Lecture 20
Wrapup

EEC 171 Parallel Architectures

John Owens
UC Davis

Credits

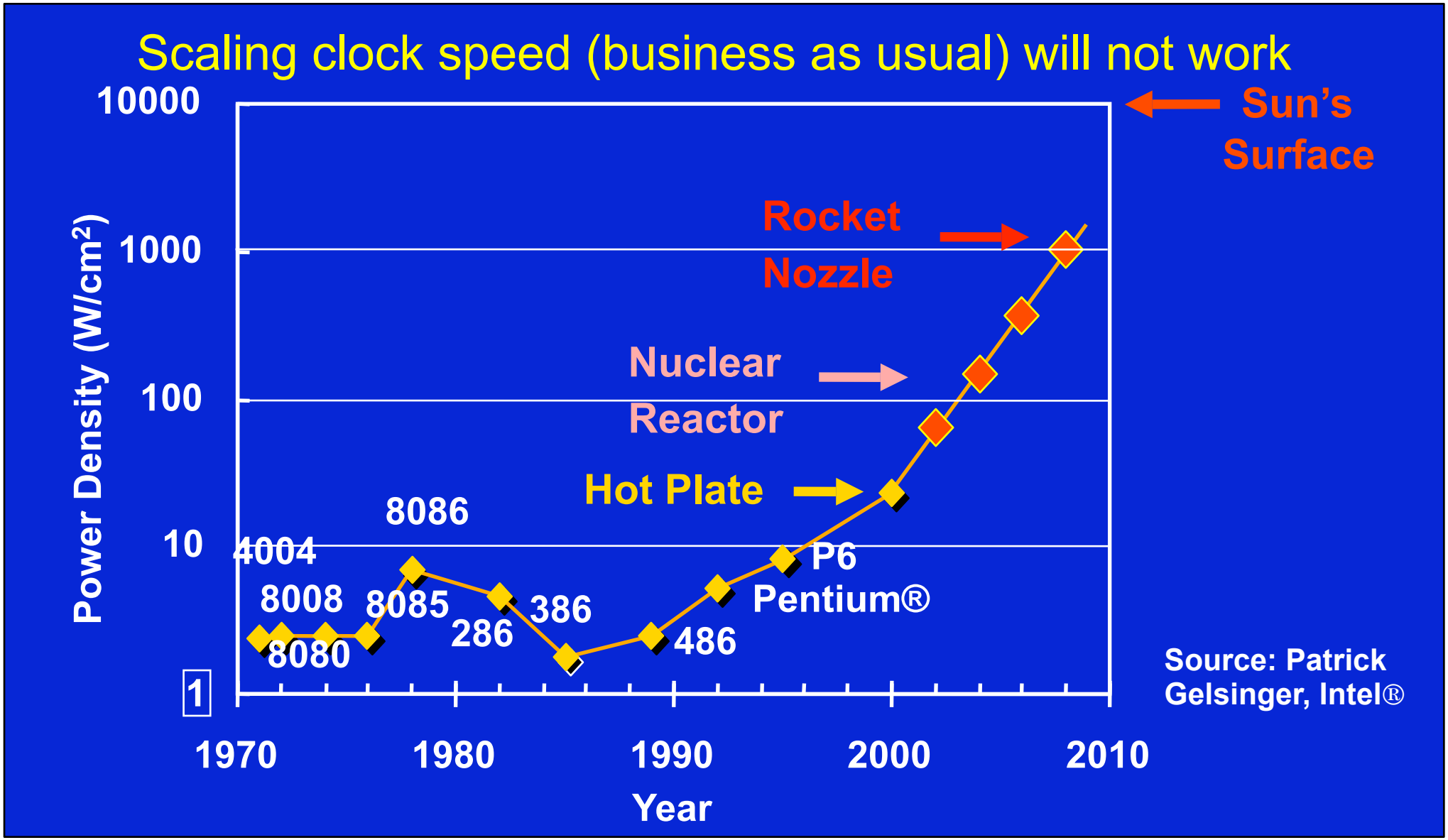
- © John Owens / UC Davis 2007–9.
- Thanks to many sources for slide material: Computer Organization and Design (Patterson & Hennessy) © 2005, Computer Architecture (Hennessy & Patterson) © 2007, Inside the Machine (Jon Stokes) © 2007, © Dan Connors / University of Colorado 2007, © Kathy Yelick / UCB 2007, © Wen-Mei Hwu/David Kirk, University of Illinois 2007, © David Patterson / UCB 2003–7, © John Lazzaro / UCB 2006, © Mary Jane Irwin / Penn State 2005, © John Kubiatowicz / UCB 2002, © Krste Asinovic/Arvind / MIT 2002, © Morgan Kaufmann Publishers 1998.

Also thanks to …

- Kathy Yelick, for her slides on “A Berkeley View on the Parallel Computing Landscape”

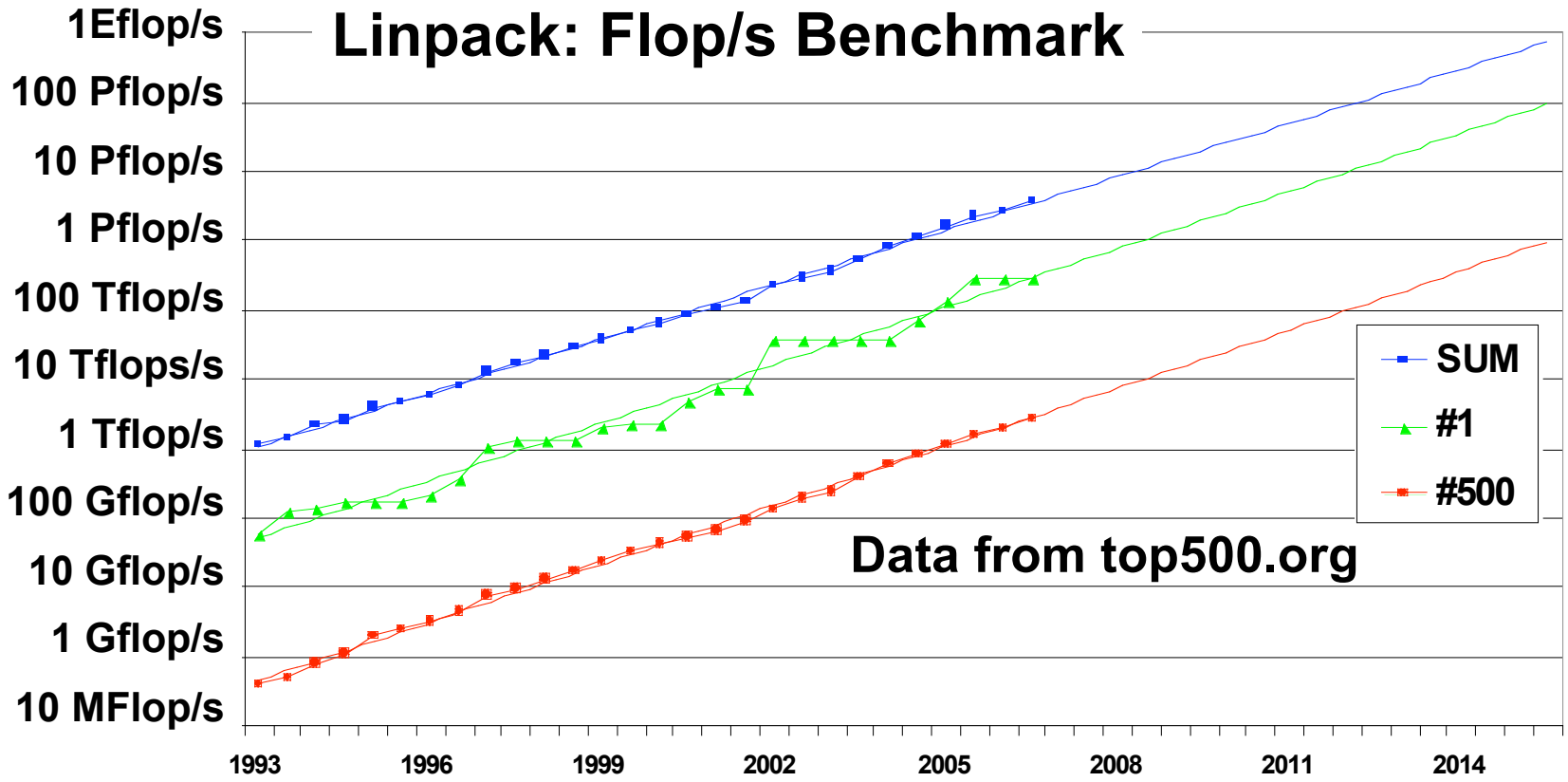
New: Power Wall

Can put more transistors on a chip than can afford to turn on



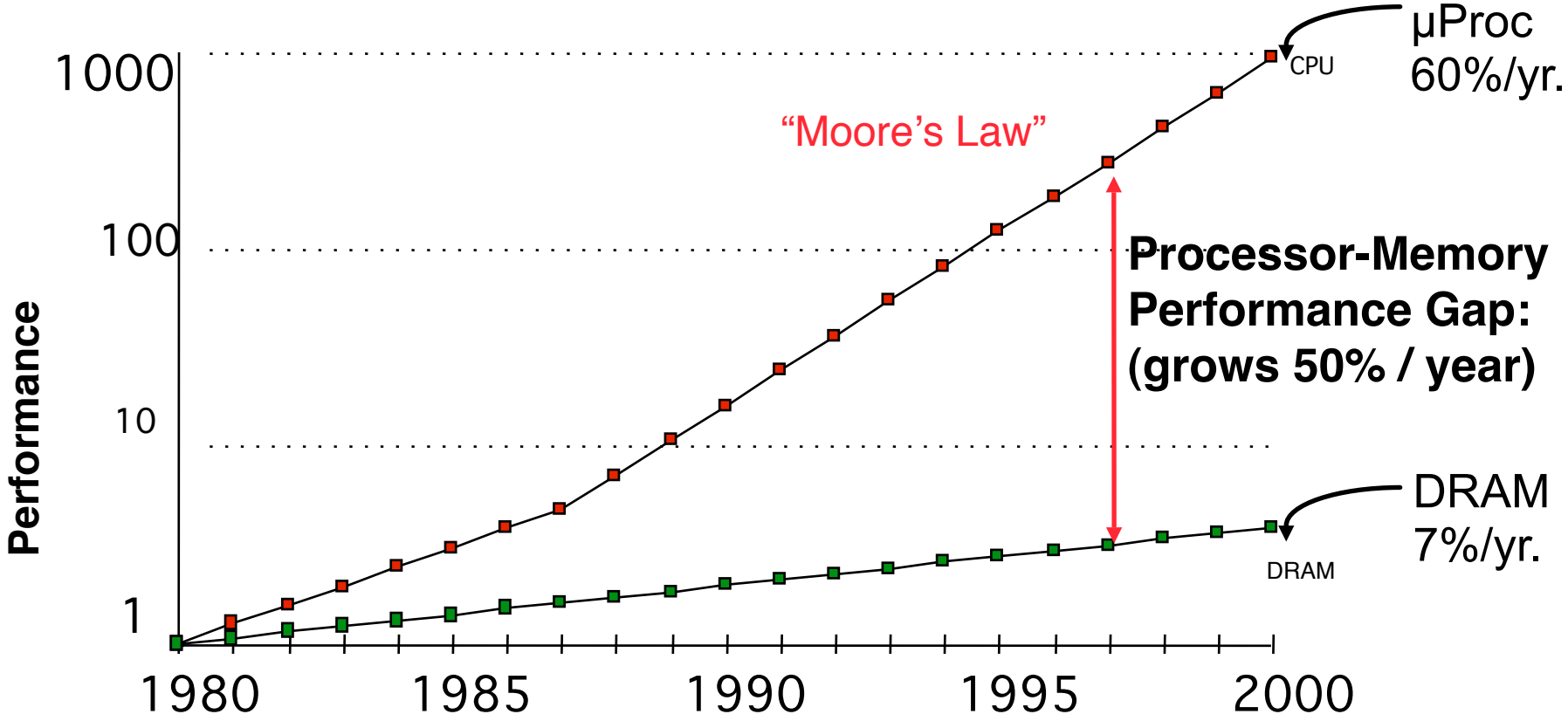
Very Old: Multiplies Slow, Loads fast

- Design algorithms to reduce floating point operations
- Machines measured on peak flop/s



New: Memory Performance is Key

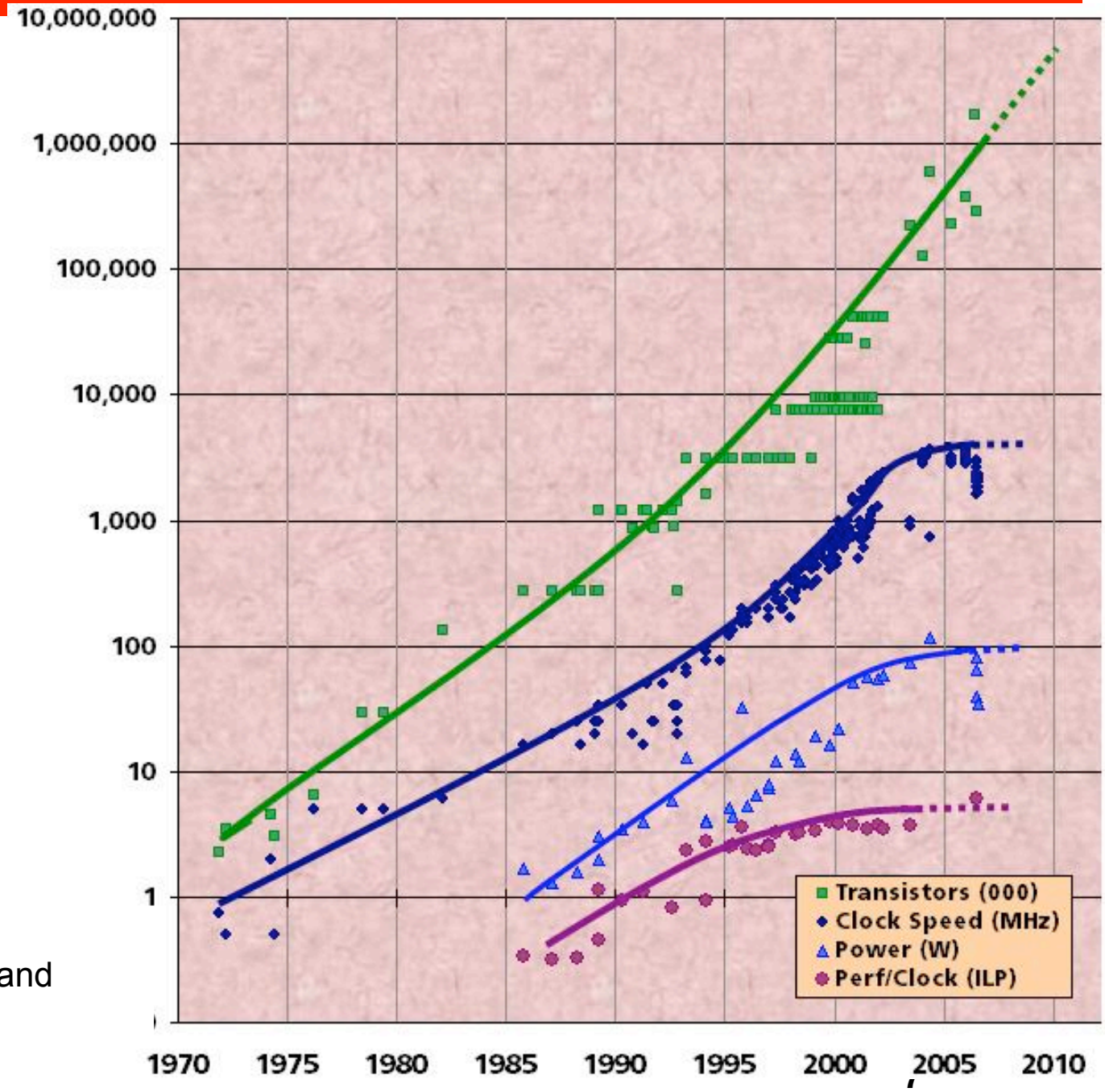
Ever-growing processor-memory performance gap



- **Total chip performance still growing with Moore's Law**
- **Bandwidth rather than latency will be growing concern**

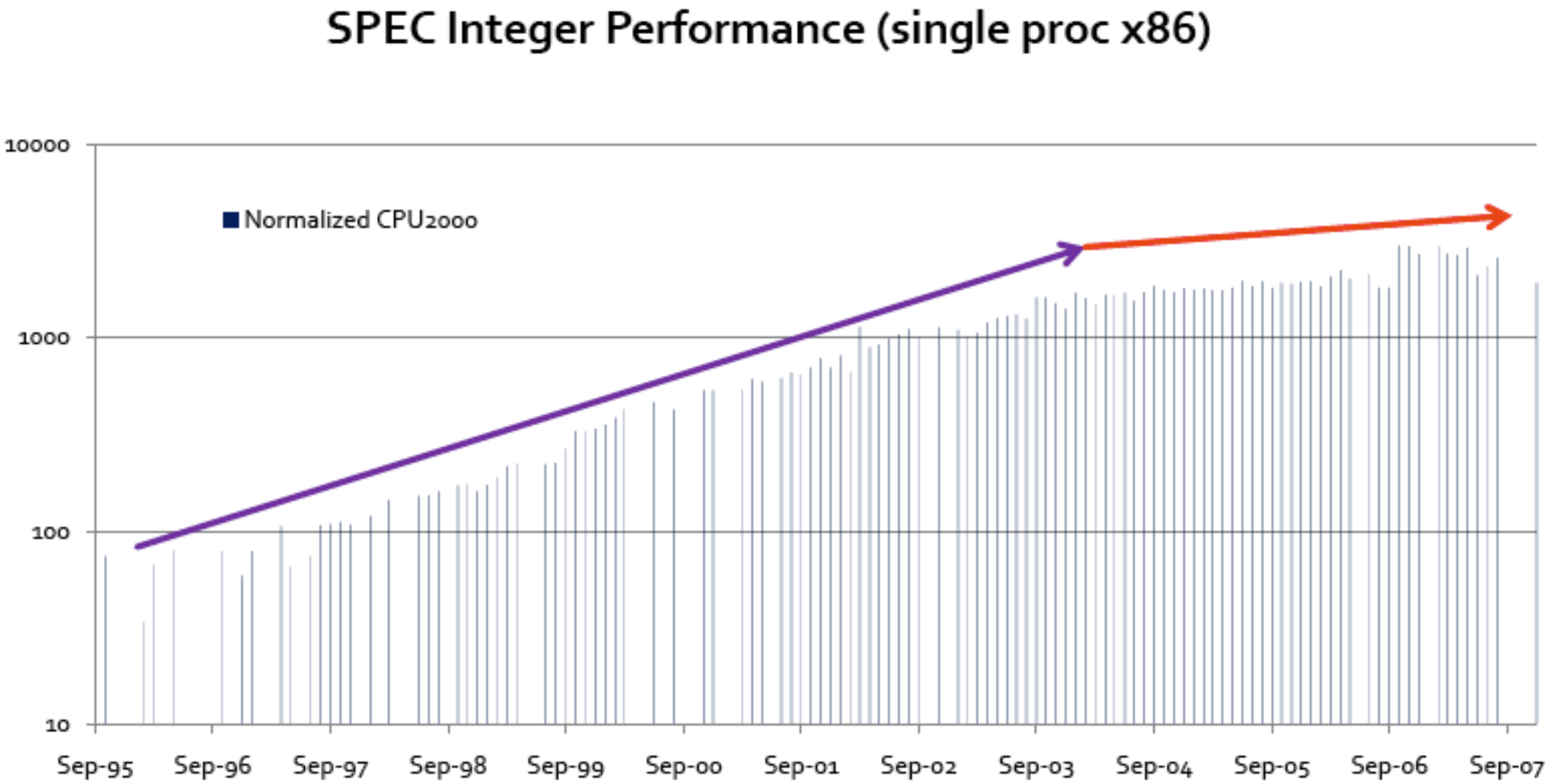
New: Clock Scaling Bonanza Has Ended

- Chip density is continuing increase ~2x every 2 years
  - Clock speed is not
  - Number of processor cores may double instead
- There is little or no hidden parallelism (ILP) to be found
- Parallelism must be exposed to and managed by software



Source: Intel, Microsoft (Sutter) and Stanford (Olukotun, Hammond)

Single-core performance slowing

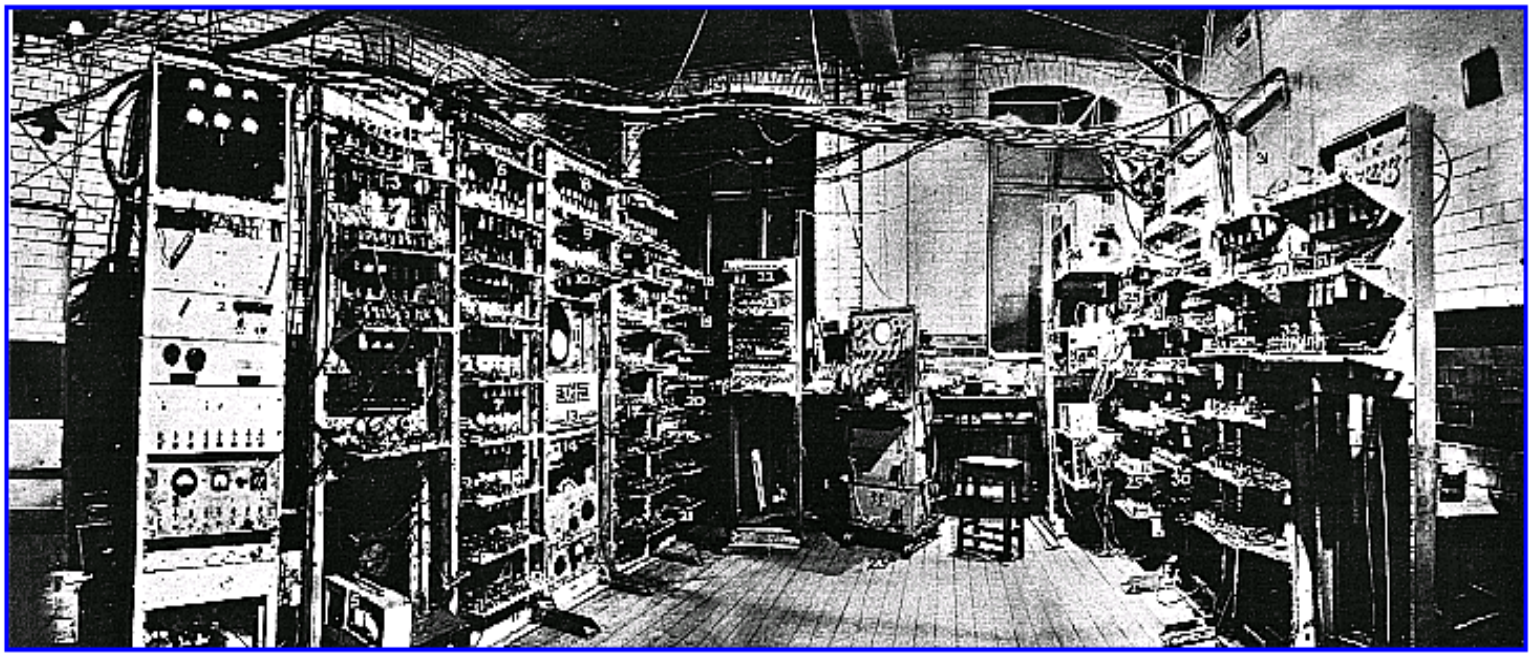


Jim Larus, Microsoft, from a talk at UC Davis in May 2009

Old: Parallelism only for High End Computing

Address http://www.paralogos.com/DeadSuper/index.html Go Links

The Dead Supercomputer Society



The Passing of a Golden Age?

From the construction of the first programmed computers until the mid 1990s, there was always room in the computer industry for someone with a clever, if sometimes challenging, idea on how to make a more powerful machine. Computing became strategic during the Second World War, and remained so during the Cold War that followed. High-performance computing is essential to any modern nuclear weapons program, and a computer technology "race" was a logical corollary to the arms race. While powerful computers are of great value to a number

New: Parallelism by Necessity

"This shift toward increasing parallelism is not a triumphant stride forward based on breakthroughs in novel software and architectures for parallelism; instead, this plunge into parallelism is actually a retreat from even greater challenges that thwart efficient silicon implementation of traditional uniprocessor architectures."

Kurt Keutzer, Berkeley View, December 2006

- HW/SW Industry bet its future that breakthroughs will appear before it's too late

Conventional Wisdom (CW) in Computer Architecture

1. *Old CW*: Power is free, but transistors expensive
- *New CW: Power wall* Power expensive, transistors "free"
  - Can put more transistors on a chip than have the power to turn on
2. *Very Old CW*: Multiplies slow, but loads fast
- *New CW*: *Memory wall* Loads slow, multiplies fast
  - 200 clocks to DRAM, but even FP multiplies only 4 clocks
3. *Old CW*: More ILP via compiler/architecture innovation
  - Branch prediction, speculation, Out-of-order execution, VLIW, …
- *New CW*: *ILP wall* Diminishing returns on more ILP
4. *Old CW*: 2X CPU Performance every 18 months
- *New CW*: *Power + Memory + ILP Walls* = *Brick Wall*
5. *Old CW*: Parallelism is only for Scientific fringe
- *New CW*: *Parallelism is everywhere*

7 Questions for Parallelism

- **Applications:**

1. What are the apps?
2. What are kernels of apps?

- **Hardware:**

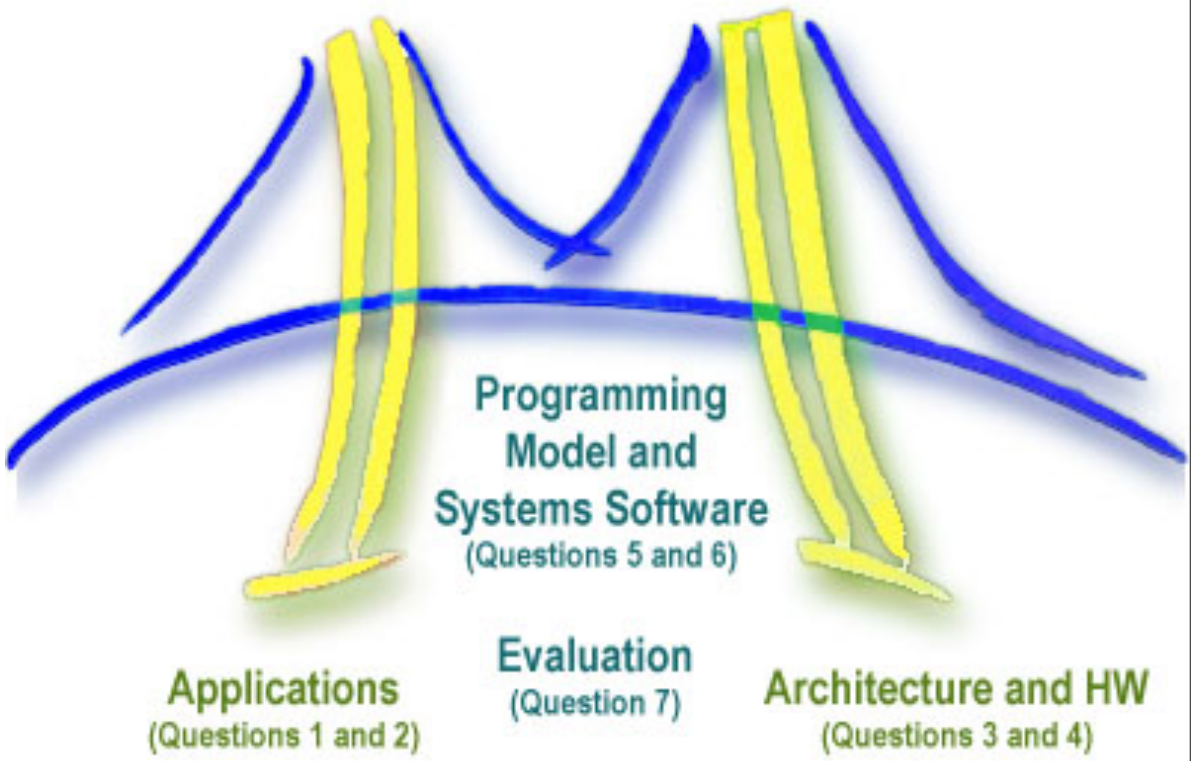
3. What are the HW building blocks?
4. How to connect them?

- **Programming Model & Systems Software:**

5. How to describe apps and kernels?
6. How to program the HW?

- **Evaluation:**

7. How to measure success?

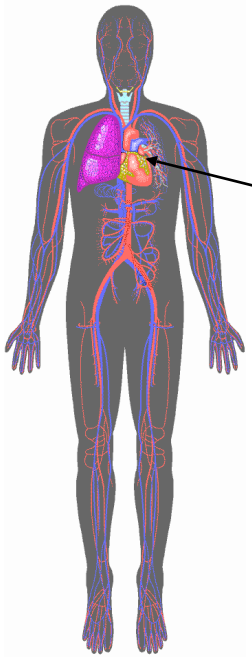


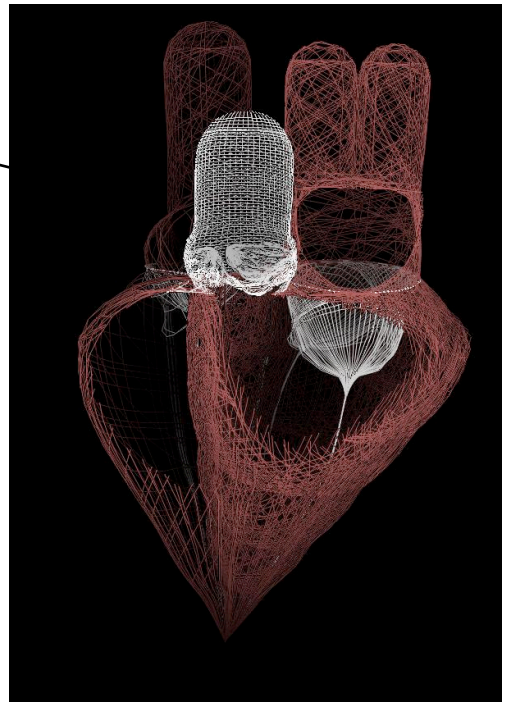
(Inspired by a view of the Golden Gate Bridge from Berkeley)

What do you see as future directions for higher-performance computing?

Example Applications in Health

- Imagine a “digital body double”
  - 3D image-based medical record
  - Includes diagnostic, pathologic, and other information
- Used for:
  - Diagnosis
  - Less invasive surgery-by-robot
  - Experimental treatments
  - Real-time diet and exercise recommendations

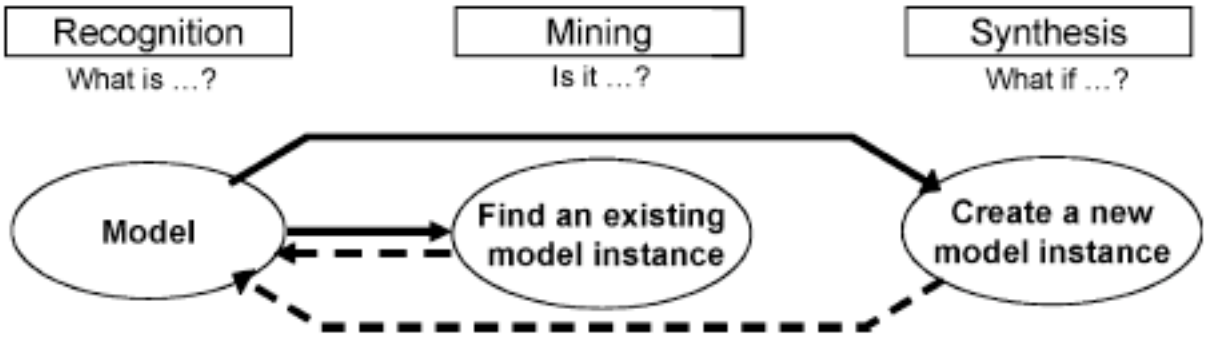


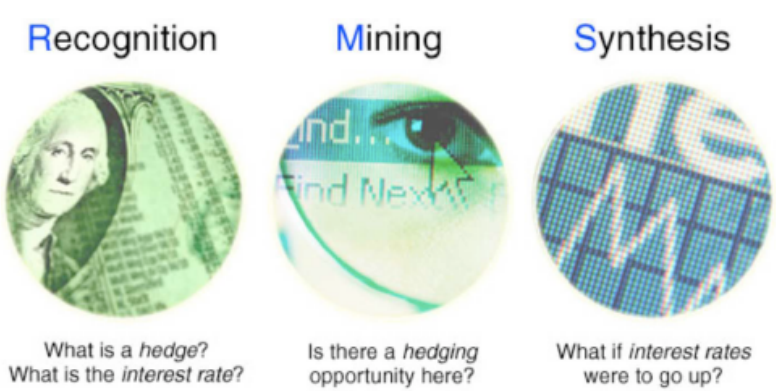


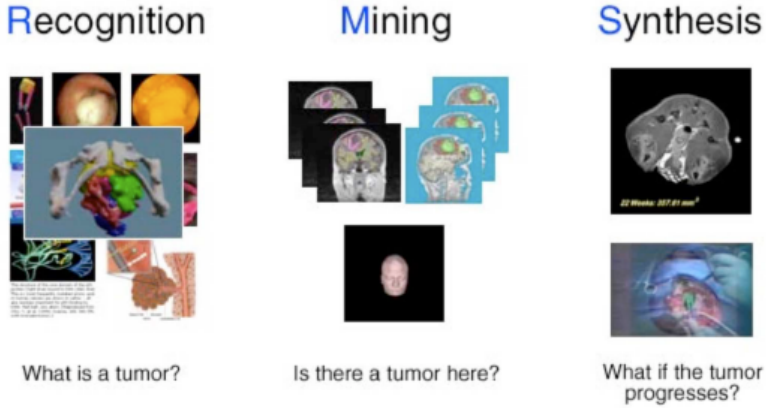
Existing simulations

- Heart
- Lung
- Brain
- Kidney
- Bone mass

RMS Applications

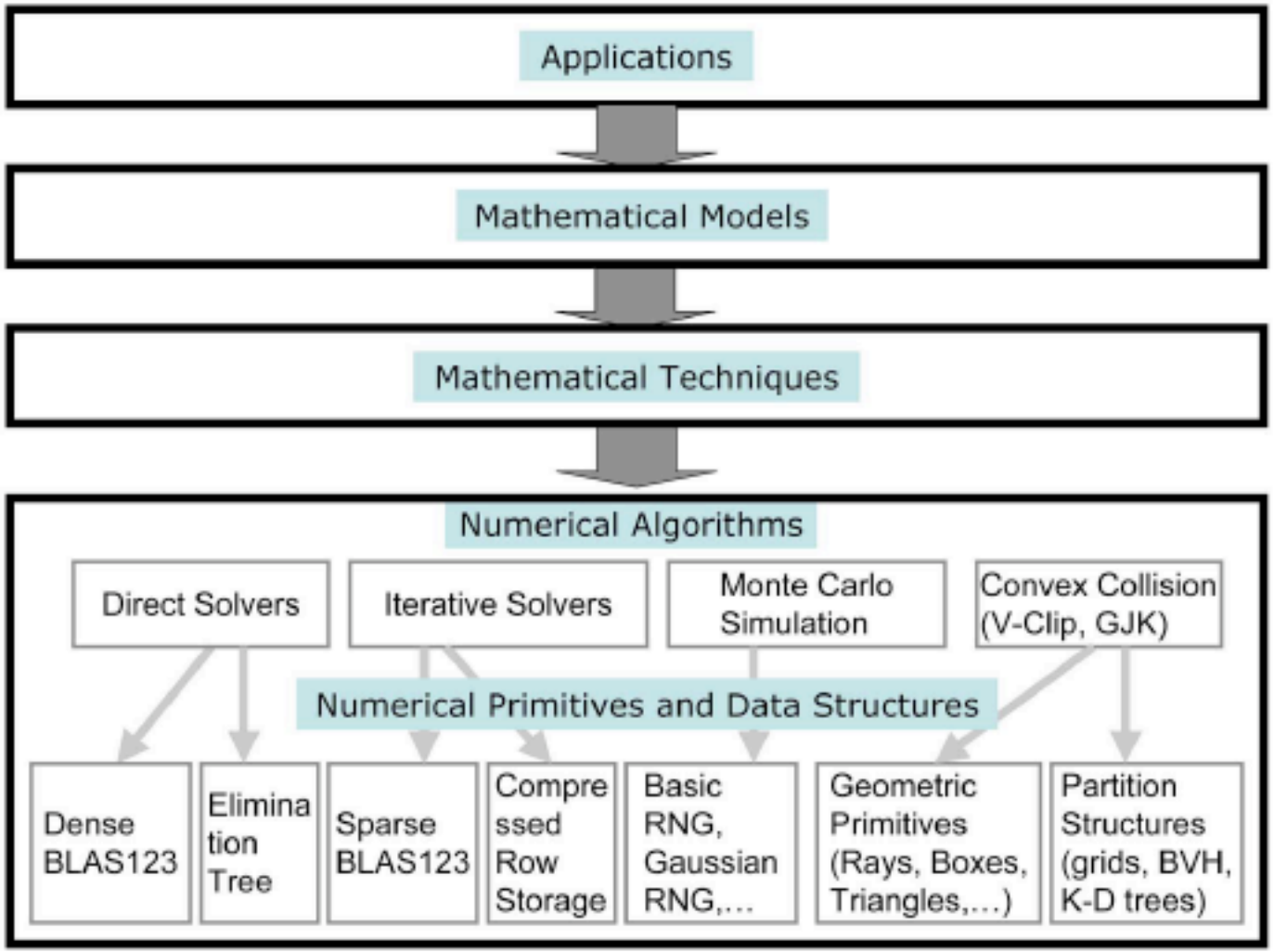




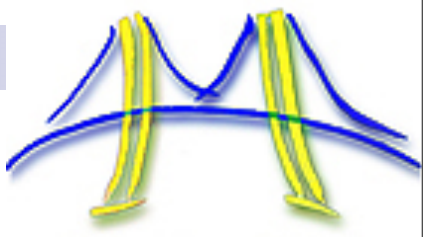


- Chen et al., *Convergence of Recognition, Mining, and Synthesis Workloads and Its Implications*. Proceedings of the IEEE, May 2008.

RMS App Commonality

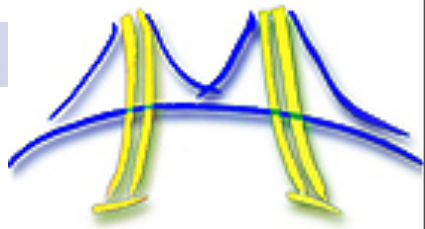


Apps and Kernels Tower: What are the problems?



- Old Conventional Wisdom: Use old programs to evaluate future computers
  - For example, SPEC2006, EEMBC
  - Tied to peculiarities of code artifact vs. fundamentals
  - Black-box benchmarks: don't understand or change internals
- Berkeley View
  - Computer HW and SW designers must *understand* applications
  - Killer apps for future systems are not yet known: understand the building blocks and algorithmic trends

Phillip Colella's "Seven dwarfs"



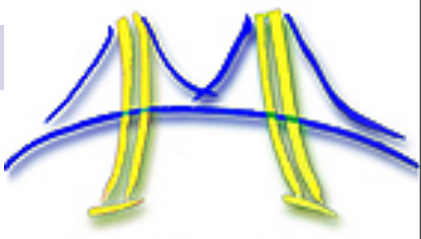
High-end simulation in the physical sciences = 7 numerical methods:

1. Structured Grids (including locally structured grids, e.g. Adaptive Mesh Refinement)
2. Unstructured Grids
3. Fast Fourier Transform
4. Dense Linear Algebra
5. Sparse Linear Algebra
6. Particles
7. Monte Carlo

- A dwarf is a pattern of computation and communication
- Dwarfs are well-defined targets from algorithmic, software, and architecture standpoints

Slide from "Defining Software Requirements for Scientific Computing", Phillip Colella, 2004

Do dwarfs work well outside HPC?

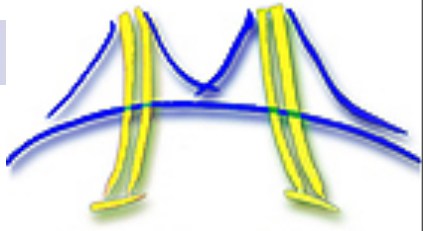


- Examine use of 7 dwarfs elsewhere

1. Embedded Computing (EEMBC benchmark)
2. Desktop/Server Computing (SPEC2006)
3. Data Base / Text Mining Software
   - Advice from Jim Gray of Microsoft and Joe Hellerstein of UC
4. Games/Graphics/Vision
5. Machine Learning
   - Advice from Mike Jordan and Dan Klein of UC Berkeley

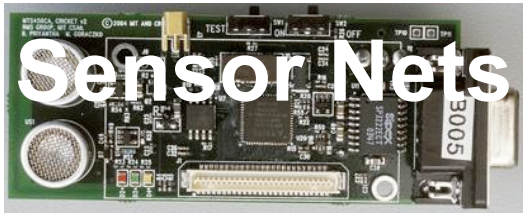
- Result: Added 7 more dwarfs, revised 2 original dwarfs, renumbered list

Dwarf Use (Red Important → Blue Not)

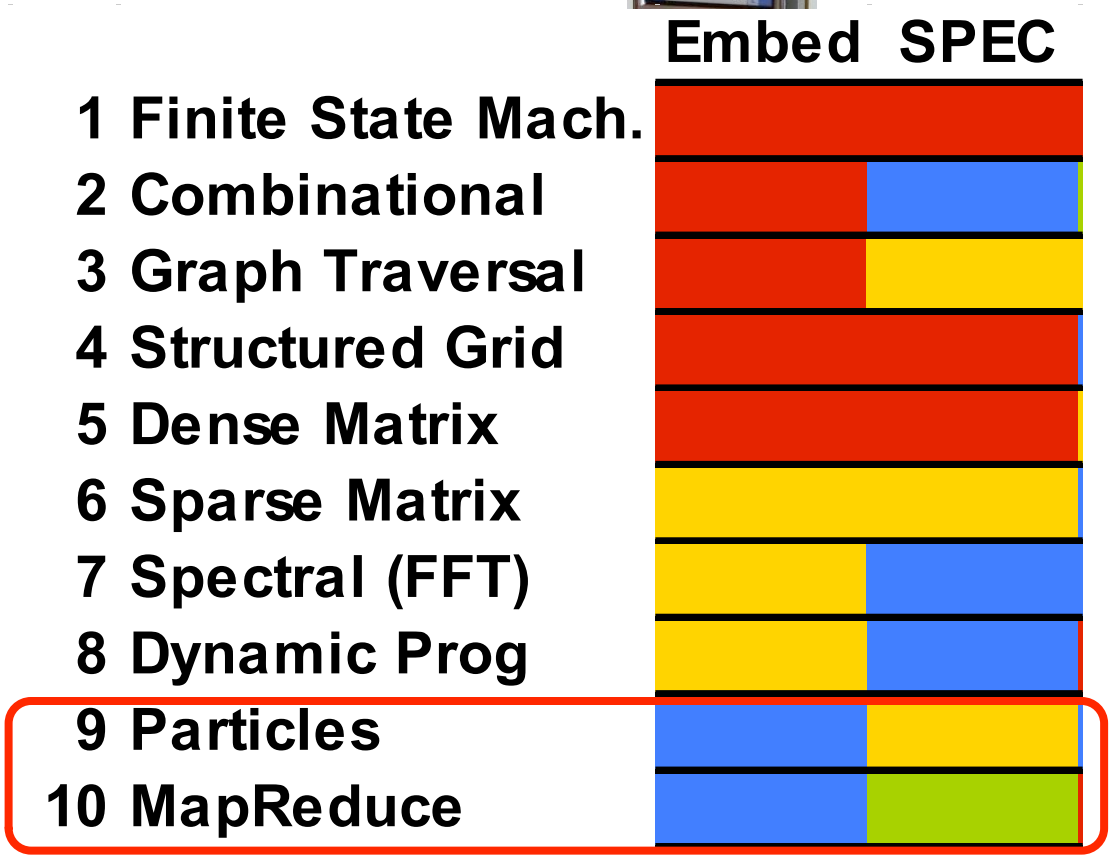


1. Embedded (42 EEMBC benchmarks)

	Embed
1 Finite State Mach.	
2 Combinational	
3 Graph Traversal	
4 Structured Grid	
5 Dense Matrix	
6 Sparse Matrix	
7 Spectral (FFT)	
8 Dynamic Prog	

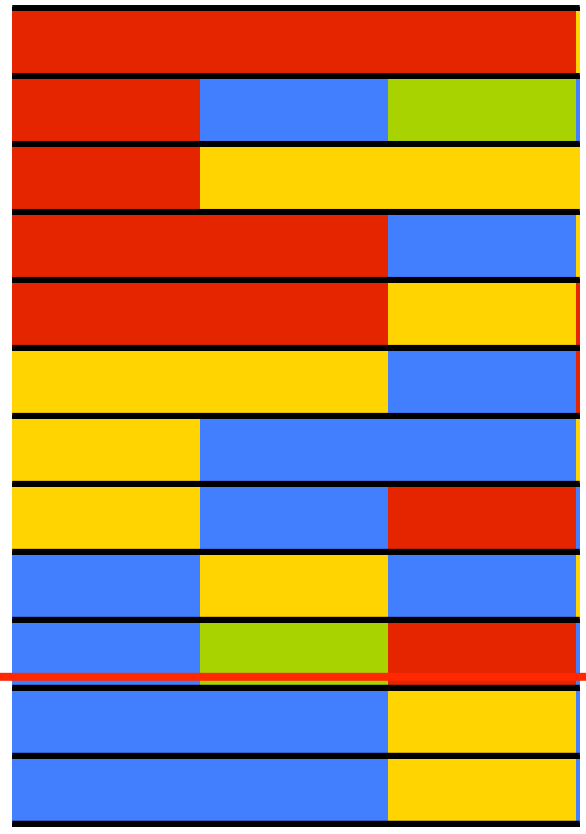


Dwarf Use (Red Important → Blue Not)



2. Desktop/Server (28 SPEC2006 benchmarks)

Dwarf Use (Red Important → Blue Not)



3. Database / Text Mining

ORACLE

MySQL™

Google™

Dwarf Use (Red Important → Blue Not)

MySQL

ORACLE

4. Video Games

	Embed	SPEC	DB	Games
1 Finite State Mach.				
2 Combinational				
3 Graph Traversal				
4 Structured Grid				
5 Dense Matrix				
6 Sparse Matrix				
7 Spectral (FFT)				
8 Dynamic Prog				
9 Particles				
10 MapReduce				
11 Backtrack/ B&B				
12 Graphical Models				
13 Unstructured Grid				

Dwarf Use (Red Important → Blue Not)



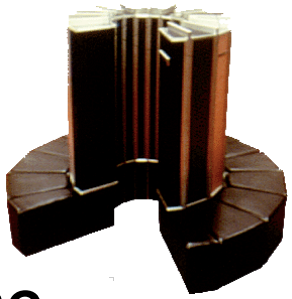

5. Machine Learning

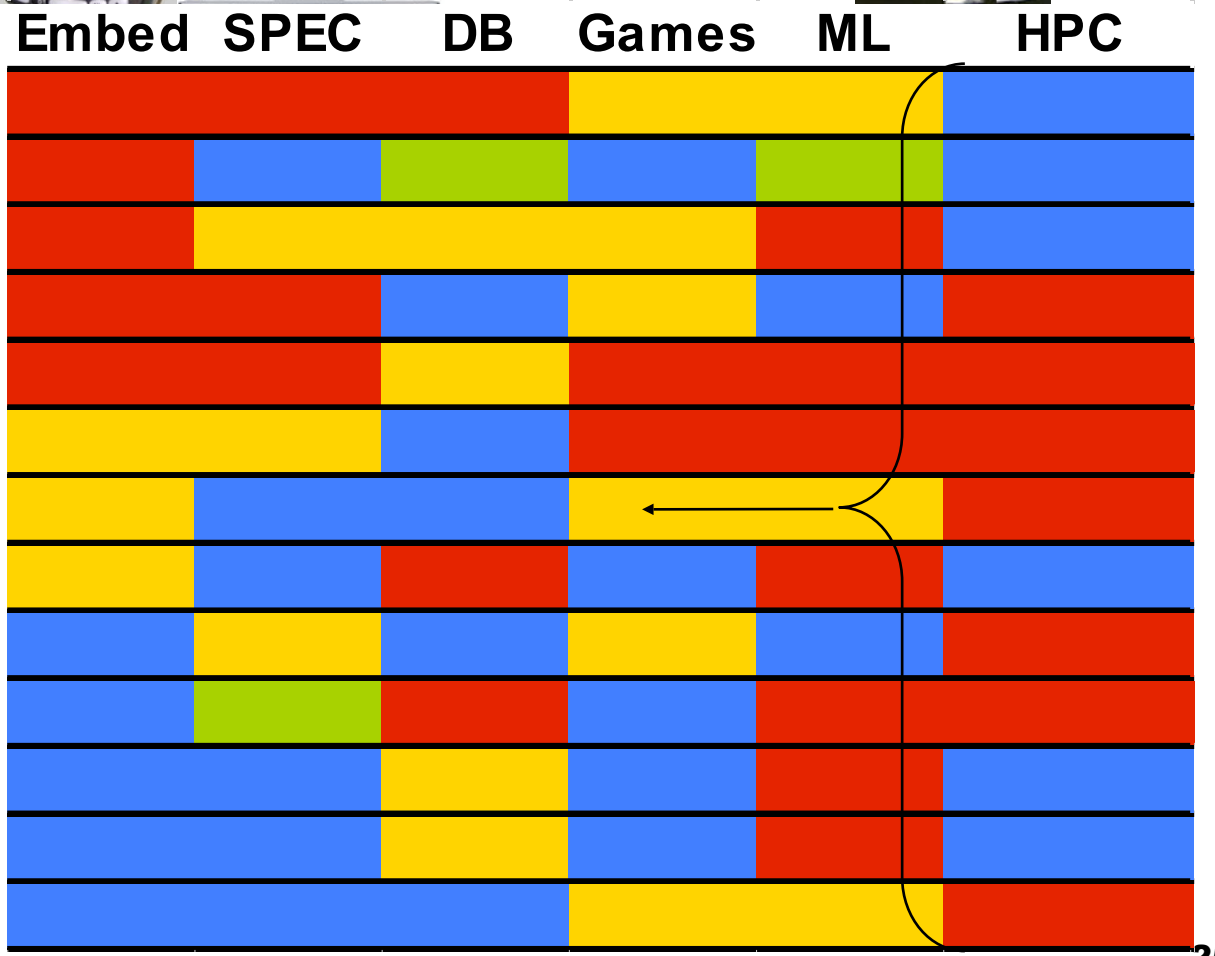
	Embed	SPEC	DB	Games	ML
1 Finite State Mach.					
2 Combinational					
3 Graph Traversal					
4 Structured Grid					
5 Dense Matrix					
6 Sparse Matrix					
7 Spectral (FFT)					
8 Dynamic Prog					
9 Particles					
10 MapReduce					
11 Backtrack/ B&B					
12 Graphical Models					
13 Unstructured Grid					

Dwarf Use (Red Important → Blue Not)



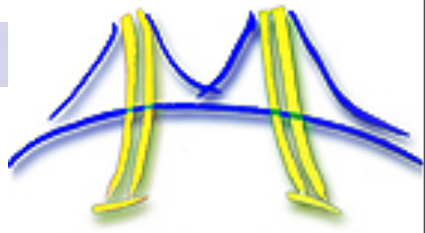




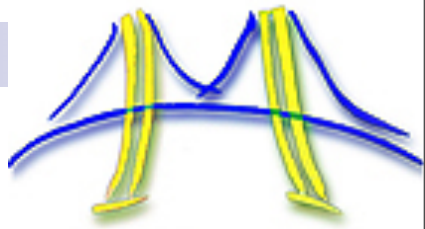
Roles of Dwarfs

1. Give us a vocabulary/organization to talk across disciplinary boundaries
2. Define minimum set of necessary functionality for new hardware/software systems
3. Define building blocks for creating libraries that cut across application domains
4. “Anti-benchmarks” not tied to code or language artifacts ⇒ encourage innovation in algorithms, languages, data structures, and/or hardware
5. They decouple research, allowing analysis of HW & SW programming support without waiting years for full app development



Hardware Tower: What are the problems?

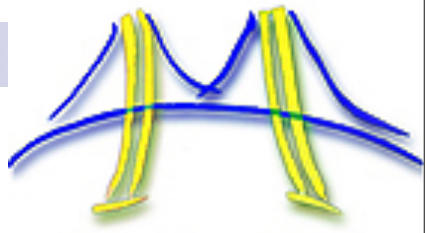
- Power limits leading edge chip designs
  - Intel Tejas Pentium 4 cancelled due to power issues
- Yield on leading edge processes dropping dramatically
  - IBM quotes yields of 10–20% on 8-processor Cell
- Design/validation leading edge chip is becoming unmanageable
  - Verification teams > design teams on leading edge processors



HW Solution: Small is Beautiful

- Expect modestly pipelined (5- to 9-stage) CPUs, FPUs, vector, SIMD PEs
  - Small cores not much slower than large cores
- Parallel is energy efficient path to performance: Power = CV^2F
  - Lower voltage, and increase parallelism lowers energy per op
- Redundant processors can improve chip yield
  - Cisco Metro 188 CPUs + 4 spares; Sun Niagara sells 6 or 8 CPUs
- Small, regular processors easier to verify
- One size fits all?
  - Amdahl's Law ⇒ Heterogeneous processors?

Number of Cores/Socket



- We need revolution, not evolution
- Software or architecture alone can't fix parallel programming problem, need innovations in both
- "Multicore" 2X cores per generation: 2, 4, 8, …
- "Manycore" 100s is highest performance per unit area, and per Watt, then 2X per generation: 64, 128, 256, 512, 1024 …
- **Multicore architectures, programming models, and applications good for 2 to 32 cores won't evolve to Manycore systems of 1000's of cores**
  ⇒ Desperately need HW/SW models that work for Manycore or will run out of steam (as ILP ran out of steam at 4 instructions)

7 Questions for Parallelism

- **Applications:**

1. What are the apps?
2. What are kernels of apps?

- **Hardware:**

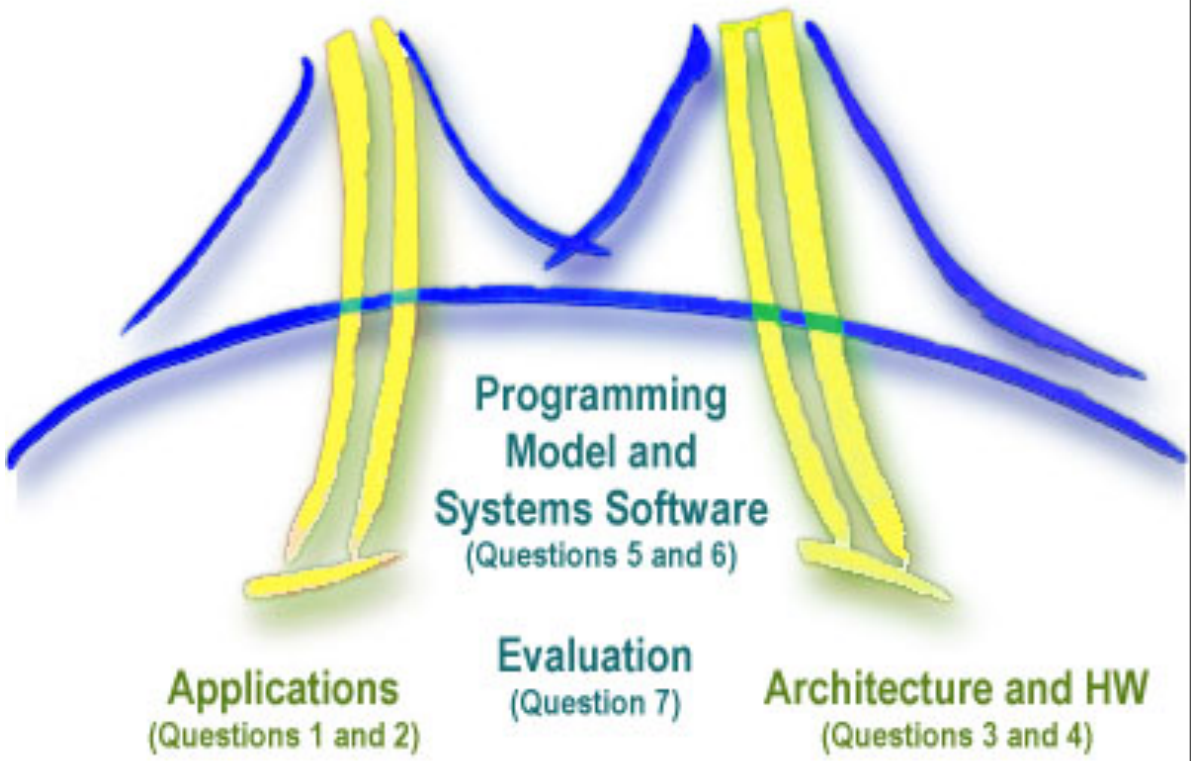
3. What are the HW building blocks?
4. How to connect them?

- **Programming Model & Systems Software:**

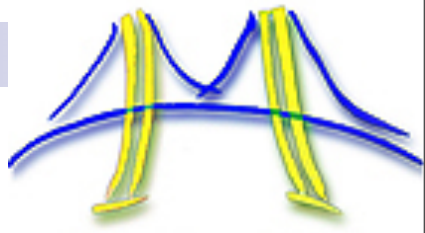
5. How to describe apps and kernels?
6. How to program the HW?

- **Evaluation:**

7. How to measure success?



(Inspired by a view of the Golden Gate Bridge from Berkeley)



Programming Model: What are the problems?

- Primary recent focus on correctness, not performance
  - Relied on “Moore’s Law” to make programs faster
- New generation of performance programmers
  - Why parallel if performance doesn’t matter?
- Programming model must balance *productivity* and *implementation efficiency*
  - Enable software industry for manycore
  - Usable by most programmers

Old CW in Programming Models

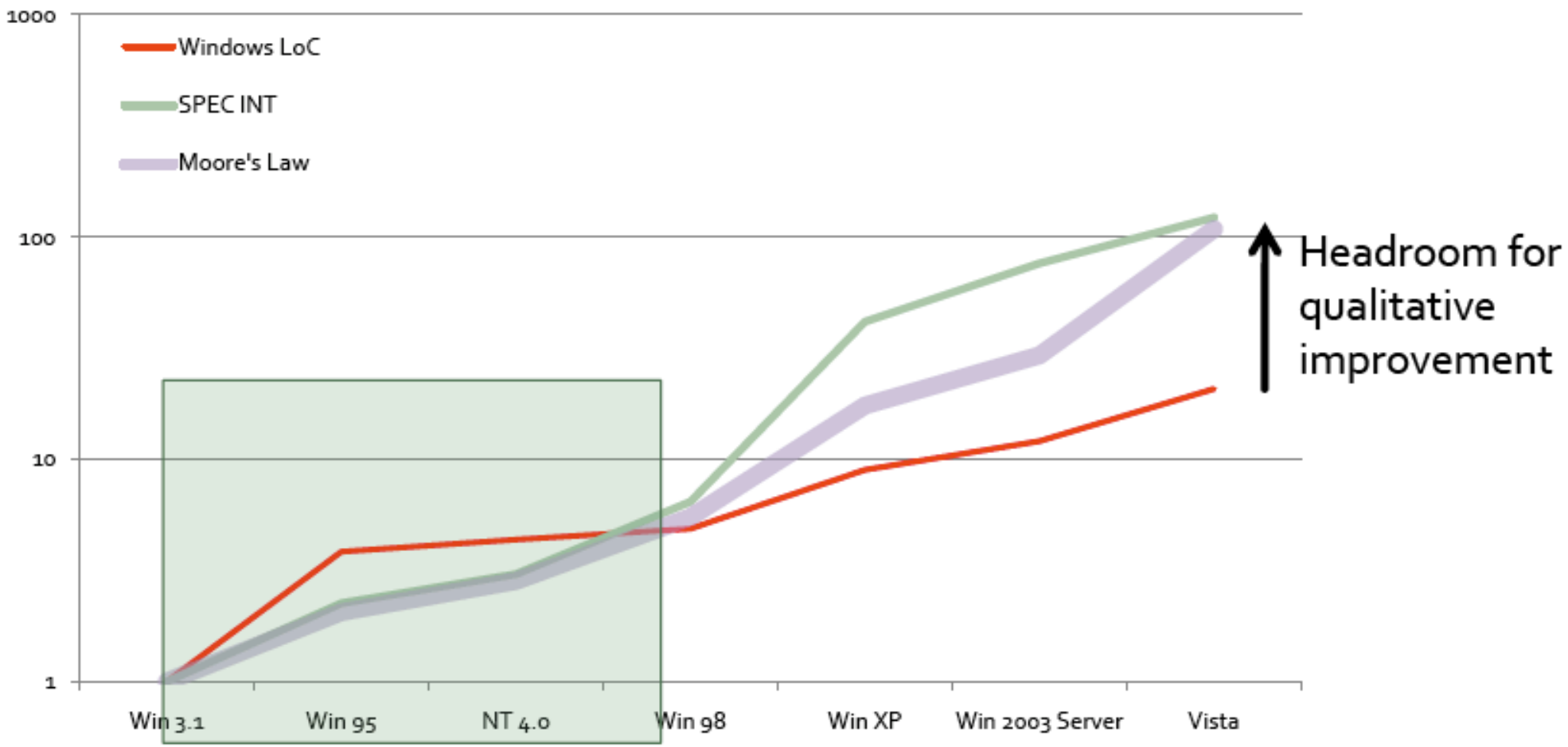
- Design new hardware with exotic performance features
- Hand to software communication
  - Develop new compiler technology and wait for maturity and integration into commercial compilers
  - Takes ~10 years in practice
- Compilers should
  - Compiler arbitrary code efficiently
  - Hide performance issues from programmer
  - Run quickly (secs-mins, human is in the loop)
- Search for the holy grail language
  - One language for all problems

New CW in Programming Models

- Feature creep in languages annotations → de facto new languages
- Programs written in many languages
  - Python, C++, Perl, Java, Javascript, C#,…
- Automatic performance tuning
  - Use machine time in place of human time for tuning
  - Search over possible implementations
  - Autotuned libraries for dwarfs (up to 10x speedup)

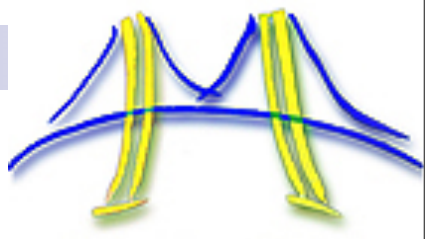
•Spectral (FFTW, Spiral)
•Dense (Atlas, PHiPAC)
•Sparse (OSKI)
•Structured (OSKI')

Software complexity



Fine print: Wikipedia estimates of LoC. Does not measure code shipped to customers.
SPEC normalized between SPEC95 and SPEC2000.

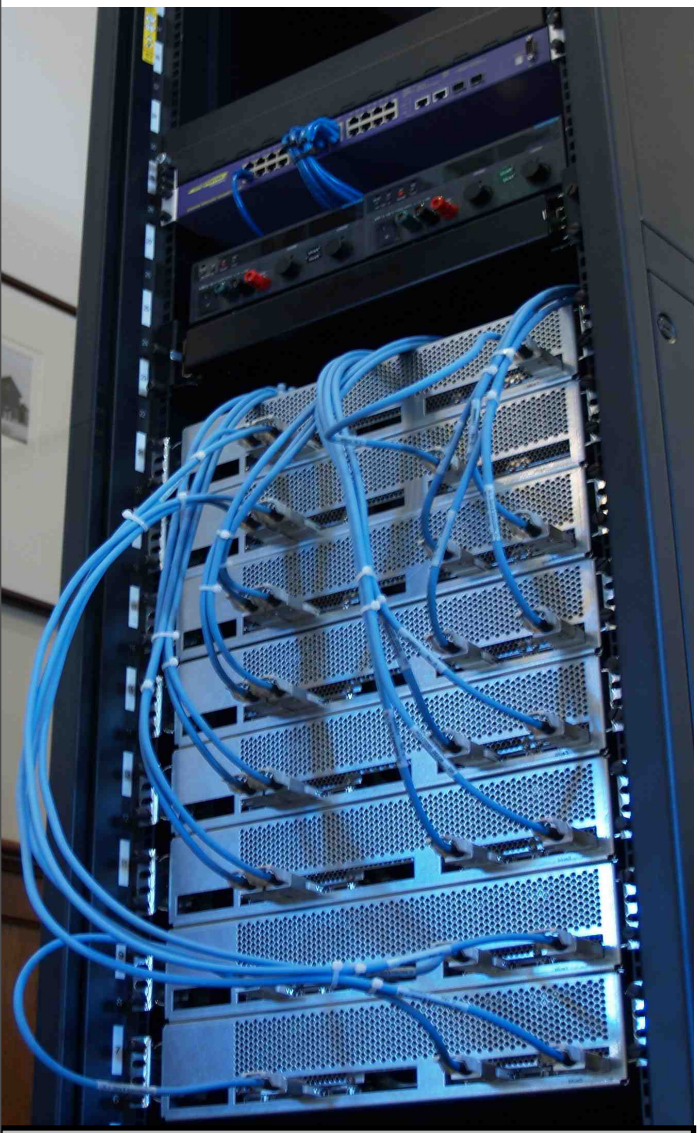
Jim Larus, Microsoft, from a talk at UC Davis in May 2009



Measuring Success: What are the problems?

1. ≈ Only companies can build HW, and it takes years
2. Software people don't start working hard until hardware arrives
   - 3 months after HW arrives, SW people list everything that must be fixed, then we all wait 4 years for next iteration of HW/SW
3. How get 1000 CPU systems in hands of researchers to innovate in timely fashion on in algorithms, compilers, languages, OS, architectures, … ?
4. Can avoid waiting years between HW/SW iterations?

Rapid Prototyping with FPGAs



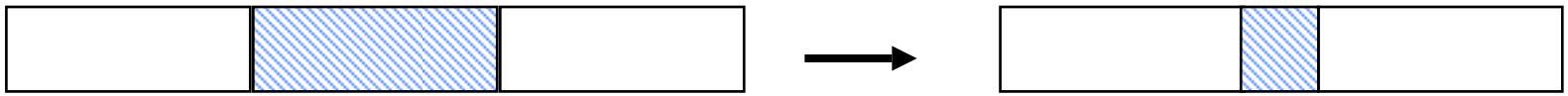
RAMP Blue, January 2007
- 256 RISC cores @100MHz
- Works! Runs UPC version of NAS benchmarks.

- "Research Accelerator for Multi-Processors"
  - Multi-University collaboration developing FPGA "gateware" for manycore emulations (10 faculty at UCB, CMU, MIT, Stanford, Texas, Washington)
- Enables rapid interaction between hardware and software developers
  - "Tapeout" every day, not once in five years
  - Fast enough (100MHz) for software development
- RAMP Design Language (RDL) provides "gateware linker" and ***cycle-accurate timing models***.
  - *All operations (DRAM access, FP multiply, disk access) take exact same number of clock cycles as on desired target machine*
- Multiple machine styles in progress
  - RAMP Blue (UC Berkeley) cluster/message-passing
  - RAMP Red (Stanford) transactional memory
  - RAMP White (Everyone) cache-coherent CMP

Amdahl's Law

- Speedup due to enhancement E:

$$\text{Speedup}(E) = \frac{\text{Execution Time without } E}{\text{Execution Time with } E} = \frac{\text{Performance with } E}{\text{Performance without } E}$$



- Suppose that enhancement E accelerates a fraction F of the task by a factor S and the remainder of the task is unaffected:

$$\text{Execution time (with } E) = ((1 - F) + F/S) \cdot \text{Execution time (without } E)$$

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

- Design Principle: Make the common case fast!

Why EEC 171?

- Old CW: Don't bother parallelizing your application, as you can just wait a little while and run it on a much faster sequential computer.
- New CW: It will be a very long wait for a faster sequential computer.
- Old CW: Increasing clock frequency is the primary method of improving processor performance.
- New CW: Increasing parallelism is the primary method of improving processor performance.
- Old CW: Less than linear scaling for a multiprocessor application is failure.
- New CW: Given the switch to parallel computing, any speedup via parallelism is a success.

Extracting Yet More Performance

- Two options:
  - Increase the depth of the pipeline to increase the clock rate — superpipelining
    - How does this help performance? (What does it impact in the performance equation?)
  - Fetch (and execute) more than one instruction at one time (expand every pipeline stage to accommodate multiple instructions) — multiple-issue
    - How does this help performance? (What does it impact in the performance equation?)
    - Today's topic!

$$\frac{\text{seconds}}{\text{program}} = \frac{\text{instructions}}{\text{program}} \times \frac{\text{cycles}}{\text{instruction}} \times \frac{\text{seconds}}{\text{cycle}}$$

Instruction vs Machine Parallelism

- Instruction-level parallelism (ILP) of a program – a measure of the average number of instructions in a program that a processor might be able to execute at the same time
  - Mostly determined by the number of true (data) dependencies and procedural (control) dependencies in relation to the number of other instructions
  - ILP is traditionally “extracting parallelism from a single instruction stream working on a single stream of data”

Instruction vs Machine Parallelism

- Machine parallelism of a processor – a measure of the ability of the processor to take advantage of the ILP of the program
  - Determined by the number of instructions that can be fetched and executed at the same time

- ***To achieve high performance, need both ILP and machine parallelism***

Why is ILP a good idea? If you were designing a computer system, why would you choose ILP instead of, say, multiple processors?

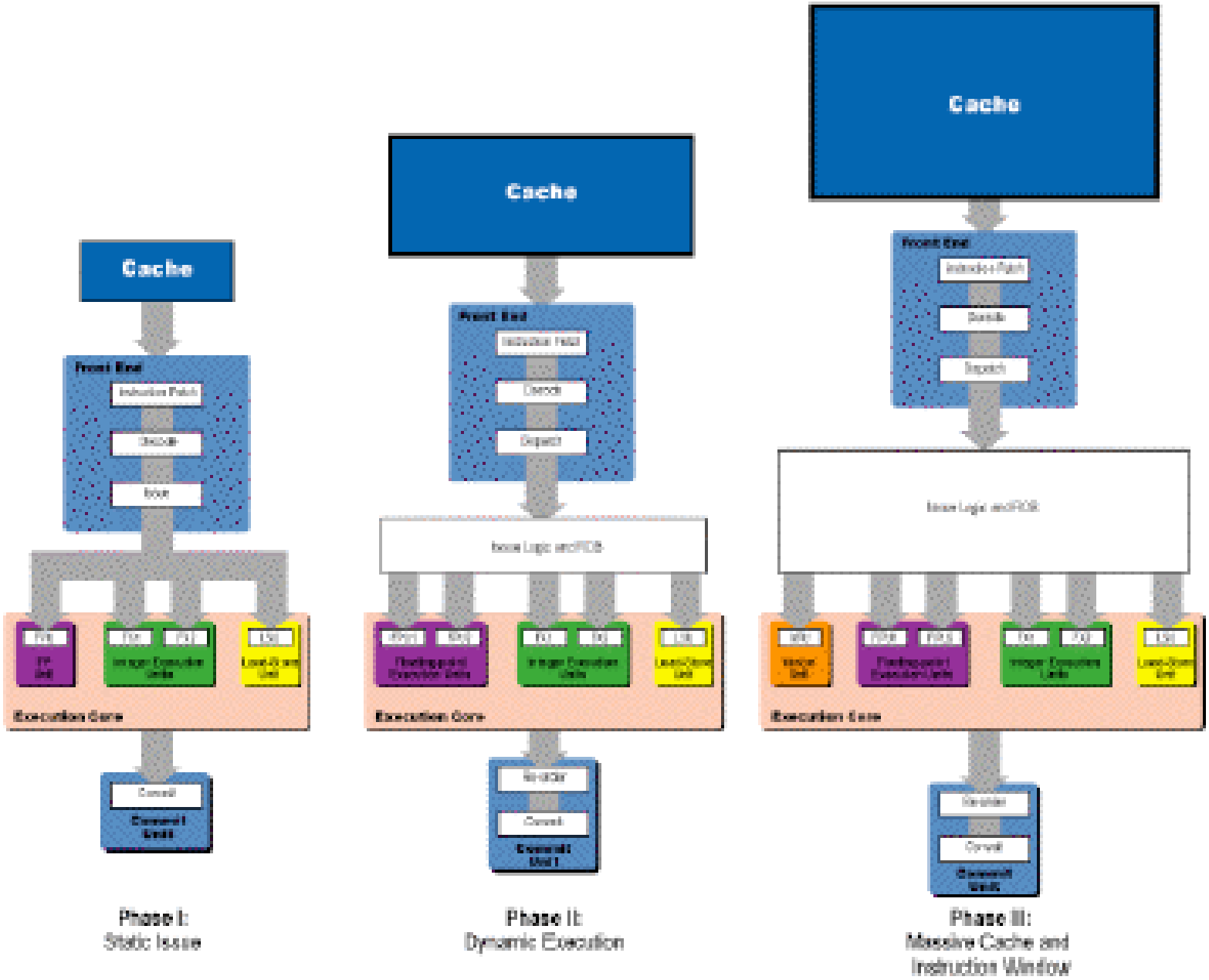
What kind of code has lots of ILP?

What kind of code has little ILP?

Machine Parallelism

- There are 2 main approaches for machine parallelism. Responsibility of resolving hazards is ...
  - Primarily hardware-based—“dynamic issue”, “superscalar”
  - Primarily software-based—“VLIW”

Growing complexity ...

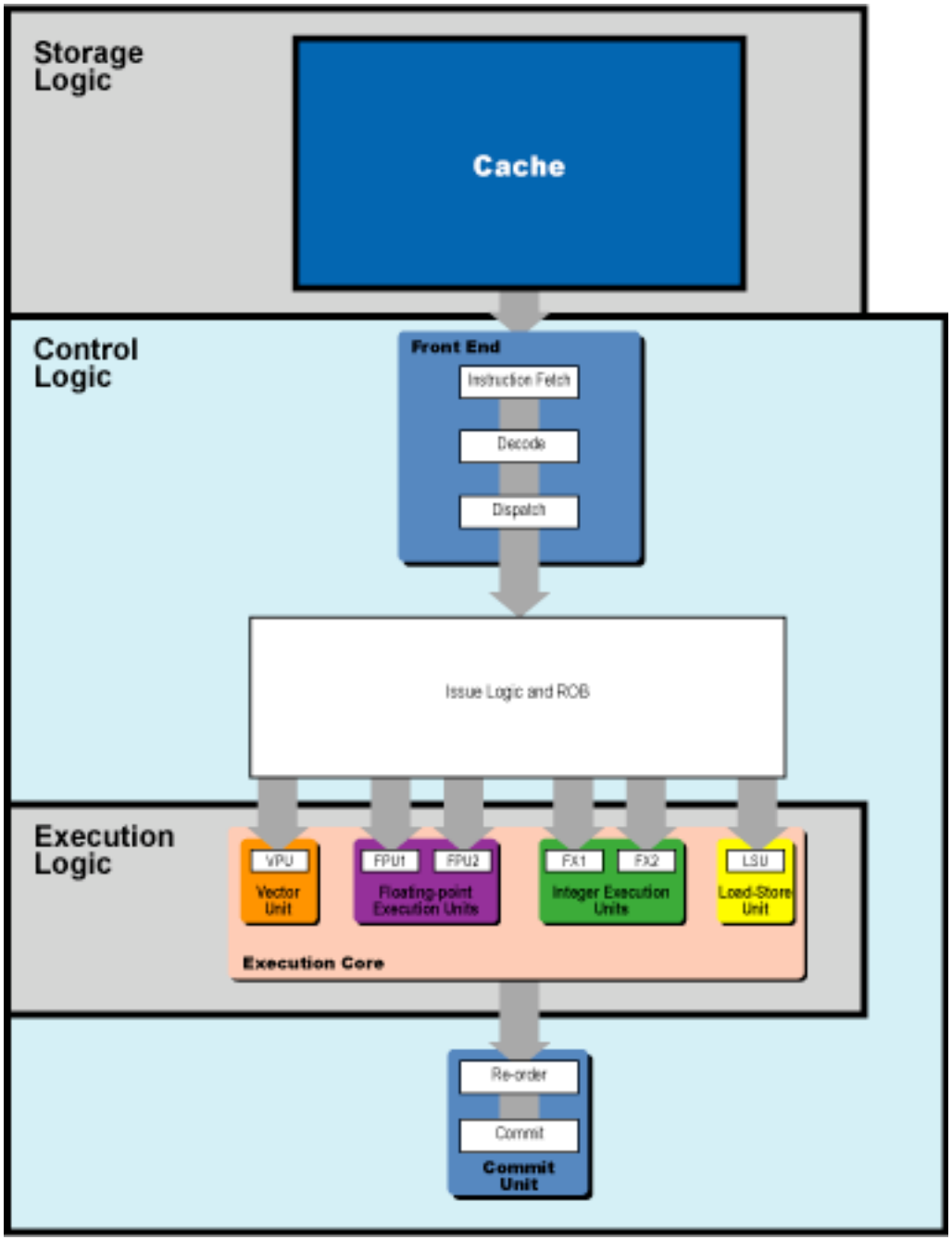


Phase I:
Static Issue

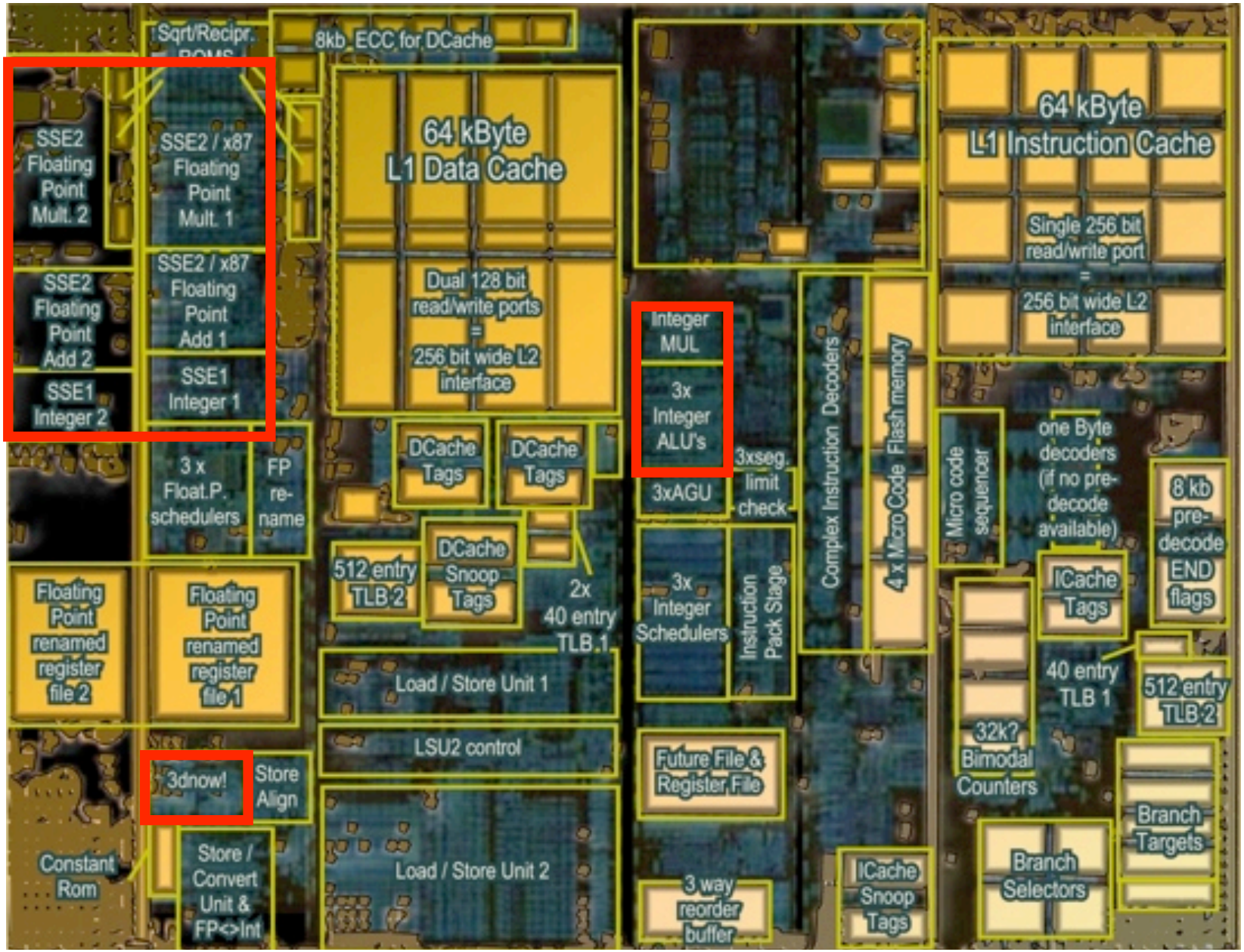
Phase II:
Dynamic Execution

Phase III:
Massive Cache and
Instruction Window

Small fraction for datapath



AMD “Deerhound” (K8L)



How does out-of-order issue help?

How does out-of-order completion help?

How does register renaming help?

Static Multiple Issue Machines (VLIW)

- Static multiple-issue processors (aka VLIW) use the compiler to decide which instructions to issue and execute simultaneously
  - Issue packet—the set of instructions that are bundled together and issued in one clock cycle—think of it as one large instruction with multiple operations
  - The mix of instructions in the packet (bundle) is usually restricted—a single “instruction” with several predefined fields
  - The compiler does static branch prediction and code scheduling to reduce (ctrl) or eliminate (data) hazards

What's good about VLIW?

What's bad about VLIW?

Predication

- Predication can be used to eliminate branches by making the execution of an instruction dependent on a “predicate”, e.g.,

```
if (p) {statement 1 } else {statement 2 }
```

would normally compile using two branches. (Why?) With predication it would compile as

```
(p) statement 1

(~p) statement 2
```

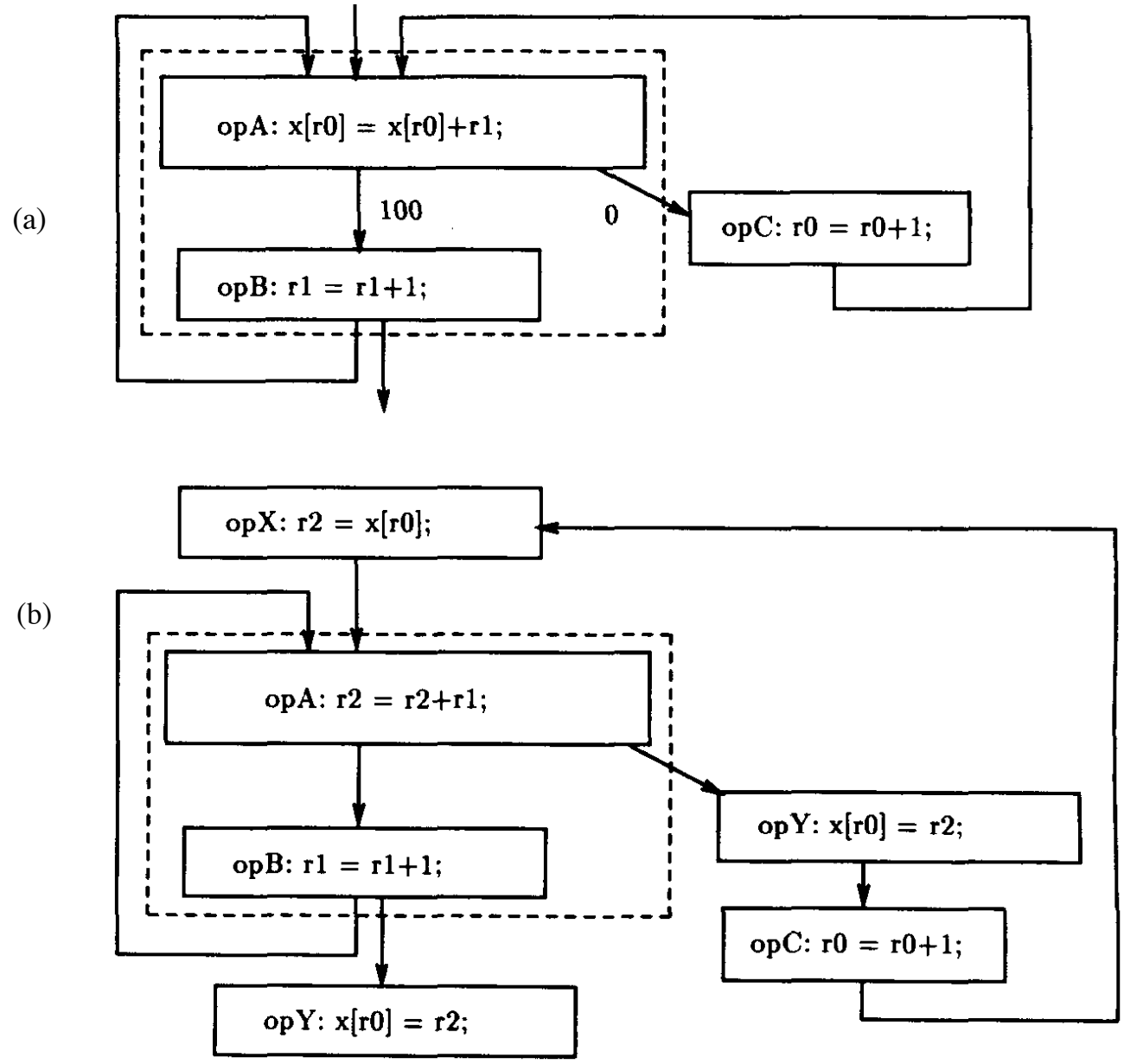
- The use of `(condition)` indicates that the instruction is committed only if `condition` is true
- Predication can be used to speculate as well as to eliminate branches

Speculation

- Speculation is used to allow execution of future instructions that (may) depend on the speculated instruction
  - Speculate on the outcome of a conditional branch (branch prediction)
    - Compare to out-of-order machine with branch prediction
  - Speculate that a store (for which we don't yet know the address) that precedes a load does not refer to the same address, allowing the load to be scheduled before the store (load speculation)

Trace Scheduling

- Original loop allows us to increment either ro or r1: x[ro]=x[ro]+r1
- Profile says incrementing r1 is much more common
- Optimize for that case



Chang et al. SPE Dec. 1991

ILP Summary

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

Flynn's Classification Scheme

- SISD – single instruction, single data stream
  - Uniprocessors
- SIMD – single instruction, multiple data streams
  - single control unit broadcasting operations to multiple datapaths
- MISD – multiple instruction, single data
  - no such machine (although some people put vector machines in this category)
- MIMD – multiple instructions, multiple data streams
  - aka multiprocessors (SMPs, MPPs, clusters, NOWs)

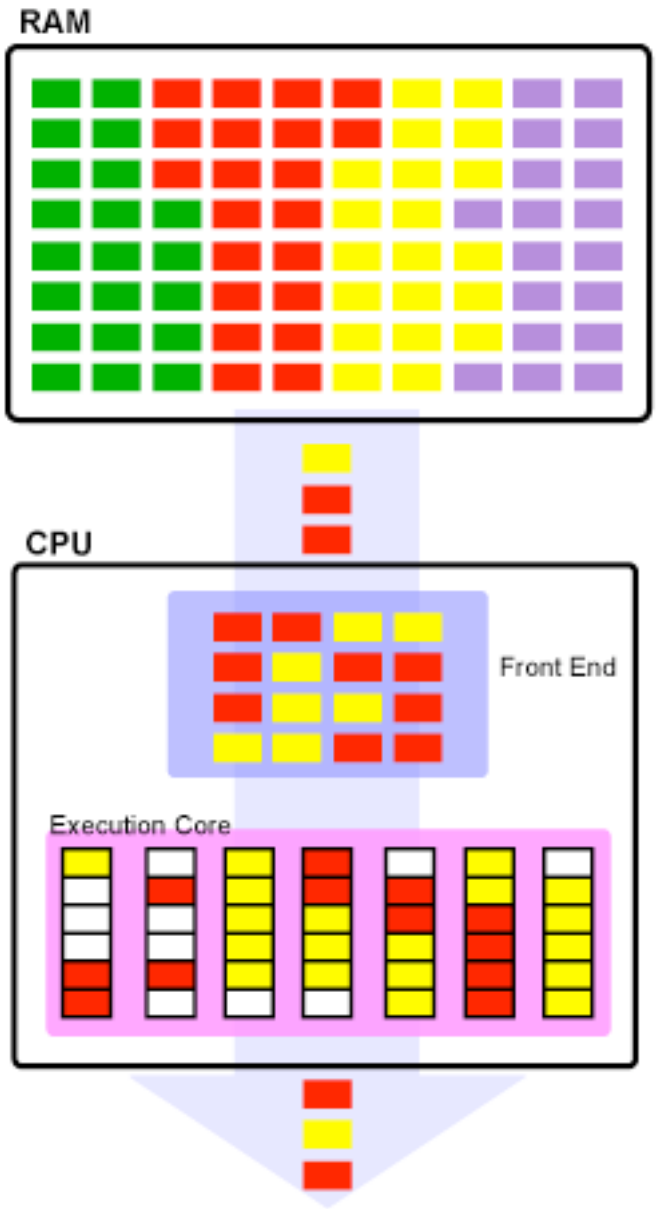
Continuum of Granularity

- "Coarse"
  - Each processor is more powerful
  - Usually fewer processors
  - Communication is more expensive between processors
  - Processors are more loosely coupled
  - Tend toward MIMD
- "Fine"
  - Each processor is less powerful
  - Usually more processors
  - Communication is cheaper between processors
  - Processors are more tightly coupled
  - Tend toward SIMD

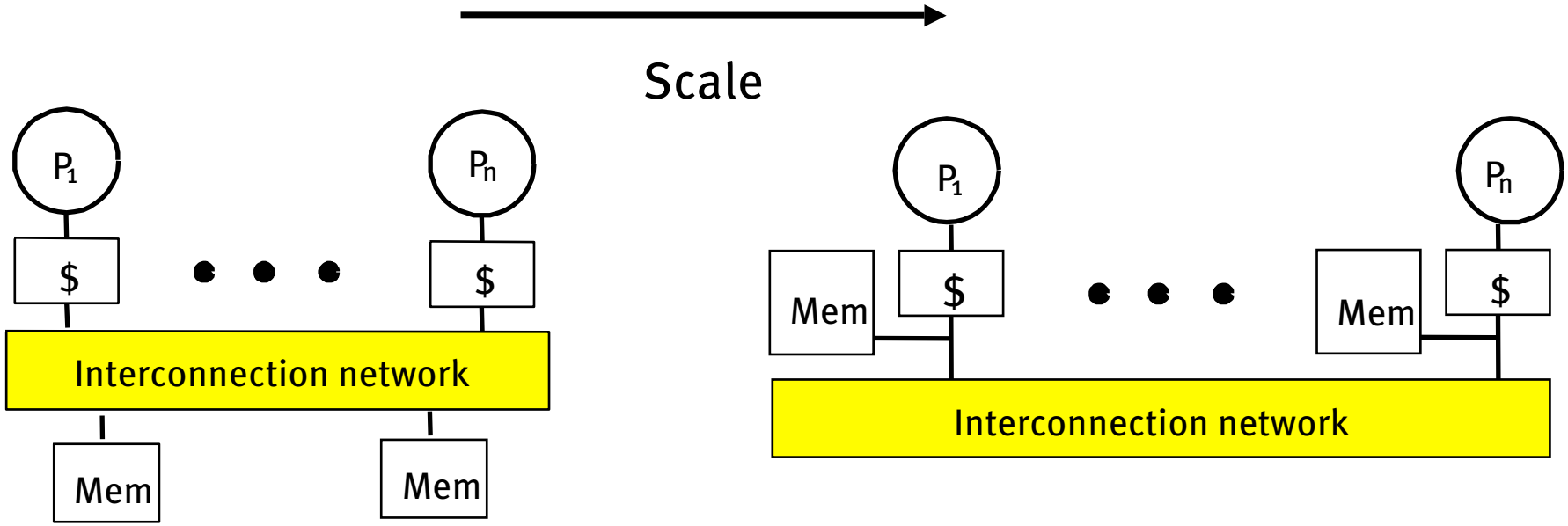
What kind of problems are good for coarse-grained parallelism?

What kind of problems are good for fine-grained parallelism?

Simultaneous multithreading (SMT)



Centralized vs. Distributed Memory



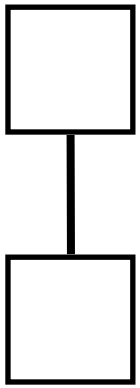
Centralized Memory

Distributed Memory

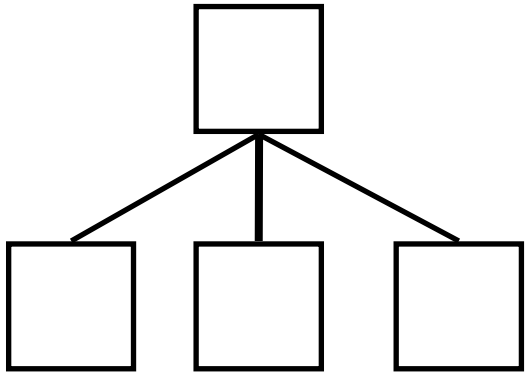
2 Classes of Cache Coherence Protocols

- Directory based — Sharing status of a block of physical memory is kept in just one location, the directory
- Snooping — Every cache with a copy of data also has a copy of sharing status of block, but no centralized state is kept
  - All caches are accessible via some broadcast medium (a bus or switch)
  - All cache controllers monitor or snoop on the medium to determine whether or not they have a copy of a block that is requested on a bus or switch access

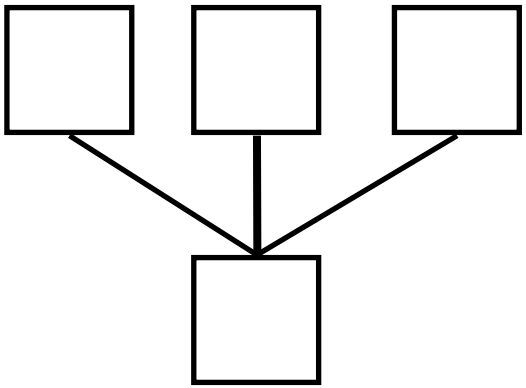
Types of Communication



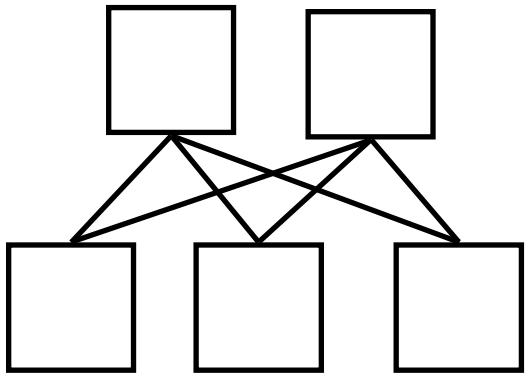
1 to 1



1 to N

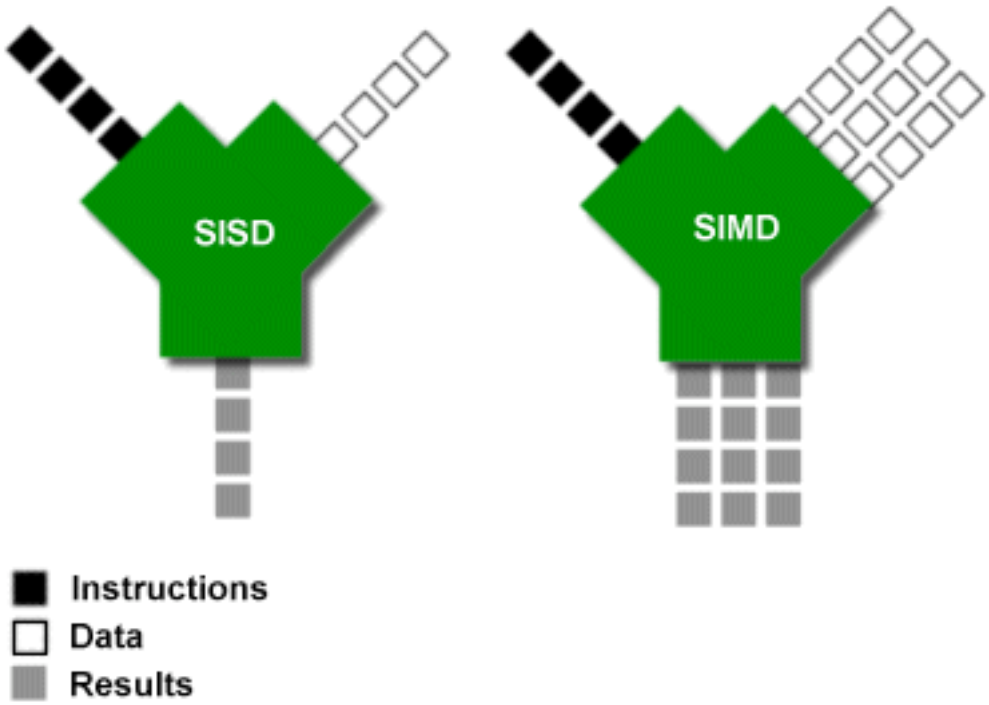


N to 1

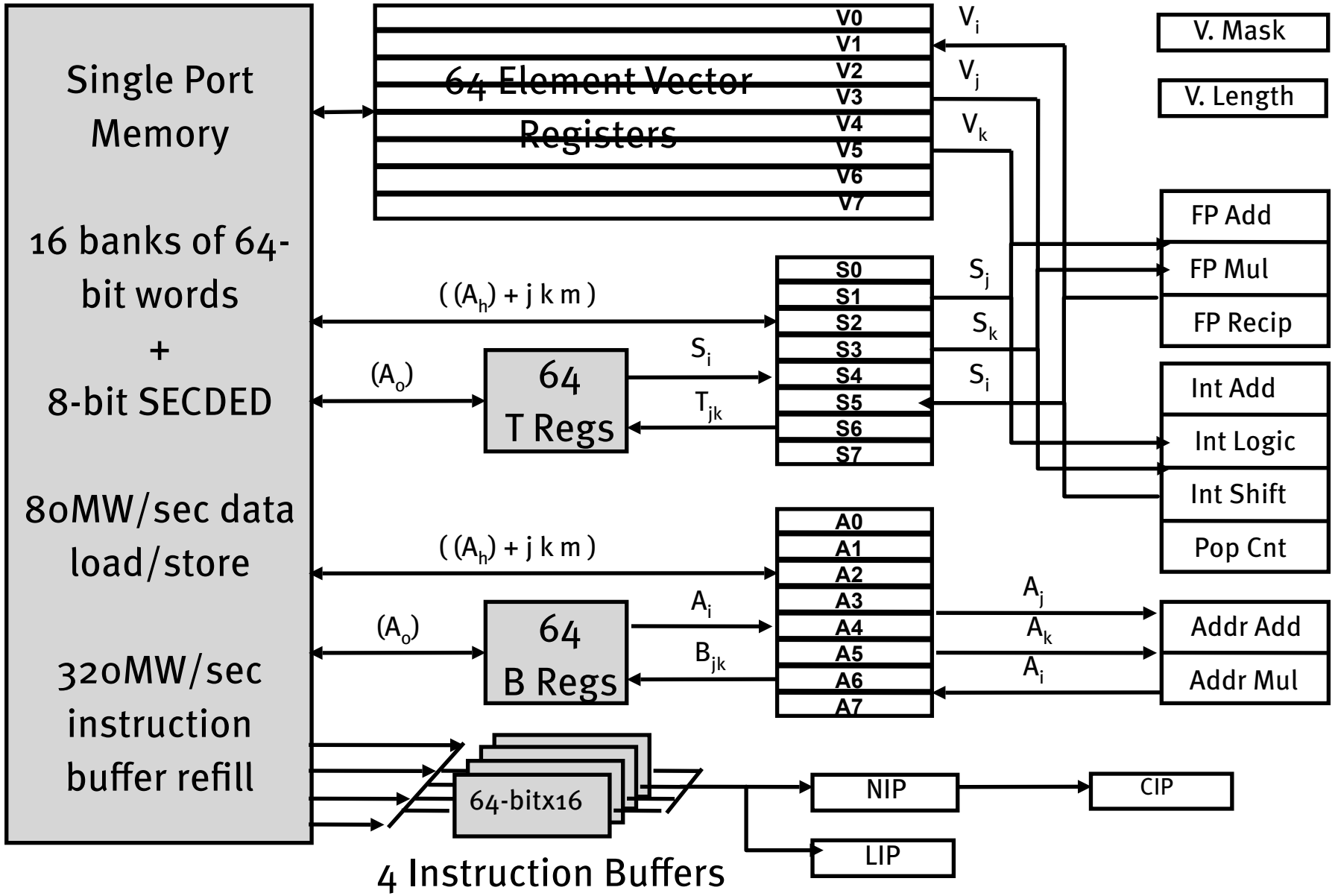


N to M

SIMD Instructions

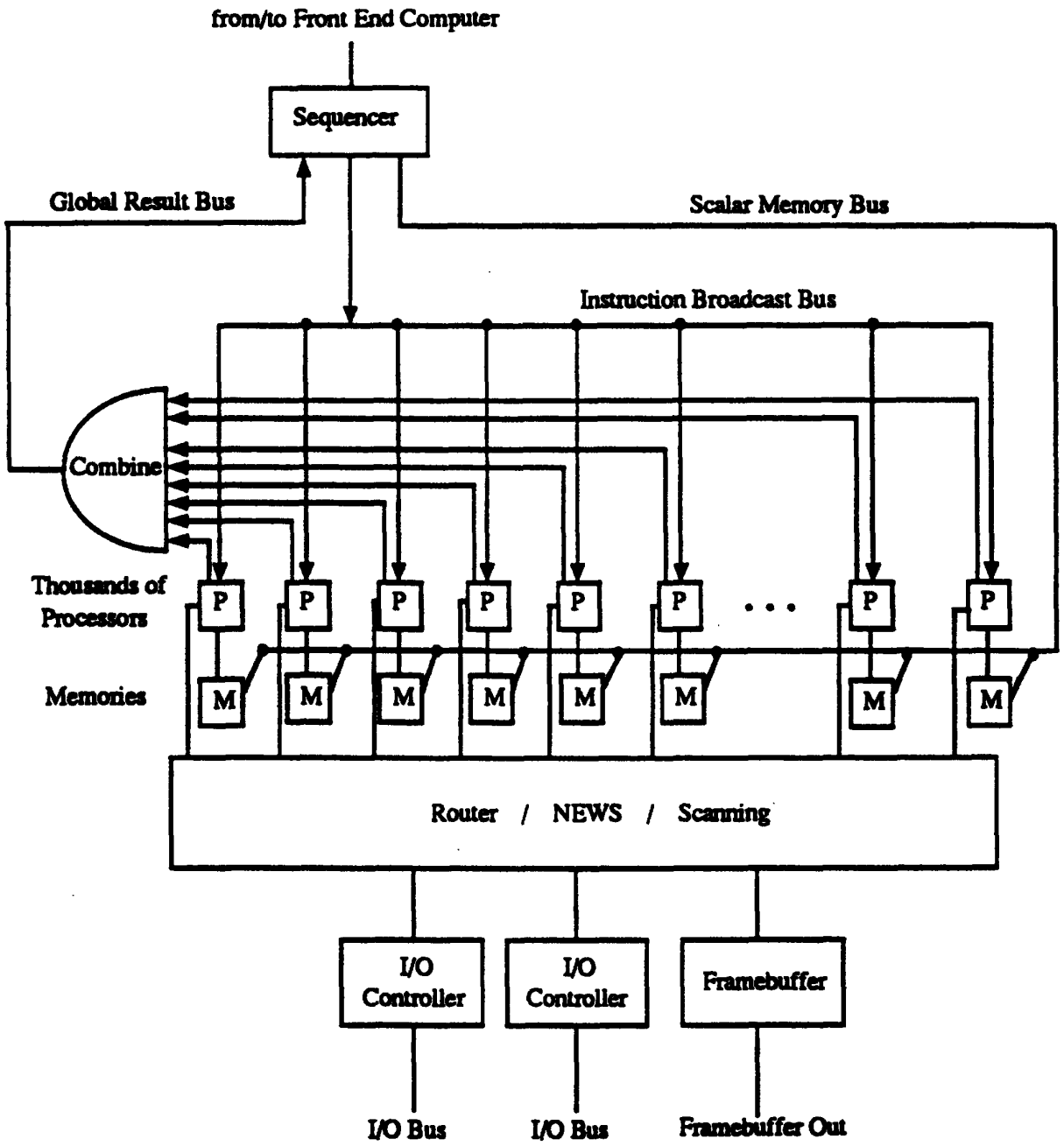


Cray-1 (1976)

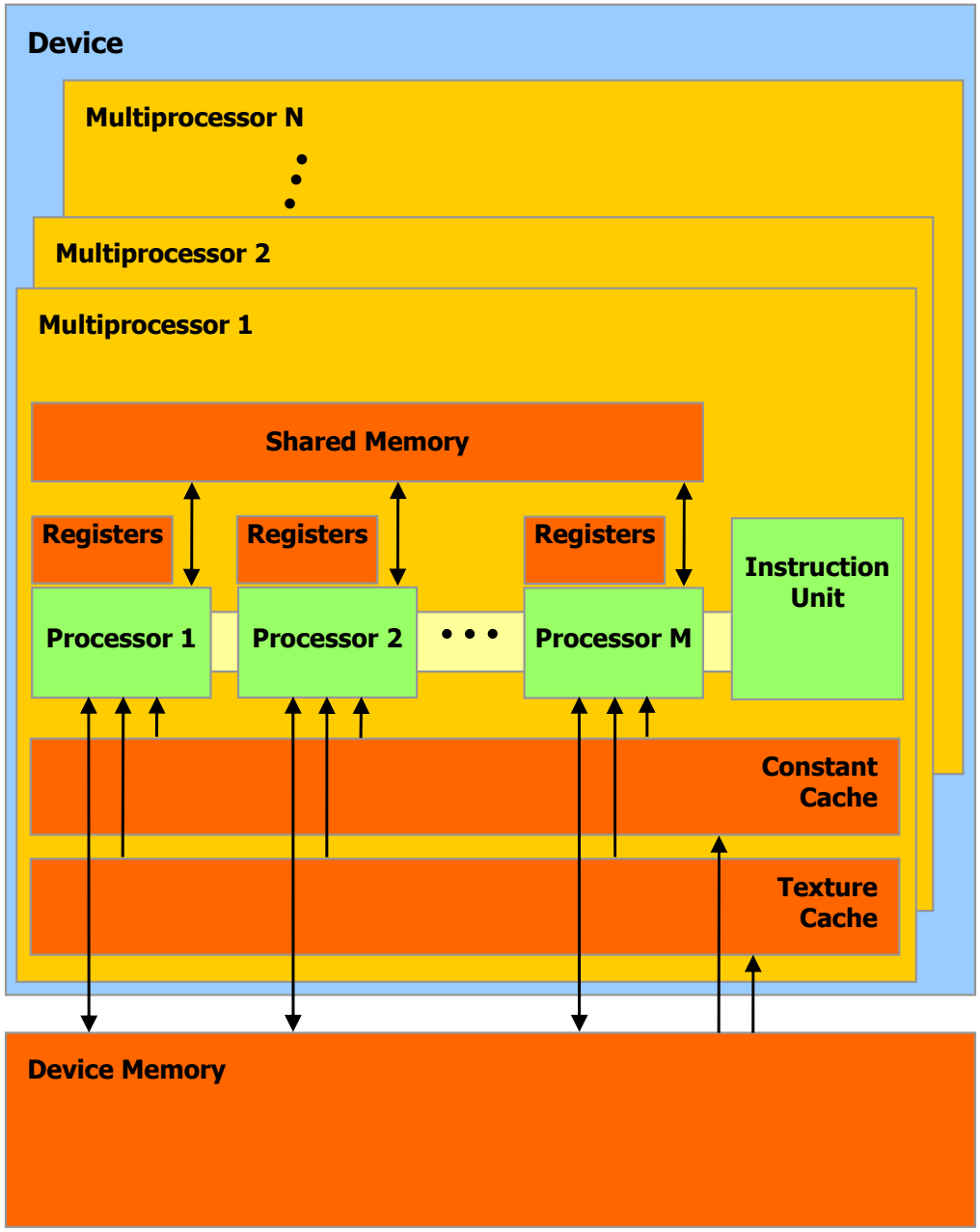


memory bank cycle 50 ns processor cycle 12.5 ns (80 MHz)

CM-2 Hardware Overview



CUDA Hardware Abstraction



Data-Parallel Algorithms

- Efficient algorithms require efficient building blocks
- Data-parallel building blocks
  - Map
  - Gather & Scatter
  - Reduce
  - Scan
  - Sort