Multicore & Interconnection Networks

15-740 SPRING'18

NATHAN BECKMANN (SLIDES BASED ON ONUR MULTLU'S)

Topics

Overview of multicore

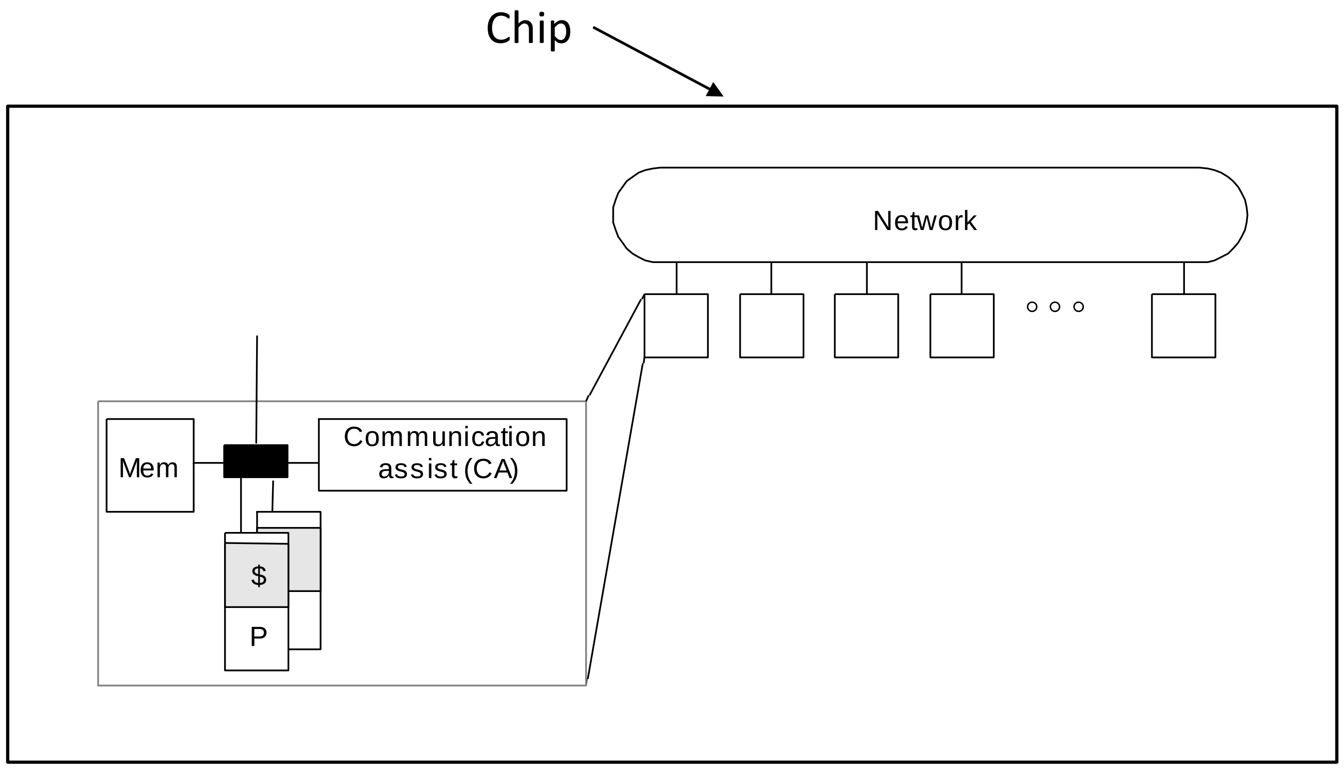
Why multicore?

Interconnection networks

What is multicore?

Technology let’s us put build a parallel architecture on a single chip

Recall from last time: “General parallel architecture” (for SAS/MP)



Example: IBM POWER4 (2001)

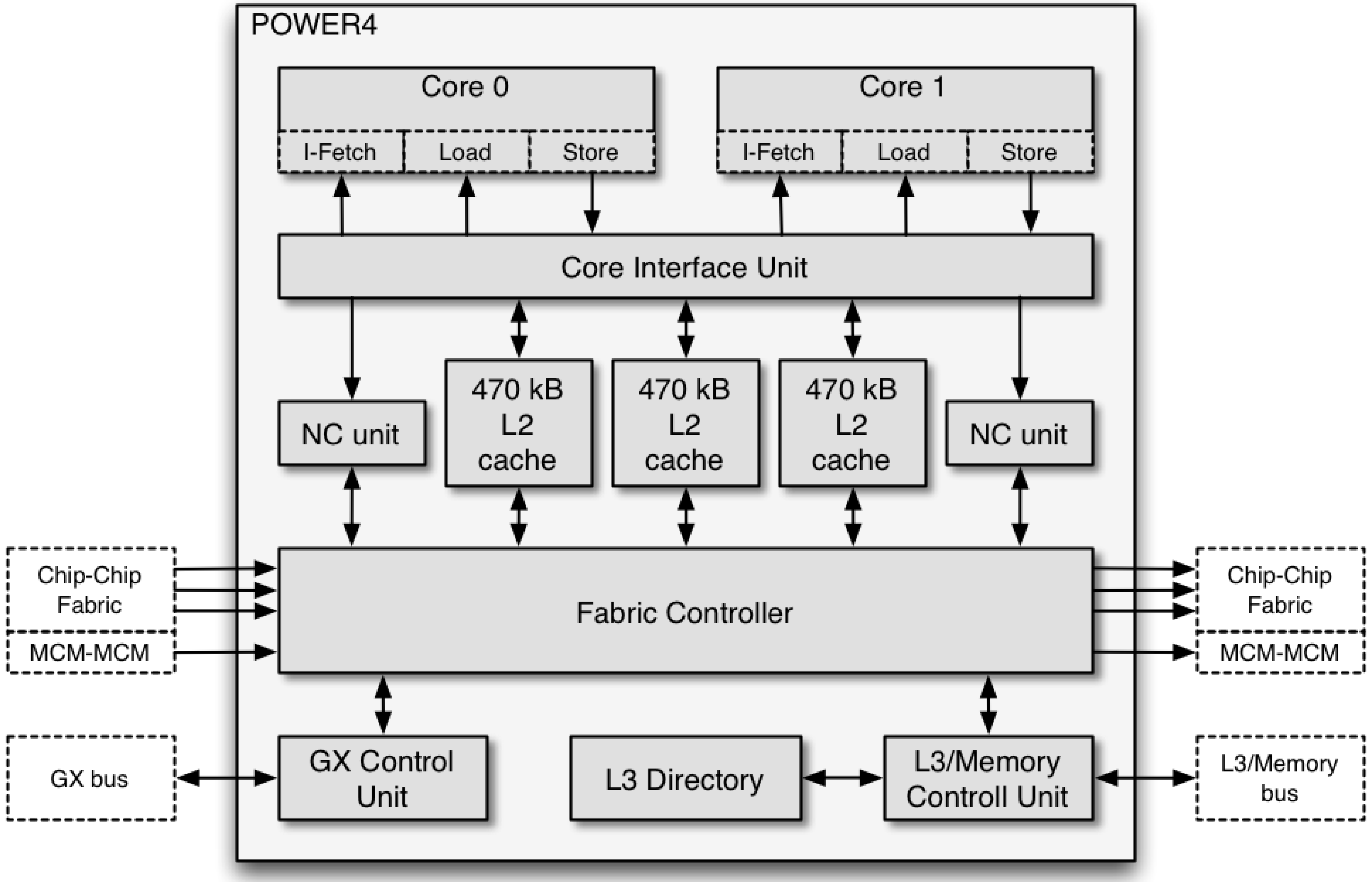
174M transistors @ 180nm

2 cores
- 8-wide superscalar
- Out-of-order
- 1.3 GHz

L2 is banked x3

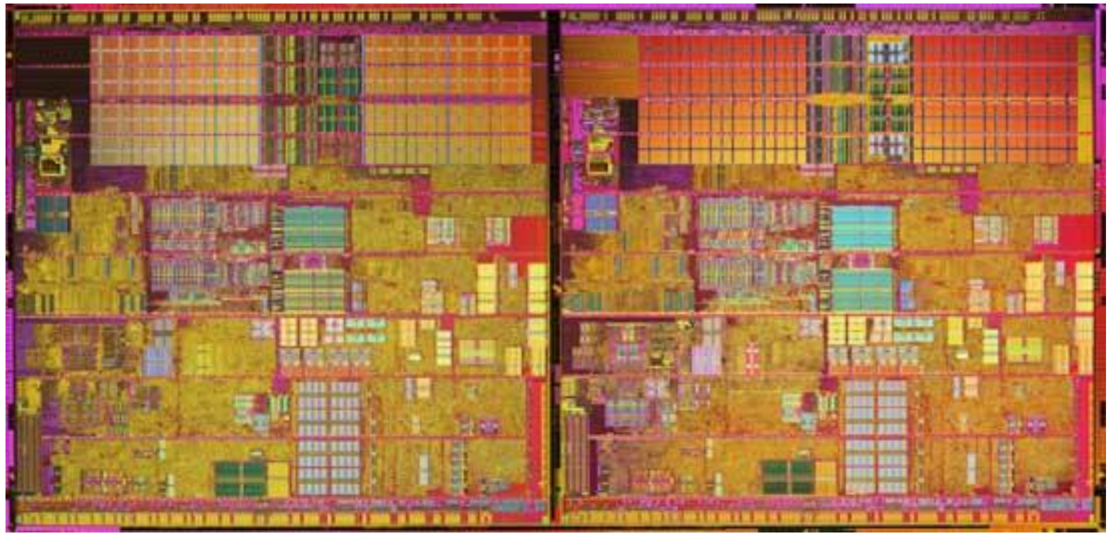
L3 control on-chip, memory off-chip

Multi-chip module (MCM) config
- 4 dies (8 cores) in single package
- Shared bus “fabric”
- 128MB combined L3



Example: Intel Pentium D (2005)

230M transistors @ 90nm



Core

- High frequency & low ILP
- 3.8 GHz (designed for 10 GHz!!!)
- 31-stage pipeline
- 3-wide superscalar
- Out-of-order
- Trace cache

Multi-chip module (MCM)

- 2 dies in single package

2MB L2 cache

130 Watts!

Example: Intel Core Duo (2006)

Intel forced to use mobile designs derived from Pentium 3

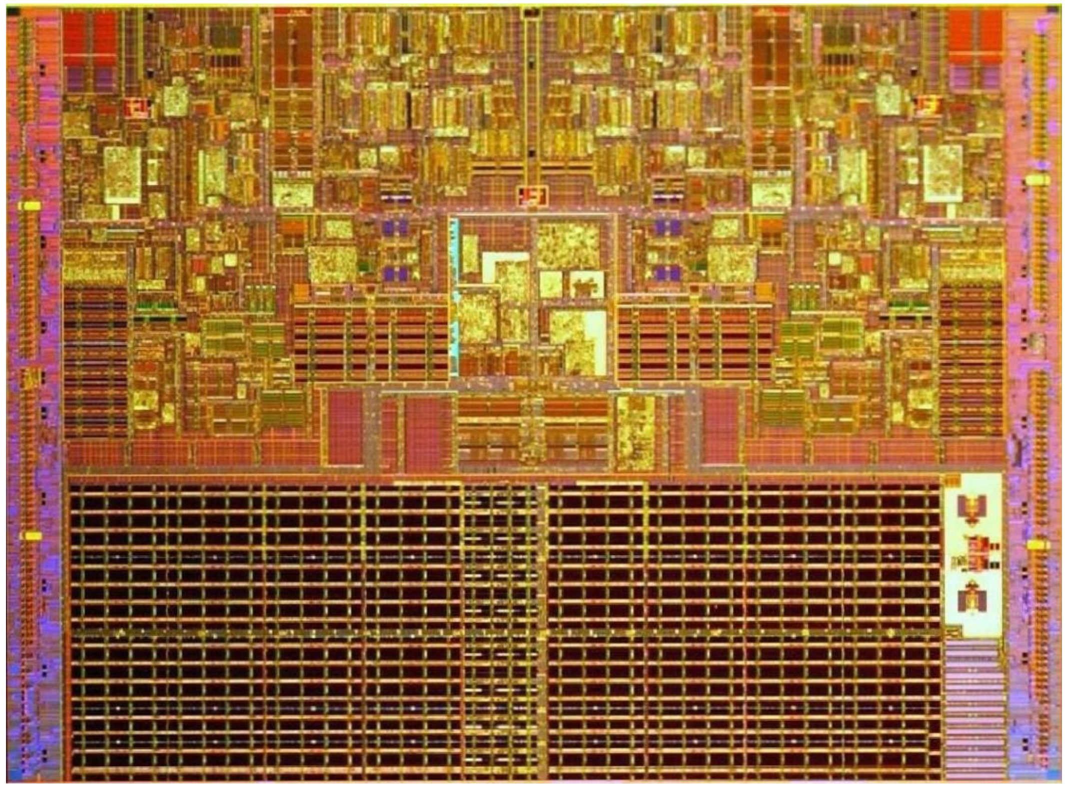
151M transistors @ 65nm

Core
- 2.33 GHz
- 4-wide issue
- 12-stage pipeline
- Out-of-order

2 cores on single die

2MB L2 shared cache

31 Watts



Example: Sun UltraSPARC T1 (2005)

279M transistors @ 90nm

- 378mm^2 (!!)

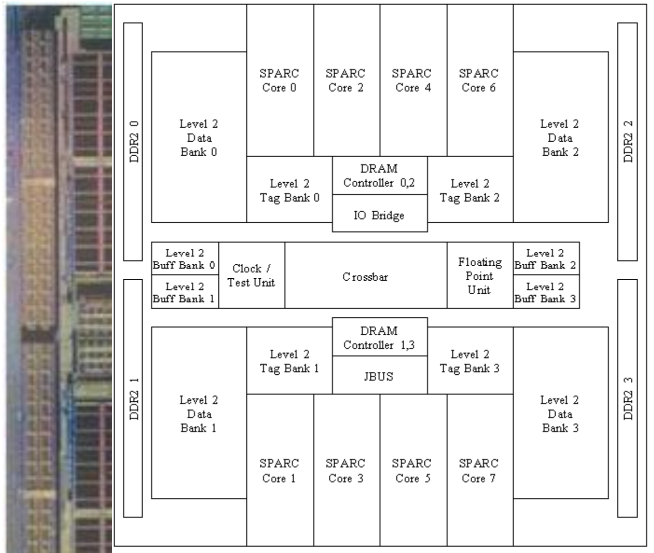
Multicore, multithreaded processor

- 8 cores × 4 threads = 32 threads total
- Maximize parallelism, sacrifice sequential perf.

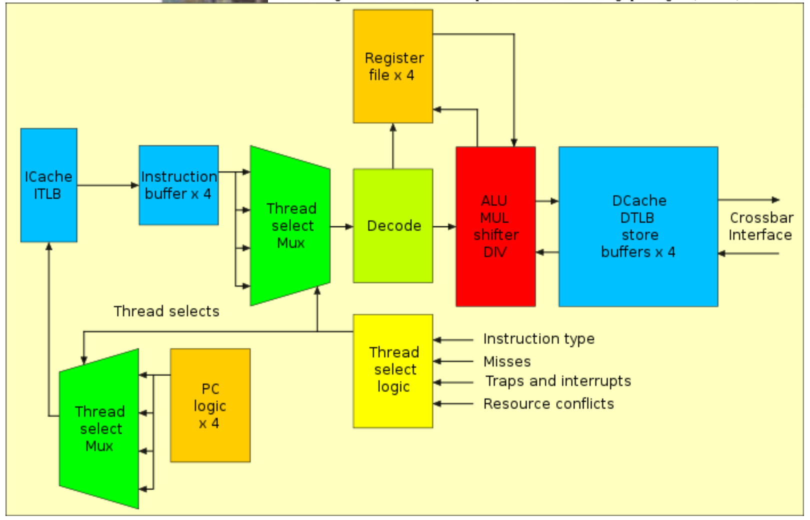
Core

- 1.4 GHz
- Fine-grain multithreading
- In-order, simple 6-stage pipeline

74 Watts



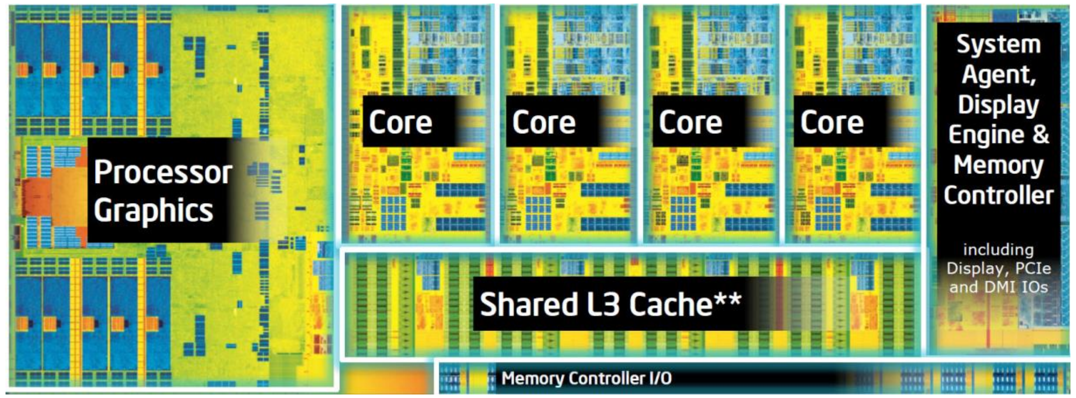
Niagra 1 / UltraSPARC T1 / OpenSPARC T1 - Die Micrograph Diagram (davidhalko)



Example: Intel Core i7 (2013)

1.4B transistors @ 22nm
- 177 mm^2

Core
- 3.5 GHz to 3.9 GHz
- 14-stage pipelined datapath
- 4-wide superscalar
- 3 levels of large cache


Processor Graphics
Core
Core
Core
Core
System Agent, Display Engine & Memory Controller
including Display, PCIe and DMI IOs
Shared L3 Cache**
Memory Controller I/O

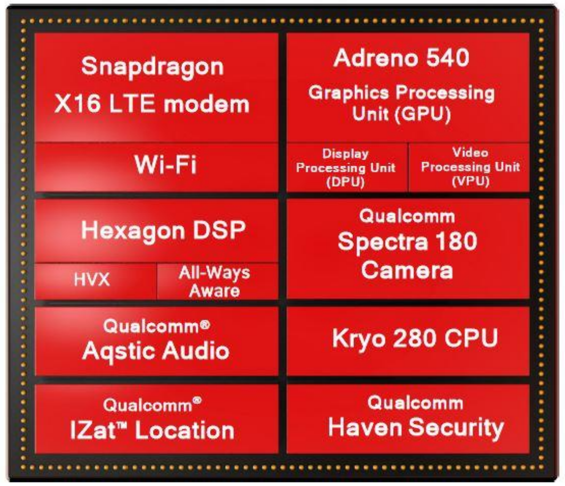
Four cores in single die

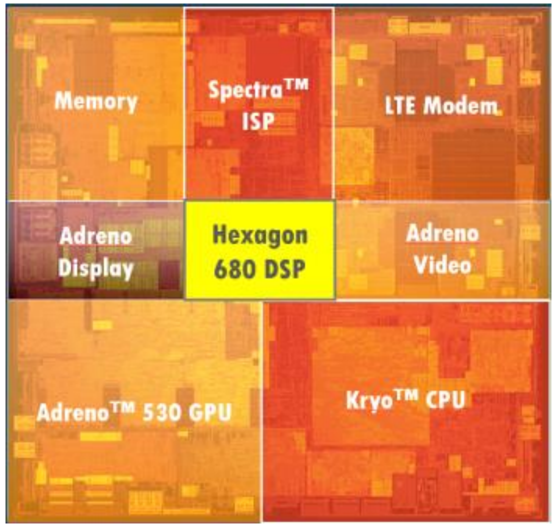
Example: Qualcomm Snapdragon 835 (2017)

?? Transistors @ 10nm

ARM cores – heterogeneous "big.LITTLE" design
- 4 "performance" cores – 2.45 GHz, 2MB L2 cache
- 4 "efficiency" cores – 1.9 GHz, 1MB L2 cache
- "Performance" cores are 20% faster;
  "efficiency" cores used 80% of the time

- Graphics processing unit (GPU)
- Digital signal processor (DSP)
- Other custom accelerators (camera, modem, etc)





*Snapdragon 820
(only die shot I could find)

Multicore design issues

Multicore is all about **scalability**

- Sequential CPUs don't scale
  - Algorithmically: $O(\text{issue width}^2)$ comparisons
  - Technologically: Long wires are slow
- Solution: Replicate a smaller design

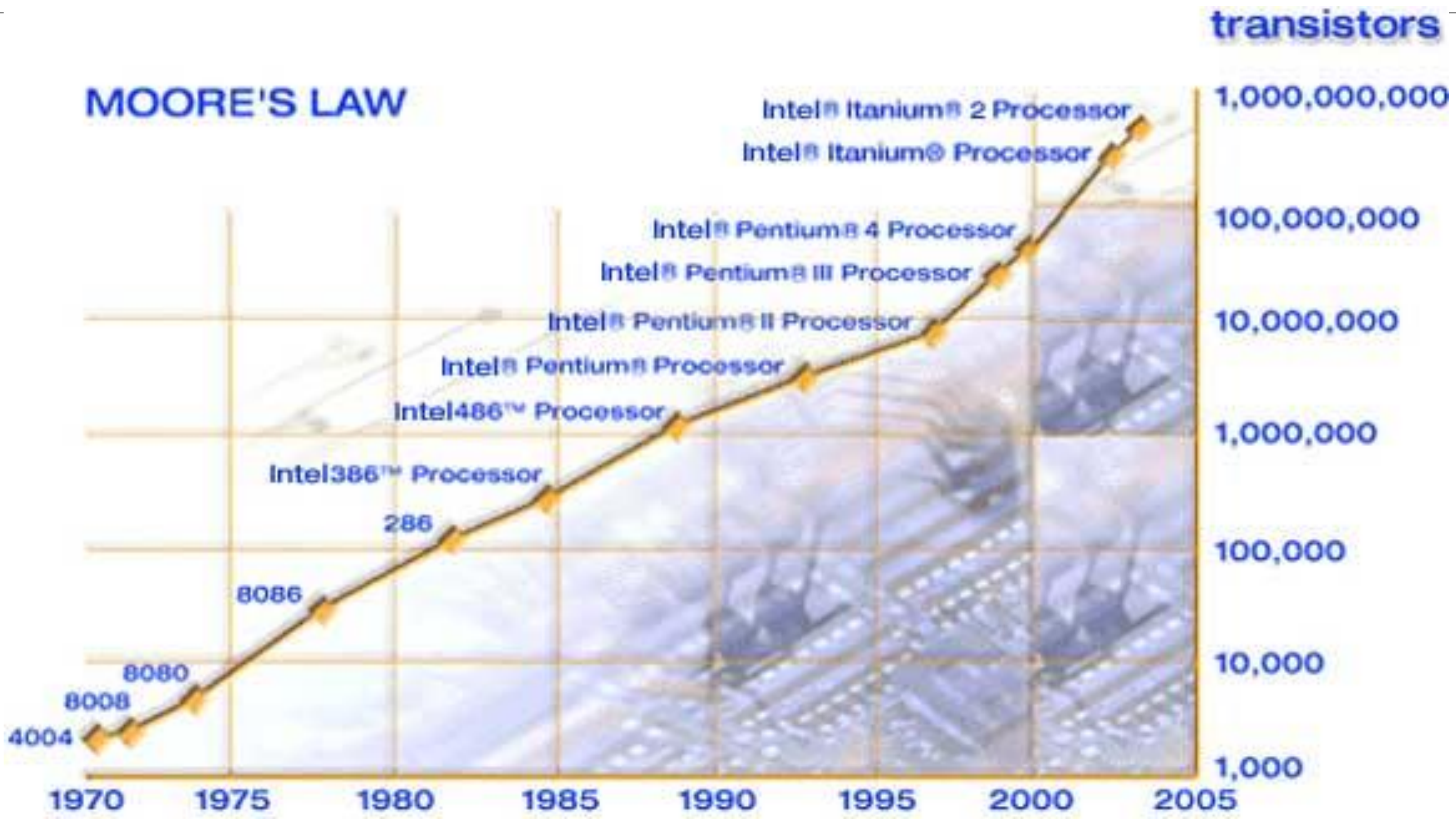
New design challenge: On-chip interconnect

What doesn't scale?

- On-chip communication (network distance & contention)
- Off-chip communication (limited by pins)
- Power/thermal dissipation (same physical object)

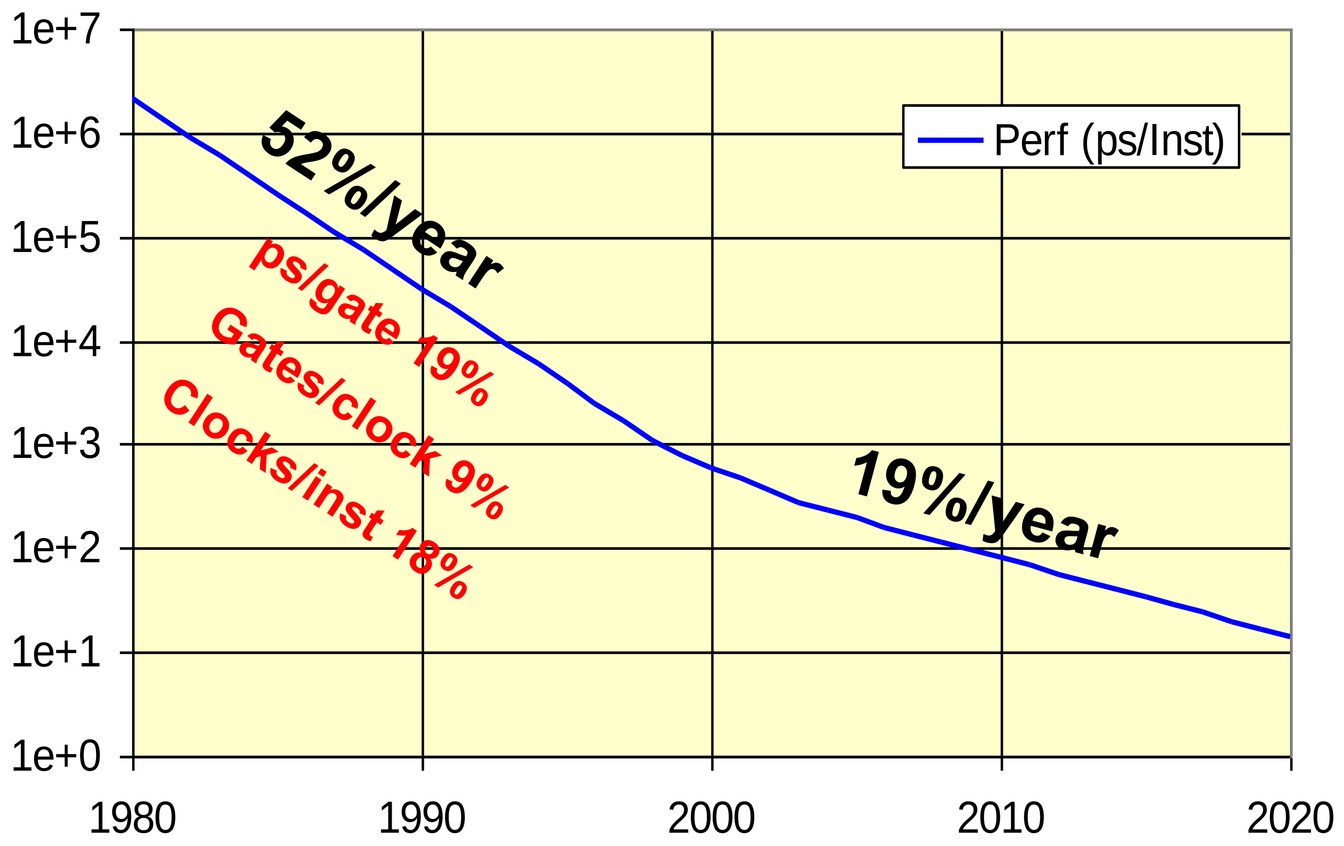
Why Multicore?

Moore’s Law



Moore, “Cramming more components onto integrated circuits,” Electronics, 1965.

50% / year performance scaling stops



Bill Dally

Multicore

Idea: Put multiple processors on the same die.

- *"The case for a single-chip multiprocessor," Kunle Olukotun et al, ASPLOS '96*

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

Why? What other ways could you use the extra transistors?

- 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)
- …

Why not a bigger, better single core?

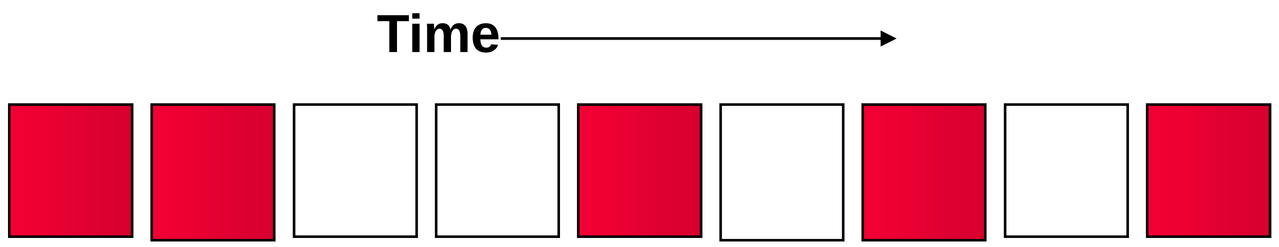
Alternative: Bigger, more powerful single core

- More superscalar – increase *issue width* (instrs / cycle)
- Add execution units
- More out-of-order scheduling – increase *instruction window*
- Larger L1 instruction / data caches
- Larger branch predictors
- Etc.

Why not bigger cores?

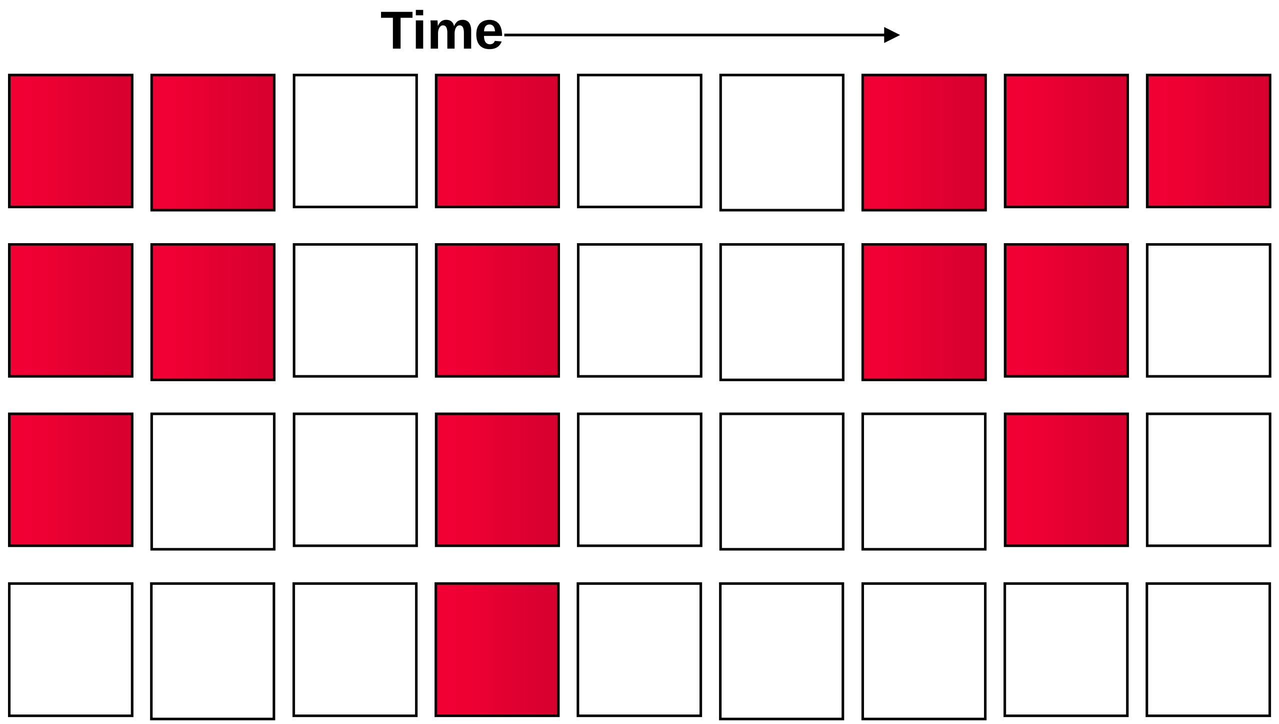
AKA “SUPERSCALAR”

Functional Unit Utilization



Data dependencies reduce functional unit utilization in pipelined processors

Functional Unit Utilization in Superscalar

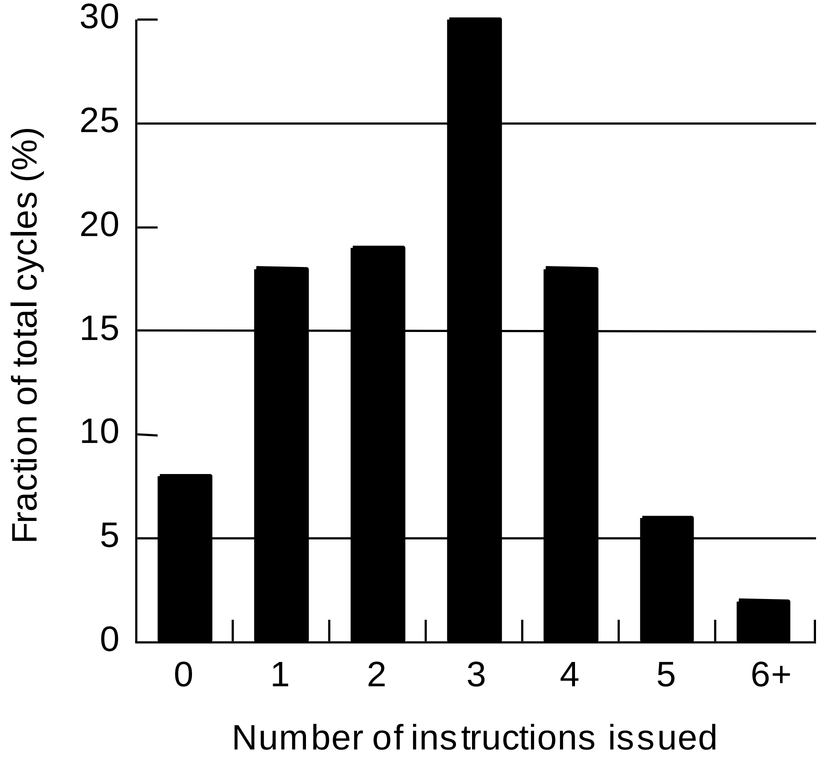


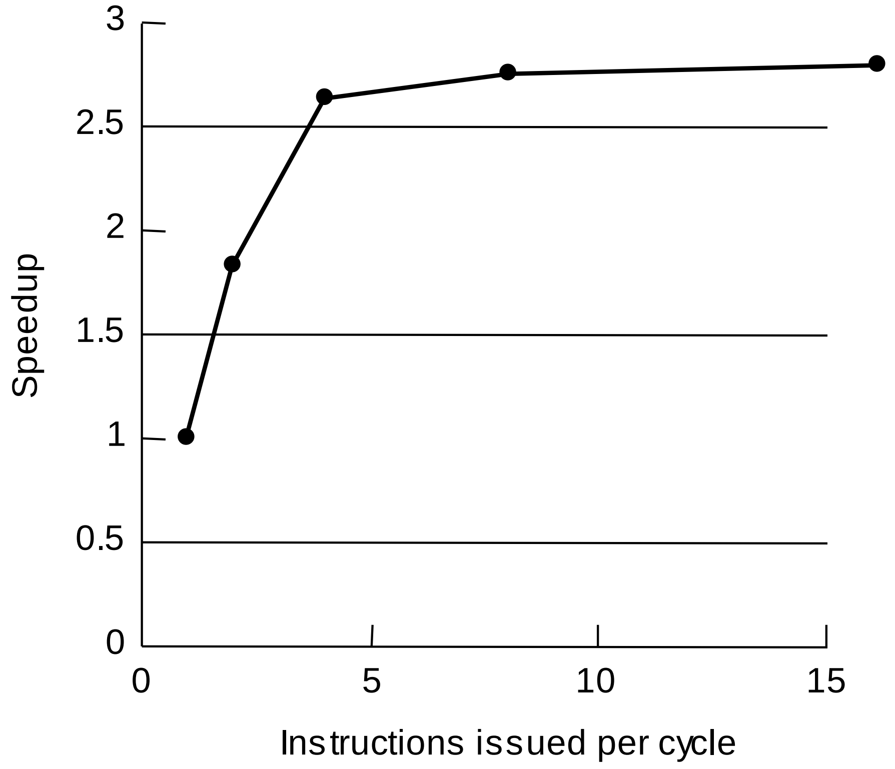
Functional unit utilization becomes lower in superscalar, OoO machines. Finding 4 instructions in parallel is not always possible

➔ Superscalar has *utilization* $\ll 1$ (as defined last lecture)

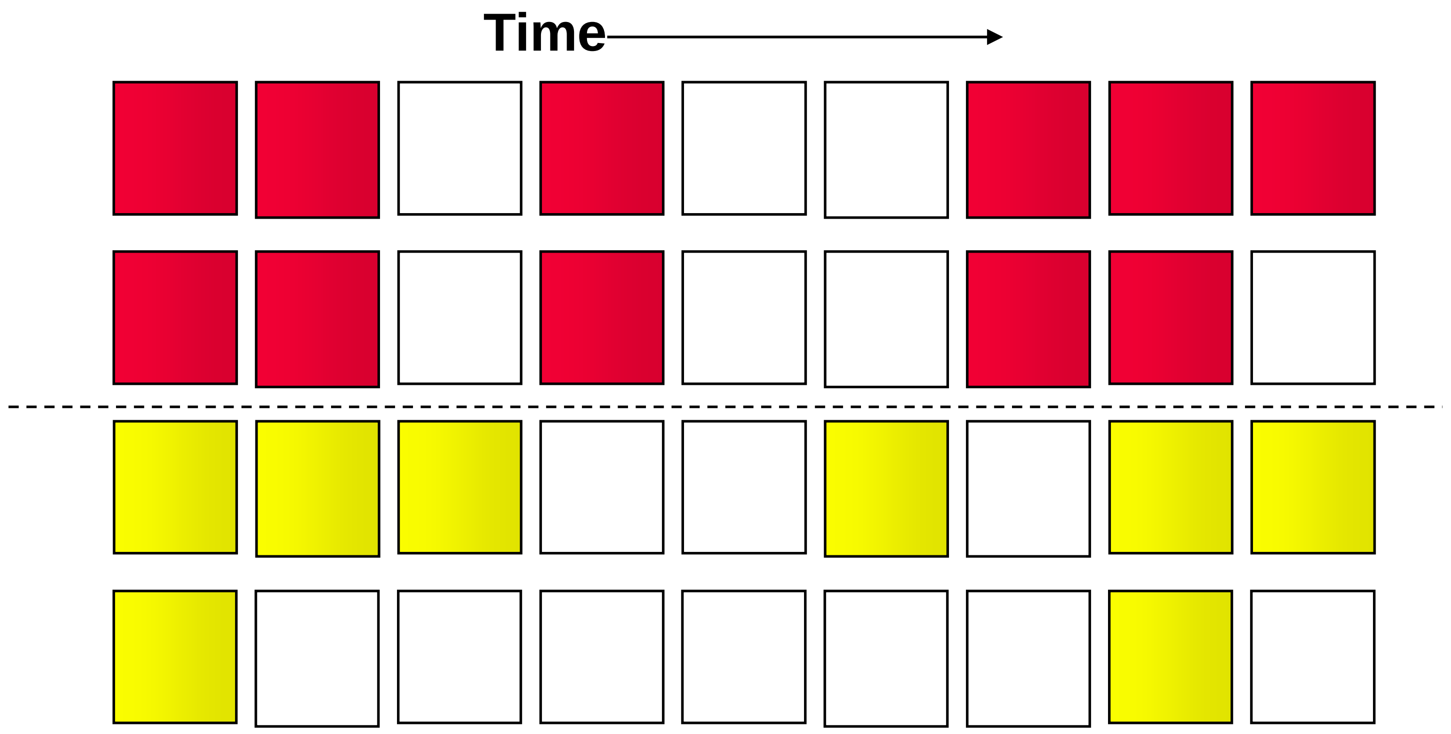
Limits to instruction-level parallelism

For most programs, its hard to find >4 instructions to schedule at once (and often less than this)





Chip Multiprocessor



Idea: Partition functional units across cores

Parallelism is explicit ➔ No dependences across threads ➔ Better FU utilization

Why not a bigger, better core?

+ Improves single-thread performance transparently to programmer, compiler

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

- Power & area hungry – many out-of-order execution structures scale $O(\text{issue width}^2)$

- Diminishing returns on performance

- Does not help memory-bound applications very much

Large Superscalar+OoO vs. Multi-Core

Olukotun et al., "The Case for a Single-Chip Multiprocessor," ASPLOS 1996.

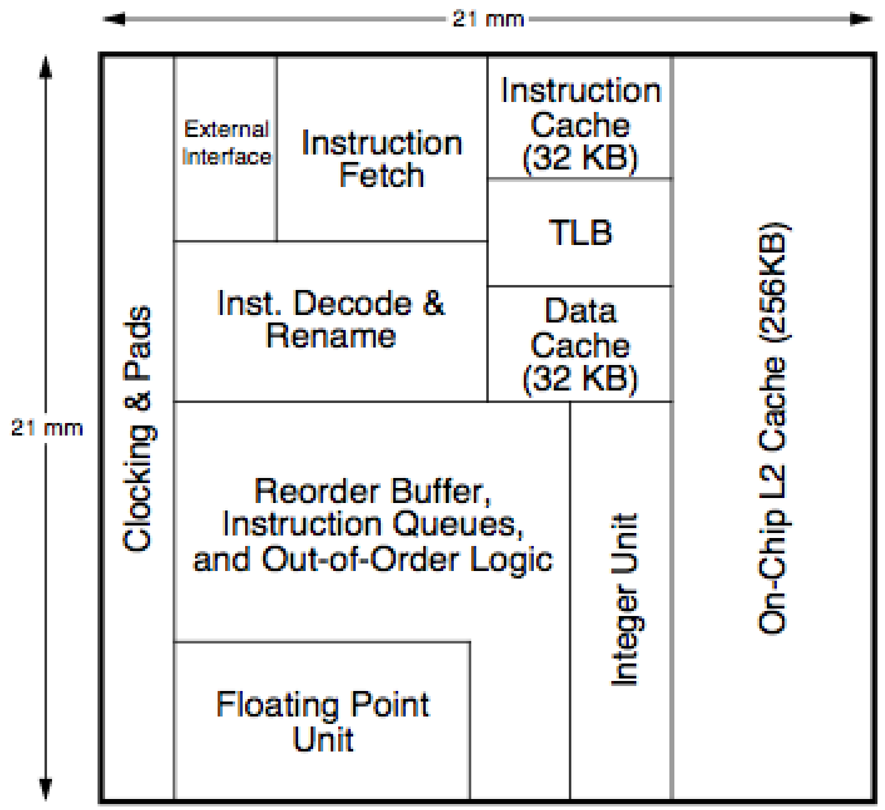


Figure 2. Floorplan for the six-issue dynamic superscalar microprocessor.

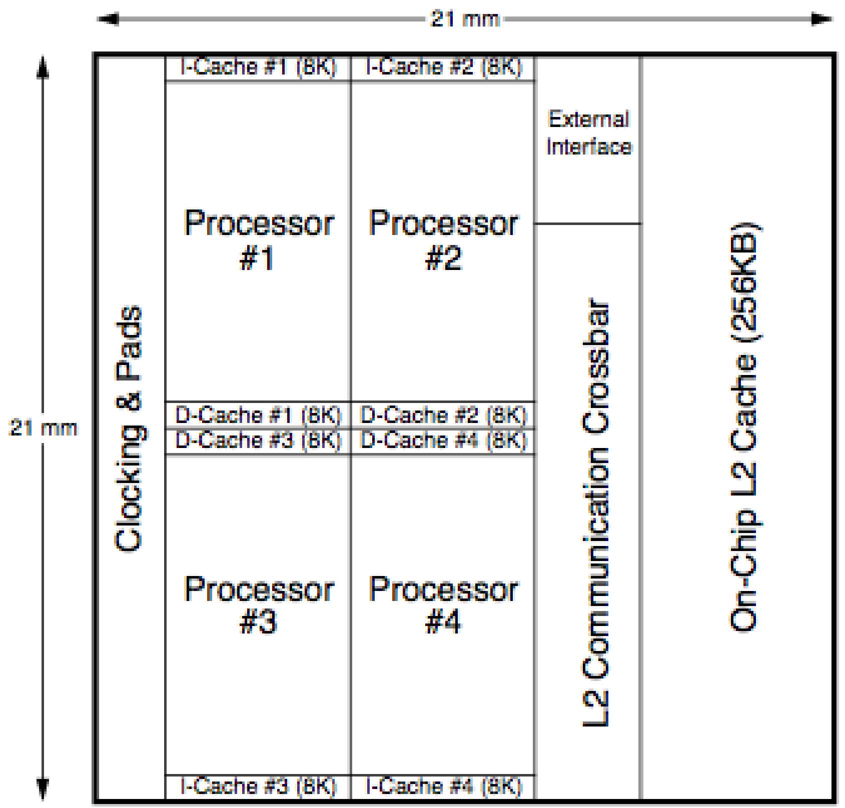


Figure 3. Floorplan for the four-way single-chip multiprocessor.

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 disadvantages

- Requires parallel tasks/threads to improve performance (parallel programming + Amdahl's Law)
- Resource sharing can reduce single-thread performance
- Shared hardware resources need to be managed
- Increased demand for off-chip bandwidth (limited by pins)

Simpler cores aren't *that* much slower on sequential programs (~30%)

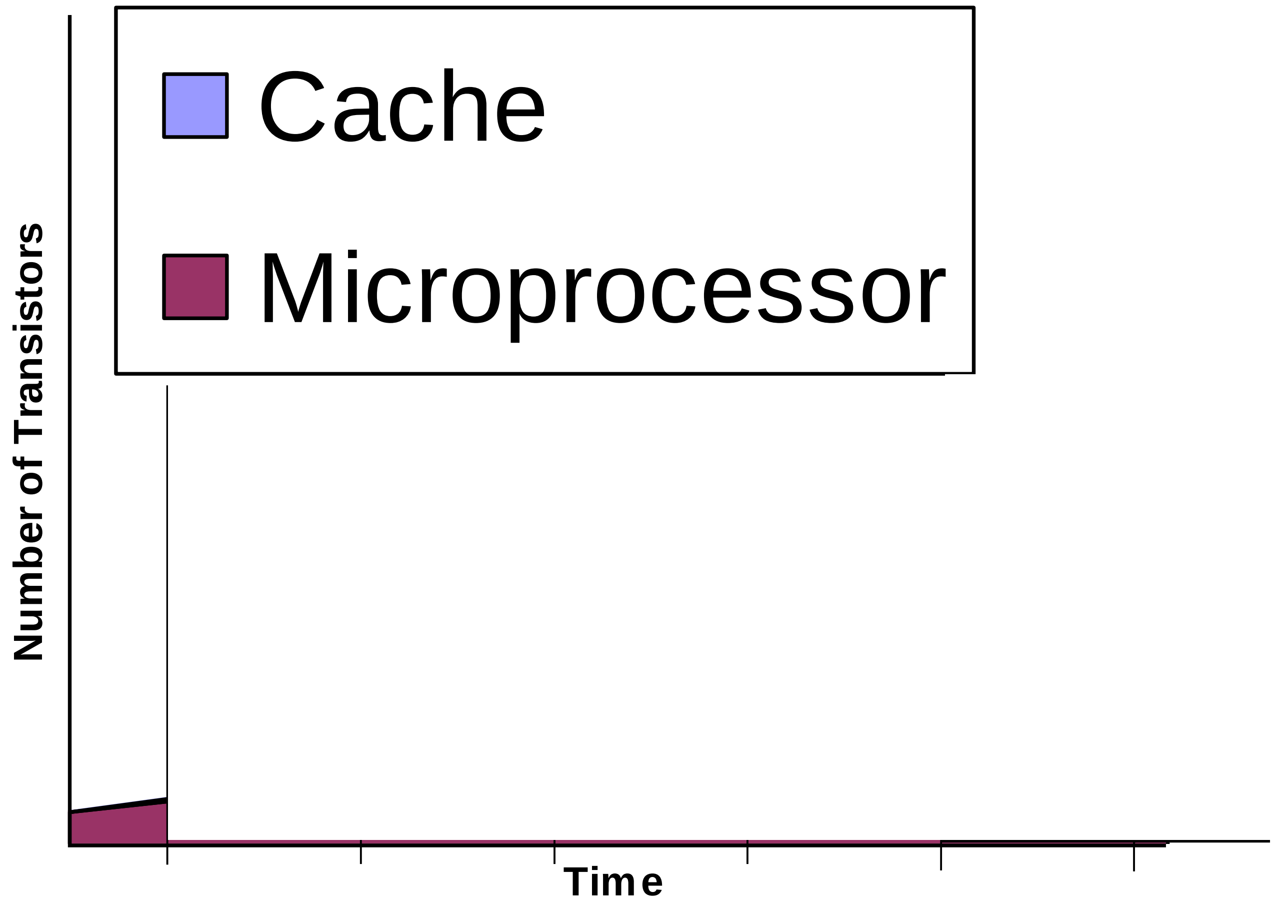
Why not bigger caches?

Why Not bigger caches?

+ Improves single-thread performance transparently to programmer, compiler
+ Simple to design

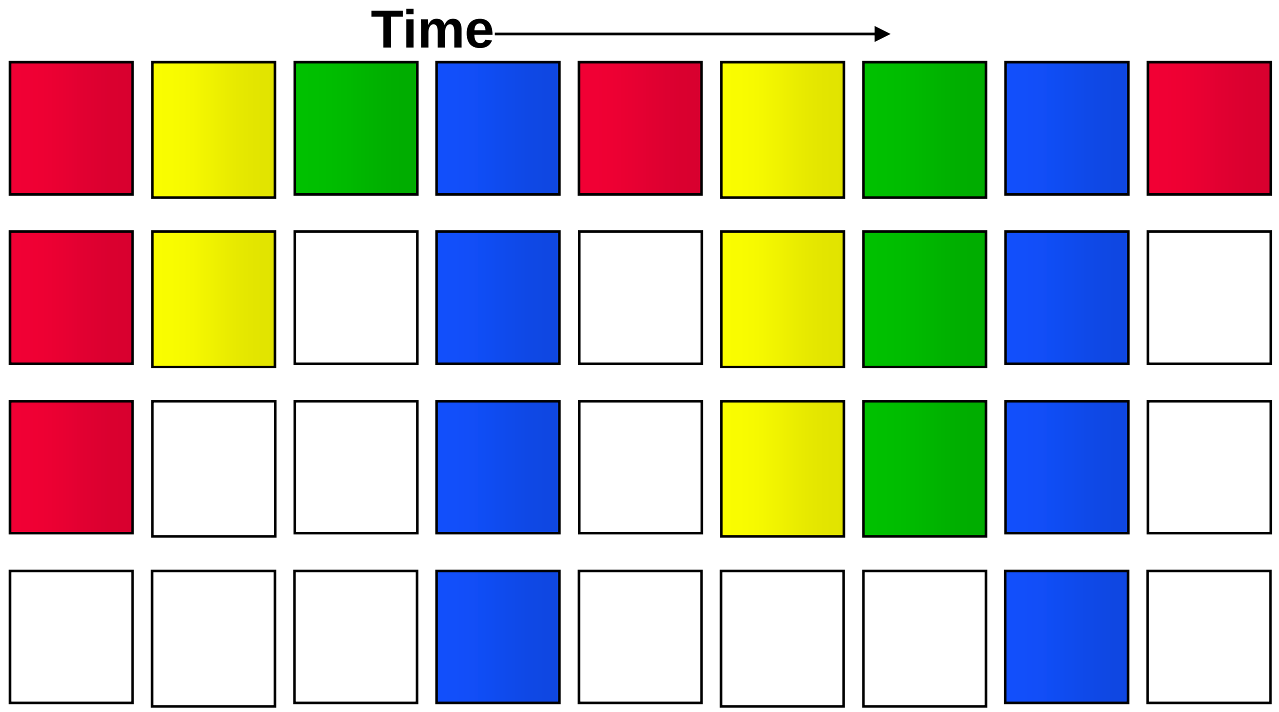
- Diminishing single-thread performance returns from cache size. Why?
- Multiple levels complicate memory hierarchy

Area for Cache vs. Core



Why not multithreading?

Fine-grained Multithreading



Idea: Time-multiplex execution units across threads

Hides latency of long operations, improving utilization

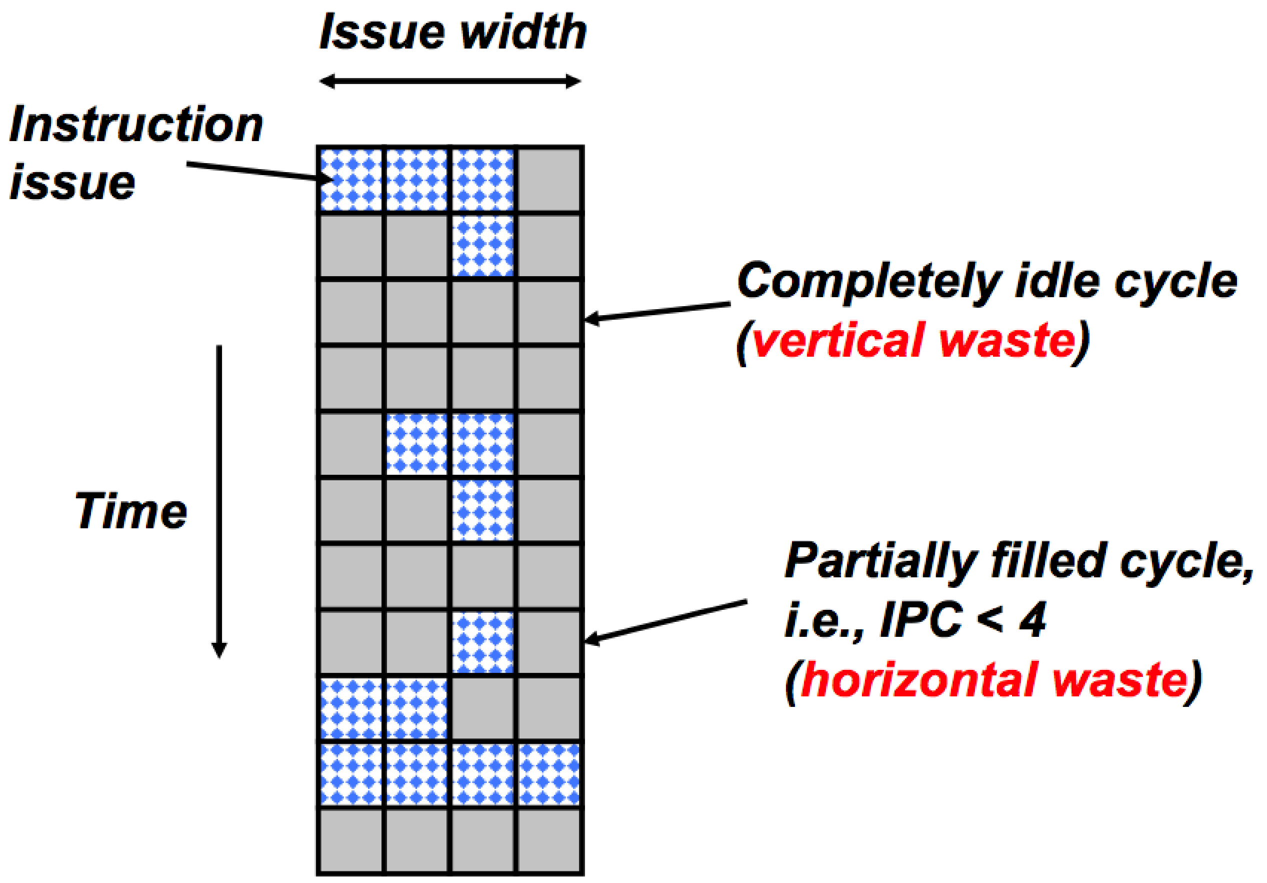
…But single thread performance suffers (in naïve versions)

Horizontal vs. Vertical Waste

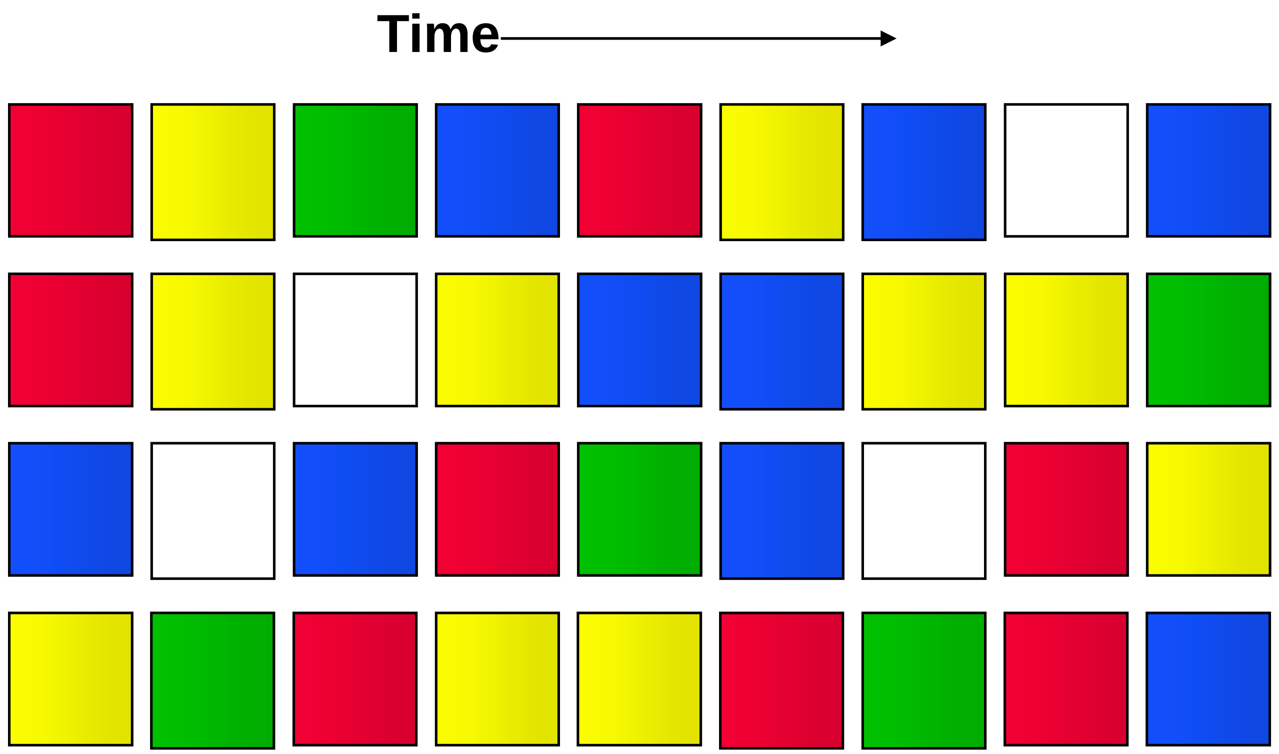
What causes horizontal waste?

vertical waste?

How do you reduce each?


Issue width
Instruction issue
Completely idle cycle (vertical waste)
Time
Partially filled cycle, i.e., IPC < 4 (horizontal waste)

Simultaneous Multithreading



Idea: Utilize functional units with independent operations from the same or different threads

Simultaneous Multithreading

Reduces both horizontal and vertical waste

Required hardware

- The ability to dispatch instructions from multiple threads simultaneously into different functional units

Superscalar, OoO processors already have this machinery

- Dynamic instruction scheduler searches the scheduling window to wake up and select ready instructions
- As long as dependencies are correctly tracked (via renaming and memory disambiguation), scheduler can be thread-agnostic

Why Not Multithreading?

Alternative: (Simultaneous) Multithreading

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

+ Good single-thread performance in SMT

+ Efficient: Don’t need an entire core for another thread

+ Communication faster through shared L1 caches (SAS model)

- Scalability is limited: need bigger register files, more function units, larger issue width (and associated costs) to have many threads $\rightarrow$ 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 not clustering?

& MANY OTHER PROPOSALS IN LATE ‘90S/EARLY ‘00S TO SCALE INDIVIDUAL CORES

Clustered Superscalar+OoO Processors

Clustering (e.g., Alpha 21264 integer units)

- Divide the scheduling window (and register file) into multiple clusters
- Instructions steered into clusters (e.g. based on dependence)
- Clusters schedule instructions out-of-order, within cluster scheduling can be in-order
- Inter-cluster communication happens via register files (no full bypass)

+ Helps scalability of monolithic OOO: Smaller scheduling windows, simpler wakeup algorithms

+ Fewer ports into register files

+ Faster within-cluster bypass

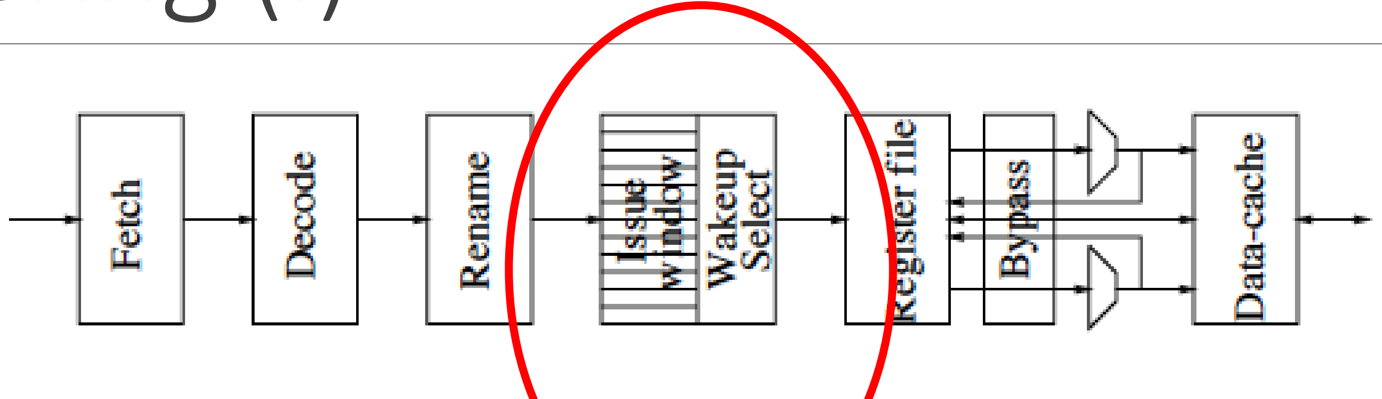
- Extra delay when instructions require across-cluster communication

- Inherent difficulty of steering logic

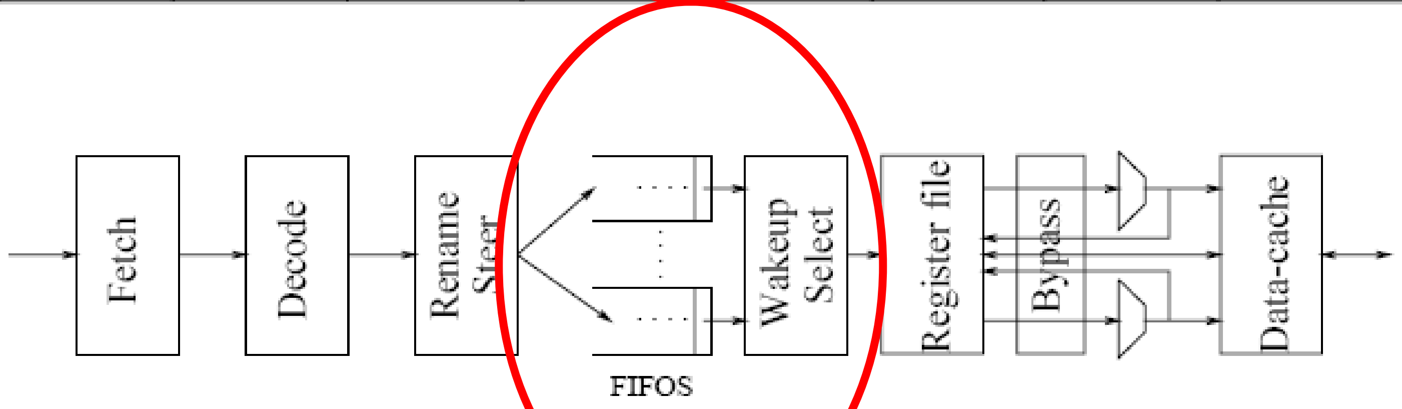
Kessler, "The Alpha 21264 Microprocessor," IEEE Micro 1999.

Clustering (I)

Palacharla et al., “Complexity Effective Superscalar Processors,” ISCA 1997.



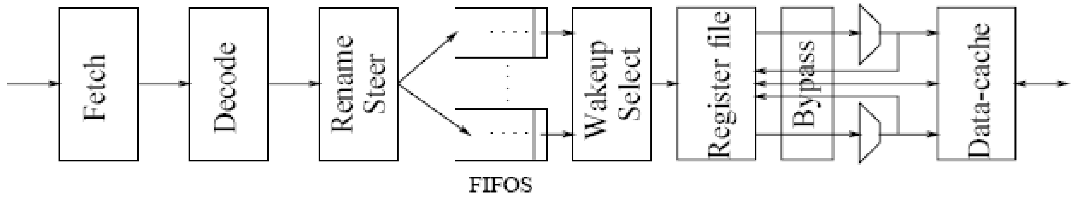
FETCH	DECODE	RENAME	WAKEUP SELECT	REG READ	EXECUTE BYPASS	DCACHE ACCESS	REG WRITE COMMIT



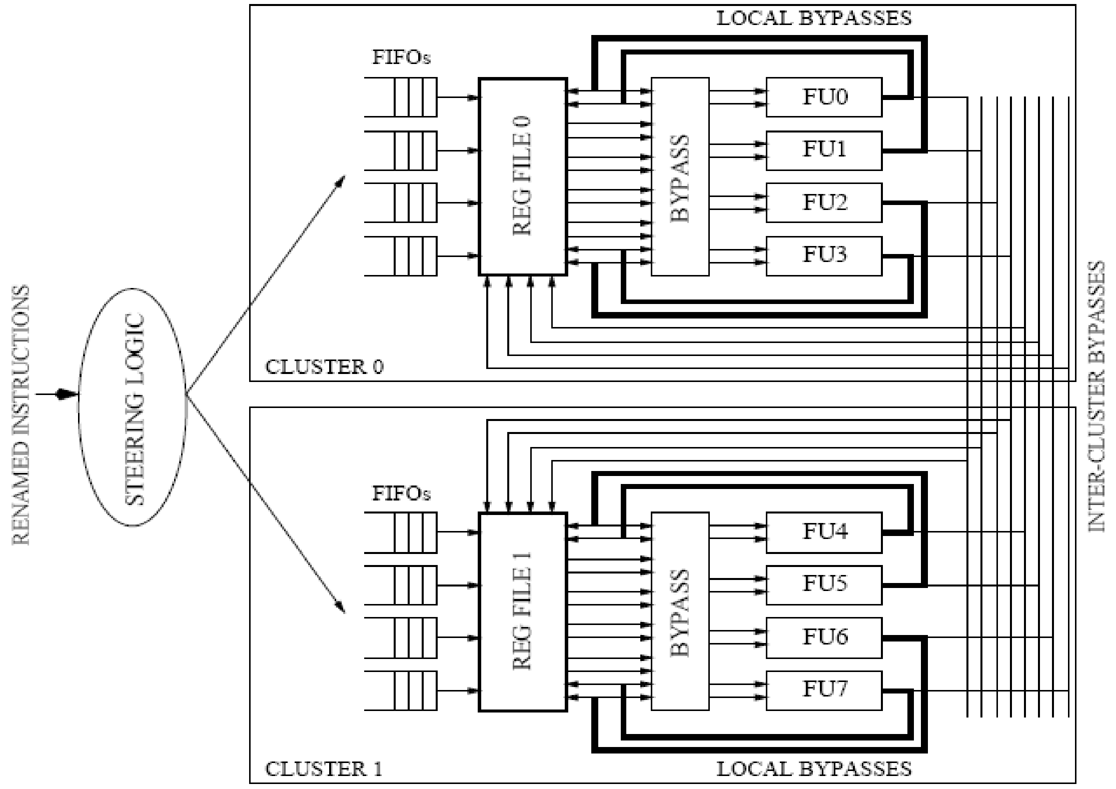
FETCH	DECODE	RENAME STEER	WAKEUP SELECT	REG READ	EXECUTE BYPASS	DCACHE ACCESS	REG WRITE COMMIT

Clustering (II)

Palacharla et al., "Complexity Effective Superscalar Processors," ISCA 1997.



FETCH	DECODE	RENAME STEER	WAKEUP SELECT	REG READ	EXECUTE BYPASS	DCACHE ACCESS	REG WRITE COMMIT

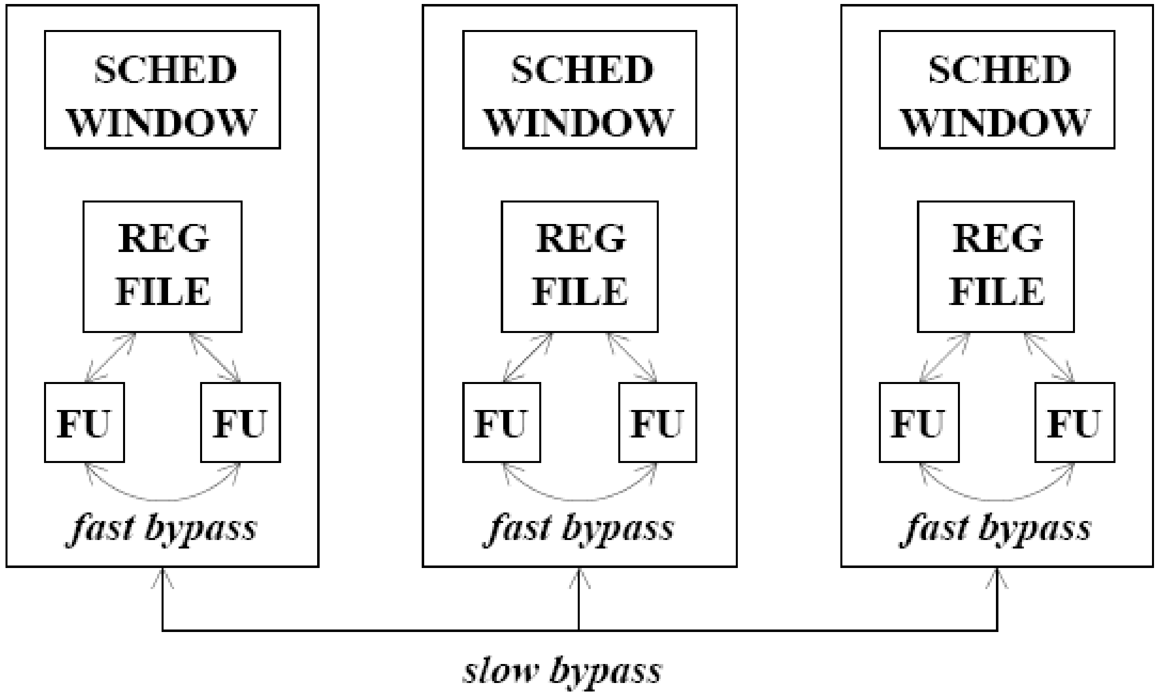


Each scheduler is a FIFO
+ Simpler
+ Can have N FIFOs
(OoO w.r.t. each other)
+ Reduces scheduling complexity
- More dispatch stalls

Inter-cluster bypass: Results produced by an FU in Cluster 0 is not individually forwarded to each FU in another cluster.

Clustering (III)

Scheduling within each cluster can be out of order



Brown, "Reducing Critical Path Execution Time by Breaking Critical Loops," UT-Austin 2005.

Why Not Clustering?

+ Simpler to design than superscalar, more scalable than simultaneous multithreading (less resource sharing)

+ Can improve both single-thread and parallel application performance

- Diminishing performance returns on single thread: Clustering reduces IPC performance compared to monolithic superscalar. Why?

- Parallel performance limited by shared fetch bandwidth

- Difficult to design

Why Not …?

Why Not System on a Chip?

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)

Today system-on-chip is increasingly common, but it’s worth remembering that SoC is **third-best option** (after sequential scaling & multicore)

Why Not Multi-Chip Multiprocessor?

Alternative: Traditional symmetric multiprocessors

+ Smaller die size (for the same processing core)

+ More memory bandwidth (no pin bottleneck)

+ Fewer shared resources (eg, cache)
  → less contention between threads

- Long latencies between cores (need to go off chip)
  → communication limits performance
  → parallel application scalability suffers

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

Why Not …?

Dataflow?

- Yes—OOO scheduling, but has scaling problems beyond

Vector processors (SIMD)?

- Yes—SSE/AVX + GPUs, but not a general solution

Streaming processors/systolic arrays?

- Too specialized outside embedded

VLIW? (very-long instruction word)

- Compilers struggle to find ILP too (bigger window, but must prove independence statically)

Integrating DRAM on chip?

- Rarely, but DRAM wants different manufacturing process

Reconfigurable logic?

- General purpose?

Why Multi-Core (Cynically)

Huge investment and need ROI

Have to offer some kind of upgrade path

It is easy for the processor manufacturers

Why Multi-Core (Cynically)

Huge investment and need ROI

Have to offer some kind of upgrade path

It is easy for the processor manufacturers

But, seriously…

Some easy parallelism
- Most general purpose machines run multiple tasks at a time
- Some (very important) apps have easy parallelism

Power is a real issue

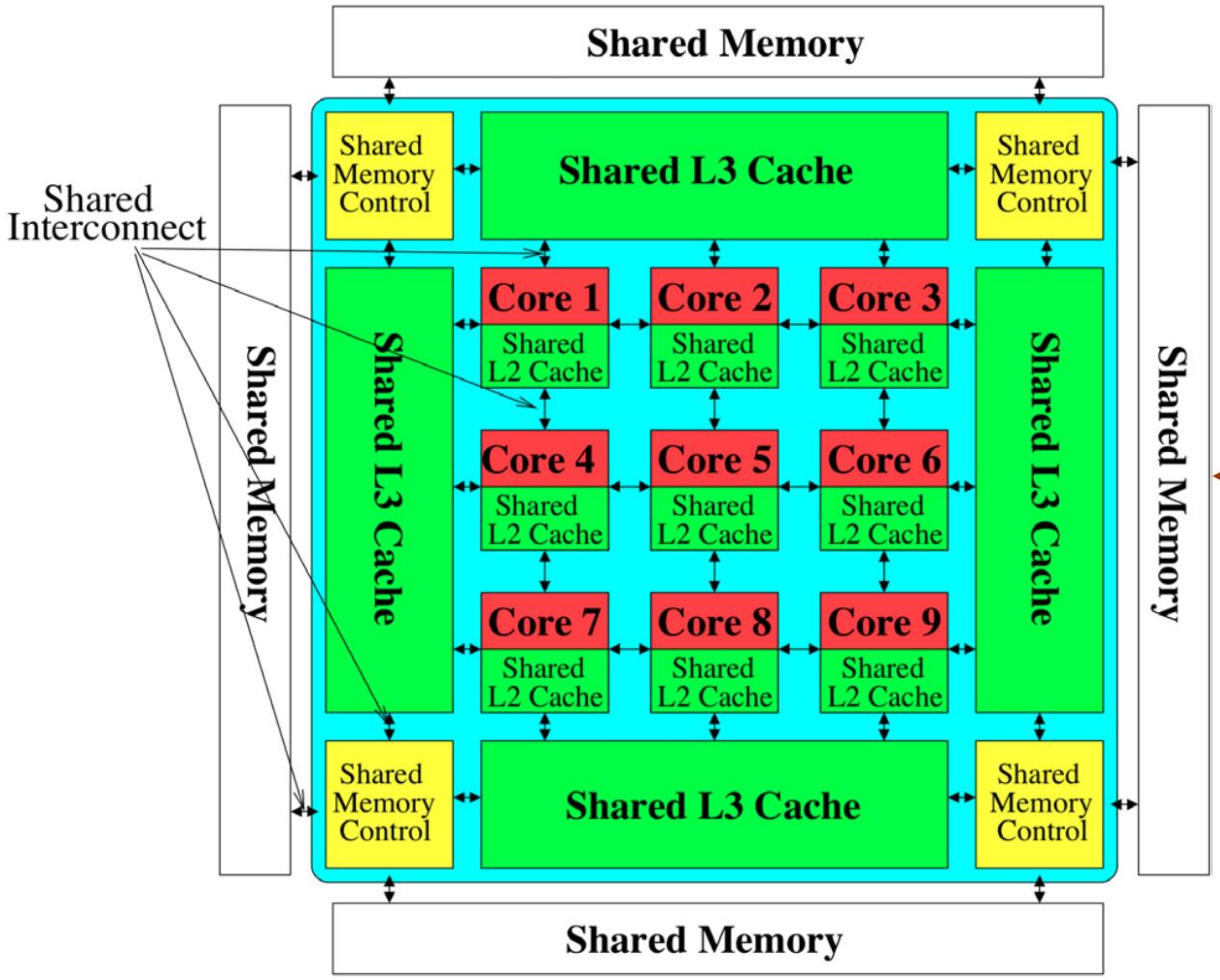
Design complexity is very costly

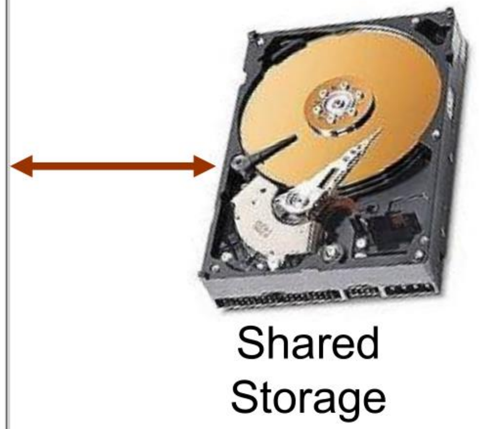
Still need good sequential performance (Amdahl's Law)

Is it the right solution?

On-chip Interconnect

On-chip interconnect in a multicore





Where is interconnect used?

To connect components

Processor-to-processor

Processor-to-cache

Cache-to-cache

Cache-to-memory

I/O-to-memory

Etc.

Why is interconnect important?

Affects **scalability** of the system

- How large a system can you build?
- How easily can you add more processors/caches?

Affects performance & energy efficiency

- How fast can processors, caches, memories communicate? (longer than cache access)
- How much energy is spent on communication? (10-35%)

Interconnect basics

Topology

- How switches are wired to each other
- Affects routing, reliability, throughput, latency, cost

Routing (algorithm)

- How does a message get from source to destination?
- Static vs adaptive

Buffering and flow control

- What do we store within the network? (Packets, headers, …?)
- How do we throttle when oversubscribed?
- Tightly coupled with routing

Interconnect topologies

Bus (simplest)

Point-to-point (ideal and most costly)

Crossbar (less costly)

Ring

Mesh

Tree

Omega

Hypercube

Torus

Butterfly

…

Interconnect metrics

Cost (area)

Latency (hops, cycles, nanoseconds)

Contention

Energy

Bandwidth (“bisection” b/w)

End-to-end system performance

Bus

+ Simple

+ Cost-effective for small number of nodes

+ Easy to implement coherence (global broadcast)

- Poor scalability (electrical limitations)

- High contention

Point-to-point

Every node connected directly to every other

+ Lowest contention

+ Lowest latency (maybe—wire length, wasted area)

+ Ideal except for cost

- Highest cost
  - $O(N^2)$ links

- Not scalable

- Physical layout??

Crossbar

Every node connected to every other, but only one at a time

Concurrent communication to different destinations

Good with few nodes

+ Low latency & high throughput

- Expensive

- Doesn’t scale -- $O(N^2)$ switches

- Difficult to arbitrate with many nodes

Used in many designs (e.g., Sun UltraSPARC T1)

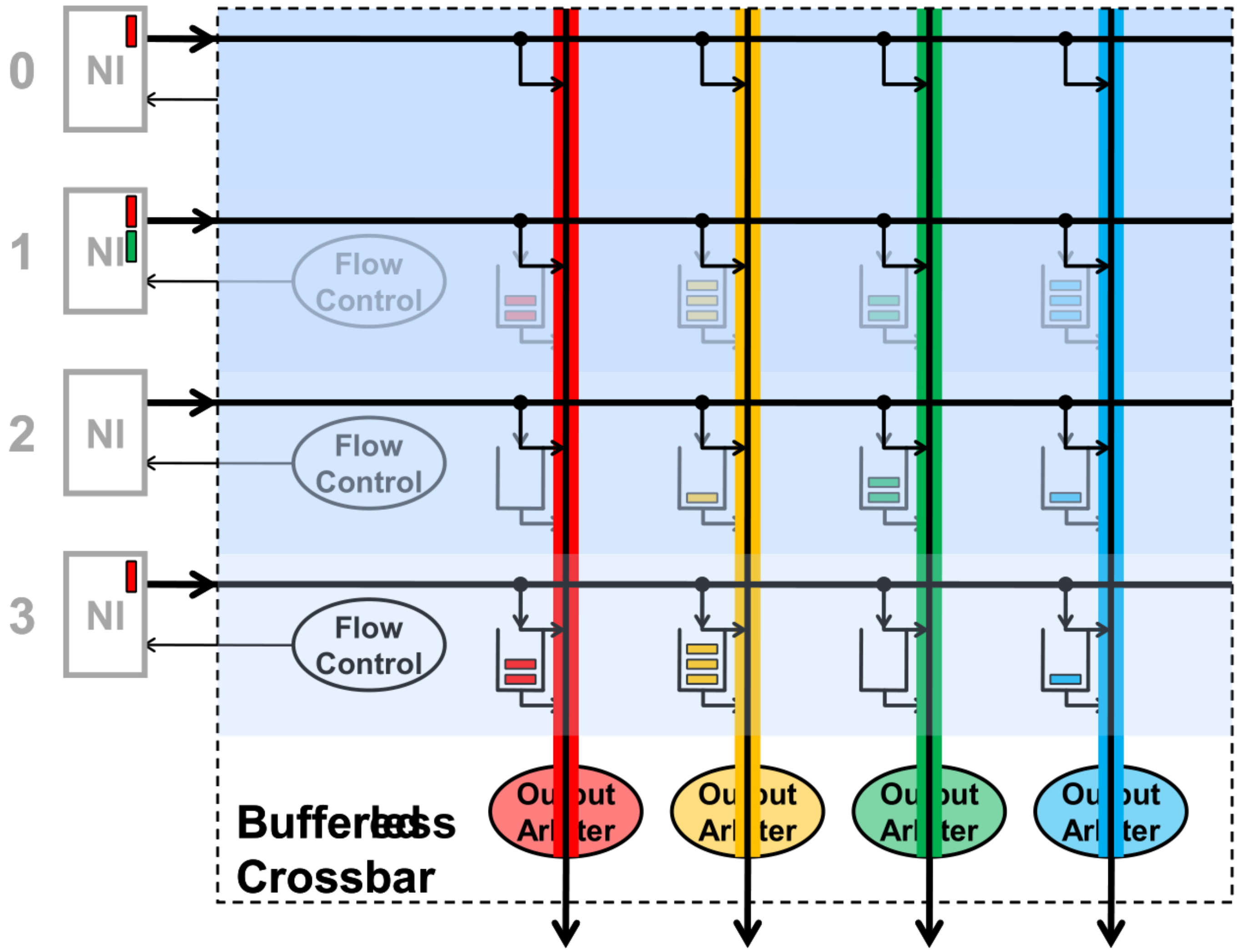
Buffered crossbar

+ Simpler arbitration & scheduling

+ Efficient support for variable sized packets

- Requires $O(N^2)$ buffers

Can we scale the interconnect without contention?

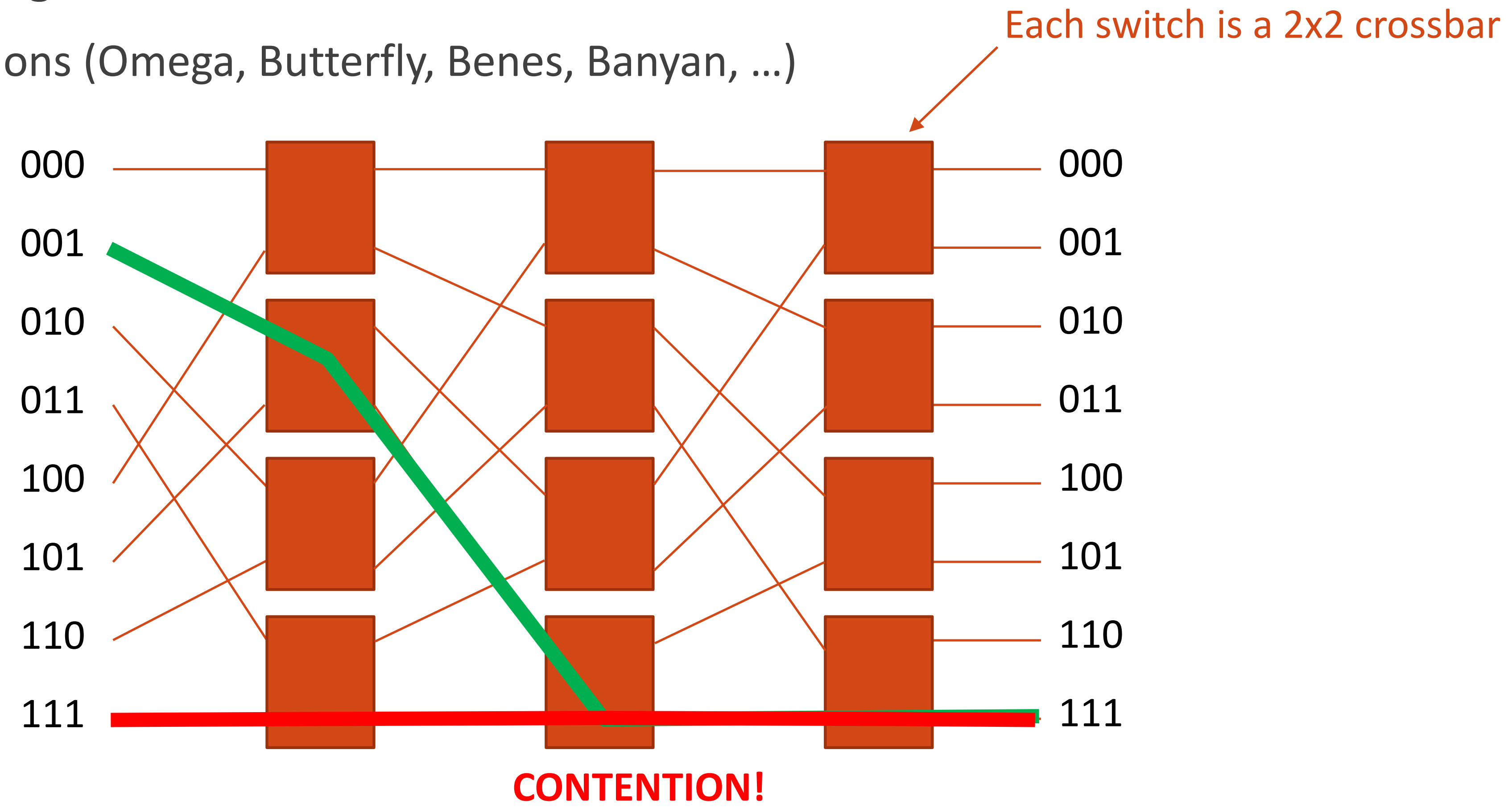


Multistage networks

Idea: $\log N$ switches between nodes

+ Cost $O(N \log N)$

Many variations (Omega, Butterfly, Benes, Banyan, …)



Handling contention

Two packets try to use same link at the same time

What do you do?

- Buffer one
- Drop one
- Misroute one (deflection)

Let’s assume buffering for now

Ring

Unidirectional or bidirectional

+ Cheap $O(N)$ switches

+ Simple switches ➔ Low hop latency

- High latency $O(N)$

- Not scalable; **bisection bandwidth** is constant

Used in many commercial systems today; recently Intel switched to “ring of rings” topology

2D Mesh

+ $O(N)$ cost

+ $O(\sqrt{N})$ average latency

+ Natural physical layout

+ Path diversity: Many routes between most sources & destinations
  - Potentially lower contention

+ Decent bisection bandwidth

- More complex routers ➔ Higher hop latency

Used in Tilera 100-core chip & most research prototypes

Trees

Planar, hierarchical topology

+ $O(\log N)$ latency

+ $O(N)$ cost

+ Easy to layout

- Root is bottlenect; constant bisection bandwidth

Trees common for local communication; e.g., banks of single cache

Bisection bandwidth mitigated by “fat trees”, at add’l cost

- Replicate root node, randomize routing
- Used in Thinking Machines CM-5 (1992)

Flow control methods

Circuit switching

Packet switching

- Store and forward
- Virtual cut-through
- Wormhole

Circuit switching

Pre-allocate resources across multiple switches

Requires “probe” ahead of message

+ No need for buffering

+ No contention (after circuit established)

+ Handles arbitrary message sizes

- Low link utilization

- Delay to set up circuit

Store and forward

Copy entire packet between switches

+ Simple

- High per-packet latency

- Requires big buffers / small messages

Virtual cut-through

Start forwarding as soon as header is received

+ Dramatic reduction in latency vs store and forward

- Still buffers entire message in worst case: requires large buffers / small messages

Wormhole

Break packets into much smaller “flits”

Pipeline delivery: Each flit follows its predecessor through network

If head is blocked, rest of packet waits in earlier switches

+ No large buffering in network

+ Latency independent of distance for large messages

- Head-of-line blocking

Routing algorithms

Deterministic: Simplest, high contention

- Dimension-order (e.g., XY)
- Deadlock-free

Oblivious: Simple, mitigates contention

- Valiant’s algorithm: Route deterministically via a random node
- Balances network load, adds latency
- Optimization: Use only at high load

Adaptive: Complex, most efficient

- Minimal adaptive: Always route closer to destination on least-contended port
- Fully adaptive: “Misroute” packets to optimize overall network load
  - Must guard against livelock
  - How to coordinate overall network state?

Computer architecture today

Multicore introduces many new challenges

Today is a very exciting time to study computer architecture

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

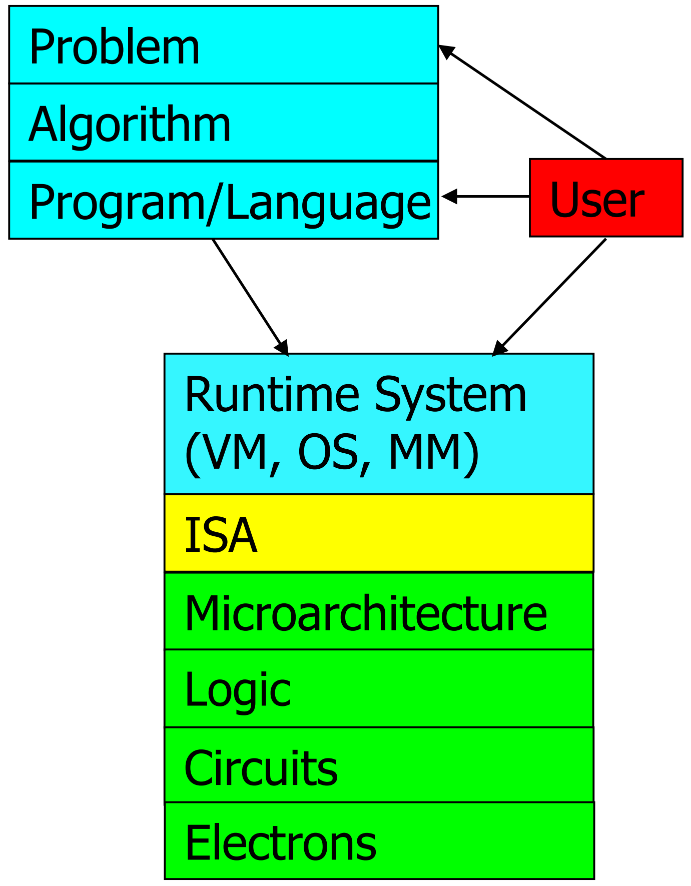
Many difficult problems *motivating* and *caused by* the shift

- 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?

No clear, definitive answers to these problems

Multicore affects full system stack

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



No clear, definitive answers to these problems

…But is multicore the answer?

Multicore: more transistors ➔ more cores

But…Amdahl's Law

- More cores only helps parallel region of programs
  ➔ Still want better sequential performance
  ➔ But we went multicore to avoid all the problems with scaling sequential performance!

But…Moore's Law finally dying?

- $$ / transistor rising; Intel slowing *tick-tock* cycle
- ➔ Multicore stops scaling even parallel performance

Many now believe *specialization* is the answer

- Very disruptive to software
- Again, an exciting time to be in architecture