

Computer Architecture

2007-2008

Organization (www.liacs.nl/ca)

People

- Lecturer: Lex Wolters
- Assignment leader: Harmen van der Spek
- Assistant: Van Thieu Vu
- Student assistants: Eyal Halm & Joris Huizer

Lectures (3 EC)

- Wednesday 11.15-13.00h till Dec 5th (except Oct 3rd)
- Book: Hennessy & Patterson, **fourth edition!**
- Exam: date unknown yet



Assignment (4 EC)

- Parts 1 (10%), 2a (30%), 2b (30%), 3 (30%): strict deadlines
- Assistance (room 306):
 - » Wed 13.45-15.30h (scheduled): **this afternoon Intro part 1**
 - » Mon, Tue, Thu 15.30-16.30h

Lecture 1 - Introduction

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

Outline

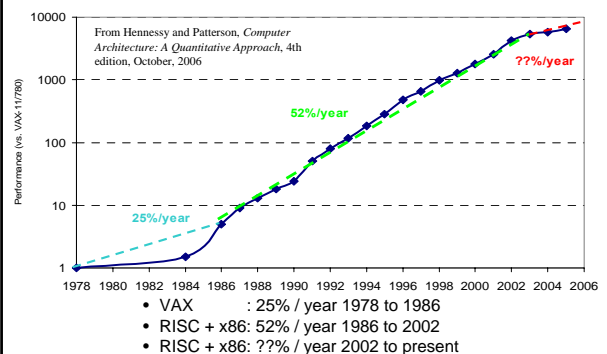
- Computer Science at a Crossroads
- Computer Architecture v. Instruction Set Arch.
- What Computer Architecture brings to table

Break

Crossroads: Conventional Wisdom in Comp. Arch

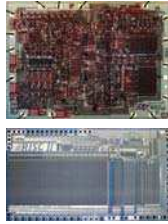
- Old Conventional Wisdom: Power is free, Transistors expensive
 - New Conventional Wisdom: "Power wall" Power expensive, Xtors free (can put more on chip than can afford to turn on)
 - Old CW: Sufficiently increasing Instruction Level Parallelism via compilers, innovation (Out-of-order, speculation, VLIW, ...)
 - New CW: "ILP wall" law of diminishing returns on more HW for ILP
 - Old CW: Multiplies are slow, Memory access is fast
 - New CW: "Memory wall" Memory slow, multiplies fast (200 clock cycles to DRAM memory, 4 clocks for multiply)
 - Old CW: Uniprocessor performance 2X / 1.5 yrs
 - New CW: Power Wall + ILP Wall + Memory Wall = **Brick Wall**
 - Uniprocessor performance now 2X / 5(?) yrs
- ⇒ Sea change in chip design: multiple "cores"
(2X processors per chip / ~ 2 years)
» More simpler processors are more power efficient

Crossroads: Uniprocessor Performance



Sea Change in Chip Design

- Intel 4004 (1971): 4-bit processor, 2312 transistors, 0.4 MHz, 10 micron PMOS, 11 mm² chip
- RISC II (1983): 32-bit, 5 stage pipeline, 40,760 transistors, 3 MHz, 3 micron NMOS, 60 mm² chip
- 125 mm² chip, 0.065 micron CMOS = 2312 RISC II+FPU+Icache+Dcache
 - RISC II shrinks to ~0.02 mm² at 65 nm
 - Caches via DRAM or 1 transistor SRAM?



- Processor is the new transistor?

Déjà vu all over again?

- Multiprocessors imminent in 1970s, '80s, '90s, ...
- "... today's processors ... are nearing an impasse as technologies approach the speed of light.."
 - David Mitchell, *The Transputer: The Time Is Now* (1989)
- Transputer was premature
 - ⇒ Custom multiprocessors strove to lead uniprocessors
 - ⇒ 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 (2004)
- Difference is all microprocessor companies switch to multiprocessors (AMD, Intel, IBM, Sun; all new Apples 2 CPUs)
 - ⇒ Procrastination penalized: 2X sequential perf. / 5 yrs
 - ⇒ Biggest programming challenge: 1 to 2 CPUs

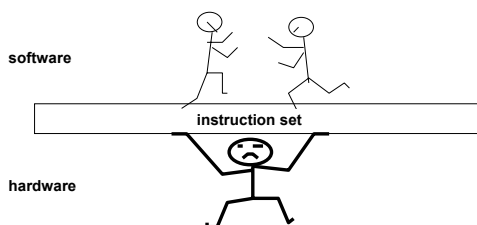
Problems with Sea Change

- Algorithms, Programming Languages, Compilers, Operating Systems, Architectures, Libraries, ... not ready to supply Thread Level Parallelism or Data Level Parallelism for 1000 CPUs / chip
- Architectures not ready for 1000 CPUs / chip
 - Unlike Instruction Level Parallelism, cannot be solved by just by computer architects and compiler writers alone, but also cannot be solved *without* participation of computer architects
- The 4th edition of the textbook '*Computer Architecture: A Quantitative Approach*' explores shift from Instruction Level Parallelism to Thread Level Parallelism / Data Level Parallelism

Outline

- Computer Science at a Crossroads
- Computer Architecture v. Instruction Set Arch.
- What Computer Architecture brings to table

Instruction Set Architecture: Critical Interface



- Properties of a good abstraction
 - Lasts through many generations (portability)
 - Used in many different ways (generality)
 - Provides **convenient** functionality to higher levels
 - Permits an **efficient** implementation at lower levels

Example: MIPS

r0	0	Programmable storage	Data types ?
r1			
o			
o			
r31			
PC		2 ³² x bytes	Format ?
lo		31 x 32-bit GPRs (R0=0)	Addressing Modes?
hi		32 x 32-bit FP regs (paired DP)	
		HI, LO, PC	
Arithmetic logical			
Add, AddU, Sub, SubU, And, Or, Xor, Nor, SLT, SLTU, Addl, AddIU, SLTI, SLTIU, Andl, Ori, Xori, LUI			
SLL, SRL, SRA, SLLV, SRLV, SRAV			
Memory Access			
LB, LBU, LH, LHU, LW, LWL, LWR			
SB, SH, SW, SWL, SWR			
Control			
J, JAL, JR, JALR			
BEq, BNE, BLEZ, BGTZ, BLTZ, BGEZ, BLTZAL, BGEZAL			

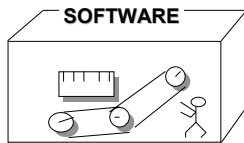
32-bit instructions on word boundary

Instruction Set Architecture

“... the attributes of a [computing] system as seen by the programmer, *i.e.* the conceptual structure and functional behavior, as distinct from the organization of the data flows and controls the logic design, and the physical implementation.”

– Amdahl, Blaauw, and Brooks, 1964

- Organization of Programmable Storage
- Data Types & Data Structures: Encodings & Representations
- Instruction Formats
- Instruction (or Operation Code) Set
- Modes of Addressing and Accessing Data Items and Instructions
- Exceptional Conditions



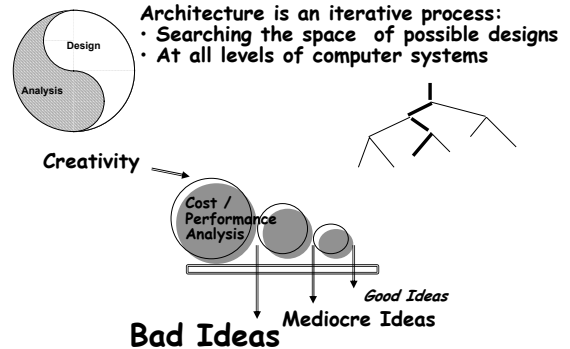
ISA vs. Computer Architecture

- Old definition of computer architecture = instruction set design
 - Other aspects of computer design called implementation
 - Insinuates implementation is uninteresting or less challenging
- Our view is computer architecture >> ISA
- Architect's job much more than instruction set design; technical hurdles today *more* challenging than those in instruction set design
- Since instruction set design not where action is, some conclude computer architecture (using old definition) is not where action is
 - We disagree on conclusion
 - Agree that ISA not where action is (ISA in appendix B)

Comp. Arch. is an Integrated Approach

- What really matters is the functioning of the complete system
 - hardware, runtime system, compiler, operating system, and application
 - In networking, this is called the “End to End argument”
- Computer architecture is not just about transistors, individual instructions, or particular implementations
 - E.g., Original RISC projects replaced complex instructions with a compiler + simple instructions

Computer Architecture is Design and Analysis



Outline

- Computer Science at a Crossroads
- Computer Architecture v. Instruction Set Arch.
- What Computer Architecture brings to table

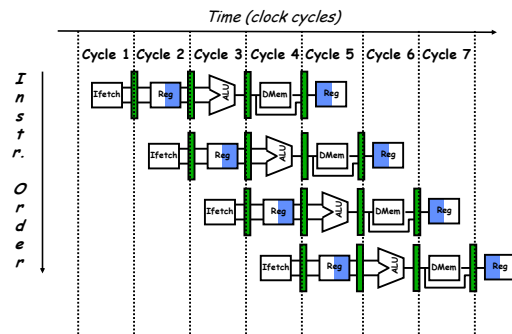
What Computer Architecture brings to Table

- Other fields often borrow ideas from architecture
- Quantitative Principles of Design
 1. Take Advantage of Parallelism
 2. Principle of Locality
 3. Focus on the Common Case
 4. Amdahl's Law
 5. The Processor Performance Equation
- Careful, quantitative comparisons
 - Define, quantify, and summarize relative performance
 - Define and quantify relative cost
 - Define and quantify dependability
 - Define and quantify power
- Culture of anticipating and exploiting advances in technology
- Culture of well-defined interfaces that are carefully implemented and thoroughly checked

1) Take Advantage of Parallelism

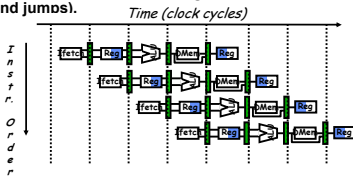
- Increasing throughput of server computer via multiple processors or multiple disks
- Detailed HW design
 - Carry lookahead adders uses parallelism to speed up computing sums from linear to logarithmic in number of bits per operand
 - Multiple memory banks searched in parallel in set-associative caches
- Pipelining: overlap instruction execution to reduce the total time to complete an instruction sequence.
 - Not every instruction depends on immediate predecessor \Rightarrow executing instructions completely/partially in parallel possible
 - Classic 5-stage pipeline:
 - 1) Instruction Fetch (Ifetch),
 - 2) Register Read (Reg),
 - 3) Execute (ALU),
 - 4) Data Memory Access (Dmem),
 - 5) Register Write (Reg)

Pipelined Instruction Execution



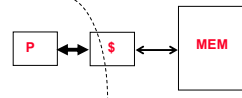
Limits to pipelining

- Hazards prevent next instruction from executing during its designated clock cycle
 - Structural hazards:** attempt to use the same hardware to do two different things at once
 - Data hazards:** Instruction depends on result of prior instruction still in the pipeline
 - Control hazards:** Caused by delay between the fetching of instructions and decisions about changes in control flow (branches and jumps).

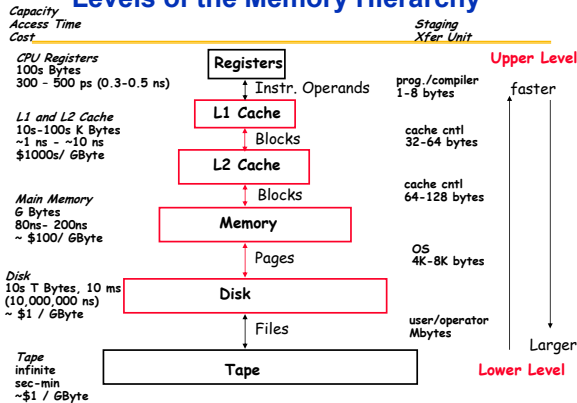


2) The Principle of Locality

- The Principle of Locality:
 - Program access a relatively small portion of the address space at any instant of time.
- Two Different Types of Locality:
 - Temporal Locality** (Locality in Time): If an item is referenced, it will tend to be referenced again soon (e.g., loops, reuse)
 - Spatial Locality** (Locality in Space): If an item is referenced, items whose addresses are close by tend to be referenced soon (e.g., straight-line code, array access)
- Last 30 years, HW relied on locality for memory perf.



Levels of the Memory Hierarchy



3) Focus on the Common Case

- Common sense guides computer design
 - Since its engineering, common sense is valuable
- In making a design trade-off, favor the frequent case over the infrequent case
 - E.g., Instruction fetch and decode unit used more frequently than multiplier, so optimize it 1st
 - E.g., If database server has 50 disks / processor, storage dependability dominates system dependability, so optimize it 1st
- Frequent case is often simpler and can be done faster than the infrequent case
 - E.g., overflow is rare when adding 2 numbers, so improve performance by optimizing more common case of no overflow
 - May slow down overflow, but overall performance improved by optimizing for the normal case
- What is frequent case and how much performance improved by making case faster \Rightarrow Amdahl's Law

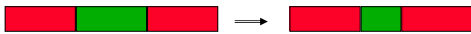
4) Amdahl's Law

$$\text{ExTime}_{\text{new}} = \text{ExTime}_{\text{old}} \times \left[(1 - \text{Fraction}_{\text{enhanced}}) + \frac{\text{Fraction}_{\text{enhanced}}}{\text{Speedup}_{\text{enhanced}}} \right]$$

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

Best you could ever hope to do:

$$\text{Speedup}_{\text{maximum}} = \frac{1}{(1 - \text{Fraction}_{\text{enhanced}})}$$



Amdahl's Law example

- New CPU 10X faster
- I/O bound server, so 60% time waiting for I/O

$$\begin{aligned} \text{Speedup}_{\text{overall}} &= \frac{1}{(1 - \text{Fraction}_{\text{enhanced}}) + \frac{\text{Fraction}_{\text{enhanced}}}{\text{Speedup}_{\text{enhanced}}}} \\ &= \frac{1}{(1 - 0.4) + \frac{0.4}{10}} = \frac{1}{0.64} = 1.56 \end{aligned}$$

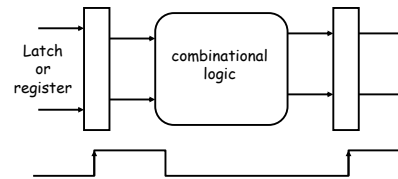
- Apparently, its human nature to be attracted by 10X faster, vs. keeping in perspective its just 1.6X faster

5) Processor performance equation

$$\text{CPU time} = \frac{\text{Seconds}}{\text{Program}} = \frac{\text{Instructions}}{\text{Program}} \times \frac{\text{Cycles}}{\text{Instruction}} \times \frac{\text{Seconds}}{\text{Cycle}}$$

	Inst Count	CPI	Clock Rate
Program	X		
Compiler	X	(X)	
Inst. Set.	X	X	
Organization		X	X
Technology			X

What is a Clock Cycle?



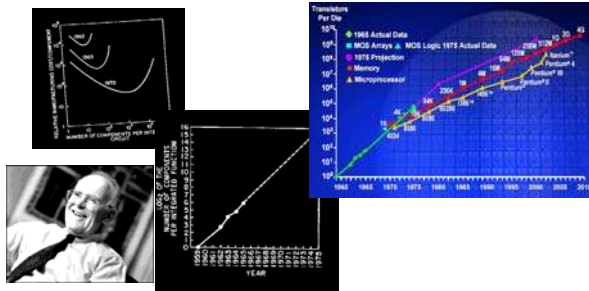
- Old days: 10 levels of gates
- Today: determined by numerous time-of-flight issues + gate delays
 - clock propagation, wire lengths, drivers

Break

Outline

- Technology Trends: Culture of tracking, anticipating and exploiting advances in technology
- Careful, quantitative comparisons:
 1. Define, quantify, and summarize relative performance
 2. Define and quantify relative cost
 3. Define and quantify dependability
 4. Define and quantify power

Moore's Law: 2X transistors / "year"



- "Cramming More Components onto Integrated Circuits"
 - Gordon Moore, Electronics, 1965
- # on transistors / cost-effective integrated circuit double every N months ($12 \leq N \leq 24$)

Tracking Technology Performance Trends

- Drill down into 4 technologies:
 - Disks
 - Memory
 - Network
 - Processors
- Compare ~1980 Archaic vs. ~2000 Modern
 - Performance Milestones in each technology
- Compare for Bandwidth vs. Latency improvements in performance over time
- Bandwidth: number of events per unit time
 - E.g., Mbits / second over network, Mbytes / second from disk
- Latency: elapsed time for a single event
 - E.g., one-way network delay in microseconds, average disk access time in milliseconds

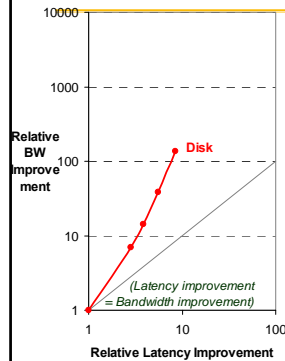
Disks Archaic

- CDC Wren I, 1983
- 3600 RPM
- 0.03 GBytes capacity
- Tracks/Inch: 800
- Bits/Inch: 9550
- Three 5.25" platters
- Bandwidth: 0.6 MBytes/sec
- Latency: 48.3 ms
- Cache: none

Modern

- Seagate 373453, 2003
- 15000 RPM (4X)
- 73.4 GBytes (2500X)
- Tracks/Inch: 64000 (80X)
- Bits/Inch: 533,000 (60X)
- Four 2.5" platters (in 3.5" form factor)
- Bandwidth: 86 MBytes/sec (140X)
- Latency: 5.7 ms (8X)
- Cache: 8 MBytes

Latency Lags Bandwidth (for last ~20 years)



Performance Milestones

- Disk: 3600, 5400, 7200, 10000, 15000 RPM (8x, 143x)
(latency = simple operation w/o contention
BW = best-case)

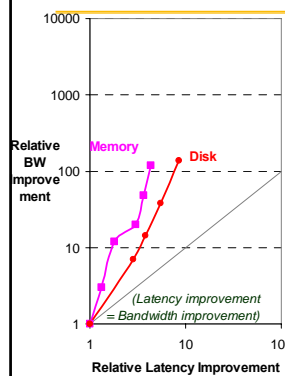
Memory Archaic

- 1980 DRAM (asynchronous)
- 0.06 Mbits/chip
- 64,000 xtors, 35 mm²
- 16-bit data bus per module, 16 pins/chip
- 13 Mbytes/sec
- Latency: 225 ns
- (no block transfer)

Modern

- 2000 Double Data Rate Synchr. (clocked) DRAM
- 256.00 Mbits/chip (4000X)
- 256,000,000 xtors, 204 mm²
- 64-bit data bus per DIMM, 66 pins/chip
- 1600 Mbytes/sec (120X)
- Latency: 52 ns (4X)
- Block transfers (page mode)

Latency Lags Bandwidth (last ~20 years)



Performance Milestones

- Memory Module: 16bit plain DRAM, Page Mode DRAM, 32b, 64b, SDRAM, DDR SDRAM (4x, 120x)
- Disk: 3600, 5400, 7200, 10000, 15000 RPM (8x, 143x)
(latency = simple operation w/o contention
BW = best-case)

LANs

Archaic	Modern
<ul style="list-style-type: none"> Ethernet 802.3 Year of Standard: 1978 10 Mbits/s link speed Latency: 3000 μsec Shared media Coaxial cable 	<ul style="list-style-type: none"> Ethernet 802.3ae Year of Standard: 2003 10,000 Mbits/s (1000X) link speed Latency: 190 μsec (15X) Switched media Category 5 copper wire

"Cat 5" is 4 twisted pairs in bundle

Coaxial Cable:

Twisted Pair:

Copper, 1mm thick, twisted to avoid antenna effect

Latency Lags Bandwidth (last ~20 years)

Relative BW Improvement (y-axis, log scale 1 to 10000)

Relative Latency Improvement (x-axis, log scale 1 to 100)

Legend: Memory (pink), Disk (red), Network (green)

(Latency improvement = Bandwidth improvement)

- Performance Milestones**
- Ethernet:** 10Mb, 100Mb, 1000Mb, 10000 Mb/s (16x, 1000x)
- Memory Module:** 16bit plain DRAM, Page Mode DRAM, 32b, 64b, SDRAM, DDR SDRAM (4x, 120x)
- Disk:** 3600, 5400, 7200, 10000, 15000 RPM (8x, 143x)

(latency = simple operation w/o contention
BW = best-case)

CPUs

Archaic	Modern
<ul style="list-style-type: none"> 1982 Intel 80286 12.5 MHz 2 MIPS (peak) Latency 320 ns 134,000 xtors, 47 mm² 16-bit data bus, 68 pins Microcode interpreter, separate FPU chip (no caches) 	<ul style="list-style-type: none"> 2001 Intel Pentium 4 1500 MHz (120X) 4500 MIPS (peak) (2250X) Latency 15 ns (20X) 42,000,000 xtors, 217 mm² 64-bit data bus, 423 pins 3-way superscalar, Dynamic translate to RISC, Superpipelined (22 stage), Out-of-Order execution On-chip 8KB Data caches, 96KB Instr. Trace cache, 256KB L2 cache

Latency Lags Bandwidth (last ~20 years)

Relative BW Improvement (y-axis, log scale 1 to 10000)

Relative Latency Improvement (x-axis, log scale 1 to 100)

Legend: Processor (blue), Memory (pink), Disk (red), Network (green)

(Latency improvement = Bandwidth improvement)

- Performance Milestones**
- Processor:** '286, '386, '486, Pentium, Pentium Pro, Pentium 4 (21x, 2250x)
- Ethernet:** 10Mb, 100Mb, 1000Mb, 10000 Mb/s (16x, 1000x)
- Memory Module:** 16bit plain DRAM, Page Mode DRAM, 32b, 64b, SDRAM, DDR SDRAM (4x, 120x)
- Disk:** 3600, 5400, 7200, 10000, 15000 RPM (8x, 143x)

CPU high, Memory low ("Memory Wall")

Rule of Thumb for Latency Lagging BW

- In the time that bandwidth doubles, latency improves by no more than a factor of 1.2 to 1.4 (and capacity improves faster than bandwidth)
- Stated alternatively: **Bandwidth improves by more than the square of the improvement in Latency**

6 Reasons Latency Lags Bandwidth

- Moore's Law helps BW more than latency
 - Faster transistors, more transistors, more pins help Bandwidth
 - MPU Transistors: 0.130 vs. 42 M xtors (300X)
 - DRAM Transistors: 0.064 vs. 256 M xtors (4000X)
 - MPU Pins: 68 vs. 423 pins (6X)
 - DRAM Pins: 16 vs. 66 pins (4X)
 - Smaller, faster transistors but communicate over (relatively) longer lines: limits latency
 - Feature size: 1.5 to 3 vs. 0.18 micron (8X, 17X)
 - MPU Die Size: 35 vs. 204 mm² (ratio sqrt \Rightarrow 2X)
 - DRAM Die Size: 47 vs. 217 mm² (ratio sqrt \Rightarrow 2X)

6 Reasons Latency Lags Bandwidth (cont'd)

2. Distance limits latency

- Size of DRAM block \Rightarrow long bit and word lines
 \Rightarrow most of DRAM access time
- Speed of light and computers on network
- 1. & 2. explains linear latency vs. square BW?

3. Bandwidth easier to sell ("bigger=better")

- E.g., 10 Gbits/s Ethernet ("10 Gig") vs. 10 μ sec latency Ethernet
- 4400 MB/s DIMM ("PC4400") vs. 50 ns latency
- Even if just marketing, customers now trained
- Since bandwidth sells, more resources thrown at bandwidth, which further tips the balance

6 Reasons Latency Lags Bandwidth (cont'd)

4. Latency helps BW, but not vice versa

- Spinning disk faster improves both bandwidth and rotational latency
 - » 3600 RPM \Rightarrow 15000 RPM = 4.2X
 - » Average rotational latency: 8.3 ms \Rightarrow 2.0 ms
 - » Things being equal, also helps BW by 4.2X
- Lower DRAM latency \Rightarrow More access/second (higher bandwidth)
- Higher linear density helps disk BW (and capacity), but not disk Latency
 - » 9,550 BPI \Rightarrow 533,000 BPI \Rightarrow 60X in BW

6 Reasons Latency Lags Bandwidth (cont'd)

5. Bandwidth hurts latency

- Queues help Bandwidth, hurt Latency (Queuing Theory)
- Adding chips to widen a memory module increases Bandwidth but higher fan-out on address lines may increase Latency

6. Operating System overhead hurts Latency more than Bandwidth

- Long messages amortize overhead; overhead bigger part of short messages

Summary of Technology Trends

- **For disk, LAN, memory, and microprocessor, bandwidth improves by square of latency improvement**
 - In the time that bandwidth doubles, latency improves by no more than 1.2X to 1.4X
- **Lag probably even larger in real systems, as bandwidth gains multiplied by replicated components**
 - Multiple processors in a cluster or even in a chip
 - Multiple disks in a disk array
 - Multiple memory modules in a large memory
 - Simultaneous communication in switched LAN
- **HW and SW developers should innovate assuming Latency Lags Bandwidth**
 - If everything improves at the same rate, then nothing really changes
 - When rates vary, require real innovation

Outline

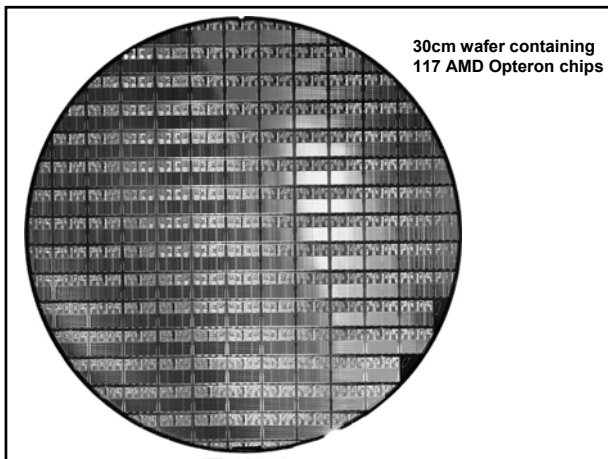
- Technology Trends: Culture of tracking, anticipating and exploiting advances in technology
- **Careful, quantitative comparisons:**
 1. Define and quantify cost
 2. Define and quantify power
 3. Define and quantify dependability
 4. Define, quantify, and summarize relative performance

Define and quantify cost (1/3)

Three factors lower cost:

1. **Learning curve** – manufacturing costs decrease over time, measured by change in **yield**
 - % manufactured devices that survives the testing procedure
2. **Volume** – doubling volume cuts cost 10%
 - Decrease time to get down the learning curve
 - Increases purchasing and manufacturing efficiency
 - Amortizes development costs over more devices
3. **Commodities** reduce costs by reducing margins
 - Products sold by multiple vendors in large volumes that essentially identical
 - E.g. keyboards, monitors, DRAMs, disks, PCs

Most of computer cost in Integrated Circuits (ICs)



Define and quantify cost: ICs (2/3)

$$\text{IC cost} = \frac{\text{Die cost} + \text{Testing cost} + \text{Packaging cost}}{\text{Final test yields}}$$

$$\text{Die cost} = \frac{\text{Wafer cost}}{\text{Dies per wafer} \times \text{Die yield}}$$

$$\text{Dies per wafer} = \frac{\pi \times (\text{Wafer diameter} / 2)^2}{\text{Die area}} = \frac{\pi \times \text{Wafer diameter}}{\sqrt{2} \times \text{Die area}}$$

$$\text{Die yield} = \text{Wafer yield} \times \left(1 + \frac{\text{Defect density} \times \text{Die area}}{\alpha} \right)^{-\alpha}$$

In 2006: $\alpha = 4.0$
Defect density = $0.4/\text{cm}^2$
30cm wafer \approx \$5k-\$6k

For cost effective dies:
cost $\approx f(\text{die_area}^2)$

Define and quantify cost: cost vs. price (3/3)

- **Margin** = price product sells – cost to manufacture
- Margins pay for a research and development (R&D), marketing, sales, manufacturing equipment maintenance, building rental, cost of financing, pretax profits, and taxes.
- Most companies spend 4% (commodity PC business) to 12% (high-end server business) of income on R&D, which includes all engineering.

Outline

- **Technology Trends: Culture of tracking, anticipating and exploiting advances in technology**
- **Careful, quantitative comparisons:**
 1. Define and quantify cost
 2. Define and quantify power
 3. Define and quantify dependability
 4. Define, quantify, and summarize relative performance

Define and quantify power (1/2)

- For CMOS chips, traditional dominant energy consumption has been in switching transistors, called **dynamic power**

$$\text{Power}_{\text{dynamic}} = \frac{1}{2} \times \text{CapacitiveLoad} \times \text{Voltage}^2 \times \text{FrequencySwitched}$$

- For mobile devices, energy better metric

$$\text{Energy}_{\text{dynamic}} = \text{CapacitiveLoad} \times \text{Voltage}^2$$

- For a fixed task, slowing clock rate (frequency switched) reduces power, but not energy
- Capacitive load a function of number of transistors connected to output and technology, which determines capacitance of wires and transistors
- Dropping voltage helps both, so went from 5V to 1V
- To save energy & dynamic power, most CPUs now turn off clock of inactive modules (e.g. Fl. Pt. Unit)

Example of quantifying power

- Suppose 15% reduction in voltage results in a 15% reduction in frequency. What is impact on dynamic power?

$$\begin{aligned} \text{Power}_{\text{dynamic}} &= 1/2 \times \text{CapacitiveLoad} \times \text{Voltage}^2 \times \text{FrequencySwitched} \\ &= 1/2 \times .85 \times \text{CapacitiveLoad} \times (.85 \times \text{Voltage})^2 \times \text{FrequencySwitched} \\ &= (.85)^3 \times \text{OldPower}_{\text{dynamic}} \\ &\approx 0.6 \times \text{OldPower}_{\text{dynamic}} \end{aligned}$$

Define and quantify power (2/2)

- Because leakage current flows even when a transistor is off, now **static power** important too

$$\text{Power}_{\text{static}} = \text{Current}_{\text{static}} \times \text{Voltage}$$

- Leakage current increases in processors with smaller transistor sizes
- Increasing the number of transistors increases power even if they are turned off
- In 2006, goal for leakage is 25% of total power consumption; high performance designs at 40%
- Very low power systems even gate voltage to inactive modules to control loss due to leakage

Outline

- Review
- Technology Trends: Culture of tracking, anticipating and exploiting advances in technology
- Careful, quantitative comparisons:
 - Define and quantify relative cost
 - Define and quantify power
 - Define and quantify dependability
 - Define, quantify, and summarize relative performance

Define and quantify dependability (1/3)

- How decide when a system is operating properly?
- Infrastructure providers now offer Service Level Agreements (SLA) to guarantee that their networking or power service would be dependable
- Systems alternate between 2 states of service with respect to an SLA:
 - Service accomplishment, where the service is delivered as specified in SLA
 - Service interruption, where the delivered service is different from the SLA
- Failure = transition from state 1 to state 2
- Restoration = transition from state 2 to state 1

Define and quantify dependability (2/3)

- Module reliability** = measure of continuous service accomplishment (or time to failure).
Two metrics:
 - Mean Time To Failure (MTTF) measures Reliability
 - Failures In Time (FIT) = 1/MTTF, the rate of failures
- Mean Time To Repair (MTTR)** measures Service Interruption
 - Mean Time Between Failures (MTBF) = MTTF+MTTR
- Module availability** measures service as alternate between the 2 states of accomplishment and interruption (number between 0 and 1, e.g. 0.9)
 - Module availability = $MTTF / (MTTF + MTTR)$

Example calculating reliability

- If modules have **exponentially distributed lifetimes** (age of module does not affect probability of failure), overall failure rate is the sum of failure rates of the modules
- Calculate FIT and MTTF for 10 disks (1M hour MTTF per disk), 1 disk controller (0.5M hour MTTF), and 1 power supply (0.2M hour MTTF):

$$\begin{aligned} \text{FailureRate} &= 10 \times (1/1,000,000) + 1/500,000 + 1/200,000 \\ &= (10 + 2 + 5) / 1,000,000 \\ &= 17 / 1,000,000 \\ &= 17,000 \text{ FIT} \\ \text{MTTF} &= 1,000,000,000 / 17,000 \\ &\approx 59,000 \text{ hours} \end{aligned}$$

And in conclusion ...

- Computer Architecture >> instruction sets
- Computer Architecture skill sets are different
 - Quantitative principles of design
 - Quantitative approach to design
 - Solid interfaces that really work
 - Technology tracking and anticipation
- Computer Science at the crossroads from sequential to parallel computing
 - Salvation requires innovation in many fields, including computer architecture
- Tracking and extrapolating technology part of architect's responsibility
- Expect Bandwidth in disks, DRAM, network, and processors to improve by at least as much as the square of the improvement in Latency
- Quantify dynamic and static power
 - Capacitance x Voltage² x frequency, Energy vs. power
- Quantify dependability
 - Reliability (MTTF, FIT), Availability (99.9...)

Reading

- This lecture: chapter 1
- Next lecture: appendix A
- Assignment 1: appendix B

Lecture 2 – Performance & Pipelining

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

Review from last lecture

- Tracking and extrapolating technology part of architect's responsibility
- Expect Bandwidth in disks, DRAM, network, and processors to improve by at least as much as the square of the improvement in Latency
- Quantify Cost (vs. Price)
 - $IC \approx f(\text{Area}^2) + \text{Learning curve, volume, commodity, margins}$
- Quantify dynamic and static power
 - Capacitance x Voltage² x frequency, Energy vs. power
- Quantify dependability
 - Reliability (MTTF vs. FIT), Availability (MTTF/(MTTF+MTTR))

Outline

- Quantify and summarize performance
 - Ratios, Geometric Mean, Multiplicative Standard Deviation
 - Fallacies & Pitfalls: Benchmarks age, disks fail, 1 point fail danger
- Pipelining
 - MIPS: an ISA for Pipelining
 - 5 stage pipelining
 - Structural and Data Hazards
 - Forwarding
 - Branch Schemes
 - Exceptions and Interrupts
- Conclusion

Definition: Performance

- Performance is in units of things per sec
 - bigger is better
- If we are primarily concerned with response time

$$\text{Performance}(X) = \frac{1}{\text{ExecutionTime}(X)}$$

"X is n times faster than Y" means

$$n = \frac{\text{Performance}(X)}{\text{Performance}(Y)} = \frac{\text{ExecutionTime}(Y)}{\text{ExecutionTime}(X)}$$

Performance: What to measure?

- Usually rely on benchmarks vs. real workloads
- To increase predictability, collections of benchmark applications, called *benchmark suites*, are popular
- **SPECCPU**: popular desktop benchmark suite
 - CPU only, split between integer and floating point programs
 - SPECCPU2006:
 - » Motto: "An ounce of honest data is worth a pound of marketing hype"
 - » 12 integer and 17 floating point programs
 - **SPECSFS** (NFS file server) and **SPECWeb** (WebServer) added as server benchmarks
- **Transaction Processing Council** measures server performance and cost-performance for databases
 - **TPC-C** Complex query for Online Transaction Processing
 - **TPC-H** models ad hoc decision support
 - **TPC-W** a transactional web benchmark
 - **TPC-App** application server and web services benchmark

How Summarize Suite Performance (1/5)

- Arithmetic average of execution time of all programs?
 - But they vary by 4X in speed, so some would be more important than others in arithmetic average
- Could add a weight per program, but how pick a weight?
 - Different companies want different weights for their products
- **SPECRatio**: Normalize execution times to reference computer, yielding a ratio proportional to

$$\text{Performance} = \frac{\text{time on reference computer}}{\text{time on computer rated}}$$

How Summarize Suite Performance (2/5)

- If program SPECratio on Computer A is 1.25 times bigger than Computer B, then

$$1.25 = \frac{SPECratio_A}{SPECratio_B} = \frac{\left(\frac{ExecutionTime_{reference}}{ExecutionTime_A}\right)}{\left(\frac{ExecutionTime_{reference}}{ExecutionTime_B}\right)}$$

$$= \frac{ExecutionTime_B}{ExecutionTime_A} = \frac{Performance_A}{Performance_B}$$

- Note that when comparing 2 computers as a ratio, execution times on the reference computer drop out, so choice of reference computer is irrelevant

How Summarize Suite Performance (3/5)

- Since ratios, proper mean is geometric mean (SPECratio unitless, so arithmetic mean meaningless)

$$GeometricMean = \sqrt[n]{\prod_{i=1}^n SPECratio_i}$$

- Geometric mean of the ratios is the same as the ratio of the geometric means
- Ratio of geometric means = Geometric mean of performance ratios
 ⇒ choice of reference computer is irrelevant!

These two points make geometric mean of ratios attractive to summarize performance

How Summarize Suite Performance (4/5)

- Does a single mean well summarize performance of programs in benchmark suite?
- Can decide if mean a good predictor by characterizing variability of distribution using standard deviation
- Like geometric mean, geometric standard deviation is multiplicative rather than arithmetic
- Can simply take the logarithm of SPECratios, compute the standard mean and standard deviation, and then take the exponent to convert back:

$$GeometricMean = \exp\left(\frac{1}{n} \times \sum_{i=1}^n \ln(SPECratio_i)\right)$$

$$GeometricStDev = \exp(StDev(\ln(SPECratio_i)))$$

How Summarize Suite Performance (5/5)

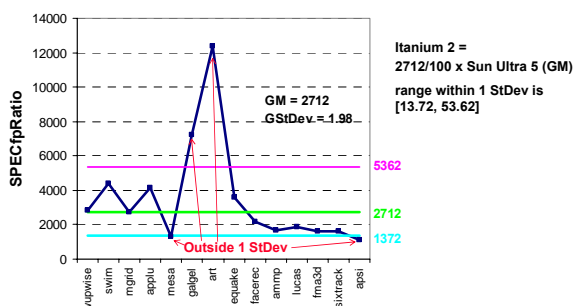
- Standard deviation is more informative if know distribution has a standard form
 - bell-shaped normal distribution, whose data are symmetric around mean
 - lognormal distribution, where logarithms of data – not data itself – are normally distributed (symmetric) on a logarithmic scale
- For a lognormal distribution, we expect that

68% of samples fall in range
 $[mean / gstddev, mean \times gstddev]$

95% of samples fall in range
 $[mean / gstddev^2, mean \times gstddev^2]$

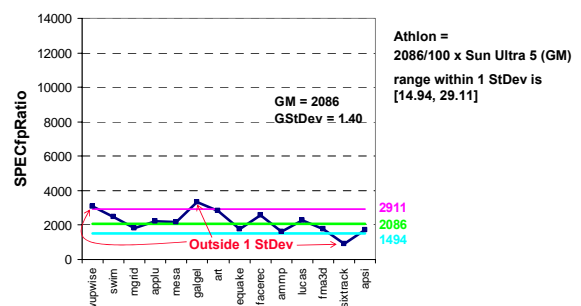
Example Standard Deviation (1/3)

- GM and multiplicative StDev of SPECfp2000 for Itanium 2

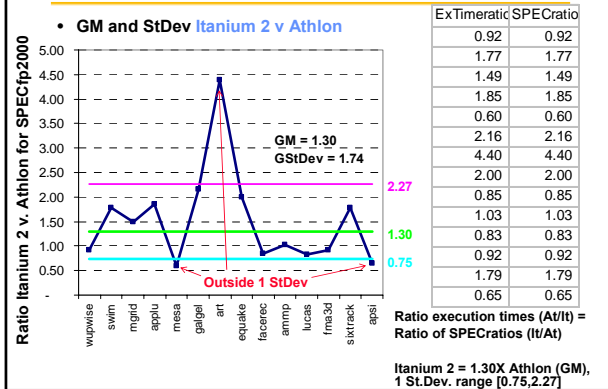


Example Standard Deviation (2/3)

- GM and multiplicative StDev of SPECfp2000 for AMD Athlon



Example Standard Deviation (3/3)



Comments on Itanium 2 and Athlon

- Standard deviation for SPECRatio of 1.98 for Itanium 2 is much higher –vs. 1.40– so results will differ more widely from the mean, and therefore are likely less predictable
- SPECratios falling within one standard deviation:
 - 10 of 14 benchmarks (71%) for Itanium 2
 - 11 of 14 benchmarks (78%) for Athlon
- Thus, results are quite compatible with a lognormal distribution (expect 68% for 1 StDev)
- Itanium 2 vs. Athlon St.Dev is 1.74, which is high, so less confidence in claim that Itanium 1.30 times as fast as Athlon
 - Indeed, Athlon faster on 6 of 14 programs
- Range is [0.75,2.27] with 11/14 inside 1 StDev (78%)

Fallacies and Pitfalls (1/2)

- **Fallacies - commonly held misconceptions**
 - When discussing a fallacy, we try to give a counterexample.
- **Pitfalls - easily made mistakes.**
 - Often generalizations of principles true in limited context

Show Fallacies and Pitfalls to help you avoid these errors
- **Fallacy: Benchmarks remain valid indefinitely**
 - Once a benchmark becomes popular, tremendous pressure to improve performance by targeted optimizations or by aggressive interpretation of the rules for running the benchmark: “benchmarkmanship.”
 - 70 benchmarks from the 5 SPEC releases. 70% were dropped from the next release since no longer useful
- **Pitfall: A single point of failure**
 - Rule of thumb for fault tolerant systems: make sure that every component was redundant so that no single component failure could bring down the whole system (e.g., power supply)

Fallacies and Pitfalls (2/2)

- **Fallacy - Rated MTTF of disks is 1,200,000 hours or ≈ 140 years, so disks practically never fail**
- **But disk lifetime is 5 years ⇒ replace a disk every 5 years; on average, 28 replacements wouldn't fail**
- **A better unit: % that fail (1.2M MTTF = 833 FIT)**
- **Fail over lifetime: if had 1000 disks for 5 years**
 $= 1000 * (5 * 365 * 24) * 833 / 10^9 = 36,485,000 / 10^6 = 37$
 $= 3.7\% (37/1000)$ fail over 5 yr lifetime (1.2M hr MTTF)
- **But this is under pristine conditions**
 - little vibration, narrow temperature range ⇒ no power failures
- **Real world:**
 - 3% to 6% of SCSI drives fail per year
 - » 3400 - 6800 FIT or 150,000 - 300,000 hour MTTF [Gray & van Ingen 05]
 - 3% to 7% of ATA drives fail per year
 - » 3400 - 8000 FIT or 125,000 - 300,000 hour MTTF [Gray & van Ingen 05]

Outline

- **Quantify and summarize performance**
 - Ratios, Geometric Mean, Multiplicative Standard Deviation
 - Fallacies & Pitfalls: Benchmarks age, disks fail, 1 point fail danger
- **Pipelining**
 - MIPS: an ISA for Pipelining
 - 5 stage pipelining
 - Structural and Data Hazards
 - Forwarding
 - Branch Schemes
 - Exceptions and Interrupts
- **Conclusion**

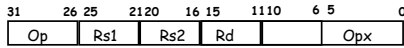
A "Typical" RISC ISA

- 32-bit fixed format instruction (3 formats)
- 32 32-bit GPR (R0 contains zero, DP take pair)
- 3-address, reg-reg arithmetic instruction
- Single address mode for load/store: base + displacement
 - no indirection
- Simple branch conditions
- Delayed branch

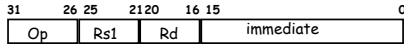
see: SPARC, MIPS, HP PA-Risc, DEC Alpha, IBM PowerPC, CDC 6600, CDC 7600, Cray-1, Cray-2, Cray-3

Example: MIPS

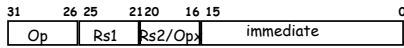
Register-Register



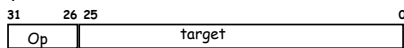
Register-Immediate



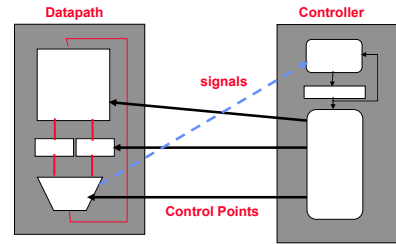
Branch



Jump / Call



Datapath vs Control



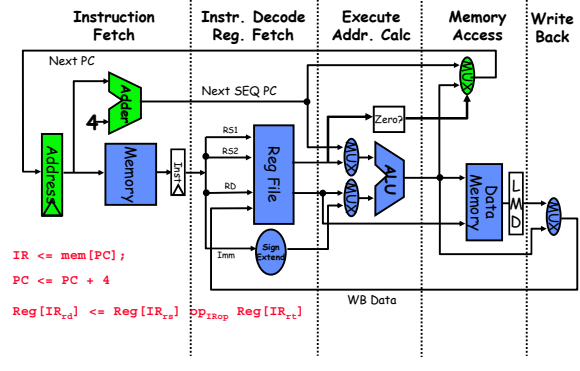
- **Datapath:** Storage, FU, interconnect sufficient to perform the desired functions
 - Inputs are Control Points
 - Outputs are signals
- **Controller:** State machine to orchestrate operation on the data path
 - Based on desired function and signals

Approaching an ISA

- **Instruction Set Architecture**
 - Defines set of operations, instruction format, hardware supported data types, named storage, addressing modes, sequencing
- **Meaning of each instruction is described by the register transfer language (RTL) on architected registers and memory**
- **Given technology constraints assemble adequate datapath**
 - Architected storage mapped to actual storage
 - Function units to do all the required operations
 - Possible additional storage (eg. MAR, MBR, ...)
 - Interconnect to move information among regs and FUs
- **Map each instruction to sequence of RTLs**
- **Collate sequences into symbolic controller state transition diagram (STD)**
- **Lower symbolic STD to control points**
- **Implement controller**

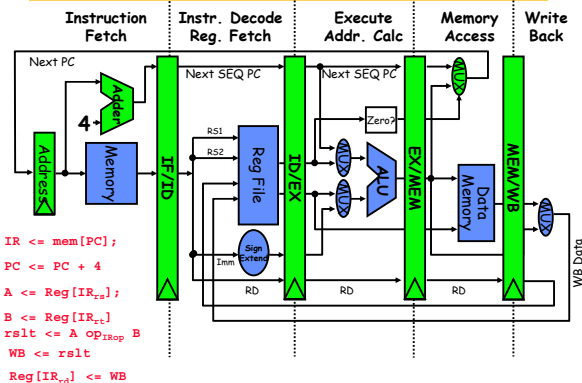
5 Steps of MIPS Datapath

Figure A.2, Page A-8

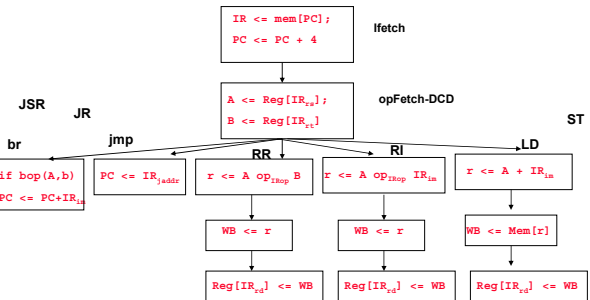


5 Steps of MIPS Datapath

Figure A.3, Page A-9

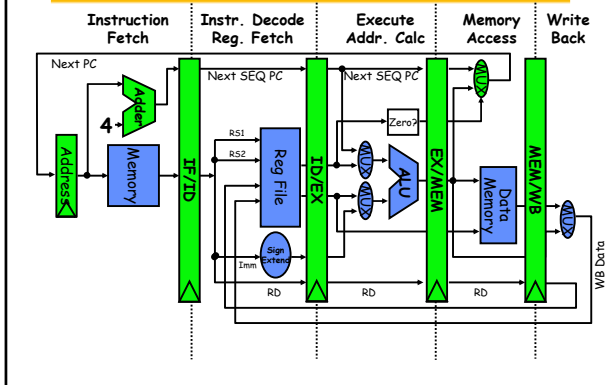


Inst. Set Processor Controller



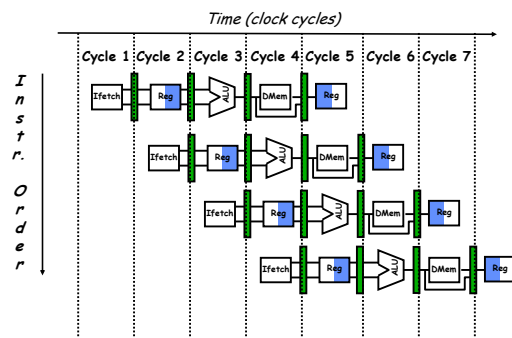
5 Steps of MIPS Datapath

Figure A.3, Page A-9



Visualizing Pipelining

Figure A.3, Page A-9

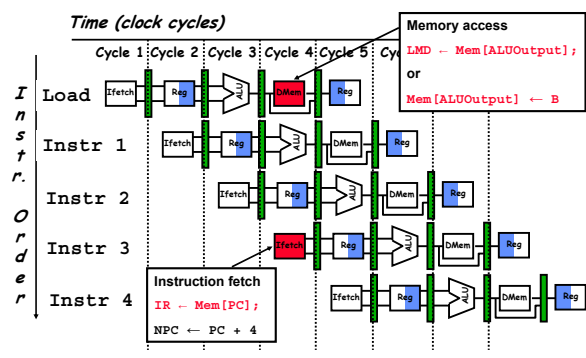


Pipelining is not quite that easy!

- Limits to pipelining: **Hazards** prevent next instruction from executing during its designated clock cycle
 - **Structural hazards:** HW cannot support this combination of instructions (single person to fold and put clothes away)
 - **Data hazards:** Instruction depends on result of prior instruction still in the pipeline (missing sock)
 - **Control hazards:** Caused by delay between the fetching of instructions and decisions about changes in control flow (branches and jumps).

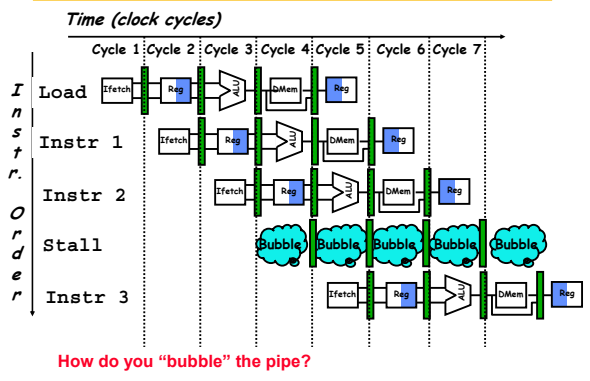
One Memory Port/Structural Hazards

Figure A.4, Page A-14



One Memory Port/Structural Hazards

(Similar to Figure A.5, Page A-15)



Speed Up Equation for Pipelining

$$\text{Speedup} = \frac{\text{Average instruction time unpipelined}}{\text{Average instruction time pipelined}} = \frac{CPI_{\text{unpipelined}} \times \text{Cycle Time}_{\text{unpipelined}}}{CPI_{\text{pipelined}} \times \text{Cycle Time}_{\text{pipelined}}}$$

$$CPI_{\text{pipelined}} = \text{Ideal CPI} + \text{Pipeline stall cycles per instruction}$$

$$CPI_{\text{pipelined}} = \frac{\text{Clock cycle unpipelined}}{\text{Pipeline depth}}$$

$$\text{Speedup} = \frac{\text{Ideal CPI} \times \text{Pipeline depth}}{\text{Ideal CPI} + \text{Pipeline stall CPI}} \times \frac{\text{Cycle Time}_{\text{unpipelined}}}{\text{Cycle Time}_{\text{pipelined}}}$$

For simple RISC pipeline, Ideal CPI = 1:

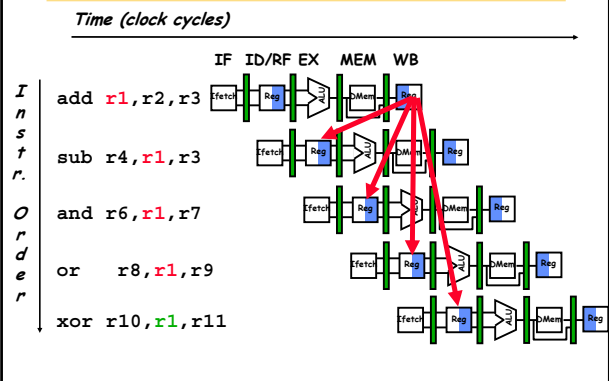
$$\text{Speedup} = \frac{\text{Pipeline depth}}{1 + \text{Pipeline stall CPI}} \times \frac{\text{Cycle Time}_{\text{unpipelined}}}{\text{Cycle Time}_{\text{pipelined}}}$$

Example: Dual-port vs. Single-port

- Machine A: Dual ported memory (“Harvard Architecture”)
- Machine B: Single ported memory, but its pipelined implementation has a 1.05 times faster clock rate
- Ideal CPI = 1 for both
- Loads are 40% of instructions executed
 - $SpeedUp_A = \text{Pipeline Depth} / (1 + 0) \times (\text{clock}_{unpipe} / \text{clock}_{pipe}) = \text{Pipeline Depth}$
 - $SpeedUp_B = \text{Pipeline Depth} / (1 + 0.4 \times 1) \times (\text{clock}_{unpipe} / (\text{clock}_{unpipe} / 1.05)) = (\text{Pipeline Depth} / 1.4) \times 1.05 = 0.75 \times \text{Pipeline Depth}$
 - $SpeedUp_A / SpeedUp_B = \text{Pipeline Depth} / (0.75 \times \text{Pipeline Depth}) = 1.33$
- Machine A is 1.33 times faster

Data Hazard on R1

Figure A.6, Page A-17



Three Generic Data Hazards

- Read After Write (RAW)**
Instr_j tries to read operand before Instr_i writes it
- ```

 I: add r1, r2, r3
 J: sub r4, r1, r3

```
- Caused by a “**Dependence**” (in compiler nomenclature). This hazard results from an actual need for communication.

### Three Generic Data Hazards

- Write After Read (WAR)**  
Instr<sub>j</sub> writes operand *before* Instr<sub>i</sub> reads it
- ```

    I: sub r4, r1, r3
    J: add r1, r2, r3
    K: mul r6, r1, r7
    
```
- Called an “**anti-dependence**” by compiler writers. This results from reuse of the name “r1”.
 - Can’t happen in MIPS 5 stage pipeline because:
 - All instructions take 5 stages, and
 - Reads are always in stage 2, and
 - Writes are always in stage 5

Three Generic Data Hazards

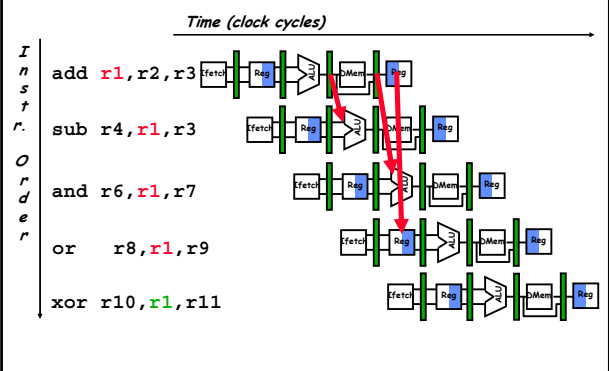
- Write After Write (WAW)**
Instr_j writes operand *before* Instr_i writes it.
- ```

 I: sub r1, r4, r3
 J: add r1, r2, r3
 K: mul r6, r1, r7

```
- Called an “**output dependence**” by compiler writers. This also results from the reuse of name “r1”.
  - Can’t happen in MIPS 5 stage pipeline because:
    - All instructions take 5 stages, and
    - Writes are always in stage 5
  - Will see WAR and WAW in more complicated pipes

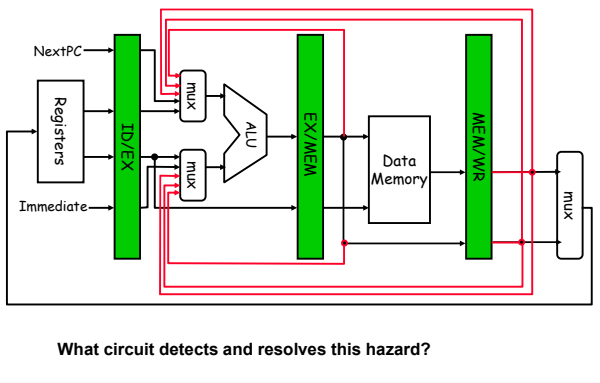
### Forwarding to Avoid Data Hazard

Figure A.7, Page A-19



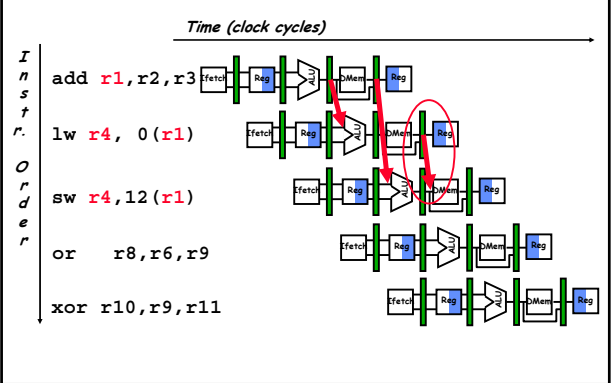
## HW Change for Forwarding

Figure A.23, Page A-37



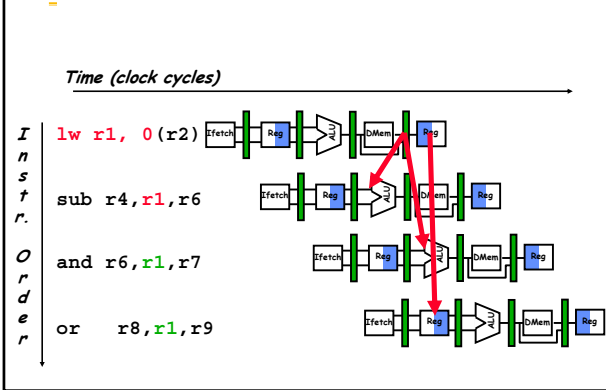
## Forwarding to Avoid LW-SW Data Hazard

Figure A.8, Page A-20



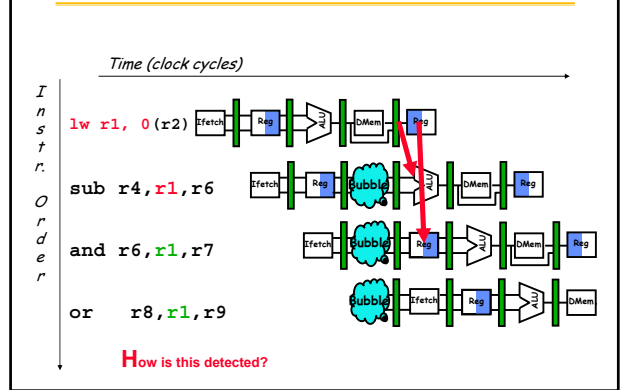
## Data Hazard Even with Forwarding

Figure A.9, Page A-20



## Data Hazard Even with Forwarding

(Similar to Figure A.10, Page A-21)



## Software Scheduling to Avoid Load Hazards

Try producing fast code for

$a = b + c;$

$d = e - f;$

assuming a, b, c, d, e, and f in memory.

Slow code:

LW Rb,b

LW Rc,c

ADD Ra,Rb,Rc

SW a,Ra

LW Re,e

LW Rf,f

SUB Rd,Re,Rf

SW d,Rd

Fast code:

LW Rb,b

LW Rc,c

LW Re,e

ADD Ra,Rb,Rc

LW Rf,f

SW a,Ra

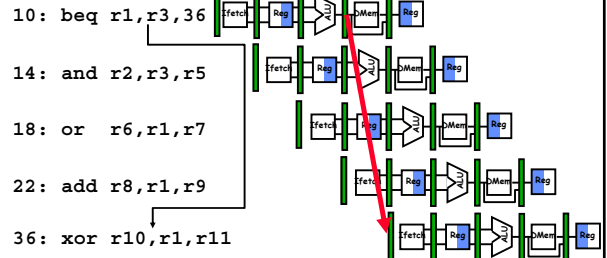
SUB Rd,Re,Rf

SW d,Rd

Compiler optimizes for performance. Hardware checks for safety.

## Control Hazard on Branches

### Three Stage Stall



What do you do with the 3 instructions in between?

How do you do it?

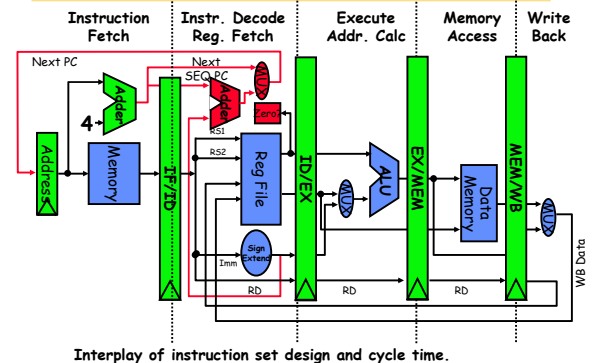
Where is the "commit"?

## Branch Stall Impact

- If CPI = 1, 30% branch, Stall 3 cycles => new CPI = 1.9!
- Two part solution:
  - Determine branch taken or not sooner, AND
  - Compute taken branch address earlier
- MIPS branch tests if register = 0 or ≠ 0
- MIPS Solution:
  - Move Zero test to ID/RF stage
  - Adder to calculate new PC in ID/RF stage
  - 1 clock cycle penalty for branch versus 3

## Pipelined MIPS Datapath

Figure A.24, page A-38



## Four Branch Hazard Alternatives

- #1: Stall until branch direction is clear
- #2: Predict Branch Not Taken
  - Execute successor instructions in sequence
  - “Squash” instructions in pipeline if branch actually taken
  - Advantage of late pipeline state update
  - 47% MIPS branches not taken on average
  - PC+4 already calculated, so use it to get next instruction
- #3: Predict Branch Taken
  - 53% MIPS branches taken on average
  - **But haven't calculated branch target address in MIPS**
    - » MIPS still incurs 1 cycle branch penalty
    - » Other machines: branch target known before outcome

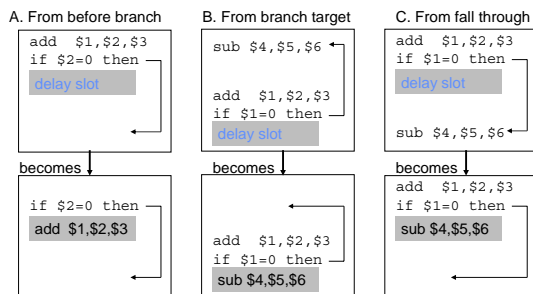
## Four Branch Hazard Alternatives

### #4: Delayed Branch

- Define branch to take place **AFTER** a following instruction
- ```

branch instruction
sequential successor1
sequential successor2
.....
sequential successorn
branch target if taken
    
```
- Branch delay of length *n*
- 1 slot delay allows proper decision and branch target address in 5 stage pipeline
 - MIPS uses this

Scheduling Branch Delay Slots (Fig A.14)



- A is the best choice, fills delay slot & reduces instruction count (IC)
- In B, the sub instruction may need to be copied, increasing IC
- In B and C, must be okay to execute sub when branch fails

Delayed Branch

- **Compiler effectiveness for single branch delay slot:**
 - Fills about 60% of branch delay slots
 - About 80% of instructions executed in branch delay slots useful in computation
 - About 50% (60% x 80%) of slots usefully filled
- **Delayed Branch downside: As processor go to deeper pipelines and multiple issue, the branch delay grows and need more than one delay slot**
 - Delayed branching has lost popularity compared to more expensive but more flexible dynamic approaches
 - Growth in available transistors has made dynamic approaches relatively cheaper

Evaluating Branch Alternatives

$$\text{Pipeline speedup} = \frac{\text{Pipeline depth}}{1 + \text{Branch frequency} \times \text{Branch penalty}}$$

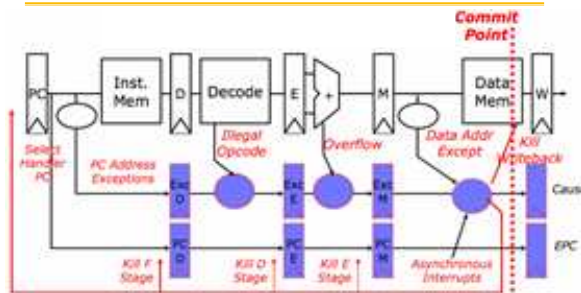
Assume 4% unconditional branch, 6% conditional branch-untaken, 10% conditional branch-taken

Scheduling scheme	Branch penalty	CPI	speedup v. unpipelined	speedup v. stall
Stall pipeline	3	1.60	3.1	1.0
Predict taken	1	1.20	4.2	1.33
Predict not taken	1	1.14	4.4	1.40
Delayed branch	0.5	1.10	4.5	1.45

Problems with Pipelining

- **Exception:** An unusual event happens to an instruction during its execution
 - Examples: divide by zero, undefined opcode
- **Interrupt:** Hardware signal to switch the processor to a new instruction stream
 - Example: a sound card interrupts when it needs more audio output samples (an audio "click" happens if it is left waiting)
- **Problem:** It must appear that the exception or interrupt must appear between 2 instructions (I_i and I_{i+1})
 - The effect of all instructions up to and including I_i is totalling complete
 - No effect of any instruction after I_i can take place
- The interrupt (exception) handler either aborts program or restarts at instruction I_{i+1}

Precise Exceptions in Static Pipelines



Key observation: architected state only change in memory and register write stages.

And In Conclusion:

- Quantify and summarize performance
 - Ratios, Geometric Mean, Multiplicative Standard Deviation
- F&P: Benchmarks age, disks fail, 1 point fail danger
- Control via **State Machines** and **Microprogramming**
- Just overlap tasks; easy if tasks are independent
- Speed Up \leq Pipeline Depth; if ideal CPI is 1, then:

$$\text{Speedup} = \frac{\text{Pipeline depth}}{1 + \text{Pipeline stall CPI}} \times \frac{\text{Cycle Time}_{\text{unpipelined}}}{\text{Cycle Time}_{\text{pipelined}}}$$
- Hazards limit performance on computers:
 - Structural: need more HW resources
 - Data (RAW,WAR,WAW): need forwarding, compiler scheduling
 - Control: delayed branch, prediction
- Exceptions, Interrupts add complexity

Reading

- This lecture: appendix A *Pipelining*
- Next lecture: appendix C *Memory Hierarchy*

Lecture 3 – Memory Hierarchy

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

Review from last lecture

- Quantify and summarize performance
 - Ratios, Geometric Mean, Multiplicative Standard Deviation
- F&P: Benchmarks age, disks fail, 1 point fail danger
- Control VIA State Machines and Microprogramming
- Just overlap tasks; easy if tasks are independent
- Speed Up \leq Pipeline Depth; if ideal CPI is 1, then:

$$\text{Speedup} = \frac{\text{Pipeline depth}}{1 + \text{Pipeline stall CPI}} \times \frac{\text{Cycle Time}_{\text{unpipelined}}}{\text{Cycle Time}_{\text{pipelined}}}$$

- Hazards limit performance on computers:
 - Structural: need more HW resources
 - Data (RAW,WAR,WAW): need forwarding, compiler scheduling
 - Control: delayed branch, prediction
- Exceptions, Interrupts add complexity

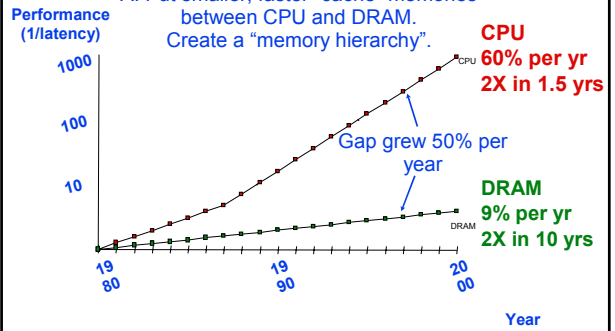
Outline

- Review
- Memory hierarchy
- Locality
- Cache design
- Virtual address spaces
- Page table layout
- TLB design options
- Conclusion

Since 1980, CPU has outpaced DRAM ...

Q. How do architects address this gap?

A. Put smaller, faster "cache" memories between CPU and DRAM. Create a "memory hierarchy".



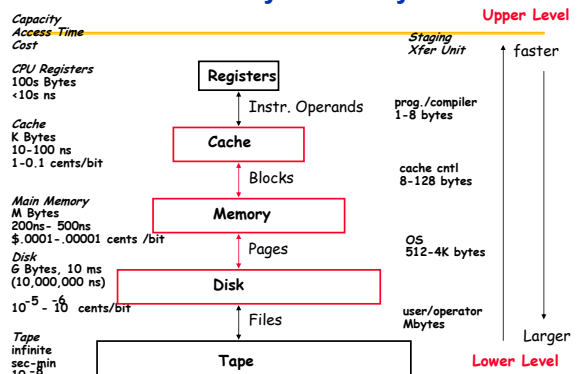
1977: DRAM faster than microprocessors

Apple II (1977)
CPU: 1000 ns
DRAM: 400 ns

Steve Jobs, Steve Wozniak

RAM Complement	Apple II System
4K	\$ 1,298.00
48K	2,838.00

Levels of the Memory Hierarchy




Memory Hierarchy: Apple iMac G5

Managed by compiler Managed by hardware Managed by OS, hardware, application

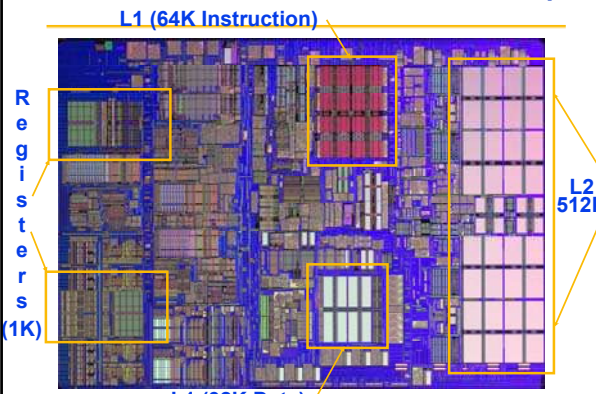
07	Reg	L1 Inst	L1 Data	L2	DRAM	Disk
Size	1K	64K	32K	512K	256M	80G
Latency Cycles, Time	1, 0.6 ns	3, 1.9 ns	3, 1.9 ns	11, 6.9 ns	88, 55 ns	10 ⁷ , 12 ms

iMac G5
1.6 GHz

Goal: Illusion of large, fast, cheap memory
Let programs address a memory space that scales to the disk size, at a speed that is usually as fast as register access



iMac's PowerPC 970: All caches on-chip



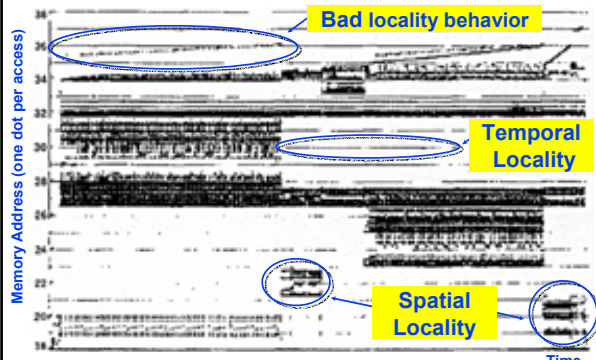
Registers (1K)

The Principle of Locality

- The Principle of Locality:**
 - Program access a relatively small portion of the address space at any instant of time.
- Two Different Types of Locality:**
 - Temporal Locality** (Locality in Time): If an item is referenced, it will tend to be referenced again soon (e.g., loops, reuse)
 - Spatial Locality** (Locality in Space): If an item is referenced, items whose addresses are close by tend to be referenced soon (e.g., straightline code, array access)
- Last 15 years, HW relied on locality for speed**

It is a property of programs which is exploited in machine design.

Programs with locality cache well ...



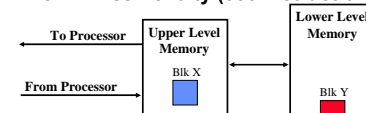
Memory Address (one dot per access)

Time

Donald J. Hatfield, Jeanette Gerald: Program Restructuring for Virtual Memory. IBM Systems Journal 10(3): 168-192 (1971)

Memory Hierarchy: Terminology

- Hit: data appears in some block in the upper level (example: Block X)**
 - Hit Rate:** the fraction of memory access found in the upper level
 - Hit Time:** Time to access the upper level which consists of RAM access time + Time to determine hit/miss
- Miss: data needs to be retrieve from a block in the lower level (Block Y)**
 - Miss Rate** = 1 - (Hit Rate)
 - Miss Penalty:** Time to replace a block in the upper level + Time to deliver the block the processor
- Hit Time << Miss Penalty (500 instructions on 21264!)**



Cache Measures

- Hit rate: fraction found in that level**
 - So high that usually talk about **Miss rate**
 - Miss rate fallacy: as MIPS to CPU performance, miss rate to average memory access time in memory
- Average memory-access time**
= Hit time + Miss rate x Miss penalty (ns or clocks)
- Miss penalty: time to replace a block from lower level, including time to replace in CPU**
 - access time:** time to lower level = f(latency to lower level)
 - transfer time:** time to transfer block = f(bandwidth between upper & lower levels)

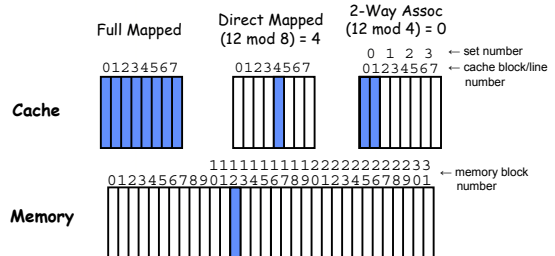
4 Questions for Memory Hierarchy

- Q1: Where can a block be placed in the upper level?
(Block placement)
- Q2: How is a block found if it is in the upper level?
(Block identification)
- Q3: Which block should be replaced on a miss?
(Block replacement)
- Q4: What happens on a write?
(Write strategy)

Q1: Where can a block be placed in the upper level?

- Block 12 placed in 8 block cache:

- Fully associative, direct mapped, set associative
- S.A. Mapping = Block Number modulo #Sets



Q2: How is a block found if it is in the upper level?

Block Address		Block Offset
Tag	Index	

- Tag on each block
 - No need to check index or block offset
- Increasing associativity shrinks index, expands tag

Q3: Which block should be replaced on a miss?

- Easy for Direct Mapped
- Set Associative or Fully Associative:
 - LRU (Least Recently Used); appealing, but hard to implement for high associativity
 - Random; easy to implement, how well does it work?

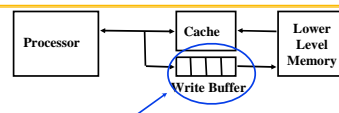
Assoc:	2-way		4-way		8-way	
Size	LRU	Ran	LRU	Ran	LRU	Ran
16 KB	5.2%	5.7%	4.7%	5.3%	4.4%	5.0%
64 KB	1.9%	2.0%	1.5%	1.7%	1.4%	1.5%
256 KB	1.15%	1.17%	1.13%	1.13%	1.12%	1.12%

Q4: What happens on a write?

	Write-Through	Write-Back
Policy	Data written to cache block also written to lower-level memory	Write data only to the cache Update lower level when a block falls out of the cache
Debug	Easy	Hard
Do read misses produce writes?	No	Yes
Do repeated writes make it to lower level?	Yes	No

Additional option: let writes to an un-cached address allocate a new cache line ("write-allocate").

Write Buffers for Write-Through Caches



Holds data awaiting write-through to lower level memory

- | | |
|--|--|
| Q. Why a write buffer ? | A. So CPU doesn't stall |
| Q. Why a buffer, why not just one register ? | A. Bursts of writes are common. |
| Q. Are Read After Write (RAW) hazards an issue for write buffer? | A. Yes! Drain buffer before next read, or send read first after check write buffers. |

Cache misses

Cache misses can be divided into three categories

- Compulsory** First access miss, cold start miss.
- Capacity** Cache is full.
- Conflict** Two blocks are mapped to the same location.

6 Basic Cache Optimizations

Reducing Miss Rate

1. Larger Block size (compulsory misses)
2. Larger Cache size (capacity misses)
3. Higher Associativity (conflict misses)

Reducing Miss Penalty

4. Multilevel Caches

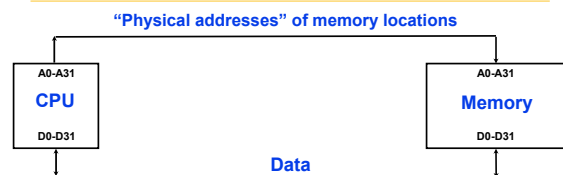
Reducing hit time

5. Giving Reads Priority over Writes
E.g., Read complete before earlier writes in write buffer
6. Avoiding Address Translation during Indexing of the Cache

Outline

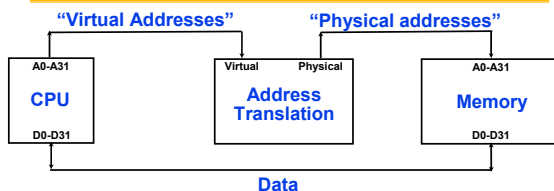
- Review
- Memory hierarchy
- Locality
- Cache design
- **Virtual address spaces**
- **Page table layout**
- **TLB design options**
- **Conclusion**

The Limits of Physical Addressing



- All programs share one address space: The **physical** address space
- Machine language programs must be aware of the machine organization
- No way to prevent a program from accessing **any** machine resource

Solution: Add a Layer of Indirection



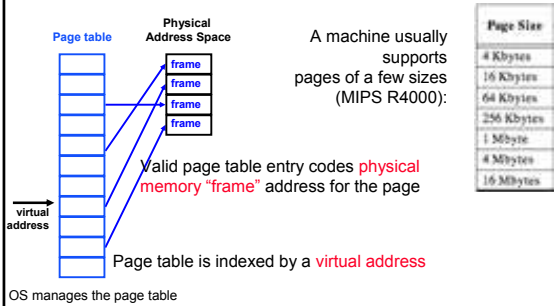
- User programs run in a standardized **virtual** address space
- **Address Translation** hardware managed by the operating system (OS) maps virtual address to physical memory
- Hardware supports "modern" OS features: **Protection, Translation, Sharing**

Three Advantages of Virtual Memory

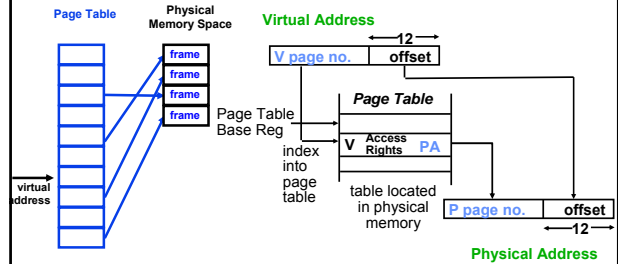
- **Translation:**
 - Program can be given consistent view of memory, even though physical memory is scrambled
 - Makes multithreading reasonable (now used a lot!)
 - Only the most important part of program ("Working Set") must be in physical memory.
 - Contiguous structures (like stacks) use only as much physical memory as necessary yet still grow later.
- **Protection:**
 - Different threads (or processes) protected from each other.
 - Different pages can be given special behavior
 - » (Read Only, Invisible to user programs, etc).
 - Kernel data protected from User programs
 - Very important for protection from malicious programs
- **Sharing:**
 - Can map same physical page to multiple users ("Shared memory")

Page tables encode virtual address spaces

A virtual address space is divided into blocks of memory called **pages**



Details of Page Table



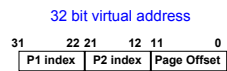
- Page table maps virtual page numbers to physical frames ("PTE" = Page Table Entry)
- Virtual memory => treat memory ≈ cache for disk

Page tables may not fit in memory!

A table for 4KB pages for a 32-bit address space has 1M entries

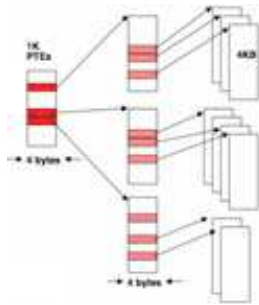
Each process needs its own address space!

Two-level Page Tables

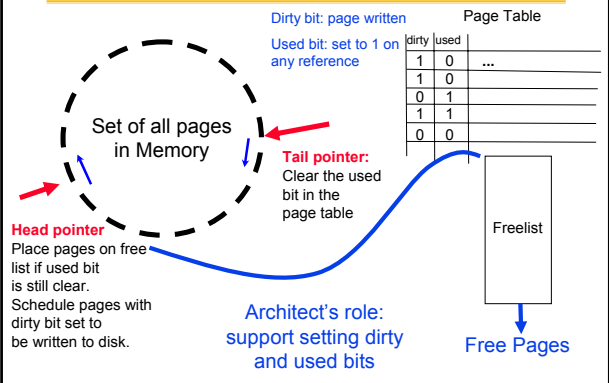


Top-level table wired in main memory

Subset of 1024 second-level tables in main memory; rest are on disk or unallocated

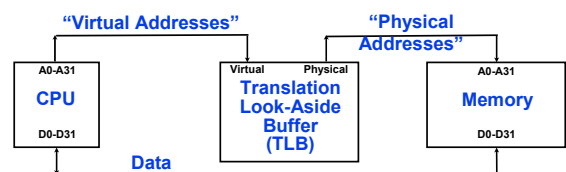


VM and Disk: Page replacement policy



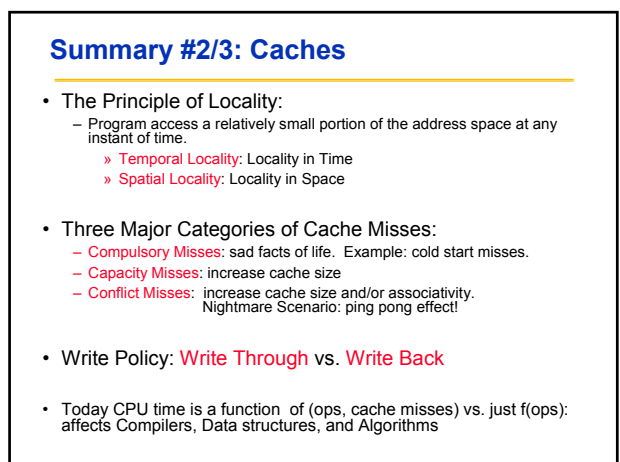
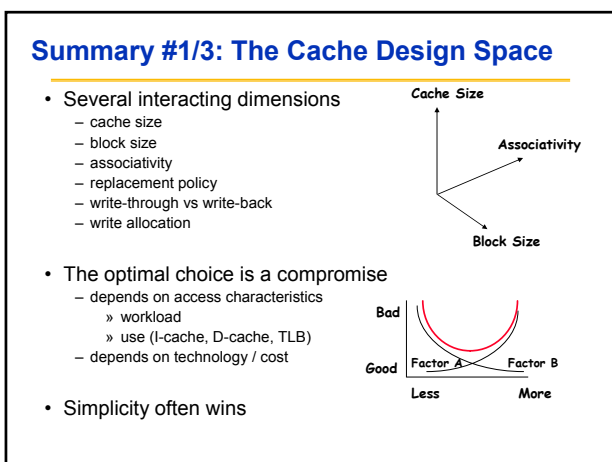
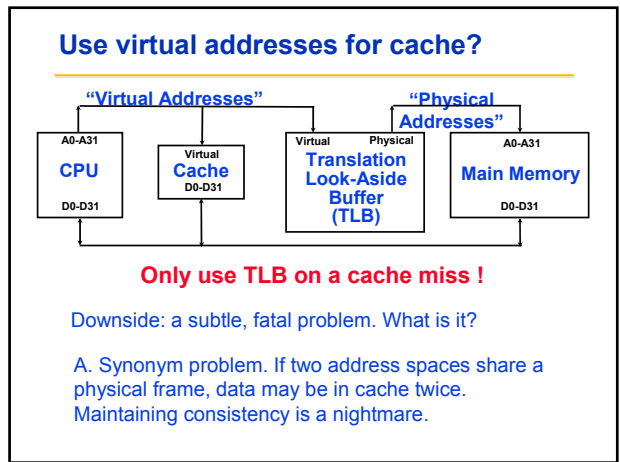
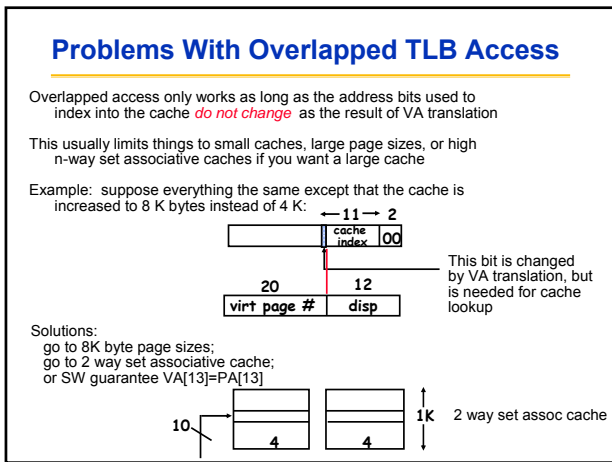
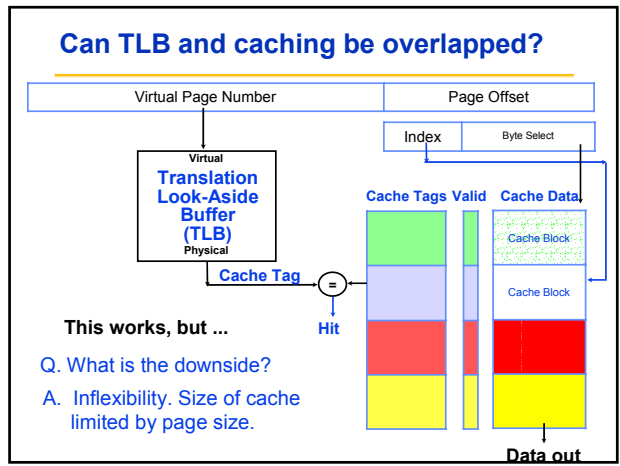
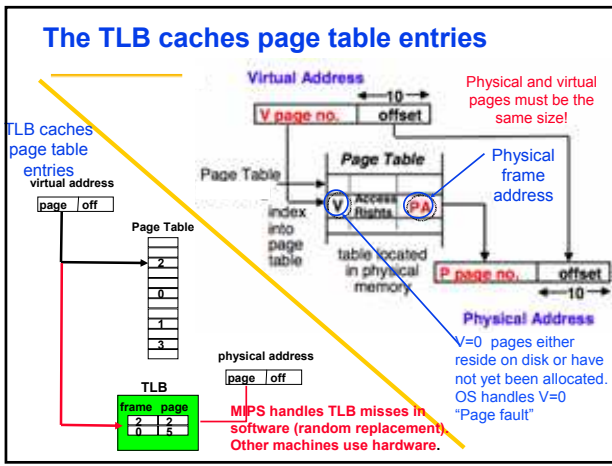
TLB Design Concepts

MIPS Address Translation: How does it work?



Translation Look-Aside Buffer (TLB)

- A small fully-associative cache of mappings from virtual to physical addresses
- TLB also contains protection bits for virtual address
- Fast common case: Virtual address is in TLB, process has permission to read/write it.



Summary #3/3: TLB, Virtual Memory

- Page tables map virtual address to physical address
- TLBs are important for fast translation
- TLB misses are significant in processor performance
 - funny times, as most systems can't access all of 2nd level cache without TLB misses!
- Caches, TLBs, Virtual Memory all understood by examining how they deal with 4 questions:
 - 1) Where can block be placed?
 - 2) How is block found?
 - 3) What block is replaced on miss?
 - 4) How are writes handled?
- Today VM allows many processes to share single memory without having to swap all processes to disk; [today VM protection is more important than memory hierarchy benefits, but computers insecure](#)

Reading

- **This lecture: appendix C *Memory Hierarchy***
- **Next lecture: chapter 2 *Instruction-Level Parallelism***

Lecture 4 – Instruction Level Parallelism

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

Outline

- ILP
- Compiler techniques to increase ILP
- Loop Unrolling
- Static Branch Prediction
- Dynamic Branch Prediction
- Overcoming Data Hazards with Dynamic Scheduling
- Tomasulo Algorithm
- Conclusion

Recall from Pipelining

Pipeline CPI = Ideal pipeline CPI + Structural Stalls + Data Hazard Stalls + Control Stalls

- **Ideal pipeline CPI**: measure of the maximum performance attainable by the implementation
- **Structural hazards**: HW cannot support this combination of instructions
- **Data hazards**: instruction depends on result of prior instruction still in the pipeline
- **Control hazards**: caused by delay between the fetching of instructions and decisions about changes in control flow (branches and jumps)

Instruction Level Parallelism

Instruction-Level Parallelism (ILP): overlap the execution of instructions to improve performance

Two approaches to exploit ILP:

- 1) Rely on hardware to help discover and exploit the parallelism **dynamically** (e.g., Pentium 4, AMD Opteron, IBM Power)
- 2) Rely on software technology to find parallelism, **statically** at compile-time (e.g., Itanium 2)

Instruction-Level Parallelism (ILP)

- **Basic Block (BB) ILP is quite small**
 - BB: a straight-line code sequence with no branches in except to the entry and no branches out except at the exit
 - average dynamic branch frequency 15% to 25%
⇒ 4 to 7 instructions execute between a pair of branches
 - plus instructions in BB likely to depend on each other
- To obtain substantial performance enhancements, we must exploit ILP across multiple basic blocks
- Simplest: **loop-level parallelism** to exploit parallelism among iterations of a loop. E.g.,

```
for (i=1; i<=1000; i=i+1)
    x[i] = x[i] + y[i];
```

Loop-Level Parallelism

- Exploit loop-level parallelism to parallelism by “unrolling loop” either by
 1. dynamic via branch prediction or
 2. static via loop unrolling by compiler*(Another way is vectors, to be covered later)*
- Determining instruction dependence is critical to Loop Level Parallelism
- If 2 instructions are
 - **parallel**, they can execute simultaneously in a pipeline of arbitrary depth without causing any stalls (assuming no structural hazards)
 - **dependent**, they are not parallel and must be executed in order, although they may often be partially overlapped

Data Dependence and Hazards

- Instr_j is **data dependent** (aka **true dependence**) on Instr_i.
 1. Instr_j tries to read operand before Instr_i writes it


```

          I: add r1, r2, r3
          J: sub r4, r1, r3
          
```
 2. or Instr_j is data dependent on Instr_k which is dependent on Instr_i
- If two instructions are data dependent, they cannot execute simultaneously or be completely overlapped
- Data dependence in instruction sequence ⇒ data dependence in source code ⇒ effect of original data dependence must be preserved
- If data dependence caused a hazard in pipeline, called a **Read After Write (RAW) hazard**

ILP and Data Dependencies, Hazards

- HW/SW must preserve **program order**: order instructions would execute in if executed sequentially as determined by original source program
 - Dependences are a property of **programs**
- Presence of dependence indicates **potential** for a hazard, but actual hazard and length of any stall is property of the **pipeline**
- Importance of the data dependencies
 - 1) indicates the possibility of a hazard
 - 2) determines order in which results must be calculated
 - 3) sets an upper bound on how much parallelism can possibly be exploited
- HW/SW goal: exploit parallelism by preserving program order **only where it affects the outcome of the program**

Name Dependence #1: Anti-dependence

- **Name dependence**: when 2 instructions use same register or memory location, called a **name**, but no flow of data between the instructions associated with that name; **two versions of name dependence**
- Instr_j writes operand **before** Instr_i reads it


```

      I: sub r4, r1, r3
      J: add r1, r2, r3
      K: mul r6, r1, r7
      
```

Called an “**anti-dependence**” by compiler writers. This results from reuse of the name “r1”
- If anti-dependence caused a hazard in the pipeline, called a **Write After Read (WAR) hazard**

Name Dependence #2: Output dependence

- Instr_j writes operand **before** Instr_i writes it.


```

      I: sub r1, r4, r3
      J: add r1, r2, r3
      K: mul r6, r1, r7
      
```
- Called an “**output dependence**” by compiler writers. This also results from the reuse of name “r1”
- If output-dependence caused a hazard in the pipeline, called a **Write After Write (WAW) hazard**
- Instructions involved in a name dependence can execute simultaneously **if name used in instructions is changed** so instructions do not conflict
 - **Register renaming** resolves name dependence for regs
 - Either by compiler or by HW

Control Dependencies

Every instruction is control dependent on some set of branches, and, in general, these control dependencies must be preserved to preserve program order

```

if p1 {
  S1;
};
if p2 {
  S2;
}

```

S1 is control dependent on **p1**, and **S2** is control dependent on **p2** but not on **p1**.

Control Dependence Ignored

- **Control dependence need not be preserved**
 - willing to execute instructions that should not have been executed, thereby violating the control dependences, **if** can do so without affecting correctness of the program
- Instead, two properties critical to program correctness are
 - 1) **exception behavior** and
 - 2) **data flow**

Exception Behavior

- Preserving exception behavior
 ⇒ any changes in instruction execution order must not change how exceptions are raised in program
 (⇒ no new exceptions)

• Example:

```
DADDU    R2, R3, R4
BEQZ    R2, L1
LW      R1, 0(R2)
L1:
- (Assume branches not delayed)
```

- Problem with moving LW before BEQZ?

Data Flow

- **Data flow:** actual flow of data values among instructions that produce results and those that consume them
 – branches make flow dynamic, determine which instruction is supplier of data

• Example:

```
DADDU    R1, R2, R3
BEQZ    R4, L
DSUBU    R1, R5, R6
L: ...
OR      R7, R1, R8
```

- OR depends on DADDU or DSUBU?
 Must preserve data flow on execution

Software Techniques - Example

- This code, add a scalar to a vector:

```
for (i=1000; i>0; i=i-1)
    x[i] = x[i] + s;
```
- Assume following latencies for all examples
 – Ignore delayed branch in these examples

Instruction producing result	Instruction using result	Latency in cycles	stalls between in cycles
FP ALU op	Another FP ALU op	4	3
FP ALU op	Store double	3	2
Load double	FP ALU op	1	1
Load double	Store double	1	0
Integer op	Integer op	1	0

FP Loop: Where are the Hazards?

First translate into MIPS code:

-To simplify, assume 8 is lowest address

```
Loop: L.D    F0,0(R1);F0=vector element
      ADD.D  F4,F0,F2;add scalar from F2
      S.D   0(R1),F4;store result
      DADDUI R1,R1,-8;decrement pointer 8B (DW)
      BNEZ  R1,Loop ;branch R1!=zero
```

FP Loop Showing Stalls

```
1 Loop: L.D    F0,0(R1);F0=vector element
2      stall
3      ADD.D  F4,F0,F2;add scalar in F2
4      stall
5      stall
6      S.D   0(R1),F4;store result
7      DADDUI R1,R1,-8;decrement pointer 8B (DW)
8      stall ;assumes can't forward to branch
9      BNEZ  R1,Loop ;branch R1!=zero
```

Instruction producing result	Instruction using result	Latency in clock cycles
FP ALU op	Another FP ALU op	3
FP ALU op	Store double	2
Load double	FP ALU op	1

9 clock cycles: Rewrite code to minimize stalls?

Revised FP Loop Minimizing Stalls

```
1 Loop: L.D    F0,0(R1)
2      DADDUI R1,R1,-8
3      ADD.D  F4,F0,F2
4      stall
5      stall
6      S.D   8(R1),F4 ;altered offset when move DSUBUI
7      BNEZ  R1,Loop
```

Swap DADDUI and S.D by changing address of S.D

Instruction producing result	Instruction using result	Latency in clock cycles
FP ALU op	Another FP ALU op	3
FP ALU op	Store double	2
Load double	FP ALU op	1

7 clock cycles, but just 3 for execution (L.D, ADD.D,S.D), 4 for loop overhead; How make faster?

Unroll Loop Four Times (straightforward way)

```

1 Loop: L.D    F0, 0(R1)
3      ADD.D  F4, F0, F2
6      S.D    0(R1), F4           ;drop DSUBUI & BNEZ
7      L.D    F6, -8(R1)
9      ADD.D  F8, F6, F2
12     S.D    -8(R1), F8         ;drop DSUBUI & BNEZ
13     L.D    F10, -16(R1)
15     ADD.D  F12, F10, F2
18     S.D    -16(R1), F12      ;drop DSUBUI & BNEZ
19     L.D    F14, -24(R1)
21     ADD.D  F16, F14, F2
24     S.D    -24(R1), F16
25     DADDUI R1, R1, #-32      ;alter to 4*8
27     BNEZ  R1, LOOP
  
```

27 clock cycles, or 6.75 per iteration
(Assumes R1 is multiple of 4)

Unrolled Loop That Minimizes Stalls

```

1 Loop: L.D    F0, 0(R1)
2      L.D    F6, -8(R1)
3      L.D    F10, -16(R1)
4      L.D    F14, -24(R1)
5      ADD.D  F4, F0, F2
6      ADD.D  F8, F6, F2
7      ADD.D  F12, F10, F2
8      ADD.D  F16, F14, F2
9      S.D    0(R1), F4
10     S.D    -8(R1), F8
11     S.D    -16(R1), F12
12     DSUBUI R1, R1, #32
13     S.D    8(R1), F16; 8-32 = -24
14     BNEZ  R1, LOOP
  
```

14 clock cycles, or 3.5 per iteration

Unrolled Loop Detail

- Do not usually know upper bound of loop
- Suppose it is n , and we would like to unroll the loop to make k copies of the body
 - 1st executes $(n \bmod k)$ times and has a body that is the original loop
 - 2nd is the unrolled body surrounded by an outer loop that iterates (n/k) times
- For large values of n , most of the execution time will be spent in the unrolled loop

5 Loop Unrolling Decisions

Requires understanding how one instruction depends on another and how the instructions can be changed or reordered given the dependences:

- Determine loop unrolling useful by finding that loop iterations were independent (except for maintenance code)
- Use different registers to avoid unnecessary constraints forced by using same registers for different computations
- Eliminate the extra test and branch instructions and adjust the loop termination and iteration code
- Determine that loads and stores in unrolled loop can be interchanged by observing that loads and stores from different iterations are independent
 - Transformation requires analyzing memory addresses and finding that they do not refer to the same address
- Schedule the code, preserving any dependences needed to yield the same result as the original code

3 Limits to Loop Unrolling

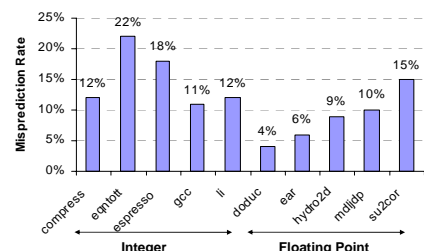
- Decrease in amount of overhead amortized with each extra unrolling
 - Amdahl's Law
- Growth in code size
 - For larger loops, concern it increases the instruction cache miss rate
- Register pressure: potential shortfall in registers created by aggressive unrolling and scheduling
 - If not be possible to allocate all live values to registers, may lose some or all of its advantage

Loop unrolling reduces impact of branches on pipeline; another way is branch prediction

Static Branch Prediction

- We saw scheduling code around delayed branch
- To reorder code around branches, need to predict branch statically when compile
- Simplest scheme is to predict a branch as taken
 - Average misprediction = untaken branch frequency = 34% SPEC

More accurate scheme predicts branches using profile information collected from earlier runs, and modify prediction based on last run:



Dynamic Branch Prediction

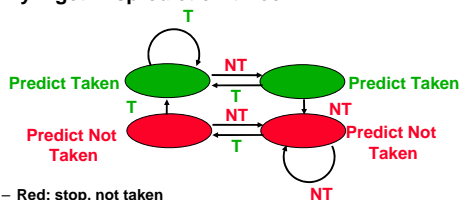
- **Why does prediction work?**
 - Underlying algorithm has regularities
 - Data that is being operated on has regularities
 - Instruction sequence has redundancies that are artifacts of way that humans/compiler think about problems
- **Is dynamic branch prediction better than static branch prediction?**
 - Seems to be
 - There are a small number of important branches in programs which have dynamic behavior

Dynamic Branch Prediction

- **Performance = $f(\text{accuracy, cost of misprediction})$**
- **Branch History Table: Lower bits of PC address index table of 1-bit values**
 - Says whether or not branch taken last time
 - No address check
- **Problem: in a loop, 1-bit BHT will cause two mispredictions (avg is 9 iterations before exit):**
 - End of loop case, when it exits instead of looping as before
 - First time through loop on *next* time through code, when it predicts exit instead of looping

Dynamic Branch Prediction

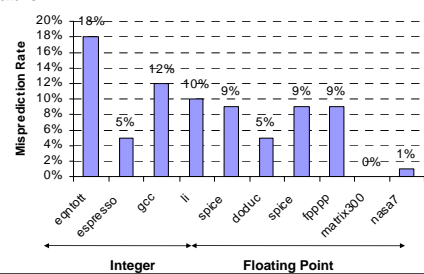
- **Solution: 2-bit scheme where change prediction only if get misprediction *twice***



- Red: stop, not taken
- Green: go, taken
- Adds *hysteresis* to decision making process

BHT Accuracy

- **Mispredict because either:**
 - Wrong guess for that branch
 - Got branch history of wrong branch when index the table
- **4096 entry table:**



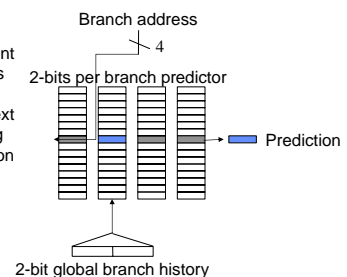
Correlated Branch Prediction

- **Idea: record m most recently executed branches as taken or not taken, and use that pattern to select the proper n -bit branch history table**
- **In general, (m,n) predictor means record last m branches to select between 2^m history tables, each with n -bit counters**
 - Thus, old 2-bit BHT is a $(0,2)$ predictor
- **Global Branch History: m -bit shift register keeping T/NT status of last m branches.**

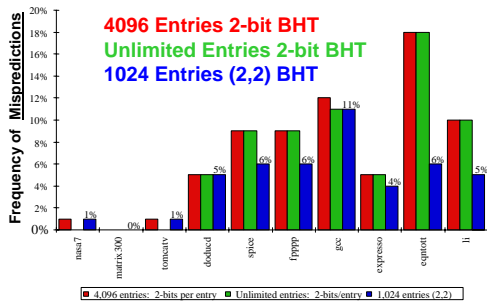
Correlating Branches

$(2,2)$ predictor

- Behavior of recent branches selects 2-bits per branch predictor between four predictions of next branch, updating just that prediction

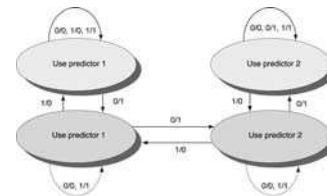


Accuracy of Different Schemes



Tournament Predictors

- Multilevel branch predictor
- Use n -bit saturating counter to choose between predictors
- Usual choice between global and local predictors



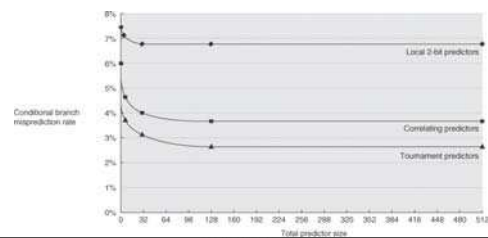
Tournament Predictors

Tournament predictor using, say, 4K 2-bit counters indexed by local branch address. Chooses between:

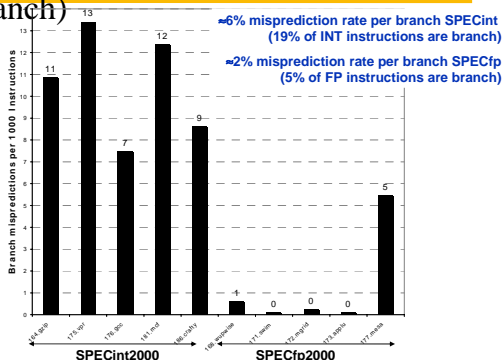
- Global predictor
 - 4K entries index by history of last 12 branches ($2^{12} = 4K$)
 - Each entry is a standard 2-bit predictor
- Local predictor
 - Local history table: 1024 10-bit entries recording last 10 branches, index by branch address
 - The pattern of the last 10 occurrences of that particular branch used to index table of 1K entries with 3-bit saturating counters

Comparing Predictors (Fig. 2.8)

- Advantage of tournament predictor is ability to select the right predictor for a particular branch
 - Particularly crucial for integer benchmarks.
 - A typical tournament predictor will select the global predictor almost 40% of the time for the SPEC integer benchmarks and less than 15% of the time for the SPEC FP benchmarks



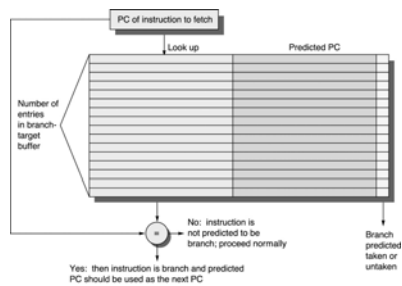
Branch misprediction rate (per 1000 instructions, not per branch)



Branch Target Buffers (BTB)

- Branch target calculation is costly and stalls the instruction fetch.
- BTB stores PCs the same way as caches.
- The PC of a branch is sent to the BTB.
- When a match is found the corresponding Predicted PC is returned.
- If the branch was predicted taken, instruction fetch continues at the returned predicted PC.

Branch Target Buffers



© 2003 Elsevier Science (USA). All rights reserved.

Dynamic Branch Prediction

Summary

- Prediction becoming important part of execution
- Branch History Table: 2 bits for loop accuracy
- Correlation: Recently executed branches correlated with next branch
 - Either different branches (GA)
 - Or different executions of same branches (PA)
- Tournament predictors take insight to next level, by using multiple predictors
 - usually one based on global information and one based on local information, and combining them with a selector
 - In 2006, tournament predictors using \approx 30K bits are in processors like the Power5 and Pentium 4
- Branch Target Buffer: include branch address & prediction

break

Outline

- ILP
- Compiler techniques to increase ILP
- Loop Unrolling
- Static Branch Prediction
- Dynamic Branch Prediction
- Overcoming Data Hazards with Dynamic Scheduling
- Tomasulo Algorithm
- Conclusion

Advantages of Dynamic

Scheduling

- **Dynamic scheduling** - hardware rearranges the instruction execution to reduce stalls while maintaining data flow and exception behavior
- It handles cases when dependences unknown at compile time
 - it allows the processor to tolerate unpredictable delays such as cache misses, by executing other code while waiting for the miss to resolve
- It allows code that compiled for one pipeline to run efficiently on a different pipeline
- It simplifies the compiler
- **Hardware speculation**, a technique with significant performance advantages, builds on dynamic scheduling (next lecture)

HW Schemes: Instruction

Parallelism

- Key idea: Allow instructions behind stall to proceed


```
DIVD F0, F2, F4
ADDD F10, F0, F8
SUBD F12, F8, F14
```
- Enables **out-of-order execution** and allows **out-of-order completion** (e.g., SUBD)
 - In a dynamically scheduled pipeline, all instructions still pass through issue stage in order (**in-order issue**)
- Will distinguish when an instruction **begins execution** and when it **completes execution**; between 2 times, the instruction is **in execution**
- Note: Dynamic execution creates WAR and WAW hazards and makes exceptions harder

Dynamic Scheduling Step 1

- Simple pipeline had 1 stage to check both structural and data hazards: Instruction Decode (ID), also called Instruction Issue
- Split the ID pipe stage of simple 5-stage pipeline into 2 stages:
 - 1) **Issue**—Decode instructions, check for structural hazards
 - 2) **Read operands**—Wait until no data hazards, then read operands

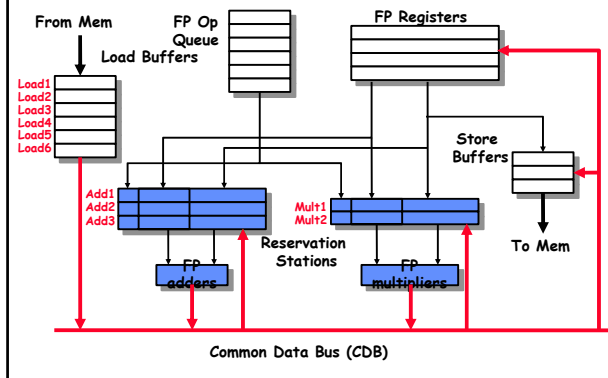
A Dynamic Algorithm: Tomasulo's

- For IBM 360/91 (before caches!)
 - ⇒ Long memory latency
- Goal: High Performance without special compilers
- Small number of floating point registers (4 in 360) prevented interesting compiler scheduling of operations
 - This led Tomasulo to try to figure out how to get more effective registers — **renaming in hardware!**
- Why Study 1966 Computer?
- The descendants of this have flourished!
 - Alpha 21264, Pentium 4, AMD Opteron, Power 5, ...

Tomasulo Algorithm

- Control & buffers **distributed** with Function Units (FU)
 - FU buffers called “**reservation stations**”; have pending operands
- Registers in instructions replaced by values or pointers to reservation stations (RS); called **register renaming** ;
 - Renaming avoids WAR, WAW hazards
 - More reservation stations than registers, so can do optimizations compilers cannot
- Results to FU from RS, **not through registers**, over **Common Data Bus** that broadcasts results to all FUs
 - Avoids RAW hazards by executing an instruction only when its operands are available
- Load and Stores treated as FUs with RSs as well
- Integer instructions can go past branches (predict taken), allowing FP ops beyond basic block in FP queue

Tomasulo Organization



Reservation Station Components

Op: Operation to perform in the unit (e.g., + or -)

Vj, Vk: Value of Source operands

- Store buffers has V field, result to be stored

Qj, Qk: Reservation stations producing source registers (value to be written)

- Note: Qj, Qk=0 ⇒ ready
- Store buffers only have Qi for RS producing result

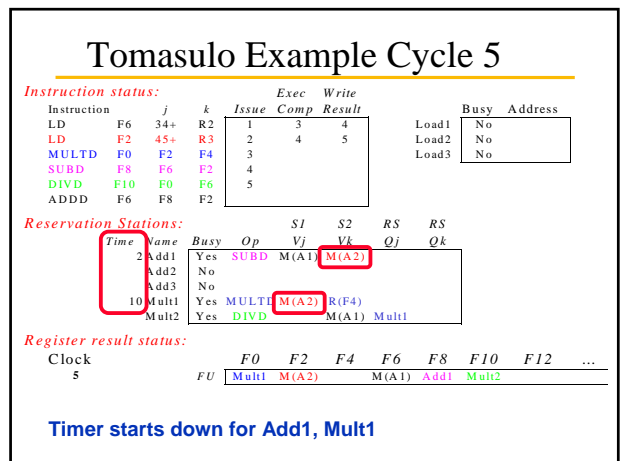
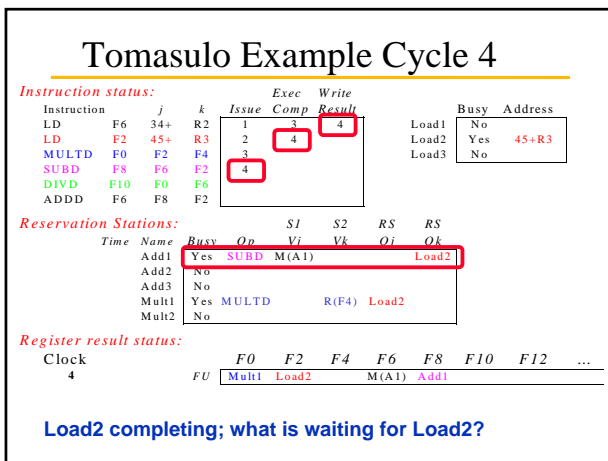
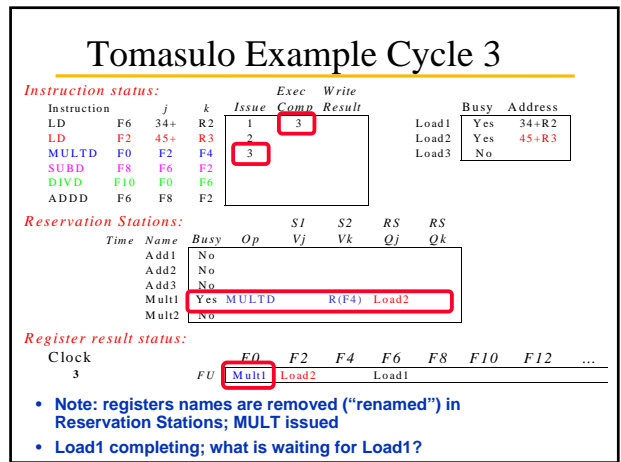
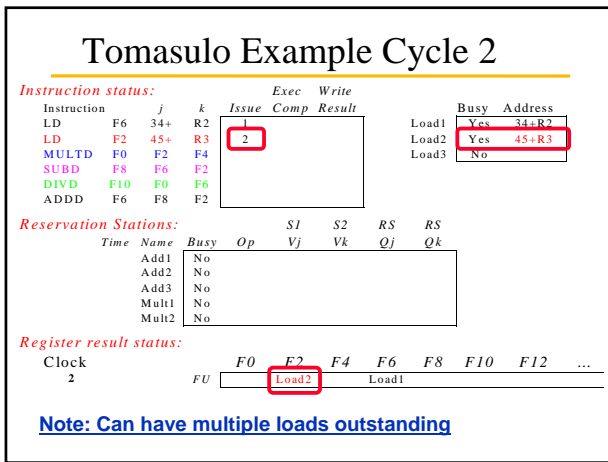
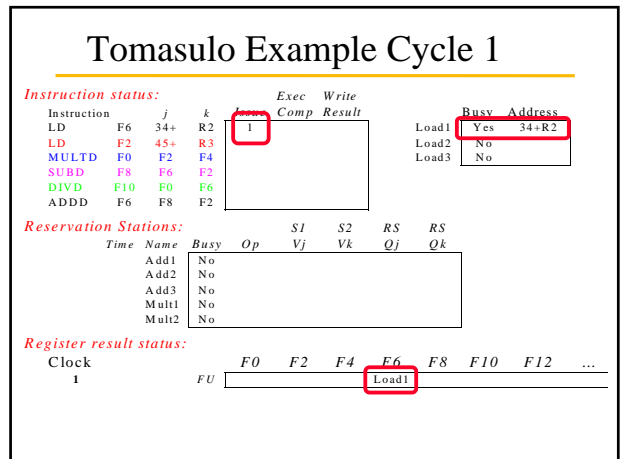
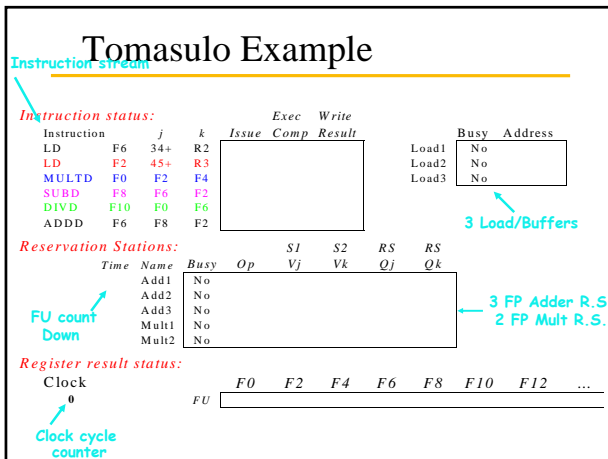
Busy: Indicates reservation station or FU is busy

Register result status—Indicates which functional unit will write each register, if one exists. Blank when no pending instructions that will write that register.

Three Stages of Tomasulo Algorithm

1. **Issue**—get instruction from FP Op Queue
 - If reservation station free (no structural hazard), control issues instr & sends operands (renames registers).
2. **Execute**—operate on operands (EX)
 - When both operands ready then execute; if not ready, watch Common Data Bus for result
3. **Write result**—finish execution (WB)
 - Write on Common Data Bus to all awaiting units; mark reservation station available

- Normal data bus: data + destination (“go to” bus)
- **Common data bus:** data + **source** (“come from” bus)
 - 64 bits of data + 4 bits of Functional Unit **source** address
 - Write if matches expected Functional Unit (produces result)
 - Does the broadcast
- Example speed: 3 clocks for Fl. pt. +,-; 10 for *; 40 clks for /



Tomasulo Example Cycle 6

Instruction status:

Instruction	j	k	Exec			Write	Busy	Address
			Issue	Comp	Result			
LD	F6	34+	R2	1	3	4	Load1	No
LD	F2	45+	R3	2	4	5	Load2	No
MULTD	F0	F2	F4	3			Load3	No
SUBD	F8	F6	F2	4				
DIVD	F10	F0	F6	5				
ADDD	F6	F8	F2	6				

Reservation Stations:

Time	Name	Busy	Op	S1		S2		RS	RS
				Vj	Vk	Qj	Qk		
1	Add1	Yes	SUBD	M(A1)	M(A2)				
2	Add2	Yes	ADDD	M(A2)	Add1				
3	Add3	No							
9	Multi1	Yes	MULTD	M(A2)	R(F4)				
	Multi2	Yes	DIVD	M(A1)	Multi1				

Register result status:

Clock	FU							
	F0	F2	F4	F6	F8	F10	F12	...
6	Multi1	M(A2)	Add2	Add1	Multi2			

Issue ADDD here despite name dependency on F6?

Tomasulo Example Cycle 7

Instruction status:

Instruction	j	k	Exec			Write	Busy	Address
			Issue	Comp	Result			
LD	F6	34+	R2	1	3	4	Load1	No
LD	F2	45+	R3	2	4	5	Load2	No
MULTD	F0	F2	F4	3			Load3	No
SUBD	F8	F6	F2	4				
DIVD	F10	F0	F6	5				
ADDD	F6	F8	F2	6				

Reservation Stations:

Time	Name	Busy	Op	S1		S2		RS	RS
				Vj	Vk	Qj	Qk		
0	Add1	Yes	SUBD	M(A1)	M(A2)				
	Add2	Yes	ADDD	M(A2)	Add1				
	Add3	No							
8	Multi1	Yes	MULTD	M(A2)	R(F4)				
	Multi2	Yes	DIVD	M(A1)	Multi1				

Register result status:

Clock	FU							
	F0	F2	F4	F6	F8	F10	F12	...
7	Multi1	M(A2)	Add2	Add1	Multi2			

Add1 (SUBD) completing; what is waiting for it?

Tomasulo Example Cycle 8

Instruction status:

Instruction	j	k	Exec			Write	Busy	Address
			Issue	Comp	Result			
LD	F6	34+	R2	1	3	4	Load1	No
LD	F2	45+	R3	2	4	5	Load2	No
MULTD	F0	F2	F4	3			Load3	No
SUBD	F8	F6	F2	4	7	8		
DIVD	F10	F0	F6	5				
ADDD	F6	F8	F2	6				

Reservation Stations:

Time	Name	Busy	Op	S1		S2		RS	RS
				Vj	Vk	Qj	Qk		
	Add1	No							
2	Add2	Yes	ADDD	(M-M)	M(A2)				
	Add3	No							
7	Multi1	Yes	MULTD	M(A2)	R(F4)				
	Multi2	Yes	DIVD	M(A1)	Multi1				

Register result status:

Clock	FU							
	F0	F2	F4	F6	F8	F10	F12	...
8	Multi1	M(A2)	Add2	(M-M)	Multi2			

Tomasulo Example Cycle 9

Instruction status:

Instruction	j	k	Exec			Write	Busy	Address
			Issue	Comp	Result			
LD	F6	34+	R2	1	3	4	Load1	No
LD	F2	45+	R3	2	4	5	Load2	No
MULTD	F0	F2	F4	3			Load3	No
SUBD	F8	F6	F2	4	7	8		
DIVD	F10	F0	F6	5				
ADDD	F6	F8	F2	6				

Reservation Stations:

Time	Name	Busy	Op	S1		S2		RS	RS
				Vj	Vk	Qj	Qk		
	Add1	No							
1	Add2	Yes	ADDD	(M-M)	M(A2)				
	Add3	No							
6	Multi1	Yes	MULTD	M(A2)	R(F4)				
	Multi2	Yes	DIVD	M(A1)	Multi1				

Register result status:

Clock	FU							
	F0	F2	F4	F6	F8	F10	F12	...
9	Multi1	M(A2)	Add2	(M-M)	Multi2			

Tomasulo Example Cycle 10

Instruction status:

Instruction	j	k	Exec			Write	Busy	Address
			Issue	Comp	Result			
LD	F6	34+	R2	1	3	4	Load1	No
LD	F2	45+	R3	2	4	5	Load2	No
MULTD	F0	F2	F4	3			Load3	No
SUBD	F8	F6	F2	4	7	8		
DIVD	F10	F0	F6	5				
ADDD	F6	F8	F2	6	10			

Reservation Stations:

Time	Name	Busy	Op	S1		S2		RS	RS
				Vj	Vk	Qj	Qk		
	Add1	No							
0	Add2	Yes	ADDD	(M-M)	M(A2)				
	Add3	No							
5	Multi1	Yes	MULTD	M(A2)	R(F4)				
	Multi2	Yes	DIVD	M(A1)	Multi1				

Register result status:

Clock	FU							
	F0	F2	F4	F6	F8	F10	F12	...
10	Multi1	M(A2)	Add2	(M-M)	Multi2			

Add2 (ADDD) completing; what is waiting for it?

Tomasulo Example Cycle 11

Instruction status:

Instruction	j	k	Exec			Write	Busy	Address
			Issue	Comp	Result			
LD	F6	34+	R2	1	3	4	Load1	No
LD	F2	45+	R3	2	4	5	Load2	No
MULTD	F0	F2	F4	3			Load3	No
SUBD	F8	F6	F2	4	7	8		
DIVD	F10	F0	F6	5				
ADDD	F6	F8	F2	6	10	11		

Reservation Stations:

Time	Name	Busy	Op	S1		S2		RS	RS
				Vj	Vk	Qj	Qk		
	Add1	No							
	Add2	No							
	Add3	No							
4	Multi1	Yes	MULTD	M(A2)	R(F4)				
	Multi2	Yes	DIVD	M(A1)	Multi1				

Register result status:

Clock	FU							
	F0	F2	F4	F6	F8	F10	F12	...
11	Multi1	M(A2)	(M-M)	(M-M)	Multi2			

- Write result of ADDD here?
- All quick instructions complete in this cycle!

Tomasulo Example Cycle 12

Instruction status:

Instruction	j	k	Exec			Write	Busy	Address
			Issue	Comp	Result			
LD	F6	34+	R2	1	3	4	Load1	No
LD	F2	45+	R3	2	4	5	Load2	No
MULTD	F0	F2	F4	3			Load3	No
SUBD	F8	F6	F2	4	7	8		
DIVD	F10	F0	F6	5				
ADDD	F6	F8	F2	6	10	11		

Reservation Stations:

Time	Name	Busy	Op	Vj	Vk	Qj	Qk
Add1		No					
Add2		No					
Add3		No					
3 Mult1	Yes	MULTD	M(A2)		R(F4)		
2 Mult2	Yes	DIVD			M(A1)	Multi	

Register result status:

Clock	F0	F2	F4	F6	F8	F10	F12	...
12	FU	Multi	M(A2)		(M-M+)(M-M)	Multi2		

Tomasulo Example Cycle 13

Instruction status:

Instruction	j	k	Exec			Write	Busy	Address
			Issue	Comp	Result			
LD	F6	34+	R2	1	3	4	Load1	No
LD	F2	45+	R3	2	4	5	Load2	No
MULTD	F0	F2	F4	3			Load3	No
SUBD	F8	F6	F2	4	7	8		
DIVD	F10	F0	F6	5				
ADDD	F6	F8	F2	6	10	11		

Reservation Stations:

Time	Name	Busy	Op	Vj	Vk	Qj	Qk
Add1		No					
Add2		No					
Add3		No					
2 Mult1	Yes	MULTD	M(A2)		R(F4)		
1 Mult2	Yes	DIVD			M(A1)	Multi	

Register result status:

Clock	F0	F2	F4	F6	F8	F10	F12	...
13	FU	Multi	M(A2)		(M-M+)(M-M)	Multi2		

Tomasulo Example Cycle 14

Instruction status:

Instruction	j	k	Exec			Write	Busy	Address
			Issue	Comp	Result			
LD	F6	34+	R2	1	3	4	Load1	No
LD	F2	45+	R3	2	4	5	Load2	No
MULTD	F0	F2	F4	3			Load3	No
SUBD	F8	F6	F2	4	7	8		
DIVD	F10	F0	F6	5				
ADDD	F6	F8	F2	6	10	11		

Reservation Stations:

Time	Name	Busy	Op	Vj	Vk	Qj	Qk
Add1		No					
Add2		No					
Add3		No					
1 Mult1	Yes	MULTD	M(A2)		R(F4)		
2 Mult2	Yes	DIVD			M(A1)	Multi	

Register result status:

Clock	F0	F2	F4	F6	F8	F10	F12	...
14	FU	Multi	M(A2)		(M-M+)(M-M)	Multi2		

Tomasulo Example Cycle 15

Instruction status:

Instruction	j	k	Exec			Write	Busy	Address
			Issue	Comp	Result			
LD	F6	34+	R2	1	3	4	Load1	No
LD	F2	45+	R3	2	4	5	Load2	No
MULTD	F0	F2	F4	3	15		Load3	No
SUBD	F8	F6	F2	4	7	8		
DIVD	F10	F0	F6	5				
ADDD	F6	F8	F2	6	10	11		

Reservation Stations:

Time	Name	Busy	Op	Vj	Vk	Qj	Qk
Add1		No					
Add2		No					
Add3		No					
0 Mult1	Yes	MULTD	M(A2)		R(F4)		
1 Mult2	Yes	DIVD			M(A1)	Multi	

Register result status:

Clock	F0	F2	F4	F6	F8	F10	F12	...
15	FU	Multi	M(A2)		(M-M+)(M-M)	Multi2		

Mult1 (MULTD) completing; what is waiting for it?

Tomasulo Example Cycle 16

Instruction status:

Instruction	j	k	Exec			Write	Busy	Address
			Issue	Comp	Result			
LD	F6	34+	R2	1	3	4	Load1	No
LD	F2	45+	R3	2	4	5	Load2	No
MULTD	F0	F2	F4	3	15	16	Load3	No
SUBD	F8	F6	F2	4	7	8		
DIVD	F10	F0	F6	5				
ADDD	F6	F8	F2	6	10	11		

Reservation Stations:

Time	Name	Busy	Op	Vj	Vk	Qj	Qk
Add1		No					
Add2		No					
Add3		No					
0 Mult1	No						
40 Mult2	Yes	DIVD	M*F4		M(A1)		

Register result status:

Clock	F0	F2	F4	F6	F8	F10	F12	...
16	FU	M*F4	M(A2)		(M-M+)(M-M)	Multi2		

Just waiting for Mult2 (DIVD) to complete

Faster than light computation
(skip a couple of cycles)

Tomasulo Example Cycle 55

Instruction status:

Instruction	j	k	Issue	Exec	Write	Load1	Address
				Comp	Result		
LD	F6	34+	R2	1	3	4	No
LD	F2	45+	R3	2	4	5	No
MULTD	F0	F2	F4	3	15	16	No
SUBD	F8	F6	F2	4	7	8	
DIVD	F10	F0	F6	5			
ADDD	F6	F8	F2	6	10	11	

Reservation Stations:

Time	Name	Busy	Op	Vj	Vk	RS	RS
						Qj	Qk
Add1	No						
Add2	No						
Add3	No						
Mult1	No						
1 Mult2	Yes	DIVD	M*F4	M(A1)			

Register result status:

Clock	F0	F2	F4	F6	F8	F10	F12	...
55	FU	M*F4	M(A2)	(M-M+)	(M-M)	Mult2		

Tomasulo Example Cycle 56

Instruction status:

Instruction	j	k	Issue	Exec	Write	Load1	Address
				Comp	Result		
LD	F6	34+	R2	1	3	4	No
LD	F2	45+	R3	2	4	5	No
MULTD	F0	F2	F4	3	15	16	No
SUBD	F8	F6	F2	4	7	8	
DIVD	F10	F0	F6	5	56		
ADDD	F6	F8	F2	6	10	11	

Reservation Stations:

Time	Name	Busy	Op	Vj	Vk	RS	RS
						Qj	Qk
Add1	No						
Add2	No						
Add3	No						
Mult1	No						
0 Mult2	Yes	DIVD	M*F4	M(A1)			

Register result status:

Clock	F0	F2	F4	F6	F8	F10	F12	...
56	FU	M*F4	M(A2)	(M-M+)	(M-M)	Mult2		

Mult2 (DIVD) is completing; what is waiting for it?

Tomasulo Example Cycle 57

Instruction status:

Instruction	j	k	Issue	Exec	Write	Load1	Address
				Comp	Result		
LD	F6	34+	R2	1	3	4	No
LD	F2	45+	R3	2	4	5	No
MULTD	F0	F2	F4	3	15	16	No
SUBD	F8	F6	F2	4	7	8	
DIVD	F10	F0	F6	5	56	57	
ADDD	F6	F8	F2	6	10	11	

Reservation Stations:

Time	Name	Busy	Op	Vj	Vk	RS	RS
						Qj	Qk
Add1	No						
Add2	No						
Add3	No						
Mult1	No						
Mult2	Yes	DIVD	M*F4	M(A1)			

Register result status:

Clock	F0	F2	F4	F6	F8	F10	F12	...
56	FU	M*F4	M(A2)	(M-M+)	(M-M)	Result		

Once again: In-order issue, out-of-order execution and out-of-order completion.

Why can Tomasulo overlap loop iterations?

- **Register renaming**
 - Multiple iterations use different physical destinations for registers (dynamic loop unrolling).
- **Reservation stations**
 - Permit instruction issue to advance past integer control flow operations
 - Also buffer old values of registers - totally avoiding the WAR stall
- **Other perspective: Tomasulo building data flow dependency graph on the fly**

offers

two major advantages

1. **Distribution of the hazard detection logic**
 - distributed reservation stations and the CDB
 - If multiple instructions waiting on single result, & each instruction has other operand, then instructions can be released simultaneously by broadcast on CDB
 - If a centralized register file were used, the units would have to read their results from the registers when register buses are available
2. **Elimination of stalls for WAW and WAR hazards**

Tomasulo Drawbacks

- **Complexity**
 - delays of 360/91, MIPS 10000, Alpha 21264, IBM PPC 620 in CA: AQA 2/e, but not in silicon!
- **Many associative stores (CDB) at high speed**
- **Performance limited by Common Data Bus**
 - Each CDB must go to multiple functional units
 - ⇒ high capacitance, high wiring density
 - Number of functional units that can complete per cycle limited to one!
 - » Multiple CDBs ⇒ more FU logic for parallel assoc stores
- **Non-precise interrupts!**
 - We will address this later

And In Conclusion ... (1)

- **Leverage Implicit Parallelism for Performance: Instruction Level Parallelism**
- **Loop unrolling by compiler to increase ILP**
- **Branch prediction to increase ILP**
- **Dynamic HW exploiting ILP**
 - Works when can't know dependence at compile time
 - Can hide L1 cache misses
 - Code for one machine runs well on another

And In Conclusion ... (2)

- **Reservations stations: *renaming* to larger set of registers + buffering source operands**
 - Prevents registers as bottleneck
 - Avoids WAR, WAW hazards
 - Allows loop unrolling in HW
- **Not limited to basic blocks (integer units gets ahead, beyond branches)**
- **Helps cache misses as well**
- **Lasting Contributions**
 - Dynamic scheduling
 - Register renaming
 - Load/store disambiguation
- **360/91 descendants are Intel Pentium 4, IBM Power 5, AMD Athlon/Opteron, ...**

Reading

- **This lecture: chapter 2 *Instruction-Level Parallelism***
- **Next week: no class, Oct 3rd**
- **Next class, Oct 10th: *ILP (cont'd)***
- **This afternoon: *introduction on assignment 2; highly recommended!***

Lecture 5 – Instruction Level Parallelism (cont'd)

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

Review from Last Time (1)

- Leverage Implicit Parallelism for Performance: Instruction Level Parallelism
- Loop unrolling by compiler to increase ILP
- Branch prediction to increase ILP
- Dynamic HW exploiting ILP
 - Works when can't know dependence at compile time
 - Can hide L1 cache misses
 - Code for one machine runs well on another

Review from Last Time (2)

- Reservations stations: *renaming* to larger set of registers + buffering source operands
 - Prevents registers as bottleneck
 - Avoids WAR, WAW hazards
 - Allows loop unrolling in HW
- Not limited to basic blocks (integer units gets ahead, beyond branches)
- Helps cache misses as well
- Lasting Contributions
 - Dynamic scheduling
 - Register renaming
 - Load/store disambiguation
- 360/91 descendants are Pentium 4, Power 5, AMD Athlon/Opteron, ...

Outline

- ILP
- Speculation
- Speculative Tomasulo Example
- Memory Aliases
- Exceptions
- VLIW
- Increasing instruction bandwidth
- Register Renaming vs. Reorder Buffer
- Value Prediction
- Limits to ILP

Speculation to greater ILP

- Greater ILP: Overcome control dependence by hardware speculating on outcome of branches and executing program as if guesses were correct
 - Speculation ⇒ fetch, issue, and execute instructions as if branch predictions were always correct
 - Dynamic scheduling ⇒ only fetches and issues instructions
- Essentially a data flow execution model: Operations execute as soon as their operands are available

Speculation to greater ILP

3 components of HW-based speculation:

1. Dynamic branch prediction to choose which instructions to execute
2. Speculation to allow execution of instructions before control dependences are resolved
 - + ability to undo effects of incorrectly speculated sequence
3. Dynamic scheduling to deal with scheduling of different combinations of basic blocks

Adding Speculation to Tomasulo

- Must separate execution from allowing instruction to finish or “commit”
- This additional step called **instruction commit**
- When an instruction is no longer speculative, allow it to update the register file or memory
- Requires additional set of buffers to hold results of instructions that have finished execution but have not committed
- This **reorder buffer (ROB)** is also used to pass results among instructions that may be speculated

Reorder Buffer (ROB)

- In Tomasulo’s algorithm, once an instruction writes its result, any subsequently issued instructions will find result in the register file
- With speculation, the register file is not updated until the instruction commits
 - (we know definitively that the instruction should execute)
- Thus, the ROB supplies operands in interval between completion of instruction execution and instruction commit
 - ROB is a source of operands for instructions, just as reservation stations (RS) provide operands in Tomasulo’s algorithm
 - ROB extends architected registers like RS

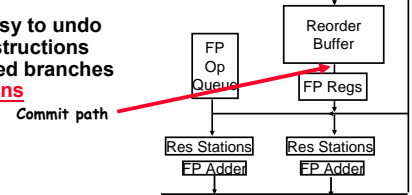
Reorder Buffer Entry

Each entry in the ROB contains four fields:

- 1. Instruction type**
 - A branch (has no destination result), a store (has a memory address destination), or a register operation (ALU operation or load, which has register destinations)
- 2. Destination**
 - Register number (for loads and ALU operations) or memory address (for stores) where the instruction result should be written
- 3. Value**
 - Value of instruction result until the instruction commits
- 4. Ready**
 - Indicates that instruction has completed execution, and the value is ready

Reorder Buffer operation

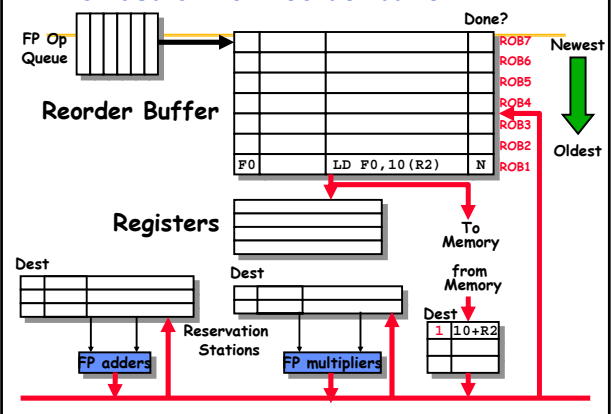
- Holds instructions in FIFO order, exactly as issued
- When instructions complete, results placed into ROB
 - Supplies operands to other instruction between execution complete & commit \Rightarrow more registers like RS
 - Tag results with ROB buffer number instead of reservation station
- Instructions **commit** \Rightarrow values at head of ROB placed in registers
- As a result, easy to undo speculated instructions on mispredicted branches or on exceptions

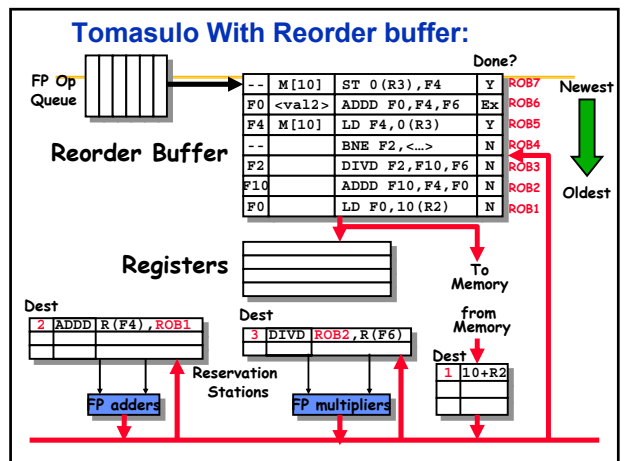
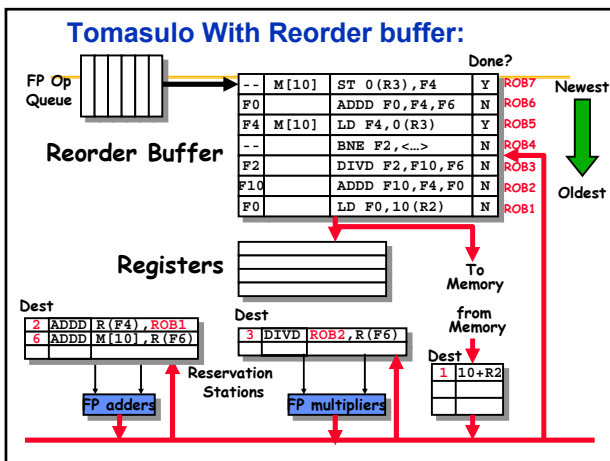
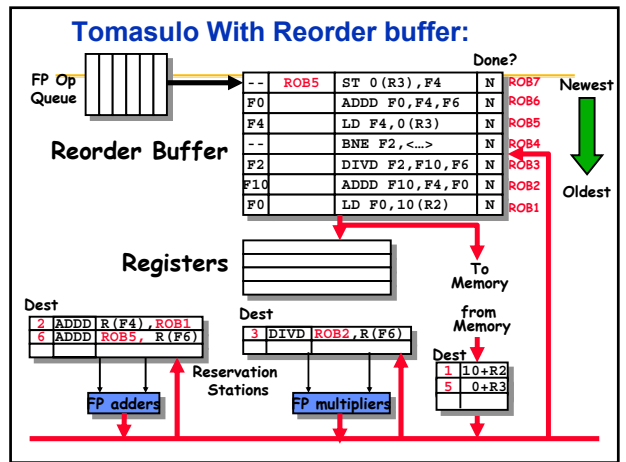
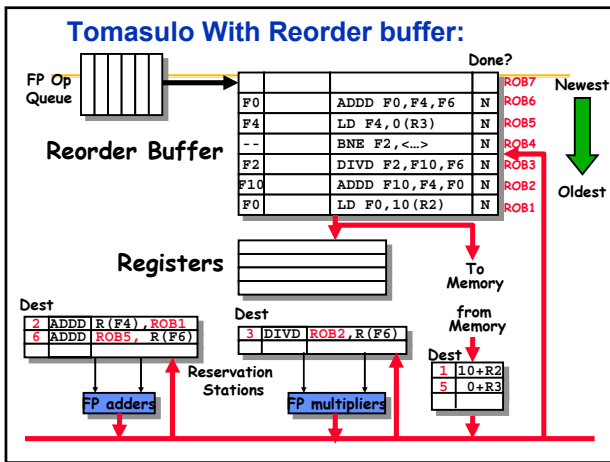
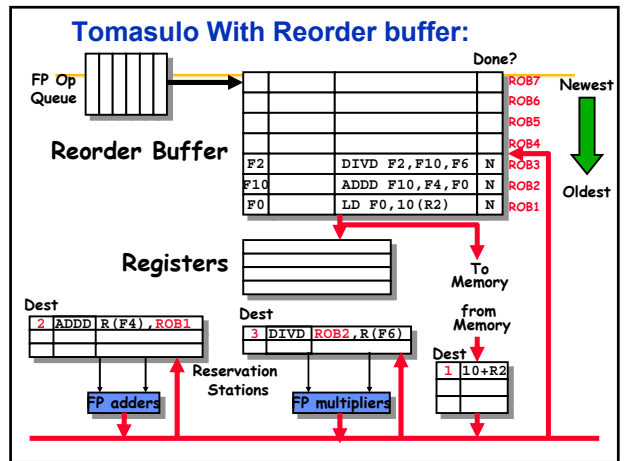
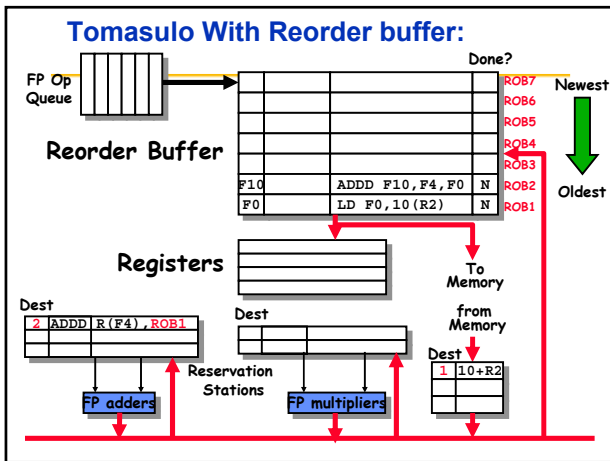


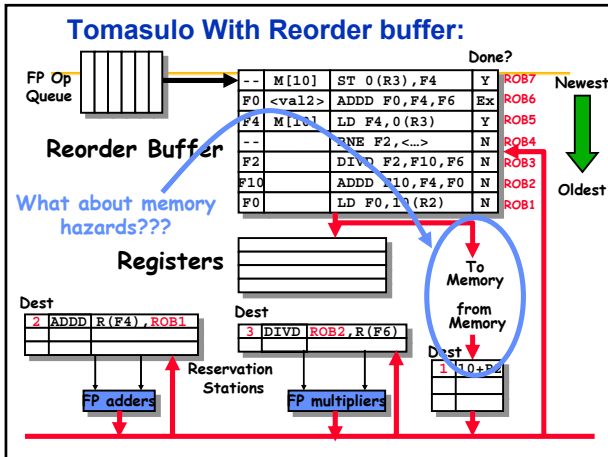
Recall: 4 Steps of Speculative Tomasulo Algorithm

- 1. Issue**—get instruction from FP Op Queue
If reservation station and reorder buffer slot free, issue instr & send operands & reorder buffer no. for destination (this stage sometimes called “dispatch”)
- 2. Execution**—operate on operands (EX)
When both operands ready then execute; if not ready, watch CDB for result; when both in reservation station, execute; checks RAW (sometimes called “issue”)
- 3. Write result**—finish execution (WB)
Write on Common Data Bus to all awaiting FUs & reorder buffer; mark reservation station available.
- 4. Commit**—update register with reorder result
When instr. at head of reorder buffer & result present, update register with result (or store to memory) and remove instr from reorder buffer. Mispredicted branch flushes reorder buffer (sometimes called “graduation”)

Tomasulo With Reorder buffer:







- ### Avoiding Memory Hazards
- WAW and WAR hazards through memory are eliminated with speculation because actual updating of memory occurs in order, when a store is at head of the ROB, and hence, no earlier loads or stores can still be pending
 - RAW hazards through memory are maintained by two restrictions:
 - not allowing a load to initiate the second step of its execution if any active ROB entry occupied by a store has a Destination field that matches the value of the A field of the load, and
 - maintaining the program order for the computation of an effective address of a load with respect to all earlier stores.
 - these restrictions ensure that any load that accesses a memory location written to by an earlier store cannot perform the memory access until the store has written the data

- ### Exceptions and Interrupts
- IBM 360/91 invented "imprecise interrupts"
 - Computer stopped at this PC; its likely close to this address
 - Not so popular with programmers
 - Also, what about Virtual Memory? (Not in IBM 360)
 - Technique for both precise interrupts/exceptions and speculation: in-order completion and in-order commit
 - If we speculate and are wrong, need to back up and restart execution to point at which we predicted incorrectly
 - This is exactly same as need to do with precise exceptions
 - Exceptions are handled by not recognizing the exception until instruction that caused it is ready to commit in ROB
 - If a speculated instruction raises an exception, the exception is recorded in the ROB
 - This is why reorder buffers in all new processors

- ### Getting CPI below 1
- CPI ≥ 1 if issue only 1 instruction every clock cycle
 - Multiple-issue processors come in 3 flavors:
 - statically-scheduled superscalar processors,
 - dynamically-scheduled superscalar processors, and
 - VLIW (very long instruction word) processors
 - 2 types of superscalar processors issue varying numbers of instructions per clock
 - use in-order execution if they are statically scheduled, or
 - out-of-order execution if they are dynamically scheduled
 - VLIW processors, in contrast, issue a fixed number of instructions formatted either as one large instruction or as a fixed instruction packet with the parallelism among instructions explicitly indicated by the instruction (Intel/HP Itanium)

- ### VLIW: Very Large Instruction Word
- Each "instruction" has explicit coding for multiple operations
 - In IA-64, grouping called a "packet"
 - In Transmeta, grouping called a "molecule" (with "atoms" as ops)
 - Tradeoff instruction space for simple decoding
 - The long instruction word has room for many operations
 - By definition, all the operations the compiler puts in the long instruction word are independent => execute in parallel
 - E.g., 2 integer operations, 2 FP ops, 2 Memory refs, 1 branch
 - 16 to 24 bits per field => 7*16 or 112 bits to 7*24 or 168 bits wide
 - Need compiling technique that schedules across several branches

Recall: Unrolled Loop that Minimizes Stalls for Scalar

```

1 Loop: L.D   F0, 0 (R1)           LD to ADD.D: 1 Cycle
2       L.D   F6, -8 (R1)         ADD.D to S.D: 2 Cycles
3       L.D   F10, -16 (R1)
4       L.D   F14, -24 (R1)
5       ADD.D F4, F0, F2
6       ADD.D F8, F6, F2
7       ADD.D F12, F10, F2
8       ADD.D F16, F14, F2
9       S.D   0 (R1), F4
10      S.D   -8 (R1), F8
11      S.D   -16 (R1), F12
12      DSUBUI R1, R1, #32
13      BNEZ  R1, LOOP
14      S.D   8 (R1), F16 ; 8-32 = -24
  
```

14 clock cycles, or 3.5 per iteration

Loop Unrolling in VLIW

Memory reference 1	Memory reference 2	FP operation 1	FP op. 2	Int. op./branch	Clock
L.D F0,0(R1)	L.D F6,-8(R1)				1
L.D F10,-16(R1)	L.D F14,-24(R1)				2
L.D F18,-32(R1)	L.D F22,-40(R1)	ADD.D F4,F0,F2	ADD.D F8,F6,F2		3
L.D F26,-48(R1)		ADD.D F12,F10,F2	ADD.D F16,F14,F2		4
		ADD.D F20,F18,F2	ADD.D F24,F22,F2		5
S.D 0(R1),F4	S.D -8(R1),F8	ADD.D F28,F26,F2			6
S.D -16(R1),F12	S.D -24(R1),F16				7
S.D -32(R1),F20	S.D -40(R1),F24			DSUBUI R1,R1,#48	8
S.D -0(R1),F28				BNEZ R1,LOOP	9

Unrolled 7 times to avoid delays

7 results in 9 clocks, or 1.3 clocks per iteration (1.8X)

Average: 2.5 ops per clock, 50% efficiency

Note: Need more registers in VLIW (15 vs. 6 in SS)

Problems with 1st Generation VLIW

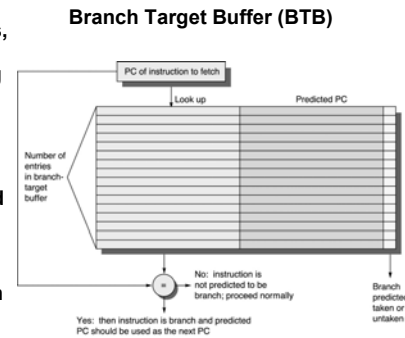
- **Increase in code size**
 - generating enough operations in a straight-line code fragment requires ambitiously unrolling loops
 - whenever VLIW instructions are not full, unused functional units translate to wasted bits in instruction encoding
- **Operated in lock-step; no hazard detection HW**
 - a stall in any functional unit pipeline caused entire processor to stall, since all functional units must be kept synchronized
 - Compiler might prediction function units, but caches hard to predict
- **Binary code compatibility**
 - Pure VLIW => different numbers of functional units and unit latencies require different versions of the code

Intel/HP IA-64 “Explicitly Parallel Instruction Computer (EPIC)”

- **IA-64**: instruction set architecture
- **128 64-bit integer regs + 128 82-bit floating point regs**
 - Not separate register files per functional unit as in old VLIW
- **Hardware checks dependencies (interlocks => binary compatibility over time)**
- **Predicated execution (select 1 out of 64 1-bit flags)**
=> 40% fewer mispredictions?
- **Itanium™** was first implementation (2001)
 - Highly parallel and deeply pipelined hardware at 800Mhz
 - 6-wide, 10-stage pipeline at 800Mhz on 0.18 μ process
- **Itanium 2™** is name of 2nd implementation (2005)
 - 6-wide, 8-stage pipeline at 1666Mhz on 0.13 μ process
 - Caches: 32 KB I, 32 KB D, 128 KB L2I, 128 KB L2D, 9216 KB L3

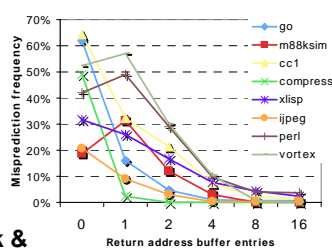
Increasing Instruction Fetch Bandwidth

- Predicts next instruct address, sends it out **before** decoding instruction
- PC of branch sent to BTB
- When match is found, Predicted PC is returned
- If branch predicted taken, instruction fetch continues at Predicted PC



IF BW: Return Address Predictor

- Small buffer of return addresses acts as a stack
- Caches most recent return addresses
- Call => Push a return address on stack
- Return => Pop an address off stack & predict as new PC



More Instruction Fetch Bandwidth

- **Integrated branch prediction** Branch predictor is part of instruction fetch unit and is constantly predicting branches
- **Instruction prefetch** Instruction fetch units prefetch to deliver multiple instruct. per clock, integrating it with branch prediction
- **Instruction memory access and buffering** Fetching multiple instructions per cycle:
 - May require accessing multiple cache blocks (prefetch to hide cost of crossing cache blocks)
 - Provides buffering, acting as on-demand unit to provide instructions to issue stage as needed and in quantity needed

Speculation: Register Renaming vs. ROB

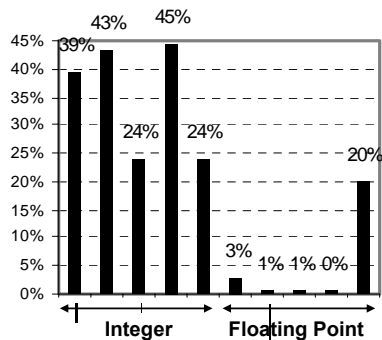
- Alternative to ROB is a larger physical set of registers combined with register renaming
 - Extended registers replace function of both ROB and reservation stations
- Instruction issue maps names of architectural registers to physical register numbers in extended register set
 - On issue, allocates a new unused register for the destination (which avoids WAW and WAR hazards)
 - Speculation recovery easy because a physical register holding an instruction destination does not become the architectural register until the instruction commits
- Most Out-of-Order processors today use extended registers with renaming

Value Prediction

- Attempts to predict **value** produced by instruction
 - E.g., Loads a value that changes infrequently
- Value prediction is useful only if it significantly increases ILP
 - Focus of research has been on loads; so-so results, no processor uses value prediction
- Related topic is **address aliasing prediction**
 - RAW for load and store or WAW for 2 stores
- Address alias prediction is both more stable and simpler since need not actually predict the address values, only whether such values conflict
 - Has been used by a few processors

(Mis) Speculation on Pentium 4

% of micro-ops not used



Perspective

- 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
- Conservative in ideas, just faster clock and bigger
- 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

In Conclusion ...

- Interrupts and Exceptions either interrupt the current instruction or happen between instructions
 - Possibly large quantities of state must be saved before interrupting
- Machines with **precise exceptions** provide one single point in the program to restart execution
 - All instructions before that point have completed
 - No instructions after or including that point have completed
- Hardware techniques exist for precise exceptions even in the face of out-of-order execution!
 - Important enabling factor for out-of-order execution

break

Limits to ILP

- **Conflicting studies of amount**
 - Benchmarks (vectorized Fortran FP vs. integer C programs)
 - 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.

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**

Limits to ILP

Initial HW Model here; MIPS compilers.

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

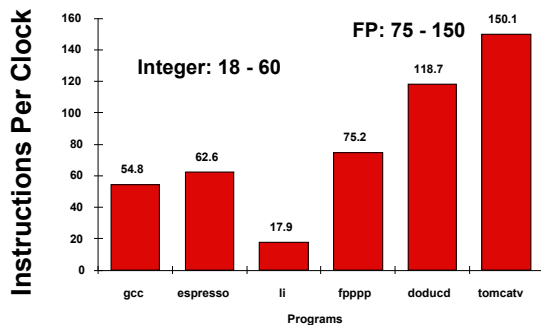
Also: perfect caches; 1 cycle latency for all instructions (FP *,/); unlimited instructions issued/clock cycle;

Limits to ILP HW Model comparison

	Model	Power 5
Instructions Issued per clock	Infinite	4
Instruction Window Size	Infinite	200
Renaming Registers	Infinite	48 integer + 40 Fl. Pt.
Branch Prediction	Perfect	2% to 6% misprediction (Tournament Branch Predictor)
Cache	Perfect	64KI, 32KD, 1.92MB L2, 36 MB L3
Memory Alias Analysis	Perfect	??

Upper Limit to ILP: Ideal Machine

Figure 3.1

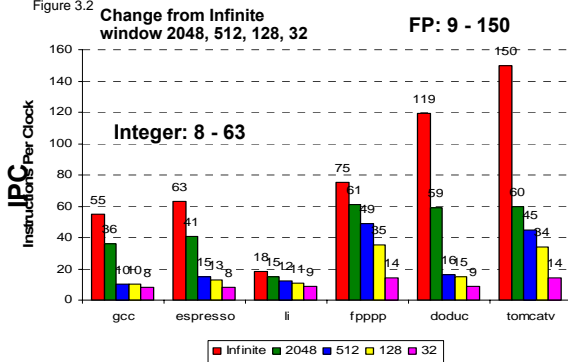


Limits to ILP HW Model comparison

	New Model	Model	Power 5
Instructions Issued per clock	Infinite	Infinite	4
Instruction Window Size	Infinite, 2K, 512, 128, 32	Infinite	200
Renaming Registers	Infinite	Infinite	48 integer + 40 Fl. Pt.
Branch Prediction	Perfect	Perfect	2% to 6% misprediction (Tournament Branch Predictor)
Cache	Perfect	Perfect	64KI, 32KD, 1.92MB L2, 36 MB L3
Memory Alias	Perfect	Perfect	??

More Realistic HW: Window Impact

Figure 3.2

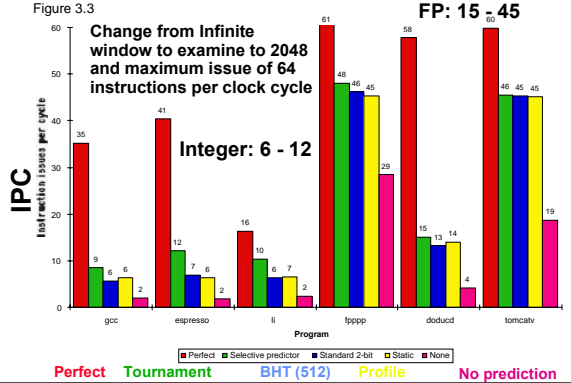


Limits to ILP HW Model comparison

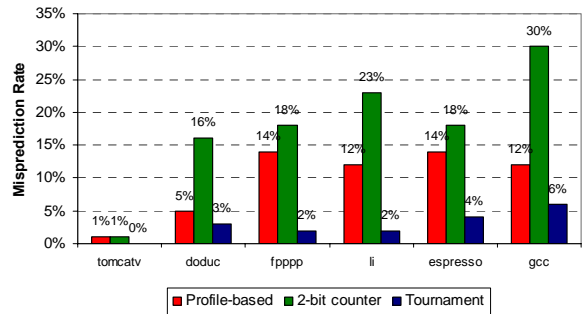
	New Model	Model	Power 5
Instructions Issued per clock	64	Infinite	4
Instruction Window Size	2048	Infinite	200
Renaming Registers	Infinite	Infinite	48 integer + 40 Fl. Pt.
Branch Prediction	Perfect vs. 8K Tournament vs. 512 2-bit vs. profile vs. none	Perfect	2% to 6% misprediction (Tournament Branch Predictor)
Cache	Perfect	Perfect	64KI, 32KD, 1.92MB L2, 36 MB L3
Memory Alias	Perfect	Perfect	??

More Realistic HW: Branch Impact

Figure 3.3



Misprediction Rates

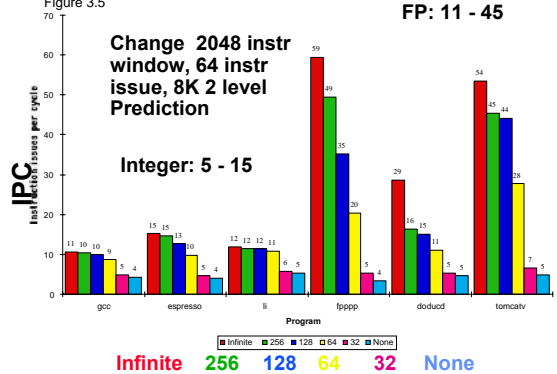


Limits to ILP HW Model comparison

	New Model	Model	Power 5
Instructions Issued per clock	64	Infinite	4
Instruction Window Size	2048	Infinite	200
Renaming Registers	Infinite v. 256, 128, 64, 32, none	Infinite	48 integer + 40 Fl. Pt.
Branch Prediction	8K 2-bit	Perfect	Tournament Branch Predictor
Cache	Perfect	Perfect	64KI, 32KD, 1.92MB L2, 36 MB L3
Memory Alias	Perfect	Perfect	Perfect

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

Figure 3.5

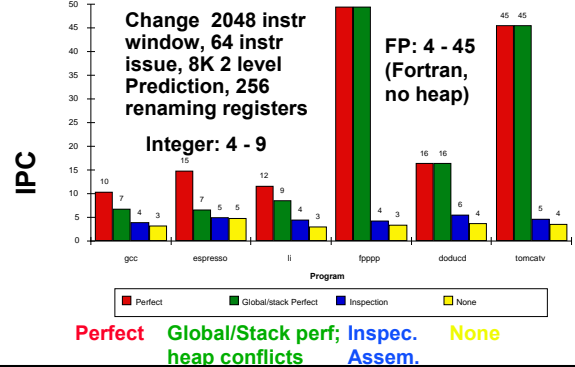


Limits to ILP HW Model comparison

	New Model	Model	Power 5
Instructions Issued per clock	64	Infinite	4
Instruction Window Size	2048	Infinite	200
Renaming Registers	256 Int + 256 FP	Infinite	48 integer + 40 Fl. Pt.
Branch Prediction	8K 2-bit	Perfect	Tournament
Cache	Perfect	Perfect	64KI, 32KD, 1.92MB L2, 36 MB L3
Memory Alias	Perfect v. Stack v. Inspec. v. none	Perfect	Perfect

More Realistic HW: Memory Address Alias Impact

Figure 3.6

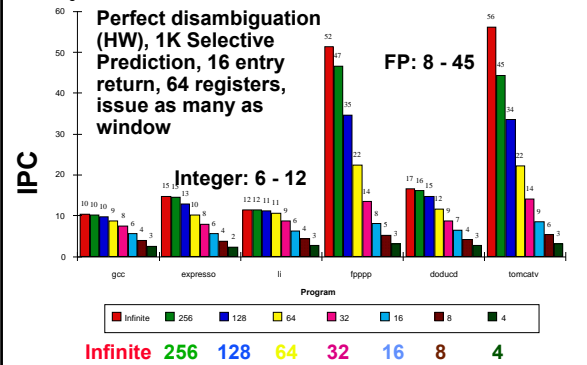


Limits to ILP HW Model comparison

	New Model	Model	Power 5
Instructions Issued per clock	64 (no restrictions)	Infinite	4
Instruction Window Size	Infinite vs. 256, 128, 64, 32	Infinite	200
Renaming Registers	64 Int + 64 FP	Infinite	48 integer + 40 Fl. Pt.
Branch Prediction	1K 2-bit	Perfect	Tournament
Cache	Perfect	Perfect	64KI, 32KD, 1.92MB L2, 36 MB L3
Memory Alias	HW disambiguation	Perfect	Perfect

Realistic HW: Window Impact

Figure 3.7



Limits to ILP (1)

- 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!

Limits to ILP (2)

- 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 = $f(\text{peak issue rate})$, and performance = $f(\text{sustained rate})$, growing gap between peak and sustained performance
 - ⇒ increasing energy per unit of performance

Reading

- **This lecture:**
 - chapter 2 *ILP*
 - chapter 3: 3.1-3.4 *Limits to ILP*
- **Next lecture:**
 - chapter 3: 3.5-3.8 *Simultaneous Multithreading (SMT)*
- **No class on Wed Oct 31st**
- **Wed Nov 14th 11.15-13.00h & 13.45-15.30h, room 402**

Lecture 6

Simultaneous Multithreading

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

Outline

- Thread Level Parallelism (TLP)
- Multithreading
- Simultaneous Multithreading (SMT)
- Power 4 vs. Power 5
- Head to Head: VLIW vs. Superscalar vs. SMT
- Commentary
- Conclusion

How to Exceed ILP Limits?

- 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

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

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

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

New Approach: Multithreaded 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 \approx 100s to 1000s of clocks
- **When switch?**
 - Alternate instruction per thread (fine grain)
 - When a thread is stalled, perhaps for a cache miss, another thread can be executed (coarse grain)

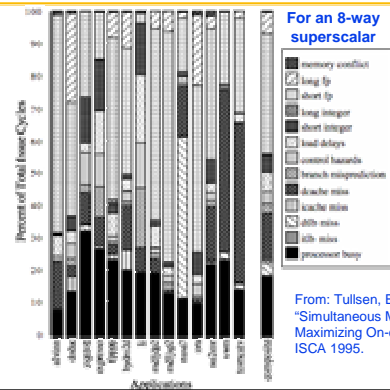
Fine-Grained Multithreading

- 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 is it can hide both short and long stalls, since instructions from other threads executed when one thread stalls
- Disadvantage is it 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 (will see later)

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 \ll stall time**
- Used in IBM AS/400

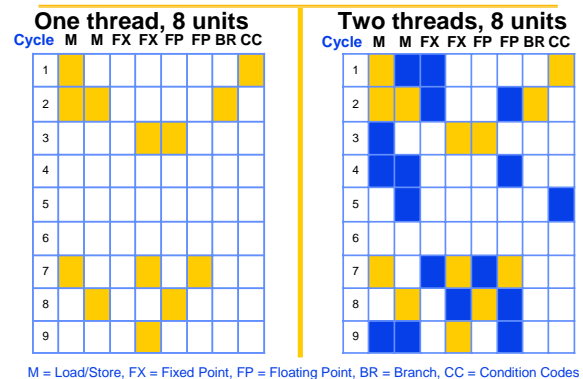
For most apps, most execution units lie idle



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?**

Simultaneous Multi-threading ...

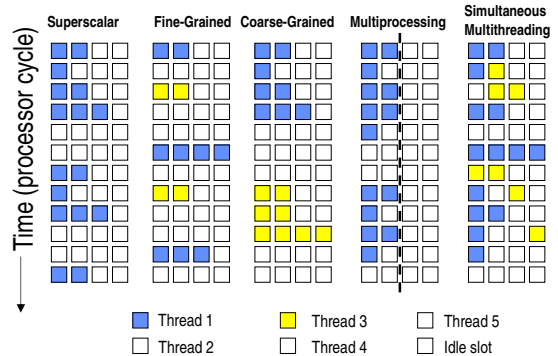


Simultaneous Multithreading (SMT)

- **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: Microprocessor Report, December 6, 1999
"Compuq Chooses SMT for Alpha"

Multithreaded Categories

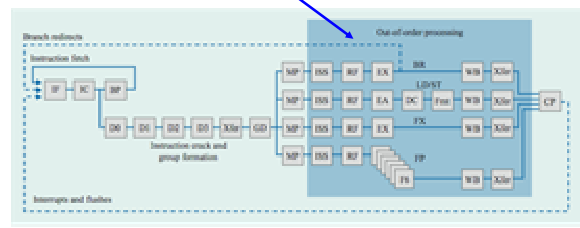


Design Challenges in SMT

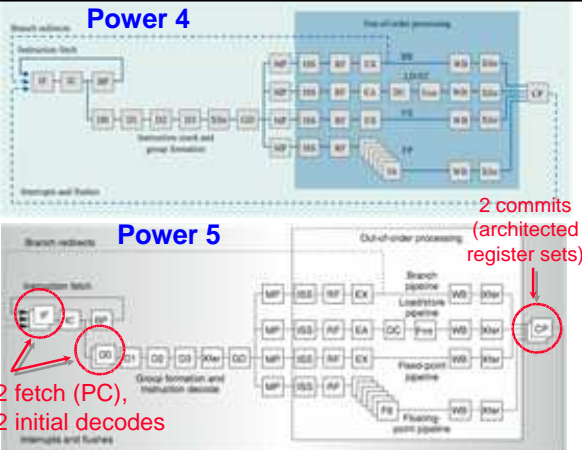
- **Since SMT makes sense only with fine-grained implementation, impact of fine-grained scheduling on single thread performance?**
 - A preferred thread approach sacrifices neither throughput nor single-thread performance?
 - Unfortunately, with a preferred thread, the processor is likely to sacrifice 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**

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

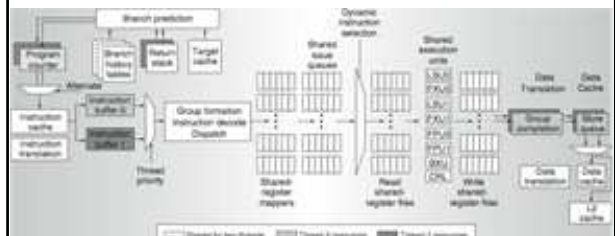


Power 5

2 fetch (PC),
2 initial decodes

2 commits (architected register sets)

Power 5 data flow ...

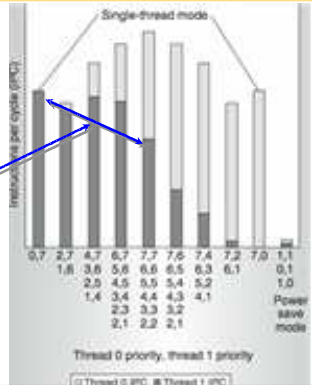


Why only 2 threads? With 4, one of the shared resources (physical registers, cache, memory bandwidth) would be prone to bottleneck

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

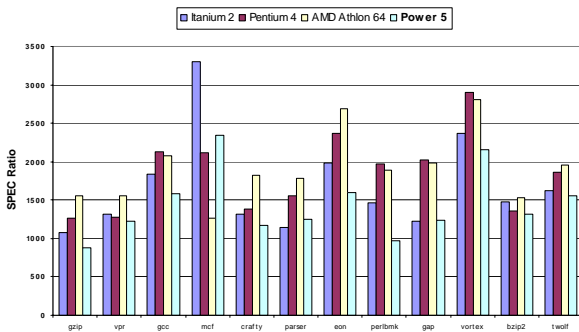
Initial Performance of SMT

- Pentium 4 Extreme SMT yields 1.01 speedup for SPECint_rate benchmark and 1.07 for SPECfp_rate
 - Pentium 4 is dual threaded SMT
 - SPECrate requires that each SPEC benchmark be run against a vendor-selected number of copies of the same benchmark
- Running on Pentium 4 each of 26 SPEC benchmarks paired with every other (26² runs) speed-ups from 0.90 to 1.58; average was 1.20
- Power 5, 8 processor server 1.23 faster for SPECint_rate with SMT, 1.16 faster for SPECfp_rate
- Power 5 running 2 copies of each app speedup between 0.89 and 1.41
 - Most gained some
 - Ft.Pt. apps had most cache conflicts and least gains

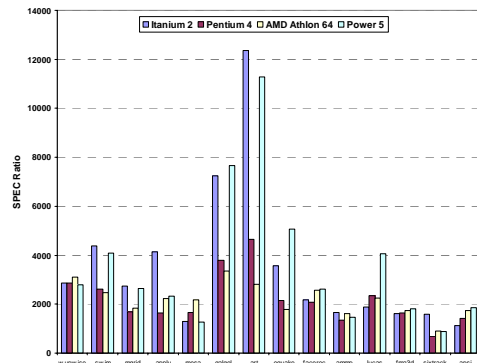
Head to Head ILP competition

Processor	Micro architecture	Fetch / Issue / Execute	Functional Units	Clock Rate (GHz)	Transistors, Die size	Power
Intel Pentium 4 Extreme	Speculative dynamically scheduled; deeply pipelined; SMT	3/3/4	7 int. 1 FP	3.8	125 M, 122 mm ²	115 W
AMD Athlon 64 FX-57	Speculative dynamically scheduled	3/3/4	6 int. 3 FP	2.8	114 M, 115 mm ²	104 W
IBM Power5 (1 CPU only)	Speculative dynamically scheduled; SMT; 2 CPU cores/chip	8/4/8	6 int. 2 FP	1.9	200 M, 300 mm ² (est.)	80W (est.)
Intel Itanium 2	Statically scheduled VLIW-style	6/5/11	9 int. 2 FP	1.6	592 M, 423 mm ²	130 W

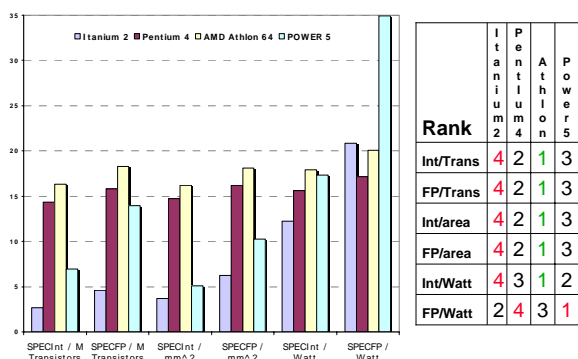
Performance on SPECint2000



Performance on SPECfp2000



Normalized Performance: Efficiency



Rank	Itanium 2	Pentium 4	Athlon 64	Power 5
Int/Trans	4	2	1	3
FP/Trans	4	2	1	3
Int/area	4	2	1	3
FP/area	4	2	1	3
Int/Watt	4	3	1	2
FP/Watt	2	4	3	1

No Silver Bullet for ILP

- No obvious over all leader in performance
- The AMD Athlon leads on SPECint performance followed by the Pentium 4, Itanium 2, and Power5
- Itanium 2 and Power5, which perform similarly on SPECFP, clearly dominate the Athlon and Pentium 4 on SPECFP
- Itanium 2 is the most inefficient processor both for FI. Pt. and integer code for all but one efficiency measure (SPECFP/Watt)
- Athlon and Pentium 4 both make good use of transistors and area in terms of efficiency,
- IBM Power5 is the most effective user of energy on SPECFP and essentially tied on SPECINT

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.
- Complexities of implementing these capabilities likely means sacrifices in 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!

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 = $f(\text{peak issue rate})$, and performance = $f(\text{sustained rate})$, growing gap between peak and sustained performance \Rightarrow increasing energy per unit of performance

Commentary

- Itanium architecture does **not** represent a significant breakthrough in scaling ILP or in avoiding the problems of complexity and power consumption
- Instead of pursuing more ILP, architects are increasingly focusing on TLP implemented with single-chip multiprocessors
- 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.
- Right balance of ILP and TLP is unclear today
 - Perhaps right choice for server market, which can exploit more TLP, may differ from desktop, where single-thread performance may continue to be a primary requirement

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 multithreading
 - Only on big stall vs. every clock cycle
- Simultaneous Multithreading if fine grained multithreading based on OOO superscalar microarchitecture
 - Instead of replicating registers, reuse rename registers
- Itanium/EPIC/VLIW is not a breakthrough in ILP
- Balance of ILP and TLP unclear in marketplace

Reading

- **This lecture:**
 - chapter 3: *Limits on ILP; Multithreading*
- **Next lecture:**
 - appendix F (on CD): *Vector processors*
 - start with chapter 4: *Multiprocessors*

Lecture 7

Vector Processors & Multiprocessor Introduction

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

Outline

- Vector Processors
- Vector Metrics, Terms

- Multiprocessing Motivation
- SISD v. SIMD v. MIMD
- Centralized vs. Distributed Memory
- Challenges to Parallel Programming

- Conclusion

Supercomputers

Definition of a supercomputer:

- Fastest machine in world at given task
- A device to turn a compute-bound problem into an I/O bound problem
- Any machine costing \$30M+
- Any machine designed by Seymour Cray

CDC6600 (Cray, 1964) regarded as first supercomputer

Supercomputer Applications

Typical application areas

- Military research (nuclear weapons, cryptography)
- Scientific research
- Weather forecasting
- Oil exploration
- Industrial design (car crash simulation)

All involve huge computations on large data sets

In 70s-80s, Supercomputer = Vector Machine

Vector Supercomputers

Epitomized by Cray-1, 1976:

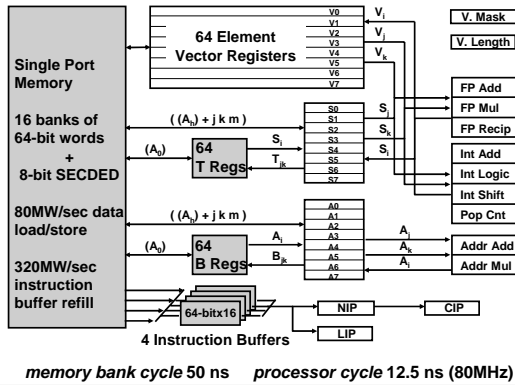
Scalar Unit + Vector Extensions

- Load/Store Architecture
- Vector Registers
- Vector Instructions
- Hardwired Control
- Highly Pipelined Functional Units
- Interleaved Memory System
- No Data Caches
- No Virtual Memory

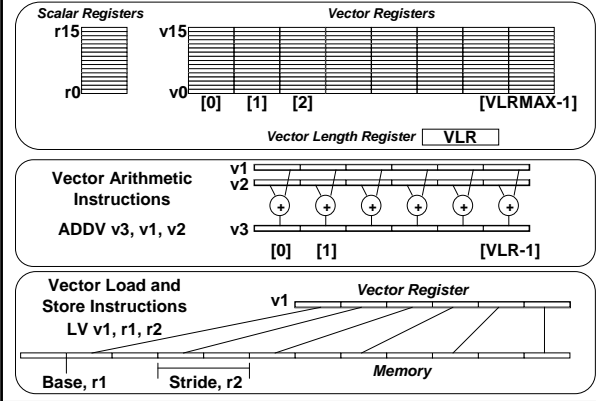
Cray-1 (1976)



Cray-1 (1976)



Vector Programming Model



Vector Code Example

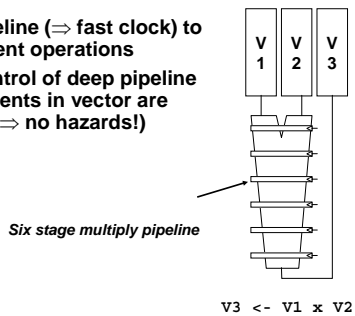
# C code	# Scalar Code	# Vector Code
for (i=0; i<64; i++)	LI R4, 64	LI VLR, 64
C[i] = A[i] + B[i];	loop:	LV V1, R1
	L.D F0, 0(R1)	LV V2, R2
	L.D F2, 0(R2)	ADDV.D V3, V1, V2
	ADD.D F4, F2, F0	SV V3, R3
	S.D F4, 0(R3)	
	DADDIU R1, 8	
	DADDIU R2, 8	
	DADDIU R3, 8	
	DSUBIU R4, 1	
	BNEZ R4, loop	

Vector Instruction Set Advantages

- **Compact**
 - one short instruction encodes N operations
- **Expressive, tells hardware that these N operations:**
 - are independent
 - use the same functional unit
 - access disjoint registers
 - access registers in the same pattern as previous instructions
 - access a contiguous block of memory (unit-stride load/store)
 - access memory in a known pattern (strided load/store)
- **Scalable**
 - can run same object code on more parallel pipelines or lanes

Vector Arithmetic Execution

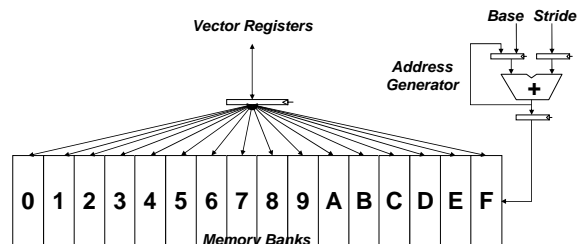
- Use deep pipeline (\Rightarrow fast clock) to execute element operations
- Simplifies control of deep pipeline because elements in vector are independent (\Rightarrow no hazards!)



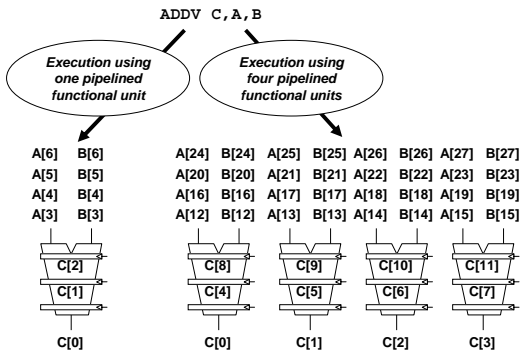
Vector Memory System

Cray-1, 16 banks, 4 cycle bank busy time, 12 cycle latency

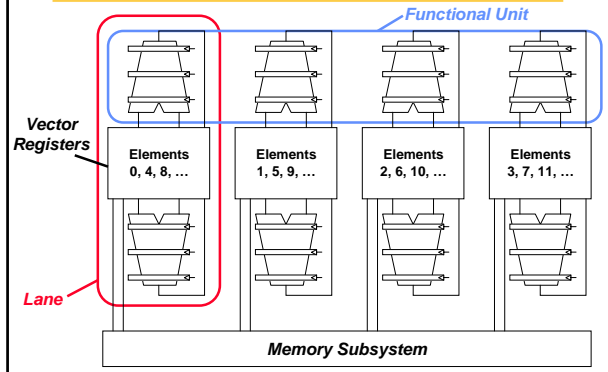
- **Bank busy time:** Cycles between accesses to same bank



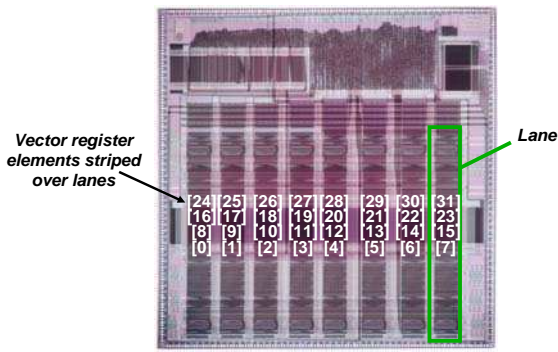
Vector Instruction Execution



Vector Unit Structure



T0 Vector Microprocessor (1995)



Vector Memory-Memory versus Vector Register Machines

- Vector memory-memory instructions hold all vector operands in main memory
- The first vector machines, CDC Star-100 ('73) and TI ASC ('71), were memory-memory machines
- Cray-1 ('76) was first vector register machine

Example Source Code

```
for (i=0; i<N; i++)
{
  C[i] = A[i] + B[i];
  D[i] = A[i] - B[i];
}
```

Vector Memory-Memory Code

```
ADDV C, A, B
SUBV D, A, B
```

Vector Register Code

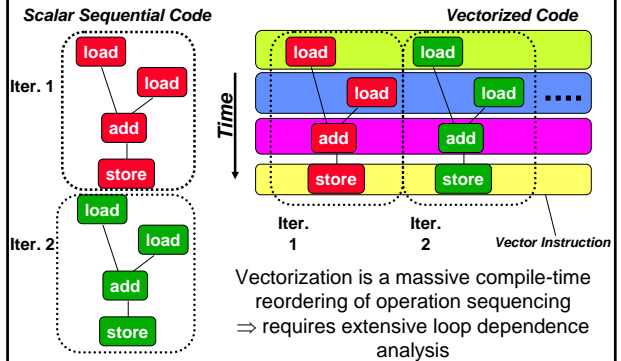
```
LV V1, A
LV V2, B
ADDV V3, V1, V2
SV V3, C
SUBV V4, V1, V2
SV V4, D
```

Vector Memory-Memory vs. Vector Register Machines

- Vector memory-memory architectures (VMMMA) require greater main memory bandwidth, why?
 - All operands must be read in and out of memory
 - VMMMA make it difficult to overlap execution of multiple vector operations, why?
 - Must check dependencies on memory addresses
 - VMMMA incur greater startup latency
 - Scalar code was faster on CDC Star-100 for vectors < 100 elements
 - For Cray-1, vector/scalar breakeven point was around 2 elements
- ⇒ Apart from CDC follow-ons (Cyber-205, ETA-10) all major vector machines since Cray-1 have had vector register architectures
- (we ignore vector memory-memory from now on)

Automatic Code Vectorization

```
for (i=0; i < N; i++)
  C[i] = A[i] + B[i];
```



Benchmark name	Operations executed in vector mode, compiler-optimized	Operations executed in vector mode, hand-optimized	Speedup from hand optimization
BDNA	96.1%	97.2%	1.52
MG3D	95.1%	94.5%	1.00
FLO52	91.5%	88.7%	N/A
ARC3D	91.1%	92.0%	1.01
SPEC77	90.3%	90.4%	1.07
MDG	87.7%	94.2%	1.49
TRFD	69.8%	73.7%	1.67
DYFESM	68.8%	65.6%	N/A
ADM	42.9%	59.6%	3.60
OCEAN	42.8%	91.2%	3.92
TRACK	14.4%	54.6%	2.52
SPICE	11.5%	79.9%	4.06
QCD	4.2%	75.1%	2.15

Figure F.14 Level of vectorization among the Perfect Club benchmarks when executed on the Cray Y-MP [Vajapeyam 1991]. The first column shows the vectorization level obtained with the compiler, while the second column shows the results after the codes have been hand-optimized by a team of Cray Research programmers. Speedup numbers are not available for FLO52 and DYFESM, as the hand-optimized runs used larger data sets than the compiler-optimized runs.

Processor	Compiler	Completely vectorized	Partially vectorized	Not vectorized
CDC CYBER 205	VAST-2 V2.21	62	5	33
Convex C-series	FC5.0	69	5	26
Cray X-MP	CFT77 V3.0	69	3	28
Cray X-MP	CFT V1.15	50	1	49
Cray-2	CFT2 V3.1a	27	1	72
ETA-10	FTN 77 V1.0	62	7	31
Hitachi S810/820	FORT77/HAP V20-2B	67	4	29
IBM 3090/VF	VS FORTRAN V2.4	52	4	44
NEC SX/2	FORTTRAN77 / SX V.040	66	5	29

Figure F.15 Result of applying vectorizing compilers to the 100 FORTRAN test kernels. For each processor we indicate how many loops were completely vectorized, partially vectorized, and unvectorized. These loops were collected by Callahan, Dongarra, and Levine [1988]. Two different compilers for the Cray X-MP show the large dependence on compiler technology.

Vector Stripmining

Problem: Vector registers have finite length
Solution: Break loops into pieces that fit into vector registers, "Stripmining"

```

for (i=0; i<N; i++)
  C[i] = A[i]+B[i];

```

Remainder
64 elements

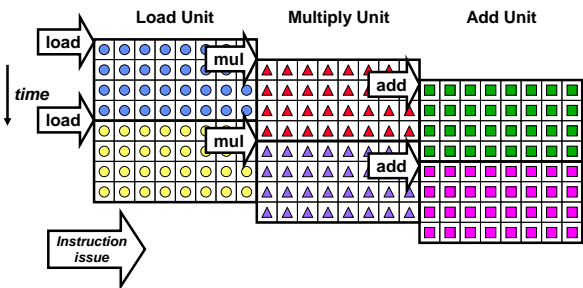
```

ANDI R1, N, 63 # N mod 64
MTC1 VLR, R1 # Do remainder
loop:
LV V1, RA
DSL R2, R1, 3 # Multiply by 8
DADDU RA, RA, R2 # Bump pointer
LV V2, RB
DADDU RB, RB, R2
ADDV.D V3, V1, V2
SV V3, RC
DADDU RC, RC, R2
DSUBU N, N, R1 # Subtract elements
LI R1, 64
MTC1 VLR, R1 # Reset full length
BGTZ N, loop # Any more to do?

```

Vector Instruction Parallelism

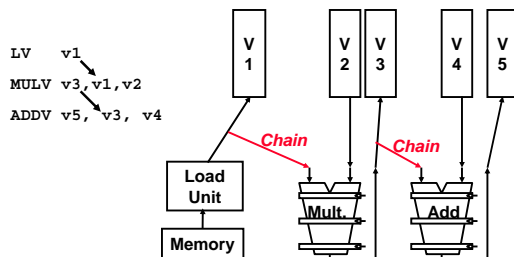
Can overlap execution of multiple vector instructions
 - example machine has 32 elements per vector register and 8 lanes



Complete 24 operations/cycle while issuing 1 short instruction/cycle

Vector Chaining

- Vector version of register bypassing
 - introduced with Cray-1

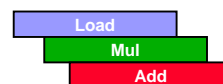


Vector Chaining Advantage

- Without chaining, must wait for last element of result to be written before starting dependent instruction



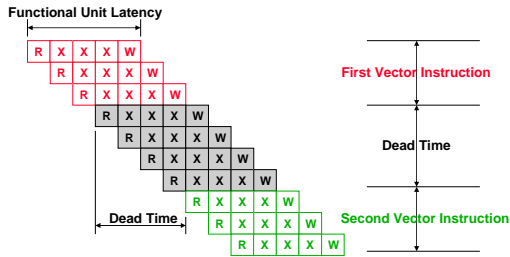
- With chaining, can start dependent instruction as soon as first result appears



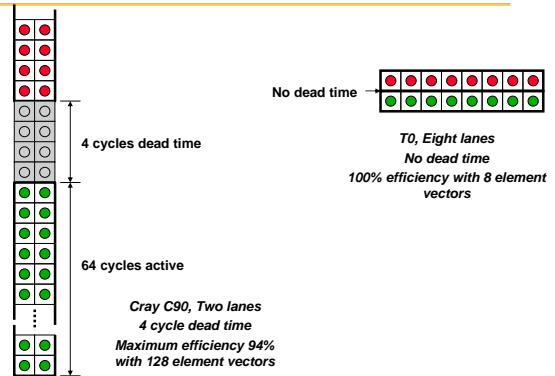
Vector Startup

Two components of vector startup penalty

- functional unit latency (time through pipeline)
- dead time or recovery time (time before another vector instruction can start down pipeline)



Dead Time and Short Vectors



Vector Scatter/Gather

Want to vectorize loops with indirect accesses:

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

Indexed load instruction (*Gather*)

```
LV vD, rD      # Load indices in D vector
LVI vC, rC, vD # Load indirect from rC base
LV vB, rB      # Load B vector
ADDV.D vA, vB, vC # Do add
SV vA, rA      # Store result
```

Vector Scatter/Gather

Scatter example:

```
for (i=0; i<N; i++)
    A[B[i]]++;
```

Is following a correct translation?

```
LV vB, rB      # Load indices in B vector
LVI vA, rA, vB # Gather initial A values
ADDV vA, vA, 1 # Increment
SVI vA, rA, vB # Scatter incremented values
```

Vector Conditional Execution

Problem: Want to vectorize loops with conditional code:

```
for (i=0; i<N; i++)
    if (A[i]>0) then
        A[i] = B[i];
```

Solution: Add vector *mask* (or *flag*) registers

- vector version of predicate registers, 1 bit per element

...and *maskable* vector instructions

- vector operation becomes NOP at elements where mask bit is clear

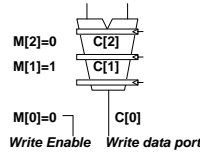
Code example:

```
CVM          # Turn on all elements
LV vA, rA    # Load entire A vector
SGTVS.D vA, F0 # Set bits in mask register where A>0
LV vA, rB    # Load B vector into A under mask
SV vA, rA    # Store A back to memory under mask
```

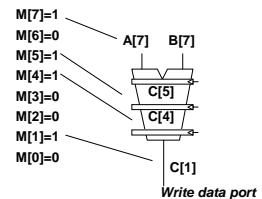
Masked Vector Instructions

Simple Implementation
execute all N operations, turn off result writeback according to mask

```
M[7]=1 A[7] B[7]
M[6]=0 A[6] B[6]
M[5]=1 A[5] B[5]
M[4]=1 A[4] B[4]
M[3]=0 A[3] B[3]
```

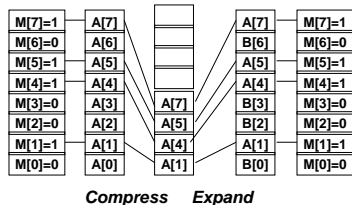


Density-Time Implementation
scan mask vector and only execute elements with non-zero masks



Compress/Expand Operations

- Compress packs non-masked elements from one vector register contiguously at start of destination vector register
 - population count of mask vector gives packed vector length
- Expand performs inverse operation



Used for density-time conditionals and also for general selection operations

Vector Reductions

Problem: Loop-carried dependence on reduction variables

```
sum = 0;
for (i=0; i<N; i++)
    sum += A[i]; # Loop-carried dependence on sum
```

Solution: Re-associate operations if possible, use binary tree to perform reduction

```
# Rearrange as:
sum[0:VL-1] = 0 # Vector of VL partial sums
for (i=0; i<N; i+=VL) # Stripmine VL-sized chunks
    sum[0:VL-1] += A[i:i+VL-1]; # Vector sum
# Now have VL partial sums in one vector register
do {
    VL = VL/2; # Halve vector length
    sum[0:VL-1] += sum[VL:2*VL-1] # Halve no. of partials
} while (VL>1)
```

A Modern Vector Super: NEC SX-6 (2003)

- CMOS Technology
 - 500 MHz CPU, fits on single chip
 - SDRAM main memory (up to 64GB)
- Scalar unit
 - 4-way superscalar with out-of-order and speculative execution
 - 64KB I-cache and 64KB data cache
- Vector unit
 - 8 foreground VRegs + 64 background VRegs (256x64-bit elements/VReg)
 - 1 multiply unit, 1 divide unit, 1 add/shift unit, 1 logical unit, 1 mask unit
 - 8 lanes (8 GFLOPS peak, 16 FLOPS/cycle)
 - 1 load & store unit (32x8 byte accesses/cycle)
 - 32 GB/s memory bandwidth per processor
- SMP structure
 - 8 CPUs connected to memory through crossbar
 - 256 GB/s shared memory bandwidth (4096 interleaved banks)



Multimedia Extensions

- Very short vectors added to existing ISAs for micros
- Usually 64-bit registers split into 2x32b or 4x16b or 8x8b
- Newer designs have 128-bit registers (AltiVec, SSE2)
- Limited instruction set:
 - no vector length control
 - no strided load/store or scatter/gather
 - unit-stride loads must be aligned to 64/128-bit boundary
- Limited vector register length:
 - requires superscalar dispatch to keep multiply/add/load units busy
 - loop unrolling to hide latencies increases register pressure
- Trend towards fuller vector support in microprocessors

Properties of Vector Processors

- Each result independent of previous result
 - => long pipeline, compiler ensures no dependencies
 - => high clock rate
- Vector instructions access memory with known pattern
 - => highly interleaved memory
 - => amortize memory latency of over - 64 elements
 - => no (data) caches required! (Do use instruction cache)
- Reduces branches and branch problems in pipelines
- Single vector instruction implies lots of work (- loop)
 - => fewer instruction fetches

Operation & Instruction Count: RISC v. Vector Processor

(from F. Quintana, U. Barcelona.)

Spec92fp Program	Operations (Millions)			Instructions (M)		
	RISC	Vector	R / V	RISC	Vector	R / V
swim256	115	95	1.1x	115	0.8	142x
hydro2d	58	40	1.4x	58	0.8	71x
nasa7	69	41	1.7x	69	2.2	31x
su2cor	51	35	1.4x	51	1.8	29x
tomcatv	15	10	1.4x	15	1.3	11x
wave5	27	25	1.1x	27	7.2	4x
mdljdp2	32	52	0.6x	32	15.8	2x

Vector reduces ops by 1.2X, instructions by 20X

Common Vector Metrics

- R_{∞} : MFLOPS rate on an infinite-length vector
 - vector "speed of light"
 - Real problems do not have unlimited vector lengths, and the start-up penalties encountered in real problems will be larger
 - (R_n is the MFLOPS rate for a vector of length n)
- $N_{1/2}$: The vector length needed to reach one-half of R_{∞}
 - a good measure of the impact of start-up
- N_V : The vector length needed to make vector mode faster than scalar mode
 - measures both start-up and speed of scalars relative to vectors, quality of connection of scalar unit to vector unit

Vector Execution Time

- Time = f (vector length, data dependencies, struct. hazards)
- **Initiation rate**: rate that FU consumes vector elements (= number of lanes; usually 1 or 2 on Cray T-90)
- **Convoy**: set of vector instructions that can begin execution in same clock (no struct. or data hazards)
- **Chime**: approx. time for a vector operation
- **m convoys take m chimes**; if each vector length is n , then they take approx. $m \times n$ clock cycles (ignores overhead; good approximation for long vectors)

```

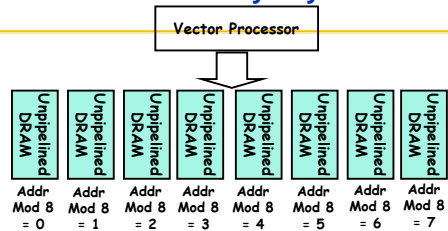
1: LV  V1,Rx ;load vector X
2: MULV V2,F0,V1 ;vector-scalar mult.
LV  V3,Ry ;load vector Y
3: ADDV V4,V2,V3 ;add
4: SV  Ry,V4 ;store the result
    
```

4 convoys, 1 lane, VL=64
 $\Rightarrow 4 \times 64 = 256$ clocks
 (or 4 clocks per result)

Memory operations

- Load/store operations move groups of data between registers and memory
- Three types of addressing
 - **Unit stride**
 - » Contiguous block of information in memory
 - » Fastest: always possible to optimize this
 - **Non-unit (constant) stride**
 - » Harder to optimize memory system for all possible strides
 - » Prime number of data banks makes it easier to support different strides at full bandwidth
 - **Indexed (gather-scatter)**
 - » Vector equivalent of register indirect
 - » Good for sparse arrays of data
 - » Increases number of programs that vectorize

Interleaved Memory Layout



- **Great for unit stride:**
 - Contiguous elements in different DRAMs
 - Startup time for vector operation is latency of single read
- **What about non-unit stride?**
 - Above good for strides that are relatively prime to 8
 - Bad for: 2, 4
 - Better: prime number of banks...!

How to get full bandwidth for Unit Stride?

- Memory system must sustain (# lanes x word) /clock
 - No. memory banks > memory latency to avoid stalls
 - m banks $\Rightarrow m$ words per memory latency l clocks
 - if $m < l$, then gap in memory pipeline:
- clock: 0 ... l $l+1$ $l+2$... $l+m-1$ $l+m$... $2l$
- word: -- ... 0 1 2 ... $m-1$ -- ... m
- may have 1024 banks in SRAM
- If desired throughput greater than one word per cycle
 - Either more banks (start multiple requests simultaneously)
 - Or wider DRAMS. Only good for unit stride or large data types
 - More banks/weird numbers of banks good to support more strides at full bandwidth
 - How to do prime number of banks efficiently?

Vectors Are Inexpensive

Scalar

- N ops per cycle
 $\Rightarrow O(N^2)$ circuitry
- HP PA-8000
 - 4-way issue
 - reorder buffer: 850K transistors
 - incl. 6,720 5-bit register number comparators

Vector

- N ops per cycle
 $\Rightarrow O(N + \epsilon N^2)$ circuitry
- T0 vector micro
 - 24 ops per cycle
 - 730K transistors total
 - only 23 5-bit register number comparators
 - No floating point

Vectors Lower Power

Single-issue Scalar

- One instruction fetch, decode, dispatch per operation
- Arbitrary register accesses, adds area and power
- Loop unrolling and software pipelining for high performance increases instruction cache footprint
- All data passes through cache; waste power if no temporal locality
- One TLB lookup per load or store
- Off-chip access in whole cache lines

Vector

- One inst fetch, decode, dispatch per vector
- Structured register accesses
- Smaller code for high performance, less power in instruction cache misses
- Bypass cache
- One TLB lookup per group of loads or stores
- Move only necessary data across chip boundary

Superscalar Energy Efficiency Even Worse

Superscalar

- Control logic grows quadratically with issue width
- Control logic consumes energy regardless of available parallelism
- Speculation to increase visible parallelism wastes energy

Vector

- Control logic grows linearly with issue width
- Vector unit switches off when not in use
- Vector instructions expose parallelism without speculation
- Software control of speculation when desired:
 - Whether to use vector mask or compress/expand for conditionals

Vector Applications

Limited to scientific computing?

- Multimedia Processing (compress., graphics, audio synth, image proc.)
- Standard benchmark kernels (Matrix Multiply, FFT, Convolution, Sort)
- Lossy Compression (JPEG, MPEG video and audio)
- Lossless Compression (Zero removal, RLE, Differencing, LZW)
- Cryptography (RSA, DES/IDEA, SHA/MD5)
- Speech and handwriting recognition
- Operating systems/Networking (memcpy, memset, parity, checksum)
- Databases (hash/join, data mining, image/video serving)
- Language run-time support (stdlib, garbage collection)
- even SPECint95

Older Vector Machines

Machine	Year	Clock	Regs	Elements	FUs	LSUs
Cray 1	1976	80 MHz	8	64	6	1
Cray XMP	1983	120 MHz	8	64	8	2 L, 1 S
Cray YMP	1988	166 MHz	8	64	8	2 L, 1 S
Cray C-90	1991	240 MHz	8	128	8	4
Cray T-90	1996	455 MHz	8	128	8	4
Convex C-1	1984	10 MHz	8	128	4	1
Convex C-4	1994	133 MHz	16	128	3	1
Fuj. VP200	1982	133 MHz	8-256	32-1024	3	2
Fuj. VP300	1996	100 MHz	8-256	32-1024	3	2
NEC SX/2	1984	160 MHz	8+8K	256+var	16	8
NEC SX/3	1995	400 MHz	8+8K	256+var	16	8

Newer Vector Computers

- Cray X1
 - MIPS like ISA + Vector in CMOS
- NEC Earth Simulator
 - Fastest computer in world for 3 years; 40 TFLOPS
 - 640 CMOS vector nodes

Key Architectural Features of X1

New vector instruction set architecture (ISA)

- Much larger register set (32x64 vector, 64+64 scalar)
- 64- and 32-bit memory and IEEE arithmetic
- Based on 25 years of experience compiling with Cray1 ISA

Decoupled Execution

- Scalar unit runs ahead of vector unit, doing addressing and control
- Hardware dynamically unrolls loops, and issues multiple loops concurrently
- Special sync operations keep pipeline full, even across barriers
- ⇒ Allows the processor to perform well on short nested loops

Scalable, distributed shared memory (DSM) architecture

- Memory hierarchy: caches, local memory, remote memory
- Low latency, load/store access to entire machine (tens of TBs)
- Processors support 1000's of outstanding refs with flexible addressing
- Very high bandwidth network
- Coherence protocol, addressing and synchronization optimized for DM

Cray X1E Mid-life Enhancement

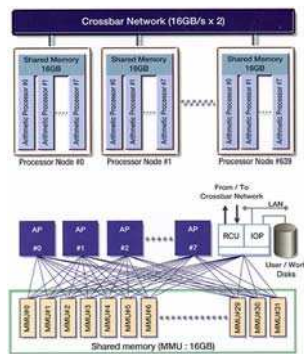
- **Technology refresh of the X1 (0.13 μ m)**
 - ~50% faster processors
 - Scalar performance enhancements
 - Doubling processor density
 - Modest increase in memory system bandwidth
 - Same interconnect and I/O
- **Machine upgradeable**
 - Can replace Cray X1 nodes with X1E nodes

ESS – configuration of a general purpose supercomputer

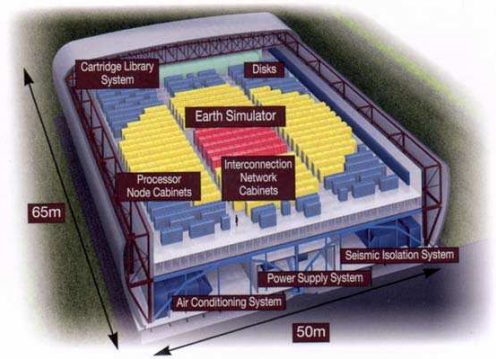
1. **Processor Nodes (PN)** Total number of processor nodes is 640. Each processor node consists of eight vector processors of 8 GFLOPS and 16GB shared memories. Therefore, total numbers of processors is 5,120 and total peak performance and main memory of the system are 40 TFLOPS and 10 TB, respectively. Two nodes are installed into one cabinet, which size is 40"x56"x80". 16 nodes are in a cluster. Power consumption per cabinet is approximately 20 KVA.
2. **Interconnection Network (IN):** Each node is coupled together with more than 83,000 copper cables via single-stage crossbar switches of 16GB/s x2 (Load + Store). The total length of the cables is approximately 1,800 miles.
3. **Hard Disk.** Raid disks are used for the system. The capacities are 450 TB for the systems operations and 250 TB for users.
4. **Mass Storage system:** 12 Automatic Cartridge Systems (STK PowderHorn9310); total storage capacity is approximately 1.6 PB.

From Horst D. Simon, NERSC/LBNL, May 15, 2002, "ESS Rapid Response Meeting"

Earth Simulator



Earth Simulator Building



ESS – complete system installed 4/1/2002



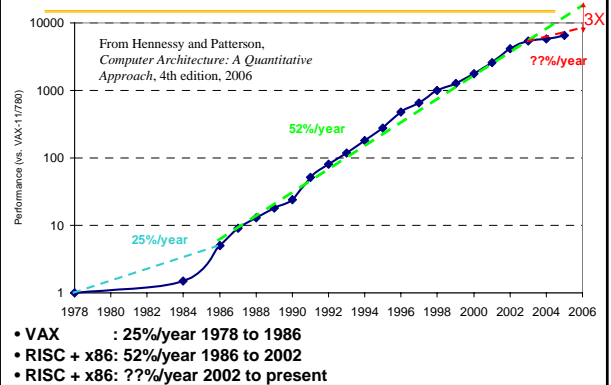
Vector Summary

- Vector is alternative model for exploiting ILP
- If code is vectorizable, then simpler hardware, more energy efficient, and better real-time model than Out-of-order machines
- Design issues include number of lanes, number of functional units, number of vector registers, length of vector registers, exception handling, conditional operations
- Fundamental design issue is memory bandwidth
 - With virtual address translation and caching
- Will multimedia popularity revive vector architectures?

Outline

- Vector Processors
- Vector Metrics, Terms
- Multiprocessing Motivation
- SISD v. SIMD v. MIMD
- Centralized vs. Distributed Memory
- Challenges to Parallel Programming
- Conclusion

Uniprocessor Performance (SPECint)



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

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

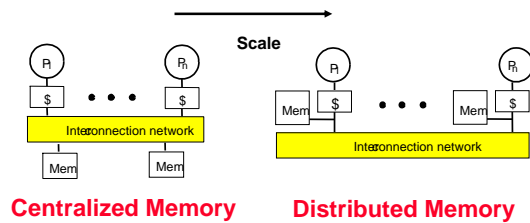
Single Instruction Single Data (SISD) (Uniprocessor)	Single Instruction Multiple Data SIMD (single PC: Vector, CM-2)
Multiple Instruction Single Data (MISD) (???)	Multiple Instruction Multiple Data MIMD (Clusters, SMP servers)

- 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

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
- Two 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

Centralized vs. Distributed Memory



Centralized Memory Multiprocessor

- Also called **symmetric multiprocessors (SMPs)** because single main memory has a symmetric relationship to all processors
- Large caches \Rightarrow 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

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

Two 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

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%

Amdahl's Law Answers

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

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

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

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

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

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

CPI Equation

$$\begin{aligned} \text{CPI} &= \text{Base CPI} + \\ &\quad \text{Remote request rate} \times \text{Remote request cost} \\ &= 0.5 + 0.2\% \times 400 = 0.5 + 0.8 = 1.3 \end{aligned}$$

No communication is 1.3/0.5 or 2.6 faster than 0.2% instructions involve remote access

And in Conclusion [1/2] ...

- One instruction operates on vectors of data
- Vector loads get data from memory into big register files, operate, and then vector store
- E.g., Indexed load, store for sparse matrix
- Easy to add vector to commodity instruction set
 - E.g., Morph SIMD into vector
- Vector is very efficient architecture for vectorizable codes, including multimedia and many scientific codes

And in Conclusion [2/2] ...

- “End” of uniprocessors speedup => Multiprocessors
- Parallelism challenges: % parallelizable, long latency to remote memory
- Centralized vs. distributed memory
 - Small MP vs. lower latency, larger BW for Larger MP
- Message Passing vs. Shared Address
 - Uniform access time vs. Non-uniform access time

Reading and Schedule

- This lecture:
 - Appendix F: *Vector Processors*
 - Chapter 4: *4.1 Introduction Multiprocessors*
- Next week, Oct 31st: **No class**
- Next lecture, Nov 7th: *remainder of chapter 4 (in the afternoon **feedback** on assignment 2a)*
- On Wed Nov 14th both at 11.15-13.00h and at **13.45-15.30h** lectures in room 402

Lecture 8

Snooping Cache Based Multiprocessors

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

Review

- 1 instruction operates on vectors of data
- Vector loads get data from memory into big register files, operate, and then vector store
- E.g., Indexed load, store for sparse matrix
- Easy to add vector to commodity instruction set
 - E.g., Morph SIMD into vector
- Vector is very efficient architecture for vectorizable codes, including multimedia and many scientific codes
- “End” of uniprocessors speedup \Rightarrow Multiprocessors
- Parallelism challenges: % parallelizable, long latency to remote memory
- Centralized vs. distributed memory
 - Small MP vs. lower latency, larger BW for larger MP
- Message Passing vs. Shared Address
 - Uniform access time vs. Non-uniform access time

Outline

- Review
- Coherence
- Write Consistency
- Snooping
- Building Blocks
- Snooping protocols and examples
- Coherence traffic and Performance on MP
- Conclusion

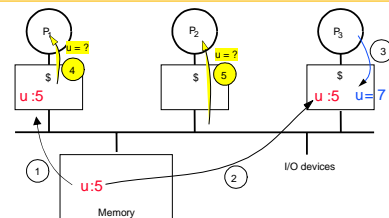
Challenges of Parallel Processing

1. Application parallelism \Rightarrow primarily via new algorithms that have better parallel performance
 2. Long remote latency impact \Rightarrow 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

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
 - \Rightarrow reduces latency to shared data, memory bandwidth for shared data, and interconnect bandwidth
 - \Rightarrow cache coherence problem

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
 - » Processes accessing main memory may see very stale value
- Unacceptable for programming, and its frequent!

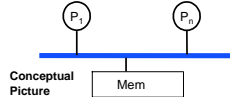
Example

```

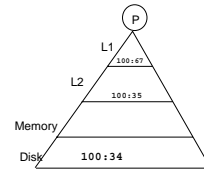
P1                               P2
-----                             -----
/* Assume initial value of A and flag is 0 */
A = 1;                               while (flag == 0); /* spin idly */
flag = 1;                             print A;

```

- 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



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

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

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

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

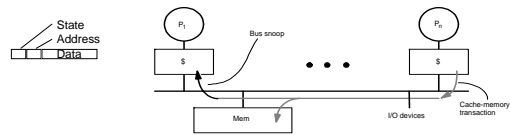
Outline

- Review
- Coherence
- Write Consistency
- Snooping
- Building Blocks
- Snooping protocols and examples
- Coherence traffic and Performance on MP
- Conclusion

Two Classes of Cache Coherence Protocols

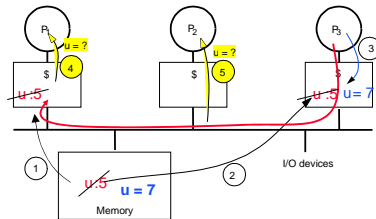
1. **Directory based** — Sharing status of a block of physical memory is kept in just one location, the **directory**
2. **Snooping** — Every cache with a copy of data also has a copy of sharing status of block, but no centralized state is kept
 - All caches are accessible via some broadcast medium (a 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

Snooping 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

Example: Write-through 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

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 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

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 \Rightarrow 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)

Cache behavior in response to bus

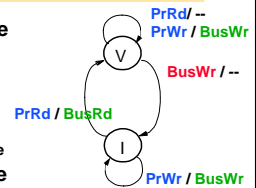
- Every bus transaction must check the cache-address tags
 - could potentially interfere with processor cache accesses
- A way to reduce interference is to duplicate tags
 - One set for caches access, one set for bus accesses
- Another way to reduce interference is to use L2 tags
 - Since L2 less heavily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

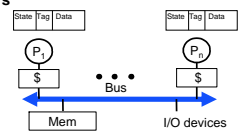
- Snooping coherence protocol is usually implemented by incorporating a finite-state controller in each node
- Logically, think of a separate controller associated with each cache block
 - That is, snooping operations or cache requests for different blocks can proceed independently
- In implementations, 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



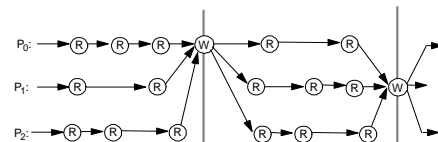
PrRd: Processor Read
PrWr: Processor Write
BusRd: Bus Read
BusWr: Bus Write



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)
 - \Rightarrow 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

step	P1			P2			Bus			Memory		
	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

Example

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

Assumes A1 and A2 map to same cache block

Example

step	P1			P2			Bus			Memory		
	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												
P2: Write 20 to A1												
P2: Write 40 to A2												

Assumes A1 and A2 map to same cache block

Example

step	P1			P2			Bus			Memory		
	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	10		RdMs	P2	A1		
P2: Write 20 to A1								WrBk	P1	A1	10	10
P2: Write 40 to A2								RdDa	P2	A1	10	10

Assumes A1 and A2 map to same cache block

Example

step	P1			P2			Bus			Memory		
	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	10		RdMs	P2	A1		
P2: Write 20 to A1								WrBk	P1	A1	10	10
P2: Write 40 to A2				Shar.	A1	10		RdDa	P2	A1	10	10
				Excl.	A1	20		WrMs	P2	A1		

Assumes A1 and A2 map to same cache block

Example

step	P1			P2			Bus			Memory		
	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	10		RdMs	P2	A1		
P2: Write 20 to A1								WrBk	P1	A1	10	10
P2: Write 40 to A2				Shar.	A1	10		RdDa	P2	A1	10	10
				Excl.	A1	20		WrMs	P2	A1		
								WrBk	P2	A2	20	20
				Excl.	A2	40						

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

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**

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

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

Outline

- Review
- Coherence
- Write Consistency
- Snooping
- Building Blocks
- Snooping protocols and examples
- Coherence traffic and Performance on MP
- Conclusion

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

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

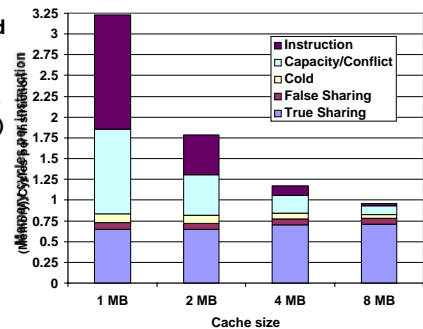
Example: True v. False Sharing v. Hit?

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

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

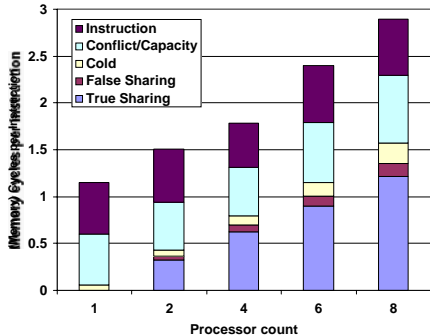
True sharing and false sharing unchanged going from 1 MB to 8 MB (L3 cache)



Uniprocessor cache misses improve with cache size increase (Instruction, Capacity/Conflict, Compulsory)

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

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



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

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 \Rightarrow Coherence (values returned by a read), Consistency (when a written value will be returned by a read)
- MPs are highly effective for multiprogrammed workloads
- MPs proved effective for intensive commercial workloads, such as OLTP (assuming enough I/O to be CPU-limited), DSS applications (where query optimization is critical), and large-scale, web searching applications

Reading and Schedule

- This lecture:
 - 4.2 *Symmetric Shared-Memory Architectures*
 - 4.3 *Performance of Symmetric Shared-Memory Multiprocessors*
- This afternoon: **feedback** on assignment 2a
- Next week, Nov 14th:
 - 11.15-13.00h: *directory-based MP & rest of chapter 4*
 - 13.45-15.30h: *chapter 5 memory hierarchy design*

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 \Rightarrow 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
- Cautionary Tale
- Sun T1 ("Niagara") Multiprocessor
- Microprocessor Comparison
- Conclusion

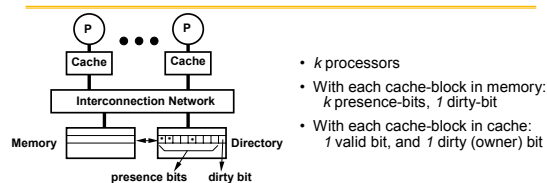
Bus-based Coherence

- 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



- 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 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 \Rightarrow 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 snoopy 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 \Rightarrow explicit invalidate & data fetch requests
- Note: on a write, a cache block is bigger, so need to read the full cache block

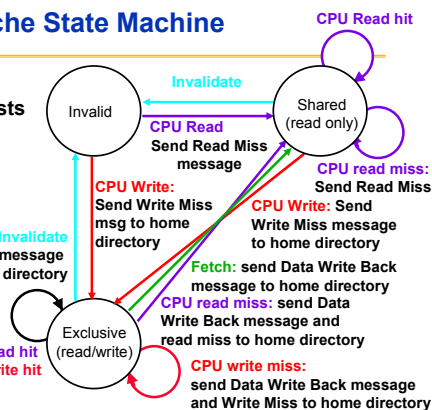
CPU -Cache State Machine

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

- Invalid state if in memory

Fetch/Invalidate
send Data Write Back message to home directory

CPU read hit
CPU write hit

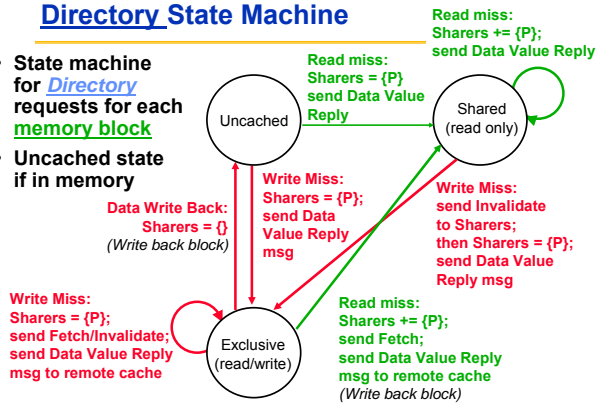


State Transition Diagram for Directory

- 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



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** \Rightarrow 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) \Rightarrow 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

step	P1		P2		Bus		Directory		Memory				
	State	Addr	Value	State	Addr	Action	Proc	Addr		Value	Addr	State	(Procs)
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

step	P1		P2		Bus		Directory		Memory				
	State	Addr	Value	State	Addr	Action	Proc	Addr		Value	Addr	State	(Procs)
P1: Write 10 to A1													
P1: Read A1	Excl.	A1	10			WtMs	P1	A1	0				
P2: Read A1						DaRd	P1	A1	0				
P2: Write 20 to A1													
P2: Write 40 to A2													

A1 and A2 map to the same cache block

Example

step	P1		P2		Bus		Directory		Memory				
	State	Addr	Value	State	Addr	Action	Proc	Addr		Value	Addr	State	(Procs)
P1: Write 10 to A1													
P1: Read A1	Excl.	A1	10			WtMs	P1	A1	0				
P2: Read A1						DaRd	P1	A1	0				
P2: Write 20 to A1													
P2: Write 40 to A2													

A1 and A2 map to the same cache block

Example

step	Processor 1			Processor 2			Interconnect			Directory			Memory	
	P1 State	Addr	Value	P2 State	Addr	Value	Action	Proc	Addr	Value	Addr	State	(Procs)	Value
P1: Write 10 to A1	Excl	A1	10				WrMs	P1	A1	0	A1	Ex	(P1)	
P1: Read A1	Excl	A1	10				DaRp	P1	A1	0				
P2: Read A1	Shar	A1	10	Shar	A1	10	RdMs	P2	A1	10	A1	Shar	(P1,P2)	10
P2: Write 20 to A1				Shar	A1	10	WrMs	P2	A1	10	A1	Shar	(P1,P2)	10
P2: Write 40 to A2							WrMs	P2	A2	40				

Write Back

A1 and A2 map to the same cache block

Example

step	Processor 1			Processor 2			Interconnect			Directory			Memory	
	P1 State	Addr	Value	P2 State	Addr	Value	Action	Proc	Addr	Value	Addr	State	(Procs)	Value
P1: Write 10 to A1	Excl	A1	10				WrMs	P1	A1	0	A1	Ex	(P1)	
P1: Read A1	Excl	A1	10				DaRp	P1	A1	0				
P2: Read A1	Shar	A1	10	Shar	A1	10	RdMs	P2	A1	10	A1	Shar	(P1,P2)	10
P2: Write 20 to A1				Shar	A1	10	WrMs	P2	A1	10	A1	Shar	(P1,P2)	10
P2: Write 40 to A2	Inv						WrMs	P2	A2	40				

A1 and A2 map to the same cache block

Example

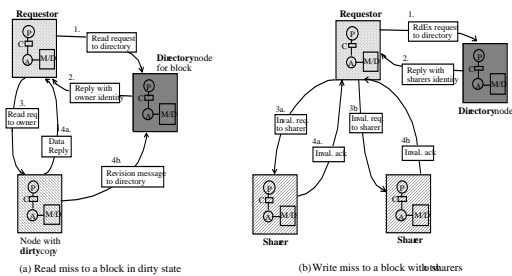
step	Processor 1			Processor 2			Interconnect			Directory			Memory	
	P1 State	Addr	Value	P2 State	Addr	Value	Action	Proc	Addr	Value	Addr	State	(Procs)	Value
P1: Write 10 to A1	Excl	A1	10				WrMs	P1	A1	0	A1	Ex	(P1)	
P1: Read A1	Excl	A1	10				DaRp	P1	A1	0				
P2: Read A1	Shar	A1	10	Shar	A1	10	RdMs	P2	A1	10	A1	Shar	(P1,P2)	10
P2: Write 20 to A1				Shar	A1	10	WrMs	P2	A1	10	A1	Shar	(P1,P2)	10
P2: Write 40 to A2	Inv						WrMs	P2	A2	40				

A1 and A2 map to the same cache block
(but different memory block addresses $A1 \neq A2$)

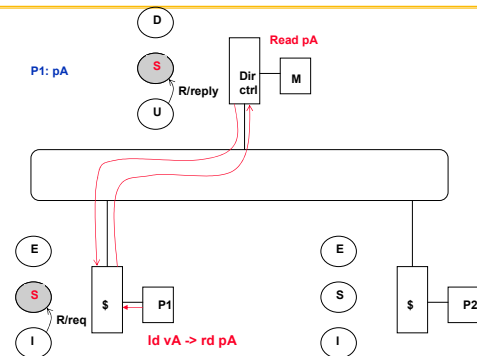
Implementing a Directory

- 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 in Exclusive: send data directly to requester from owner vs. 1st to memory and then from memory to requester

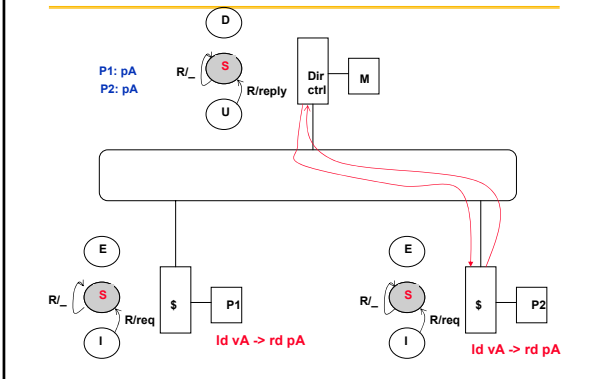
Basic Directory Transactions



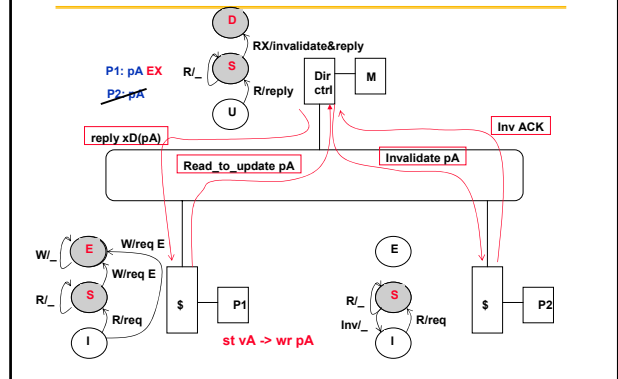
Example Directory Protocol (1st Read)



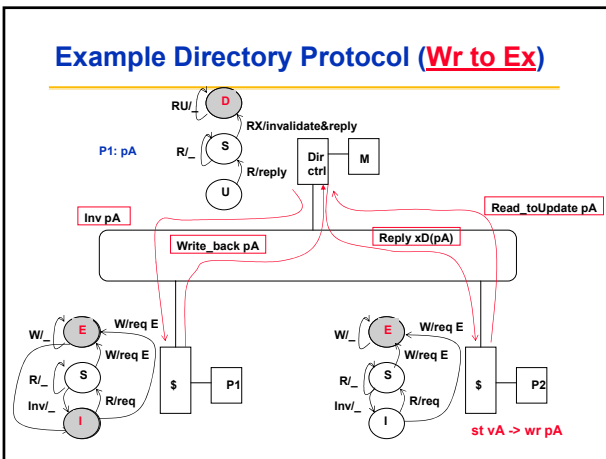
Example Directory Protocol (Read Share)



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


```
lockit:  li     R2,#1
          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;          .....
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 $\sim O(n^3)$, 100 P?
 - **Time-constrained scaling**: 1 hour $\Rightarrow 10^{12}n \Rightarrow 2.15n$ 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 \Rightarrow 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

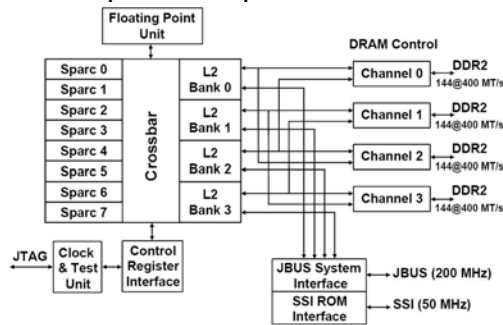
T1 ("Niagara")

- **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

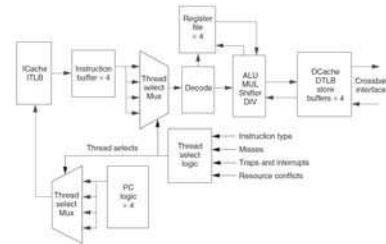
- Also ships with 6 or 4 processors



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

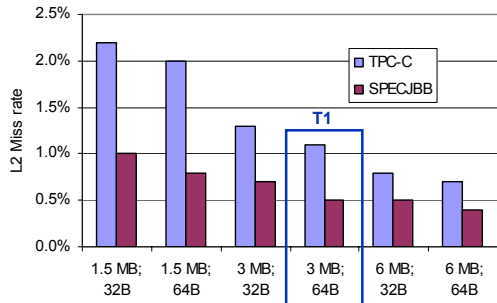
T1 Fine-Grained Multithreading

- 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

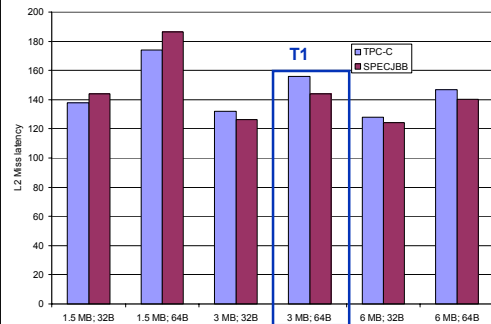
Memory, Clock, Power

- 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
- 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



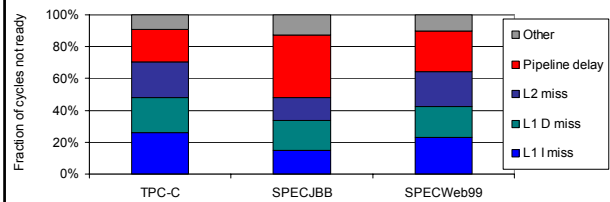
Miss Latency: L2 Cache Size, Block Size



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 JCPs)	615	471
Performance / Watt	1.922	0.361
S/WaP	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 commercial 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]

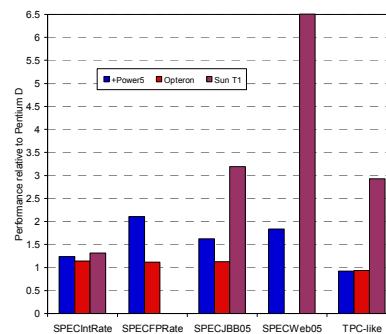
• Where is the compelling value to warrant taking such a risk?

• <http://h71028.www7.hp.com/ERC/cache/280124-0-0-0-121.html>

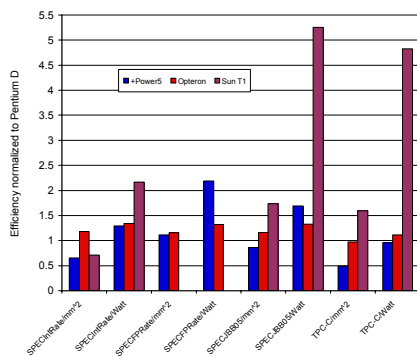
Microprocessor Comparison

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/mm², 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: *Memory Hierarchy Design*

Lecture 10

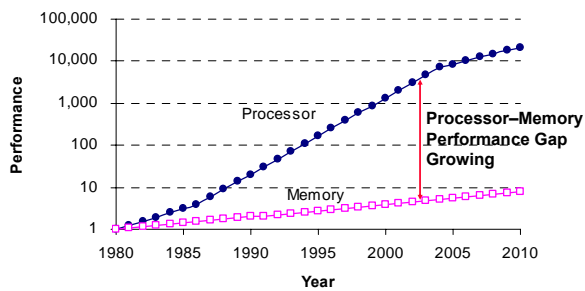
Advanced Memory Hierarchy

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

Outline

- 11 Advanced Cache Optimizations
- Memory Technology and DRAM optimizations
- Virtual Machines
- Xen VM: Design and Performance
- AMD Opteron Memory Hierarchy
- Opteron Memory Performance vs. Pentium 4
- Conclusion

Why More on Memory Hierarchy?



Review: 6 Basic Cache Optimizations

Reducing hit time

1. Giving Reads Priority over Writes
 - E.g., Read complete before earlier writes in write buffer
2. Avoiding Address Translation during Cache Indexing

Reducing Miss Penalty

3. Multilevel Caches

Reducing Miss Rate

4. Larger Block size (Compulsory misses)
5. Larger Cache size (Capacity misses)
6. Higher Associativity (Conflict misses)

11 Advanced Cache Optimizations

Reducing hit time

1. Small and simple caches
2. Way prediction
3. Trace caches

Increasing cache bandwidth

4. Pipelined caches
5. Multibanked caches
6. Nonblocking caches

Reducing Miss Penalty

7. Critical word first
8. Merging write buffers

Reducing Miss Rate

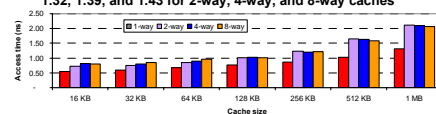
9. Compiler optimizations

Reducing miss penalty or miss rate via parallelism

10. Hardware prefetching
11. Compiler prefetching

1. Fast Hit times via Small and Simple Caches

- Index tag memory and then compare takes time
- ⇒ Small cache can help hit time since smaller memory takes less time to index
 - E.g., L1 caches same size for 3 generations of AMD microprocessors: K6, Athlon, and Opteron
 - Also L2 cache small enough to fit on chip with the processor avoids time penalty of going off chip
- Simple ⇒ direct mapping
 - Can overlap tag check with data transmission since no choice
- Access time estimate for 90 nm using CACTI model 4.0
 - Median ratios of access time relative to the direct-mapped caches are 1.32, 1.39, and 1.43 for 2-way, 4-way, and 8-way caches



2. Fast Hit times via Way Prediction

- How to combine fast hit time of Direct Mapped and have the lower conflict misses of 2-way SA cache?

- Way prediction: keep extra bits in cache to predict the “way”, or block within the set, of next cache access.
 - Multiplexor is set early to select desired block, only 1 tag comparison performed that clock cycle in parallel with reading the cache data
 - Miss \Rightarrow 1st check other blocks for matches in next clock cycle



- Accuracy \approx 85%
- Drawback: CPU pipeline is hard if hit takes 1 or 2 cycles
 - Used for instruction caches vs. data caches

3. Fast Hit times via Trace Cache (Pentium 4 only; and last time?)

- Find more instruction level parallelism? How avoid translation from x86 to microops?
- Trace cache in Pentium 4
 1. Dynamic traces of the executed instructions vs. static sequences of instructions as determined by layout in memory
 - » Built-in branch predictor
 2. Cache the micro-ops vs. x86 instructions
 - » Decode/translate from x86 to micro-ops on trace cache miss
- + 1. \Rightarrow better utilize long blocks (don't exit in middle of block, don't enter at label in middle of block)
- 1. \Rightarrow complicated address mapping since addresses no longer aligned to power-of-2 multiples of word size
- 1. \Rightarrow instructions may appear multiple times in multiple dynamic traces due to different branch outcomes

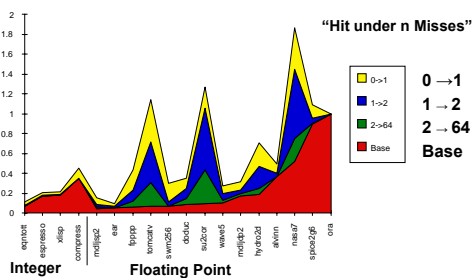
4. Increasing Cache Bandwidth by Pipelining

- Pipeline cache access to maintain bandwidth, but higher latency
- Instruction cache access pipeline stages:
 - 1: Pentium
 - 2: Pentium Pro through Pentium III
 - 4: Pentium 4
- \Rightarrow greater penalty on mispredicted branches
- \Rightarrow more clock cycles between the issue of the load and the use of the data

5. Increasing Cache Bandwidth: Non-Blocking Caches

- **Non-blocking cache** or **lockup-free cache** allow data cache to continue to supply cache hits during a miss
 - requires F/E bits on registers or out-of-order execution
 - requires multi-bank memories
- “**hit under miss**” reduces the effective miss penalty by working during miss vs. ignoring CPU requests
- “**hit under multiple miss**” or “**miss under miss**” may further lower the effective miss penalty by overlapping multiple misses
 - Significantly increases the complexity of the cache controller as there can be multiple outstanding memory accesses
 - Requires multiple memory banks (otherwise cannot support)
 - Pentium Pro allows 4 outstanding memory misses

Value of Hit Under Miss for SPEC (old data)



- FP programs on average: AMAT= 0.68 \rightarrow 0.52 \rightarrow 0.34 \rightarrow 0.26
- Int programs on average: AMAT= 0.24 \rightarrow 0.20 \rightarrow 0.19 \rightarrow 0.19
- 8 KB Data Cache, Direct Mapped, 32B block, 16 cycle miss, SPEC 92

6. Increasing Cache Bandwidth via Multiple Banks

- Rather than treat the cache as a single monolithic block, divide into independent banks that can support simultaneous accesses
 - E.g., T1 (“Niagara”) L2 has 4 banks
- Banking works best when accesses naturally spread themselves across banks \Rightarrow mapping of addresses to banks affects behavior of memory system
- Simple mapping that works well is “**sequential interleaving**”
 - Spread block addresses sequentially across banks
 - E.g, if there 4 banks, Bank 0 has all blocks whose address modulo 4 is 0; bank 1 has all blocks whose address modulo 4 is 1; ...

7. Reduce Miss Penalty: Early Restart and Critical Word First

- Don't wait for full block before restarting CPU
- **Early restart** – As soon as the requested word of the block arrives, send it to the CPU and let the CPU continue execution
 - Spatial locality \Rightarrow tend to want next sequential word, so not clear size of benefit of just early restart
- **Critical Word First** – Request the missed word first from memory and send it to the CPU as soon as it arrives; let the CPU continue execution while filling the rest of the words in the block
 - Long blocks more popular today \Rightarrow Critical Word 1st Widely used



8. Merging Write Buffer to Reduce Miss Penalty

- Write buffer to allow processor to continue while waiting to write to memory
- If buffer contains modified blocks, the addresses can be checked to see if address of new data matches the address of a valid write buffer entry
- If so, new data are combined with that entry
- Increases block size of write for write-through cache of writes to sequential words, bytes since multiword writes more efficient to memory
- The Sun T1 (Niagara) processor, among many others, uses write merging

9. Reducing Misses by Compiler Optimizations

- McFarling [1989] reduced caches misses by 75% on 8KB direct mapped cache, 4 byte blocks **in software**
- Instructions
 - Reorder procedures in memory so as to reduce conflict misses
 - Profiling to look at conflicts (using tools they developed)
- Data
 - **Merging Arrays**: Improve spatial locality by single array of compound elements vs. 2 arrays
 - **Loop Interchange**: Change nesting of loops to access data in order stored in memory
 - **Loop Fusion**: Combine 2 independent loops that have same looping and some variables overlap
 - **Blocking**: Improve temporal locality by accessing "blocks" of data repeatedly vs. going down whole columns or rows

Merging Arrays Example

```
/* Before: 2 sequential arrays */
int val[SIZE];
int key[SIZE];

/* After: 1 array of structures */
struct merge {
    int val;
    int key;
};
struct merge merged_array[SIZE];
```

Reducing conflicts between val & key;
improve spatial locality

Loop Interchange Example

```
/* Before */
for (k = 0; k < 100; k = k+1)
    for (j = 0; j < 100; j = j+1)
        for (i = 0; i < 5000; i = i+1)
            x[i][j] = 2 * x[i][j];

/* After */
for (k = 0; k < 100; k = k+1)
    for (i = 0; i < 5000; i = i+1)
        for (j = 0; j < 100; j = j+1)
            x[i][j] = 2 * x[i][j];
```

Sequential accesses instead of striding through memory every 100 words; improved spatial locality

Loop Fusion Example

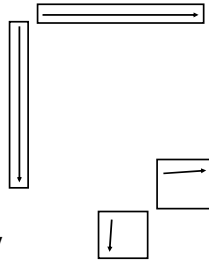
```
/* Before */
for (i = 0; i < N; i = i+1)
    for (j = 0; j < N; j = j+1)
        a[i][j] = 1/b[i][j] * c[i][j];
for (i = 0; i < N; i = i+1)
    for (j = 0; j < N; j = j+1)
        d[i][j] = a[i][j] + c[i][j];

/* After */
for (i = 0; i < N; i = i+1)
    for (j = 0; j < N; j = j+1)
    {
        a[i][j] = 1/b[i][j] * c[i][j];
        d[i][j] = a[i][j] + c[i][j];
    }
```

2 misses per access to a & c vs. one miss per access;
improve spatial locality

Blocking Example

```
/* Before */
for (i = 0; i < N; i = i+1)
  for (j = 0; j < N; j = j+1)
    {r = 0;
     for (k = 0; k < N; k = k+1) {
       r = r + y[i][k]*z[k][j];
       x[i][j] = r;
     }
    }
```



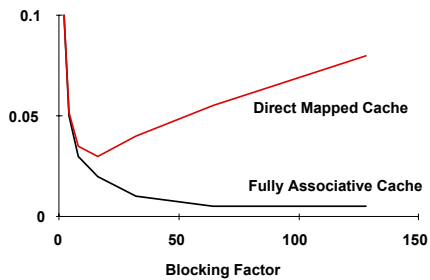
- **Two Inner Loops:**
 - Read all NxN elements of z[]
 - Read N elements of 1 row of y[] repeatedly
 - Write N elements of 1 row of x[]
- **Capacity Misses a function of N & Cache Size:**
 - $2N^3 + N^2 \Rightarrow$ (assuming no conflict; otherwise ...)
- **Idea: compute on BxB submatrix that fits**

Blocking Example

```
/* After */
for (jj = 0; jj < N; jj = jj+B)
  for (kk = 0; kk < N; kk = kk+B)
    for (i = 0; i < N; i = i+1)
      for (j = jj; j < min(jj+B-1,N); j = j+1)
        {r = 0;
         for (k = kk; k < min(kk+B-1,N); k = k+1) {
           r = r + y[i][k]*z[k][j];
           x[i][j] = x[i][j] + r;
         }
        }
```

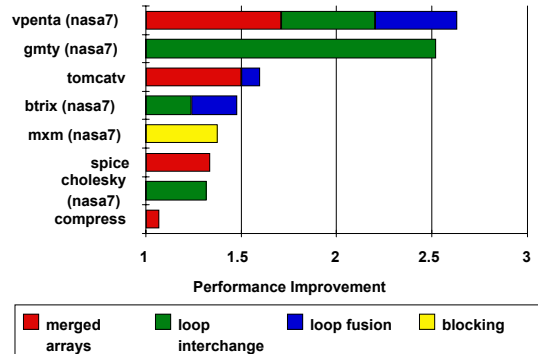
- **B called *Blocking Factor***
- **Capacity Misses from $2N^3 + N^2$ to $2N^3/B + N^2$**
- **Conflict Misses Too?**

Reducing Conflict Misses by Blocking



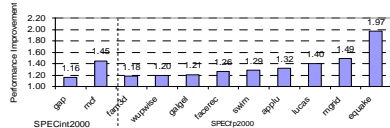
- **Conflict misses in caches not FA vs. Blocking size**
 - Lam et al [1991] a blocking factor of 24 had a fifth the misses vs. 48 despite both fit in cache

Summary of Compiler Optimizations to Reduce Cache Misses (by hand)



10. Reducing Misses by Hardware Prefetching of Instructions & Data

- **Prefetching relies on having extra memory bandwidth that can be used without penalty**
- **Instruction Prefetching**
 - Typically, CPU fetches 2 blocks on a miss: the requested block and the next consecutive block.
 - Requested block is placed in instruction cache when it returns, and prefetched block is placed into instruction stream buffer
- **Data Prefetching**
 - Pentium 4 can prefetch data into L2 cache from up to 8 streams from 8 different 4 KB pages
 - Prefetching invoked if 2 successive L2 cache misses to a page, if distance between those cache blocks is < 256 bytes



11. Reducing Misses by Software Prefetching Data

- **Data Prefetch**
 - Load data into register (HP PA-RISC loads)
 - Cache Prefetch: load into cache (MIPS IV, PowerPC, SPARC v. 9)
 - Special prefetching instructions cannot cause faults; a form of speculative execution
- **Issuing Prefetch Instructions takes time**
 - Is cost of prefetch issues < savings in reduced misses?
 - Higher superscalar reduces difficulty of issue bandwidth

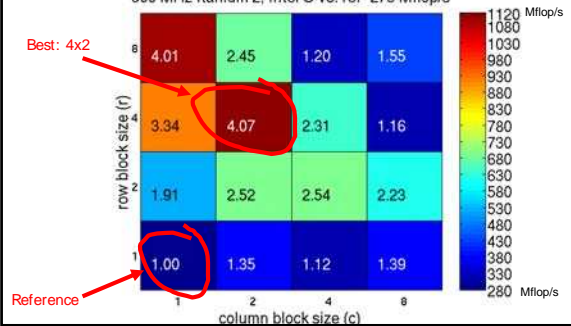
Compiler Optimization vs. Memory Hierarchy Search

- Compiler tries to figure out memory hierarchy optimizations
- New approach: “Auto-tuners” 1st run variations of program on computer to find best combinations of optimizations (blocking, padding, ...) and algorithms, then produce C code to be compiled for *that* computer
- “Auto-tuner” targeted to numerical method
 - E.g., PHIPAC (BLAS), Atlas (BLAS), Sparsity (Sparse linear algebra), Spiral (DSP), FFT-W

Sparse Matrix – Search for Blocking

for finite element problem [Im, Yelick, Vuduc, 2005]

900 MHz Itanium 2, Intel C v8: ref=275 Mflop/s



Best Sparse Blocking for 8 Computers

row block size (r)	1	2	4	8
8		Intel Pentium M		Sun Ultra 2, Sun Ultra 3, AMD Opteron
4	IBM Power 4, Intel/HP Itanium	Intel/HP Itanium 2	IBM Power 3	
2				
1				

All possible column block sizes selected for 8 computers; How could compiler know?

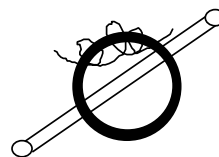
Technique	Hit Time	Bandwidth	Miss penalty	Miss rate	HW cost/complexity	Comment
Small and simple caches	+			-	0	Trivial; widely used
Way-predicting caches	+				1	Used in Pentium 4
Trace caches	+				3	Used in Pentium 4
Pipelined cache access	-	+			1	Widely used
Nonblocking caches		+	+		3	Widely used
Banked caches		+			1	Used in L2 of Opteron and Niagara
Critical word first and early restart			+		2	Widely used
Merging write buffer			+		1	Widely used with write through
Compiler techniques to reduce cache misses					0	Software is a challenge; some computers have compiler option
Hardware prefetching of instructions and data			+	+	2 instr., 3 data	Many prefetch instructions; AMD Opteron prefetches data
Compiler-controlled prefetching			+	+	3	Needs nonblocking cache; in many CPUs

Main Memory Background

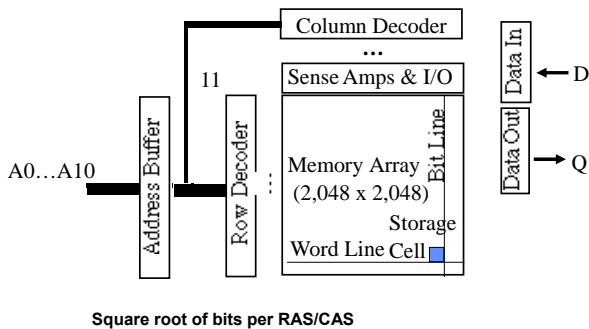
- Performance of Main Memory:
 - **Latency:** Cache Miss Penalty
 - » Access Time: time between request and word arrives
 - » Cycle Time: time between requests
 - **Bandwidth:** I/O & Large Block Miss Penalty (L2)
- Main Memory is **DRAM:** Dynamic Random Access Memory
 - Dynamic since needs to be **refreshed** periodically (8 ms, 1% time)
 - Addresses divided into 2 halves (Memory as a 2D matrix):
 - » RAS or Row Access Strobe
 - » CAS or Column Access Strobe
- Cache uses **SRAM:** Static Random Access Memory
 - No refresh (6 transistors/bit vs. 1 transistor)
 - Size: DRAM/SRAM - 4-8
 - Cost/Cycle time: SRAM/DRAM - 8-16

Main Memory Deep Background

- “Out-of-Core”, “In-Core,” “Core Dump”?
- “Core memory”?
- Non-volatile, magnetic
- Lost to 4 Kbit DRAM (today using 512Mbit DRAM)
- Access time 750 ns, cycle time 1500-3000 ns



DRAM logical organization (4 Mbit)



Quest for DRAM Performance

- 1. Fast Page mode**
 - Add timing signals that allow repeated accesses to row buffer without another row access time
 - Such a buffer comes naturally, as each array will buffer 1024 to 2048 bits for each access
- 2. Synchronous DRAM (SDRAM)**
 - Add a clock signal to DRAM interface, so that the repeated transfers would not bear overhead to synchronize with DRAM controller
- 3. Double Data Rate (DDR SDRAM)**
 - Transfer data on both the rising edge and falling edge of the DRAM clock signal \Rightarrow doubling the peak data rate
 - DDR2 lowers power by dropping the voltage from 2.5 to 1.8 volts + offers higher clock rates: up to 400 MHz
 - DDR3 drops to 1.5 volts + higher clock rates: up to 800 MHz

Improved Bandwidth, not Latency

DRAM name based on Peak Chip Transfers / Sec DIMM name based on Peak DIMM MBytes / Sec

Standard	Clock Rate (MHz)	M transfers / second	DRAM Name	Mbytes/s/ DIMM	DIMM Name
DDR	133	266	DDR266	2128	PC2100
DDR	150	300	DDR300	2400	PC2400
DDR	200	400	DDR400	3200	PC3200
DDR2	266	533	DDR2-533	4264	PC4300
DDR2	333	667	DDR2-667	5336	PC5300
DDR2	400	800	DDR2-800	6400	PC6400
DDR3	533	1066	DDR3-1066	8528	PC8500
DDR3	666	1333	DDR3-1333	10664	PC10700
DDR3	800	1600	DDR3-1600	12800	PC12800

Fastest for sale 4/06 (\$125/GB) | Fastest for sale 1/07 (\$200/GB)

Annotations: x 2 (from 400 to 800 MHz), x 8 (from 800 to 6400 Mbytes/s)

Need for Error Correction!

- **Motivation:**
 - Failures/time *proportional* to number of bits!
 - As DRAM cells shrink, more vulnerable
- Went through period in which failure rate was low enough without error correction that people didn't do correction
 - DRAM banks too large now
 - Servers always corrected memory systems
- **Basic idea: add redundancy through parity bits**
 - Common configuration: Random error correction
 - » SEC-DED (single error correct, double error detect)
 - » One example: 64 data bits + 8 parity bits (11% overhead)
 - Really want to handle failures of physical components as well
 - » Organization is multiple DRAMs/DIMM, multiple DIMMs
 - » Want to recover from failed DRAM and failed DIMM!
 - » "Chip kill" handle failures width of single DRAM chip

Outline

- 11 Advanced Cache Optimizations
- Memory Technology and DRAM optimizations
- Virtual Machines
- Xen VM: Design and Performance
- AMD Opteron Memory Hierarchy
- Opteron Memory Performance vs. Pentium 4
- Conclusion

Introduction to Virtual Machines

- VMs developed in late 1960s
 - Remained important in mainframe computing over the years
 - Largely ignored in single user computers of 1980s and 1990s
- Recently regained popularity due to
 - increasing importance of isolation and security in modern systems,
 - failures in security and reliability of standard operating systems,
 - sharing of a single computer among many unrelated users,
 - and the dramatic increases in raw speed of processors, which makes the overhead of VMs more acceptable

What is a Virtual Machine (VM)?

- Broadest definition includes all emulation methods that provide a standard software interface, such as the Java VM
- “(Operating) System Virtual Machines” provide a complete system level environment at binary ISA
 - Here assume ISAs always match the native hardware ISA
 - E.g., IBM VM/370, VMware ESX Server, and Xen
- Present illusion that VM users have entire computer to themselves, including a copy of OS
- Single computer runs multiple VMs, and can support a multiple, different OSes
 - On conventional platform, single OS “owns” all HW resources
 - With a VM, multiple OSes all share HW resources
- Underlying HW platform is called the **host**, and its resources are shared among the **guest** VMs

Virtual Machine Monitors (VMMs)

- **Virtual machine monitor (VMM) or hypervisor** is software that supports VMs
- VMM determines how to map virtual resources to physical resources
- Physical resource may be time-shared, partitioned, or emulated in software
- VMM is much smaller than a traditional OS;
 - isolation portion of a VMM is \approx 10,000 lines of code

VMM Overhead?

- Depends on the workload
- **User-level processor-bound programs** (e.g., SPEC) have zero-virtualization overhead
 - Runs at native speeds since OS rarely invoked
- **I/O-intensive workloads** \Rightarrow OS-intensive \Rightarrow execute many system calls and privileged instructions
 - \Rightarrow can result in high virtualization overhead
 - For System VMs, goal of architecture and VMM is to run almost all instructions directly on native hardware
- If I/O-intensive workload is also **I/O-bound**
 - \Rightarrow low processor utilization since waiting for I/O
 - \Rightarrow processor virtualization can be hidden
 - \Rightarrow low virtualization overhead

Requirements of a Virtual Machine Monitor

- **A VM Monitor**
 - Presents a SW interface to guest software,
 - Isolates state of guests from each other, and
 - Protects itself from guest software (including guest OSes)
- **Guest software should behave on a VM exactly as if running on the native HW**
 - Except for performance-related behavior or limitations of fixed resources shared by multiple VMs
- **Guest software should not be able to change allocation of real system resources directly**
- **Hence, VMM must control \approx everything even though guest VM and OS currently running is temporarily using them**
 - Access to privileged state, Address translation, I/O, Exceptions and Interrupts, ...

Requirements of a Virtual Machine Monitor

- **VMM must be at higher privilege level than guest VM, which generally run in user mode**
 - \Rightarrow Execution of privileged instructions handled by VMM
- **E.g., Timer interrupt: VMM suspends currently running guest VM, saves its state, handles interrupt, determine which guest VM to run next, and then load its state**
 - Guest VMs that rely on timer interrupt provided with virtual timer and an emulated timer interrupt by VMM
- **Requirements of system virtual machines are \approx same as paged-virtual memory:**
 1. At least 2 processor modes, system and user
 2. Privileged subset of instructions available only in system mode, trap if executed in user mode
 - All system resources controllable only via these instructions

ISA Support for Virtual Machines

- **If plan for VM during design of ISA, easy to reduce instructions executed by VMM, speed to emulate**
 - ISA is **virtualizable** if can execute VM directly on real machine while letting VMM retain ultimate control of CPU: “**direct execution**”
 - Since VMs have been considered for desktop/PC server apps only recently, most ISAs were created ignoring virtualization, including 80x86 and most RISC architectures
- **VMM must ensure that guest system only interacts with virtual resources \Rightarrow conventional guest OS runs as user mode program on top of VMM**
 - If guest OS accesses or modifies information related to HW resources via a privileged instruction—e.g., reading or writing the page table pointer—it will trap to VMM
- **If not, VMM must intercept instruction and support a virtual version of sensitive information as guest OS expects**

Impact of VMs on Virtual Memory

- Virtualization of virtual memory if each guest OS in every VM manages its own set of page tables?
- VMM separates **real** and **physical memory**
 - Makes real memory a separate, intermediate level between virtual memory and physical memory
 - Some use the terms **virtual memory**, **physical memory**, and **machine memory** to name the 3 levels
 - Guest OS maps virtual memory to real memory via its page tables, and VMM page tables map real memory to physical memory
- VMM maintains a **shadow page table** that maps directly from the guest virtual address space to the physical address space of HW
 - Rather than pay extra level of indirection on every memory access
 - VMM must trap any attempt by guest OS to change its page table or to access the page table pointer

ISA Support for VMs & Virtual Memory

- IBM 370 architecture added additional level of indirection that is managed by the VMM
 - Guest OS keeps its page tables as before, so the shadow pages are unnecessary
 - (AMD Pacifica proposes same improvement for 80x86)
- To virtualize software TLB, VMM manages the real TLB and has a copy of the contents of the TLB of each guest VM
 - Any instruction that accesses the TLB must trap
 - TLBs with Process ID tags support a mix of entries from different VMs and the VMM, thereby avoiding flushing of the TLB on a VM switch

Impact of I/O on Virtual Memory

- I/O most difficult part of virtualization
 - Increasing number of I/O devices attached to the computer
 - Increasing diversity of I/O device types
 - Sharing of a real device among multiple VMs
 - Supporting many device drivers that are required, especially if different guest OSes are supported on same VM system
- Give each VM generic versions of each type of I/O device driver, and let VMM to handle real I/O
- Method for mapping virtual to physical I/O device depends on the type of device:
 - Disks partitioned by VMM to create virtual disks for guest VMs
 - Network interfaces shared between VMs in short time slices, and VMM tracks messages for virtual network addresses to ensure that guest VMs only receive their messages

Example: Xen VM

- Xen: Open-source System VMM for 80x86 ISA
 - Project started at University of Cambridge, GNU license model
- Original vision of VM is running unmodified OS
 - Significant wasted effort just to keep guest OS happy
- “paravirtualization” – small modifications to guest OS to simplify virtualization

Three examples of paravirtualization in Xen:

1. To avoid flushing TLB when invoke VMM, Xen mapped into upper 64 MB of address space of each VM
2. Guest OS allowed to allocate pages, just check that didn't violate protection restrictions
3. To protect the guest OS from user programs in VM, Xen takes advantage of 4 protection levels available in 80x86
 - Most OSes for 80x86 keep everything at privilege levels 0 or at 3.
 - Xen VMM runs at the highest privilege level (0)
 - Guest OS runs at the next level (1)
 - Applications run at the lowest privilege level (3)

Xen changes for paravirtualization

- Port of Linux to Xen changed ≈ 3000 lines, or ≈ 1% of 80x86-specific code
 - Does not affect application-binary interfaces of guest OS
- OSes supported in Xen 2.0

OS	Runs as host OS	Runs as guest OS
Linux 2.4	Yes	Yes
Linux 2.6	Yes	Yes
NetBSD 2.0	No	Yes
NetBSD 3.0	Yes	Yes
Plan 9	No	Yes
FreeBSD 5	No	Yes

<http://wiki.xensource.com/xenwiki/OSCompatibility>

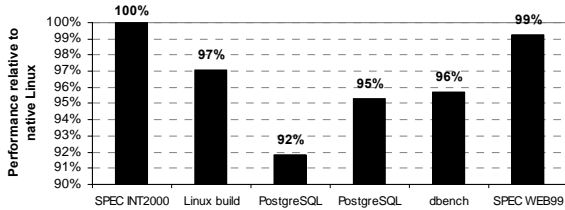
- More OSes in Xen 3.0

Xen and I/O

- To simplify I/O, privileged VMs assigned to each hardware I/O device: “**driver domains**”
 - Xen Jargon: “domains” = Virtual Machines
- Driver domains run physical device drivers, although interrupts still handled by VMM before being sent to appropriate driver domain
- Regular VMs (“**guest domains**”) run simple virtual device drivers that communicate with physical devices drivers in driver domains over a channel to access physical I/O hardware
- Data sent between guest and driver domains by page remapping

Xen Performance

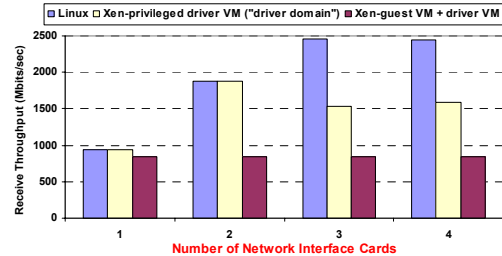
- Performance relative to native Linux for Xen for 6 benchmarks from Xen developers



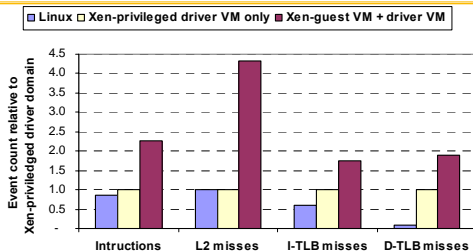
- User-level processor-bound programs? I/O-intensive workloads? I/O-Bound I/O-Intensive?
- Detailed performance analysis: see book

Xen Performance, Part II

- Subsequent study noticed Xen experiments based on 1 Ethernet network interfaces card (NIC), and single NIC was a performance bottleneck



Xen Performance, Part III



1. > 2X instructions for guest VM + driver VM
2. > 4X L2 cache misses
3. 12X – 24X Data TLB misses

Xen Performance, Part IV

1. > 2X instructions: page remapping and page transfer between driver and guest VMs and due to communication between the 2 VMs over a channel
2. 4X L2 cache misses: Linux uses zero-copy network interface that depends on ability of NIC to do DMA from different locations in memory
 - Since Xen does not support “gather DMA” in its virtual network interface, it can't do true zero-copy in the guest VM
3. 12X – 24X Data TLB misses: 2 Linux optimizations
 - Superpages for part of Linux kernel space, and 4MB pages lowers TLB misses versus using 1024 4 KB pages. Not in Xen
 - PTEs marked global are not flushed on a context switch, and Linux uses them for its kernel space. Not in Xen

Future Xen may address 2. and 3., but 1. inherent?

Protection and Instruction Set Architecture

- Example Problem: 80x86 POPF instruction loads flag registers from top of stack in memory
 - One such flag is Interrupt Enable (IE)
 - In system mode, POPF changes IE
 - In user mode, POPF simply changes all flags *except* IE
 - Problem: guest OS runs in user mode inside a VM, so it expects to see changed a IE, but it won't
- Historically, IBM mainframe HW and VMM took 3 steps:
 1. Reduce cost of processor virtualization
 - » Intel/AMD proposed ISA changes to reduce this cost
 2. Reduce interrupt overhead cost due to virtualization
 3. Reduce interrupt cost by steering interrupts to proper VM directly without invoking VMM

2. and 3. not yet addressed by Intel/AMD; in the future?

80x86 VM Challenges

18 instructions cause problems for virtualization:

1. Read control registers in user model that reveal that the guest operating system is running in a virtual machine (such as POPF), and
2. Check protection as required by the segmented architecture but assume that the operating system is running at the highest privilege level

Virtual memory: 80x86 TLBs do not support process ID tags ⇒ more expensive for VMM and guest OSES to share the TLB

- each address space change typically requires a TLB flush

Intel/AMD address 80x86 VM Challenges

- Goal is direct execution of VMs on 80x86
- Intel's VT-x
 - A new execution mode for running VMs
 - An architected definition of the VM state
 - Instructions to swap VMs rapidly
 - Large set of parameters to select the circumstances where a VMM must be invoked
 - VT-x adds 11 new instructions to 80x86
- Xen 3.0 plan proposes to use VT-x to run Windows on Xen
- AMD's Pacifica makes similar proposals
 - Plus indirection level in page table like IBM VM 370
- Ironic adding a new mode
 - If OS start using mode in kernel, new mode would cause performance problems for VMM since ≈ 100 times too slow

Outline

- 11 Advanced Cache Optimizations
- Memory Technology and DRAM optimizations
- Virtual Machines
- Xen VM: Design and Performance
- AMD Opteron Memory Hierarchy
- Opteron Memory Performance vs. Pentium 4
- Conclusion

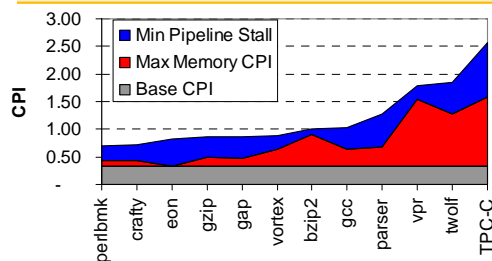
AMD Opteron Memory Hierarchy

- 12-stage integer pipeline yields a maximum clock rate of 2.8 GHz and fastest memory PC3200 DDR SDRAM
- 48-bit virtual and 40-bit physical addresses
- I and D cache: 64 KB, 2-way set associative, 64-B block, LRU
- L2 cache: 1 MB, 16-way, 64-B block, pseudo LRU
- Data and L2 caches use write back, write allocate
- L1 caches are virtually indexed and physically tagged
- L1 I TLB and L1 D TLB: fully associative, 40 entries
 - 32 entries for 4 KB pages and 8 for 2 MB or 4 MB pages
- L2 I TLB and L1 D TLB: 4-way, 512 entries of 4 KB pages
- Memory controller allows up to 10 cache misses
 - 8 from D cache and 2 from I cache

Opteron Memory Hierarchy Performance

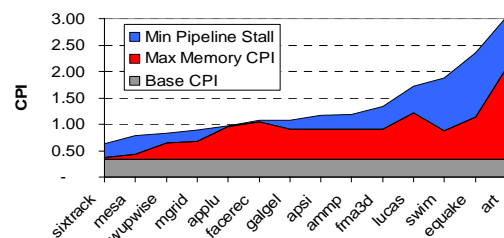
- For SPEC2000
 - I cache misses per instruction is 0.01% to 0.09%
 - D cache misses per instruction are 1.34% to 1.43%
 - L2 cache misses per instruction are 0.23% to 0.36%
- Commercial benchmark (“TPC-C-like”)
 - I cache misses per instruction is 1.83% (100X!)
 - D cache misses per instruction are 1.39% (\approx same)
 - L2 cache misses per instruction are 0.62% (2X to 3X)
- How compare to ideal CPI of 0.33?

CPI breakdown for Integer Programs



- CPI above base attributable to memory $\approx 50\%$
- L2 cache misses $\approx 25\%$ overall (50% memory CPI)
 - Assumes misses are *not* overlapped with the execution pipeline or with each other, so the pipeline stall portion is a lower bound

CPI breakdown for FP Programs



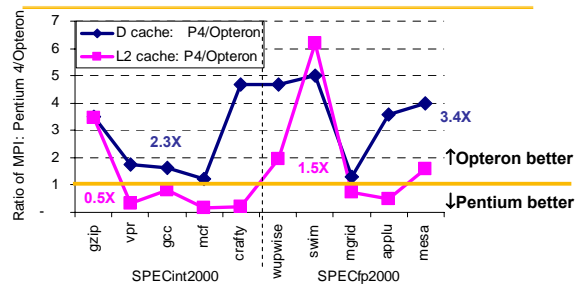
- CPI above base attributable to memory $\approx 60\%$
- L2 cache misses $\approx 40\%$ overall (70% memory CPI)
 - Assumes misses are *not* overlapped with the execution pipeline or with each other, so the pipeline stall portion is a lower bound

Pentium 4 vs. Opteron Memory Hierarchy

CPU	Pentium 4 (3.2 GHz*)	Opteron (2.8 GHz*)
Instruction Cache	Trace Cache (8K micro-ops)	2-way associative, 64 KB, 64B block
Data Cache	8-way associative, 16 KB, 64B block, inclusive in L2	2-way associative, 64 KB, 64B block, exclusive to L2
L2 cache	8-way associative, 2 MB, 128B block	16-way associative, 1 MB, 64B block
Prefetch	8 streams to L2	1 stream to L2
Memory	200 MHz x 64 bits	200 MHz x 128 bits

*Clock rate for this comparison in 2005; faster versions existed

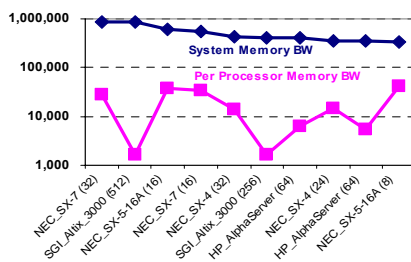
Misses Per Instruction: Pentium 4 vs. Opteron



- D cache miss: P4 is 2.3X to 3.4X vs. Opteron
- L2 cache miss: P4 is 0.5X to 1.5X vs. Opteron
- Note: Same ISA, but not same instruction count

Fallacies and Pitfalls

- Not delivering high memory bandwidth in a cache-based system
 - 10 Fastest computers at Stream benchmark [McCalpin 2005]
 - Only 4/10 computers rely on data caches, and their memory BW per processor is 7X to 25X slower than NEC SX7



And in Conclusion [1/2] ...

- Memory wall inspires optimizations since so much performance lost there
 - Reducing hit time: Small and simple caches, Way prediction, Trace caches
 - Increasing cache bandwidth: Pipelined caches, Multibanked caches, Nonblocking caches
 - Reducing Miss Penalty: Critical word first, Merging write buffers
 - Reducing Miss Rate: Compiler optimizations
 - Reducing miss penalty or miss rate via parallelism: Hardware prefetching, Compiler prefetching
- “Auto-tuners” search replacing static compilation to explore optimization space?
- DRAM – Continuing Bandwidth innovations: Fast page mode, Synchronous, Double Data Rate

And in Conclusion [2/2] ...

- VM Monitor presents a SW interface to guest software, isolates state of guests, and protects itself from guest software (including guest OSES)
- Virtual Machine Revival
 - Overcome security flaws of large OSES
 - Manage Software, Manage Hardware
 - Processor performance no longer highest priority
- Virtualization challenges for processor, virtual memory, and I/O
 - Paravirtualization to cope with those difficulties
- Xen as example VMM using paravirtualization
 - 2005 performance on non-I/O bound, I/O intensive apps: 80% of native Linux without driver VM, 34% with driver VM
- Opteron memory hierarchy still critical to performance

Reading

- This lecture:
 - chapter 5: *Memory Hierarchy Design*
- Next lecture:
 - chapter 6: *Storage Systems*

Lecture 11 – Storage

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

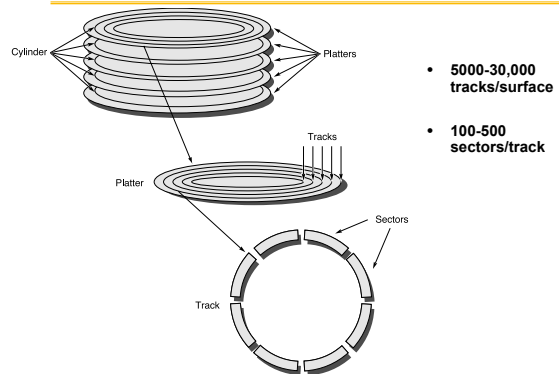
Case for Storage

- Shift in focus from computation to communication and storage of information
 - E.g., Cray Research/Thinking Machines vs. Google/Yahoo
 - “The Computing Revolution” (1960s to 1980s)
 - ⇒ “The Information Age” (1990 to today)
- Storage emphasizes reliability and scalability as well as cost-performance
- What is “Software king” that determines which HW actually features used?
 - Operating System for storage
 - Compiler for processor
- Also has own performance theory—queuing theory—balances throughput vs. response time

Outline

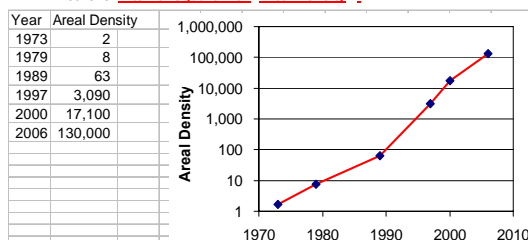
- Magnetic Disks
- RAID
- Advanced Dependability/Reliability/Availability
- I/O Benchmarks, Performance and Dependability
- Intro to Queuing Theory
- The End

Disk Organization



Disk Figure of Merit: Areal Density

- Bits recorded along a track
 - Metric is **Bits Per Inch (BPI)**
- Number of tracks per surface
 - Metric is **Tracks Per Inch (TPI)**
- Disk Designs Brag about **bit density per unit area**
 - Metric is **Bits Per Square Inch: Areal Density = BPI x TPI**



Historical Perspective

- 1956 IBM Ramac — early 1970s Winchester
 - Developed for mainframe computers, proprietary interfaces
 - Steady shrink in form factor: 27 in. to 14 in.
- Form factor and capacity drives market more than performance
- 1970s developments
 - 5.25 inch floppy disk formfactor (microcode into mainframe)
 - Emergence of industry standard disk interfaces
- Early 1980s: PCs and first generation workstations
- Mid 1980s: Client/server computing
 - Centralized storage on file server
 - » accelerates disk downsizing: 8 inch to 5.25
 - Mass market disk drives become a reality
 - » industry standards: SCSI, IPI, IDE
 - » 5.25 inch to 3.5 inch drives for PCs, End of proprietary interfaces
- 1990s: Laptops => 2.5 inch drives
- 2000s: What new devices leading to new drives?

Future Disk Size and Performance

- Continued advance in capacity (60%/yr) and bandwidth (40%/yr)
- Slow improvement in seek, rotation (8%/yr)

Time to read whole disk

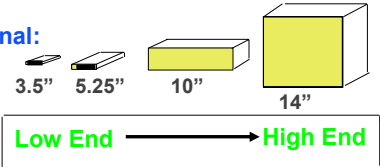
Year	Sequentially	Randomly (1 sector/seek)
1990	4 minutes	6 hours
2000	12 minutes	1 week(!)
2006	56 minutes	3 weeks (SCSI)
2006	171 minutes	7 weeks (SATA)

Use Arrays of Small Disks?

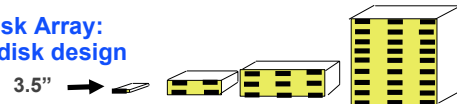
Katz and Patterson asked in 1987:

Can smaller disks be used to close gap in performance between disks and CPUs?

Conventional:
4 disk designs



Disk Array:
1 disk design



Replace Small Number of Large Disks with Large Number of Small Disks! (1988 Disks)

	IBM 3390K	IBM 3.5" 0061	x70
Capacity	20 GBytes	320 MBytes	23 GBytes
Volume	97 cu. ft.	0.1 cu. ft.	11 cu. ft. 9X
Power	3 KW	11 W	1 KW 3X
Data Rate	15 MB/s	1.5 MB/s	120 MB/s 8X
I/O Rate	600 I/Os/s	55 I/Os/s	3900 I/Os/s 6X
MTTF	250 KHrs	50 KHrs	??? Hrs
Cost	\$250K	\$2K	\$150K

Disk Arrays have potential for large data and I/O rates, high MB per cu. ft., high MB per KW, but what about reliability?

Array Reliability

Reliability of N disks = Reliability of 1 Disk ÷ N

50,000 Hours ÷ 70 disks = 700 hours

Disk system MTTF: Drops from 6 years to 1 month!

Arrays (without redundancy) too unreliable to be useful!

Hot spares support reconstruction in parallel with access: very high media availability can be achieved

Redundant Arrays of (Inexpensive) Disks

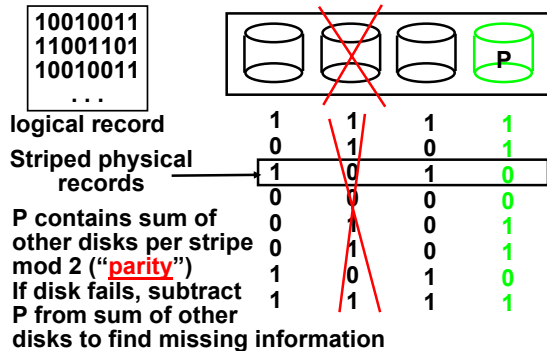
- Files are "striped" across multiple disks
- Redundancy yields high data availability
 - **Availability:** service still provided to user, even if some components failed
- Disks will still fail
- Contents reconstructed from data redundantly stored in the array
 - ⇒ Capacity penalty to store redundant info
 - ⇒ Bandwidth penalty to update redundant info

Redundant Arrays of Inexpensive Disks RAID 1: Disk Mirroring/Shadowing



- Each disk is fully duplicated onto its "**mirror**"
- Very high availability can be achieved
- Bandwidth sacrifice on write:
 - Logical write = two physical writes
- Reads may be optimized
- Most expensive solution: 100% capacity overhead
- (RAID 2 not interesting, so skip)

Redundant Array of Inexpensive Disks RAID 3: Parity Disk



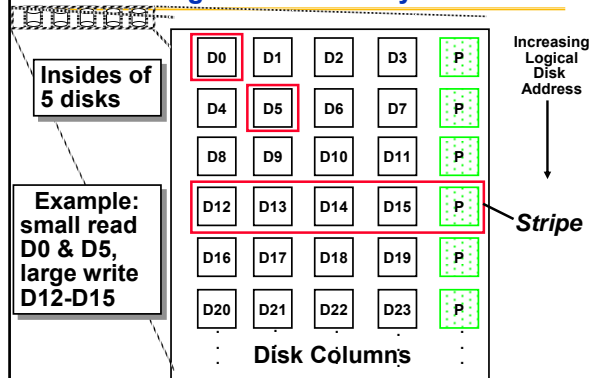
RAID 3

- Sum computed across recovery group to protect against hard disk failures, stored in P disk
- Logically, a single high capacity, high transfer rate disk: good for large transfers
- Wider arrays reduce capacity costs, but decreases availability
- 33% capacity cost for parity if 3 data disks and 1 parity disk

Inspiration for RAID 4

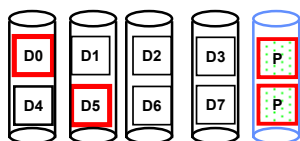
- RAID 3 relies on parity disk to discover errors on Read
- But every sector has an error detection field
- To catch errors on read, rely on error detection field vs. the parity disk
- Allows independent reads to different disks simultaneously

Redundant Arrays of Inexpensive Disks RAID 4: High I/O Rate Parity

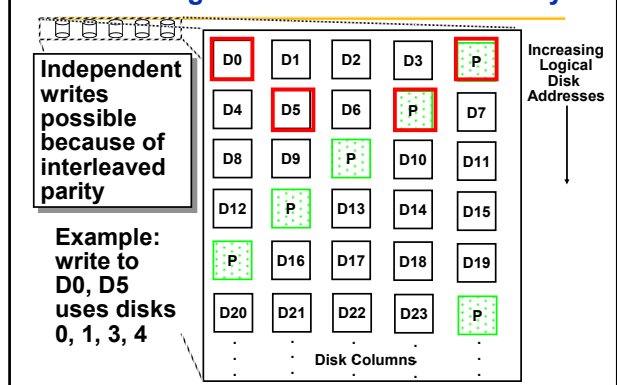


Inspiration for RAID 5

- RAID 4 works well for small reads
- Small writes (write to one disk):
 - Option 1: read other data disks, create new sum and write to Parity Disk
 - Option 2: since P has old sum, compare old data to new data, add the difference to P
- Small writes are limited by Parity Disk: Write to D0, D5 both also write to P disk



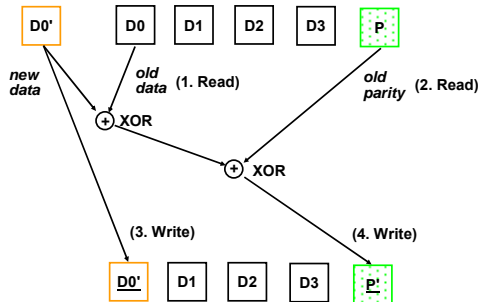
Redundant Arrays of Inexpensive Disks RAID 5: High I/O Rate Interleaved Parity



Problems of Disk Arrays: Small Writes

RAID-5: Small Write Algorithm

1 Logical Write = 2 Physical Reads + 2 Physical Writes



RAID 6: Recovering from 2 failures

Why > 1 failure recovery?

- operator accidentally replaces the wrong disk during a failure
- since disk bandwidth is growing more slowly than disk capacity, the MTT Repair a disk in a RAID system is increasing
 - ⇒ increases the chances of a 2nd failure during repair since takes longer
- reading much more data during reconstruction meant increasing the chance of an uncorrectable media failure, which would result in data loss

RAID 6: Recovering from 2 failures

- Network Appliance's *row-diagonal parity* or *RAID-DP*
- Like the standard RAID schemes, it uses redundant space based on parity calculation per stripe
- Since it is protecting against a double failure, it adds two check blocks per stripe.
 - If $p+1$ disks total, $p-1$ disks have data; assume $p=5$
- Row parity disk is just like in RAID 4
 - Even parity across the other 4 data blocks in its stripe
- Each block of the diagonal parity disk contains the even parity of the blocks in the same diagonal

Example $p = 5$

- Row diagonal parity starts by recovering one of the 4 blocks on the failed disk using diagonal parity
 - Since each diagonal misses one disk, and all diagonals miss a different disk, 2 diagonals are only missing 1 block
- Once the data for those blocks is recovered, then the standard RAID recovery scheme can be used to recover two more blocks in the standard RAID 4 stripes
- Process continues until two failed disks are restored

Data Disk 0	Data Disk 1	Data Disk 2	Data Disk 3	Row Parity	Diagonal Parity
0	1	2	3	0	0
1	2	3	4	0	1
2	3	4	0	1	2
3	4	0	1	2	3
4	0	1	2	3	4
0	1	2	3	4	0

Berkeley History: RAID-I

• RAID-I (1989)

Consisted of a Sun 4/280 workstation with 128 MB of DRAM, four dual-string SCSI controllers, 28 5.25-inch SCSI disks and specialized disk striping software

- Today RAID is \$24 billion dollar industry, 80% nonPC disks sold in RAIDs



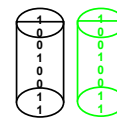
Summary: RAID Techniques: Goal was performance, popularity due to reliability of storage

• Disk Mirroring, Shadowing (RAID 1)

Each disk is fully duplicated onto its "shadow"

Logical write = two physical writes

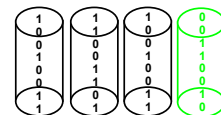
100% capacity overhead



• Parity Data Bandwidth Array (RAID 3)

Parity computed horizontally

Logically a single high data bw disk

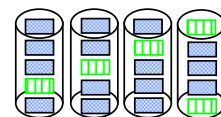


• High I/O Rate Parity Array (RAID 5)

Interleaved parity blocks

Independent reads and writes

Logical write = 2 reads + 2 writes



Definitions

- **Examples on why precise definitions so important for reliability**
- Is a programming mistake a fault, error, or failure?
 - Are we talking about the time it was designed or the time the program is run?
 - If the running program doesn't exercise the mistake, is it still a fault/error/failure?
- If an alpha particle hits a DRAM memory cell, is it a fault/error/failure if it doesn't change the value?
 - Is it a fault/error/failure if the memory doesn't access the changed bit?
 - Did a fault/error/failure still occur if the memory had error correction and delivered the corrected value to the CPU?

International Federation for Information Processing (IFIP) Standard terminology

- Computer system **dependability**: quality of delivered service such that reliance can be placed on service
- **Service** is observed **actual behavior** as perceived by other system(s) interacting with this system's users
- Each module has ideal **specified behavior**, where **service specification** is agreed description of expected behavior
- A system **failure** occurs when the actual behavior deviates from the specified behavior
- Failure occurred because an **error**, a defect in module
- The cause of an error is a **fault**
- When a fault occurs it creates a **latent error**, which becomes **effective** when it is activated
- When error actually affects the delivered service, a failure occurs (time from error to failure is **error latency**)

Fault v. (Latent) Error v. Failure

- An **error** is manifestation *in the system* of a **fault**, a **failure** is manifestation *on the service* of an **error**
- If an alpha particle hits a DRAM memory cell, is it a fault/error/failure if it doesn't change the value?
 - Is it a fault/error/failure if the memory doesn't access the changed bit?
 - Did a fault/error/failure still occur if the memory had error correction and delivered the corrected value to the CPU?
- An alpha particle hitting a DRAM can be a **fault**
- If it changes the memory, it creates an **error**
- Error remains **latent** until effected memory word is read
- If the effected word error affects the delivered service, a **failure** occurs

Fault Categories

1. **Hardware faults**: Devices that fail, such alpha particle hitting a memory cell
2. **Design faults**: Faults in software (usually) and hardware design (occasionally)
3. **Operation faults**: Mistakes by operations and maintenance personnel
4. **Environmental faults**: Fire, flood, earthquake, power failure, and sabotage

Also by duration:

1. **Transient faults** exist for limited time and not recurring
2. **Intermittent faults** cause a system to oscillate between faulty and fault-free operation
3. **Permanent faults** do not correct themselves over time

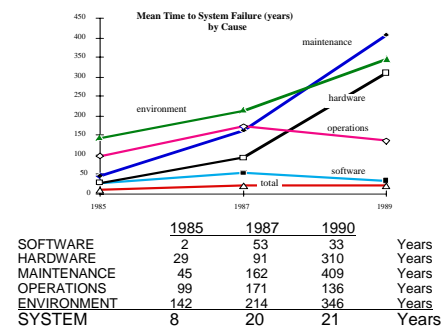
Fault Tolerance vs Disaster Tolerance

- Fault-Tolerance (or more properly, Error-Tolerance): **mask local faults (prevent errors from becoming failures)**
 - RAID disks
 - Uninterruptible Power Supplies
 - Cluster Failover
- Disaster Tolerance: **masks site errors (prevent site errors from causing service failures)**
 - Protects against fire, flood, sabotage,...
 - Redundant system and service at remote site.
 - Use design diversity

From Jim Gray's "Talk at UC Berkeley on Fault Tolerance" 11/9/00

Case Studies - Tandem Trends

Reported MTTF by Component



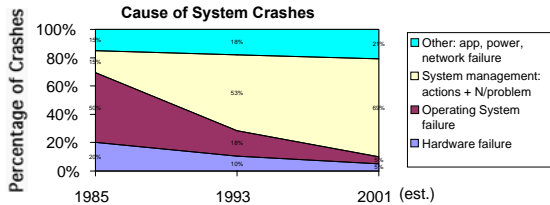
Problem: Systematic Under-reporting

From Jim Gray's "Talk at UC Berkeley on Fault Tolerance" 11/9/00

Is Maintenance the Key?

• Rule of Thumb: Maintenance 10X HW

– so over 5 year product life, ~ 95% of cost is maintenance



- **VAX crashes '85, '93 [Murp95]; extrap. to '01**
- **Sys. Man.:** N crashes/problem, SysAdmin action
 - Actions: set params bad, bad config, bad app install
- **HW/OS 70% in '85 to 28% in '93. In '01, 10%?**

HW Failures in Real Systems: Tertiary Disks

A cluster of 20 PCs in seven 7-foot high, 19-inch wide racks with 368 8.4 GB, 7200 RPM, 3.5-inch IBM disks. The PCs are P6-200MHz with 96 MB of DRAM each. They run FreeBSD 3.0 and the hosts are connected via switched 100 Mbit/second Ethernet

Component	Total in System	Total Failed	% Failed
SCSI Controller	44	1	2.3%
SCSI Cable	39	1	2.6%
SCSI Disk	368	7	1.9%
IDE Disk	24	6	25.0%
Disk Enclosure - Backplane	46	13	28.3%
Disk Enclosure - Power Supply	92	3	3.3%
Ethernet Controller	20	1	5.0%
Ethernet Switch	2	1	50.0%
Ethernet Cable	42	1	2.3%
CPU/Motherboard	20	0	0%

Does Hardware Fail Fast? 4 of 384 Disks that failed in Tertiary Disk

Messages in system log for failed disk	No. log msgs	Duration (hours)
Hardware Failure (Peripheral device write fault [for] Field Replaceable Unit)	1763	186
Not Ready (Diagnostic failure: ASCQ = Component ID [of] Field Replaceable Unit)	1460	90
Recovered Error (Failure Prediction Threshold Exceeded [for] Field Replaceable Unit)	1313	5
Recovered Error (Failure Prediction Threshold Exceeded [for] Field Replaceable Unit)	431	17

High Availability System Classes Goal: Build Class 6 Systems

System Type	Unavailable (min/year)	Availability	Availability Class
Unmanaged	50,000	90.0%	1
Managed	5,000	99.0%	2
Well Managed	500	99.9%	3
Fault Tolerant	50	99.99%	4
High-Availability	5	99.999%	5
Very-High-Availability	.5	99.9999%	6
Ultra-Availability	.05	99.99999%	7

UnAvailability = MTTR/MTBF
can cut it in 1/2 by cutting MTTR *or* MTBF

From Jim Gray's "Talk at UC Berkeley on Fault Tolerance" 11/9/00

How Realistic is "5 Nines"?

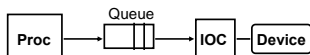
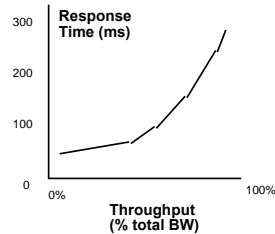
- HP claims HP-9000 server HW and HP-UX OS can deliver 99.999% availability guarantee "in certain pre-defined, pre-tested customer environments"
 - Application faults?
 - Operator faults?
 - Environmental faults?
- Collocation sites (lots of computers in 1 building on Internet) have
 - 1 network outage per year (~1 day)
 - 1 power failure per year (~1 day)
- Microsoft Network unavailable recently for a day due to problem in Domain Name Server: if only outage per year, 99.7% or 2 Nines

Outline

- Magnetic Disks
- RAID
- Advanced Dependability/Reliability/Availability
- I/O Benchmarks, Performance and Dependability
- Intro to Queuing Theory
- The End

I/O Performance

Metrics: Response Time vs. Throughput



Response time = Queue + Device Service time

I/O Benchmarks

- For better or worse, benchmarks shape a field
 - Processor benchmarks classically aimed at response time for fixed sized problem
 - I/O benchmarks typically measure throughput, possibly with upper limit on response times (or 90% of response times)
- Transaction Processing (TP) (or On-line TP=OLTP)
 - If bank computer fails when customer withdraw money, TP system guarantees account debited if customer gets \$ & account unchanged if no \$
 - Airline reservation systems & banks use TP
- Atomic transactions makes this work
- Classic metric is Transactions Per Second (TPS)

I/O Benchmarks: Transaction Processing

- Early 1980s great interest in OLTP
 - Expecting demand for high TPS (e.g., ATM machines, credit cards)
 - Tandem's success implied medium range OLTP expands
 - Each vendor picked own conditions for TPS claims, report only CPU times with widely different I/O
 - Conflicting claims led to disbelief of all benchmarks ⇒ chaos
- 1984 Jim Gray (Tandem) distributed paper to Tandem + 19 in other companies propose standard benchmark
- Published "A measure of transaction processing power," Datamation, 1985 by Anonymous et. al
 - To indicate that this was effort of large group
 - To avoid delays of legal department of each author's firm
 - Still get mail at Tandem to author "Anonymous"
- Led to Transaction Processing Council in 1988
 - www.tpc.org

I/O Benchmarks: TP1 by Anon et. al

- DebitCredit Scalability: size of account, branch, teller, history function of throughput

TPS	Number of ATMs	Account-file size
10	1,000	0.1 GB
100	10,000	1.0 GB
1,000	100,000	10.0 GB
10,000	1,000,000	100.0 GB

 - Each input TPS ⇒ 100,000 account records, 10 branches, 100 ATMs
 - Accounts must grow since a person is not likely to use the bank more frequently just because the bank has a faster computer!
- Response time: 95% transactions take ≤ 1 second
- Report price (initial purchase price + 5 year maintenance = cost of ownership)
- Hire auditor to certify results

Unusual Characteristics of TPC

- Price is included in the benchmarks
 - cost of HW, SW, and 5-year maintenance agreements included ⇒ price-performance as well as performance
- The data set generally must scale in size as the throughput increases
 - trying to model real systems, demand on system and size of the data stored in it increase together
- The benchmark results are audited
 - Must be approved by certified TPC auditor, who enforces TPC rules ⇒ only fair results are submitted
- Throughput is the performance metric but response times are limited
 - eg, TPC-C: 90% transaction response times < 5 seconds
- An independent organization maintains the benchmarks
 - COO ballots on changes, meetings, to settle disputes...

TPC Benchmark History/Status

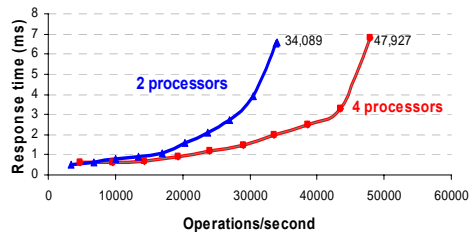
Benchmark	Data Size (GB)	Performance Metric	1st Results
A: Debit Credit (retired)	0.1 to 10	transactions/s	Jul-90
B: Batch Debit Credit (retired)	0.1 to 10	transactions/s	Jul-91
C: Complex Query OLTP	100 to 3000 (min. 07 * tpm)	new order trans/min (tpm)	Sep-92
D: Decision Support (retired)	100, 300, 1000	queries/hour	Dec-95
H: Ad hoc decision support	100, 300, 1000	queries/hour	Oct-99
R: Business reporting decision support (retired)	1000	queries/hour	Aug-99
W: Transactional web	~ 50, 500	web inter-actions/sec.	Jul-00
App: app. server & web services		Web Service Interactions/sec (SIPS)	Jun-05

I/O Benchmarks via SPEC

- **SFS 3.0 Attempt by NFS companies to agree on standard benchmark**
 - Run on multiple clients & networks (to prevent bottlenecks)
 - Same caching policy in all clients
 - Reads: 85% full block & 15% partial blocks
 - Writes: 50% full block & 50% partial blocks
 - Average response time: 40 ms
 - Scaling: for every 100 NFS ops/sec, increase capacity 1GB
- **Results: plot of server load (throughput) vs. response time & number of users**
 - Assumes: 1 user => 10 NFS ops/sec
 - 3.0 for NFS 3.0
- **Added SPECmail (mailserver), SPECweb (webserver) benchmarks**

2005 Example SPEC SFS Result: NetApp FAS3050c NFS servers

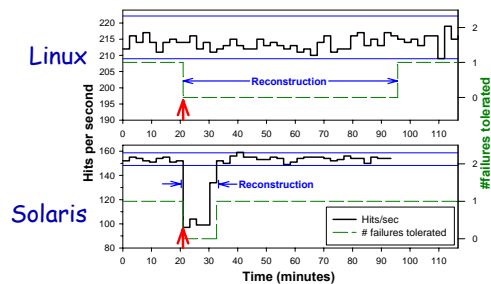
- 2.8 GHz Pentium Xeon microprocessors, 2 GB of DRAM per processor, 1GB of Non-volatile memory per system
- 4 FDDI networks; 32 NFS Daemons, 24 GB file size
- 168 fibre channel disks: 72 GB, 15000 RPM, 2 or 4 FC controllers



Availability benchmark methodology

- **Goal: quantify variation in QoS metrics as events occur that affect system availability**
- **Leverage existing performance benchmarks**
 - to generate fair workloads
 - to measure & trace quality of service metrics
- **Use fault injection to compromise system**
 - hardware faults (disk, memory, network, power)
 - software faults (corrupt input, driver error returns)
 - maintenance events (repairs, SW/HW upgrades)
- **Examine *single-fault* and *multi-fault* workloads**
 - the availability analogues of performance micro- and macro-benchmarks

Example single-fault result



Compares Linux and Solaris reconstruction

- Linux: minimal performance impact but longer window of vulnerability to second fault
- Solaris: large perf. impact but restores redundancy fast

Reconstruction policy (2)

- **Linux:** favors performance over data availability
 - automatically-initiated reconstruction, idle bandwidth
 - virtually no performance impact on application
 - very long window of vulnerability (>1hr for 3GB RAID)
- **Solaris:** favors data availability over app. perf.
 - automatically-initiated reconstruction at high BW
 - as much as 34% drop in application performance
 - short window of vulnerability (10 minutes for 3GB)
- **Windows:** favors neither!
 - *manually-initiated* reconstruction at moderate BW
 - as much as 18% app. performance drop
 - somewhat short window of vulnerability (23 min/3GB)

Introduction to Queuing Theory



- More interested in long term, steady state than in startup => Arrivals = Departures
- **Little's Law:**

$$\text{Mean number tasks in system} = \text{arrival rate} \times \text{mean response time}$$
 - Observed by many, Little was first to prove
- Applies to any system in equilibrium, as long as black box not creating or destroying tasks

Deriving Little's Law

- $Time_{observe}$ = elapsed time that observe a system
- $Number_{task}$ = number of (overlapping) tasks during $Time_{observe}$
- $Time_{accumulated}$ = sum of elapsed times for each task

Then

- **Mean number tasks in system** = $Time_{accumulated} / Time_{observe}$
- **Mean response time** = $Time_{accumulated} / Number_{task}$
- **Arrival Rate** = $Number_{task} / Time_{observe}$

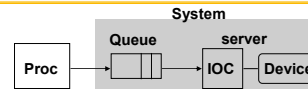
Factoring RHS of 1st equation

- $Time_{accumulated} / Time_{observe} = Time_{accumulated} / Number_{task} \times Number_{task} / Time_{observe}$

Then get Little's Law:

- **Mean number tasks in system** = **Arrival Rate** x **Mean response time**

A Little Queuing Theory: Notation



Notation:

- $Time_{server}$ average time to service a task
- **Average service rate** = $1 / Time_{server}$ (traditionally μ)
- $Time_{queue}$ average time/task in queue
- $Time_{system}$ average time/task in system
- $Time_{system} = Time_{queue} + Time_{server}$
- **Arrival rate** avg no. of arriving tasks/sec (traditionally λ)
- $Length_{server}$ average number of tasks in service
- $Length_{queue}$ average length of queue
- $Length_{system}$ average number of tasks in service
- $Length_{system} = Length_{queue} + Length_{server}$

- **Little's Law:** $Length_{server} = Arrival\ rate \times Time_{server}$
(Mean number tasks = arrival rate x mean service time)

Server Utilization

- For a single server, service rate = $1 / Time_{server}$
- **Server utilization** must be between 0 and 1, since system is in equilibrium (arrivals = departures); often called **traffic intensity**, traditionally ρ
- **Server utilization**
= mean number tasks in service
= **Arrival rate** x $Time_{server}$
- What is disk utilization if get 50 I/O requests per second for disk and average disk service time is 10 ms (0.01 sec)?
- Server utilization = $50/sec \times 0.01\ sec = 0.5$
- Or server is busy on average 50% of time

Time in Queue vs. Length of Queue

- We assume First In First Out (FIFO) queue
- Relationship of time in queue ($Time_{queue}$) to mean number of tasks in queue ($Length_{queue}$)?
- $Time_{queue} = Length_{queue} \times Time_{server}$
+ "Mean time to complete service of task when new task arrives if server is busy"
- New task can arrive at any instant; how predict last part?
- To predict performance, need to know sometime about distribution of events

Distribution of Random Variables

- A variable is random if it takes one of a specified set of values with a specified probability
 - Cannot know exactly next value, but may know probability of all possible values
- I/O Requests can be modeled by a random variable because OS normally switching between several processes generating independent I/O requests
 - Also given probabilistic nature of disks in seek and rotational delays
- Can characterize distribution of values of a random variable with discrete values using a **histogram**
 - Divides range between the min & max values into **buckets**
 - Histograms then plot the number in each bucket as columns
 - Works for discrete values e.g., number of I/O requests?
- What about if not discrete? Very fine buckets

Characterizing distribution of a random variable

Need mean time and a measure of variance

For mean, use **weighted arithmetic mean (WAM)**:

- f_i = frequency of task i
- T_i = time for tasks i

$$\text{Weighted arithmetic mean} = f_1 \times T_1 + f_2 \times T_2 + \dots + f_n \times T_n$$

For variance, instead of standard deviation, use Variance (square of standard deviation) for WAM:

$$\text{Variance} = (f_1 \times T_1^2 + f_2 \times T_2^2 + \dots + f_n \times T_n^2) - WAM^2$$

- If time is milliseconds, Variance units are square milliseconds!

Got a unitless measure of variance?

Squared Coefficient of Variance (C²)

- $C^2 = \text{Variance} / \text{WAM}^2$
 $\Rightarrow C = \sqrt{\text{Variance}/\text{WAM}} = \text{StDev}/\text{WAM}$
 - Unitless measure
- Trying to characterize random events, but need distribution of random events with tractable math
- Most popular such distribution is **exponential distribution**, where $C = 1$
- Note using constant to characterize variability about the mean
 - Invariance of C over time \Rightarrow history of events has no impact on probability of an event occurring now
 - Called **memoryless**, an important assumption to predict behavior
 - (Suppose not; then have to worry about the exact arrival times of requests relative to each other \Rightarrow make math not tractable!)

Poisson Distribution

- Most widely used exponential distribution is Poisson
- Described by probability mass function:

$$\text{Probability}(k) = e^{-a} \times a^k / k!$$
 - where $a = \text{Rate of events} \times \text{Elapsed time}$
- If interarrival times exponentially distributed & use arrival rate from above for rate of events, number of arrivals in time interval t is a **Poisson process**

Time in Queue

- Time new task must wait for server to complete a task assuming server busy
 - Assuming it's a Poisson process
- Average residual service time = $\frac{1}{2} \times \text{Arithmetic mean} \times (1 + C^2)$
 - When distribution is not random & all values = average \Rightarrow standard deviation is 0 $\Rightarrow C$ is 0
 \Rightarrow average residual service time = half average service time
 - When distribution is random & Poisson $\Rightarrow C$ is 1
 \Rightarrow average residual service time = weighted arithmetic mean

Time in Queue

- All tasks in queue ($\text{Length}_{\text{queue}}$) ahead of new task must be completed before task can be serviced
 - Each task takes on average $\text{Time}_{\text{server}}$
 - Task at server takes average residual service time to complete
- Chance server is busy is **server utilization** \Rightarrow expected time for service is Server utilization \times Average residual service time
- $\text{Time}_{\text{queue}} = \text{Length}_{\text{queue}} \times \text{Time}_{\text{server}} + \text{Server utilization} \times \text{Average residual service time}$
- Substituting definitions for $\text{Length}_{\text{queue}}$, Average residual service time, & rearranging:

$$\text{Time}_{\text{queue}} = \frac{\text{Time}_{\text{server}}}{1 - \text{Server utilization}}$$

M/M/1 Queuing Model

- System is in equilibrium
- Times between 2 successive requests arriving, "**interarrival times**", are exponentially distributed
- Number of sources of requests is unlimited "**infinite population model**"
- Server can start next job immediately
- Single queue, no limit to length of queue, and FIFO discipline, so all tasks in line must be completed
- There is one server
- Called M/M/1 (book also derives M/M/m)
 1. Exponentially random request arrival ($C^2 = 1$)
 2. Exponentially random service time ($C^2 = 1$)
 3. 1 server
 - M standing for Markov, mathematician who defined and analyzed the memoryless processes

Example

- 40 disk I/Os / sec, requests are exponentially distributed, and average service time is 20 ms
 \Rightarrow Arrival rate/sec = 40, $\text{Time}_{\text{server}} = 0.02$ sec
1. On average, how utilized is the disk?
 Server utilization = Arrival rate \times $\text{Time}_{\text{server}}$
 $= 40 \times 0.02 = 0.8 = 80\%$
 2. What is the average time spent in the queue?

$$\text{Time}_{\text{queue}} = \frac{\text{Time}_{\text{server}}}{1 - \text{Server utilization}} \times \text{Server utilization}$$

$$= 20 \text{ ms} \times 0.8 / (1 - 0.8) = 20 \times 4 = 80 \text{ ms}$$
 3. What is the average response time for a disk request, including the queuing time and disk service time?

$$\text{Time}_{\text{system}} = \text{Time}_{\text{queue}} + \text{Time}_{\text{server}} = 80 + 20 \text{ ms} = 100 \text{ ms}$$

How much better with 2X faster disk?

- Average service time is **10 ms**
 ⇒ Arrival rate/sec = 40, Time_{server} = **0.01 sec**
- On average, how utilized is the disk?
 Server utilization = Arrival rate × Time_{server}
 = 40 × 0.01 = 0.4 = **40%**
 - What is the average time spent in the queue?
 Time_{queue} = Time_{server} × Server utilization / (1 - Server utilization)
 = 10 ms × 0.4 / (1 - 0.4) = 10 × 2/3 = **6.7 ms**
 - What is the average response time for a disk request, including the queuing time and disk service time?
 Time_{system} = Time_{queue} + Time_{server} = **6.7 + 10 ms = 16.7 ms**
6X faster response time with 2X faster disk!

Value of Queuing Theory in practice

- Learn quickly do not try to utilize resource 100% but how far should back off?
- Allows designers to decide impact of faster hardware on utilization and hence on response time
- Works surprisingly well

Cross cutting Issues: Buses ⇒ point-to-point links and switches

Standard	width	length	Clock rate	MB/s	Max
(Parallel) ATA	8b	0.5 m	133 MHz	133	2
Serial ATA	2b	2 m	3 GHz	300	?
(Parallel) SCSI	16b	12 m	80 MHz (DDR)	320	15
Serial Attach SCSI	1b	10 m	--	375	16,256
PCI	32/64	0.5 m	33 / 66 MHz	533	?
PCI Express	2b	0.5 m	3 GHz	250	?

- No. bits and BW is per direction ⇒ 2X for both directions (not shown).
- Since use fewer wires, commonly increase BW via versions with 2X-12X the number of wires and BW

Storage Example: Internet Archive

- Goal of making a historical record of the Internet
 - Internet Archive began in 1996
 - Wayback Machine interface perform time travel to see what the website at a URL looked like in the past
- It contains over a petabyte (10¹⁵ bytes), and is growing by 20 terabytes (10¹² bytes) of new data per month
- In addition to storing the historical record, the same hardware is used to crawl the Web every few months to get snapshots of the Internet.

Internet Archive Cluster

- 1U storage node PetaBox GB2000 from Capricorn Technologies
- Contains 4 500 GB Parallel ATA (PATA) disk drives, 512 MB of DDR266 DRAM, one 10/100/1000 Ethernet interface, and a 1 GHz C3 Processor from VIA (80x86).
- Node dissipates ≈ 80 watts
- 40 GB2000s in a standard VME rack, ⇒ 80 TB of raw storage capacity
- 40 nodes are connected with a 48-port 10/100 or 10/100/1000 Ethernet switch
- Rack dissipates about 3 KW
- 1 PetaByte = 12 racks



Estimated Cost

- Via processor, 512 MB of DDR266 DRAM, ATA disk controller, power supply, fans, and enclosure = \$500
- 7200 RPM Parallel ATA drives holds 500 GB = \$375.
- 48-port 10/100/1000 Ethernet switch and all cables for a rack = \$3000.
- Cost \$84,500 for a 80-TB rack.
- 160 Disks are ≈ 60% of the cost
- Other costs: power, space,

Estimated Performance

- 7200 RPM Parallel ATA drives holds 500 GB, has an average time seek of 8.5 ms, transfers at 50 MB/second from the disk. The SATA link speed is 133 MB/second.
 - performance of the VIA processor is 1000 MIPS.
 - operating system uses 50,000 CPU instructions for a disk I/O.
 - network protocol stacks uses 100,000 CPU instructions to transmit a data block between the cluster and the external world
- ATA controller overhead is 0.1 ms to perform a disk I/O.
- Average I/O size is 16 KB for accesses to the historical record via the Wayback interface, and 50 KB when collecting a new snapshot
- Disks are limit: ≈ 75 I/Os/s per disk, 300/s per node, 12000/s per rack, or about 200 to 600 Mbytes/sec Bandwidth per rack
- Switch needs to support 1.6 to 3.8 Gbits/second over 40 Gbit/sec links

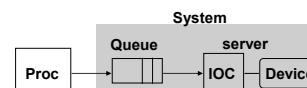
Estimated Reliability

- CPU/memory/enclosure MTTF is 1,000,000 hours (x 40)
- SATA Disk MTTF is 125,000 hours (x 160)
- SATA controller MTTF is 500,000 hours (x 40)
- Ethernet Switch MTTF is 500,000 hours (x 1)
- Power supply MTTF is 200,000 hours (x 40)
- Fan MTTF is 200,000 hours (x 40)
- SATA cable MTTF is 1,000,000 hours (x 40)
- MTTF for the system is 531 hours (≈ 3 weeks)
- 70% of time failures are disks
- 20% of time failures are fans or power supplies

Summary (1/2)

- Disks: Aerial Density now 30%/yr vs. 100%/yr in 2000s
- RAID Techniques: Goal was performance, popularity due to reliability of storage
- TPC: price performance as normalizing configuration feature
 - Auditing to ensure no foul play
 - Throughput with restricted response time is normal measure
- Fault \Rightarrow Latent errors in system \Rightarrow Failure in service
- Components often fail slowly
- Real systems: problems in maintenance, operation as well as hardware, software

Summary (2/2)



- Little's Law: $Length_{system} = rate \times Time_{system}$
(Mean number customers = arrival rate x mean service time)
- Appreciation for relationship of latency and utilization:
 - $Time_{system} = Time_{server} + Time_{queue}$
 - $Time_{queue} = \frac{Time_{server}}{x \text{ Server utilization} / (1 - \text{Server utilization})}$

The End

- The last lecture
 - chapter 6: Storage Systems
- Exam
 - Mon Jan 14th 2008, 14-17h
 - chap 1-6, app A, C & F
 - remark: sample exams on website based on previous edition of book
- Assignment
 - deadline 2b: Dec 3rd
 - deadline 3: Dec 24th (intro by Eyal on Wed Dec 5th, 13.45h)