CSC 2224: Parallel Computer Architecture and Programming Parallel Processing, Multicores

Prof. Gennady Pekhimenko

University of Toronto

Fall 2018

The content of this lecture is adapted from the lectures of Onur Mutlu @ CMU

Summary

- Parallelism

- Multiprocessing fundamentals

- Amdahl's Law

- Why Multicores?
  - Alternatives
  - Examples

Flynn's Taxonomy of Computers

- Mike Flynn, "Very High-Speed Computing Systems," Proc. of IEEE, 1966

- SISD: Single instruction operates on single data element

- SIMD: Single instruction operates on multiple data elements
  - Array processor
  - Vector processor

- MISD: Multiple instructions operate on single data element
  - Closest form: systolic array processor, streaming processor

- MIMD: Multiple instructions operate on multiple data elements (multiple instruction streams)
  - Multiprocessor
  - Multithreaded processor

Why Parallel Computers?

- Parallelism: Doing multiple things at a time

- Things: instructions, operations, tasks

- Main Goal: Improve performance (Execution time or task throughput)

  - Execution time of a program governed by Amdahl's Law

- Other Goals

  – Reduce power consumption

    - (4N units at freq F/4) consume less power than (N units at freq F)

    - Why?

  – Improve cost efficiency and scalability, reduce complexity

    - Harder to design a single unit that performs as well as N simpler units

4

Types of Parallelism & How to Exploit Them

- ## Instruction Level Parallelism
  - Different instructions within a stream can be executed in parallel
  - Pipelining, out-of-order execution, speculative execution, VLIW
  - Dataflow

- ## Data Parallelism
  - Different pieces of data can be operated on in parallel
  - SIMD: Vector processing, array processing
  - Systolic arrays, streaming processors

- ## Task Level Parallelism
  - Different "tasks/threads" can be executed in parallel
  - Multithreading
  - Multiprocessing (multi-core)

Task-Level Parallelism

- Partition a single problem into multiple related tasks (threads)
  - Explicitly: Parallel programming
    - Easy when tasks are natural in the problem
    - Difficult when natural task boundaries are unclear
  - Transparently/implicitly: Thread level speculation
    - Partition a single thread speculatively
- Run many independent tasks (processes) together
  - Easy when there are many processes
    - Batch simulations, different users, cloud computing
  - Does not improve the performance of a single task

Multiprocessing Fundamentals

Multiprocessor Types

- Loosely coupled multiprocessors
  - No shared global memory address space
  - Multicomputer network
    - Network-based multiprocessors
  - Usually programmed via message passing
    - Explicit calls (send, receive) for communication

Multiprocessor Types (2)

- Tightly coupled multiprocessors
  - Shared global memory address space
  - Traditional multiprocessing: symmetric multiprocessing (SMP)
    - Existing multi-core processors, multithreaded processors
  - Programming model similar to uniprocessors (i.e., multitasking uniprocessor) except
    - Operations on shared data require synchronization

Main Issues in Tightly-Coupled MP

- Shared memory synchronization
  - Locks, atomic operations

- Cache consistency
  - More commonly called cache coherence

- Ordering of memory operations
  - What should the programmer expect the hardware to provide?

- Resource sharing, contention, partitioning

- Communication: Interconnection networks

- Load imbalance

Metrics of Multiprocessors

Parallel Speedup
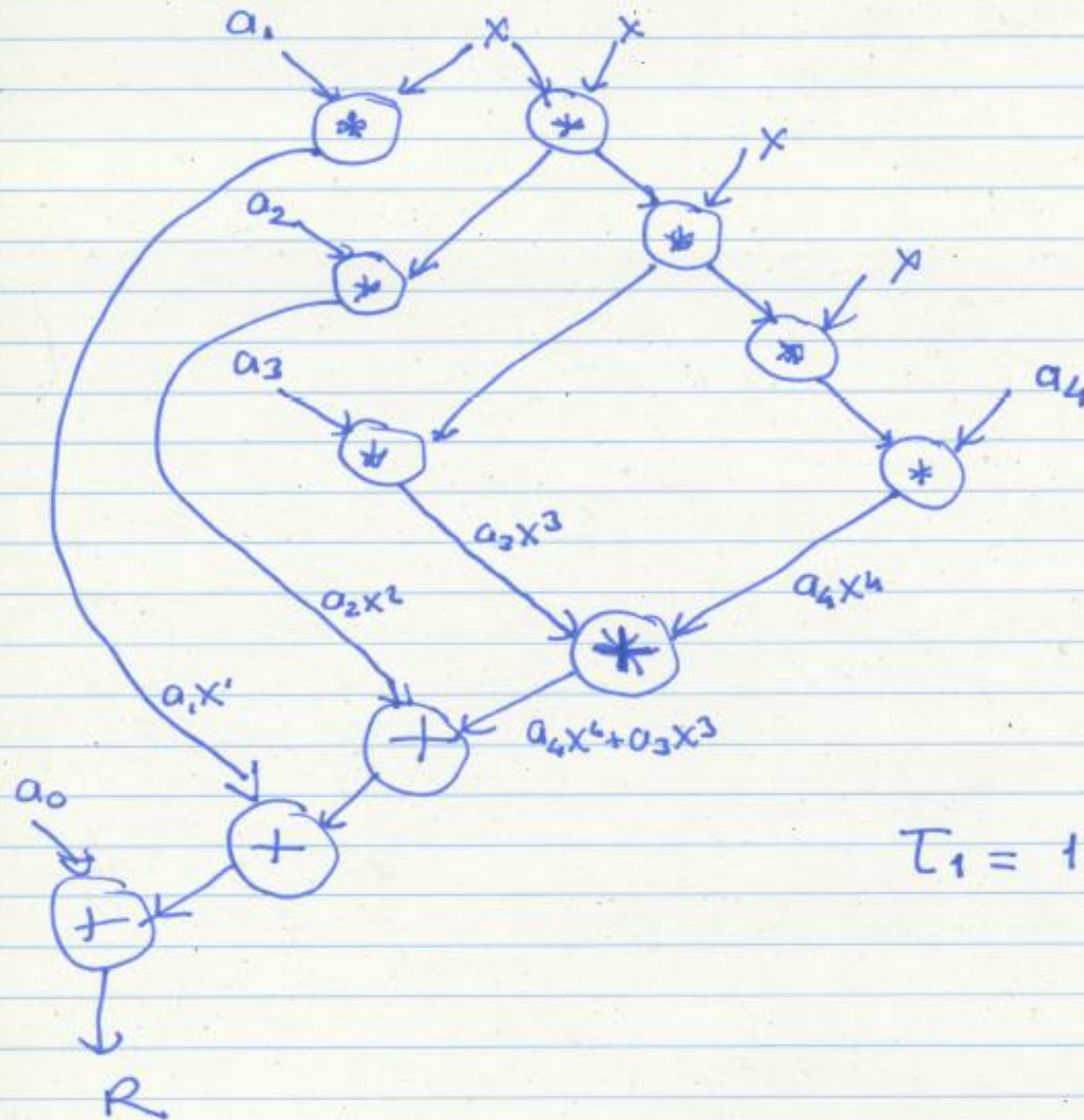
Time to execute the program with 1 processor

divided by

Time to execute the program with N processors

Parallel Speedup Example

- $a_4x^4 + a_3x^3 + a_2x^2 + a_1x + a_0$

- Assume each operation 1 cycle, no communication cost, each op can be executed in a different processor

- How fast is this with a single processor?

  – Assume no pipelining or concurrent execution of instructions

- How fast is this with 3 processors?

$$R = a_4 x^4 + a_3 x^3 + a_2 x^2 + a_1 x + a_0$$

Single processor :   11 operations   (DRAW the data flow graph)

a_1
x x

a_2

x

a_3

x

a_4

$a_3 x^3$

$a_2 x^2$

$a_4 x^4$

$a_1 x^1$

$a_4 x^4 + a_3 x^3$

a_0

$T_1 = 11$ cycles

R

$$R = a_4 x^4 + a_3 x^3 + a_2 x^2 + a_1 x + a_0$$

Three processors : T_3 (exec. time with 3 proc.)
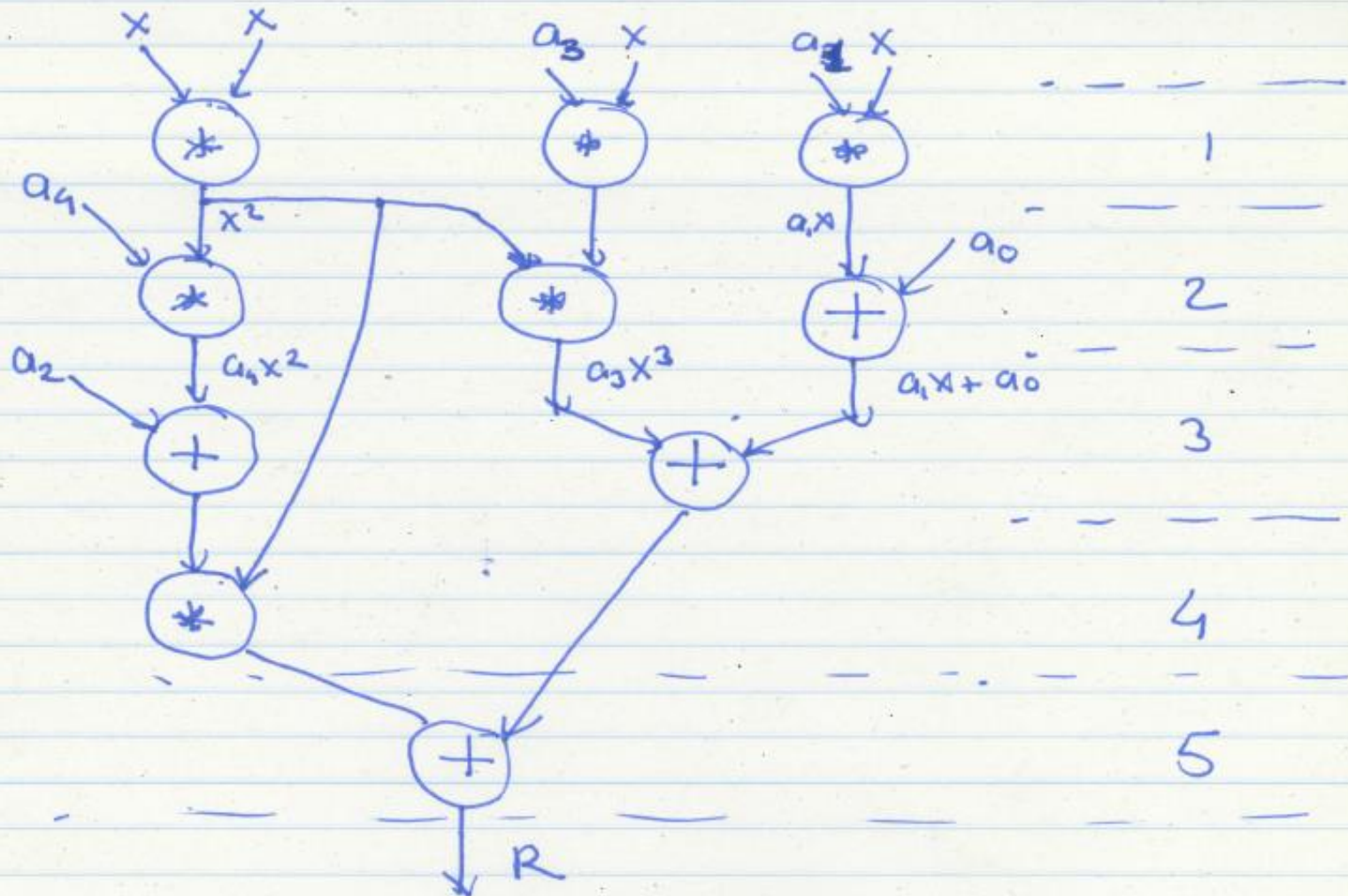
$T_3 = \underline{5 \text{ cycles}}$

Speedup with 3 Processors

$$\tau_3 = \underline{5 \text{ cycles}}$$

$$\text{Speedup with 3 processors} = \frac{11}{5} = 2.2$$

$$\left(\frac{\tau_1}{\tau_3}\right)$$

Is this a fair comparison?

Revisiting the Single-Processor

Revisit T_1

Better single-processor algorithm:

$$R = a_4 x^4 + a_3 x^3 + a_2 x^2 + a_1 x + a_0$$

$$R = (((a_4 x + a_3) x + a_2) x + a_1) x + a_0$$

(Horner's method)

Horner, "A new method of solving numerical equations of all orders, by continuous approximation," Philosophical Transactions of the Royal Society, 1819.

a_4 X

a_3

a_2

$\times$

a_1

$T_1 = 8$ cycles

$\times$

$\times$

a_0
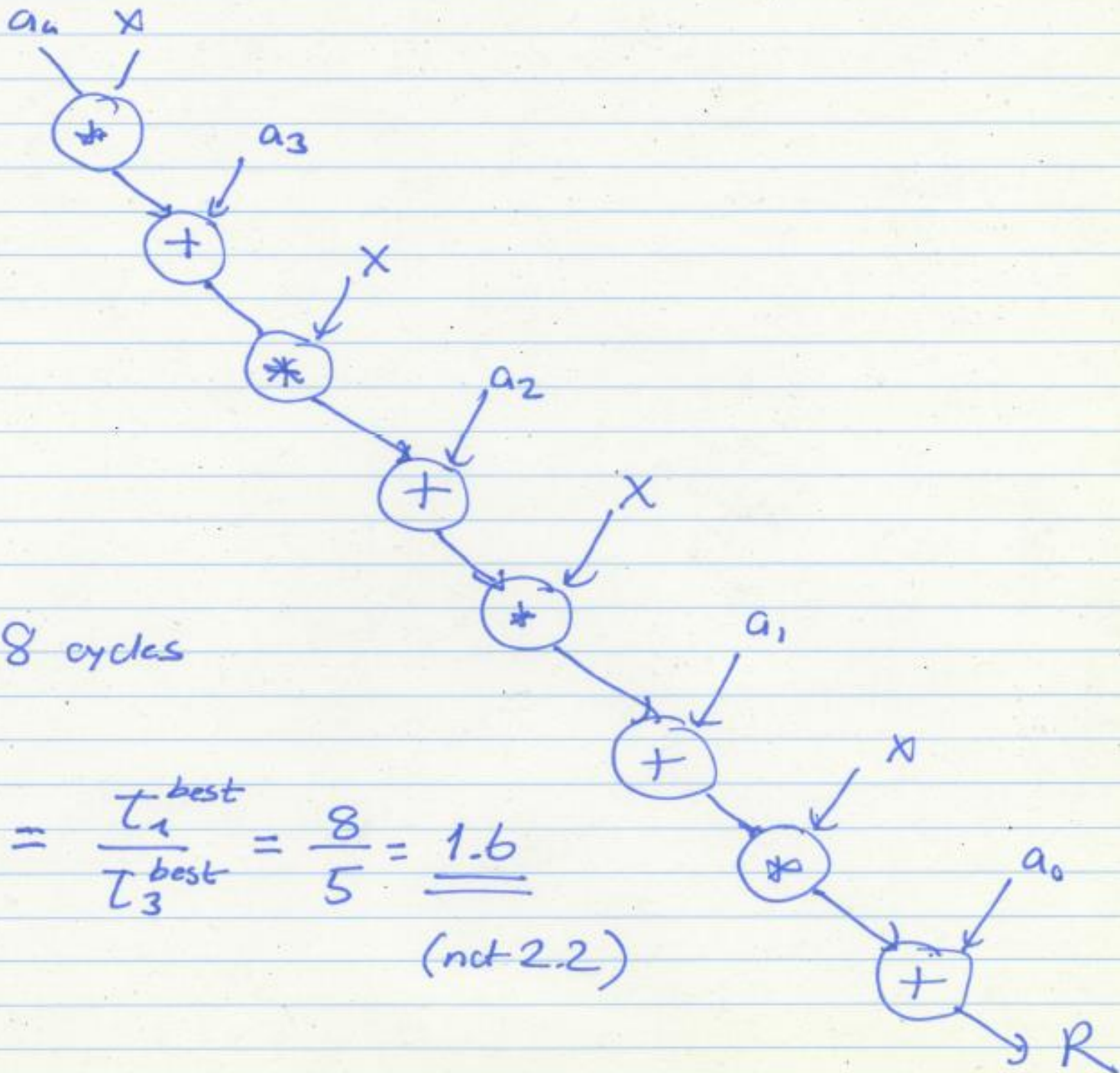
Speedup
with $\quad = \quad \dfrac{T_1^{best}}{T_3^{best}} = \dfrac{8}{5} = \underline{\underline{1.6}}$
3 procs.

$(\text{not } 2.2)$

$\to R$

Takeaway

- To calculate parallel speedup fairly you need to use the best known algorithm for each system with N processors

- If not, you can get superlinear speedup

Superlinear Speedup

- Can speedup be greater than P with P processing elements?

- Consider:
  - Cache effects
  - Memory effects
  - Working set

- Happens in two ways:
  - Unfair comparisons
  - Memory effects

Caveats of Parallelism (I)

Speedup

superlinear speedup region

— linear speedup

— reality

1
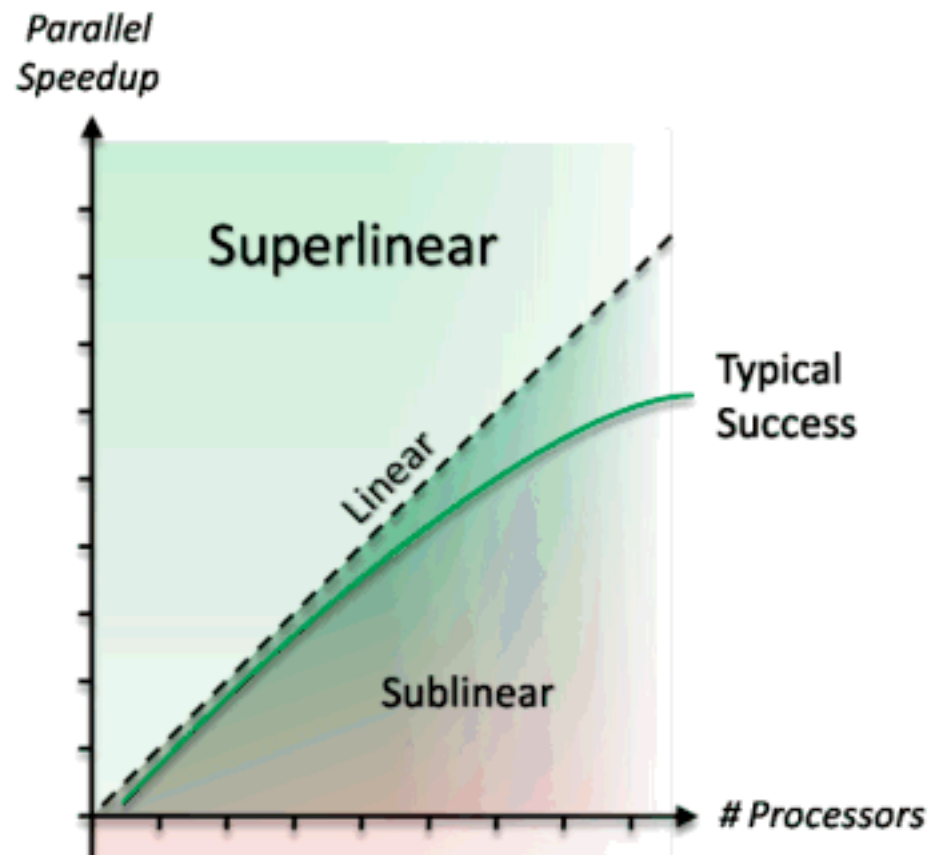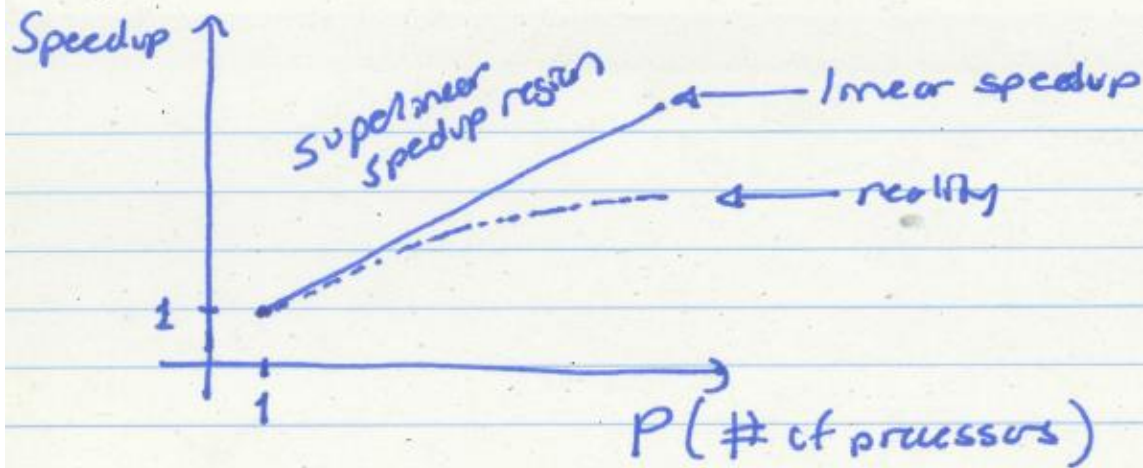
P (# of processors)

Why the reality?   (diminishing returns)

$$\tau_P = \alpha \cdot \frac{\tau_1}{P} + (1-\alpha) \cdot \tau_1$$

parallelizable part / fraction
of the single-processor
program

non-parallelizable part

Amdahl's Law

$$\text{Speedup with } p \text{ proc.} = \frac{t_1}{t_p} = \frac{1}{\frac{\alpha}{p} + (1-\alpha)}$$

$$\text{Speedup as } p \to \infty = \frac{1}{1-\boxed{\alpha}} \longrightarrow \text{bottleneck for parallel Speedup}$$

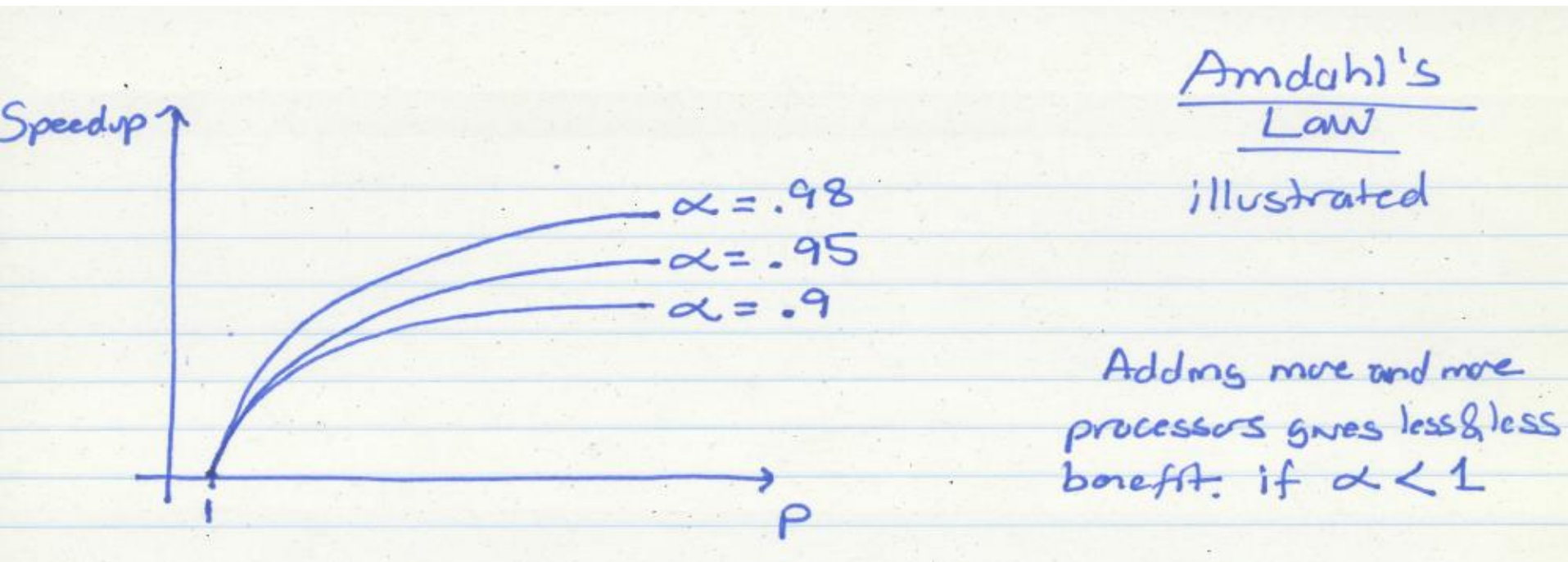Amdahl, "Validity of the single processor approach to achieving large scale computing capabilities," AFIPS 1967.

23

Amdahl's Law

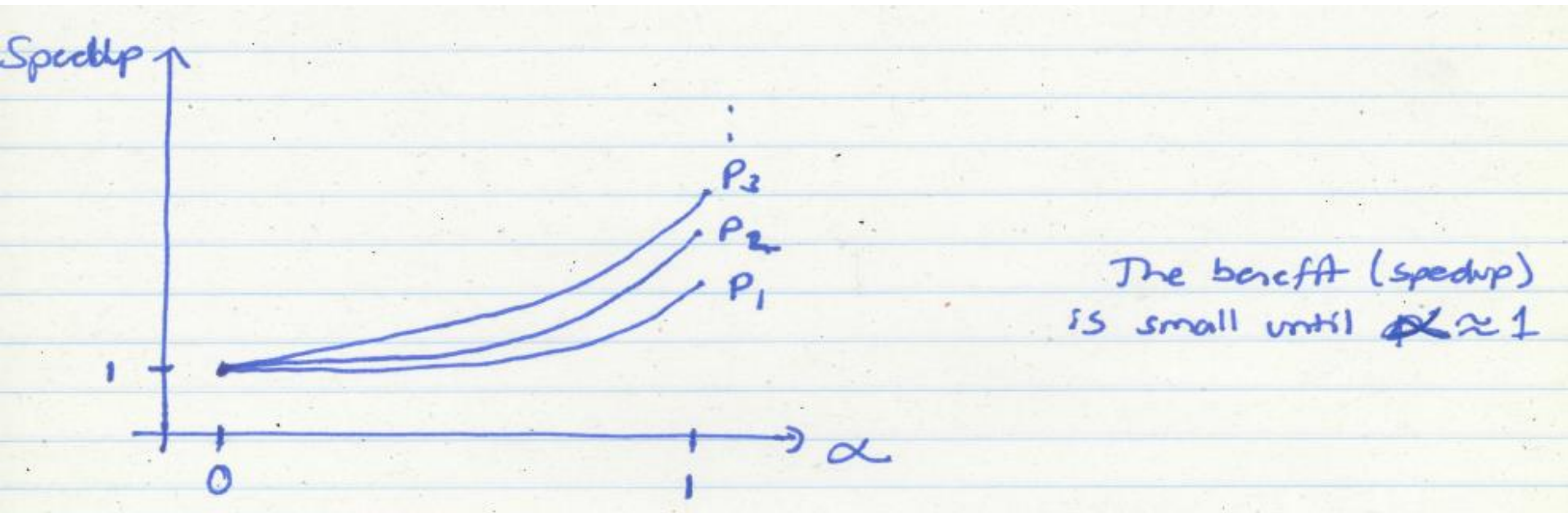– f: Parallelizable fraction of a program

– P: Number of processors

$$\text{Speedup} = \frac{1}{1 - f + \dfrac{f}{P}}$$

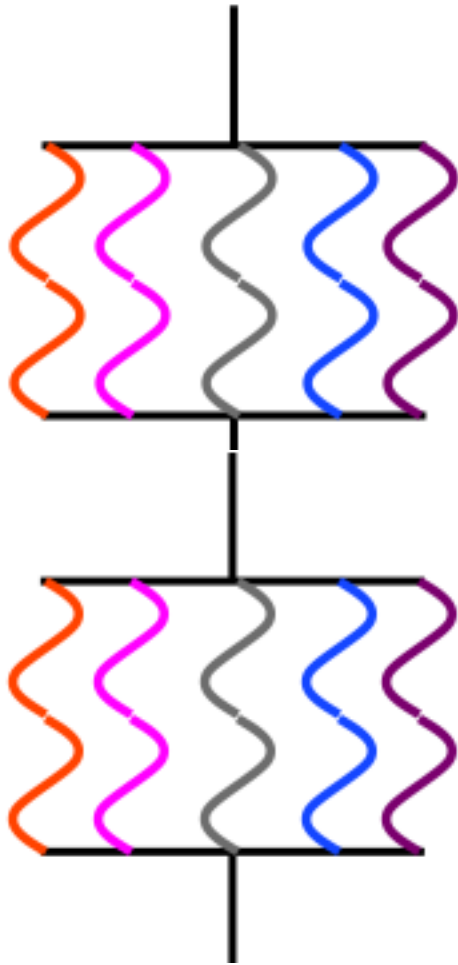• Maximum speedup limited by serial portion:

Serial bottleneck

Amdahl's Law Implication 1

Speedup ↑

$\alpha = .98$
$\alpha = .95$
$\alpha = .9$

P

Amdahl's
Law

illustrated

Adding more and more
processors gives less & less
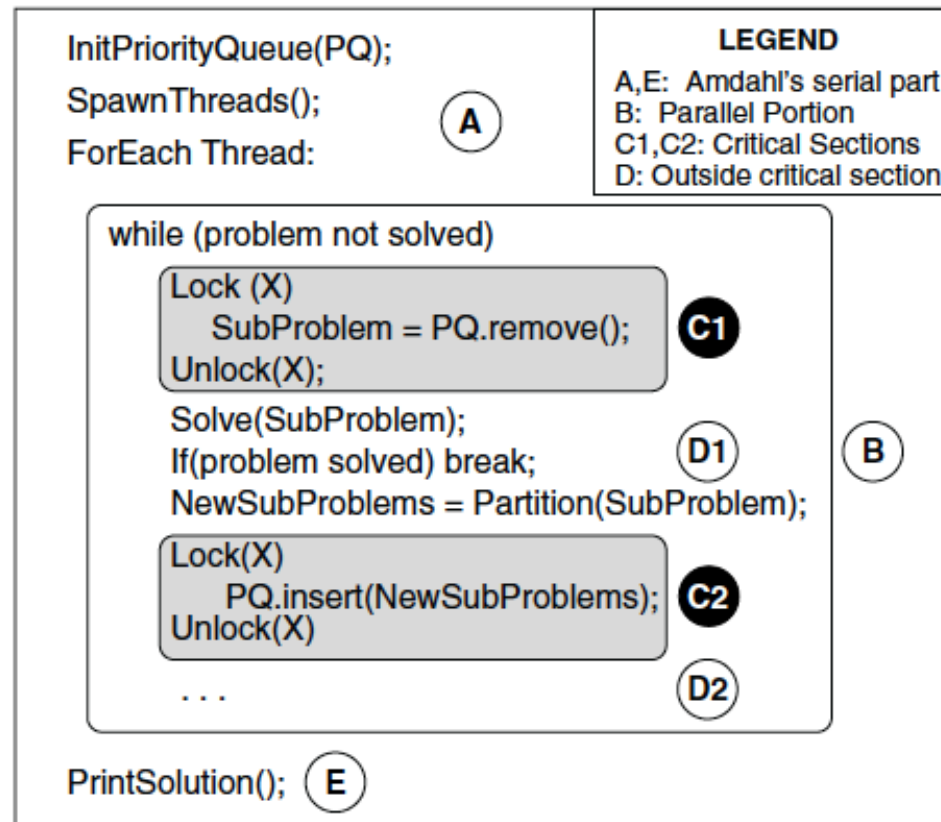benefit if $\alpha < 1$

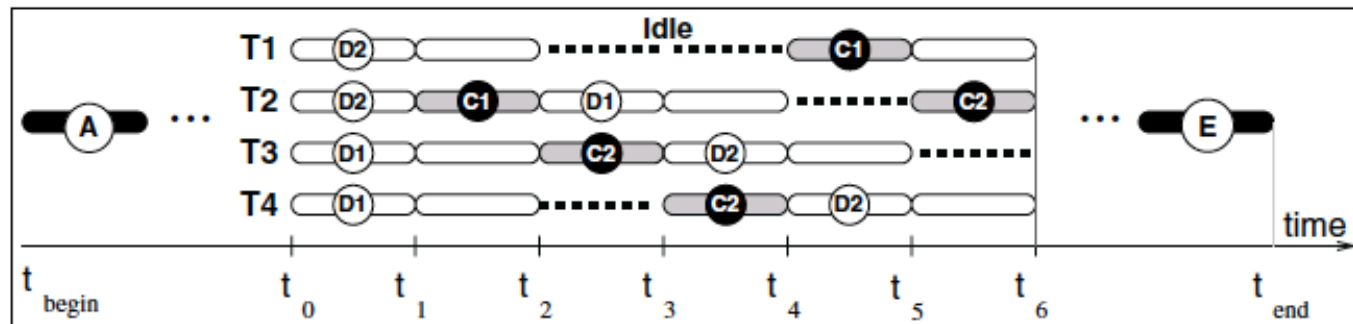# Amdahl's Law Implication 2

Why the Sequential Bottleneck?

- Parallel machines have the sequential bottleneck

- Main cause: Non-parallelizable operations on data (e.g. non-parallelizable loops)

    for (i = 0 ; i < N; i++)
        A[i] = (A[i] + A[i-1]) / 2

- Single thread prepares data and spawns parallel tasks

Another Example of Sequential Bottleneck


```
InitPriorityQueue(PQ);
SpawnThreads();                          (A)
ForEach Thread:
```

LEGEND
A,E: Amdahl's serial part
B: Parallel Portion
C1,C2: Critical Sections
D: Outside critical section

```
while (problem not solved)

    Lock (X)
        SubProblem = PQ.remove();        C1
    Unlock(X);

    Solve(SubProblem);
    If(problem solved) break;            D1          B
    NewSubProblems = Partition(SubProblem);

    Lock(X)
        PQ.insert(NewSubProblems);       C2
    Unlock(X)

    . . .                                D2

PrintSolution();    E
```

(a)

Caveats of Parallelism (II)

- Amdahl's Law
  - f: Parallelizable fraction of a program
  - P: Number of processors

$$\text{Speedup} = \frac{1}{1 - f + \dfrac{f}{P}}$$

- Parallel portion is usually not perfectly parallel
  - Synchronization overhead (e.g., updates to shared data)
  - Load imbalance overhead (imperfect parallelization)
  - Resource sharing overhead (contention among N processors)

30

Bottlenecks in Parallel Portion

- Synchronization: Operations manipulating shared data cannot be parallelized
  - Locks, mutual exclusion, barrier synchronization
  - Communication: Tasks may need values from each other

- Load Imbalance: Parallel tasks may have different lengths
  - Due to imperfect parallelization or microarchitectural effects
  - Reduces speedup in parallel portion

- Resource Contention: Parallel tasks can share hardware resources, delaying each other
  - Replicating all resources (e.g., memory) expensive
  - Additional latency not present when each task runs alone

Difficulty in Parallel Programming

- Little difficulty if parallelism is natural
  - "Embarrassingly parallel" applications
  - Multimedia, physical simulation, graphics
  - Large web servers, databases?

- Big difficulty is in
  - Harder to parallelize algorithms
  - Getting parallel programs to work correctly
  - Optimizing performance in the presence of bottlenecks

- Much of **parallel computer architecture** is about
  - Designing machines that overcome the sequential and parallel bottlenecks to achieve higher performance and efficiency
  - Making programmer's job easier in writing correct and high-performance parallel programs

Parallel and Serial Bottlenecks

- How do you alleviate some of the serial and parallel bottlenecks in a multi-core processor?

- We will return to this question in future lectures

- Reading list:
  - Annavaram et al., "Mitigating Amdahl's Law Through EPI Throttling," ISCA 2005.
  - Suleman et al., "Accelerating Critical Section Execution with Asymmetric Multi-Core Architectures," ASPLOS 2009.
  - Joao et al., "Bottleneck Identification and Scheduling in Multithreaded Applications," ASPLOS 2012.
  - Ipek et al., "Core Fusion: Accommodating Software Diversity in Chip Multiprocessors," ISCA 2007.

Multicores

Moore's Law
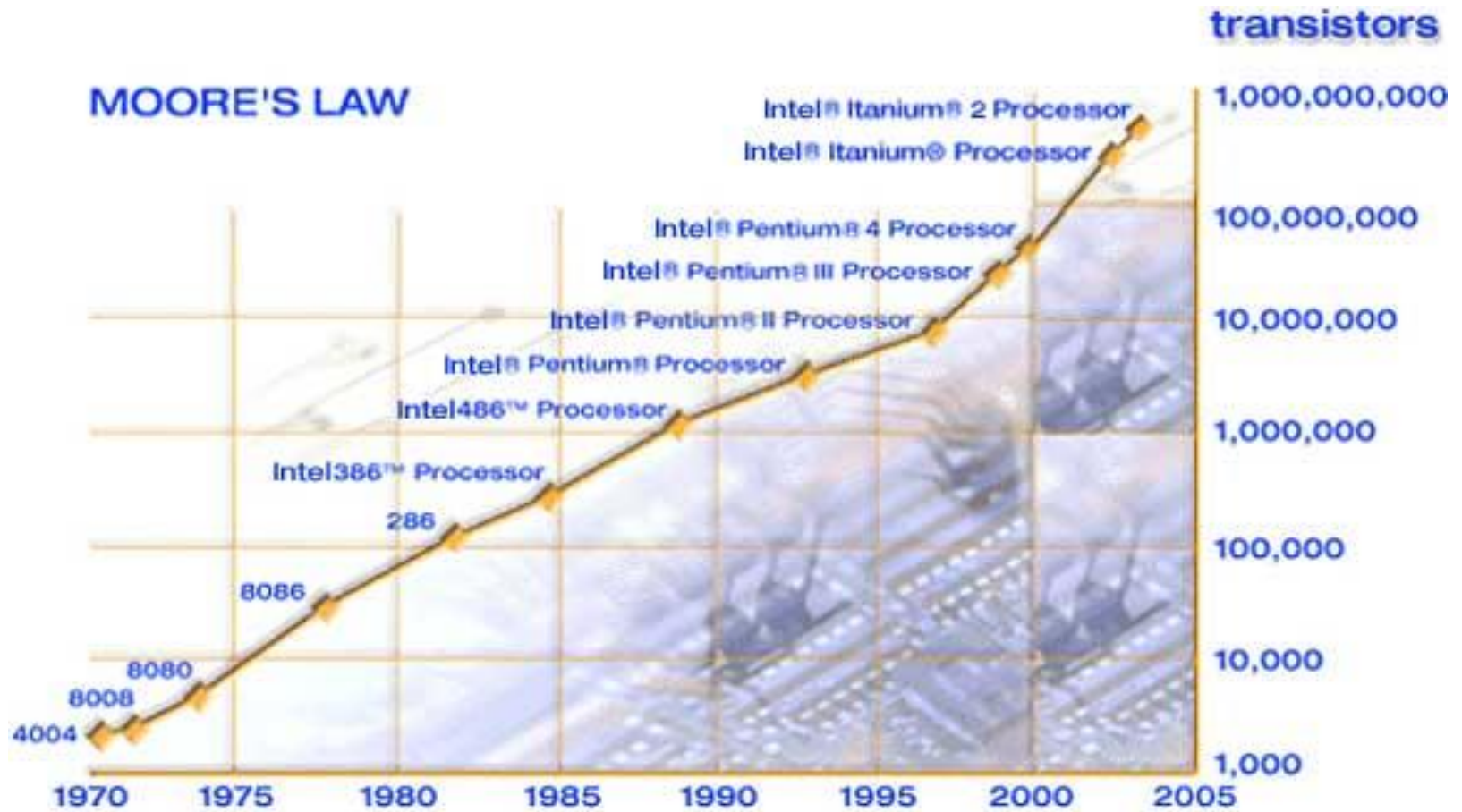
Moore, "Cramming more components onto integrated circuits," Electronics, 1965.

Microprocessor Transistor Counts 1971-2011 & Moore's Law
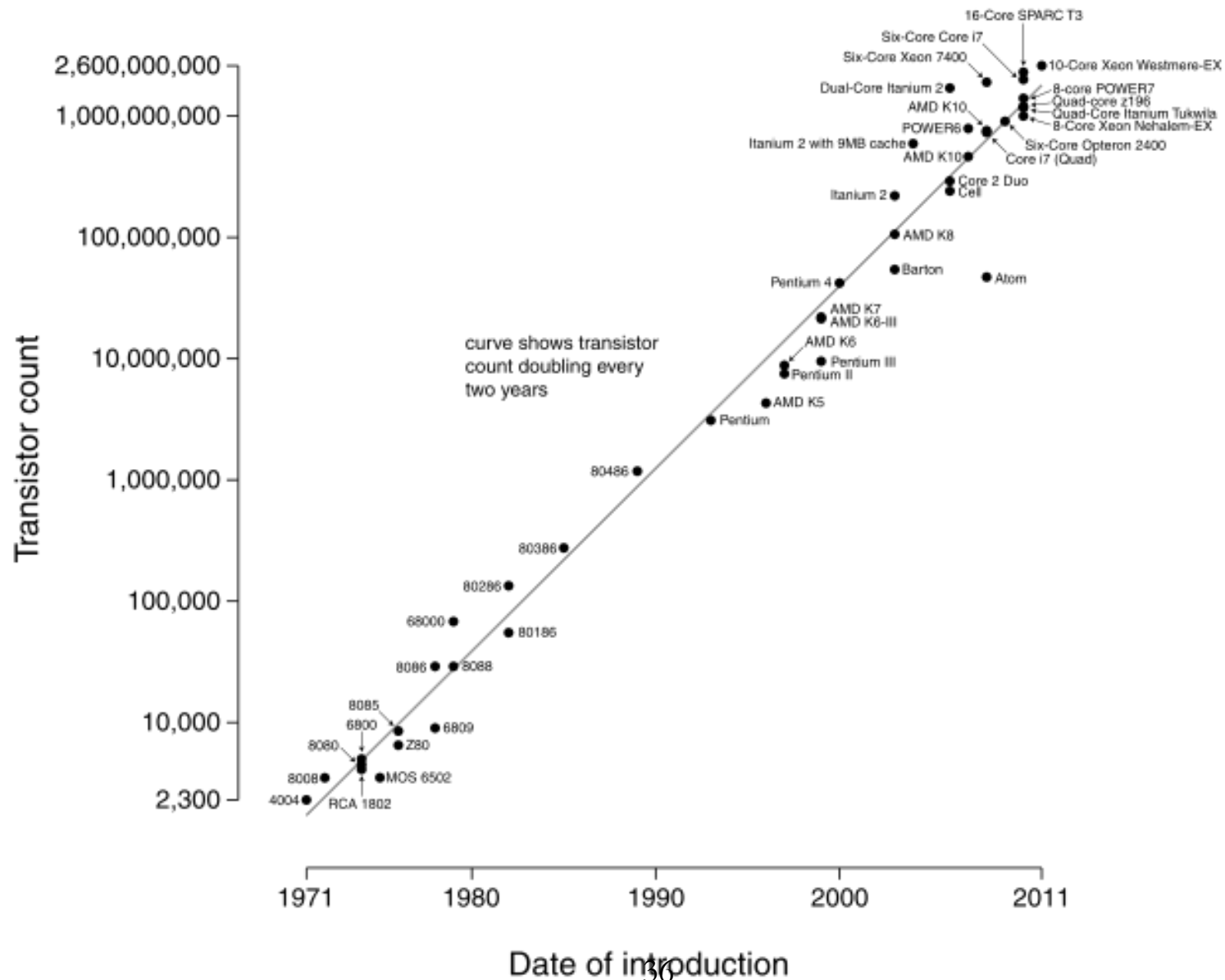
Multi-Core

- Idea: Put multiple processors on the same die

- Technology scaling (Moore's Law) enables more transistors to be placed on the same die area

- What else could you do with the die area you dedicate to multiple processors?
  - Have a bigger, more powerful core
  - Have larger caches in the memory hierarchy
  - Simultaneous multithreading
  - Integrate platform components on chip (e.g., network interface, memory controllers)
  - ...

37

Why Multi-Core?

- Alternative: Bigger, more powerful single core

  – Larger superscalar issue width, larger instruction window, more execution units, large trace caches, large branch predictors, etc

  + Improves single-thread performance transparently to programmer, compiler

Why Multi-Core?

- Alternative: Bigger, more powerful single core

  - Very difficult to design (Scalable algorithms for improving single-thread performance elusive)

  - Power hungry – many out-of-order execution structures consume significant power/area when scaled. Why?

  - Diminishing returns on performance

  - Does not significantly help memory-bound application performance (Scalable algorithms for this elusive)

Large Superscalar+OoO vs. MultiCore

- Olukotun et al., "The Case for a Single-Chip Multiprocessor," ASPLOS 1996.
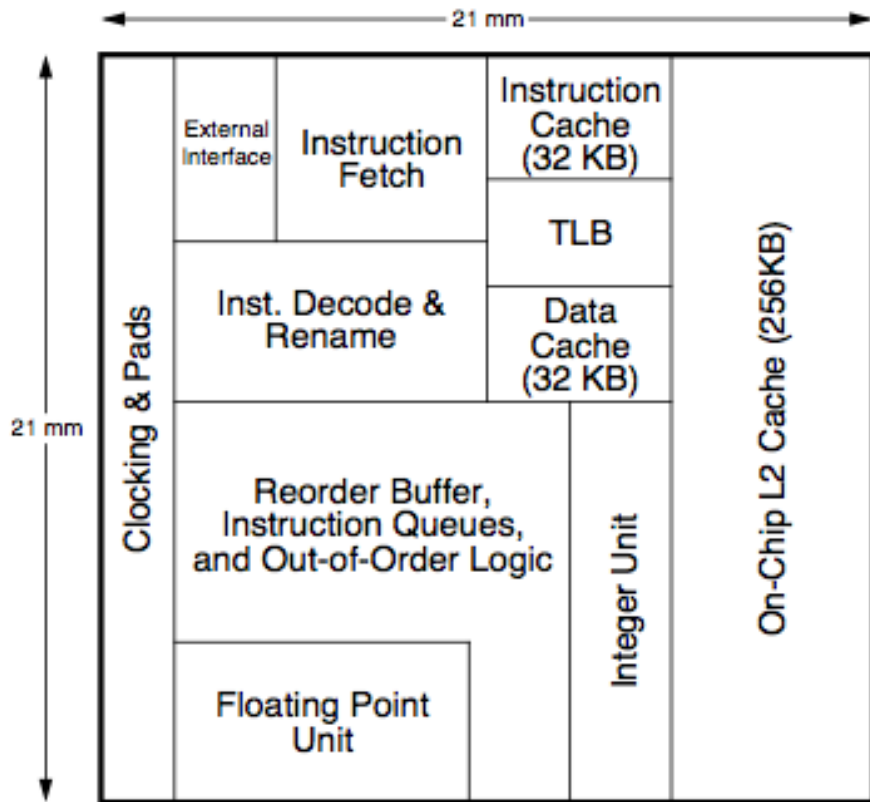
Figure 2. Floorplan for the six-issue dynamic superscalar microprocessor.
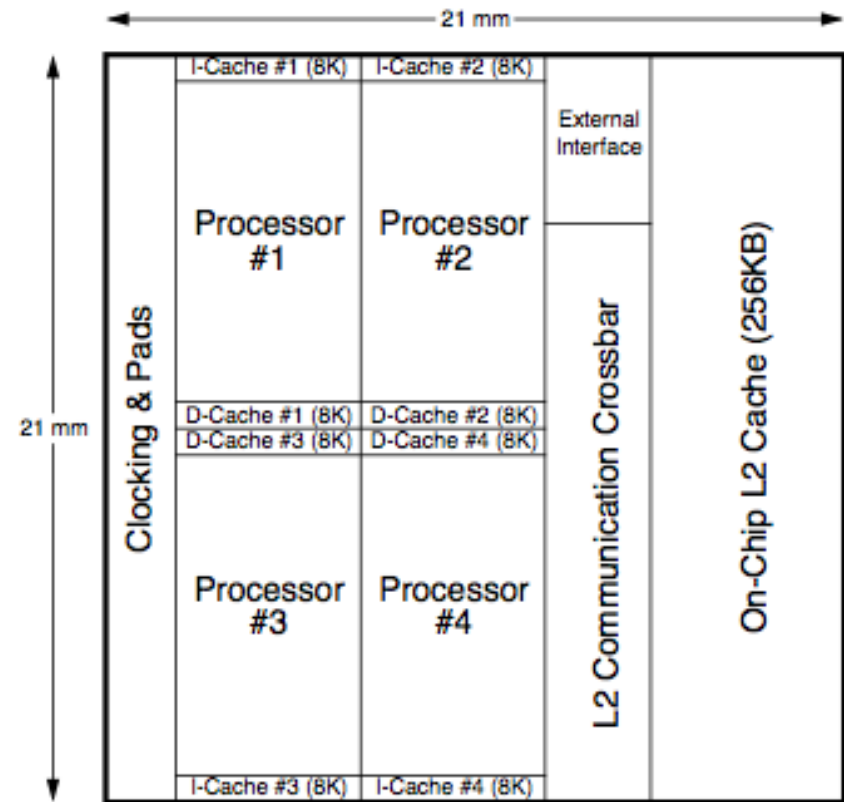
Figure 3. Floorplan for the four-way single-chip multiprocessor.

Multi-Core vs. Large Superscalar+OoO

- Multi-core advantages

  + Simpler cores → more power efficient, lower complexity, easier to design and replicate, higher frequency (shorter wires, smaller structures)

  + Higher system throughput on multiprogrammed workloads → reduced context switches

  + Higher system performance in parallel applications

Multi-Core vs. Large Superscalar+OoO

- Multi-core disadvantages
  - Requires parallel tasks/threads to improve performance (parallel programming)
  - Resource sharing can reduce single-thread performance
  - Shared hardware resources need to be managed
  - Number of pins limits data supply for increased demand

Comparison Points...

	6-way SS	4x2-way MP
# of CPUs	1	4
Degree superscalar	6	4 x 2
# of architectural registers	32int / 32fp	4 x 32int / 32fp
# of physical registers	160int / 160fp	4 x 40int / 40fp
# of integer functional units	3	4 x 1
# of floating pt. functional units	3	4 x 1
# of load/store ports	8 (one per bank)	4 x 1
BTB size	2048 entries	4 x 512 entries
Return stack size	32 entries	4 x 8 entries
Instruction issue queue size	128 entries	4 x 8 entries
I cache	32 KB, 2-way S. A.	4 x 8 KB, 2-way S. A.
D cache	32 KB, 2-way S. A.	4 x 8 KB, 2-way S. A.
L1 hit time	2 cycles (4 ns)	1 cycle (2 ns)
L1 cache interleaving	8 banks	N/A
Unified L2 cache	256 KB, 2-way S. A.	256 KB, 2-way S. A.
L2 hit time / L1 penalty	4 cycles (8 ns)	5 cycles (10 ns)
Memory latency / L2 penalty	50 cycles (100 ns)	50 cycles (100 ns)

Table 1   Key characteristics of the two microarchitectures

Why Multi-Core?

- Alternative: Bigger caches

  + Improves single-thread performance transparently to programmer, compiler

  + Simple to design

  - Diminishing single-thread performance returns from cache size. Why?

  - Multiple levels complicate memory hierarchy

Cache vs. Core
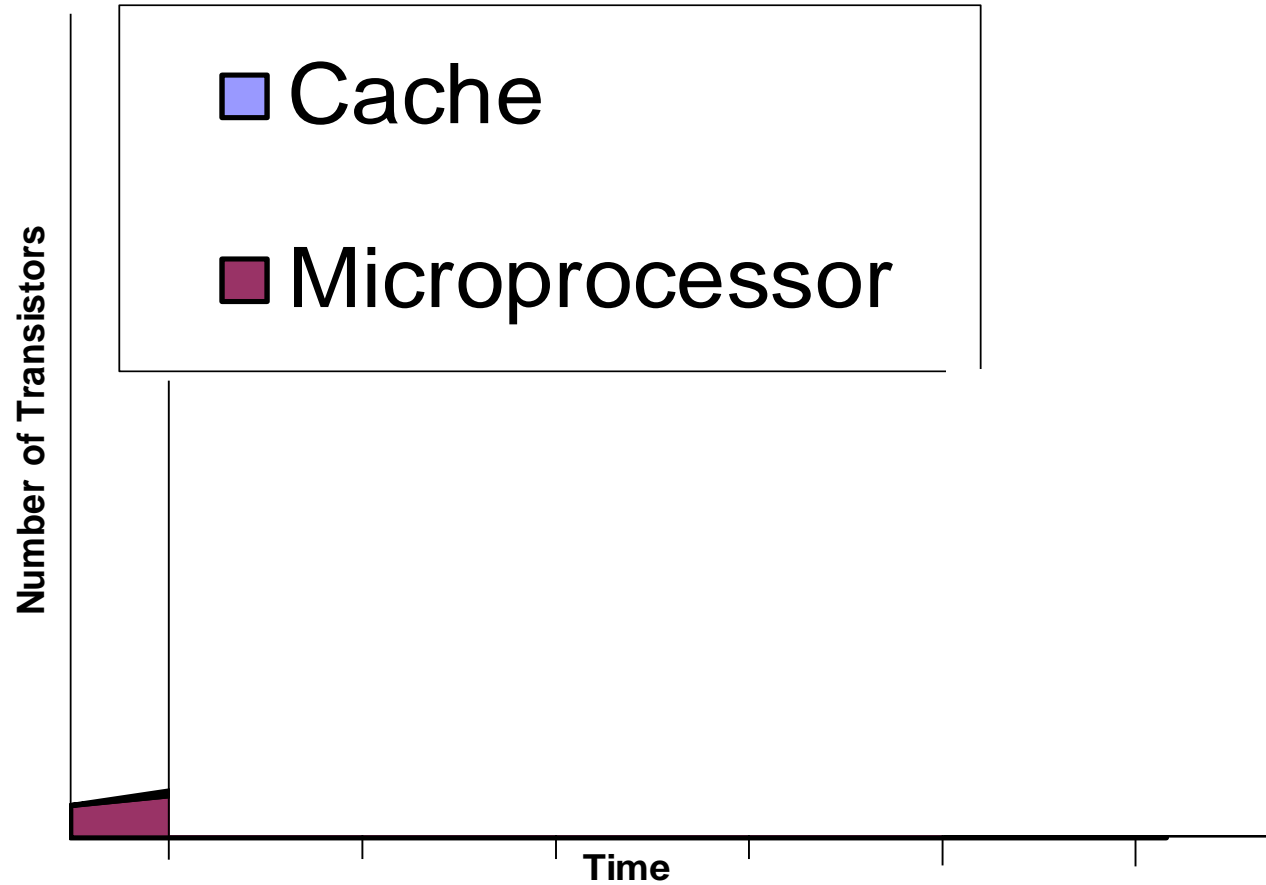
Why Multi-Core?

- Alternative: (Simultaneous) Multithreading

    + Exploits thread-level parallelism (just like multi-core)

    + Good single-thread performance with SMT

    + No need to have an entire core for another thread

    + Parallel performance aided by tight sharing of caches

Why Multi-Core?

- Alternative: (Simultaneous) Multithreading
  - Scalability is limited: need bigger register files, larger issue width (and associated costs) to have many threads → complex with many threads
  - Parallel performance limited by shared fetch bandwidth
  - Extensive resource sharing at the pipeline and memory system reduces both single-thread and parallel application performance

Why Multi-Core?

- Alternative: Integrate platform components on chip instead

  + Speeds up many system functions (e.g., network interface cards, Ethernet controller, memory controller, I/O controller)

  - Not all applications benefit (e.g., CPU intensive code sections)

Why Multi-Core?

- Alternative: Traditional symmetric multiprocessors

  + Smaller die size (for the same processing core)

  + More memory bandwidth (no pin bottleneck)

  + Fewer shared resources → less contention between threads

Why Multi-Core?

- Alternative: Traditional symmetric multiprocessors

  - Long latencies between cores (need to go off chip) → shared data accesses limit performance → parallel application scalability is limited

  - Worse resource efficiency due to less sharing → worse power/energy efficiency

Why Multi-Core?

- Other alternatives?
  - Clustering?
  - Dataflow? EDGE?
  - Vector processors (SIMD)?
  - Integrating DRAM on chip?
  - Reconfigurable logic? (general purpose?)

Review next week

- "[Exploiting ILP, TLP, and DLP with the polymorphous TRIPS architecture](#)", K. Sankaralingam, ISCA 2003.

Summary: Multi-Core Alternatives

- Bigger, more powerful single core
- Bigger caches
- (Simultaneous) multithreading
- Integrate platform components on chip instead
- More scalable superscalar, out-of-order engines
- Traditional symmetric multiprocessors
- And more!

Multicore Examples

Multiple Cores on Chip

- Simpler and lower power than a single large core

- Large scale parallelism on chip
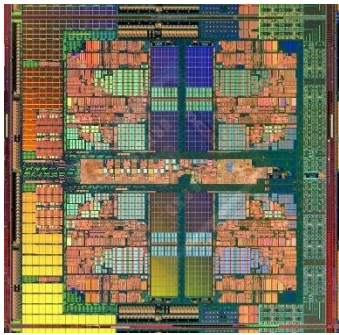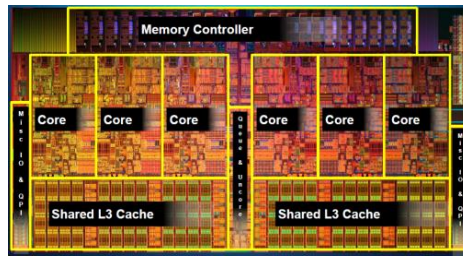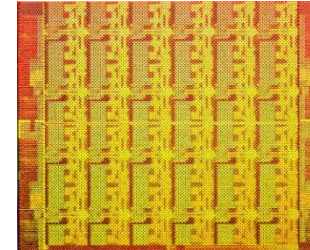

Intel Core i7
8 cores
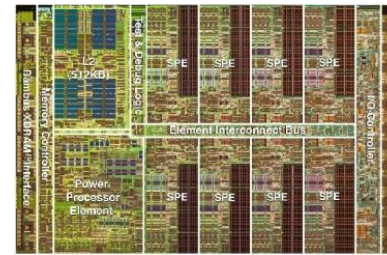
IBM Cell BE
8+1 cores

IBM POWER7
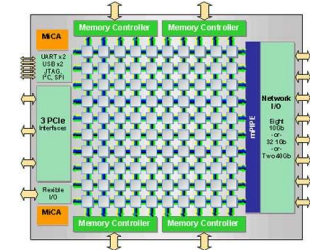8 cores
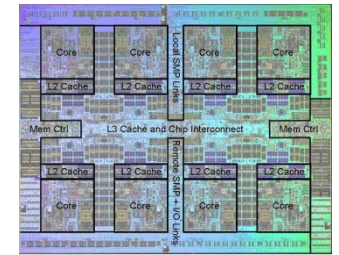
AMD Barcelona
4 cores
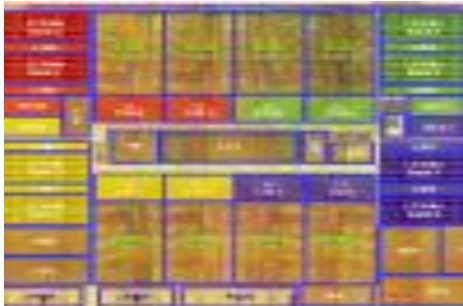
Sun Niagara II
8 cores
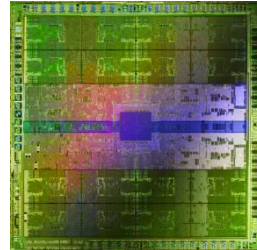
Nvidia Fermi
448 "cores"

Intel SCC
48 cores, networked

Tilera TILE Gx
100 cores, networked

With Multiple Cores on Chip

- What we want:

  - N times the performance with N times the cores when we parallelize an application on N cores

- What we get:

  - Amdahl's Law (serial bottleneck)
  - Bottlenecks in the parallel portion

The Problem: Serialized Code Sections

- Many parallel programs cannot be parallelized completely

- Causes of serialized code sections
  - Sequential portions (Amdahl's "serial part")
  - Critical sections
  - Barriers
  - Limiter stages in pipelined programs

- Serialized code sections
  - Reduce performance
  - Limit scalability
  - Waste energy

Demands in Different Code Sections

- What we want:
- In a serialized code section → one powerful "large" core
- In a parallel code section → many wimpy "small" cores
- These two conflict with each other:
  - If you have a single powerful core, you cannot have many cores
  - A small core is much more energy and area efficient than a large core

"Large" vs. "Small" Cores

Large Core	Small Core

Large Core

- *Out-of-order*
- *Wide fetch e.g. 4-wide*
- *Deeper pipeline*
- *Aggressive branch predictor (e.g. hybrid)*
- *Multiple functional units*
- *Trace cache*
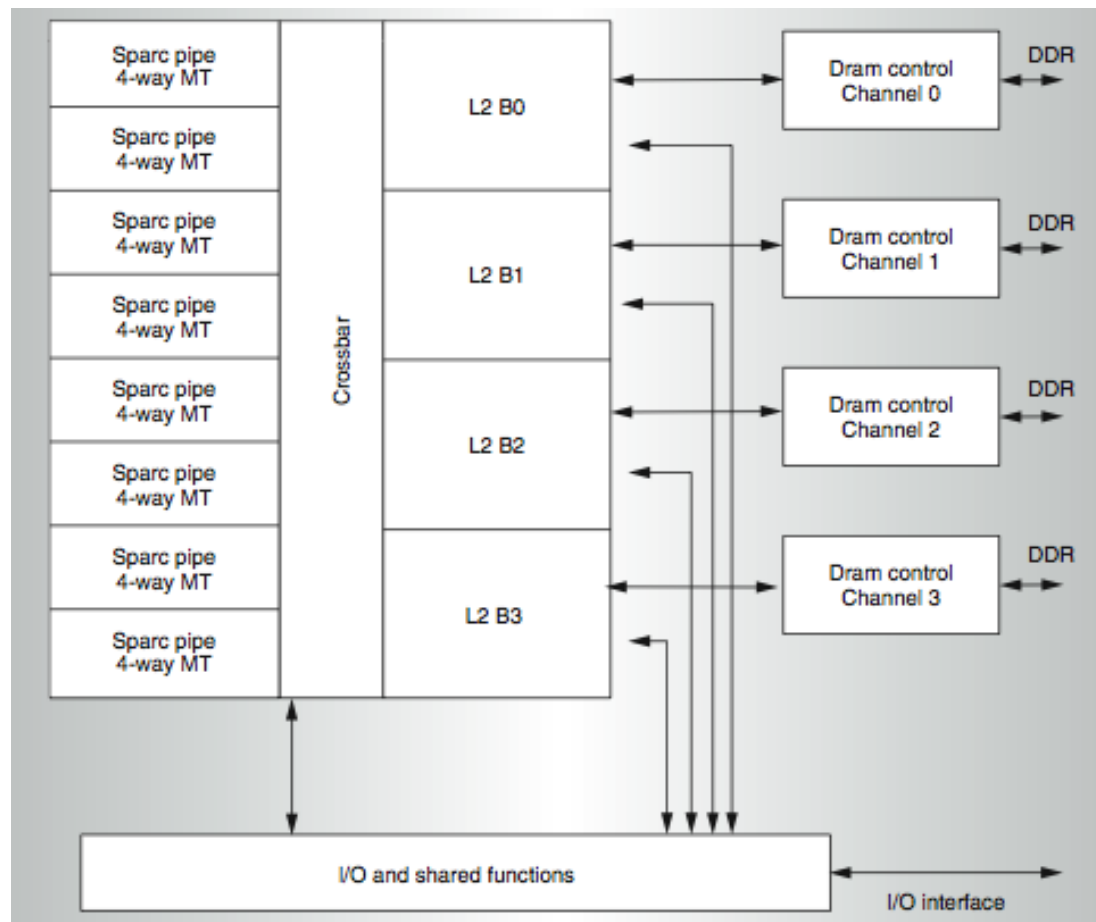- *Memory dependence speculation*

Small Core

- *In-order*
- *Narrow Fetch e.g. 2-wide*
- *Shallow pipeline*
- *Simple branch predictor (e.g. Gshare)*
- *Few functional units*

Large Cores are power inefficient:
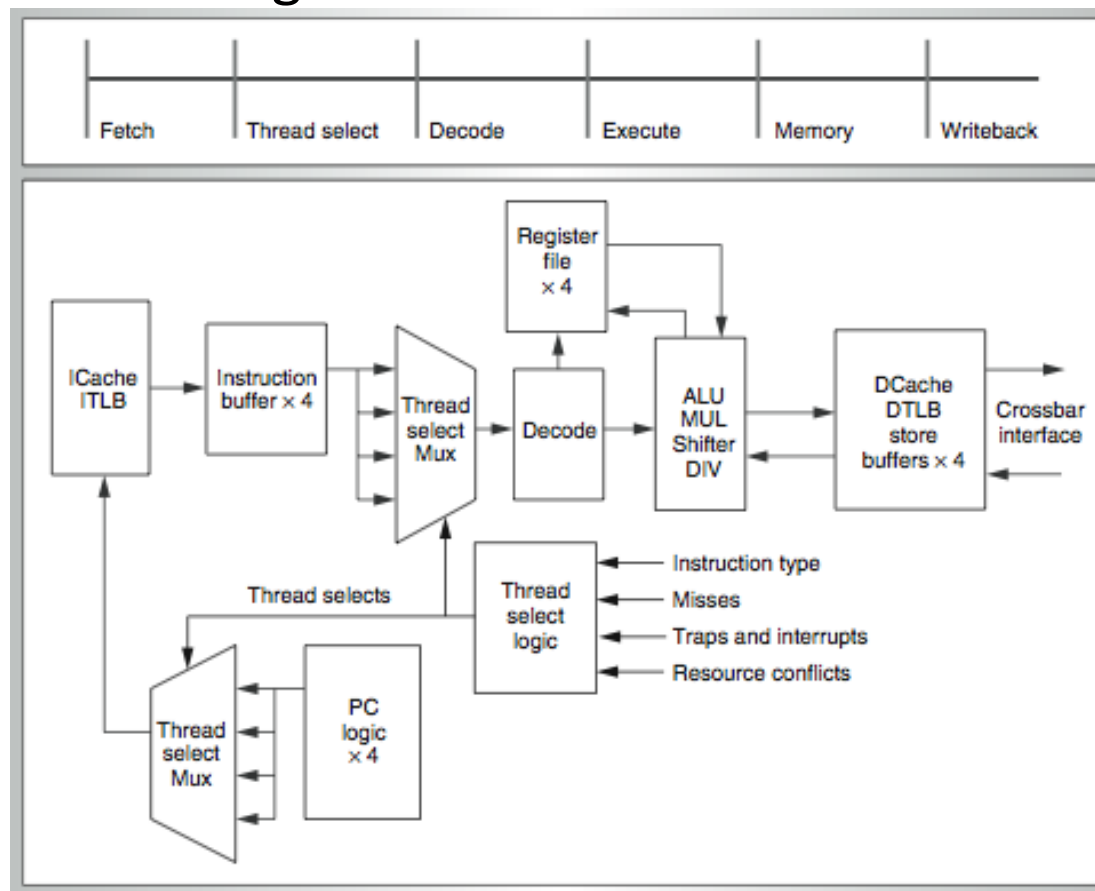e.g., 2x performance for 4x area (power)

Meet Small: Sun Niagara (UltraSPARC T1)

- Kongetira et al., "Niagara: A 32-Way Multithreaded SPARC Processor," IEEE Micro 2005.
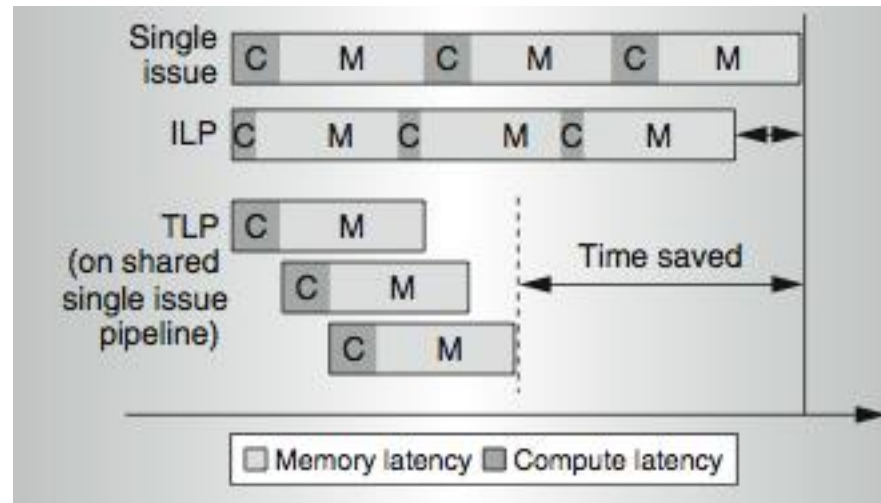
Niagara Core

- 4-way fine-grain multithreaded, 6-stage, dual-issue in-order
- Round robin thread selection (unless cache miss)
- Shared FP unit among cores

Niagara Design Point

Table 1. Commercial server applications.

| Benchmark	Application category	Instruction-level parallelism	Thread-level parallelism	Working set	Data sharing
Web99	Web server	Low	High	Large	Low
JBB	Java application server	Low	High	Large	Medium
TPC-C	Transaction processing	Low	High	Large	High
SAP-2T	Enterprise resource planning	Medium	High	Medium	Medium
SAP-3T	Enterprise resource planning	Low	High	Large	High
TPC-H	Decision support system	High	High	Large	Medium

Meet Small: Sun Niagara II (UltraSPARC T2)
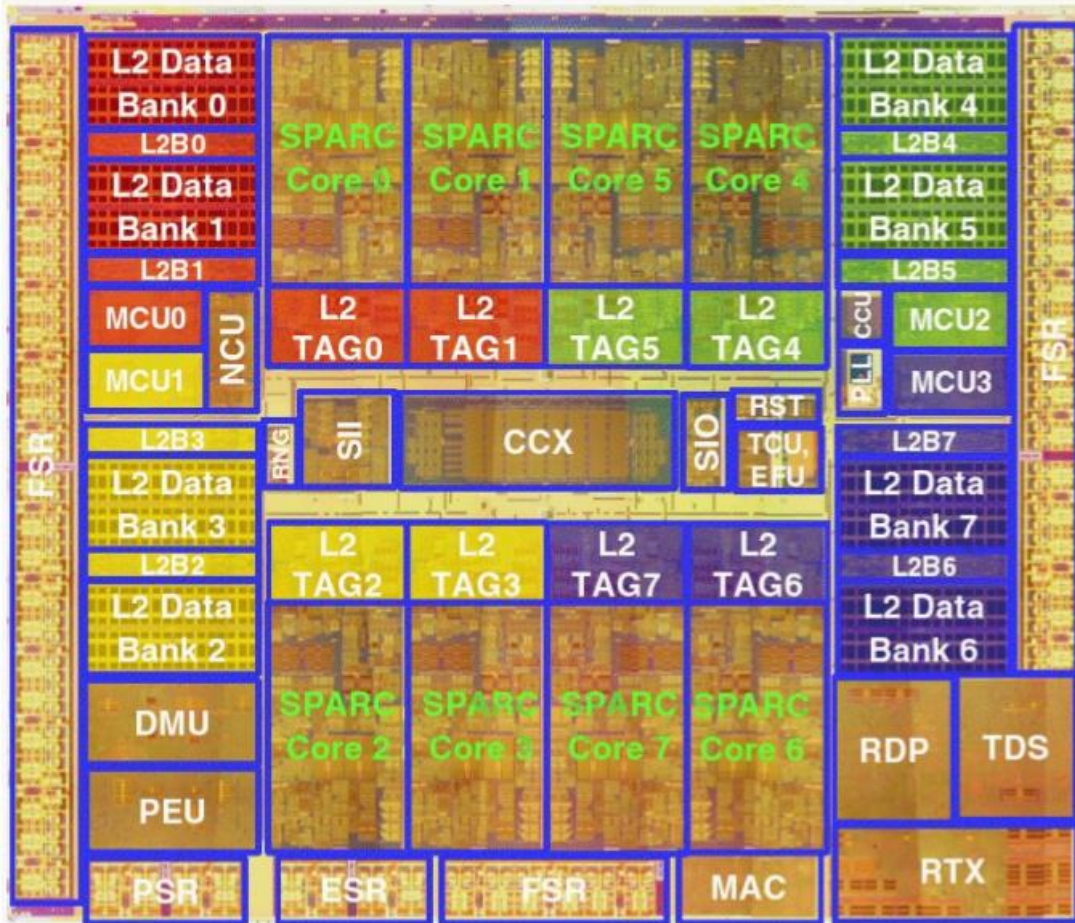
- 8 SPARC cores, 8 threads/core. 8 stages. 16 KB I$ per Core. 8 KB D$ per Core. FP, Graphics, Crypto, units per Core.

- 4 MB Shared L2, 8 banks, 16-way set associative.

- 4 dual-channel FBDIMM memory controllers.

- X8 PCI-Express @ 2.5 Gb/s.

- Two 10G Ethernet ports @ 3.125 Gb/s.

Meet Small, but Larger: Sun ROCK

- Chaudhry et al., "Simultaneous Speculative Threading: A Novel Pipeline Architecture Implemented in Sun's ROCK Processor," ISCA 2009

- Goals:
  - Maximize throughput when threads are available
  - Boost single-thread performance when threads are not available and on cache misses

- Ideas:

  - Runahead on a cache miss → ahead thread executes miss-independent instructions, behind thread executes dependent instructions

  - Branch prediction (gshare)

Sun ROCK
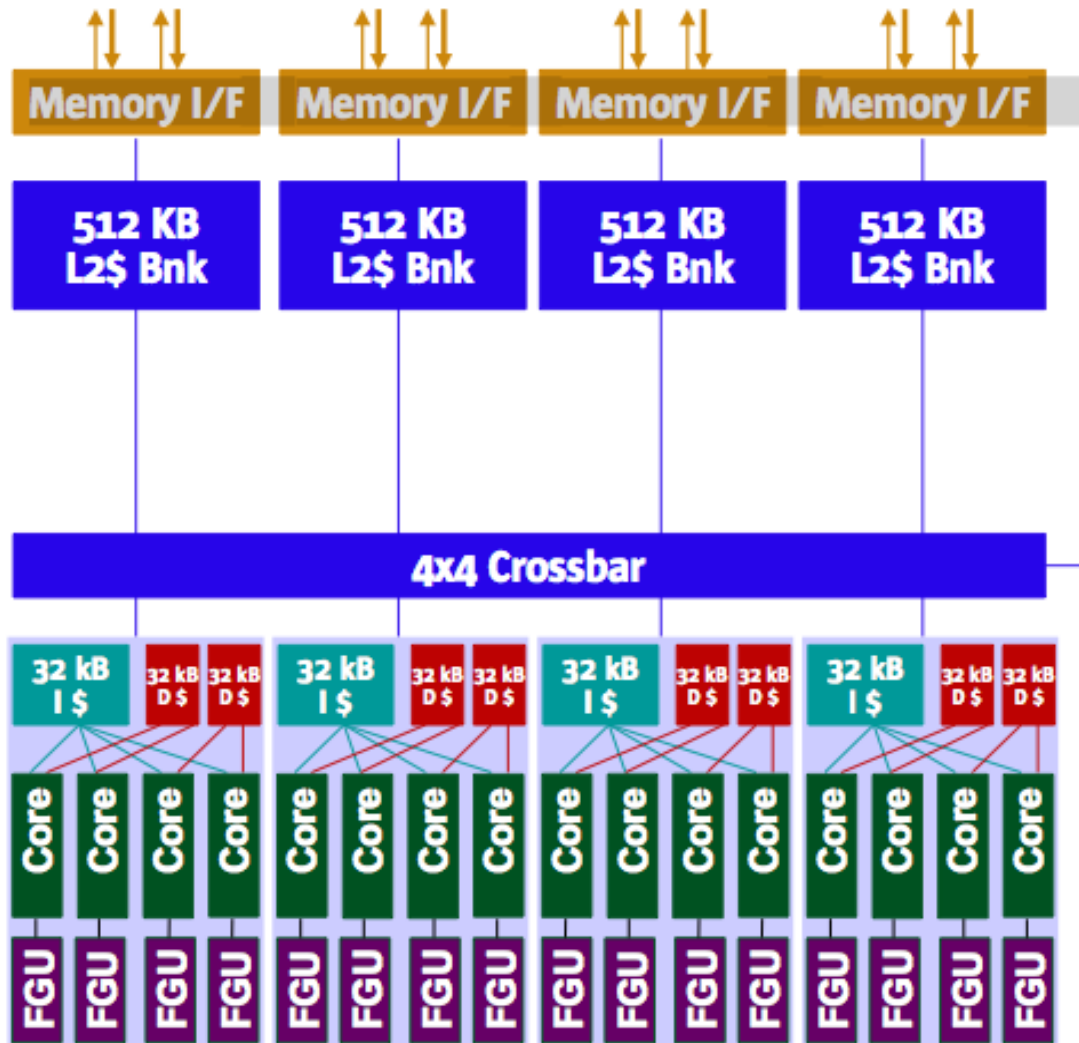
- 16 cores, 2 threads per core (fewer threads than Niagara 2)

- 4 cores share a 32KB instruction cache

- 2 cores share a 32KB data cache

- 2MB L2 cache (smaller than Niagara 2)
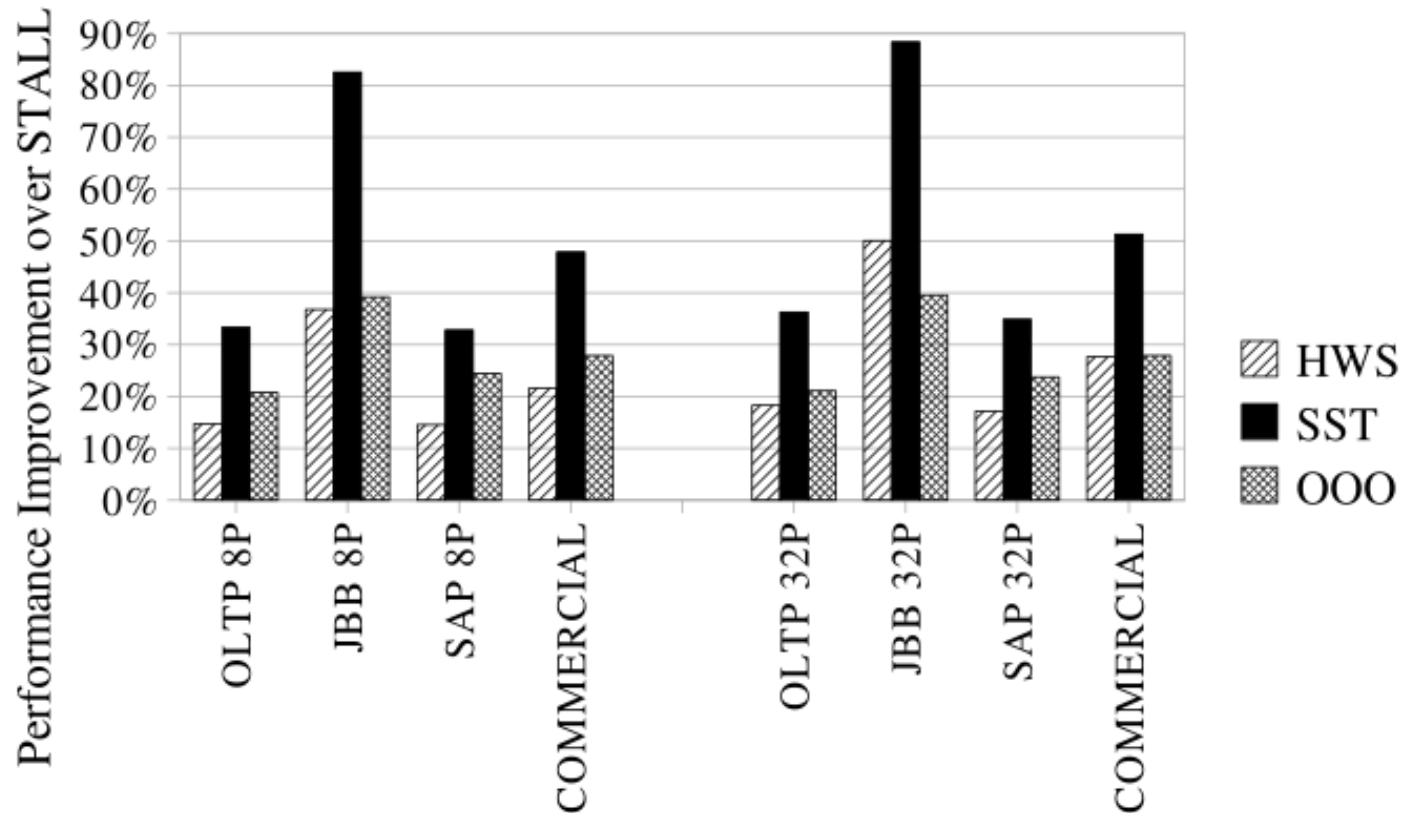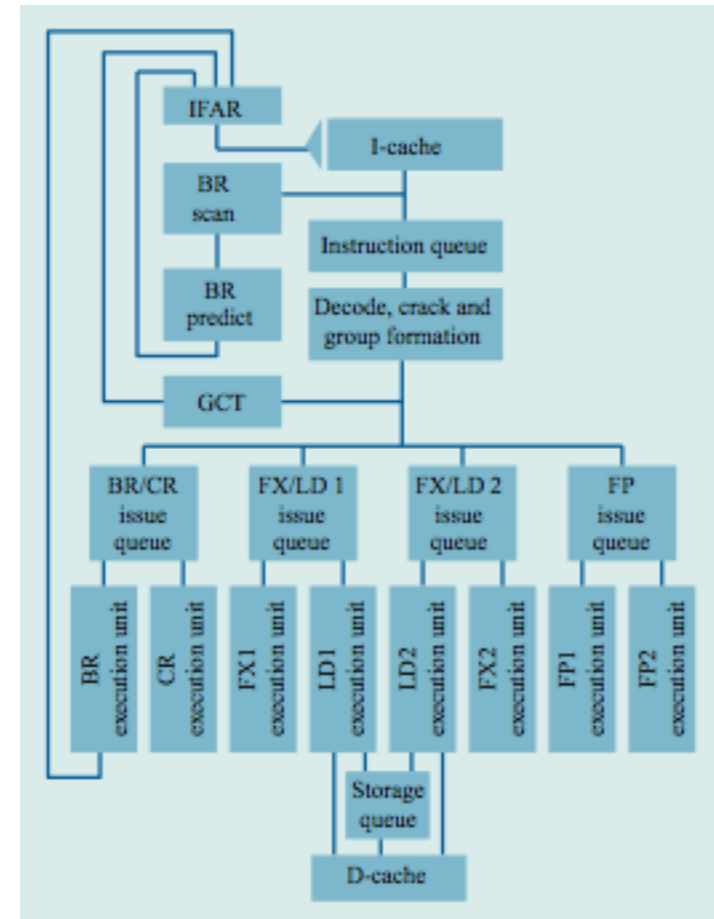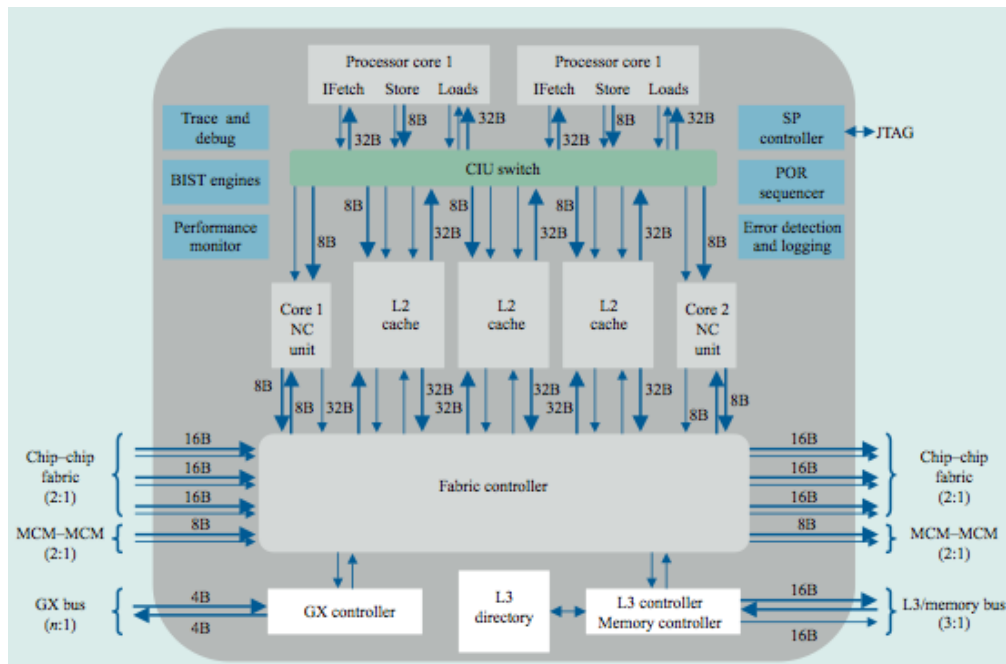
More Powerful Cores in Sun ROCK

Figure 9: Commercial Performance.

Meet Large: IBM POWER4

- Tendler et al., "POWER4 system microarchitecture," IBM J R&D, 2002.

- Another symmetric multi-core chip...
- But, fewer and more powerful cores

IBM POWER4

- 2 cores, out-of-order execution
- 100-entry instruction window in each core
- 8-wide instruction fetch, issue, execute
- Large, local+global hybrid branch predictor
- 1.5MB, 8-way L2 cache
- Aggressive stream based prefetching

IBM POWER5

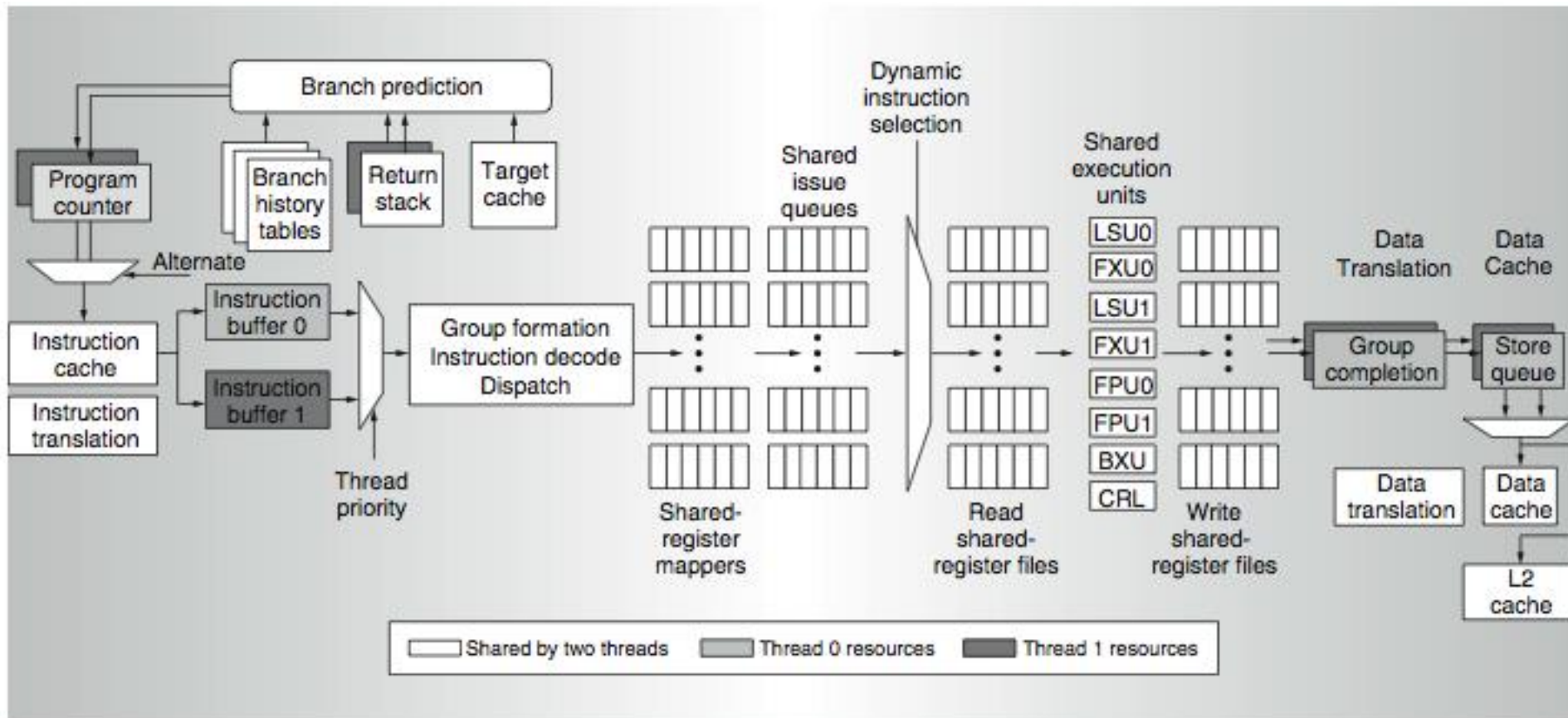- Kalla et al., "IBM Power5 Chip: A Dual-Core Multithreaded Processor," IEEE Micro 2004.

Figure 4. Power5 instruction data flow (BXU = branch execution unit and CRL = condition register logical execution unit).

Large, but Smaller: IBM POWER6

- Le et al., "IBM POWER6 microarchitecture," IBM J R&D, 2007.

- 2 cores, in order, high frequency (4.7 GHz)
- 8 wide fetch
- Simultaneous multithreading in each core
- Runahead execution in each core
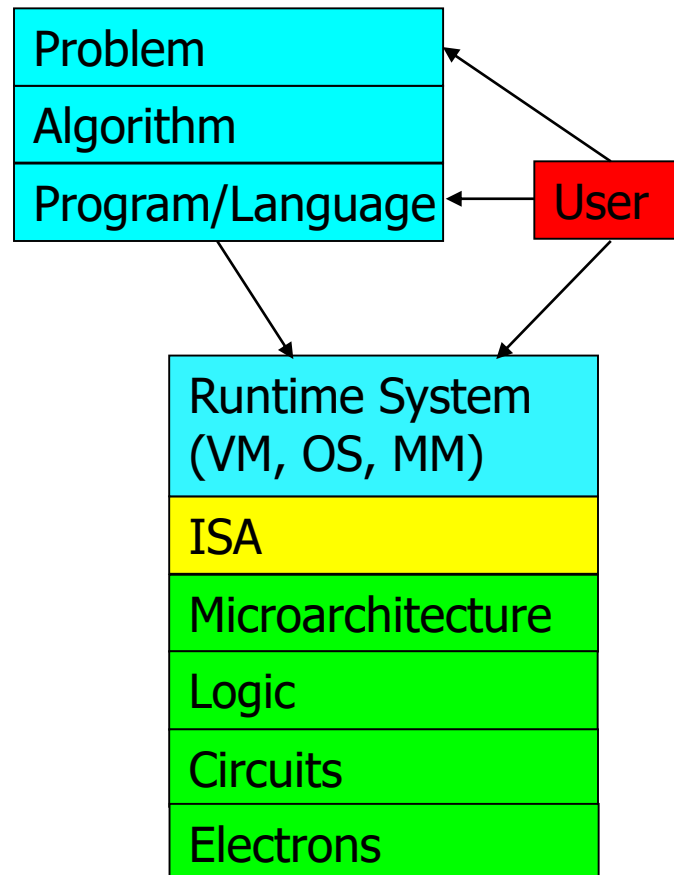 - Similar to Sun ROCK
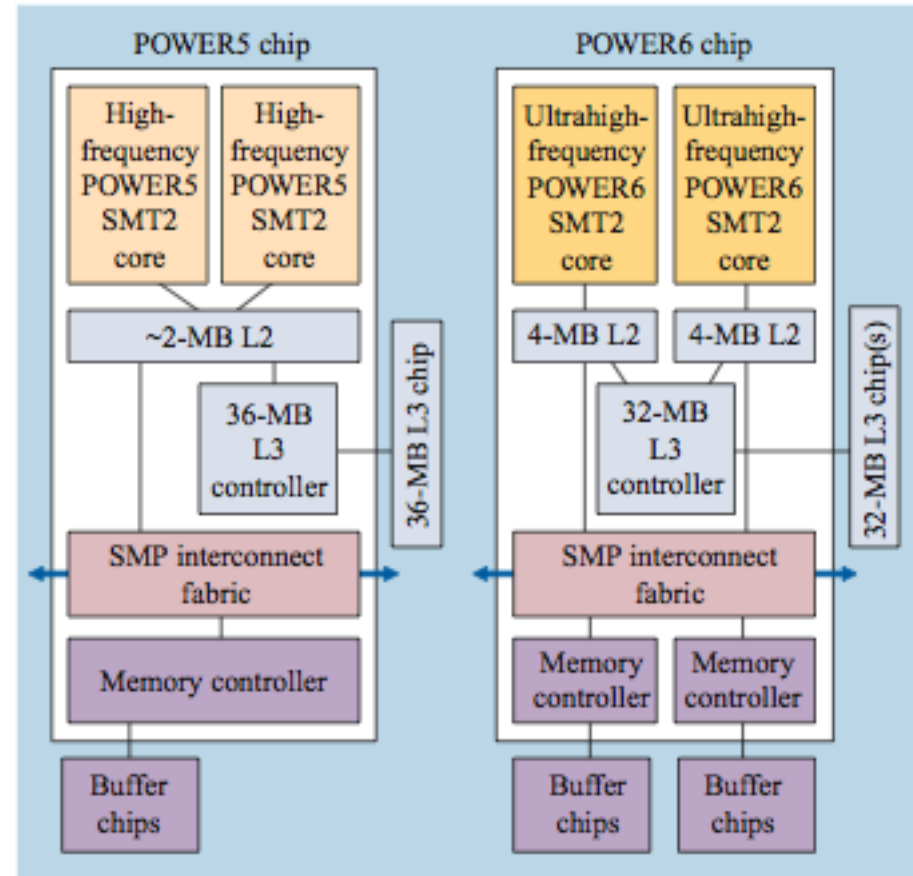
Many More…

- Wimpy nodes: Tilera

- Asymmetric multicores

- DVFS

Computer Architecture Today

- Today is a very exciting time to study computer architecture

- Industry is in a large paradigm shift (to multi-core, hardware acceleration and beyond) – many different potential system designs possible

- Many difficult problems *caused by* the shift
  - Power/energy constraints → multi-core?, accelerators?
  - Complexity of design → multi-core?
  - Difficulties in technology scaling → new technologies?
  - Memory wall/gap
  - Reliability wall/issues
  - Programmability wall/problem → single-core?

Computer Architecture Today (2)

- These problems affect all parts of the computing stack – if we do not change the way we design systems

| Problem |
| Algorithm |
| Program/Language |

| User |

| Runtime System (VM, OS, MM) |
| ISA |
| Microarchitecture |
| Logic |
| Circuits |
| Electrons |

Computer Architecture Today (3)

- You can revolutionize the way computers are built, if you understand both the hardware and the software

- You can invent new paradigms for computation, communication, and storage

- Recommended book: Kuhn, "The Structure of Scientific Revolutions" (1962)
  - Pre-paradigm science: no clear consensus in the field
  - Normal science: dominant theory used to explain things (business as usual); exceptions considered anomalies
  - Revolutionary science: underlying assumptions re-examined

… but, first …

- Let's understand the fundamentals…

- You can change the world only if you understand it well enough…
  - Especially the past and present dominant paradigms
  - And, their advantages and shortcomings -- tradeoffs

CSC 2224: Parallel Computer Architecture and Programming Parallel Processing, Multicores
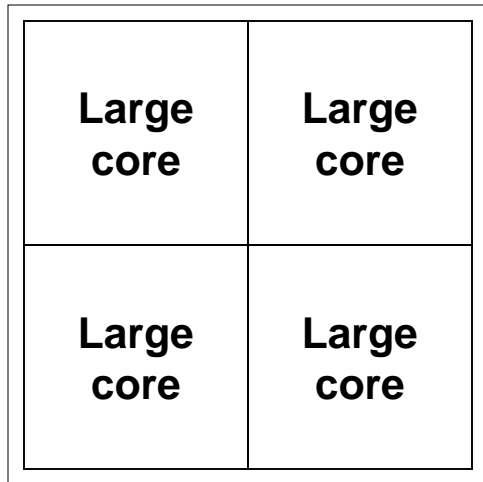
Prof. Gennady Pekhimenko
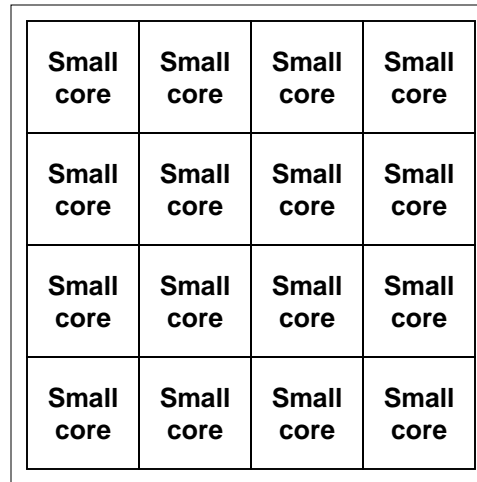
University of Toronto

Fall 2018

The content of this lecture is adapted from the lectures of Onur Mutlu @ CMU
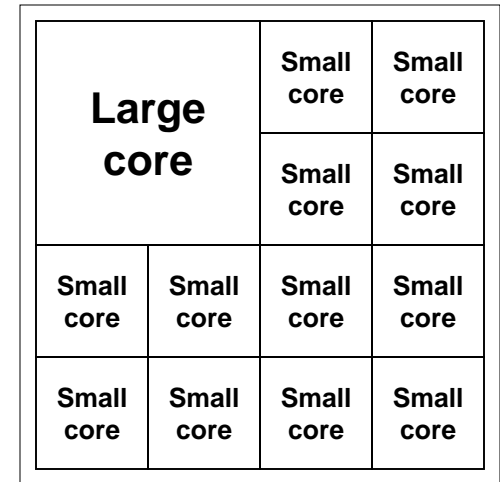
Asymmetric Multi-Core

Asymmetric Chip Multiprocessor (ACMP)

Large core	Large core
Large core	Large core

"Tile-Large"

Small core	Small core	Small core	Small core
Small core	Small core	Small core	Small core
Small core	Small core	Small core	Small core
Small core	Small core	Small core	Small core

"Tile-Small"

ACMP

- Provide one large core and many small cores

+ Accelerate serial part using the large core (2 units)

+ Execute parallel part on small cores and large core for high throughput (12+2 units)