Lecture 9
Directory Based Multiprocessors

Slides were used during lectures by
David Patterson, Berkeley, spring 2006

Review

- Caches contain all information on state of cached memory blocks
- Snooping cache over shared medium for smaller MP by invalidating other cached copies on write
- Sharing cached data ⇒ Coherence (values returned by a read), Consistency (when a written value will be returned by a read)

Outline

- Review
- Directory-based protocols and examples
- Synchronization
- Consistency
- Cross Cutting Issues
- Fallacies and Pitfalls
- Sun T1 ("Niagara") Multiprocessor
- Microprocessor Comparison
- Conclusion

A Cache Coherent System Must:

- Provide set of states, state transition diagram, and actions
- Manage coherence protocol
  - (0) Determine when to invoke coherence protocol
  - (a) Find info about state of block in other caches to determine action
    - » whether need to communicate with other cached copies
  - (b) Locate the other copies
  - (c) Communicate with those copies (invalidate/update)
- (0) is done the same way on all systems
  - state of the line is maintained in the cache
  - protocol is invoked if an "access fault" occurs on the line
- Different approaches distinguished by (a) to (c)

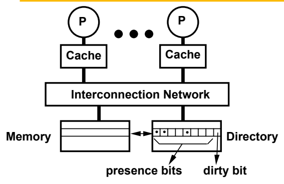
Bus-based Coherence

- All of (a), (b), (c) done through broadcast on bus
  - faulting processor sends out a "search"
  - others respond to the search probe and take necessary action
- Could do it in scalable network too
  - broadcast to all processors, and let them respond
- Conceptually simple, but broadcast doesn't scale with p
  - on bus, bus bandwidth doesn't scale
  - on scalable network, every fault leads to at least p network transactions
- Scalable coherence:
  - can have same cache states and state transition diagram
  - different mechanisms to manage protocol

Scalable Approach: Directories

- Every memory block has associated directory information
  - keeps track of copies of cached blocks and their states
  - on a miss, find directory entry, look it up, and communicate only with the nodes that have copies if necessary
  - in scalable networks, communication with directory and copies is through network transactions
- Many alternatives for organizing directory information

Basic Operation of Directory



- *k* processors
- With each cache-block in memory: *k* presence-bits, *1* dirty-bit
- With each cache-block in cache: *1* valid bit, and *1* dirty (owner) bit

- Read from main memory by processor i:
  - If dirty-bit OFF then { read from main memory; turn p[i] ON; }
  - if dirty-bit ON then { recall line from dirty proc (cache state to shared); update memory; turn dirty-bit OFF; turn p[i] ON; supply recalled data to i;}
- Write to main memory by processor i:
  - If dirty-bit OFF then { supply data to i; send invalidations to all caches that have the block; turn dirty-bit ON; turn p[i] ON; ... }
  - ...

Directory Protocol

- Similar to Snooping Protocol: Three states
  - Shared: ≥ 1 processors have data, memory up-to-date
  - Uncached (no processor has it; not valid in any cache)
  - Exclusive: 1 processor (owner) has data; memory out-of-date
- In addition to cache state, must track which processors have data when in the shared state (usually bit vector, 1 if processor has copy)
- Keep it simple(r):
  - Writes to non-exclusive data ⇒ write miss
  - Processor blocks until access completes
  - Assume messages received and acted upon in order sent

Directory Protocol

- No bus and don't want to broadcast:
  - interconnect no longer single arbitration point
  - all messages have explicit responses
- Terms: typically 3 processors involved
  - Local node where a request originates
  - Home node where the memory location of an address resides
  - Remote node has a copy of a cache block, whether exclusive or shared
- Example messages on next slide: P = processor number, A = address

Directory Protocol Messages (Fig 4.22)

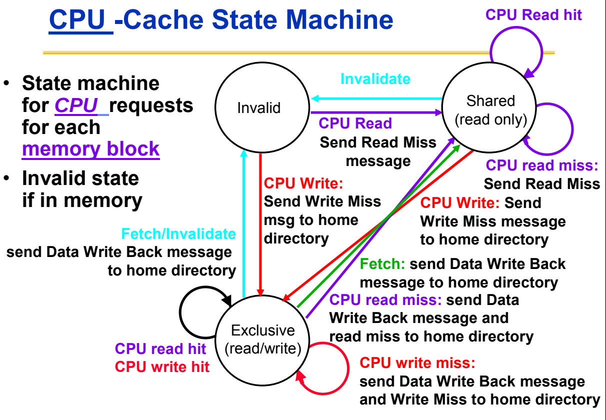
Message type	*Source*	*Destination*	*Msg Content*
Read miss	Local cache	Home directory	P, A
– *Processor P reads data at address A; make P a read sharer and request data*			
Write miss	Local cache	Home directory	P, A
– *Processor P has a write miss at address A; make P the exclusive owner and request data*			
Invalidate	Home directory	Remote caches	A
– *Invalidate a shared copy at address A*			
Fetch	Home directory	Remote cache	A
– *Fetch the block at address A and send it to its home directory; change the state of A in the remote cache to shared*			
Fetch/Invalidate	Home directory	Remote cache	A
– *Fetch the block at address A and send it to its home directory; invalidate the block in the cache*			
Data value reply	Home directory	Local cache	Data
– *Return a data value from the home memory (read miss response)*			
Data write back	Remote cache	Home directory	A, Data
– *Write back a data value for address A (invalidate response)*			

State Transition Diagram for One Cache Block in Directory Based System

- States identical to snooping case; transactions very similar
- Transitions caused by read misses, write misses, invalidates, data fetch requests
- Generates read miss & write miss message to home directory
- Write misses that were broadcast on the bus for snooping ⇒ explicit invalidate & data fetch requests
- Note: on a write, a cache block is bigger, so need to read the full cache block

CPU -Cache State Machine

- State machine for CPU requests for each memory block
- Invalid state if in memory



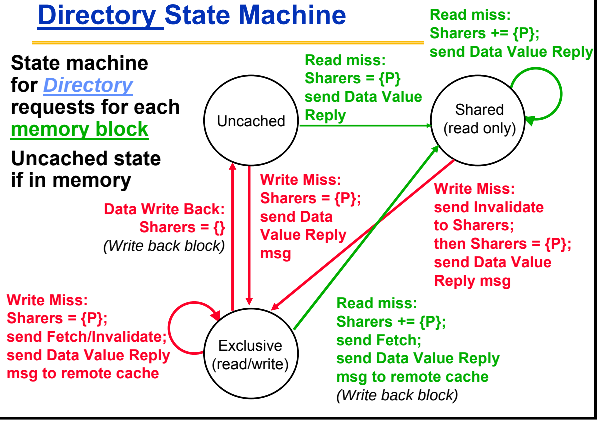
CPU Read hit
Invalid
Invalidate
Shared (read only)
CPU Read Send Read Miss message
CPU read miss: Send Read Miss
CPU Write: Send Write Miss msg to home directory
CPU Write: Send Write Miss message to home directory
Fetch/Invalidate send Data Write Back message to home directory
Fetch: send Data Write Back message to home directory
CPU read miss: send Data Write Back message and read miss to home directory
Exclusive (read/write)
CPU read hit
CPU write hit
CPU write miss: send Data Write Back message and Write Miss to home directory

State Transition Diagram for Directory

- Same states & structure as the transition diagram for an individual cache
- 2 actions: update of directory state & send messages to satisfy requests
- Tracks all copies of memory block
- Also indicates an action that updates the sharing set, Sharers, as well as sending a message

Directory State Machine

- State machine for Directory requests for each memory block
- Uncached state if in memory


Read miss: Sharers += {P}; send Data Value Reply
Read miss: Sharers = {P} send Data Value Reply
Uncached
Shared (read only)
Write Miss: Sharers = {P}; send Data Value Reply msg
Write Miss: send Invalidate to Sharers; then Sharers = {P}; send Data Value Reply msg
Data Write Back: Sharers = {} (Write back block)
Write Miss: Sharers = {P}; send Fetch/Invalidate; send Data Value Reply msg to remote cache
Exclusive (read/write)
Read miss: Sharers += {P}; send Fetch; send Data Value Reply msg to remote cache (Write back block)

Example Directory Protocol

- Message sent to directory causes two actions:
  - Update the directory
  - More messages to satisfy request
- Block is in Uncached state: the copy in memory is the current value; only possible requests for that block are:
  - Read miss: requesting processor sent data from memory &requestor made only sharing node; state of block made Shared.
  - Write miss: requesting processor is sent the value & becomes the Sharing node. The block is made Exclusive to indicate that the only valid copy is cached. Sharers indicates the identity of the owner.
- Block is Shared ⇒ the memory value is up-to-date:
  - Read miss: requesting processor is sent back the data from memory & requesting processor is added to the sharing set.
  - Write miss: requesting processor is sent the value. All processors in the set Sharers are sent invalidate messages, & Sharers is set to identity of requesting processor. The state of the block is made Exclusive.

Example Directory Protocol

- Block is Exclusive: current value of the block is held in the cache of the processor identified by the set Sharers (the owner) ⇒ three possible directory requests:
  - Read miss: owner processor sent data fetch message, causing state of block in owner's cache to transition to Shared and causes owner to send data to directory, where it is written to memory & sent back to requesting processor.
    Identity of requesting processor is added to set Sharers, which still contains the identity of the processor that was the owner (since it still has a readable copy). State is shared.
  - Data write-back: owner processor is replacing the block and hence must write it back, making memory copy up-to-date
    (the home directory essentially becomes the owner), the block is now Uncached, and the Sharer set is empty.
  - Write miss: block has a new owner. A message is sent to old owner causing the cache to send the value of the block to the directory from which it is sent to the requesting processor, which becomes the new owner. Sharers is set to identity of new owner, and state of block is made Exclusive.

Example

Processor 1 Processor 2 Interconnect Directory Memory

	P1			*P2*			*Bus*				*Directory*			*Memor*
step	*State*	*Addr*	*Value*	*State*	*Addr*	*Value*	*Action*	*Proc.*	*Addr*	*Value*	*Addr*	*State*	*{Procs}*	*Value*
P1: Write 10 to A1														
P1: Read A1														
P2: Read A1														
P2: Write 20 to A1														
P2: Write 40 to A2														

A1 and A2 map to the same cache block

Example

Processor 1 Processor 2 Interconnect Directory Memory

	P1			*P2*			*Bus*				*Directory*			*Memory*
step	*State*	*Addr*	*Value*	*State*	*Addr*	*Value*	*Action*	*Proc.*	*Addr*	*Value*	*Addr*	*State*	*{Procs}*	*Value*
P1: Write 10 to A1							*WrMs*	P1	A1		*A1*	*Ex*	*{P1}*	
	Excl.	*A1*	*10*				*DaRp*	P1	A1	0				
P1: Read A1														
P2: Read A1														
P2: Write 20 to A1														
P2: Write 40 to A2														

A1 and A2 map to the same cache block

Example

Processor 1 Processor 2 Interconnect Directory Memory

	P1			P2			Bus				Directory			Memory
step	State	Addr	Value	State	Addr	Value	Action	Proc.	Addr	Value	Addr	State	{Procs}	Value
P1: Write 10 to A1							WrMs	P1	A1		A1	Ex	{P1}	
	Excl.	A1	10				DaRp	P1	A1	0				
P1: Read A1	Excl.	A1	10											
P2: Read A1														
P2: Write 20 to A1														
P2: Write 40 to A2														

A1 and A2 map to the same cache block

Example

Processor 1 Processor 2 Interconnect Directory Memory

	P1			P2			Bus				Directory			Memory
step	State	Addr	Value	State	Addr	Value	Action	Proc.	Addr	Value	Addr	State	{Procs}	Value
P1: Write 10 to A1							WrMs	P1	A1		A1	Ex	{P1}	
	Excl.	A1	10				DaRp	P1	A1	0				
P1: Read A1	Excl.	A1	10											
P2: Read A1				Shar.	A1		RdMs	P2	A1					
	Shar.	A1	10				Ftch	P1	A1	10	A1			10
				Shar.	A1	10	DaRp	P2	A1	10	A1	Shar.	P1,P2}	10
P2: Write 20 to A1														
P2: Write 40 to A2														

Write Back

A1 and A2 map to the same cache block

Example

Processor 1 Processor 2 Interconnect Directory Memory

	P1			P2			Bus				Directory			Memory
step	State	Addr	Value	State	Addr	Value	Action	Proc.	Addr	Value	Addr	State	{Procs}	Value
P1: Write 10 to A1							WrMs	P1	A1		A1	Ex	{P1}	
	Excl.	A1	10				DaRp	P1	A1	0				
P1: Read A1	Excl.	A1	10											
P2: Read A1				Shar.	A1		RdMs	P2	A1					
	Shar.	A1	10				Ftch	P1	A1	10	A1			10
				Shar.	A1	10	DaRp	P2	A1	10	A1	Shar.	P1,P2}	10
P2: Write 20 to A1				Excl.	A1	20	WrMs	P2	A1					10
	Inv.						Inval.	P1	A1		A1	Excl.	{P2}	10
P2: Write 40 to A2														

A1 and A2 map to the same cache block

Example

Processor 1 Processor 2 Interconnect Directory Memory

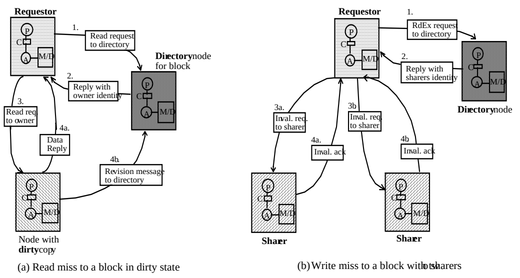
	P1			P2			Bus				Directory			Memory
step	State	Addr	Value	State	Addr	Value	Action	Proc.	Addr	Value	Addr	State	{Procs}	Value
P1: Write 10 to A1							WrMs	P1	A1		A1	Ex	{P1}	
	Excl.	A1	10				DaRp	P1	A1	0				
P1: Read A1	Excl.	A1	10											
P2: Read A1				Shar.	A1		RdMs	P2	A1					
	Shar.	A1	10				Ftch	P1	A1	10	A1			10
				Shar.	A1	10	DaRp	P2	A1	10	A1	Shar.	P1,P2}	10
P2: Write 20 to A1				Excl.	A1	20	WrMs	P2	A1					10
	Inv.						Inval.	P1	A1		A1	Excl.	{P2}	10
P2: Write 40 to A2							WrMs	P2	A2		A2	Excl.	{P2}	0
							WrBk	P2	A1	20	A1	Unca.	{}	20
				Excl.	A2	40	DaRp	P2	A2	0	A2	Excl.	{P2}	0

A1 and A2 map to the same cache block
(but different memory block addresses A1 ≠ A2)

Implementing a Directory

- We assume operations atomic, but they are not; reality is much harder; must avoid deadlock when run out of buffers in network (see Appendix E)
- Optimizations:
  - read miss or write miss is Exclusive: send data directly to requestor from owner vs. first to memory and then from memory to requestor

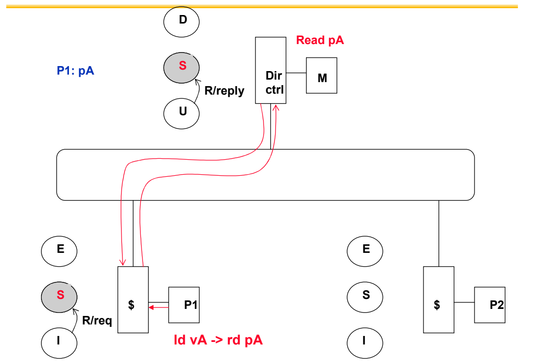
Basic Directory Transactions



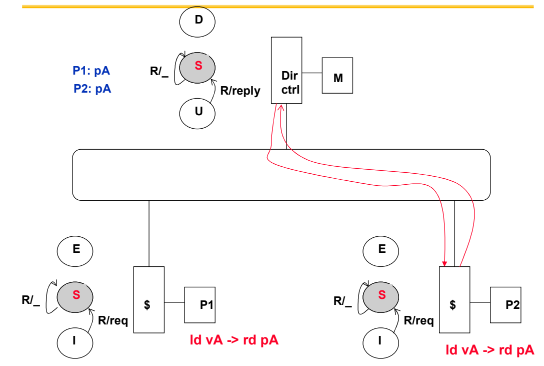
(a) Read miss to a block in dirty state

(b) Write miss to a block with two sharers

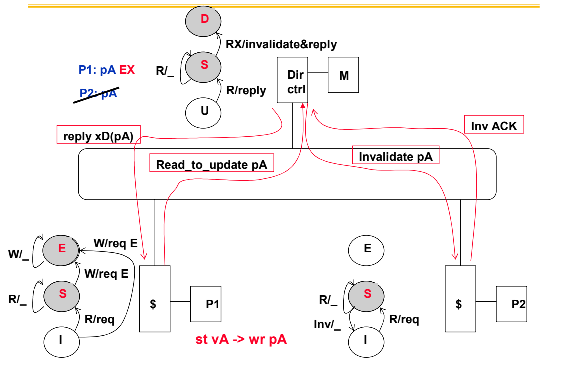
Example Directory Protocol (1st Read)



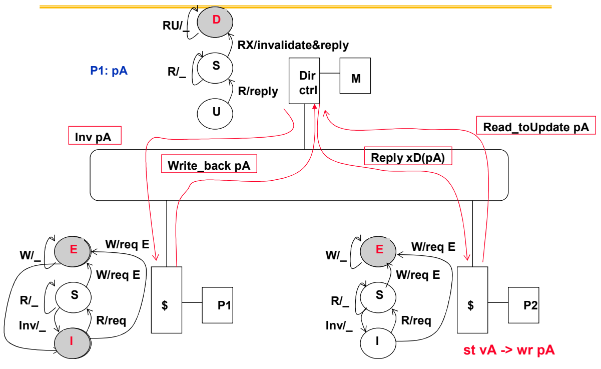
Example Directory Protocol (Read Share)



Example Directory Protocol (Wr to shared)



Example Directory Protocol (Wr to Ex)



A Popular Middle Ground

- Two-level "hierarchy"
- Individual nodes are multiprocessors, connected non-hierarchically
  - e.g. mesh of SMPs
- Coherence across nodes is directory-based
  - directory keeps track of nodes, not individual processors
- Coherence within nodes is snooping or directory
  - orthogonal, but needs a good interface of functionality
- SMP on a chip directory + snoop?

Synchronization

- Why Synchronize? Need to know when it is safe for different processes to use shared data
- Issues for Synchronization:
  - Uninterruptable instruction to fetch and update memory (atomic operation);
  - User level synchronization operation using this primitive;
  - For large scale MPs, synchronization can be a bottleneck; techniques to reduce contention and latency of synchronization

Uninterruptable Instruction to Fetch and Update Memory

- Atomic exchange: interchange a value in a register for a value in memory
  - 0 ⇒ synchronization variable is free
  - 1 ⇒ synchronization variable is locked and unavailable
  - – Set register to 1 & swap
  - – New value in register determines success in getting lock
    0 if you succeeded in setting the lock (you were first)
    1 if other processor had already claimed access
  - – Key is that exchange operation is indivisible
- Test-and-set: tests a value and sets it if the value passes the test
- Fetch-and-increment: it returns the value of a memory location and atomically increments it
  - – 0 ⇒ synchronization variable is free

Uninterruptable Instruction to Fetch and Update Memory

- Hard to have read & write in 1 instruction: use 2 instead
- Load linked (or load locked) + store conditional
  - – Load linked returns the initial value
  - – Store conditional returns 1 if it succeeds (no other store to same memory location since preceding load) and 0 otherwise
- Example doing atomic swap with LL & SC:

```
try:  mov   R3,R4        ; mov exchange value
      ll    R2,0(R1)     ; load linked
      sc    R3,0(R1)     ; store conditional
      beqz  R3,try       ; branch store fails (R3 = 0)
      mov   R4,R2        ; put load value in R4
```

- Example doing fetch & increment with LL & SC:

```
try:  ll    R2,0(R1)     ; load linked
      addi  R2,R2,#1     ; increment (OK if reg–reg)
      sc    R2,0(R1)     ; store conditional
      beqz  R2,try       ; branch store fails (R2 = 0)
```

User Level Synchronization—Operation Using this Primitive

- Spin locks: processor continuously tries to acquire, spinning around a loop trying to get the lock

```
             li      R2,#1
lockit:      exch    R2,0(R1)      ;atomic exchange
             bnez    R2,lockit     ;already locked?
```

- What about MP with cache coherency?
  - – Want to spin on cache copy to avoid full memory latency
  - – Likely to get cache hits for such variables
- Problem: exchange includes a write, which invalidates all other copies; this generates considerable bus traffic
- Solution: start by simply repeatedly reading the variable; when it changes, then try exchange ("test and test&set"):

```
try:         li      R2,#1
lockit:      lw      R3,0(R1)      ;load var
             bnez    R3,lockit     ;≠ 0 ⇒ not free ⇒ spin
             exch    R2,0(R1)      ;atomic exchange
             bnez    R2,try        ;already locked?
```

Another MP Issue: Memory Consistency Models

- What is consistency? When must a processor see the new value? e.g., seems that

```
P1:   A = 0;                 P2:   B = 0;
      .....                        .....
      A = 1;                       B = 1;
L1:   if (B == 0) ...        L2:   if (A == 0) ...
```

- Impossible for both if statements L1 & L2 to be true?
  - – What if write invalidate is delayed & processor continues?
- Memory consistency models: what are the rules for such cases?
- Sequential consistency: result of any execution is the same as if the accesses of each processor were kept in order and the accesses among different processors were interleaved ⇒ assignments before ifs above
  - – SC: delay all memory accesses until all invalidates done

Memory Consistency Model

- Schemes faster execution to sequential consistency
- Not an issue for most programs; they are synchronized
  - – A program is synchronized if all access to shared data are ordered by synchronization operations

    write (x)
    ...
    release (s) *{unlock}*
    ...
    acquire (s) *{lock}*
    ...
    read(x)
- Only those programs willing to be nondeterministic are not synchronized: "data race": outcome f(proc. speed)
- Several Relaxed Models for Memory Consistency since most programs are synchronized; characterized by their attitude towards: RAR, WAR, RAW, WAW to different addresses

Relaxed Consistency Models: The Basics

- Key idea: allow reads and writes to complete out of order, but to use synchronization operations to enforce ordering, so that a synchronized program behaves as if the processor were sequentially consistent
  - – By relaxing orderings, may obtain performance advantages
  - – Also specifies range of legal compiler optimizations on shared data
  - – Unless synchronization points are clearly defined and programs are synchronized, compiler could not interchange read and write of 2 shared data items because might affect the semantics of the program
- 3 major sets of relaxed orderings:
  1. W→R ordering (all writes completed before next read)
     - Because retains ordering among writes, many programs that operate under sequential consistency operate under this model, without additional synchronization. Called processor consistency
  2. W → W ordering (all writes completed before next write)
  3. R → W and R → R orderings, a variety of models depending on ordering restrictions and how synchronization operations enforce ordering
- Many complexities in relaxed consistency models; defining precisely what it means for a write to complete; deciding when processors can see values that it has written

Mark Hill observation

Instead, use speculation to hide latency from strict consistency model

- If processor receives invalidation for memory reference before it is committed, processor uses speculation recovery to back out computation and restart with invalidated memory reference

1. Aggressive implementation of sequential consistency or processor consistency gains most of advantage of more relaxed models
2. Implementation adds little to implementation cost of speculative processor
3. Allows the programmer to reason using the simpler programming models

Cross Cutting Issues: Performance Measurement of Parallel Processors

- Performance: how well scale as increase Proc
- Speedup fixed as well as scaleup of problem
  - Assume benchmark of size n on p processors makes sense: how scale benchmark to run on m * p processors?
  - Memory-constrained scaling: keeping the amount of memory used per processor constant
  - Time-constrained scaling: keeping total execution time, assuming perfect speedup, constant
- Example: 1 hour on 10 P, time ~ O(n^3), 100 P?
  - Time-constrained scaling: 1 hour $\Rightarrow 10^{1/3}n \Rightarrow$ 2.15n scale up
  - Memory-constrained scaling: 10n size $\Rightarrow 10^3/10 \Rightarrow$ 100X or 100 hours! 10X processors for 100X longer???
  - Need to know application well to scale: # iterations, error tolerance

Fallacy: Amdahl's Law doesn't apply to parallel computers

- Since some part linear, can't go 100X?
- 1987 claim to break it, since 1000X speedup
  - researchers scaled the benchmark to have a data set size that is 1000 times larger and compared the uniprocessor and parallel execution times of the scaled benchmark. For this particular algorithm the sequential portion of the program was constant independent of the size of the input, and the rest was fully parallel—hence, linear speedup with 1000 processors
- Usually sequential scale with data too

Fallacy: Linear speedups are needed to make multiprocessors cost-effective

Mark Hill & David Wood 1995 study

- Compare costs SGI uniprocessor and MP
  - Uniprocessor = \$38,400 + \$100 * MB
  - MP = \$81,600 + \$20,000 * P + \$100 * MB
- 1 GB: uni = \$138k v. mp = \$181k + \$20k * P
- What speedup for better MP cost performance?
  - 8 proc = \$341k; \$341k/138k $\Rightarrow$ 2.5X
  - 16 proc $\Rightarrow$ need only 3.6X, or 25% linear speedup
- Even if need some more memory for MP, not linear

Fallacy: Scalability is almost free

- "build scalability into a multiprocessor and then simply offer the multiprocessor at any point on the scale from a small number of processors to a large number"
- Cray T3E scales to 2048 CPUs vs. 4 CPU Alpha
  - At 128 CPUs, it delivers a peak bisection BW of 38.4 GB/s, or 300 MB/s per CPU (uses Alpha microprocessor)
  - Compaq Alphaserver ES40 up to 4 CPUs and has 5.6 GB/s of interconnect BW, or 1400 MB/s per CPU
- Build apps that scale requires significantly more attention to load balance, locality, potential contention, and serial (or partly parallel) portions of program. 10X is very hard

Pitfall: Not developing SW to take advantage (or optimize for) multiprocessor architecture

- SGI OS protects the page table data structure with a single lock, assuming that page allocation is infrequent
- Suppose a program uses a large number of pages that are initialized at start-up
- Program parallelized so that multiple processes allocate the pages
- But page allocation requires lock of page table data structure, so even an OS kernel that allows multiple threads will be serialized at initialization (even if separate processes)

Answers to 1995 Questions about Parallelism

In the 1995 edition of this text, we concluded the chapter with a discussion of two then current controversial issues.

1. What architecture would very large scale, microprocessor-based multiprocessors use?
2. What was the role for multiprocessing in the future of microprocessor architecture?

Answer 1. Large scale multiprocessors did not become a major and growing market ⇒ clusters of single microprocessors or moderate SMPs

Answer 2. Astonishingly clear. For at least for the next 5 years, future MPU performance comes from the exploitation of TLP through multicore processors vs. exploiting more ILP

Cautionary Tale

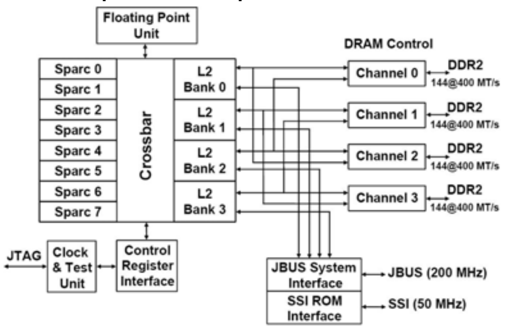
- Key to success of birth and development of ILP in 1980s and 1990s was software in the form of optimizing compilers that could exploit ILP
- Similarly, successful exploitation of TLP will depend as much on the development of suitable software systems as it will on the contributions of computer architects
- Given the slow progress on parallel software in the past 30+ years, it is likely that exploiting TLP broadly will remain challenging for years to come

T1 (“Niagara”)

- Target: Commercial server applications
  - High thread level parallelism (TLP)
    - » Large numbers of parallel client requests
  - Low instruction level parallelism (ILP)
    - » High cache miss rates
    - » Many unpredictable branches
    - » Frequent load-load dependencies
- Power, cooling, and space are major concerns for data centers
- Metric: Performance/Watt/Sq. Ft.
- Approach: Multicore, Fine-grain multithreading, Simple pipeline, Small L1 caches, Shared L2

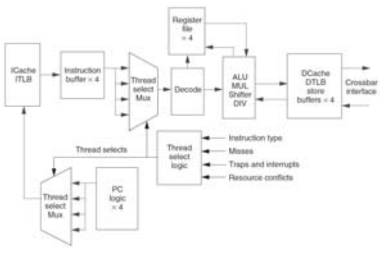
T1 Architecture

- Also ships with 6 or 4 processors



T1 pipeline

- Single issue, in-order, 6-deep pipeline: F, S, D, E, M, W
- 3 clock delays for loads & branches.
- Shared units:
  - L1 $, L2 $
  - TLB
  - X units
  - pipe registers
- Hazards:
  - Data
  - Structural



T1 Fine-Grained Multithreading

- Each core supports four threads and has its own level one caches (16KB for instructions and 8 KB for data)
- Switching to a new thread on each clock cycle
- Idle threads are bypassed in the scheduling
  - Waiting due to a pipeline delay or cache miss
  - Processor is idle only when all 4 threads are idle or stalled
- Both loads and branches incur a 3 cycle delay that can only be hidden by other threads
- A single set of floating point functional units is shared by all 8 cores
  - floating point performance was not a focus for T1

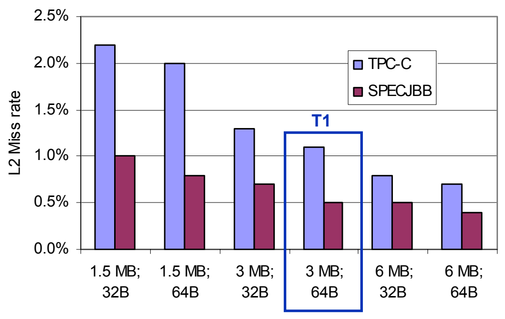
Memory, Clock, Power

- 16 KB 4 way set assoc. I$/ core
- 8 KB 4 way set assoc. D$/ core
- 3MB 12 way set assoc. L2 $ shared
  - 4 x 750KB independent banks
  - crossbar switch to connect
  - 2 cycle throughput, 8 cycle latency
  - Direct link to DRAM & Jbus
  - Manages cache coherence for the 8 cores
  - CAM based directory

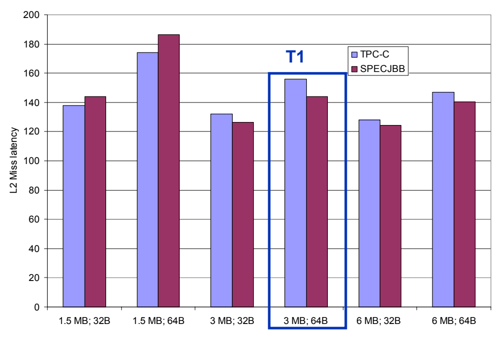
Write through
- allocate LD
- no-allocate ST

- Coherency is enforced among the L1 caches by a directory associated with each L2 cache block
- Used to track which L1 caches have copies of an L2 block
- By associating each L2 with a particular memory bank and enforcing the subset property, T1 can place the directory at L2 rather than at the memory, which reduces the directory overhead
- L1 data cache is write-through, only invalidation messages are required; the data can always be retrieved from the L2 cache
- 1.2 GHz at ≈72W typical, 79W peak power consumption

Miss Rates: L2 Cache Size, Block Size



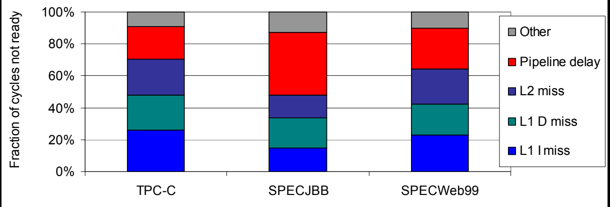
Miss Latency: L2 Cache Size, Block Size



CPI Breakdown of Performance

Benchmark	Per Thread CPI	Per core CPI	Effective CPI for 8 cores	Effective IPC for 8 cores
TPC-C	7.20	1.80	0.23	4.4
SPECJBB	5.60	1.40	0.18	5.7
SPECWeb99	6.60	1.65	0.21	4.8

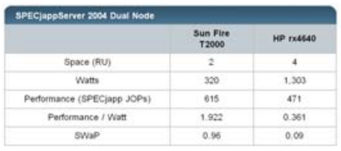
Not Ready Breakdown



- TPC-C - store buffer full is largest contributor
- SPEC-JBB - atomic instructions are largest contributor
- SPECWeb99 - both factors contribute

Performance: Benchmarks + Sun Marketing

Benchmark\Architecture	Sun Fire T2000	IBM p5-550 with 2 dual-core Power5 chips	Dell PowerEdge
SPECjbb2005 (Java server software) business operations/ sec	63,378	61,789	24,208 (SC1425 with dual single-core Xeon)
SPECweb2005 (Web server performance)	14,001	7,881	4,850 (2850 with two dual-core Xeon processors)
NotesBench (Lotus Notes performance)	16,061	14,740	



SPECjappServer 2004 Dual Node

	Sun Fire T2000	HP rx4640
Space (RU)	2	4
Watts	320	1,303
Performance (SPECjapp JOPs)	615	471
Performance / Watt	1.922	0.361
SWaP	0.96	0.09

Space, Watts, and Performance

HP marketing view of T1 Niagara

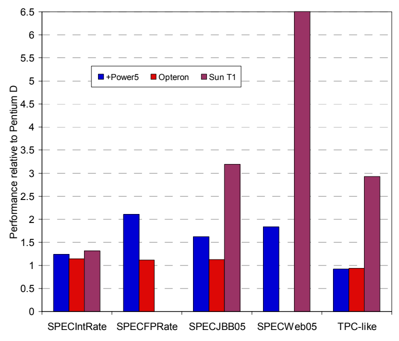
1. **Sun's radical UltraSPARC T1 chip is made up of individual cores that have much slower single thread performance when compared to the higher performing cores of the Intel Xeon, Itanium, AMD Opteron or even classic UltraSPARC processors.**
2. **The Sun Fire T2000 has poor floating-point performance, by Sun's own admission.**
3. **The Sun Fire T2000 does not support commerical Linux or Windows® and requires a lock-in to Sun and Solaris.**
4. **The UltraSPARC T1, aka CoolThreads, is new and unproven, having just been introduced in December 2005.**
5. **In January 2006, a well-known financial analyst downgraded Sun on concerns over the UltraSPARC T1's limitation to only the Solaris operating system, unique requirements, and longer adoption cycle, among other things. [10]**

- **Where is the compelling value to warrant taking such a risk?**
- **http://h71028.www7.hp.com/ERC/cache/280124-0-0-0-121.html**

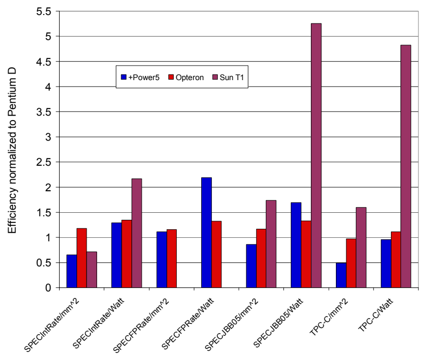
Microprocessor Comparison

Processor	SUN T1	Opteron	Pentium D	IBM Power 5
Cores	**8**	2	2	2
Instruction issues / clock / core	1	3	3	4
Peak instr. issues / chip	**8**	6	6	**8**
Multithreading	Fine-grained	No	SMT	SMT
L1 I/D in KB per core	16/8	**64/64**	12K uops/16	64/32
L2 per core/shared	**3 MB** shared	1MB / core	1MB/ core	1.9 MB shared
Clock rate (GHz)	1.2	2.4	**3.2**	1.9
Transistor count (M)	**300**	233	230	276
Die size (mm^2)	379	199	206	**389**
Power (W)	**79**	110	130	125

Performance Relative to Pentium D



Performance/mm^2, Performance/Watt



Niagara 2

- **Improve performance by increasing threads supported per chip from 32 to 64**
  - 8 cores * 8 threads per core
- **Floating-point unit for each core, not for each chip**
- **Hardware support for encryption standards EAS, 3DES, and elliptical-curve cryptography**
- **Niagara 2 will add a number of 8x PCI Express interfaces directly into the chip in addition to integrated 10Gigabit Ethernet XAU interfaces and Gigabit Ethernet ports.**
- **Integrated memory controllers will shift support from DDR2 to FB-DIMMs and double the maximum amount of system memory.**

Kevin Krewell
"Sun's Niagara Begins CMT Flood -
The Sun UltraSPARC T1 Processor Released"
Microprocessor Report, January 3, 2006

And in Conclusion …

- **Caches contain all information on state of cached memory blocks**
- **Snooping cache over shared medium for smaller MP by invalidating other cached copies on write**
- **Sharing cached data ⇒ Coherence (values returned by a read), Consistency (when a written value will be returned by a read)**
- **Snooping and Directory Protocols similar; bus makes snooping easier because of broadcast (snooping ⇒ uniform memory access)**
- **Directory has extra data structure to keep track of state of all cache blocks**
- **Distributing directory**
  ⇒ scalable shared address multiprocessor
  ⇒ Cache coherent, Non uniform memory access

Reading

- **This lecture:**
  - **chapter 4: 4.4-4.10** ***rest of Multiprocessors and TLP***
- **Next lecture:**
  - **chapter 5:** ***Advanced Memory Hierarchy***