CS 211 Computer Architecture

ILP to Multiprocessing

Review of Concept of ILP Processors

- Interest in multiple-issue because wanted to improve performance without affecting uniprocessor programming model
- Taking advantage of ILP is conceptually simple, but design problems are amazingly complex in practice
- Processors of last 5 years (Pentium 4, IBM Power 5, AMD Opteron) have the same basic structure and similar sustained issue rates (3 to 4 instructions per clock) as the 1st dynamically scheduled, multiple-issue processors announced in 1995
  - Clocks 10 to 20X faster, caches 4 to 8X bigger, 2 to 4X as many renaming registers, and 2X as many load-store units ⇒ performance 8 to 16X
- Peak v. delivered performance gap increasing

2

Next…

- Quick review of Limits to ILP: Chapter 3
- Thread Level Parallelism: Chapter 3
  - Multithreading
  - Simultaneous Multithreading
- Multiprocessing: Chapter 4
  - Fundamentals of multiprocessors
    - » Synchronization, memory, …
  - Chip level multiprocessing: Multi-Core

3

Limits to ILP

- Conflicting studies of ILP amount
  - Benchmarks
  - Hardware sophistication
  - Compiler sophistication
- How much ILP is available using existing mechanisms with increasing HW budgets?
- Do we need to invent new HW/SW mechanisms to keep on processor performance curve?
  - Intel MMX, SSE (Streaming SIMD Extensions): 64 bit ints
  - Intel SSE2: 128 bit, including 2 64-bit Fl. Pt. per clock
  - Motorola AltaVec: 128 bit ints and FPs
  - Supersparc Multimedia ops, etc.

4

Overcoming Limits

- Advances in compiler technology + significantly new and different hardware techniques *may* be able to overcome limitations assumed in studies
- However, unlikely such advances when coupled *with realistic hardware* will overcome these limits in near future

5

Limits to ILP

Assumptions for *ideal*/perfect machine to start:

1. *Register renaming* – infinite virtual registers
   => all register WAW & WAR hazards are avoided
2. *Branch prediction* – perfect; no mispredictions
3. *Jump prediction* – all jumps perfectly predicted (returns, case statements)
   2 & 3 ⇒ no control dependencies; perfect speculation & an unbounded buffer of instructions available
4. *Memory-address alias analysis* – addresses known & a load can be moved before a store provided addresses not equal; 1&4 eliminates all but RAW
5. Window size is infinite

6

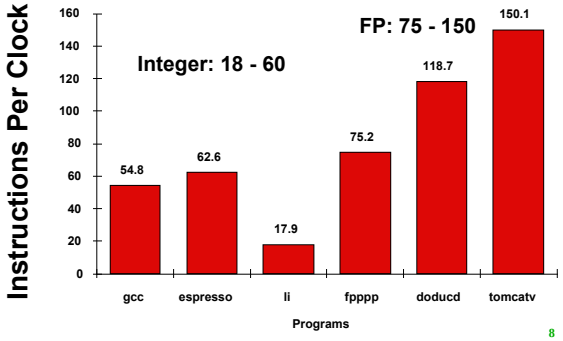
Some numbers…

- Initial MIPS HW Model here; MIPS compilers.
- perfect caches; 1 cycle latency for all instructions (FP *,/); unlimited instructions issued/clock cycle

7

Upper Limit to ILP: Ideal Machine

(Figure 3.1)



8

ILP Limitations: In Reality – Architecture "Parameters"

- **Window size**
  - Large window size could lead to more ILP, but more time to examine instruction packet to determine parallelism
- **Register file size**
  - Finite size of register file (virtual and physical) introduces more name dependencies
- **Branch prediction**
  - Effects of realistic branch predictor
- **Memory aliasing**
  - Idealized model assumes we can analyze all memory dependencies: but compile time analysis cannot be perfect
  - Realistic:
    - » Global/stack: assume perfect predictions for global and stack data but all heap references will conflict
    - » Inspection: deterministic compile time references
      - R1(10) and R1(100) cannot conflict if R1 has not changed between the two
    - » None: all memory accesses are assumed to conflict

9

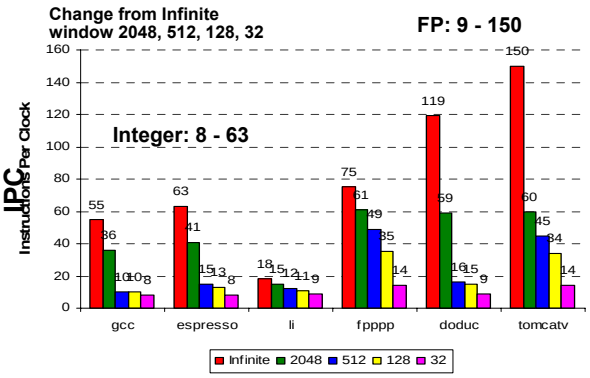
ILP Limitations: Architecture"Parameters"

- **Window size**
  - Large window size could lead to more ILP, but more time to examine instruction packet to determine parallelism

10

More Realistic HW: Window Impact

Figure 3.2



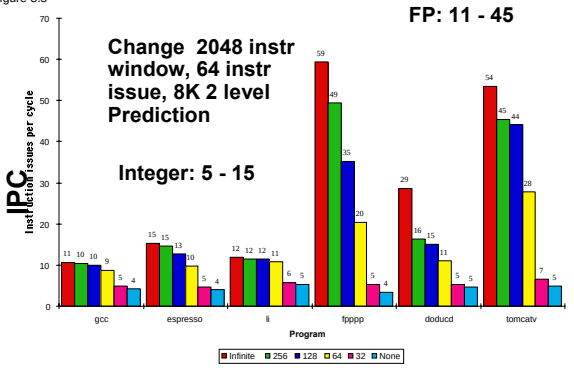
ILP Limitations: Architecture"Parameters"

- **Register file size**
  - Finite size of register file (virtual and physical) introduces more name dependencies

12

More Realistic HW: Renaming Register Impact (N int + N fp +64)

Figure 3.5



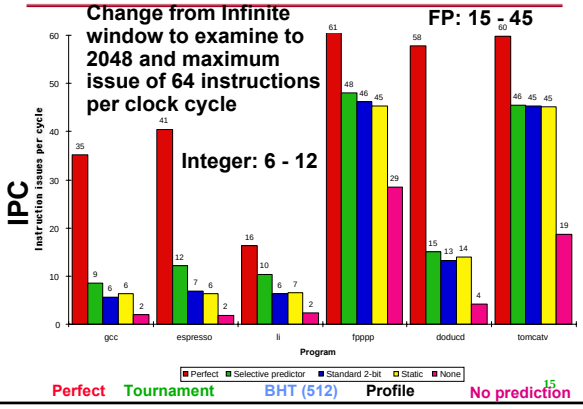
Infinite 256 128 64 32 None

ILP Limitations: Architecture"Parameters"

- Branch prediction
  - Effects of realistic branch predictor

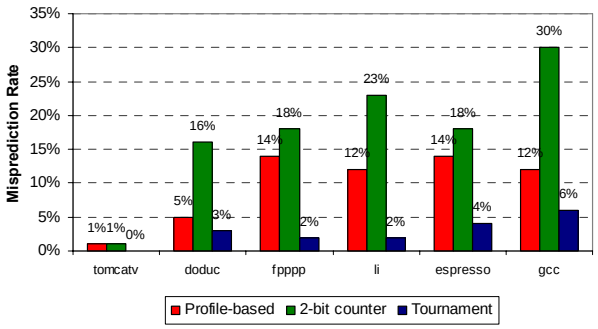
More Realistic HW: Branch Impact

Figure 3.3



Perfect Tournament BHT (512) Profile No prediction

Misprediction Rates



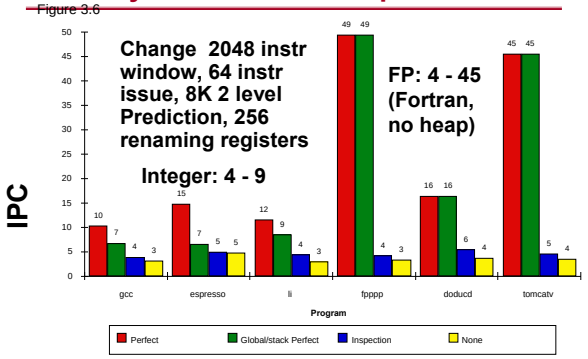
ILP Limitations: Architecture"Parameters"

- **Memory aliasing**
  - Idealized model assumes we can analyze all memory dependencies: compile time analysis cannot be perfect
  - Realistic:
    - » Global/stack: assume perfect predictions for global and stack data but all heap references will conflict
    - » Inspection: deterministic compile time references
      - R1(10) and R1(100) cannot conflict if R1 has not changed between the two
    - » None: all memory accesses are assumed to conflict

17

More Realistic HW: Memory Address Alias Impact

Figure 3.6



Perfect | Global/Stack perf; heap conflicts | Inspec. Assem. | None

18

How to Exceed ILP Limits of this study?

- These are not laws of physics; just practical limits for today, and perhaps overcome via research
- Compiler and ISA advances could change results
- WAR and WAW hazards through memory: eliminated WAW and WAR hazards through register renaming, but not in memory usage
  - Can get conflicts via allocation of stack frames as a called procedure reuses the memory addresses of a previous frame on the stack

19

HW v. SW to increase ILP

- Memory disambiguation: HW best
- Speculation:
  - HW best when dynamic branch prediction better than compile time prediction
  - Exceptions easier for HW
  - HW doesn't need bookkeeping code or compensation code
  - Very complicated to get right
- Scheduling: SW can look ahead to schedule better
- Compiler independence: does not require new compiler, recompilation to run well

20

Next….Thread Level Parallelism

21

Performance beyond single thread ILP

- There can be much higher natural parallelism in some applications (e.g., Database or Scientific codes)
- Explicit Thread Level Parallelism or Data Level Parallelism
- Thread: process with own instructions and data
  - thread may be a process part of a parallel program of multiple processes, or it may be an independent program
  - *Each thread has all the state* (instructions, data, PC, register state, and so on) necessary to allow it to execute
- Data Level Parallelism: Perform identical operations on data, and lots of data
  - Example: Vector operations, matrix computations

22

Thread Level Parallelism (TLP)

- ILP exploits implicit parallel operations within a loop or straight-line code segment
- TLP explicitly represented by the use of multiple threads of execution that are inherently parallel
- Goal: Use multiple instruction streams to improve
  1. Throughput of computers that run many programs
  2. Execution time of multi-threaded programs
- TLP could be more cost-effective to exploit than ILP

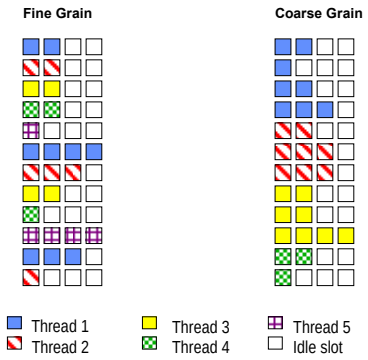
23

New Approach: Mulithreaded Execution

- Multithreading: multiple threads to share the functional units of 1 processor via overlapping
  - *processor must duplicate independent state of each thread* e.g., a separate copy of register file, a separate PC, and for running independent programs, a separate page table
  - memory shared through the virtual memory mechanisms, which already support multiple processes
  - *HW for fast thread switch*; much faster than full process switch ≈ 100s to 1000s of clocks
- When switch?
  - Fine Grain: Alternate instruction per thread
  - Coarse grain: When a thread is stalled, perhaps for a cache miss, another thread can be executed

24

Multithreaded Processing



25

Fine-Grained Multithreading

- Switches between threads on each instruction, causing the execution of multiples threads to be interleaved
  - Usually done in a round-robin fashion, skipping any stalled threads
- CPU must be able to switch threads every clock
- Advantage:
  - can hide both short and long stalls, since instructions from other threads executed when one thread stalls
- Disadvantage:
  - slows down execution of individual threads, since a thread ready to execute without stalls will be delayed by instructions from other threads
- Used on Sun's Niagara (see textbook)

26

Course-Grained Multithreading

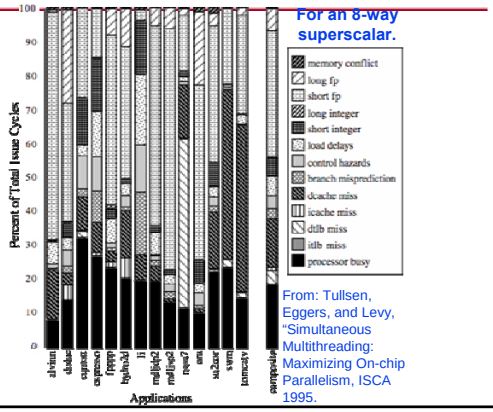
- Switches threads only on costly stalls, such as L2 cache misses
- Advantages
  - Relieves need to have very fast thread-switching
  - Doesn't slow down thread, since instructions from other threads issued only when the thread encounters a costly stall
- Disadvantage is hard to overcome throughput losses from shorter stalls, due to pipeline start-up costs
  - Since CPU issues instructions from 1 thread, when a stall occurs, the pipeline must be emptied or frozen
  - New thread must fill pipeline before instructions can complete
- Because of this start-up overhead, coarse-grained multithreading is better for reducing penalty of high cost stalls, where pipeline refill << stall time
- Used in IBM AS/400

27

ILP and TLP…

28

For most apps, most execution units lie idle



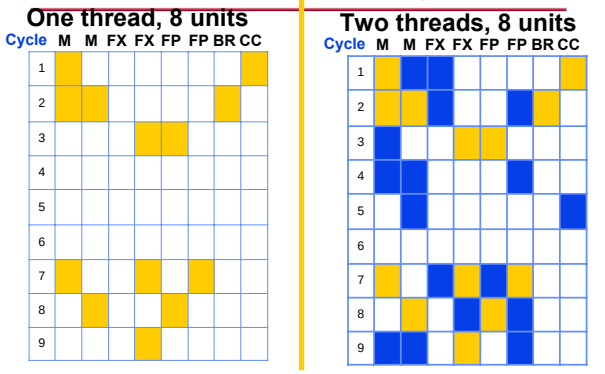
From: Tullsen, Eggers, and Levy, "Simultaneous Multithreading: Maximizing On-chip Parallelism, ISCA 1995.

Do both ILP and TLP?

- TLP and ILP exploit two different kinds of parallel structure in a program
- Could a processor oriented at ILP to exploit TLP?
  - functional units are often idle in data path designed for ILP because of either stalls or dependences in the code
- Could the TLP be used as a source of independent instructions that might keep the processor busy during stalls?
- Could TLP be used to employ the functional units that would otherwise lie idle when insufficient ILP exists?

30

Simultaneous Multi-threading ...



M = Load/Store, FX = Fixed Point, FP = Floating Point, BR = Branch, CC = Condition Codes

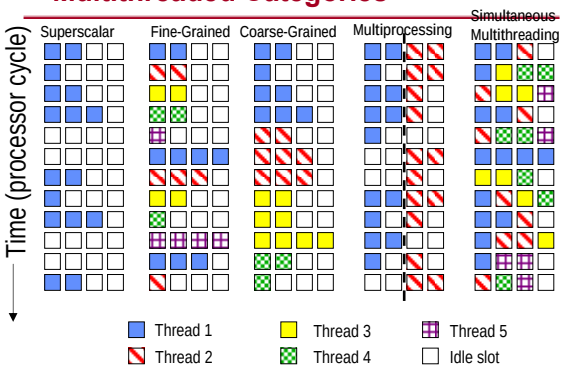
Simultaneous Multithreading (SMT)

- Simultaneous multithreading (SMT): insight that dynamically scheduled processor already has many HW mechanisms to support multithreading
  - Large set of virtual registers that can be used to hold the register sets of independent threads
  - Register renaming provides unique register identifiers, so instructions from multiple threads can be mixed in datapath without confusing sources and destinations across threads
  - Out-of-order completion allows the threads to execute out of order, and get better utilization of the HW
- Just adding a per thread renaming table and keeping separate PCs
  - Independent commitment can be supported by logically keeping a separate reorder buffer for each thread

Source: Micrprocessor Report, December 6, 1999 "Compaq Chooses SMT for Alpha"

32

Multithreaded Categories



33

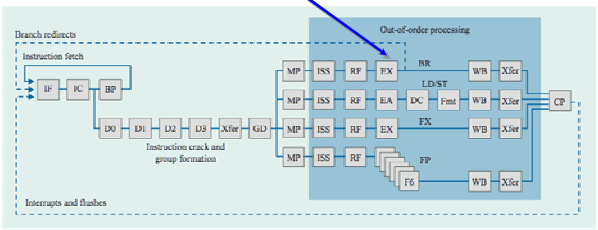
Design Challenges in SMT

- Since SMT makes sense only with fine-grained implementation, impact of fine-grained scheduling on single thread performance?
  - A preferred thread approach sacrifices neither throughput nor single-thread performance?
  - Unfortunately, with a preferred thread, the processor is likely to sacrifice some throughput, when preferred thread stalls
- Larger register file needed to hold multiple contexts
- Not affecting clock cycle time, especially in
  - Instruction issue - more candidate instructions need to be considered
  - Instruction completion - choosing which instructions to commit may be challenging
- Ensuring that cache and TLB conflicts generated by SMT do not degrade performance

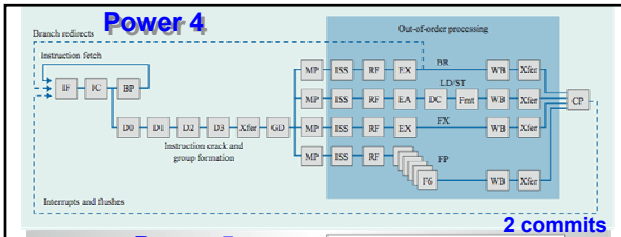
34

Power 4

Single-threaded predecessor to Power 5. 8 execution units in out-of-order engine, each may issue an instruction each cycle.



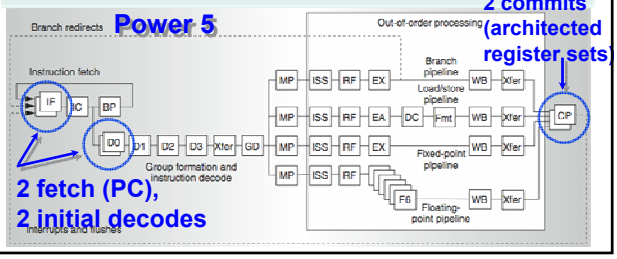
Power 4



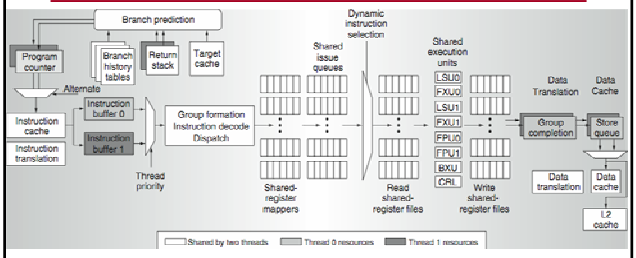
Power 5

2 commits (architected register sets)

2 fetch (PC), 2 initial decodes



Power 5 data flow ...

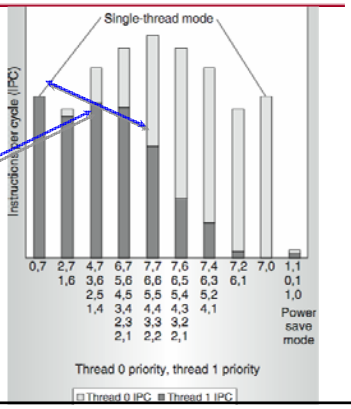


Why only 2 threads? With 4, one of the shared resources (physical registers, cache, memory bandwidth) would be prone to bottleneck

Power 5 thread performance ...

Relative priority of each thread controllable in hardware.

For balanced operation, both threads run slower than if they "owned" the machine.



Changes in Power 5 to support SMT

- Increased associativity of L1 instruction cache and the instruction address translation buffers
- Added per thread load and store queues
- Increased size of the L2 (1.92 vs. 1.44 MB) and L3 caches
- Added separate instruction prefetch and buffering per thread
- Increased the number of virtual registers from 152 to 240
- Increased the size of several issue queues
- The Power5 core is about 24% larger than the Power4 core because of the addition of SMT support

39

Summary: Limits to ILP

- Doubling issue rates above today's 3-6 instructions per clock, say to 6 to 12 instructions, probably requires a processor to
  - issue 3 or 4 data memory accesses per cycle,
  - resolve 2 or 3 branches per cycle,
  - rename and access more than 20 registers per cycle, and
  - fetch 12 to 24 instructions per cycle.
- The complexities of implementing these capabilities is likely to mean sacrifices in the maximum clock rate
  - E.g, widest issue processor is the Itanium 2, but it also has the slowest clock rate, despite the fact that it consumes the most power!

40

Limits to ILP

- Most techniques for increasing performance increase power consumption
- The key question is whether a technique is *energy efficient*: does it increase power consumption faster than it increases performance?
- Multiple issue processors techniques all are energy inefficient:
  1. Issuing multiple instructions incurs some overhead in logic that grows faster than the issue rate grows
  2. Growing gap between peak issue rates and sustained performance
- Number of transistors switching = f(peak issue rate), and performance = f(sustained rate), growing gap between peak and sustained performance
  ⇒ increasing energy per unit of performance

41

And in conclusion …

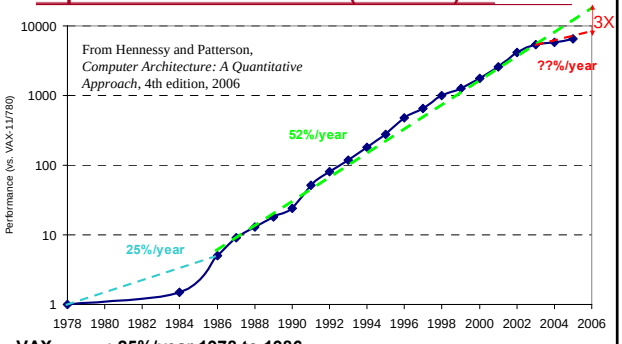
- Limits to ILP (power efficiency, compilers, dependencies …) seem to limit to 3 to 6 issue for practical options
- Explicitly parallel (Data level parallelism or Thread level parallelism) is next step to performance
  - Coarse grain vs. Fine grained multihreading
    - » Only on big stall vs. every clock cycle
- Itanium/EPIC/VLIW is not a breakthrough in ILP
  - In either scaling ILP or power consumption or complexity
- Balance of ILP and TLP decided in marketplace
- Instead of pursuing more ILP, architects are increasingly focusing on TLP implemented with single-chip multiprocessors: multi-core processors
- In 2000, IBM announced the 1st commercial single-chip, general-purpose multiprocessor, the Power4, which contains 2 Power3 processors and an integrated L2 cache
  - Since then, Sun Microsystems, AMD, and Intel have switch to a focus on single-chip multiprocessors rather than more aggressive uniprocessors.

42

Multiprocessor Architectures…

43

Uniprocessor Performance (SPECint)



- VAX : 25%/year 1978 to 1986
- RISC + x86: 52%/year 1986 to 2002
- RISC + x86: ??%/year 2002 to present

44

Déjà vu all over again?

"… today's processors … are nearing an impasse as technologies approach the speed of light.."

David Mitchell, *The Transputer: The Time Is Now* (1989)

- Transputer had bad timing (Uniprocessor performance↑)
  ⇒ Procrastination rewarded: 2X seq. perf. / 1.5 years
- "We are dedicating all of our future product development to multicore designs. … This is a sea change in computing"

  Paul Otellini, President, Intel (2005)
- All microprocessor companies switch to MP (2X CPUs / 2 yrs)
  ⇒ Procrastination penalized: 2X sequential perf. / 5 yrs

Manufacturer/Year	AMD/'05	Intel/'06	IBM/'04	Sun/'05
Processors/chip	2	2	2	8
Threads/Processor	1	2	2	4
Threads/chip	2	4	4	32

Other Factors ⇒ Multiprocessors

- Growth in data-intensive applications
  - Data bases, file servers, …
- Growing interest in servers, server perf.
- Increasing desktop perf. less important
  - Outside of graphics
- Improved understanding in how to use multiprocessors effectively
  - Especially server where significant natural TLP
- Advantage of leveraging design investment by replication
  - Rather than unique design

46

Flynn's Taxonomy

M.J. Flynn, "Very High-Speed Computers", *Proc. of the IEEE*, V 54, 1900-1909, Dec. 1966.

- Flynn classified by data and control streams in 1966

Single Instruction Single Data (SISD) (Uniprocessor)	Single Instruction Multiple Data SIMD (single PC: Vector, CM-2)
Multiple Instruction Single Data (MISD) (????)	Multiple Instruction Multiple Data MIMD (Clusters, SMP servers)

- SIMD ⇒ Data Level Parallelism
- MIMD ⇒ Thread Level Parallelism
- MIMD popular because
  - Flexible: N pgms and 1 multithreaded pgm
  - Cost-effective: same MPU in desktop & MIMD

47

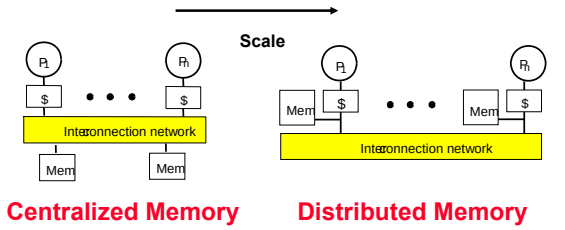
Back to Basics

- "A parallel computer is a collection of processing elements that cooperate and communicate to solve large problems fast."
- Parallel Architecture = Computer Architecture + Communication Architecture
- 2 classes of multiprocessors WRT memory:

1. Centralized Memory Multiprocessor
   - < few dozen processor chips (and < 100 cores) in 2006
   - Small enough to share single, centralized memory
2. Physically Distributed-Memory multiprocessor
   - Larger number chips and cores than 1.
   - BW demands ⇒ Memory distributed among processors

48

Centralized vs. Distributed Memory



49

Centralized Memory Multiprocessor

- Also called symmetric multiprocessors (SMPs) because single main memory has a symmetric relationship to all processors
- Large caches ⇒ single memory can satisfy memory demands of small number of processors
- Can scale to a few dozen processors by using a switch and by using many memory banks
- Although scaling beyond that is technically conceivable, it becomes less attractive as the number of processors sharing centralized memory increases

50

Distributed Memory Multiprocessor

- Pro: Cost-effective way to scale memory bandwidth
  - If most accesses are to local memory
- Pro: Reduces latency of local memory accesses
- Con: Communicating data between processors more complex
- Con: Must change software to take advantage of increased memory BW

51

2 Models for Communication and Memory Architecture

1. Communication occurs by explicitly passing messages among the processors: message-passing multiprocessors
2. Communication occurs through a shared address space (via loads and stores): shared memory multiprocessors either
   - UMA (Uniform Memory Access time) for shared address, centralized memory MP
   - NUMA (Non Uniform Memory Access time multiprocessor) for shared address, distributed memory MP
- In past, confusion whether "sharing" means sharing physical memory (Symmetric MP) or sharing address space

52

Challenges of Parallel Processing

- First challenge is % of program inherently sequential
- Suppose 80X speedup from 100 processors. What fraction of original program can be sequential?
  a. 10%
  b. 5%
  c. 1%
  d. <1%

53

Amdahl's Law Answers

$$\text{Speedup}_{\text{overall}} = \frac{1}{(1 - \text{Fraction}_{\text{enhanced}}) + \frac{\text{Fraction}_{\text{parallel}}}{\text{Speedup}_{\text{parallel}}}}$$

$$80 = \frac{1}{(1 - \text{Fraction}_{\text{parallel}}) + \frac{\text{Fraction}_{\text{parallel}}}{100}}$$

$$80 \times ((1 - \text{Fraction}_{\text{parallel}}) + \frac{\text{Fraction}_{\text{parallel}}}{100}) = 1$$

$$79 = 80 \times \text{Fraction}_{\text{parallel}} - 0.8 \times \text{Fraction}_{\text{parallel}}$$

$$\text{Fraction}_{\text{parallel}} = 79 / 79.2 = 99.75\%$$

54

Challenges of Parallel Processing

- Second challenge is long latency to remote memory
- Suppose 32 CPU MP, 2GHz, 200 ns remote memory, all local accesses hit memory hierarchy and base CPI is 0.5. (Remote access = 200/0.5 = 400 clock cycles.)
- What is performance impact if 0.2% instructions involve remote access?
  a. 1.5X
  b. 2.0X
  c. 2.5X

55

CPI Equation

- CPI = Base CPI + Remote request rate x Remote request cost
- CPI = 0.5 + 0.2% x 400 = 0.5 + 0.8 = 1.3
- No communication is 1.3/0.5 or 2.6 faster than 0.2% instructions involve local access

56

Challenges of Parallel Processing

1. Application parallelism ⇒ primarily via new algorithms that have better parallel performance
2. Long remote latency impact ⇒ both by architect and by the programmer
- For example, reduce frequency of remote accesses either by
  - Caching shared data (HW)
  - Restructuring the data layout to make more accesses local (SW)
- We will cover details on HW to help latency via caches

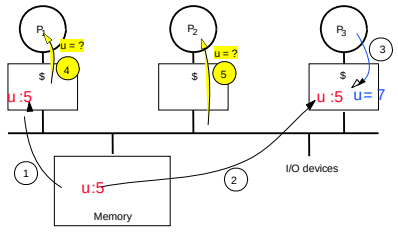
57

Symmetric Shared-Memory Architectures

- From multiple boards on a shared bus to multiple processors inside a single chip
- Caches both
  - Private data are used by a single processor
  - Shared data are used by multiple processors
- Caching shared data
  ⇒ reduces latency to shared data, memory bandwidth for shared data, and interconnect bandwidth
  ⇒ cache coherence problem

58

Example Cache Coherence Problem



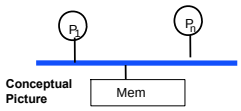
- Processors see different values for u after event 3
- With write back caches, value written back to memory depends on happenstance of which cache flushes or writes back value when
  - » Processes accessing main memory may see very stale value
- Unacceptable for programming, and its frequent!

59

Example

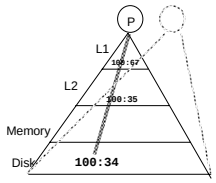
P_1	P_2
/*Assume initial value of A and flag is 0*/	
`A = 1;`	`while (flag == 0); /*spin idly*/`
`flag = 1;`	`print A;`

- Intuition not guaranteed by coherence
- expect memory to respect order between accesses to *different* locations issued by a given process
  - to preserve orders among accesses to same location by different processes
- Coherence is not enough!
  - pertains only to single location



60

Intuitive Memory Model



- Reading an address should return the last value written to that address
  - Easy in uniprocessors, except for I/O

- Too vague and simplistic; 2 issues
1. Coherence defines values returned by a read
2. Consistency determines when a written value will be returned by a read
- Coherence defines behavior to same location, Consistency defines behavior to other locations

61

What Does Coherency Mean?

- Informally:
  - "Any read must return the most recent write"
  - Too strict and too difficult to implement
- Better:
  - "Any write must eventually be seen by a read"
  - All writes are seen in proper order ("serialization")
- Two rules to ensure this:
  - "If P writes x and P1 reads it, P's write will be seen by P1 if the read and write are sufficiently far apart"
  - Writes to a single location are serialized: seen in one order
    - » Latest write will be seen
    - » Otherwise could see writes in illogical order (could see older value after a newer value)

62

Defining Coherent Memory System

1. Preserve Program Order: A read by processor P to location X that follows a write by P to X, with no writes of X by another processor occurring between the write and the read by P, always returns the value written by P
2. Coherent view of memory: Read by a processor to location X that follows a write by another processor to X returns the written value if the read and write are sufficiently separated in time and no other writes to X occur between the two accesses
3. Write serialization: 2 writes to same location by any 2 processors are seen in the same order by all processors
   - If not, a processor could keep value 1 since saw as last write
   - For example, if the values 1 and then 2 are written to a location, processors can never read the value of the location as 2 and then later read it as 1

63

Cache Coherence Protocols

1. Directory based — Sharing status of a block of physical memory is kept in just one location, the directory
2. Snooping — Every cache with a copy of data also has a copy of sharing status of block, but no centralized state is kept
   - All caches are accessible via some broadcast medium (a bus or switch)
   - All cache controllers monitor or snoop on the medium to determine whether or not they have a copy of a block that is requested on a bus or switch access
   - *We will cover details of Snooping based cache coherence protocol…*

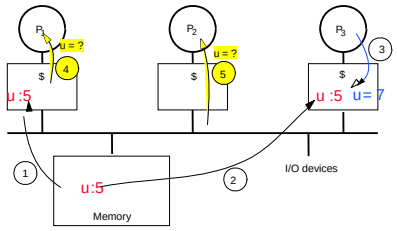
64

Symmetric Shared-Memory Architectures

- From multiple boards on a shared bus to multiple processors inside a single chip
- Caches both
  - Private data are used by a single processor
  - Shared data are used by multiple processors
- Caching shared data
  ⇒ reduces latency to shared data, memory bandwidth for shared data, and interconnect bandwidth
  ⇒ cache coherence problem

65

Example Cache Coherence Problem



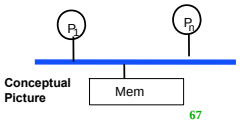
- Processors see different values for u after event 3
- With write back caches, value written back to memory depends on happenstance of which cache flushes or writes back value when
  - » Processes accessing main memory may see very stale value
- Unacceptable for programming, and its frequent!

66

Example

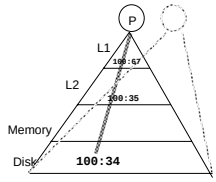
P_1	P_2
/*Assume initial value of A and flag is 0*/	
`A = 1;`	`while (flag == 0); /*spin idly*/`
`flag = 1;`	`print A;`

- Intuition not guaranteed by coherence
- expect memory to respect order between accesses to *different* locations issued by a given process
  - to preserve orders among accesses to same location by different processes
- Coherence is not enough!
  - pertains only to single location



67

Intuitive Memory Model



- Reading an address should return the last value written to that address
  - Easy in uniprocessors, except for I/O

- Too vague and simplistic; 2 issues
1. Coherence defines values returned by a read
2. Consistency determines when a written value will be returned by a read
- Coherence defines behavior to same location, Consistency defines behavior to other locations

68

Defining Coherent Memory System

1. Preserve Program Order: A read by processor P to location X that follows a write by P to X, with no writes of X by another processor occurring between the write and the read by P, always returns the value written by P
2. Coherent view of memory: Read by a processor to location X that follows a write by another processor to X returns the written value if the read and write are sufficiently separated in time and no other writes to X occur between the two accesses
3. Write serialization: 2 writes to same location by any 2 processors are seen in the same order by all processors
   - If not, a processor could keep value 1 since saw as last write
   - For example, if the values 1 and then 2 are written to a location, processors can never read the value of the location as 2 and then later read it as 1

69

Write Consistency

- For now assume

1. A write does not complete (and allow the next write to occur) until all processors have seen the effect of that write
2. The processor does not change the order of any write with respect to any other memory access

⇒ if a processor writes location A followed by location B, any processor that sees the new value of B must also see the new value of A

- These restrictions allow the processor to reorder reads, but forces the processor to finish writes in program order

70

Basic Schemes for Enforcing Coherence

- Program on multiple processors will normally have copies of the same data in several caches
  - Unlike I/O, where its rare
- Rather than trying to avoid sharing in SW, SMPs use a HW protocol to maintain coherent caches
  - Migration and Replication key to performance of shared data
- Migration - data can be moved to a local cache and used there in a transparent fashion
  - Reduces both latency to access shared data that is allocated remotely and bandwidth demand on the shared memory
- Replication – for reading shared data simultaneously, since caches make a copy of data in local cache
  - Reduces both latency of access and contention for read shared data

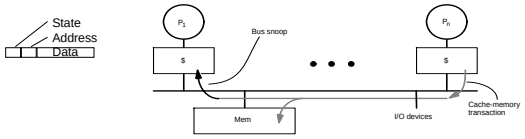
71

2 Classes of Cache Coherence Protocols

1. Directory based — Sharing status of a block of physical memory is kept in just one location, the directory
2. Snooping — Every cache with a copy of data also has a copy of sharing status of block, but no centralized state is kept
   - All caches are accessible via some broadcast medium (a bus or switch)
   - All cache controllers monitor or snoop on the medium to determine whether or not they have a copy of a block that is requested on a bus or switch access

72

Snoopy Cache-Coherence Protocols



- Cache Controller "snoops" all transactions on the shared medium (bus or switch)
  - relevant transaction if for a block it contains
  - take action to ensure coherence
    - » invalidate, update, or supply value
  - depends on state of the block and the protocol
- Either get exclusive access before write via write invalidate or update all copies on write

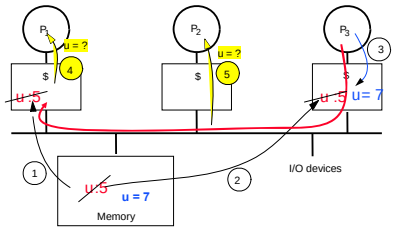
Basic Snoopy Protocols

- Write Invalidate Protocol:
  - Multiple readers, single writer
  - Write to shared data: an invalidate is sent to all caches which snoop and *invalidate* any copies
  - Read Miss:
    - » Write-through: memory is always up-to-date
    - » Write-back: snoop in caches to find most recent copy
- Write Broadcast Protocol (typically write through):
  - Write to shared data: broadcast on bus, processors snoop, and *update* any copies
  - Read miss: memory is always up-to-date
- Write serialization: bus serializes requests!
  - Bus is single point of arbitration

Basic Snoopy Protocols

- Write Invalidate versus Broadcast:
  - Invalidate requires one transaction per write-run
  - Invalidate uses spatial locality: one transaction per block
  - Broadcast has lower latency between write and read
  - Broadcast uses up bus bandwidth…does not scale well

Example: Write-thru Invalidate



- Must invalidate before step 3
- Write update uses more broadcast medium BW
  ⇒ all recent MPUs use write invalidate

Architectural Building Blocks

- Cache block state transition diagram
  - FSM specifying how disposition of block changes
    - » invalid, valid, exclusive
- Broadcast Medium Transactions (e.g., bus)
  - Fundamental system design abstraction
  - Logically single set of wires connect several devices
  - Protocol: arbitration, command/addr, data
  - ⇒ Every device observes every transaction
- Broadcast medium enforces serialization of read or write accesses ⇒ Write serialization
  - 1st processor to get medium invalidates others copies
  - Implies cannot complete write until it obtains bus
  - All coherence schemes require serializing accesses to same cache block
- Also need to find up-to-date copy of cache block

77

Locate up-to-date copy of data

- Write-through: get up-to-date copy from memory
  - Write through simpler if enough memory BW
- Write-back harder
  - Most recent copy can be in a cache
- Can use same snooping mechanism
  1. Snoop every address placed on the bus
  2. If a processor has dirty copy of requested cache block, it provides it in response to a read request and aborts the memory access
  - Complexity from retrieving cache block from cache, which can take longer than retrieving it from memory
- Write-back needs lower memory bandwidth
  ⇒ Support larger numbers of faster processors
  ⇒ Most multiprocessors use write-back

Cache Resources for WB Snooping

- Normal cache tags can be used for snooping
- Valid bit per block makes invalidation easy
- Read misses easy since rely on snooping
- Writes ⇒ Need to know if know whether any other copies of the block are cached
  - No other copies ⇒ No need to place write on bus for WB
  - Other copies ⇒ Need to place invalidate on bus

79

Cache Resources for WB Snooping

- To track whether a cache block is shared, add extra state bit associated with each cache block, like valid bit and dirty bit
  - Write to Shared block ⇒ Need to place invalidate on bus and mark cache block as private (if an option)
  - No further invalidations will be sent for that block
  - This processor called owner of cache block
  - Owner then changes state from shared to unshared (or exclusive)

80

Cache behavior in response to bus

- Every bus transaction must check the cache-address tags
  - could potentially interfere with processor cache accesses
- A way to reduce interference is to duplicate tags
  - One set for caches access, one set for bus accesses
- Another way to reduce interference is to use L2 tags
  - Since L2 less heavily used than L1

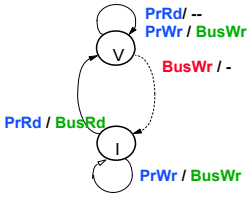
  ⇒ Every entry in L1 cache must be present in the L2 cache, called the inclusion property
  - If Snoop gets a hit in L2 cache, then it must arbitrate for the L1 cache to update the state and possibly retrieve the data, which usually requires a stall of the processor

Example Protocol

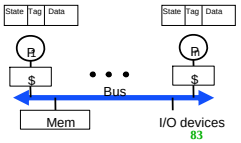
- Snooping coherence protocol is usually implemented by incorporating a finite-state controller in each node
- Logically, think of a separate controller associated with each cache block
  - That is, snooping operations or cache requests for different blocks can proceed independently
- In implementations, a single controller allows multiple operations to distinct blocks to proceed in interleaved fashion
  - that is, one operation may be initiated before another is completed, even through only one cache access or one bus access is allowed at time

Write-through Invalidate Protocol

- 2 states per block in each cache
  - as in uniprocessor
  - state of a block is a *p*-vector of states
  - Hardware state bits associated with blocks that are in the cache
  - other blocks can be seen as being in invalid (not-present) state in that cache
- Writes invalidate all other cache copies
  - can have multiple simultaneous readers of block,but write invalidates them



PrRd: Processor Read
PrWr: Processor Write
BusRd: Bus Read
BusWr: Bus Write

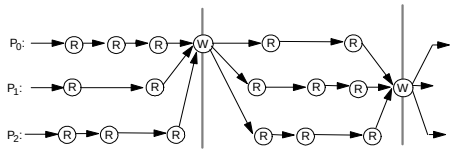


Is 2-state Protocol Coherent?

- Processor only observes state of memory system by issuing memory operations
- Assume bus transactions and memory operations are atomic and a one-level cache
  - all phases of one bus transaction complete before next one starts
  - processor waits for memory operation to complete before issuing next
  - with one-level cache, assume invalidations applied during bus transaction
- All writes go to bus + atomicity
  - Writes serialized by order in which they appear on bus (bus order)

  => invalidations applied to caches in bus order
- How to insert reads in this order?
  - Important since processors see writes through reads, so determines whether write serialization is satisfied
  - But read hits may happen independently and do not appear on bus or enter directly in bus order
- Let's understand other ordering issues

Ordering



- Writes establish a partial order
- Doesn't constrain ordering of reads, though shared-medium (bus) will order read misses too
  - any order among reads between writes is fine, as long as in program order

Example Write Back Snoopy Protocol

- Invalidation protocol, write-back cache
  - Snoops every address on bus
  - If it has a dirty copy of requested block, provides that block in response to the read request and aborts the memory access
- Each memory block is in one state:
  - Clean in all caches and up-to-date in memory (Shared)
  - OR Dirty in exactly one cache (Exclusive)
  - OR Not in any caches
- Each cache block is in one state (track these):
  - Shared : block can be read
  - OR Exclusive : cache has only copy, its writeable, and dirty
  - OR Invalid : block contains no data (in uniprocessor cache too)
- Read misses: cause all caches to snoop bus
- Writes to clean blocks are treated as misses

Cache Coherence Mechanism

- Cache coherence mechanism (protocol/controller) receives requests from:
  - Processor
  - Bus
- Responds to requests
  - Depending on hit or miss, read or write, and state of block

Request	Source	State of addressed cache block	Type of cache action	Function and explanation
Read Hit	Proc.	Shared or modified	Normal hit	Read data in cache
Read miss	Proc.	Invalid	Normal miss	Place read miss on bus
Read miss	Proc.	Shared	Replacement	Address conflict miss:place read miss on bus
Read miss	Proc.	Modified	Replacement	Address conflict miss: write back block, then place read miss on bus

Request	Source	State of addressed cache block	Type of cache action	Function and explanation
Write Hit	Proc.	Modified	Normal hit	Write data in cache
Write Hit	Proc.	Shared	Coherence	*Place invalidate on bus. These operations only change state and do not fetch data*
Write miss	Proc.	Invalid	Normal miss	Place write miss on bus
Write miss	Proc.	Shared	Replacement	Address conflict miss: place write miss on bus
Write miss	Proc.	Modified	Replacement	Address conflict miss: write back block, then place write miss on bus

89

Request	Source	State of addressed cache block	Type of cache action	Function and explanation
Read Miss	Bus	Shared	No action	Allow memory to service read miss
Read miss	Bus	Modified	Coherence	Attempt to share data: place cache block on bus and change state to shared
Invalidate	Bus	Shared	Coherence	Attempt to write shared block: invalidate the block
Write miss	Bus	Shared	Coherence	Attempt to write block that is shared: invalidate the block
Write miss	Bus	Modified	Coherence	Attempt to write block that is exclusive elsewhere: write back the cache block and make its state invalid

90

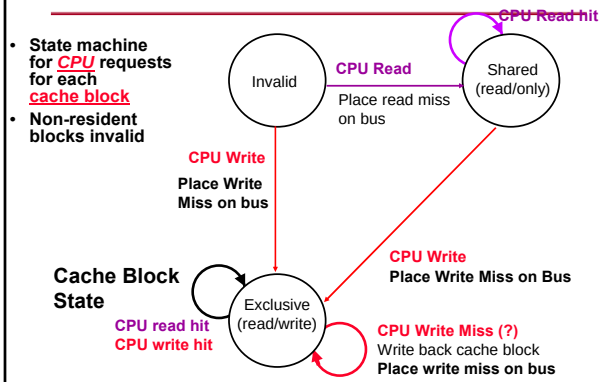
Cache Coherence Protocol

- Can implement the protocol using Finite State Machines

91

Write-Back State Machine - CPU

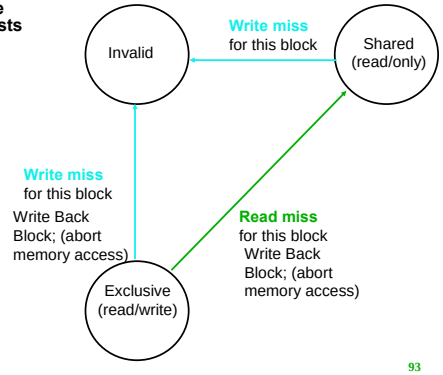
- State machine for *CPU* requests for each cache block
- Non-resident blocks invalid



92

Write-Back State Machine- Bus request

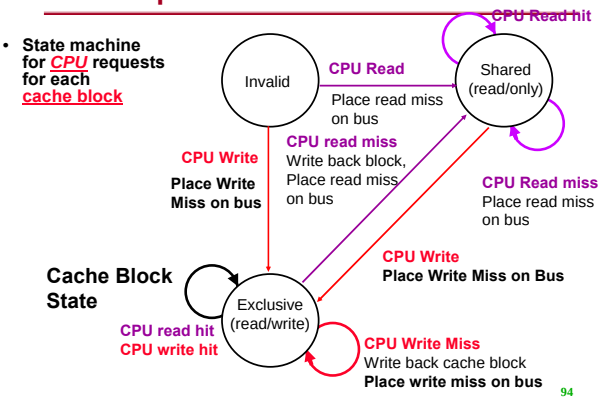
- State machine for *bus* requests for each cache block



93

Block-replacement

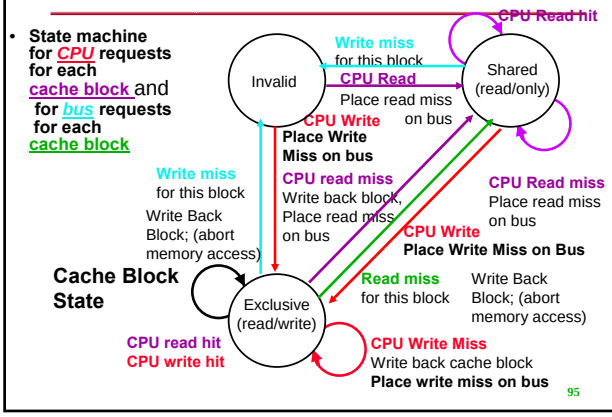
- State machine for *CPU* requests for each cache block



94

Write-back State Machine-III

- State machine for *CPU* requests for each cache block and for *bus* requests for each cache block



95

Example

	P1			P2			Bus				Memory	
step	State	Addr	Value	State	Addr	Value	Action	Proc.	Addr	Value	Addr	Value
P1 Write 10 to A1												
P1: Read A1												
P2: Read A1												
P2: Write 20 to A1												
P2: Write 40 to A2												

Assumes A1 and A2 map to same cache block, initial cache state is invalid

96

Example

step	P1 State	P1 Addr	P1 Value	P2 State	P2 Addr	P2 Value	Bus Action	Bus Proc.	Bus Addr	Bus Value	Memory Addr	Memory Value
P1 Write 10 to A1	Excl.	A1	10				WrMs	P1	A1			
P1: Read A1												
P2: Read A1												
P2: Write 20 to A1												
P2: Write 40 to A2												

Assumes A1 and A2 map to same cache block

97

Example

step	P1 State	P1 Addr	P1 Value	P2 State	P2 Addr	P2 Value	Bus Action	Bus Proc.	Bus Addr	Bus Value	Memory Addr	Memory Value
P1 Write 10 to A1	Excl.	A1	10				WrMs	P1	A1			
P1: Read A1	Excl.	A1	10									
P2: Read A1												
P2: Write 20 to A1												
P2: Write 40 to A2												

Assumes A1 and A2 map to same cache block

98

Example

step	P1 State	P1 Addr	P1 Value	P2 State	P2 Addr	P2 Value	Bus Action	Bus Proc.	Bus Addr	Bus Value	Memory Addr	Memory Value
P1 Write 10 to A1	Excl.	A1	10				WrMs	P1	A1			
P1: Read A1	Excl.	A1	10									
P2: Read A1				Shar.	A1		RdMs	P2	A1			
	Shar.	A1	10				WrBk	P1	A1	10	A1	10
				Shar.	A1	10	RdDa	P2	A1	10	A1	10
P2: Write 20 to A1												
P2: Write 40 to A2												

Assumes A1 and A2 map to same cache block

99

Example

step	P1 State	P1 Addr	P1 Value	P2 State	P2 Addr	P2 Value	Bus Action	Bus Proc.	Bus Addr	Bus Value	Memory Addr	Memory Value
P1 Write 10 to A1	Excl.	A1	10				WrMs	P1	A1			
P1: Read A1	Excl.	A1	10									
P2: Read A1				Shar.	A1		RdMs	P2	A1			
	Shar.	A1	10				WrBk	P1	A1	10	A1	10
				Shar.	A1	10	RdDa	P2	A1	10	A1	10
P2: Write 20 to A1	Inv.			Excl.	A1	20	WrMs	P2	A1		A1	10
P2: Write 40 to A2												

Assumes A1 and A2 map to same cache block

100

Example

	P1			P2			Bus				Memory	
step	State	Addr	Value	State	Addr	Value	Action	Proc.	Addr	Value	Addr	Value
P1 Write 10 to A1	Excl.	A1	10				WrMs	P1	A1			
P1: Read A1	Excl.	A1	10									
P2: Read A1				Shar.	A1		RdMs	P2	A1			
	Shar.	A1	10				WrBk	P1	A1	10	A1	10
				Shar.	A1	10	RdDa	P2	A1	10	A1	10
P2: Write 20 to A1	Inv.			Excl.	A1	20	WrMs	P2	A1		A1	10
P2: Write 40 to A2							WrMs	P2	A2		A1	10
				Excl.	A2	40	WrBk	P2	A1	20	A1	20

Assumes A1 and A2 map to same cache block,
but A1 != A2

101

Implementation Complications

- **Write Races:**
  - **Cannot update cache until bus is obtained**
    - » **Otherwise, another processor may get bus first, and then write the same cache block!**
  - **Two step process:**
    - » **Arbitrate for bus**
    - » **Place miss on bus and complete operation**
  - **If miss occurs to block while waiting for bus, handle miss (invalidate may be needed) and then restart.**
  - **Split transaction bus:**
    - » **Bus transaction is not atomic: can have multiple outstanding transactions for a block**
    - » **Multiple misses can interleave, allowing two caches to grab block in the Exclusive state**
    - » **Must track and prevent multiple misses for one block**
- **Must support interventions and invalidations**

102

Implementing Snooping Caches

- **Multiple processors must be on bus, access to both addresses and data**
- **Add a few new commands to perform coherency, in addition to read and write**
- **Processors continuously snoop on address bus**
  - **If address matches tag, either invalidate or update**
- **Since every bus transaction checks cache tags, could interfere with CPU just to check:**
  - **solution 1: duplicate set of tags for L1 caches just to allow checks in parallel with CPU**
  - **solution 2: L2 cache already duplicate, provided L2 obeys inclusion with L1 cache**
    - » **block size, associativity of L2 affects L1**

103

Limitations in Symmetric Shared-Memory Multiprocessors and Snooping Protocols

- **Single memory accommodate all CPUs ⇒ Multiple memory banks**
- **Bus-based multiprocessor, bus must support both coherence traffic & normal memory traffic**

⇒ Multiple buses or interconnection networks (cross bar or small point-to-point)

- **Opteron**
  - **Memory connected directly to each dual-core chip**
  - **Point-to-point connections for up to 4 chips**
  - **Remote memory and local memory latency are similar, allowing OS Opteron as UMA computer**

104

Performance of Symmetric Shared-Memory Multiprocessors

- Cache performance is combination of
1. Uniprocessor cache miss traffic
2. Traffic caused by communication
   - Results in invalidations and subsequent cache misses
- 4th C: *coherence miss*
   - Joins Compulsory, Capacity, Conflict

105

Coherency Misses

1. True sharing misses arise from the communication of data through the cache coherence mechanism
   - Invalidates due to 1st write to shared block
   - Reads by another CPU of modified block in different cache
   - Miss would still occur if block size were 1 word
2. False sharing misses when a block is invalidated because some word in the block, other than the one being read, is written into
   - Invalidation does not cause a new value to be communicated, but only causes an extra cache miss
   - Block is shared, but no word in block is actually shared ⇒ miss would not occur if block size were 1 word

106

Example: True v. False Sharing v. Hit?

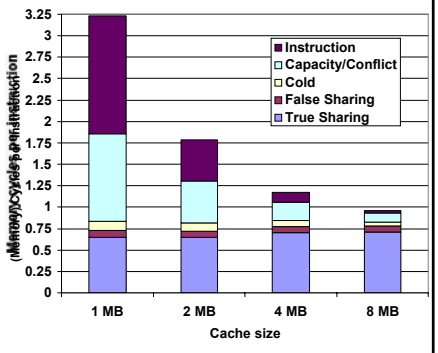
- Assume x1 and x2 in same cache block. P1 and P2 both read x1 and x2 before.

Time	P1	P2	True, False, Hit? Why?
1	Write x1		True miss; invalidate x1 in P2
2		Read x2	False miss; x1 irrelevant to P2
3	Write x1		False miss; x1 irrelevant to P2
4		Write x2	False miss; x1 irrelevant to P2
5	Read x2		True miss; invalidate x2 in P1

107

MP Performance 4 Processor Commercial Workload: OLTP, Decision Support (Database), Search Engine

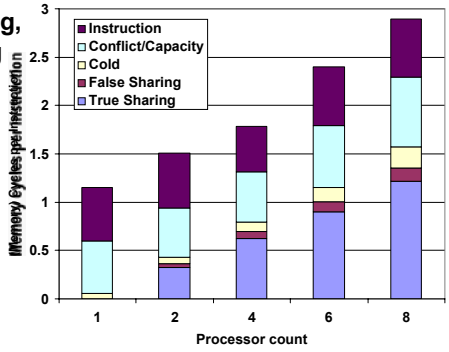
- True sharing and false sharing unchanged going from 1 MB to 8 MB (L3 cache)
- Uniprocessor cache misses improve with cache size increase (Instruction, Capacity/Conflict, Compulsory)



108

MP Performance 2MB Cache Commercial Workload: OLTP, Decision Support (Database), Search Engine

- True sharing, false sharing increase going from 1 to 8 CPUs



Bus-based Coherence

- done through broadcast on bus
- 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

Directory Protocol

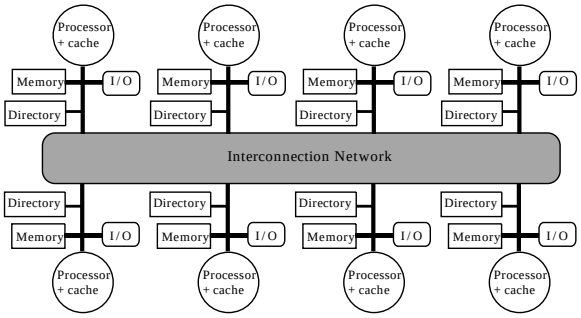
- No bus and don't want to broadcast:
  - interconnect no longer single arbitration point
  - all messages have explicit responses
- Terms: typically 3 processors involved
  - Local node where a request originates
  - Home node where the memory location of an address resides
  - Remote node has a copy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

Directory Protocol

- Similar to Snoopy 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

113

Distributed Directory MPs



114

A Popular Middle Ground

- Two-level "hierarchy"
- Individual nodes are multiprocessors, connected non-hiearchically
  - 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?

115

Summary

- Caches contain all information on state of cached memory blocks
- 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

116

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

117

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

118

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

119

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

120

Fundamental Issue: Synchronization

- To cooperate, processes must coordinate
- Message passing is implicit coordination with transmission or arrival of data
- Shared address
  => additional operations to explicitly coordinate: e.g., write a flag, awaken a thread, interrupt a processor

121

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

122

Coordination/Synchronization Constructs

- For shared memory and message passing two types of synchronization activity
  - Sequence control ... to enable correct operation
  - Access control ... to allow access to common resources
- synchronization activities constitute an overhead!

123

Synchronization Constructs

- Barrier synchronization
  - for sequence control
  - processors wait at barrier till all (or subset) have completed
  - hardware implementations available
  - can also implement in s/w
- Critical section access control mechanisms
  - Test&Set Lock protocols
  - Semaphores

124

Barrier Synchronization

- Many programs require that all processes come to a "barrier" point before proceeding further
  - this constitutes a synchronization point
- Concept of Barrier
  - When processor hits a barrier it cannot proceed further until ALL processors have hit the barrier point
    - » note that this forces a global synchronization point
- Can implement in S/W or Hardware
  - in s/w can implement using a shared variable; proc checks value of shared variable

125

Barrier Synch. . . Example

```
For i := 1 to N   do in parallel
    A[i] :=  k* A[i];
    B[i] := A[i] + B[i];
endfor
BARRIER POINT
for i := 1 to N   do in parallel
    C[i] :=  B[i] + B[i-1] + B[i-2];
```

126

Barrier Synchronization: Implementation

- Bus based
  - each processor sets single bit when it arrives at barrier
  - collection of bits sent to AND (or OR) gates
  - send outputs of gates to all processors
  - number of synchs/cycle grows with N (proc) if change in bit at one proc can be propagated in since cycle
    - » takes O(log N) in reality
  - delay in performance due to barrier measured how?
- Multiple Barrier lines
  - a barrier bit sent to each processor
  - each can set bit for each barrier line
  - X1,...,Xn in processor; Y1,...,Yn is barrier setting

127

Synchronization: Message Passing

- Synchronous vs. Asynchronous
- Synchronous: sending and receiving process synch in time and space
  - system must check if (i) receiver ready, (ii) path available and (iii) one or more to be sent to same or multiple dest
  - also known as blocking send-receive
  - send and receive process cannot continue past instruction till message transfer complete
- Asychronous: send&rec do not have to synch

128

Lock Protocols

```
Test&Set (lock)
    temp <- lock
    lock := 1
    return (temp);

Reset (lock)
    lock :=0

Process waits for lock to be 0
    can remove indefinite waits by ???
```


Semaphores

- P(S) for shared variable/section S
  - test if S>0 & enter critical section and decrement S else wait
- V(S)
  - increment S and exit
- note that P and V are blocking synchronization constructs
- can allow number of concurrent accesses to S

Semaphores : Example

Z= A*B + [(C*D) * (I+G)]

```
var S_w, S_y are semaphores
initial:  S_w=S_y= 0
P1:  begin                      P2: begin
        U = A*B                 W = C*D
        P( S_y)                         V(S_w)
        Z = U + Y                       end
    end

P3: begin
        X= I+G
        P(S_w)
        Y= W*X
        V(S_y)
end
```


Hardware level synchronization

- Key is to provide uninterruptible instruction or instruction sequence capable of atomically retrieving a value
  - S/W mechanisms then constructed from these H/W primitives
- Uninterruptible instruction
  - Atomic exchange
  - Test & Set
  - Fetch & Increment
  - Build high level synchronization primitives using one of these…
- ….

Fundamental Issue: Performance and Scalability

- Performance must scale with
  - system size
  - problem/workload size
- Amdahl's Law
  - perfect speedup cannot be achieved since there is a inherently sequential part to every program
- Scalability measures
  - Efficiency (speedup per processor)

133

Parallel Algorithms

- Solving problems on a multiprocessor architecture requires design of parallel algorithms
- How do we measure efficiency of a parallel algorithm ?
  - 10 seconds on Machine 1 and 20 seconds on machine 2 – which algorithm is better ?

134

Parallel Algorithm Complexity

- Parallel time complexity
  - Express in terms of Input size and System size (num of processors)
  - T(N,P): input size N, P processors
  - Relationship between N and P
    - » Independent size analysis – no link between N and P
    - » Dependent size – P is a function of N; eg. P=N/2
- Speedup: how much faster than sequential
  - S(P)= T(N,1)/T(N,P)
- Efficiency: speedup per processor
  - S(P)/P

135

Parallel Computation Models

- Shared Memory
  - Protocol for shared memory ?..what happens when two processors/processes try to access same data
    - » EREW: Exclusive Read, Exclusive Write
    - » CREW: Concurrent Read, Exclusive Write
    - » CRCW: Concurrent read, Concurrent write
- Distributed Memory
  - Explicit communication through message passing
    - » Send/Receive instructions

136

Formal Models of Parallel Computation

- Alternating Turing machine
- P-RAM model
  - Extension of sequential Random Access Machine (RAM) model
- RAM model
  - One program
  - One memory
  - One accumulator
  - One read/write tape

P-RAM model

- P programs, one per processor
- One memory
  - In distributed memory it becomes P memories
- P accumulators
- One read/write tape
- Depending on shared memory protocol we have
  - CREW P-RAM
  - EREW PRAM
  - CRCW PRAM

PRAM Model

- Assumes synchronous execution
- Idealized machine
  - Helps in developing theoretically sound solutions
  - Actual performance will depend on machine characteristics and language implementation

PRAM Algorithms -- Summing

- Add N numbers in parallel using P processors
  - How to parallelize ?

Parallel Summing

- Using N/2 processors to sum N numbers in O(Log N) time
- Independent size analysis:
  - Do sequential sum on N/P values and then add in parallel
  - Time= O(N/P + log P)

141

Parallel Sorting on CREW PRAM

- Sort N numbers using P processors
  - Assume P unlimited for now.
- Given an unsorted list (a1, a2,…,an) created sorted list W, where W[i]<W[i+1]
- Where does a1 go ?

142

Parallel Sorting on CREW PRAM

- Using $P=N^2$ processors
- For each processor P(i,j) compare ai>aj
  - If ai>aj then R[i,j]=1 else 0
  - Time = O(1)
- For each "row" of processors P(i,j) for j=1 to j=N do parallel sum to compute rank
  - Compute R[i]= sum of R[i,j]
  - Time = O(log N)
- Write ai into W[R(i)]
- Total time complexity= O(log N)

143

Parallel Algorithms

- Design of parallel algorithm has to take system architecture into consideration
- Must minimize interprocessor communication in a distributed memory system
  - Communication time is much larger than computation
  - Comm. Time can dominate computation if not problem is not "partitioned" well
- Efficiency

144

145

Synchronization..continued..

- 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

146

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

147

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)
```

148

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?
```

- Other user level synchronization primitives can be constructed similarly from HW primitives

149

Challenges of Parallel Processing

1. Application parallelism ⇒ primarily via new algorithms that have better parallel performance
2. Long remote latency impact ⇒ both by architect and by the programmer

- For example, reduce frequency of remote accesses either by
  - Caching shared data (HW)
  - Restructuring the data layout to make more accesses local (SW)
- Today's lecture on HW to help latency via caches

150

Multiprocessors: Summary

- Parallel processing is "old news"
- ILP is "old news"
- To get over the "ILP Wall" use Simultaneous multithreading (SMT) – this is "new" news!
- Put multiprocessing on a single chip, to get multi-core processors –this is "new" news!

151

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

152