Multicore: History and Design Challenges

15-740 FALL'21

NATHAN BECKMANN (SLIDES BASED ON ONUR MULTLU'S)

Topics

History: Review of early → current multicores

Why multicore?

- Why did it take so long?
- What were the alternatives? (via cartoons)

Multicore architectural challenges

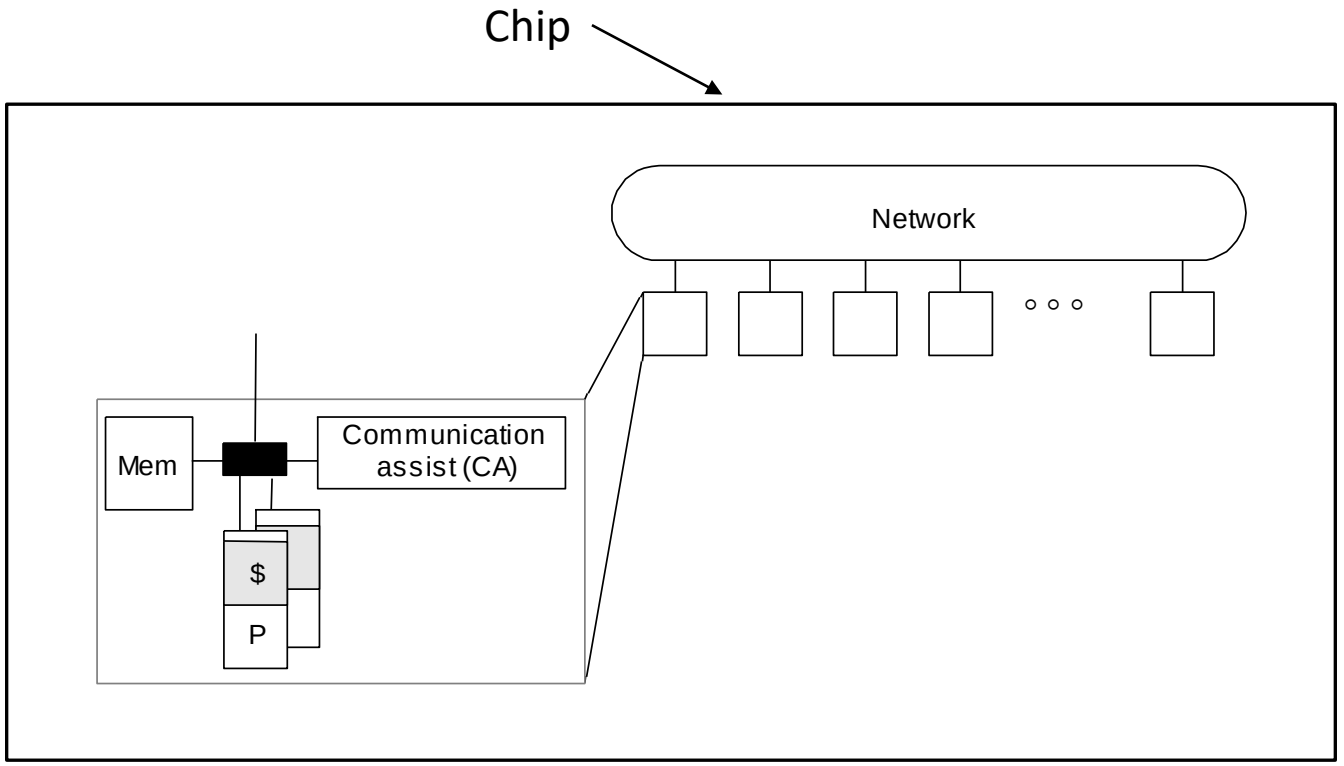
- Granularity
- Resource sharing
- Communication

Retrospective

What is multicore?

Technology lets us put build a parallel architecture on a single chip

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



Example: IBM POWER4 (2001)

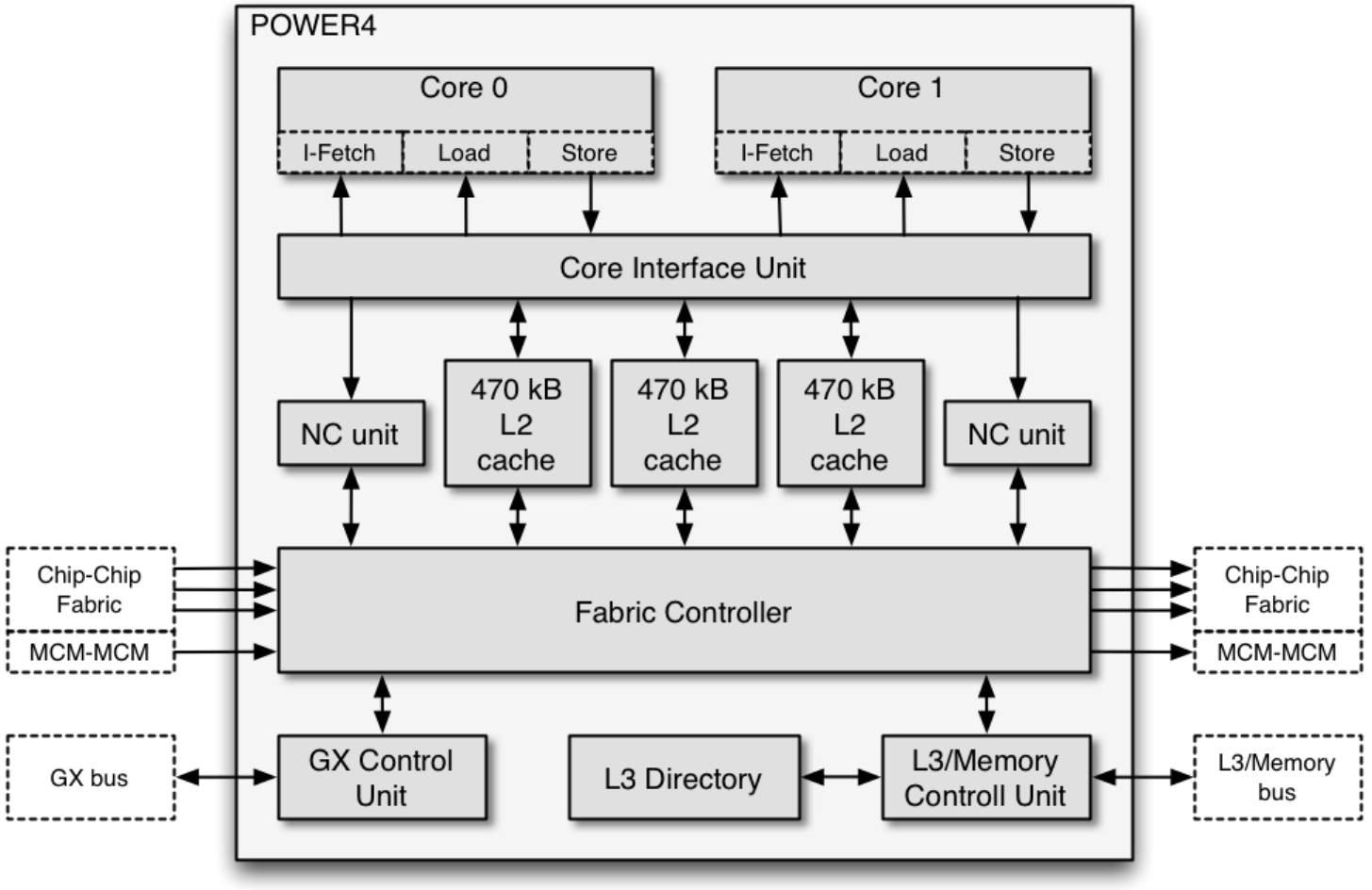
174M transistors @ 180nm

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

L2 is banked x3

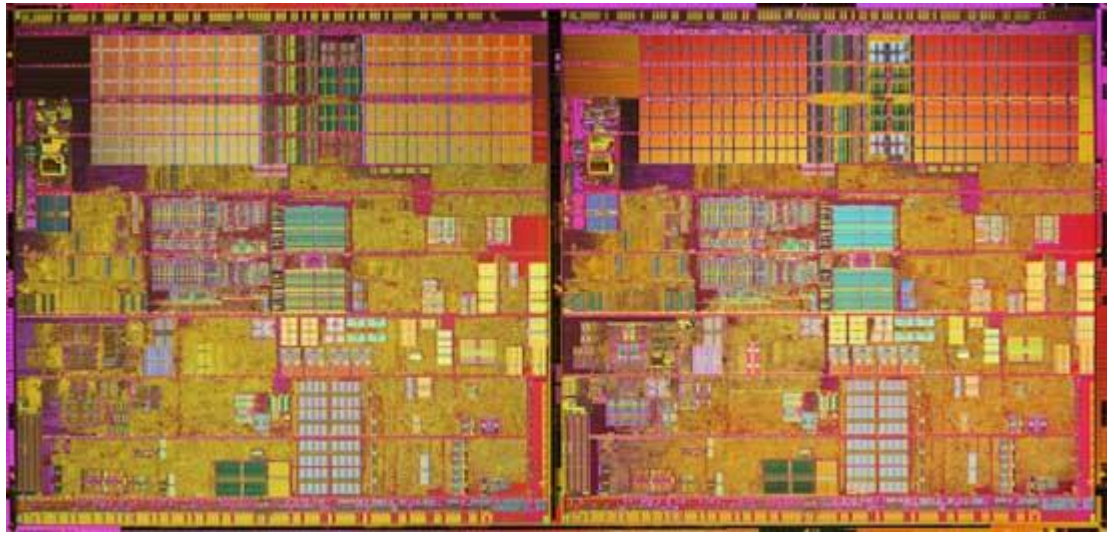
L3 control on-chip, memory off-chip

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

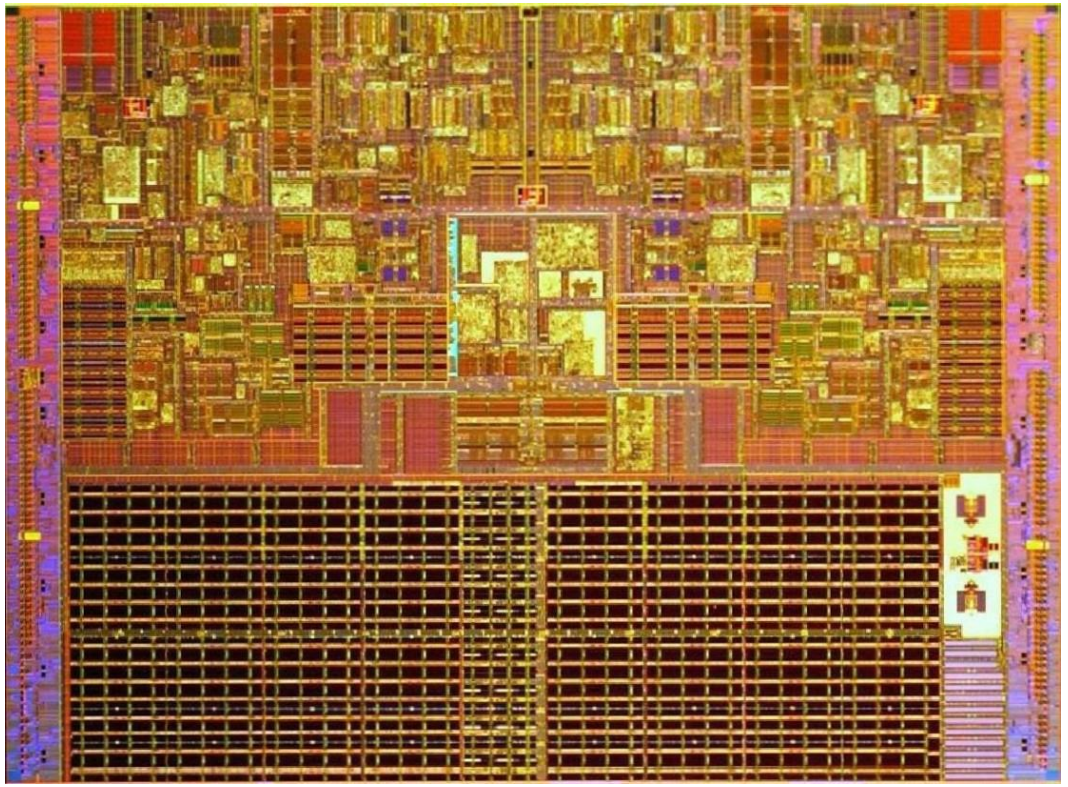
Core

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

2 cores on single die

2MB L2 shared cache

31 Watts



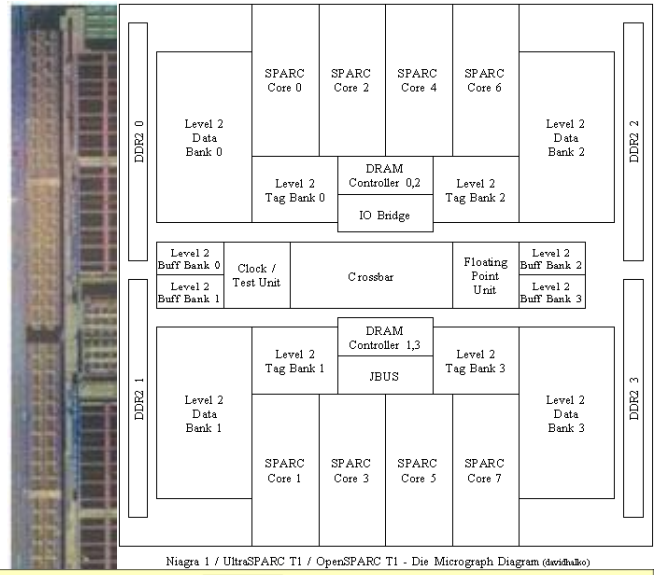
Example: Sun UltraSPARC T1 (2005)

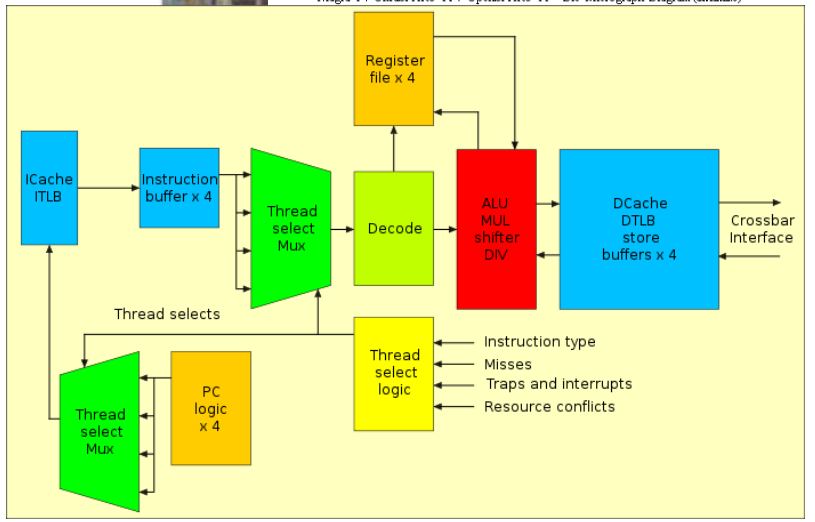
279M transistors @ 90nm
- 378mm^2 (!!)

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





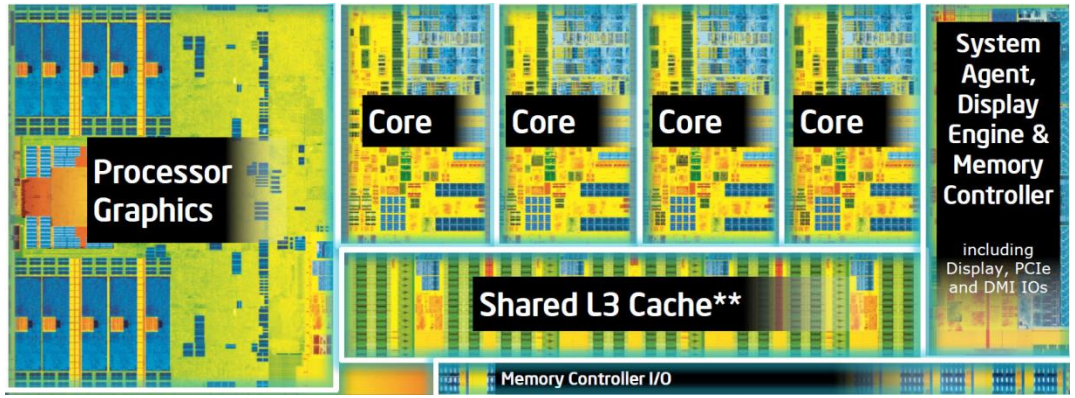
Example: Intel Core i7 (2013)

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



Four cores in single die

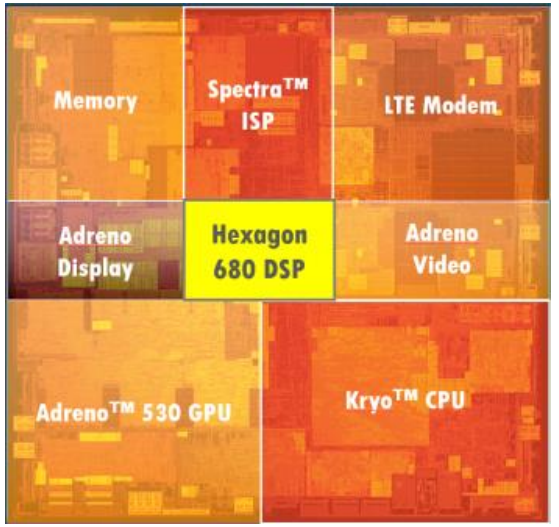
Example: Qualcomm Snapdragon 835 (2017)

?? 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)

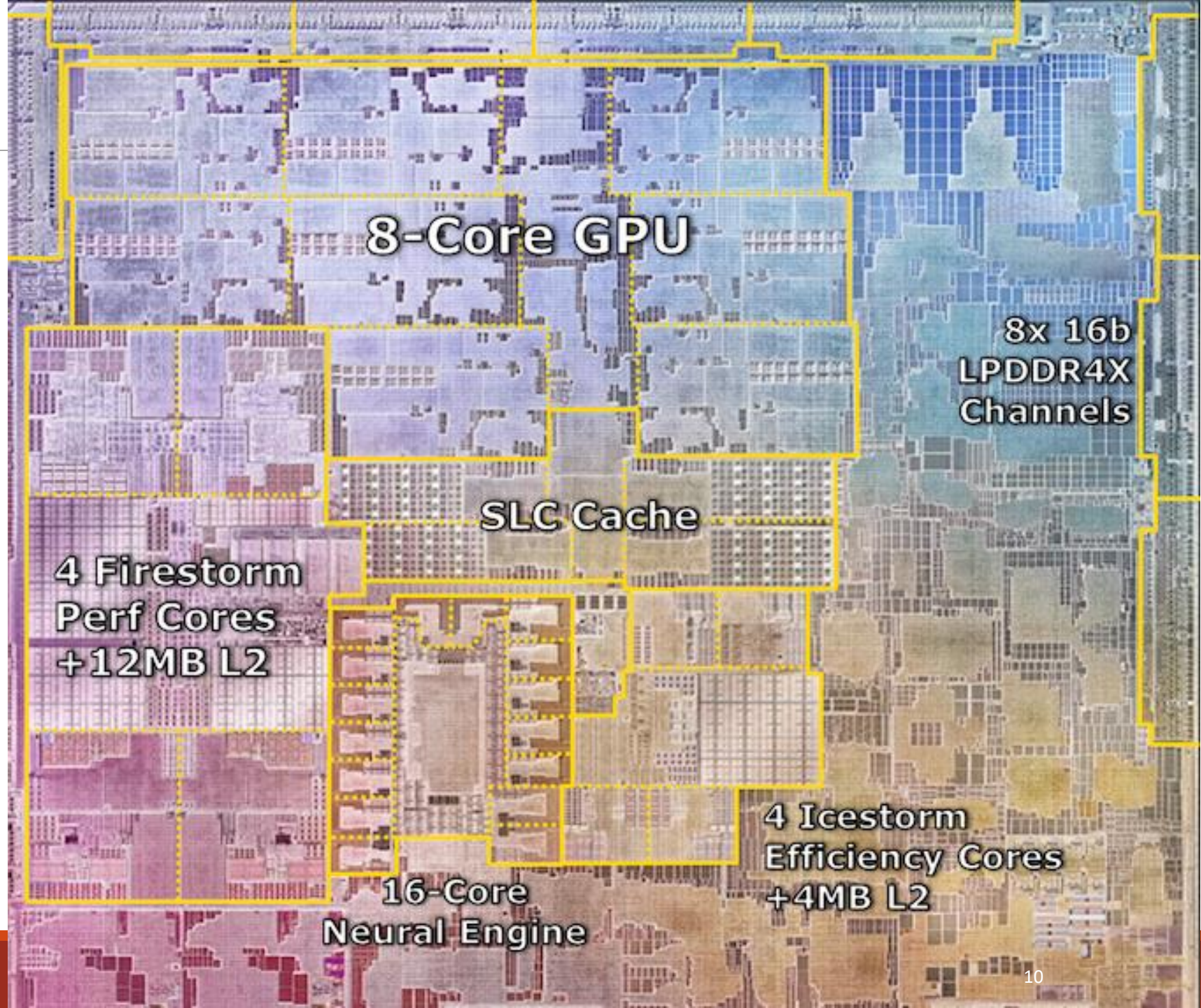
Apple M1 (‘20)

Newly released ARM chip for laptops

Basically a turbo-charged phone SoC

CPUs + GPU + accelerators

Big emphasis on “unified memory”



Multicore design issues

Multicore is all about **scalability**

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

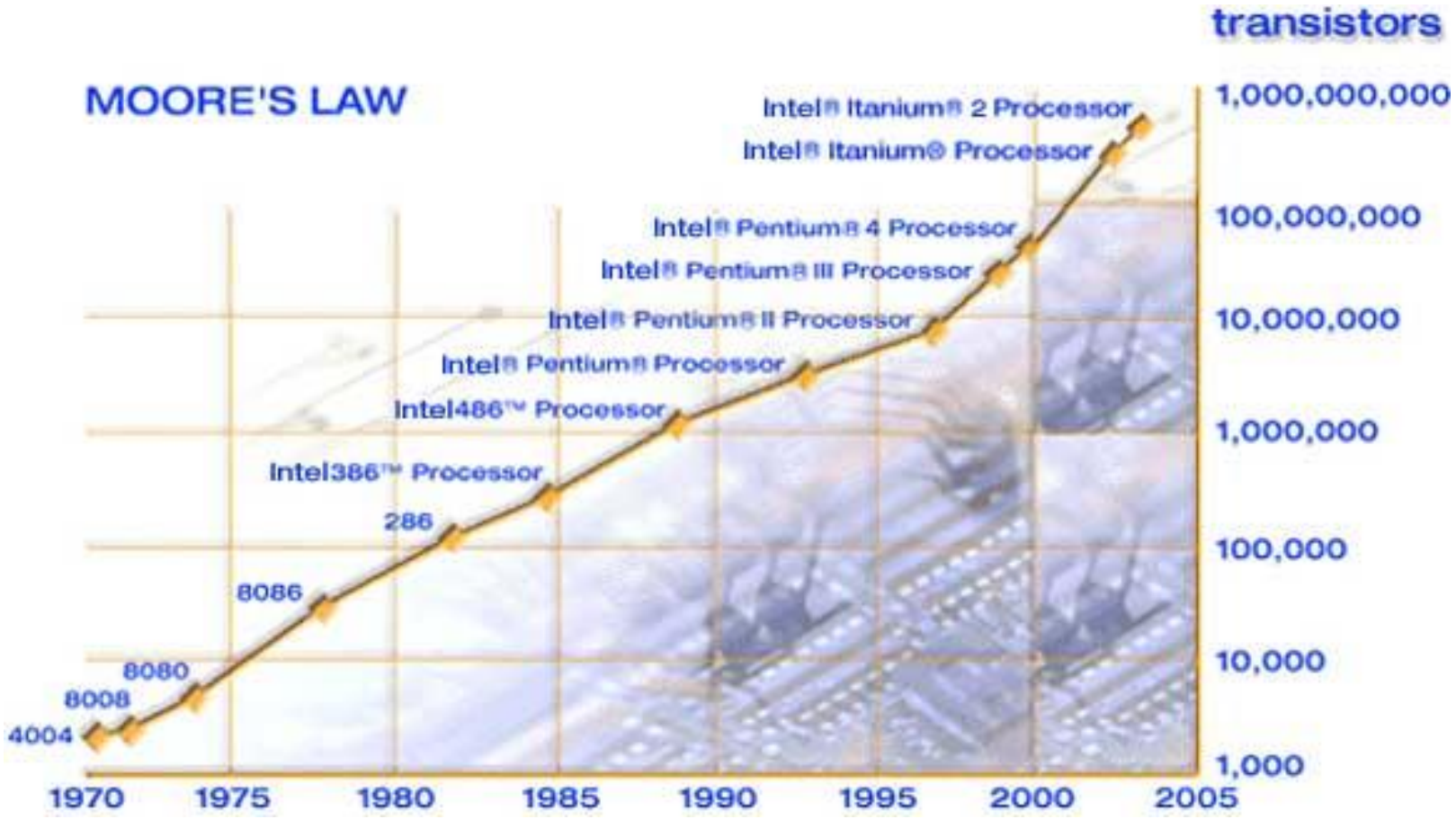
New design challenge: Coordination btwn cores via *on-chip interconnect*

What doesn't scale?

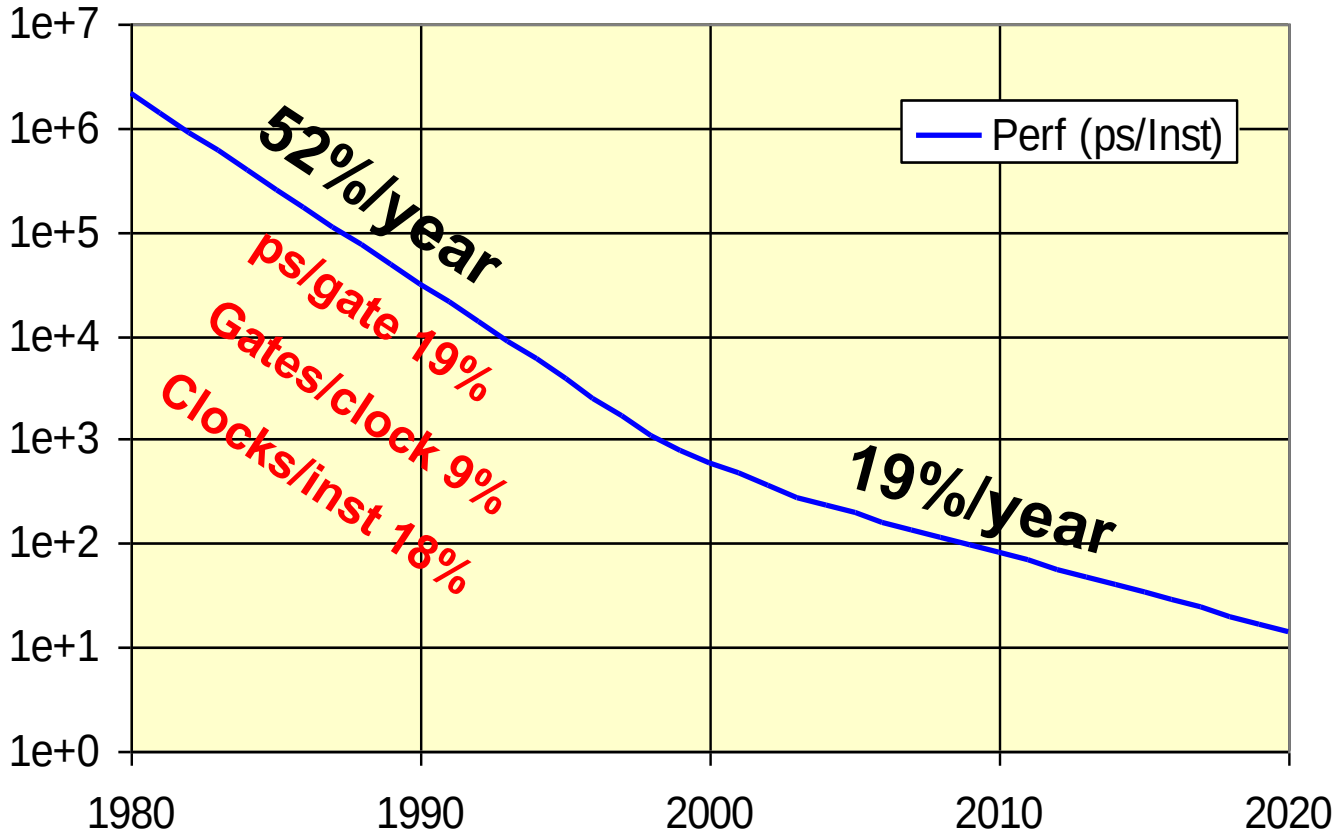
- On-chip communication (network distance & contention)
- Off-chip communication (limited by pins)
- Power/thermal dissipation (same physical object)

The case for multicore

Moore's Law


MOORE'S LAW
transistors
Intel® Itanium® 2 Processor
Intel® Itanium® Processor
Intel® Pentium® 4 Processor
Intel® Pentium® III Processor
Intel® Pentium® II Processor
Intel® Pentium® Processor
Intel486™ Processor
Intel386™ Processor
286
8086
8080
8008
4004
1,000,000,000
100,000,000
10,000,000
1,000,000
100,000
10,000
1,000
1970
1975
1980
1985
1990
1995
2000
2005

50% / year performance scaling stops

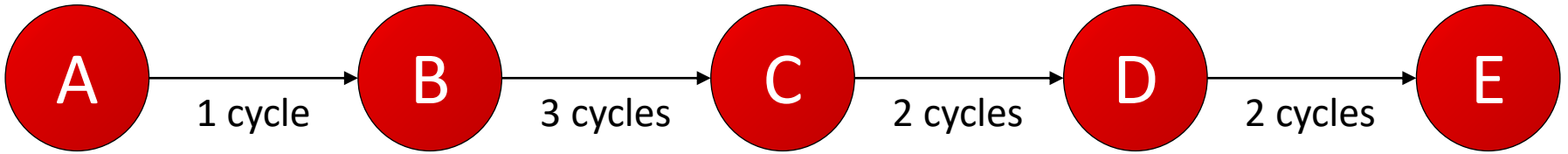


Bill Dally

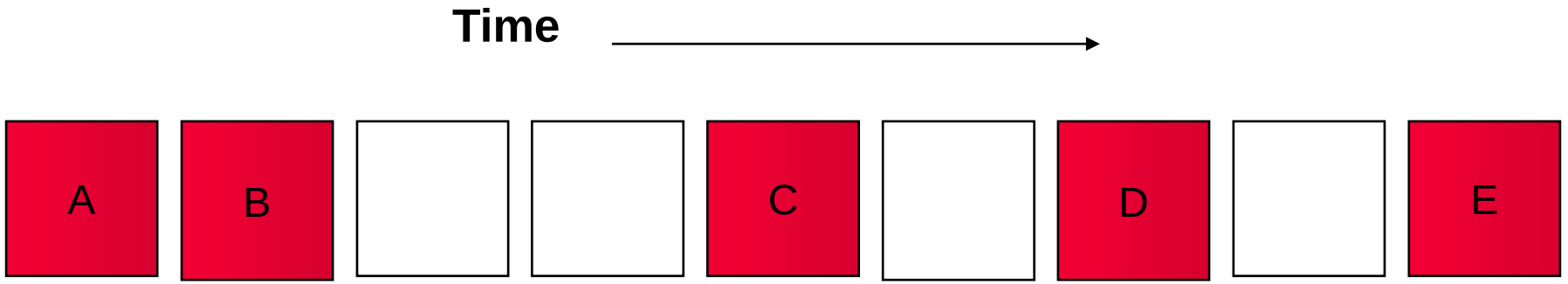
Why not bigger cores?

AKA “SUPERSCALAR”

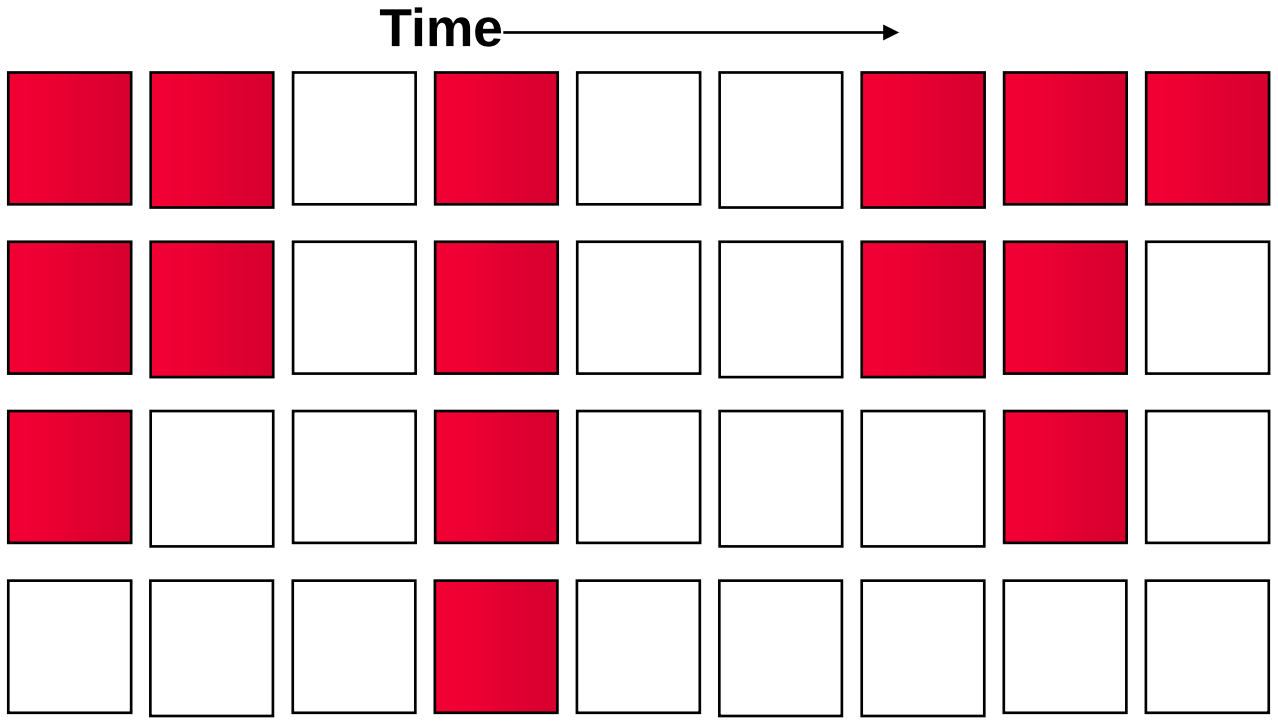
Functional Unit Utilization



Data dependencies reduce functional unit utilization in pipelined processors



Functional Unit Utilization in Superscalar

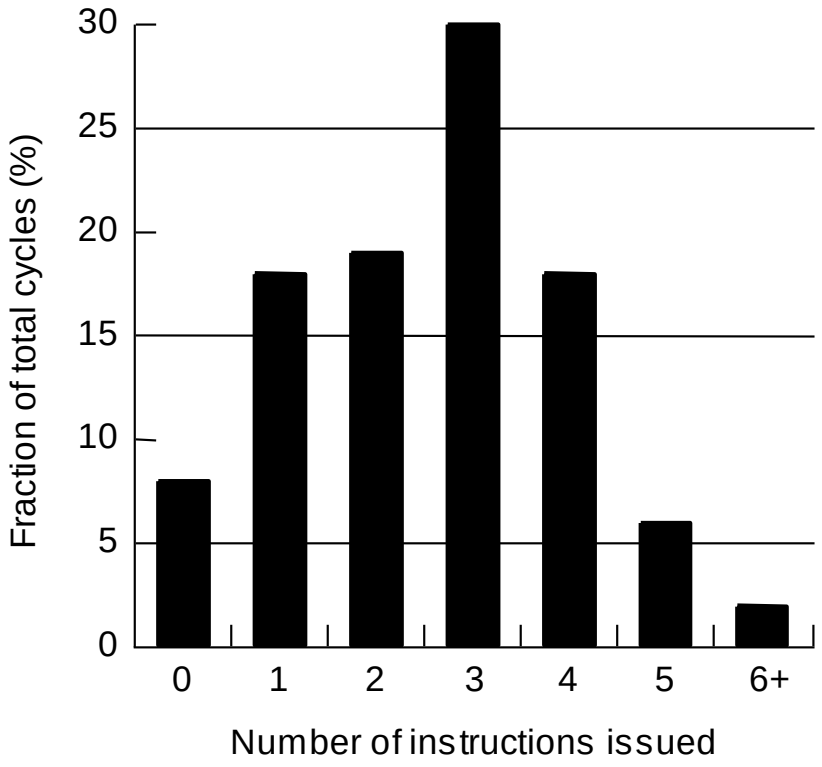


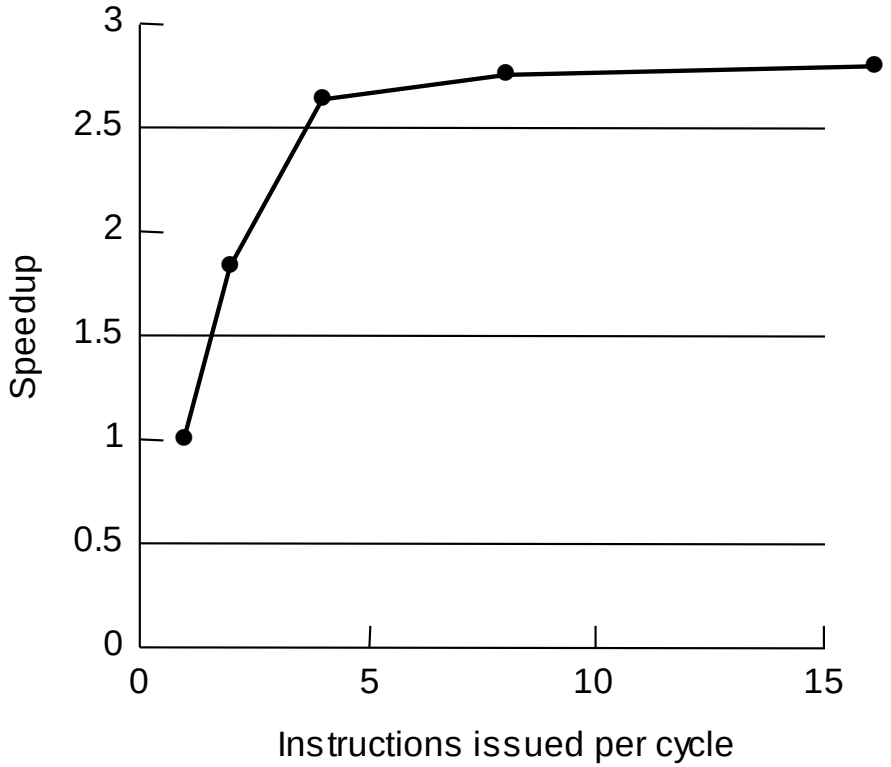
Functional unit utilization becomes lower in superscalar, OoO machines. Finding 4 instructions in parallel is not always possible

➔ Superscalar has *utilization* ≪ 1 (see definition in slides on parallelism)

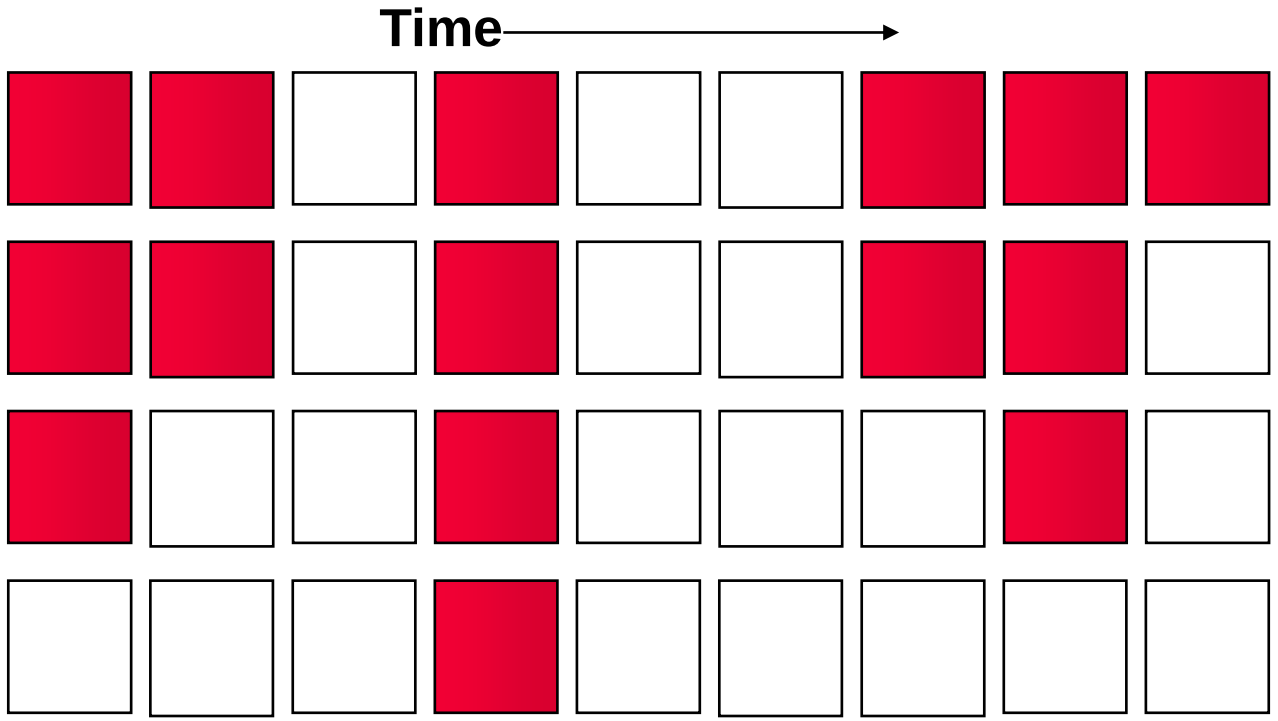
Limits to instruction-level parallelism

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





Utilization in Superscalar Core



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

➔ Superscalar has *utilization* ≪ 1 (see definition in slides on parallelism)

Multicore

Idea: Put multiple processors on the same die.

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

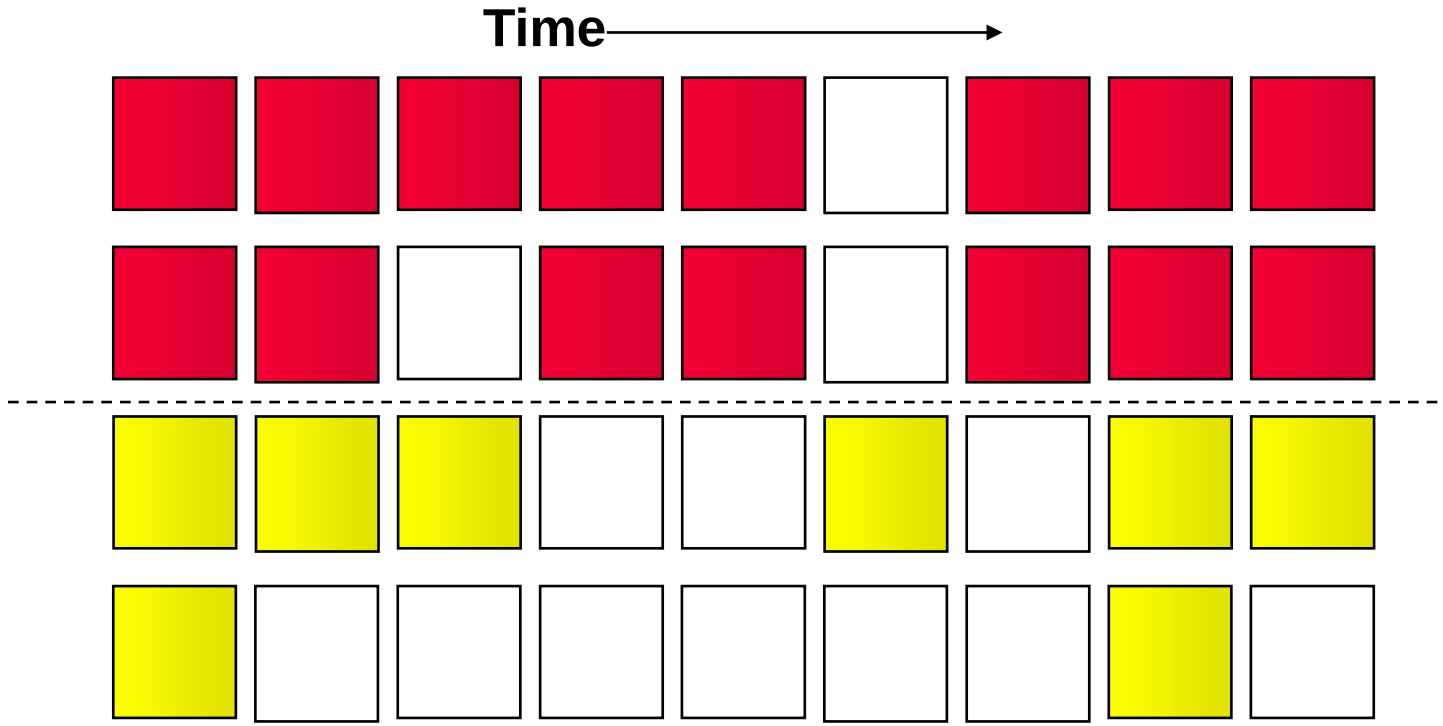
Technology scaling (Moore's Law) enables more transistors to be placed on the same die area

But there was **heavy resistance** to move to multicore

- Architects had been delivering 50% improved performance **without software changes** for decades
- Multicore require **massive software investment** to be successful
- ➔ Parallel programming no longer a niche area for scientific computing

Let's go through the alternatives…

Multicore Improves FU Utilization



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.

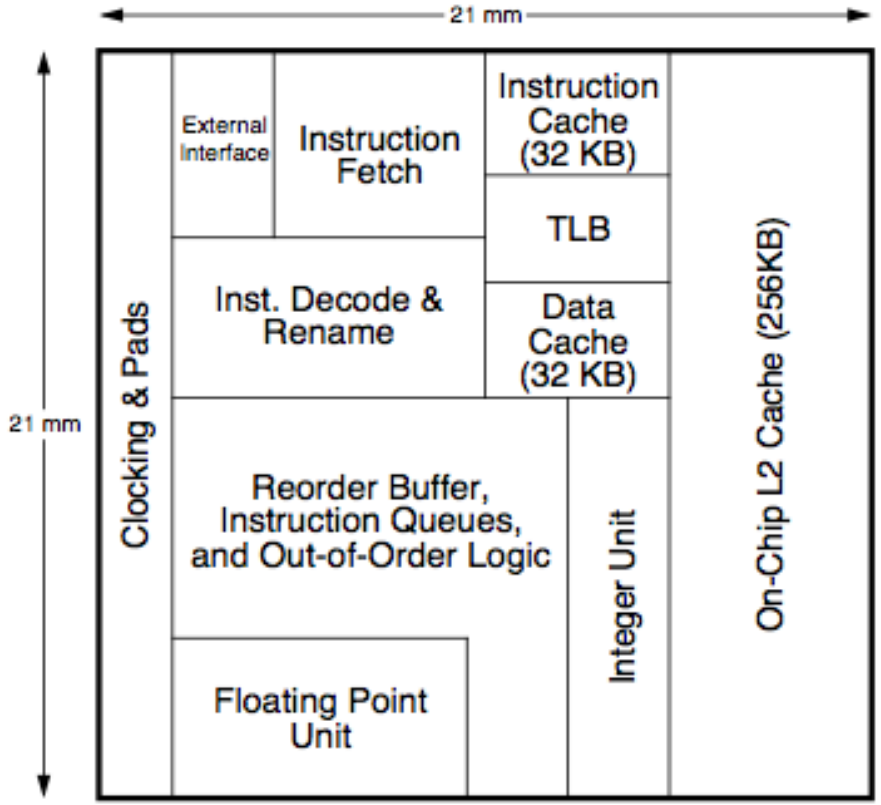


Figure 2. Floorplan for the six-issue dynamic superscalar microprocessor.

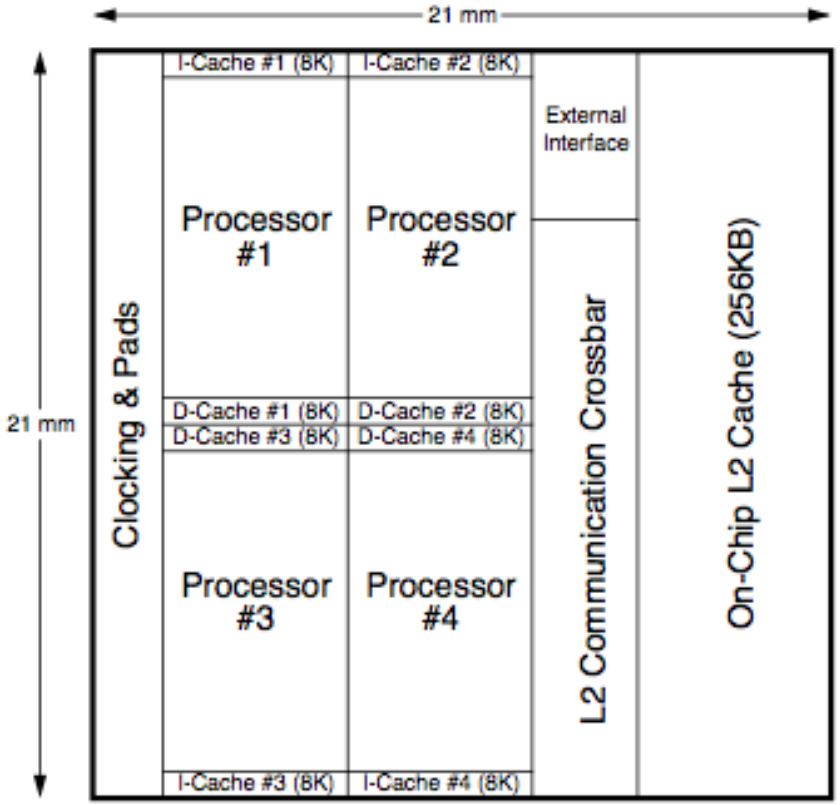


Figure 3. Floorplan for the four-way single-chip multiprocessor.

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

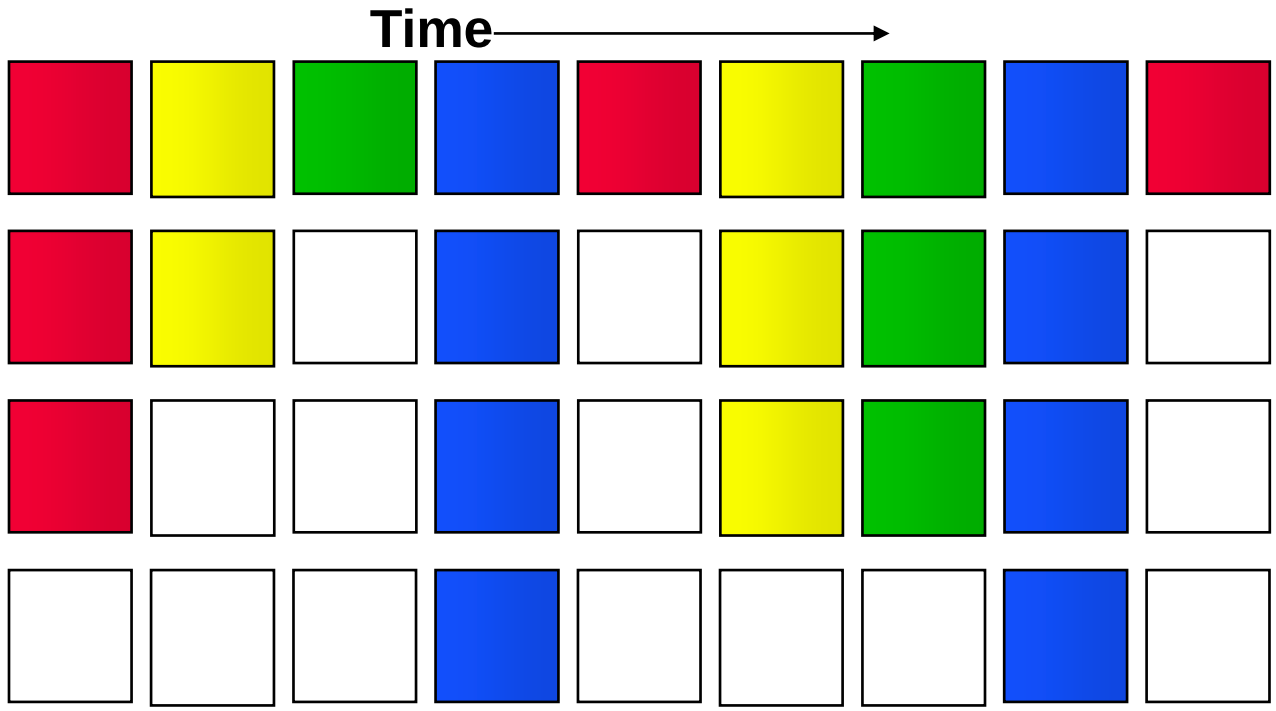
Multi-core disadvantages

- 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 multithreading?

Fine-grained Multithreading



Idea: Time-multiplex execution units across threads

Hides latency of long operations, improving utilization

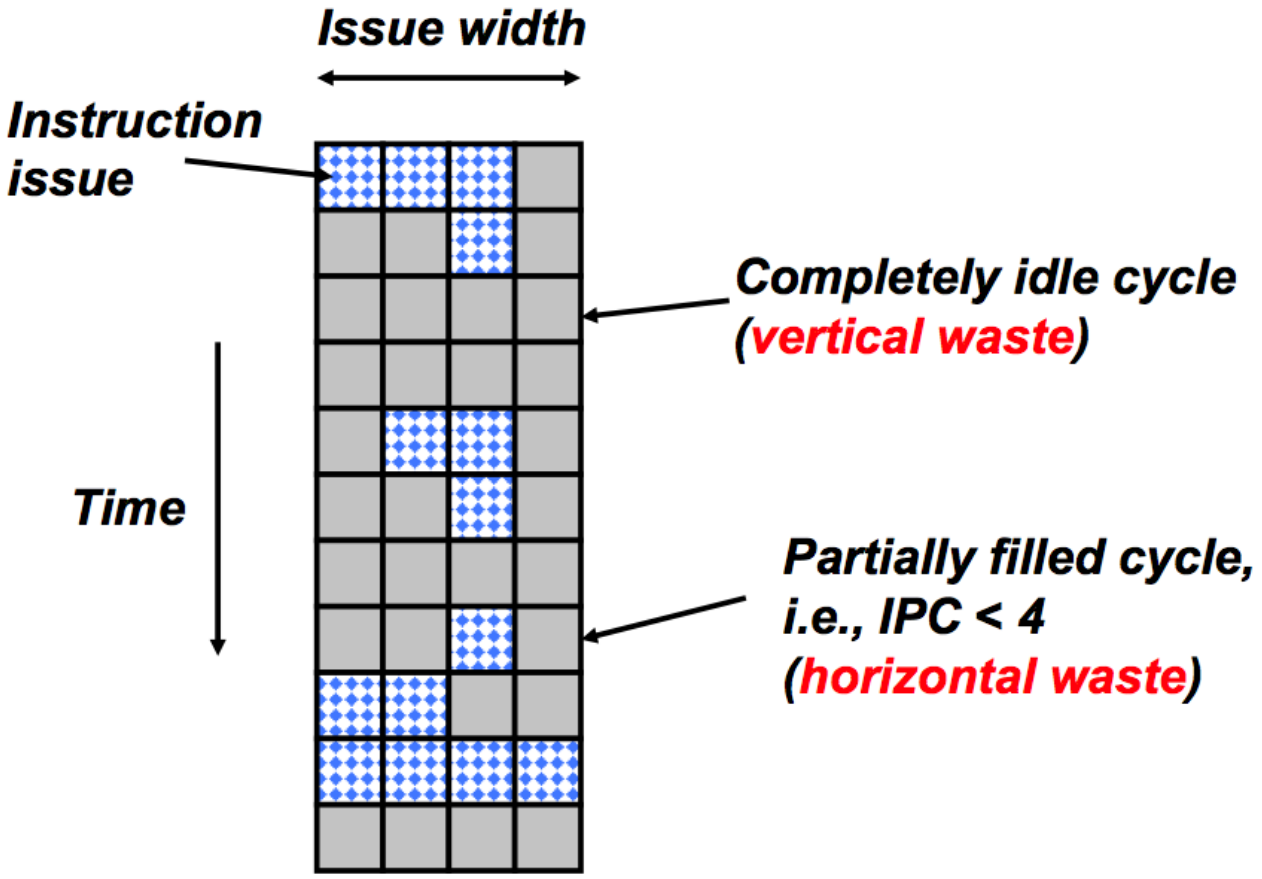
…But single thread performance suffers (in naïve versions)

Horizontal vs. Vertical Waste

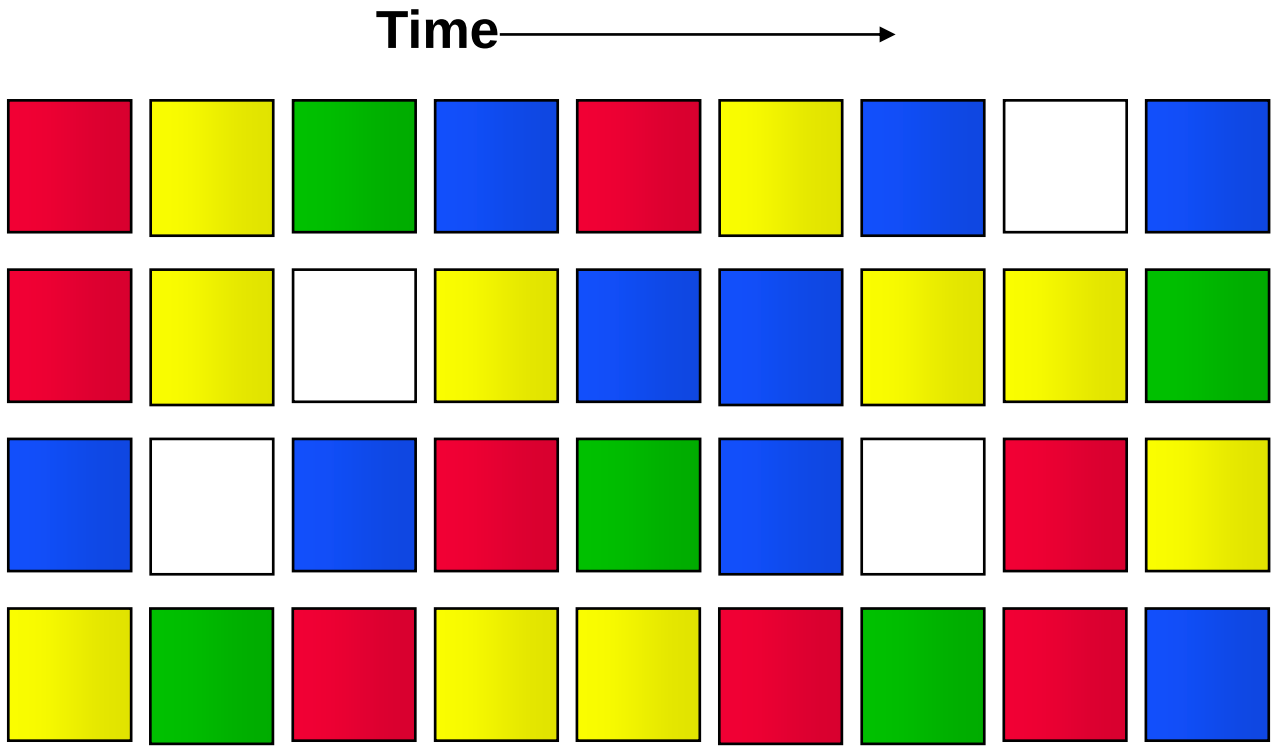
What causes horizontal waste?

vertical waste?

How do you reduce each?



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**
+ Efficient: Don't need an entire core for another thread
+ Communication faster through shared L1 caches (SAS model)

– Limited scalability: still need…
  - Bigger register files
  - More functional units
  - Larger issue width (and associated costs)
– Parallel performance limited by shared fetch bandwidth
– Extensive resource sharing (pipeline and memory system) reduces both seq & par performance

Why Not …?

Why not bigger caches?

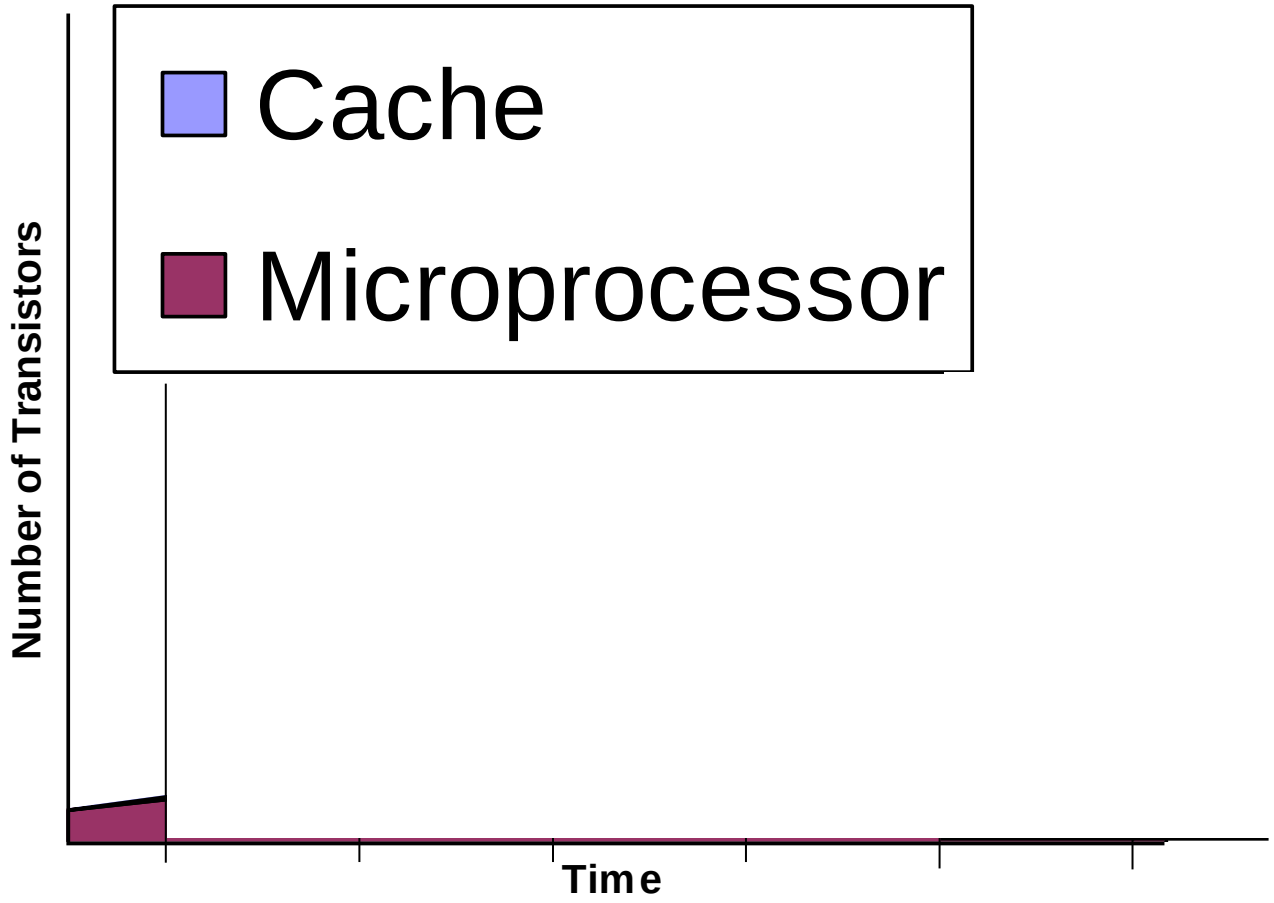
+ Improves single-thread performance transparently to programmer, compiler

+ Simple to design

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

– Multiple levels complicate memory hierarchy

Area for Cache vs. Core



Why Not System on a Chip?

Alternative: Integrate platform components on chip instead

\+ Speeds up many system functions (e.g., NIC, memory controller, I/O, etc.)

– Few applications benefit (e.g., CPU intensive code sections)

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

More scalable sequential cores?

& 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)

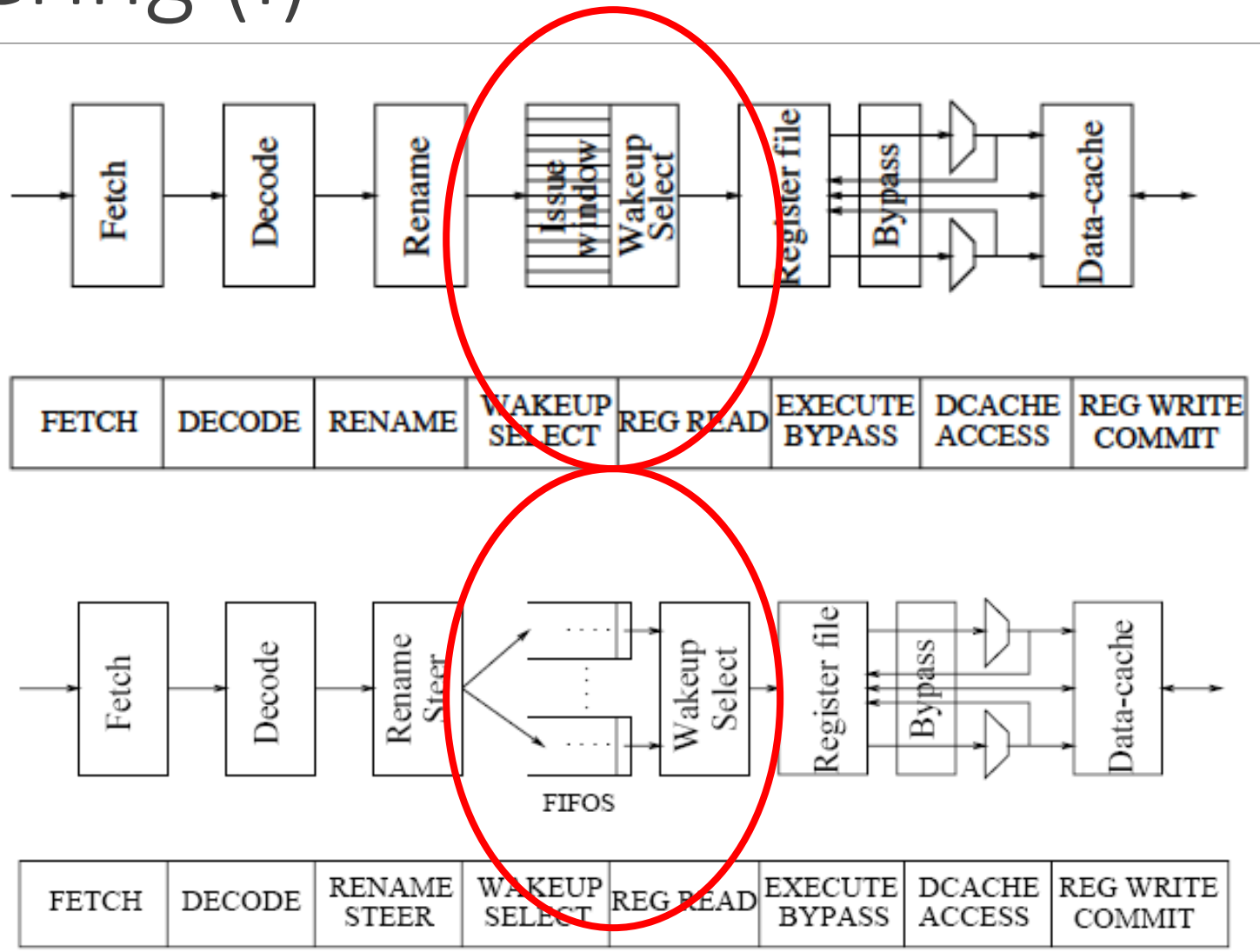
+ Helps scalability of monolithic OOO: Smaller scheduling windows, simpler wakeup algorithms
+ Fewer ports into register files
+ Faster within-cluster bypass

- Extra delay when instructions require across-cluster communication
- Inherent difficulty of steering logic

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

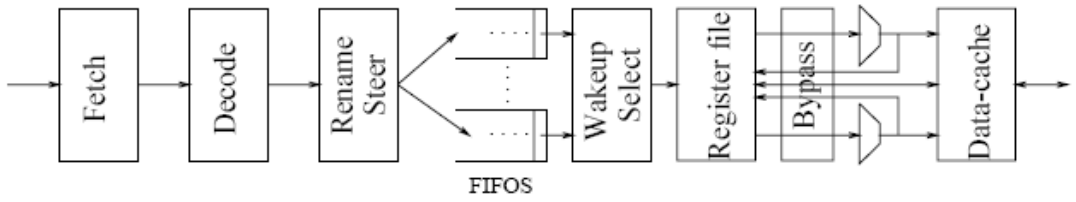
Clustering (I)

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

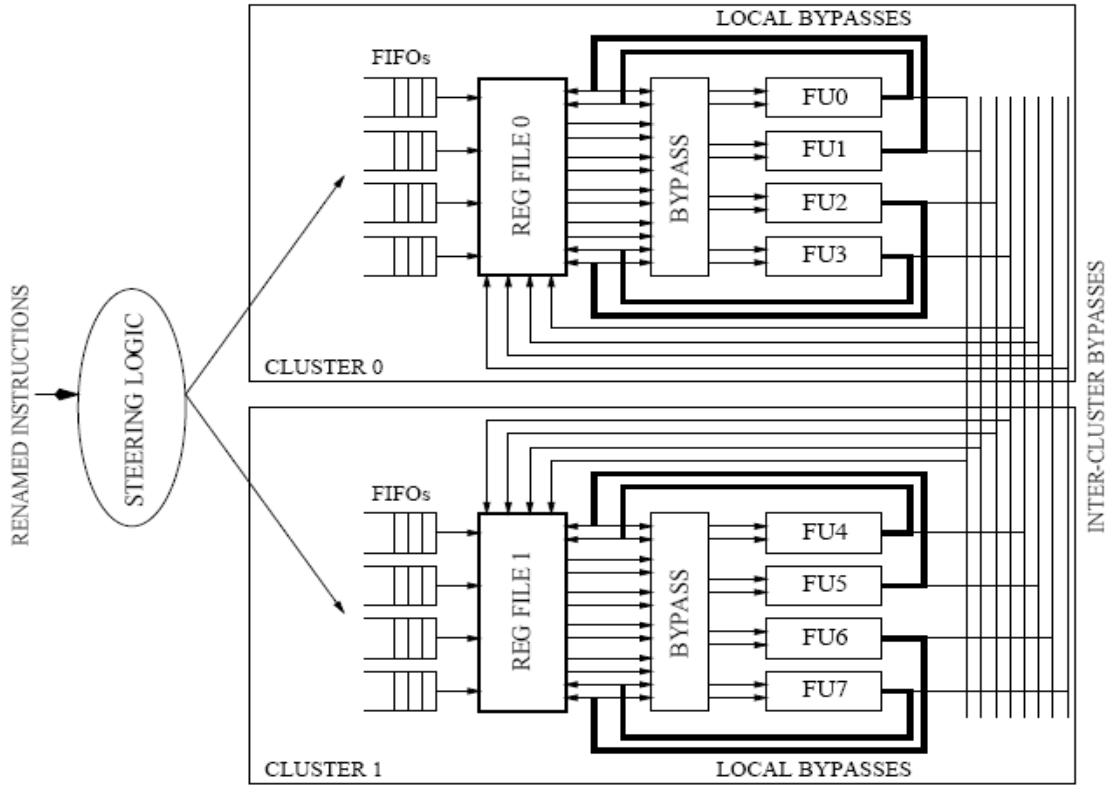


Clustering (II)

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



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

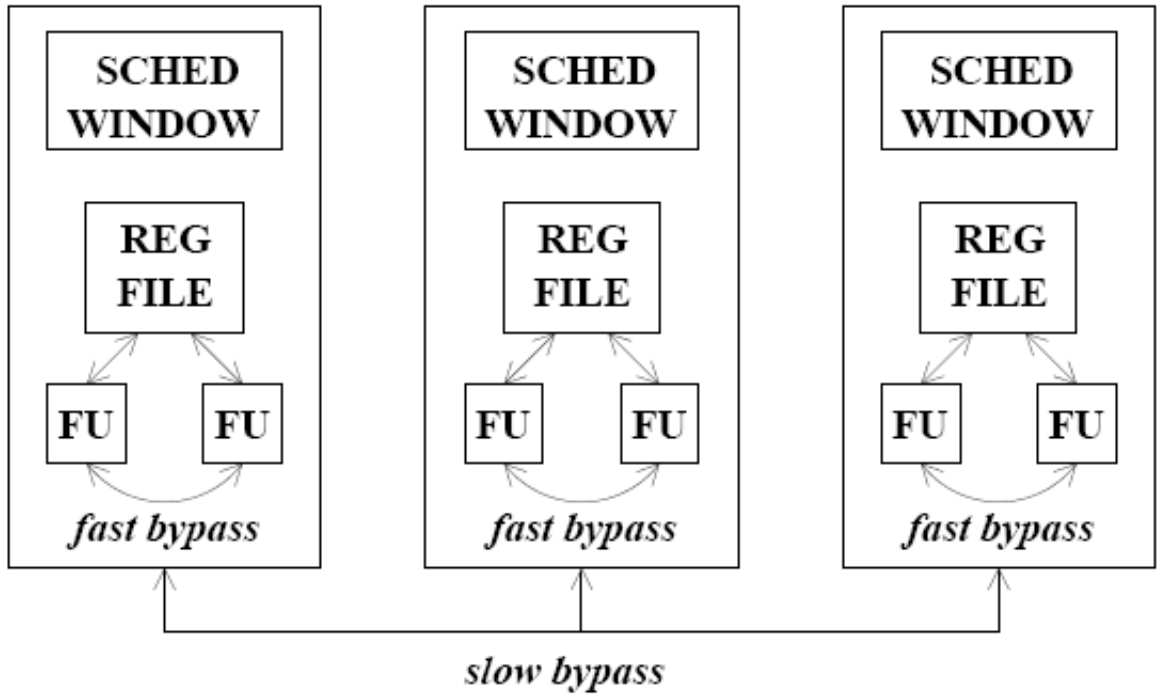


Each scheduler is a FIFO
+ Simpler
+ Can have N FIFOs
  (OoO w.r.t. each other)
+ Reduces scheduling complexity
- More dispatch stalls

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.

Other side of the coin: Core Fusion

[Ipek+, ISCA'07]

Idea: Build chip with many small cores, dynamically combine them into bigger cores as needed

If you have a multicore already, why do this?

- Legacy software is not parallel – will take any speedup you can get
- Amdahl's law – need to accelerate sequential region

Many complex design issues:

- How to coordinate fetch & commit across cores?
- Solution: Centralized "management units"
- …But these have high latency + add buffering

Not as fast as a monolithic core!

- 6-wide monolithic: 73% faster than 2-wide
- 8-wide fused: 50% faster than 2-wide

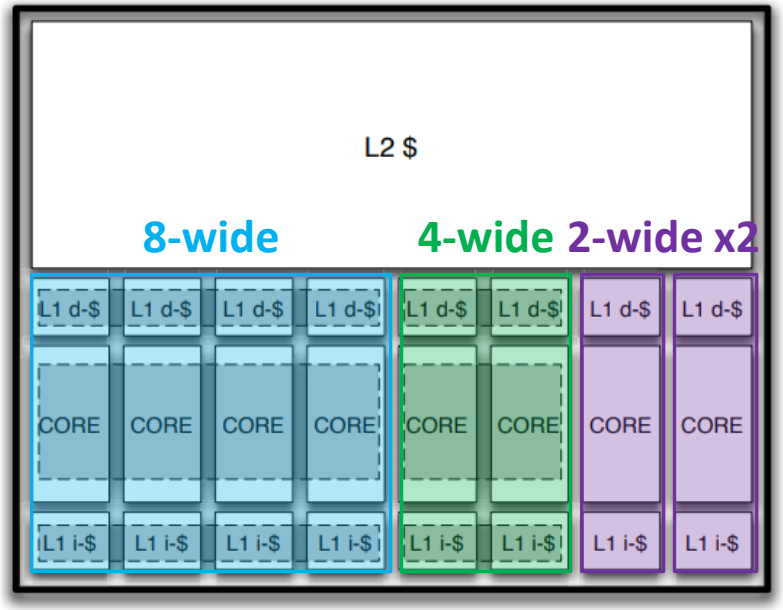


Figure 1: Conceptual floorplan of an eight-core CMP with core fusion capability. The figure shows a configuration example comprising two independent cores, a two-core fused group, and a four-core fused group. The figure is not meant to represent an actual floorplan.

Why not clustering / fusion?

+ 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 compared to monolithic superscalar.

– Parallel performance limited by shared fetch bandwidth

(Sequential program ordering makes this fundamentally very hard to scale)

– Difficult to design

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?

- Compilation/tooling problems, doesn't help memory-bound apps

Why Multi-Core (Optimistically)

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 → multicore scales performance w/out blowing up power

Design complexity is very costly → multicore simpler to design

Still need good sequential performance (Amdahl's Law) → makes sense to keep OoO cores

Also…

- Huge investment & need ROI ➔ Must offer some upgrade path
- Easy option for processor manufacturers (and for software too, up to a point)

Multicore design

Granularity

How many cores should we have?

Many simple cores:

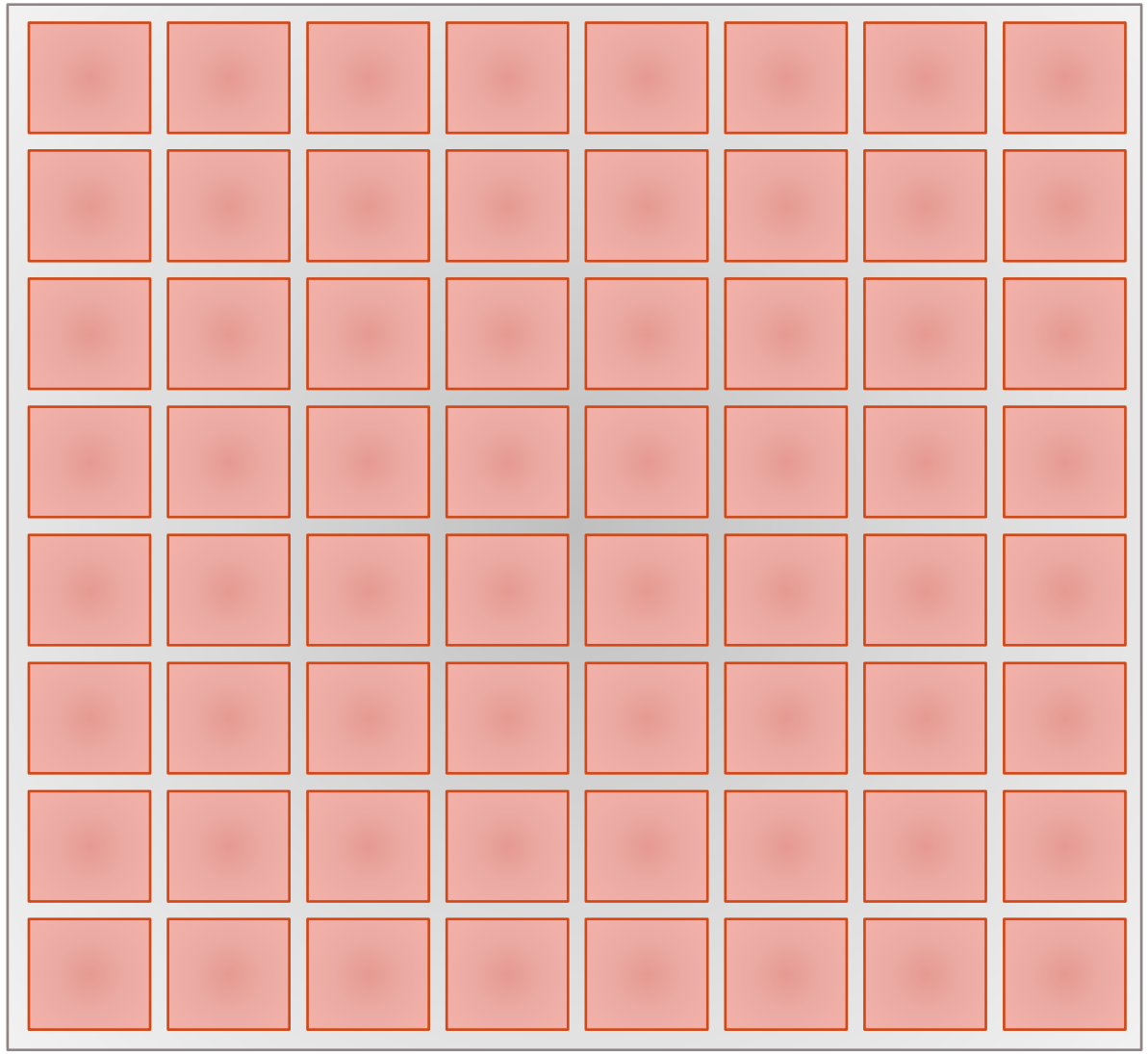
- Better parallel performance
- Better area & energy efficiency

How big should each be?

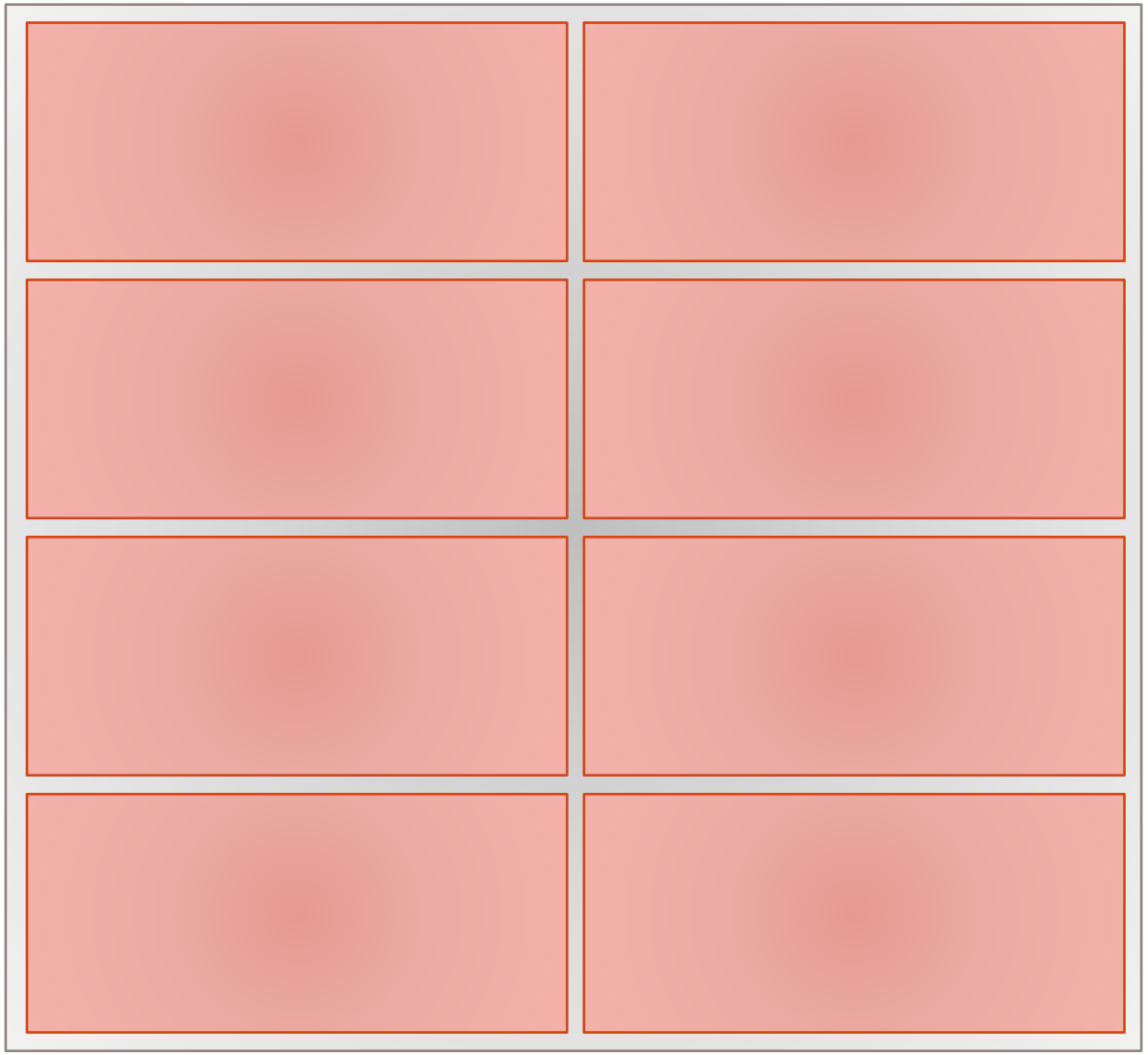
Few big cores:

- Better sequential performance
- Lower communication cost (?)

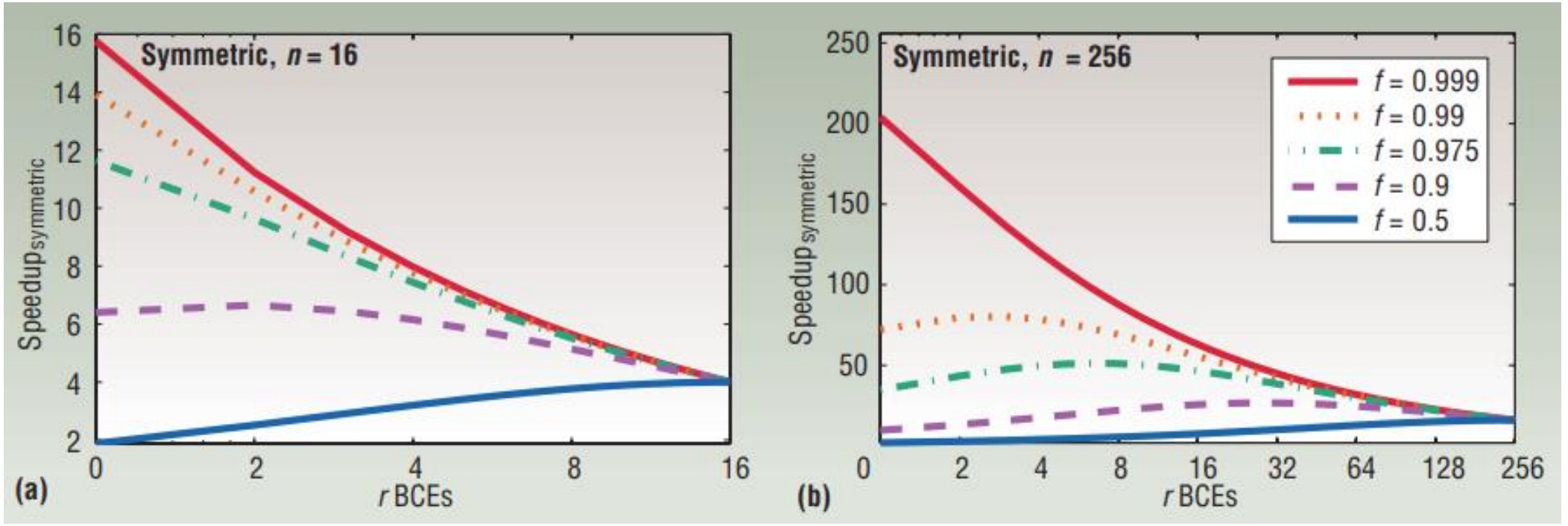
Many small cores



Few big cores

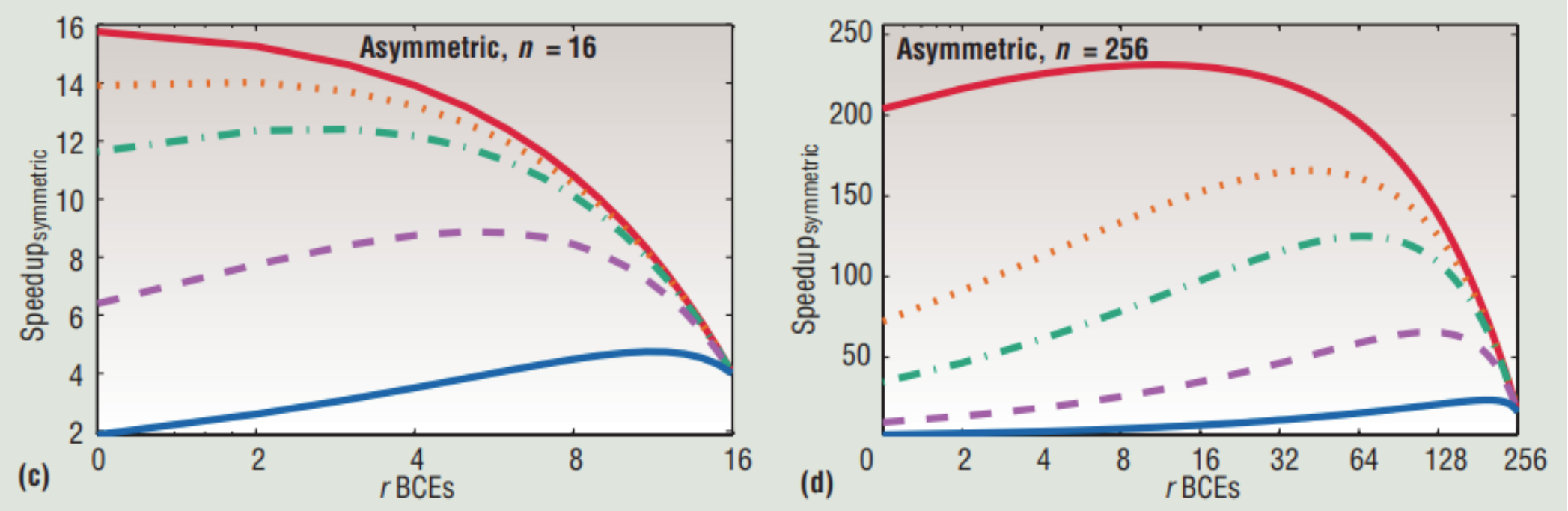


Symmetric perf scaling



Amdahl's Law in the Multicore Era, Mark Hill '07

Asymmetric perf scaling



Amdahl's Law in the Multicore Era, Mark Hill '07

Granularity

How many cores should we have?

Many simple cores:

- Better parallel performance
- Better area & energy efficiency

Best solution?

- Typically, some of both
- Good sequential performance on sequential apps / code regions
- Good parallel performance on parallel codes

How big should each be?

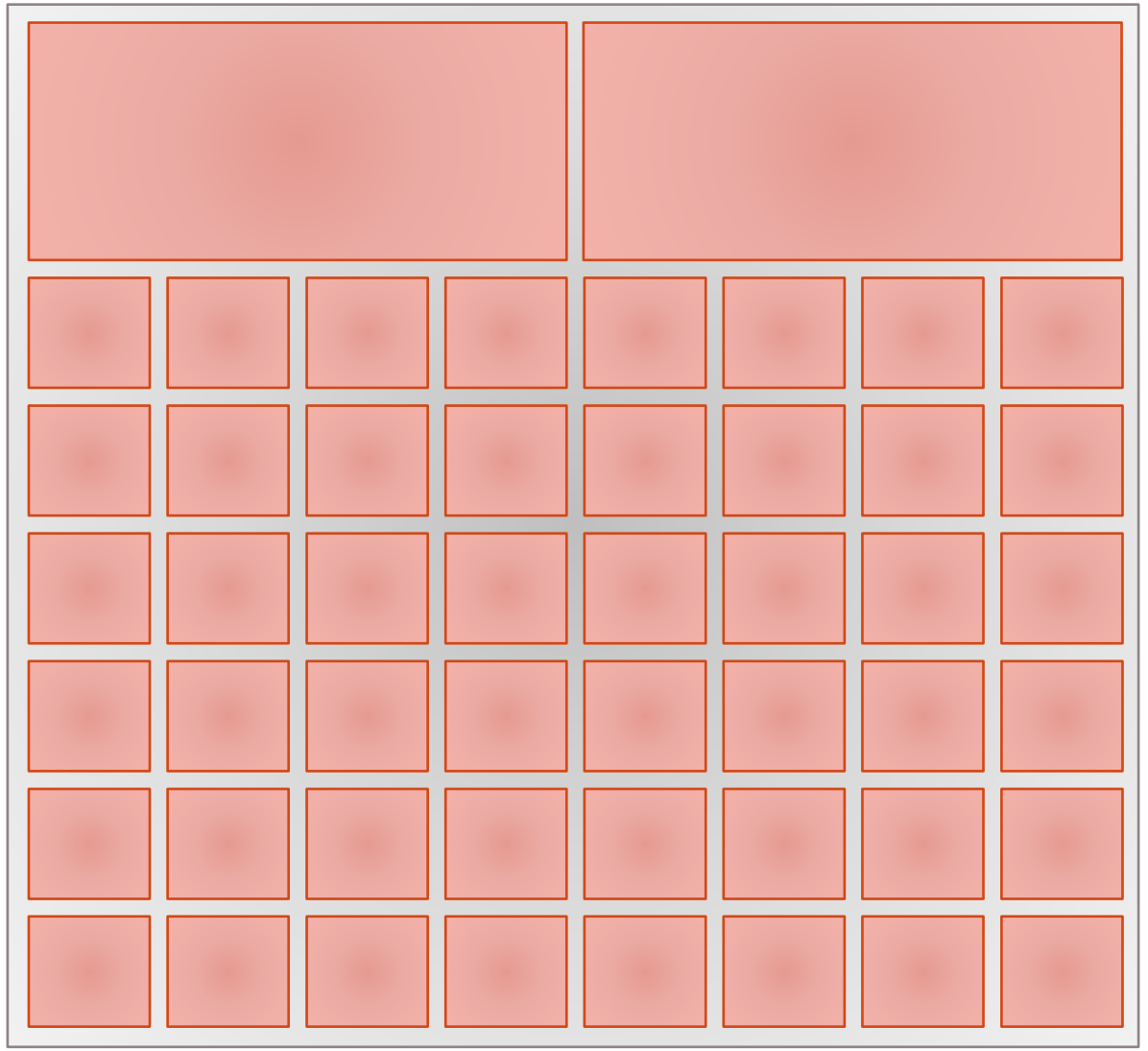
Few big cores:

- Better sequential performance
- Lower communication cost (?)

Many further issues

- Should cores use the same ISA?
- When to schedule/migrate threads?
- What about load imbalance/fairness?
- Etc.

Heterogeneous multicore (big.LITTLE)



Dynamic heterogeneity

Can we get a similar effect without statically building different types of cores?

Yes! through **dynamic voltage and frequency scaling** (DVFS)

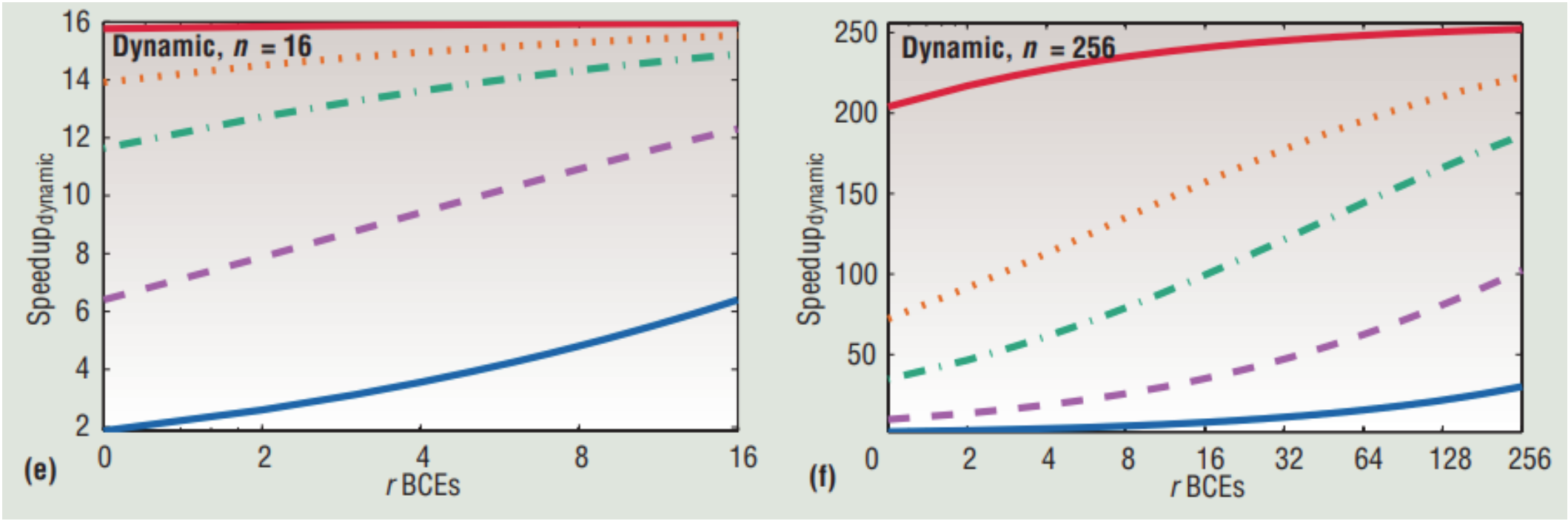
- Boost frequency of one/few cores → better performance (but sub-linear perf-energy tradeoff)
- Other cores must run at reduced frequency & local heat build-up may force DVFS to be temporary

Logic: *If power forced us to do multicore, can we dynamically allocate the power budget among cores to get the right balance of performance?*

Not identical to true heterogeneity: Higher frequency != wider-issue OOO

Key challenges: when to boost? how much to boost?

Dynamic perf scaling

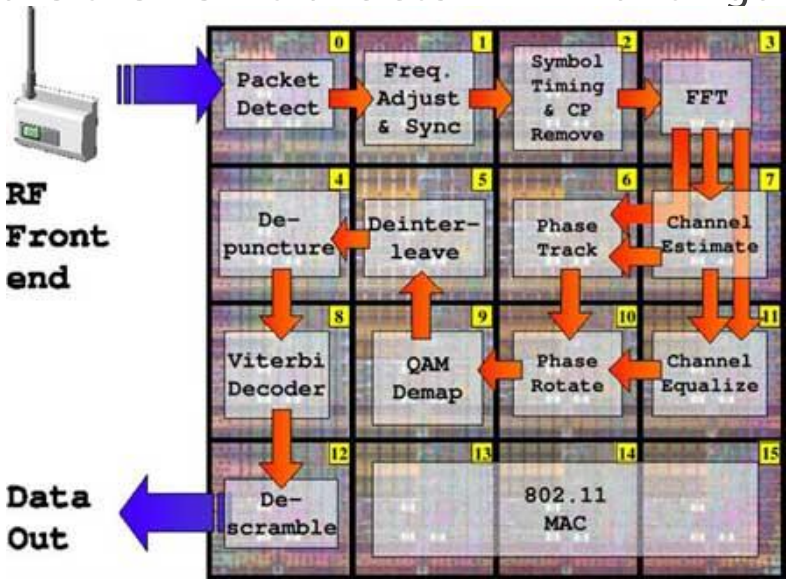


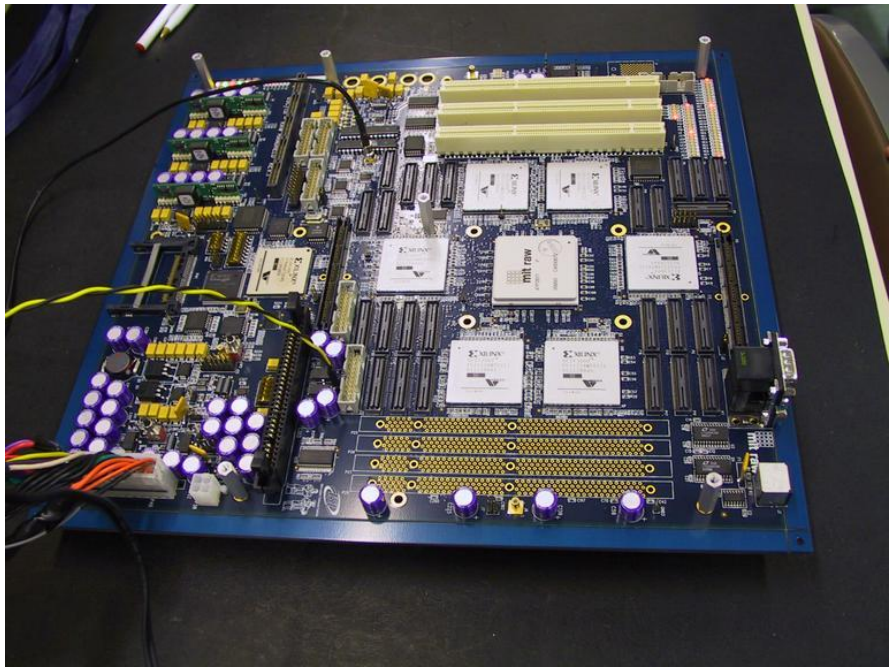
Amdahl's Law in the Multicore Era, Mark Hill '07

Anant Agarwal, Saman Amarsinghe @ MIT

Crazy Idea (1/2): MIT RAW

"Cores are the new transistor." – Anant Agarwal





Many small cores + programmable network routers

- Data can go directly from network into core pipeline
- Raw chips composable in multi-chip systems to form giant multicore fabric

Compiler automatically distributed sequential code across cores & routers

- In practice, compiler had a hard time disambiguating memory (still unsolved)

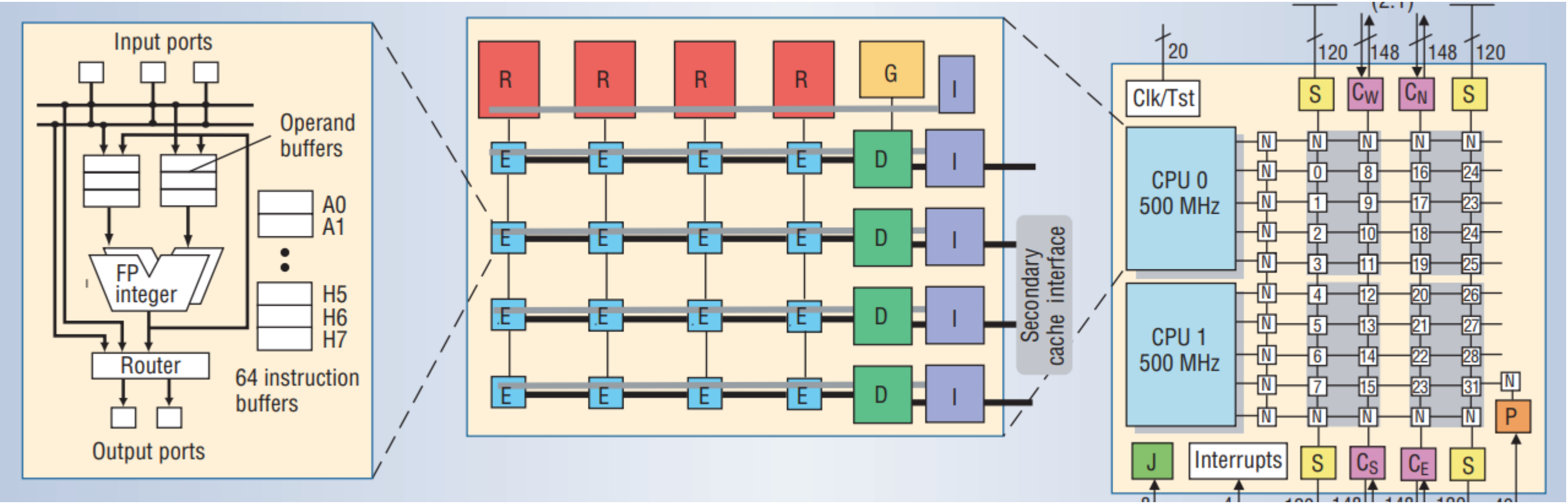
Crazy Idea (2/2): TRIPS

Steve Keckler, Doug Burger @ UT Austin

EDGE (explicit data-graph execution):

- Compile a static, spatial dataflow across an array of very simple processing elements

TRIPS implements multiple EDGE cores



Network-on-chip (NoC): Scaling communication in multicores

Communication is major challenge

Memory-processor gap continues to grow

- Multicore makes this worse b/c more cores ➔ more bandwidth demand

Multicores must adjudicate communication on-chip & off-chip

- Typically, coherence protocol drives on-chip communication

Cache & on-chip network architecture determine communication efficiency

- Cache design determines where data will be on-chip & how much off-chip traffic
- On-chip network determines how fast, efficient communication is

Resource sharing in multicore

Resource sharing & interference

Cores share many resources & compete with each other

- Power
- Off-chip memory bandwidth
- Shared caches
- On-chip network

Misbehaving applications can degrade performance, e.g.:

- Tight loop that burns tons of power
- Frequent off-chip memory accesses
- Thrashing on-chip working set
- …

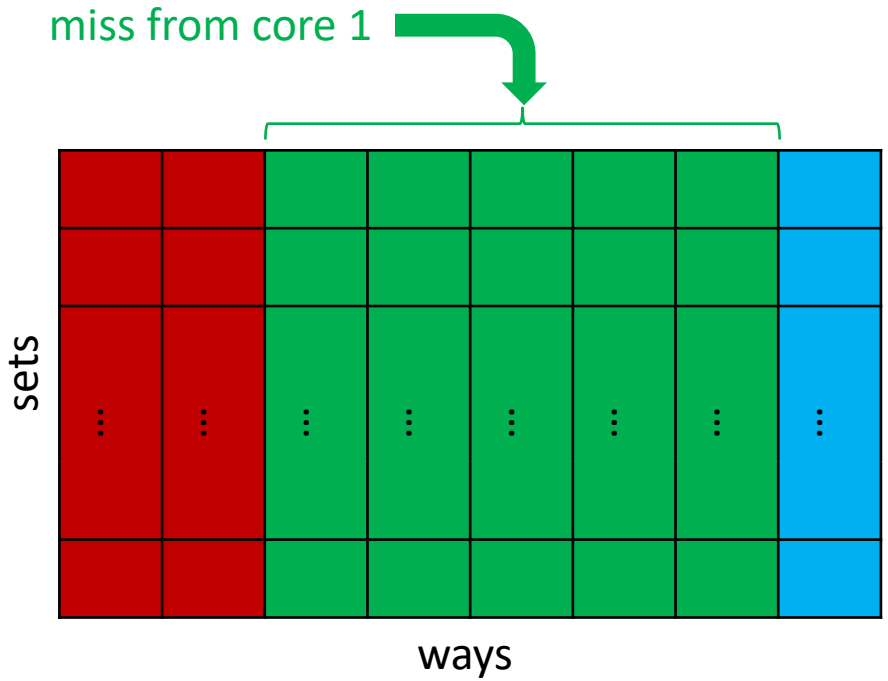
Multicore requires mechanisms to *partition* resources among cores + applications

Utility-based cache partitioning

[Qureshi+, MICRO'06]

Allocate cache ways among cores (*way partitioning*)

- E.g., **core 1** is only allowed to evict objects in ways 2-6
- Lookups unchanged: coherence maintained trivially



How many ways to allocate to each core?

- Applications differ in how well they use memory
- Idea: Use per-core *miss curves*
- Divide ways to minimize *total misses across cores*

UMONs: Simple hardware to gather miss curves using set-sampling

Lookahead: Greedy local-search algorithm to partition ways

Multicore summary

Multicore addresses scaling limitations of sequential arch by pushing problems to software

- No obvious architectural alternative

Many new design issues are introduced

- E.g., grain size, network on-chip, resource sharing & contention

It's been ~twenty years, has it worked out?

- Mixed success at best – multicore did not offer long-term app performance scaling
- Architectural problems can be overcome, but software challenges remain
- Many (most?) apps don't use multicore well
- Other architectures dominate highly-parallel apps (e.g., GPU)
- Multicore does not address Amdahl's Law – sequential perf still matters!

Many architects believe specialization is the answer, and no longer expect to see hundreds or thousands of MIMD cores in one CMP

- Specialization is even more disruptive across the stack
- ➔ **Very exciting time to study computer architecture!**

Self-check questions

Why did multicore become commonplace in the early 2000s?

What were the key challenges in scaling sequential performance?

What were (technical) reasons that the Sun Niagara processor was not successful? What modern architecture adopts a similar strategy with great success?