to Multithreading

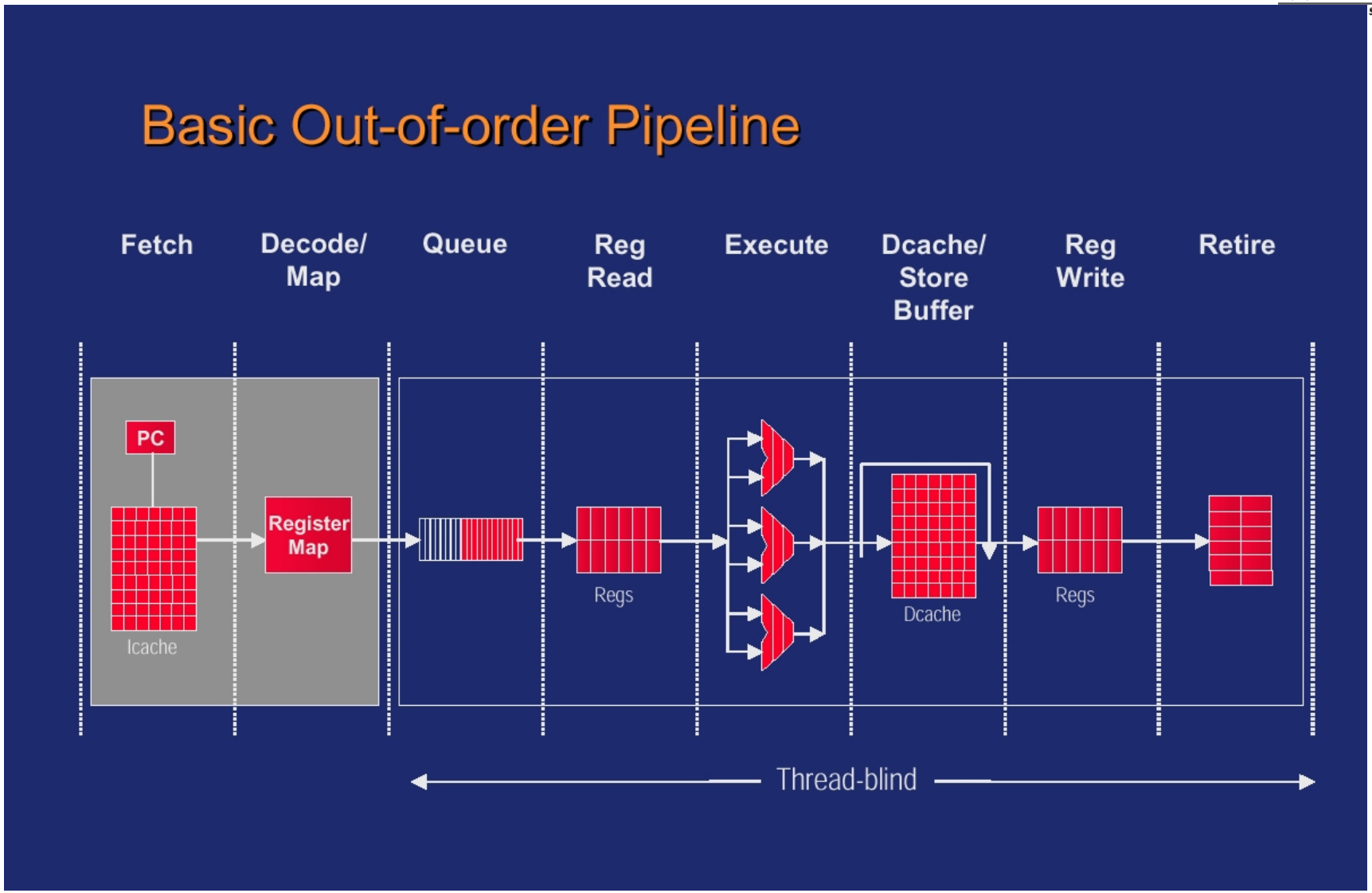
- Chip Multithreading (CMT)
  - Similar to CMP
- Share something in the core:
  - Expensive resource, e.g. floating-point unit (FPU)
  - Also share L2, system interconnect (memory and I/O bus)
- Examples:
  - Sun Niagara, 8 cores per die, one FPU
  - AMD Bulldozer: one FP cluster for every two INT clusters
- 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

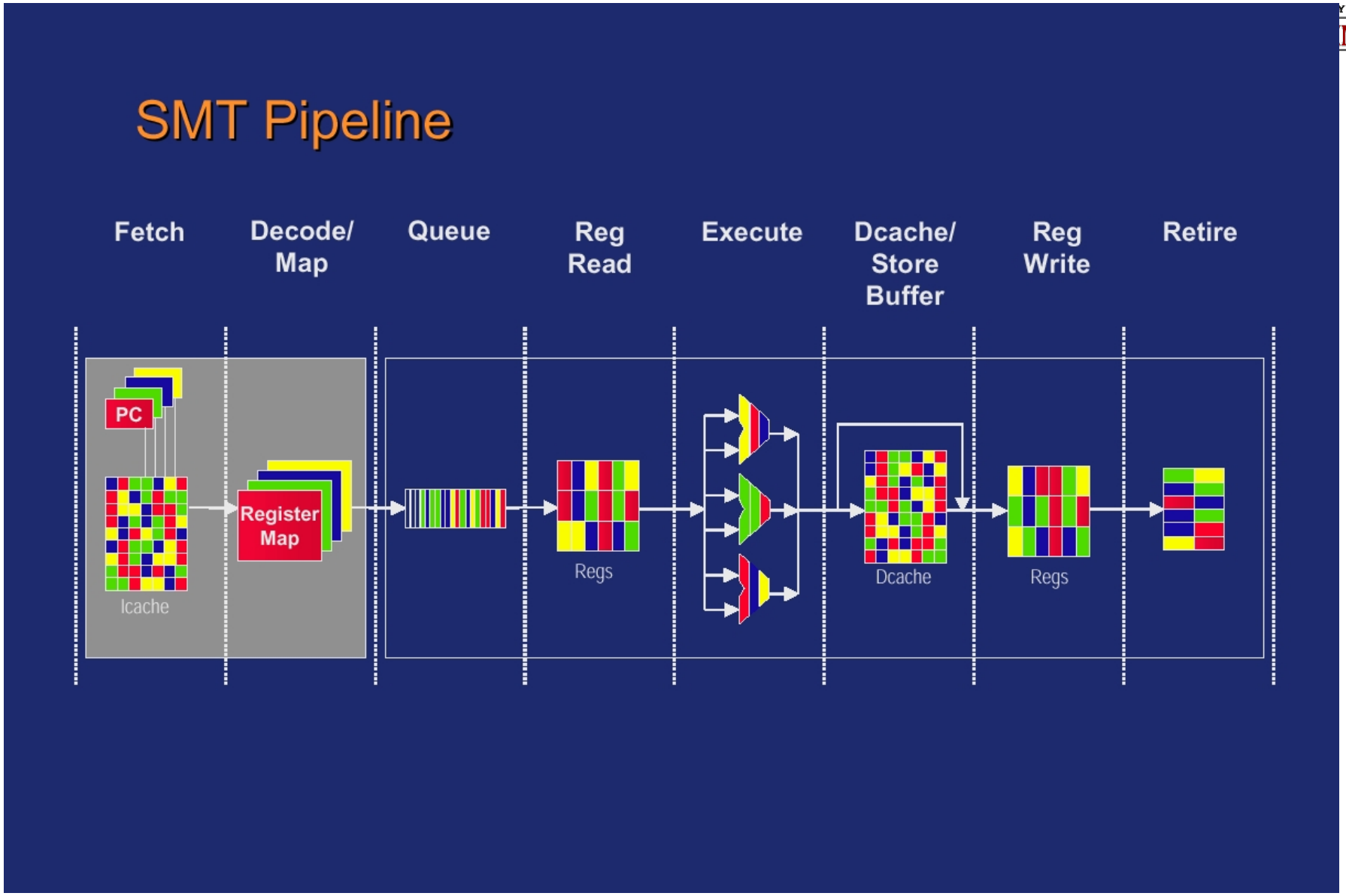
MT Approach	Resources shared between threads	Context Switch Mechanism
None	Everything	Explicit operating system context switch
Fine-grained	Everything but register file and control logic/state	Switch every cycle
Coarse-grained	Everything but I-fetch buffers, register file and con trol logic/state	Switch 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

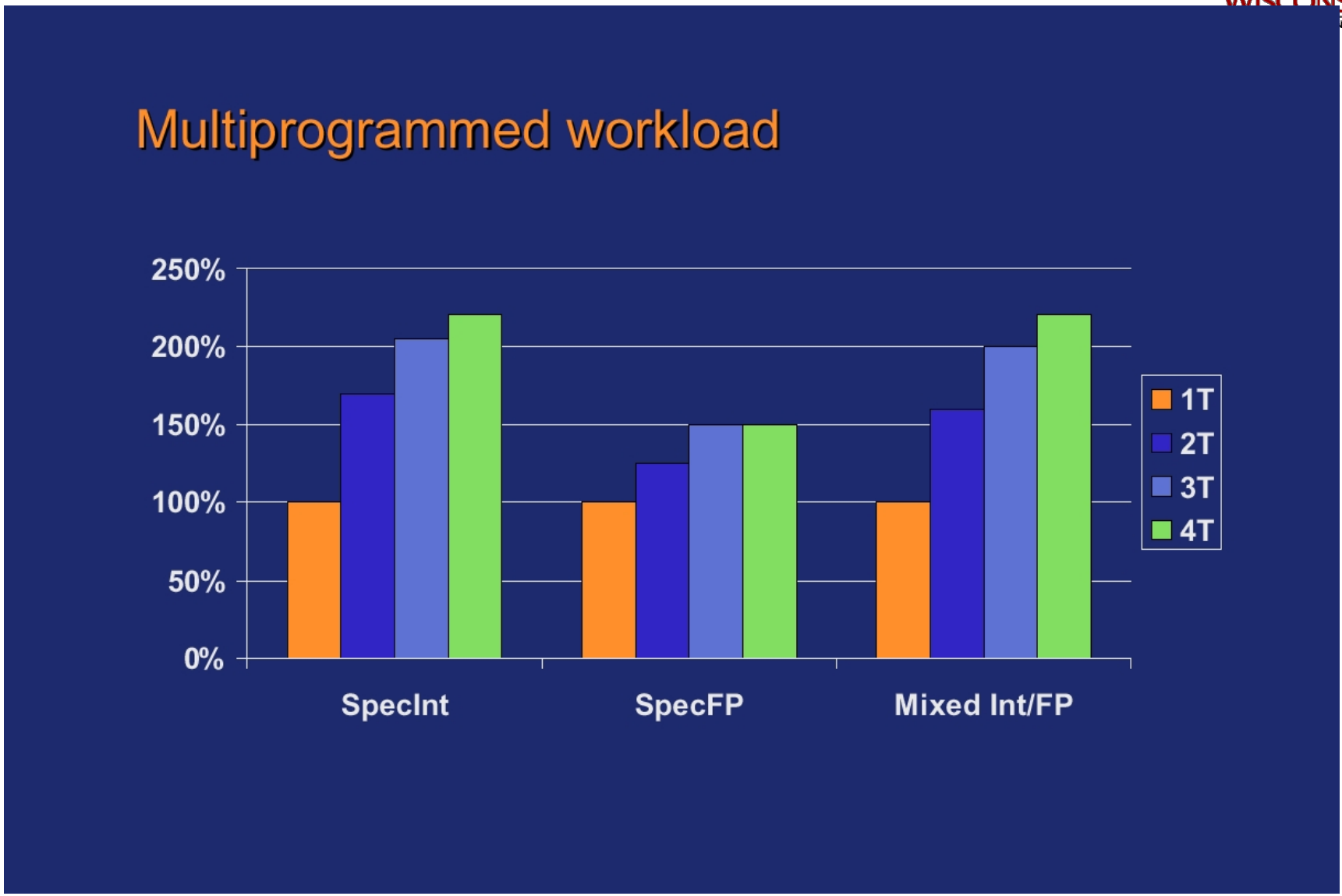
SMT Microarchitecture (from Emer, PACT '01)



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

Data Parallel Architectures

```
for ( i = 0; i < n; i++ )
  C[i] = x * A[i] + B[2*i];
```

(a) Regular DA, Regular CF

```
for ( i = 0; i < n; i++ )
  E[C[i]] = D[A[i]] + B[i];
```

(b) Irregular DA, Regular CF

```
for ( i = 0; i < n; i++ )
  x = ( A[i] > 0 ) ? y : z;
  C[i] = x * A[i] + B[i];
```

(c) Regular DA, Irregular CF

```
for ( i = 0; i < n; i++ )
  if ( A[i] > 0 )
    C[i] = x * A[i] + B[i];
```

(d) Irregular DA, Irregular CF

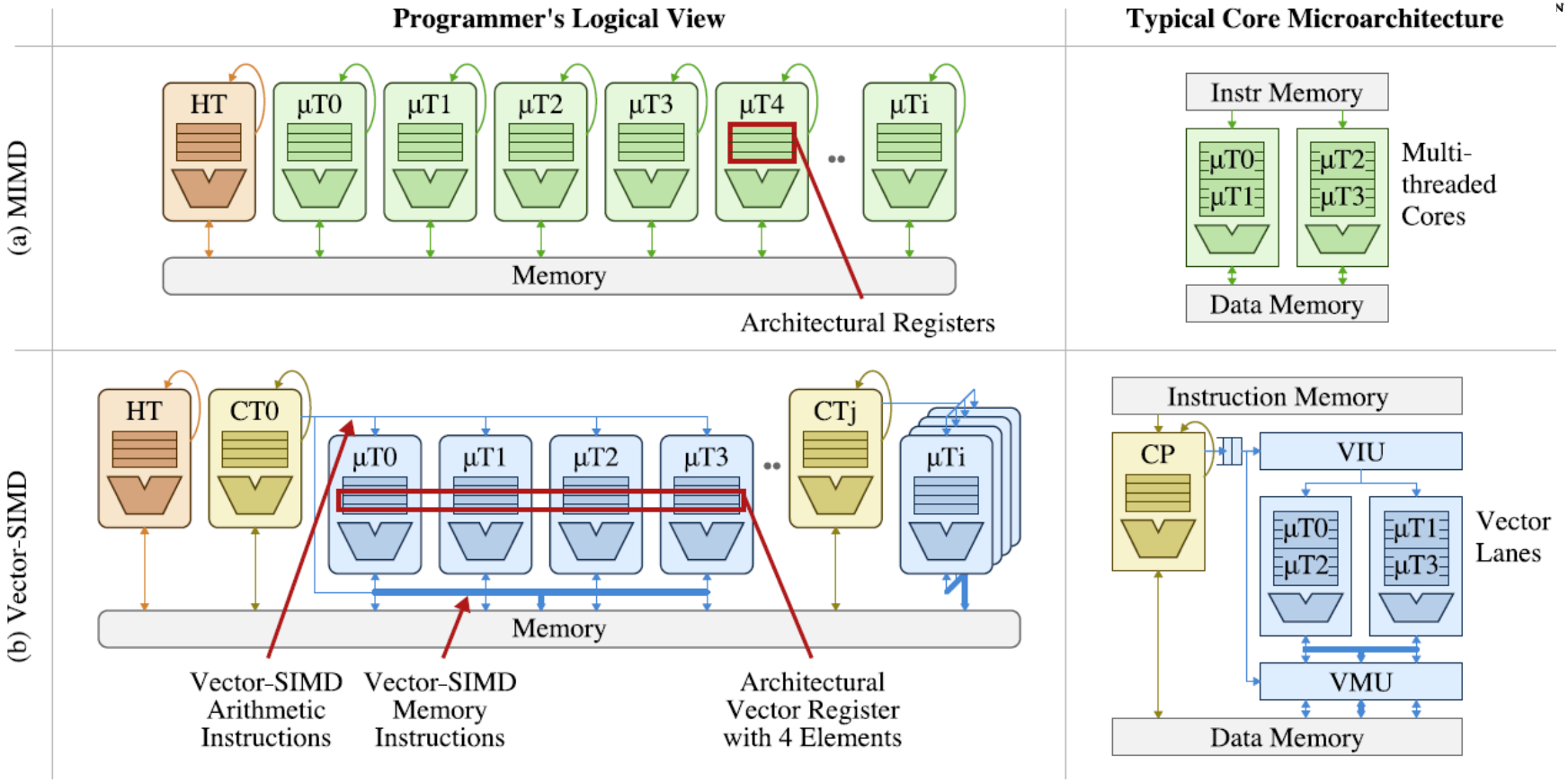
```
for ( i = 0; i < n; i++ )
  C[i] = false; j = 0;
  while ( !C[i] & (j < m) )
    if ( A[i] == B[j++] )
      C[i] = true;
```

(e) Irregular DA, Irregular CF

From [Lee et al., ISCA '11]

- Regular vs. Irregular data access, control flow

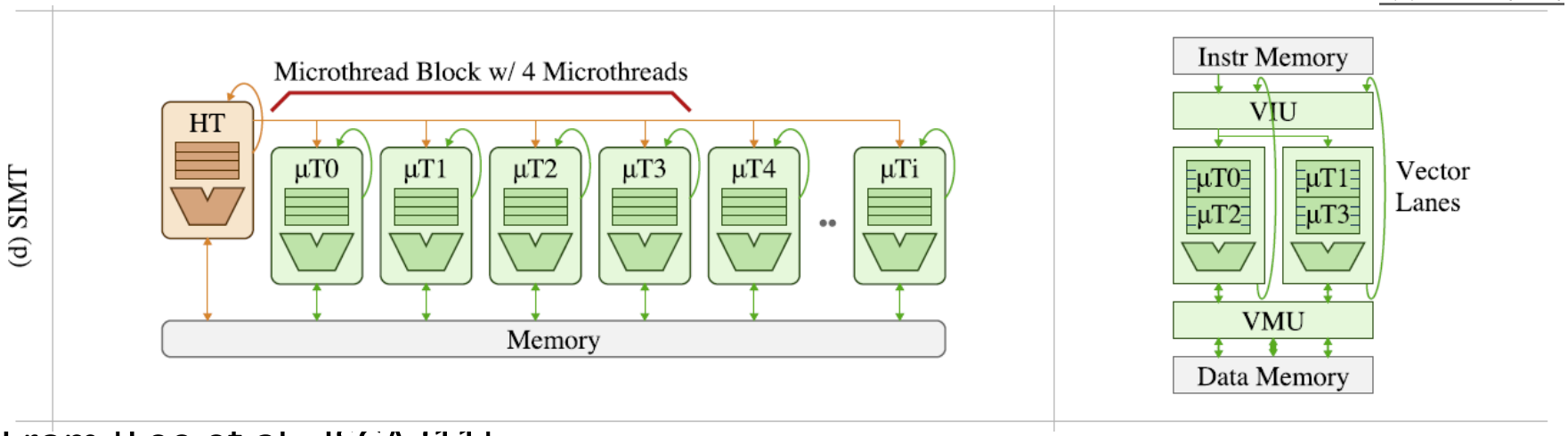
Data Parallel Execution



From [Lee et al., ISCA '11]

- MIMD vs. SIMD

Data Parallel Execution



From [Lee et al., ISCA '11]

- SIMT [Nvidia GPUs]
  - Large number of threads, MIMD programmer view
  - Threads ganged into warps, executed in SIMD fashion for efficiency
  - Control/data divergence causes inefficiency
  - Programmer optimization required (ECE 759)

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

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
- Simple, straightforward, nice ordering properties
  - Wiring is a nightmare (for crossbar)
  - Bus bandwidth is weak (even multiple busses)
  - Workload:
    - “Commercial” applications usually latency-limited
    - “Scientific” applications usually bandwidth-limited

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

On-Chip Mesh

- Widely assumed in academic literature
- Tilera (MIT startup), 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 existing literature

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

Implicitly Multithreaded Processors

- Goal: speed up execution of a single thread (latency)
- Implicitly break program up into multiple smaller threads, execute them in parallel, e.g.:
  - Parallelize loop iterations across multiple processing units
  - Usually, exploit control independence in some fashion
  - **Not** parallelism of order 100x, more like 3-5x
- Typically, one of two goals:
  - Expose more ILP for a single window, or
  - Build a more scalable, partitioned execution window
- Or, try to achieve both

Implicitly Multithreaded Processors

- Many challenges:
  - Find additional ILP, past hard-to-predict branches
    - Control independence
  - Maintain data dependences (RAW, WAR, WAW) for registers
  - Maintain precise state for exception handling
  - Maintain memory dependences (RAW/WAR/WAW)
  - Maintain memory consistency model
- Still a research topic
  - Multiscalar reading provides historical context
  - Lots of related work in TLS (thread-level speculation)

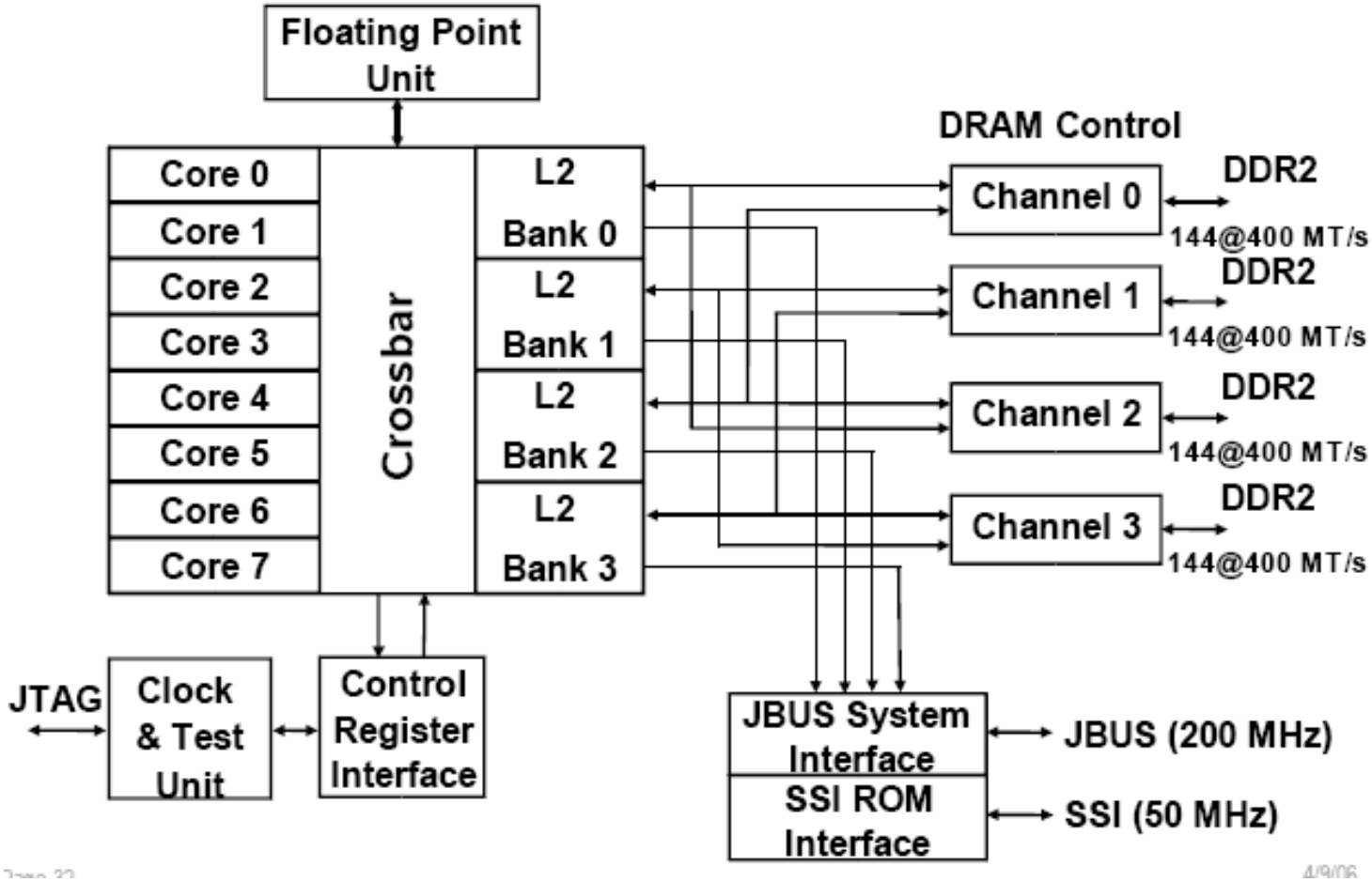
Multiscalar

- Seminal work on implicit multithreading
  - Started in mid 80's under Guri Sohi @ Wisconsin
- Solved many of the "hard" problems
- Threads or *tasks* identified by **compiler**
  - Tasks look like mini-programs, can contain loops, branches
- Hardware consists of a ring of processing nodes
  - Head processor executes most speculative task
  - Tail processor commits and resolves
  - Miss-speculation causes task and all newer tasks to get flushed
- Nodes connected to:
  - Sequencing unit that dispatches tasks to each one
  - Shared register file that resolves RAW/WAR/WAW
  - Address Resolution Buffer: resolves memory dependences
- http://www.cs.wisc.edu/mscalar
  - Publications, theses, tools, contact information

Niagara Case Study

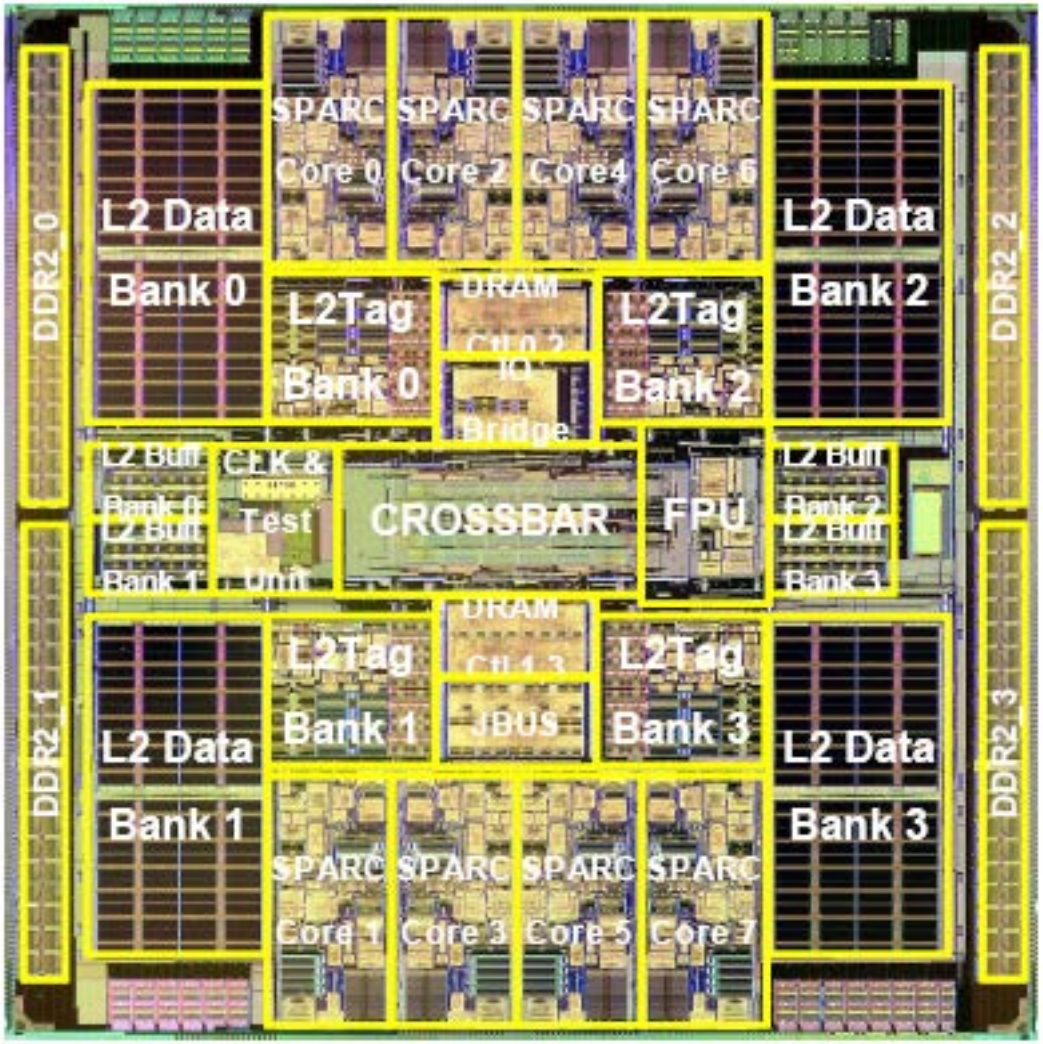
- Targeted application: web servers
  - Memory intensive (many cache misses)
  - ILP limited by memory behavior
  - TLP: Lots of available threads (one per client)
- Design goal: maximize throughput (/watt)
- Results:
  - Pack many cores on die (8)
  - Keep cores simple to fit 8 on a die, share FPU
  - Use multithreading to cover pipeline stalls
  - Modest frequency target (1.2 GHz)

Niagara Block Diagram [Source: J. Laudon]



- 8 in-order cores, 4 threads each
- 4 L2 banks, 4 DDR2 memory controllers

Ultrasparc T1 Die Photo [Source: J. Laudon]



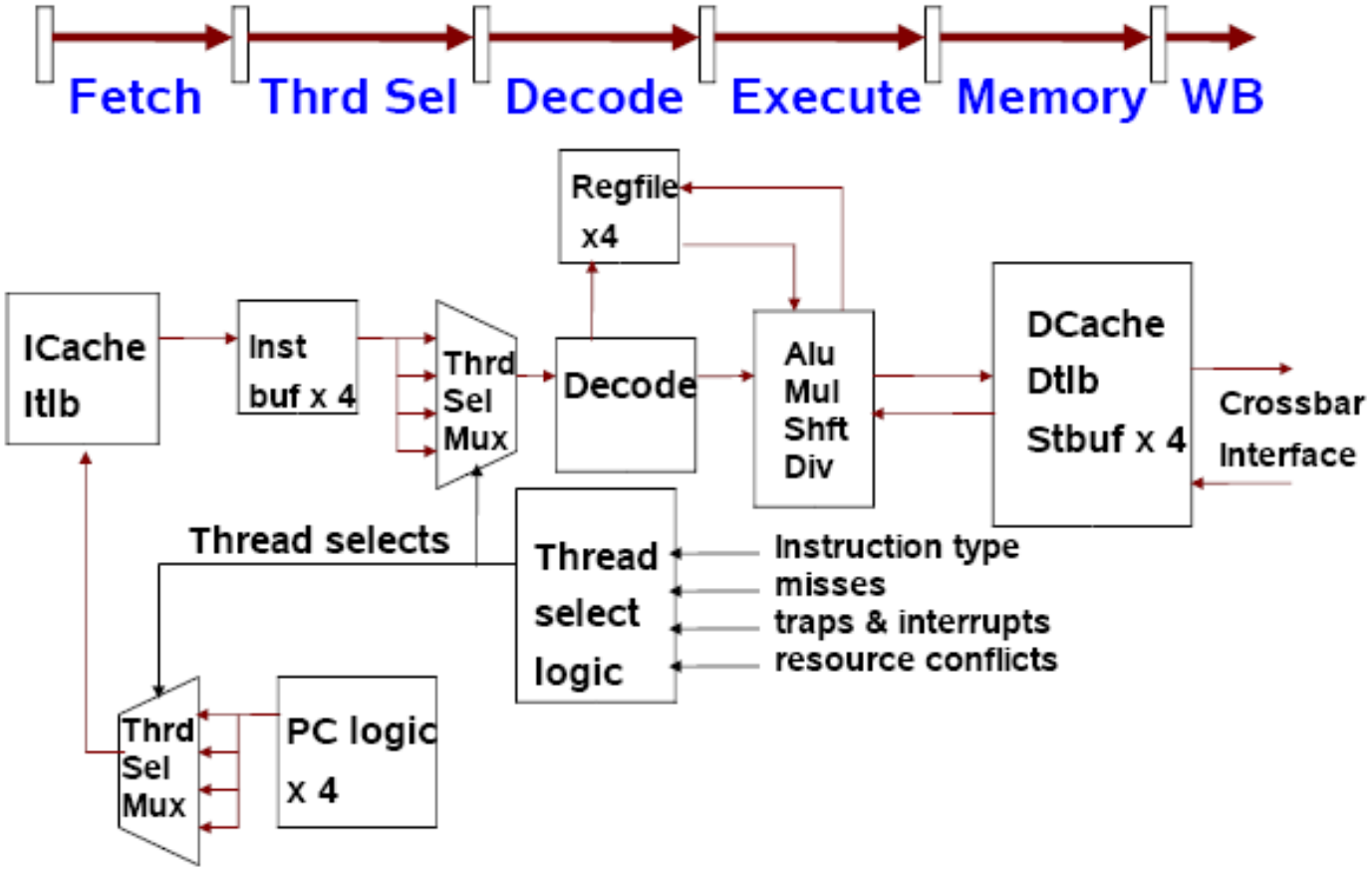
Features:

- 8 64-bit Multithreaded SPARC Cores
- Shared 3 MB, 12-way 64B line writeback L2 Cache
- 16 KB, 4-way 32B line ICache per Core
- 8 KB, 4-way 16B line write-through DCache per Core
- 4 144-bit DDR-2 channels
- 3.2 GB/sec JBUS I/O

Technology:

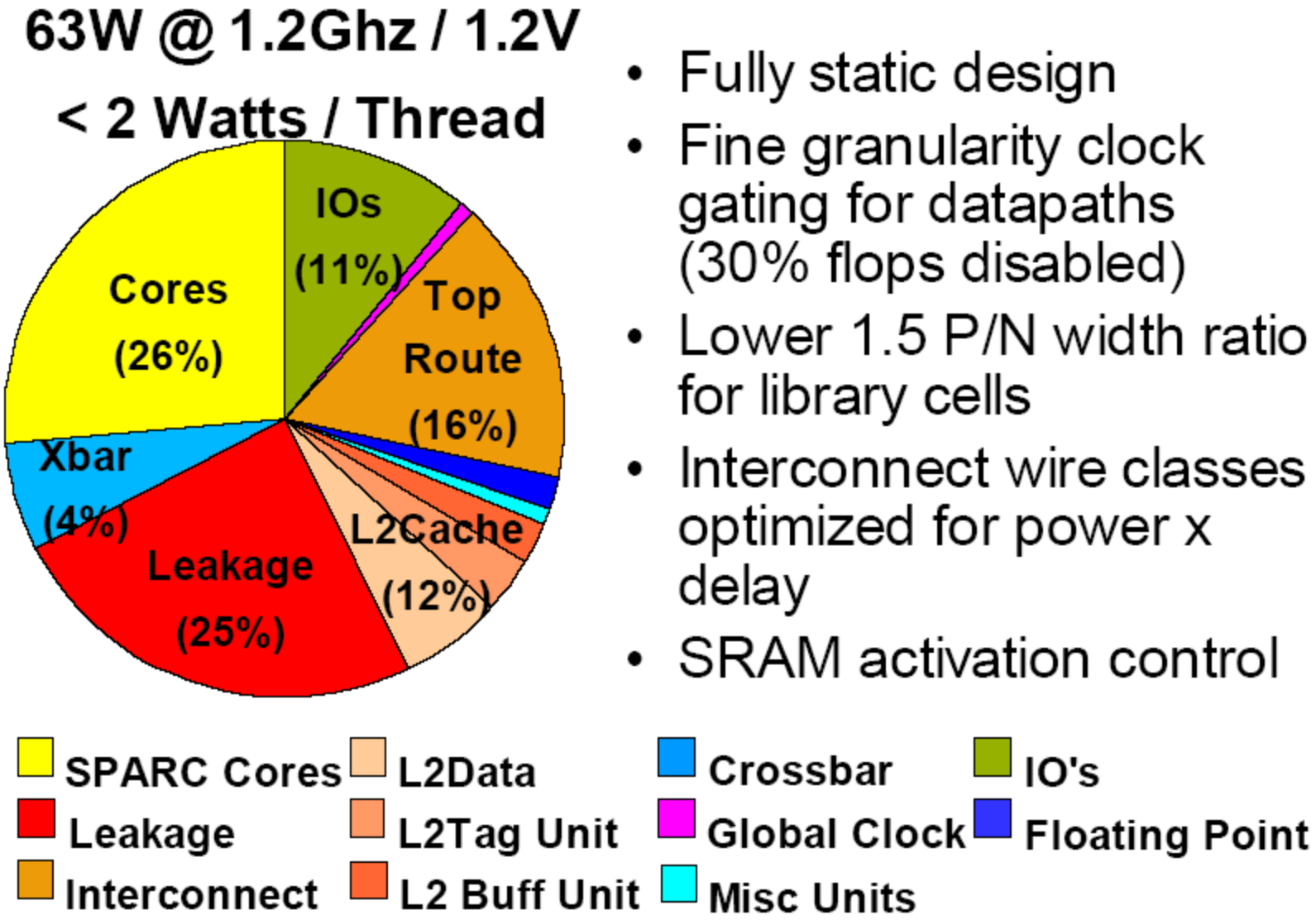
- TI's 90nm CMOS Process
- 9LM Cu Interconnect
- 63 Watts @ 1.2GHz/1.2V
- Die Size: 379mm²
- 279M Transistors
- Flip-chip ceramic LGA

Niagara Pipeline [Source: J. Laudon]



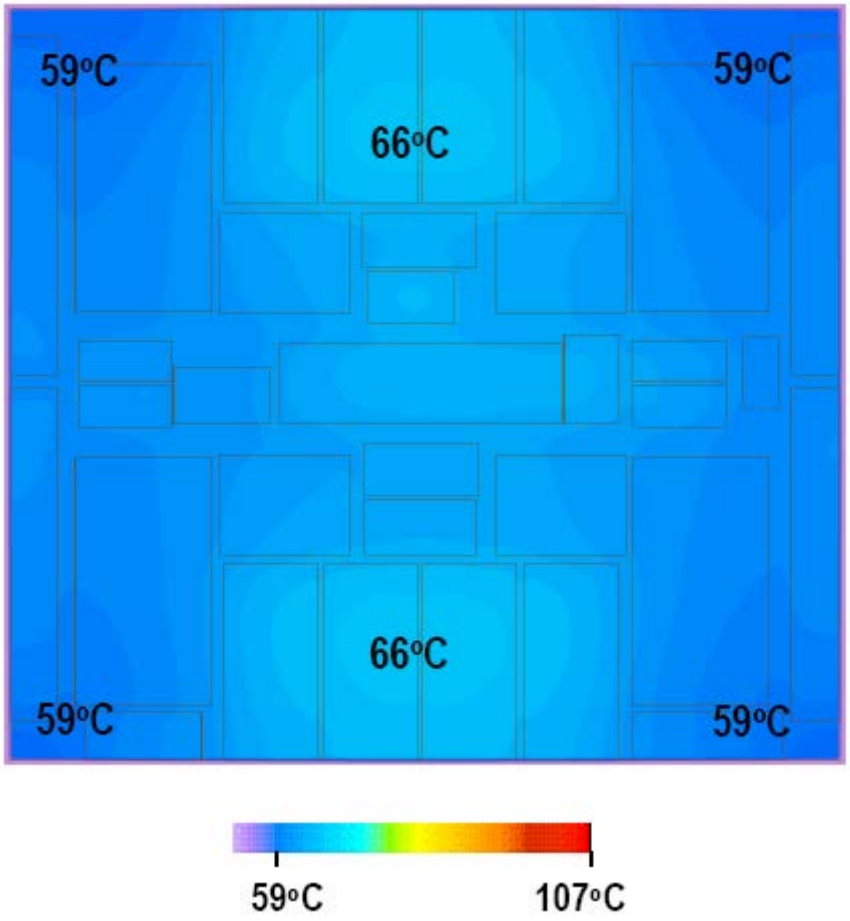
- Shallow 6-stage pipeline
- Fine-grained multithreading

Power Consumption [Source: J. Laudon]



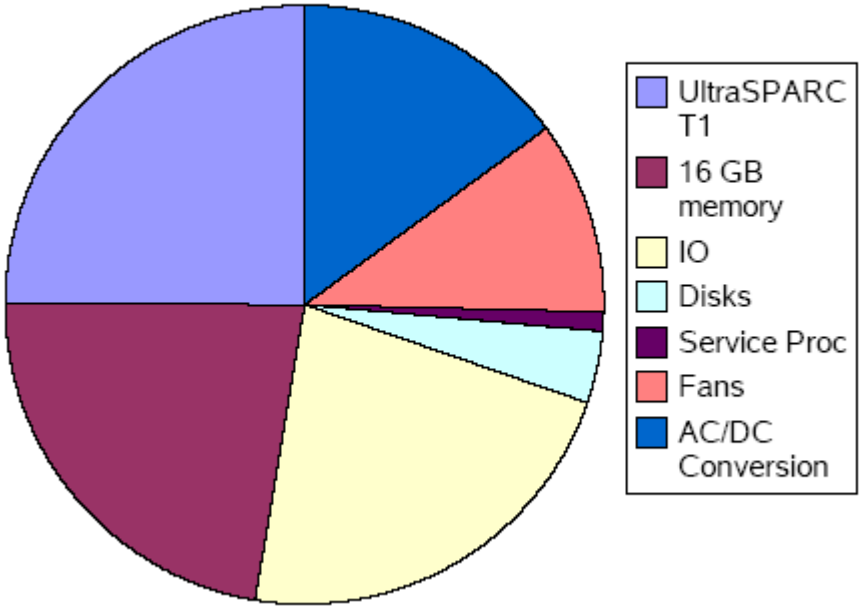
- Fully static design
- Fine granularity clock gating for datapaths (30% flops disabled)
- Lower 1.5 P/N width ratio for library cells
- Interconnect wire classes optimized for power x delay
- SRAM activation control

Thermal Profile



- Low operating temp
- No hot spots
- Improved reliability
- No need for exotic cooling

T2000 System Power



- 271W running SpecJBB2000
- Processor is only 25% of total
- DRAM & I/O next, then conversion losses

Niagara Summary

- Example of *application-specific* system optimization
  - Exploit application behavior (e.g. TLP, cache misses, low ILP)
  - Build very efficient solution
- Downsides
  - Loss of *general-purpose* suitability
  - E.g. poorly suited for software development (parallel make, gcc)
  - Very poor FP performance (fixed in Niagara 2)

Lecture Summary

- Thread-level parallelism
- Synchronization
- Multiprocessors
- Explicit multithreading
- Data parallel architectures
- Multicore interconnects
- Implicit multithreading: Multiscalar
- Niagara case study