

Mestrado em Informática

2010/11

A.J.Proença

Tema

Arquitecturas Paralelas (1)

Adaptado de

Computer Organization and Design, 4th Ed, Patterson & Hennessy, © 2009, MK

Performance Summary

The BIG Picture

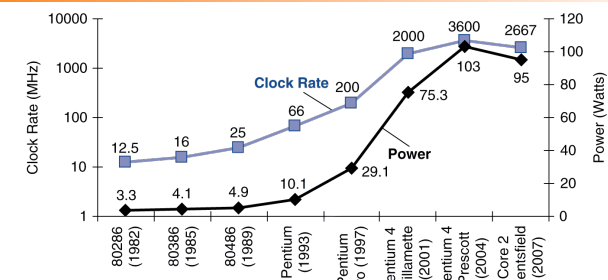
$$\text{CPU Time} = \frac{\text{Instructions}}{\text{Program}} \times \frac{\text{Clock cycles}}{\text{Instruction}} \times \frac{\text{Seconds}}{\text{Clock cycle}}$$

- Performance depends on
 - Algorithm: affects IC, possibly CPI
 - Programming language: affects IC, CPI
 - Compiler: affects IC, CPI
 - Instruction set architecture: affects IC, CPI, T_c

Estrutura do tema AP

1. A evolução das arquitecturas pelo paralelismo
2. Multiprocessadores (SMP e MPP)
3. Data Parallelism: SIMD, Vector, GPU, ...
4. Topologias de interligação

The Power Wall: Power Trends



- In CMOS IC technology

$$\text{Power} = \text{Capacitive load} \times \text{Voltage}^2 \times \text{Frequency}$$

×30

5V → 1V

×1000

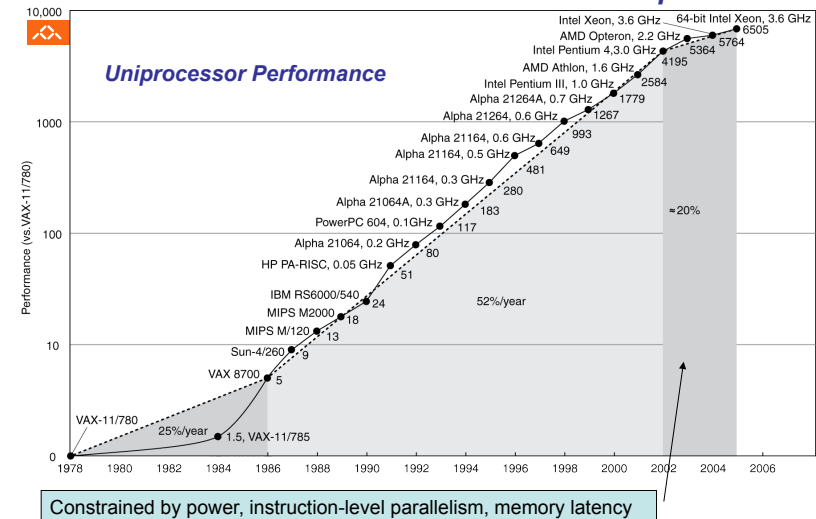
Reducing Power

- Suppose a new CPU has
 - 85% of capacitive load of old CPU
 - 15% voltage and 15% frequency reduction

$$\frac{P_{\text{new}}}{P_{\text{old}}} = \frac{C_{\text{old}} \times 0.85 \times (V_{\text{old}} \times 0.85)^2 \times F_{\text{old}} \times 0.85}{C_{\text{old}} \times V_{\text{old}}^2 \times F_{\text{old}}} = 0.85^4 = 0.52$$

- The power wall
 - We can't reduce voltage further
 - We can't remove more heat
- How else can we improve performance?

The Sea Change: The Switch to Multiprocessors



Multiprocessors

- Multicore microprocessors
 - More than one processor per chip
- Requires explicitly parallel programming
 - Compare with instruction level parallelism
 - Hardware executes multiple instructions at once
 - Hidden from the programmer
 - Hard to do
 - Programming for performance
 - Load balancing
 - Optimizing communication and synchronization

Parallelism and Instructions: Synchronization

- Two processors sharing an area of memory
 - P1 writes, then P2 reads
 - Data race if P1 and P2 don't synchronize
 - Result depends of order of accesses
- Hardware support required
 - Atomic read/write memory operation
 - No other access to the location allowed between the read and write
- Could be a single instruction
 - E.g., atomic swap of register ↔ memory
 - Or an atomic pair of instructions



- Parallel programs may interleave operations in unexpected orders

– Assumptions of associativity may fail

		$(x+y)+z$	$x+(y+z)$
x	-1.50E+38		-1.50E+38
y	1.50E+38	0.00E+00	
z	1.0	1.0	1.50E+38
		1.00E+00	0.00E+00

- Need to validate parallel programs under varying degrees of parallelism

ILP: Multiple Issue



- Static multiple issue
 - Compiler groups instructions into “issue packets”
 - An “issue packet” Specifies multiple concurrent operations
 - Very Long Instruction Word (VLIW)
 - Compiler detects and avoids hazards
- Dynamic multiple issue (“superscalar” processor)
 - CPU examines instruction stream and chooses instructions to issue each cycle
 - Compiler can help by reordering instructions
 - CPU resolves hazards using advanced techniques at runtime



- Pipelining: executing multiple instructions in parallel
- To increase ILP
 - Deeper pipeline
 - Less work per stage \Rightarrow shorter clock cycle
 - Multiple issue
 - Replicate pipeline stages \Rightarrow multiple pipelines
 - Start multiple instructions per clock cycle
 - $CPI < 1$, so use Instructions Per Cycle (IPC)
 - E.g., 4GHz 4-way multiple-issue
 - 16 BIPS, peak $CPI = 0.25$, peak $IPC = 4$
 - But dependencies reduce this in practice

ILP: Speculation



- “Guess” what to do with an instruction
 - Start operation as soon as possible
 - Check whether guess was right
 - If so, complete the operation
 - If not, roll-back and do the right thing
- Common to static and dynamic multiple issue
- Examples
 - Speculate on branch outcome
 - Roll back if path taken is different
 - Speculate on load
 - Avoid load and cache miss delay
 - Roll back if location is updated



- Compiler can reorder instructions
 - e.g., move load before branch
 - Can include “fix-up” instructions to recover from incorrect guess
- Hardware can look ahead for instructions to execute
 - Buffer results until it determines they are actually needed
 - Flush buffers on incorrect speculation

Multilevel Cache Considerations



- Primary cache
 - Focus on **minimizing hit time** for a shorter clock cycle
 - Smaller with smaller block sizes
- L-2 cache
 - Focus on **reducing miss rate** to reduce the penalty of long main memory access times
 - Larger with larger block sizes
 - Higher levels of associativity
 - Hit time has less overall impact
- **Results**
 - L-1 cache usually smaller than a single cache
 - L-1 block size smaller than L-2 block size



- Components of CPU time
 - Program execution cycles
 - Includes cache hit time
 - Memory stall cycles
 - Mainly from cache misses
- With simplifying assumptions: Memory stall cycles

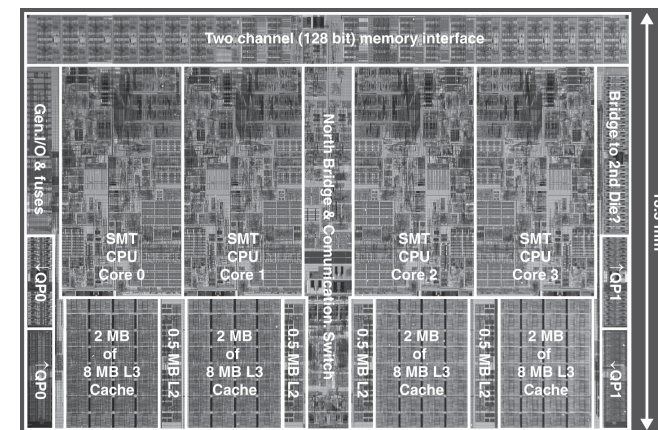
$$= \frac{\text{Memory accesses}}{\text{Program}} \times \text{Miss rate} \times \text{Miss penalty}$$

$$= \frac{\text{Instructions}}{\text{Program}} \times \frac{\text{Misses}}{\text{Instruction}} \times \text{Miss penalty}$$

Multilevel On-Chip Caches: Intel Nehalem



Intel Nehalem 4-core processor



Per core: 32KB L1 I-cache, 32KB L1 D-cache, 512KB L2 cache

3-Level Cache Organization

	Intel Nehalem	AMD Opteron X4
L1 caches (per core)	L1 I-cache: 32KB, 64-byte blocks, 4-way, approx LRU replacement, hit time n/a L1 D-cache: 32KB, 64-byte blocks, 8-way, approx LRU replacement, write-back/allocate, hit time n/a	L1 I-cache: 32KB, 64-byte blocks, 2-way, LRU replacement, hit time 3 cycles L1 D-cache: 32KB, 64-byte blocks, 2-way, LRU replacement, write-back/allocate, hit time 9 cycles
L2 unified cache (per core)	256KB, 64-byte blocks, 8-way, approx LRU replacement, write-back/allocate, hit time n/a	512KB, 64-byte blocks, 16-way, approx LRU replacement, write-back/allocate, hit time n/a
L3 unified cache (shared)	8MB, 64-byte blocks, 16-way, replacement n/a, write-back/allocate, hit time n/a	2MB, 64-byte blocks, 32-way, replace block shared by fewest cores, write-back/allocate, hit time 32 cycles

n/a: data not available

Measuring Performance: SPEC CPU Benchmark

- Programs used to measure performance
 - Supposedly typical of actual workload
- Standard Performance Evaluation Corp (SPEC)
 - Develops benchmarks for CPU, I/O, Web, ...
- SPEC CPU2006
 - Elapsed time to execute a selection of programs
 - Negligible I/O, so focuses on CPU performance
 - Normalize relative to reference machine
 - Summarize as geometric mean of performance ratios
 - CINT2006 (integer) and CFP2006 (floating-point)

$$\sqrt[n]{\prod_{i=1}^n \text{Execution time ratio}_i}$$

Measuring Performance: CINT2006 for Opteron X4 2356 (Barcelona)

Name	Description	IC×10 ⁹	CPI	Tc (ns)	Exec time	Ref time	SPECratio
perl	Interpreted string processing	2,118	0.75	0.40	637	9,777	15.3
bzip2	Block-sorting compression	2,389	0.85	0.40	817	9,650	11.8
gcc	GNU C Compiler	1,050	1.72	0.47	24	8,050	11.1
mcf	Combinatorial optimization	336	10.00	0.40	1,345	9,120	6.8
go	Go game (AI)	1,658	1.09	0.40	721	10,490	14.6
hmmer	Search gene sequence	2,783	0.80	0.40	890	9,330	10.5
sjeng	Chess game (AI)	2,176	0.96	0.48	37	12,100	14.5
libquantum	Quantum computer simulation	1,623	1.61	0.40	1,047	20,720	19.8
h264avc	Video compression	3,102	0.80	0.40	993	22,130	22.3
omnetpp	Discrete event simulation	587	2.94	0.40	690	6,250	9.1
astar	Games/path finding	1,082	1.79	0.40	773	7,020	9.1
xalancbmk	XML parsing	1,058	2.70	0.40	1,143	6,900	6.0
Geometric mean							11.7

High cache miss rates

Measuring Performance: CINT2006 for Opteron X4 2356 (Barcelona)

Name	CPI	L1 D cache misses/1000 instr	L2 D cache misses/1000 instr	DRAM accesses/1000 instr
perl	0.75	3.5	1.1	1.3
bzip2	0.85	11.0	5.8	2.5
gcc	1.72	24.3	13.4	14.8
mcf	10.00	106.8	88.0	88.5
go	1.09	4.5	1.4	1.7
hmmer	0.80	4.4	2.5	0.6
sjeng	0.96	1.9	0.6	0.8
libquantum	1.61	33.0	33.1	47.7
h264avc	0.80	8.8	1.6	0.2
omnetpp	2.94	30.9	27.7	29.8
astar	1.79	16.3	9.2	8.2
xalancbmk	2.70	38.0	15.8	11.4
Median	1.35	13.6	7.5	5.4

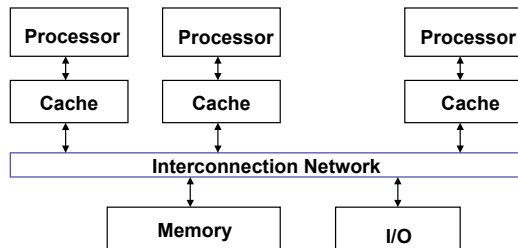
Estrutura do tema AP

1. A evolução das arquiteturas pelo paralelismo
2. Multiprocessadores (SMP e MPP)
3. Data Parallelism: SIMD, Vector, GPU, ...
4. Topologias de interligação

- Goal: connecting multiple computers to get higher performance
 - Multiprocessors
 - Scalability, availability, power efficiency
- Job-level (process-level) parallelism
 - High throughput for independent jobs
- Parallel processing program
 - Single program run on multiple processors
- Multicore microprocessors
 - Chips with multiple processors (cores)

The Big Picture:
Where are We Now?

- **Multiprocessor** – a computer system with at least two processors



- Can deliver high throughput for independent jobs via **job-level parallelism** or **process-level parallelism**
- And improve the run time of a *single* program that has been specially crafted to run on a multiprocessor - a **parallel processing program**

Multicores Now Common

- The power challenge has forced a change in the design of microprocessors
 - Since 2002 the rate of improvement in the response time of programs has slowed from a factor of 1.5 per year to less than a factor of 1.2 per year
- Today's microprocessors typically contain more than one core – **Chip Multicore microProcessors (CMPs)** – in a single IC
 - The number of cores is expected to double every two years

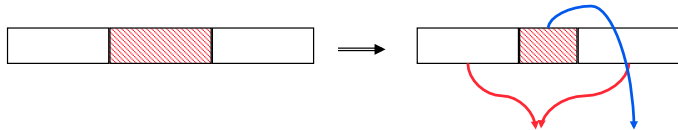
Product	AMD Barcelona	Intel Nehalem	IBM Power 6	Sun Niagara 2
Cores per chip	4	4	2	8
Clock rate	2.5 GHz	~2.5 GHz?	4.7 GHz	1.4 GHz
Power	120 W	~100 W?	~100 W?	94 W

Encountering Amdahl's Law

- Speedup due to enhancement E is

$$\text{Speedup w/ E} = \frac{\text{Exec time w/o E}}{\text{Exec time w/ E}}$$

- Suppose that enhancement E accelerates a fraction F (F < 1) of the task by a factor S (S > 1) and the remainder of the task is unaffected



$$\text{ExTime w/ E} = \text{ExTime w/o E} \times ((1-F) + F/S)$$

$$\text{Speedup w/ E} = 1 / ((1-F) + F/S)$$

Example 2: Amdahl's Law

$$\text{Speedup w/ E} = 1 / ((1-F) + F/S)$$

- Consider summing 10 scalar variables and two 10 by 10 matrices (matrix sum) on 10 processors

$$\text{Speedup w/ E} = 1 / (.091 + .909/10) = 1/0.1819 = 5.5$$

- What if there are 100 processors?

$$\text{Speedup w/ E} = 1 / (.091 + .909/100) = 1/0.10009 = 10.0$$

- What if the matrices are 100 by 100 (or 10,010 adds in total) on 10 processors?

$$\text{Speedup w/ E} = 1 / (.001 + .999/10) = 1/0.1009 = 9.9$$

- What if there are 100 processors?

$$\text{Speedup w/ E} = 1 / (.001 + .999/100) = 1/0.01099 = 91$$

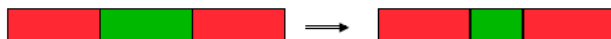
Not Fooling Yourself: Amdahl's Law

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

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

Best you could ever hope to do:

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



Scaling

- To get good speedup on a multiprocessor while keeping the problem size fixed is harder than getting good speedup by increasing the size of the problem.

- **Strong scaling** – when speedup can be achieved on a multiprocessor without increasing the size of the problem
- **Weak scaling** – when speedup is achieved on a multiprocessor by increasing the size of the problem proportionally to the increase in the number of processors

- Load balancing is another important factor. Just a single processor with twice the load of the others cuts the speedup almost in half

Multiprocessor/Clusters Key Questions

- Q1 – How do they share data?
- Q2 – How do they coordinate?
- Q3 – How scalable is the architecture?
How many processors can be supported?

Shared Memory Multiprocessor (SMP)

- Q1 – Single address space shared by all processors
- Q2 – Processors coordinate/communicate through shared variables in memory (via loads and stores)
 - Use of shared data must be coordinated via **synchronization** primitives (locks) that allow access to data to only one processor at a time
- They come in two styles
 - Uniform memory access (UMA) multiprocessors
 - Nonuniform memory access (NUMA) multiprocessors
 - Programming NUMAs are harder
 - But NUMAs can scale to larger sizes and have lower latency to local memory

Summing 100,000 Numbers on 100 Proc. SMP

- Processors start by running a loop that sums their subset of vector A numbers (vectors A and sum are **shared** variables, Pn is the processor's number, i is a **private** variable)

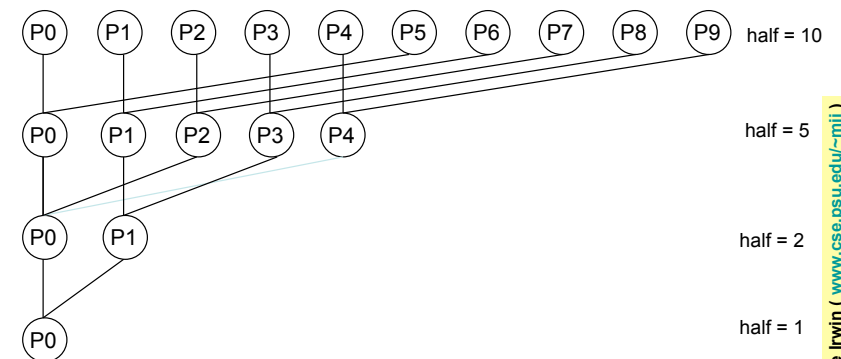
```
sum[Pn] = 0;
for (i = 1000*Pn; i < 1000*(Pn+1); i = i + 1)
    sum[Pn] = sum[Pn] + A[i];
```

- The processors then coordinate in adding together the partial sums (half is a **private** variable initialized to 100 (the number of processors)) – **reduction**

```
repeat
    synch(); /*synchronize first
    if (half%2 != 0 && Pn == 0)
        sum[0] = sum[0] + sum[half-1];
    half = half/2;
    if (Pn < half) sum[Pn] = sum[Pn] + sum[Pn+half]
until (half == 1); /*final sum in sum[0]
```

An Example with 10 Processors

sum[P0]sum[P1]sum[P2] sum[P3]sum[P4]sum[P5]sum[P6] sum[P7]sum[P8] sum[P9]

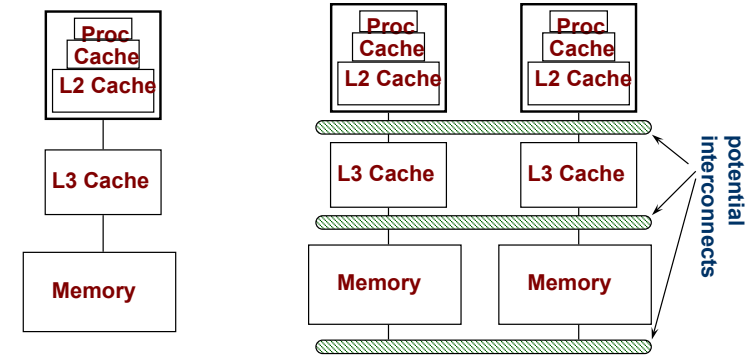


Process Synchronization

- Need to be able to coordinate processes working on a common task
- Lock variables (**semaphores**) are used to coordinate or synchronize processes
- Need an architecture-supported arbitration mechanism to decide which processor gets access to the lock variable
 - Single bus provides arbitration mechanism, since the bus is the only path to memory – the processor that gets the bus wins
- Need an architecture-supported operation that locks the variable
 - Locking can be done via an **atomic swap operation**

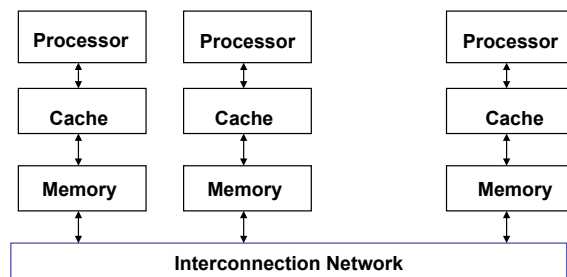
Locality and Parallelism

Conventional Storage Hierarchy



Message Passing Multiprocessors (MPP)

- Each processor has its own private address space
- Q1 – Processors share data by *explicitly* sending and receiving information (**message passing**)
- Q2 – Coordination is built into message passing primitives (**message send** and **message receive**)



Summing 100,000 Numbers on 100 Proc. MPP

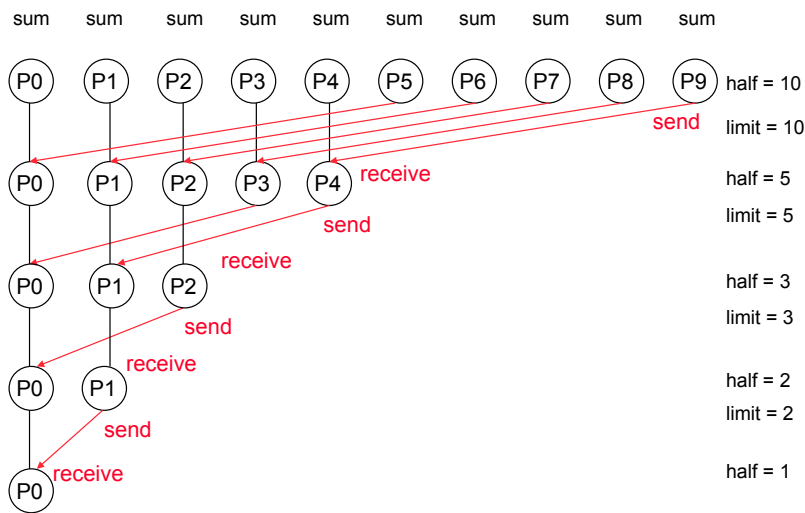
- Start by distributing 1000 elements of vector A to each of the local memories and summing each subset in parallel

```
sum = 0;
for (i = 0; i < 1000; i = i + 1)
    sum = sum + A[i];    /* sum local array subset
```

- The processors then coordinate in adding together the sub sums (P_n is the number of processors, `send(x, y)` sends value y to processor x, and `receive()` receives a value)

```
half = 100;
limit = 100;
repeat
    half = (half+1)/2;    /*dividing line
    if (Pn >= half && Pn < limit) send(Pn-half, sum);
    if (Pn < (limit/2)) sum = sum + receive();
    limit = half;
until (half == 1);    /*final sum in P0's sum
```

An Example with 10 Processors



CSE431 Chapter 7A.37

Irwin, PSU, 2008

Pros and Cons of Message Passing

- ❑ Message sending and receiving is *much* slower than addition, for example
- ❑ But message passing multiprocessors are much easier for hardware designers to design
 - Don't have to worry about cache coherency for example
- ❑ The advantage for programmers is that communication is explicit, so there are fewer "performance surprises" than with the implicit communication in cache-coherent SMPs.
 - Message passing standard MPI-2 (www.mpi-forum.org)
- ❑ However, it's harder to port a sequential program to a message passing multiprocessor since every communication must be identified in advance.
 - With cache-coherent shared memory the hardware figures out what data needs to be communicated

CSE431 Chapter 7A.38

Irwin, PSU, 2008

Multithreading on A Chip

- ❑ Find a way to "hide" true data dependency stalls, cache miss stalls, and branch stalls by finding instructions (from other process threads) that are **independent** of those stalling instructions
- ❑ **Hardware multithreading** – increase the utilization of resources on a chip by allowing multiple processes (**threads**) to share the functional units of a single processor
 - Processor must duplicate the state hardware for each thread – a separate register file, PC, instruction buffer, and store buffer for each thread
 - The caches, TLBs, BHT, BTB, RUU can be shared (although the miss rates may increase if they are not sized accordingly)
 - The memory can be shared through virtual memory mechanisms
 - Hardware must support *efficient* thread context switching

CSE431 Chapter 7A.39

Irwin, PSU, 2008

Multithreading



- Performing multiple threads of execution in parallel
 - Replicate registers, PC, etc.
 - Fast switching between threads
- Fine-grain multithreading
 - Switch threads after each cycle
 - Interleave instruction execution
 - If one thread stalls, others are executed
- Coarse-grain multithreading
 - Only switch on long stall (e.g., L2-cache miss)
 - Simplifies hardware, but doesn't hide short stalls (eg, data hazards)

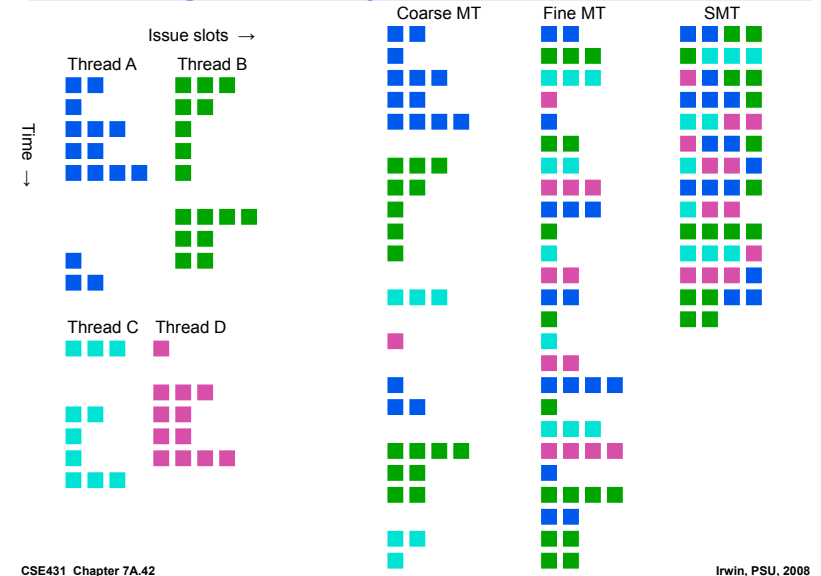
AJProença, Sistemas de Computação e Desempenho, MInf, UMinho, 2010/11

40

Simultaneous Multithreading

- In multiple-issue dynamically scheduled processor
 - Schedule instructions from multiple threads
 - Instructions from independent threads execute when function units are available
 - Within threads, dependencies handled by scheduling and register renaming
- Example: Intel Pentium-4 HT (*Hyper-Threading*)
 - Two threads: duplicated registers, shared function units and caches

Threading on a 4-way SS Processor Example



Future of Multithreading

- Will it survive? In what form?
- Power considerations \Rightarrow simplified microarchitectures
 - Simpler forms of multithreading
- Tolerating cache-miss latency
 - Thread switch may be most effective
- Multiple simple cores might share resources more effectively

Review: Multiprocessor Basics

- Q1 – How do they share data?
- Q2 – How do they coordinate?
- Q3 – How scalable is the architecture? How many processors?

		# of Proc	
Communication model	Message passing	8 to 2048	
	Shared address	NUMA	8 to 256
		UMA	2 to 64
Physical connection	Network	8 to 256	
	Bus	2 to 36	