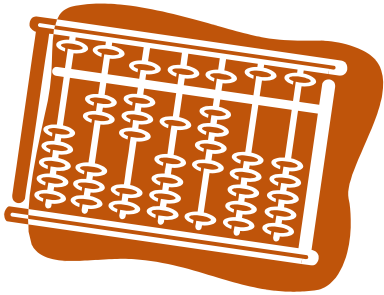


Parallel Programming Principle and Practice

Lecture 2 — Parallel Architecture



Jin, Hai

School of Computer Science and Technology

Huazhong University of Science and Technology

Outline

- Uniprocessor Parallelism
 - Pipelining, Superscalar, Out-of-order execution
 - Vector Processing/SIMD
 - Multithreading: including pThreads
 - Uniprocessor Memory Systems
- Parallel Computer Architecture
 - What is Parallel Architecture?
 - A Parallel Zoo Of Architectures
- Multicore Chips

Parallel architecture

UNIPROCESSOR PARALLELISM

Parallelism is Everywhere

- Modern Processor Chips have \approx 1 billion transistors
 - Clearly must get them working in parallel
 - Question: how much of this parallelism must programmer understand?

- How do uniprocessor computer architectures extract parallelism?
 - By finding parallelism within instruction stream
 - Called “Instruction Level Parallelism” (ILP)
 - The theory: hide parallelism from programmer

Parallelism is Everywhere

- Goal of Computer Architects until about 2002:
 - Hide Underlying Parallelism from everyone: OS, Compiler, Programmer

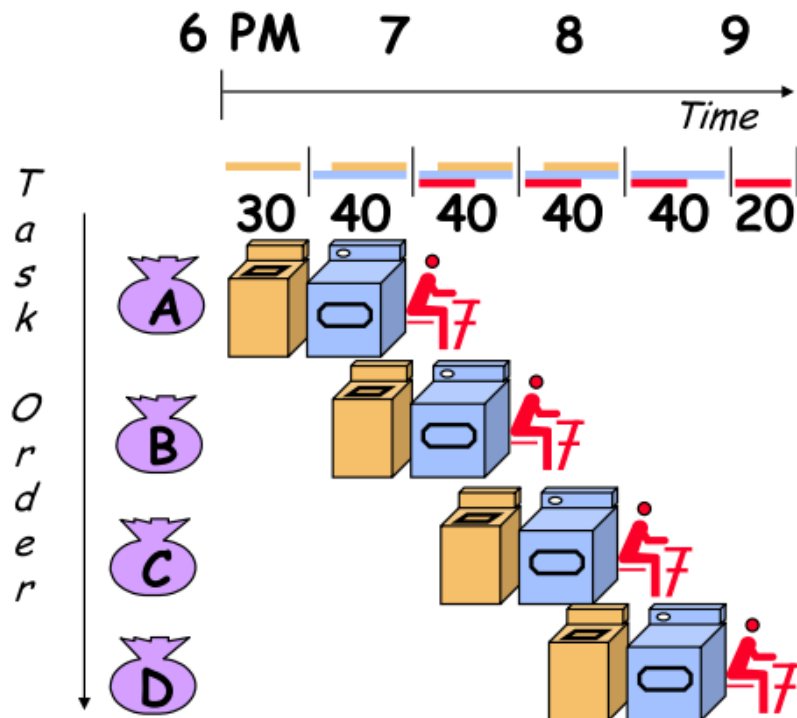
- Examples of ILP techniques
 - Pipelining: Overlapping individual parts of instructions
 - Superscalar execution: Do multiple things at same time
 - VLIW: Let compiler specify which operations can run in parallel
 - Vector Processing: Specify groups of similar (independent) operations
 - Out of Order Execution (OOO): Allow long operations to happen

Parallel architecture

PIPELINING, SUPERSCALAR, OUT-OF-ORDER EXECUTION

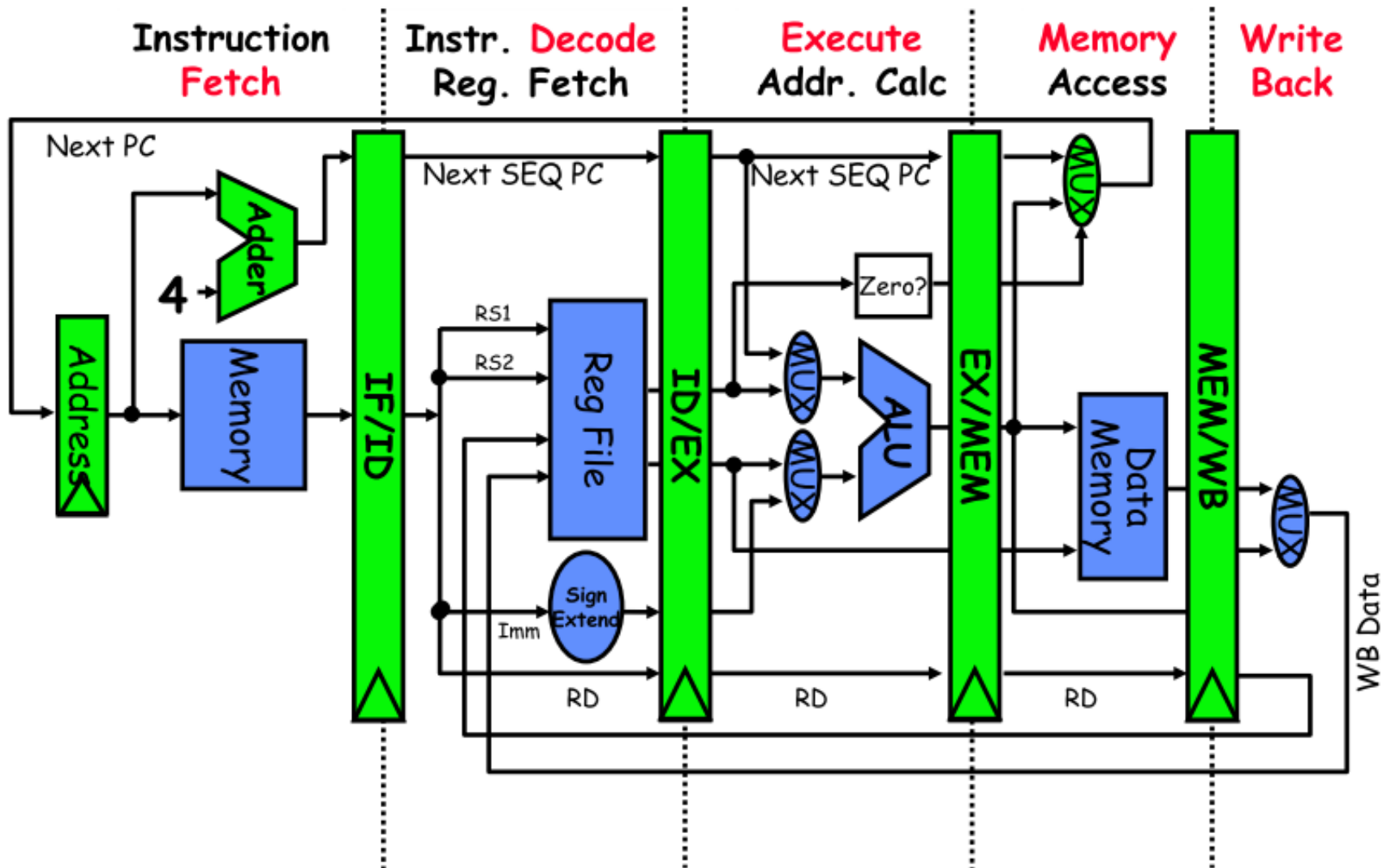
What is Pipelining?

Dave Patterson's Laundry example: 4 people doing laundry
 wash (30 min) + dry (40 min) + fold (20 min) = 90 min **Latency**

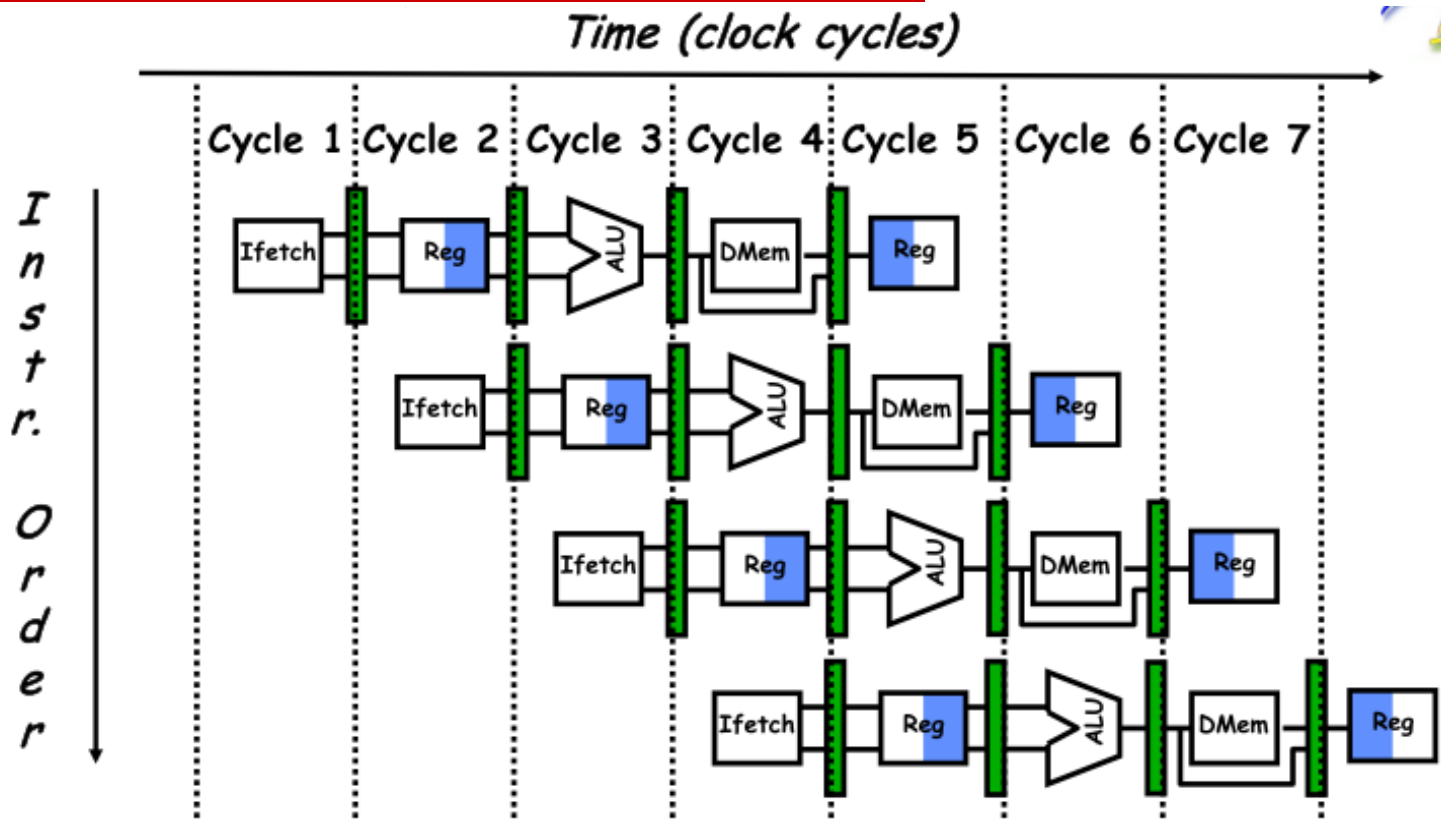


- In this example:
 - Sequential execution takes $4 * 90\text{min} = 6$ hours
 - Pipelined execution takes $30 + 4 * 40 + 20 = 3.5$ hours
- **Bandwidth** = loads/hour
 - $BW = 4/6$ l/h w/o pipelining
 - $BW = 4/3.5$ l/h w pipelining
 - $BW \leq 1.5$ l/h w pipelining, more total loads
- Pipelining helps **bandwidth** but not **latency** (90 min)
- Bandwidth limited by **slowest** pipeline stage
- Potential speedup = **Number of pipe stages**

5 Steps of MIPS Pipeline



Visualizing The Pipeline

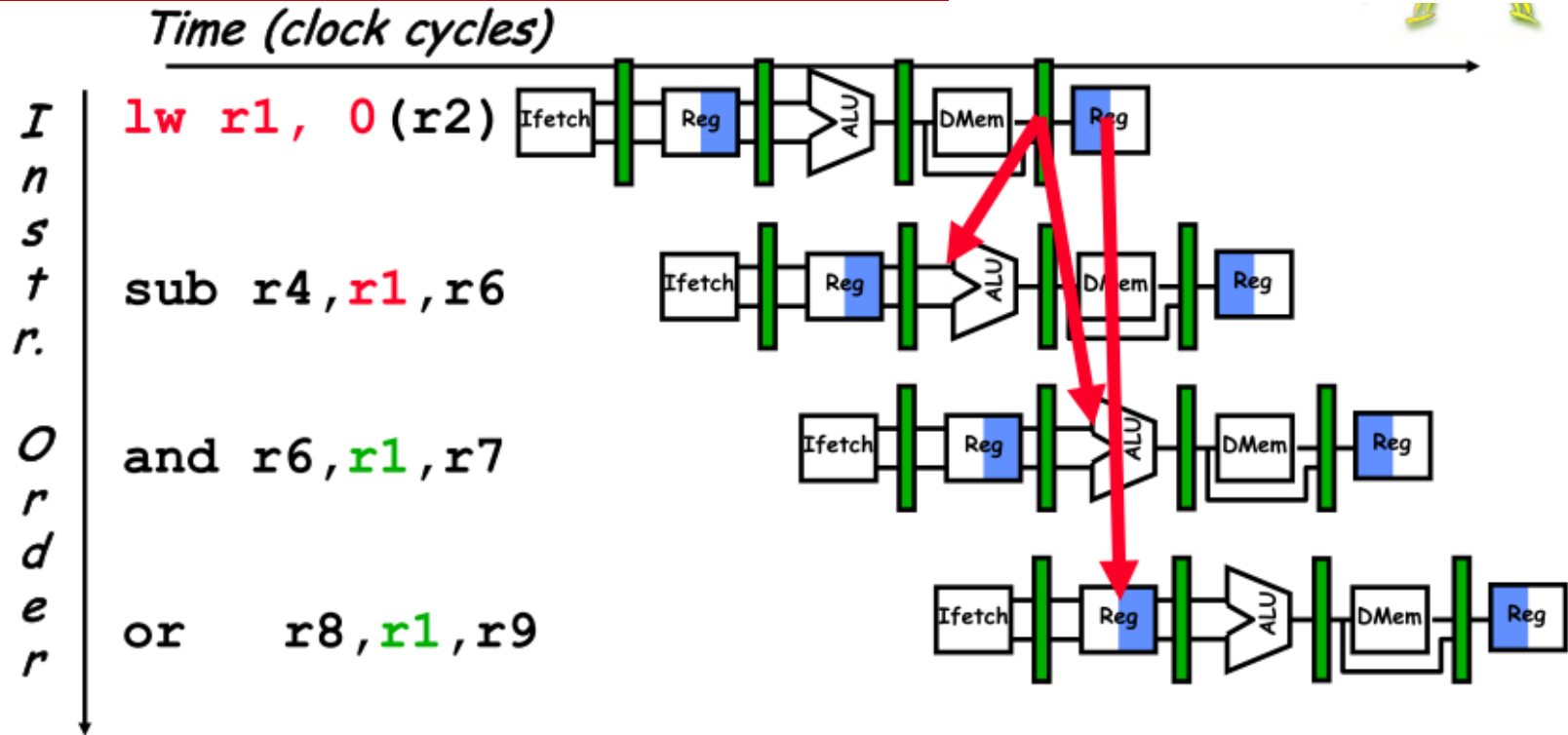


- ❑ In ideal case: CPI (cycles/instruction) = 1!
- On average, put one instruction into pipeline, get one out
- ❑ Superscalar: Launch more than one instruction/cycle
- In ideal case, CPI < 1

Limits to Pipelining

- ❑ Overhead prevents arbitrary division
 - Cost of latches (between stages) limits what can do within stage
 - Sets minimum amount of work/stage
- ❑ **Hazards** prevent next instruction from executing during its designated clock cycle
 - **Structural hazards:** Attempt to use the same hardware to do two different things at once
 - **Data hazards:** Instruction depends on result of prior instruction still in the pipeline
 - **Control hazards:** Caused by delay between the fetching of instructions and decisions about changes in control flow (branches and jumps)
- ❑ Superscalar increases occurrence of hazards
 - More conflicting instructions/cycle

Data Hazard: Must go Back in Time?



- ❑ Data dependencies between adjacent instructions
 - Must wait (“stall”) for result to be done (No “back in time” exists!)
 - Net result is that $CPI > 1$
- ❑ Superscalar increases frequency of hazards

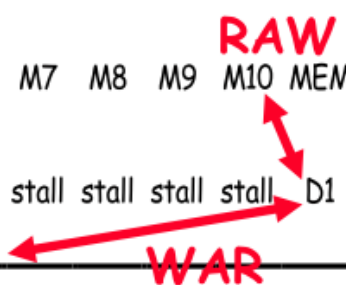
Out-of-Order (OOO) Execution

- Key idea: Allow instructions behind stall to proceed

DIVD **F0,F2,F4**
ADDD **F10,F0,F8**
SUBD **F12,F8,F14**

- Out-of-order execution → out-of-order completion
- Dynamic Scheduling Issues from OOO scheduling
 - Must match up results with consumers of instructions
 - Precise Interrupts

Instruction	Clock Cycle Number																
	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
LD F6,34(R2)	IF	ID	EX	MEM	WB												
LD F2,45(R3)		IF	ID	EX	MEM	WB											
MULTD F0,F2,F4			IF	ID	stall	M1	M2	M3	M4	M5	M6	M7	M8	M9	M10	MEM	WB
SUBD F8,F6,F2				IF	ID	A1	A2	MEM	WB								
DIVD F10,F0,F6					IF	ID	stall	stall	stall	stall	stall	stall	stall	stall	stall	D1	D2
ADDD F6,F8,F2						IF	ID	A1	A2	MEM	WB						



Modern ILP

- Dynamically scheduled, out-of-order execution
 - Current microprocessors fetch 6-8 instructions per cycle
 - Pipelines are 10s of cycles deep → many overlapped instructions in execution at once, although work often discarded

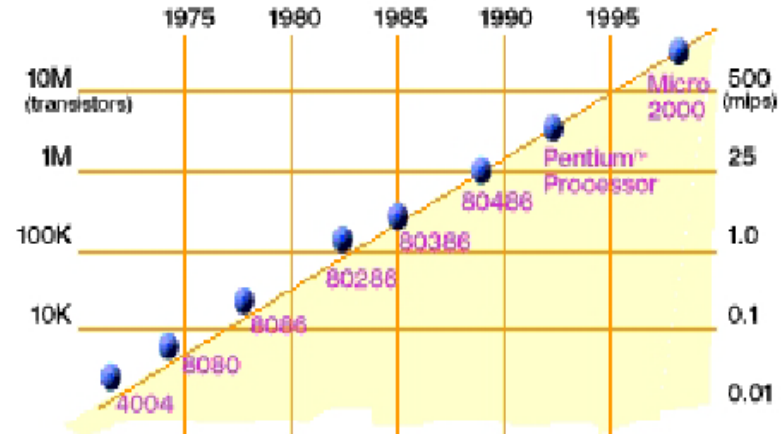
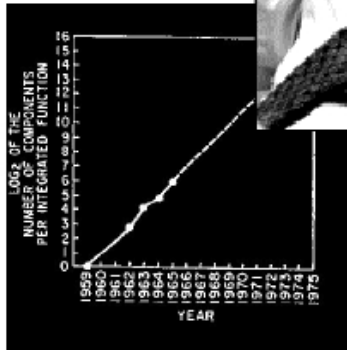
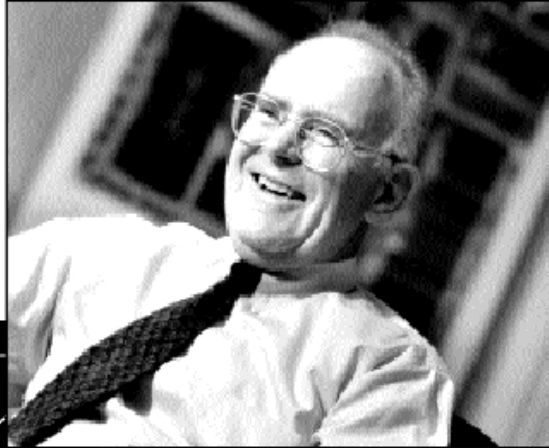
- What happens
 - Grab a bunch of instructions, determine all their dependences, eliminate dep's wherever possible, throw them all into the execution unit, let each one move forward as its dependences are resolved
 - Appears as if executed sequentially

Modern ILP (Cont.)

- Dealing with Hazards: May need to *guess!*
 - Called “Speculative Execution”
 - Speculate on Branch results, Dependencies, even Values!
 - If correct, don't need to stall for result → yields performance
 - If not correct, waste time *and power*
 - Must be able to UNDO a result if guess is wrong
 - Problem: accuracy of guesses decreases with number of simultaneous instructions in pipeline

- Huge complexity
 - Complexity of many components scales as n^2 (issue width)
 - Power consumption big problem

Technology Trends: Moore's Law



2X transistors/Chip Every 1.5 years

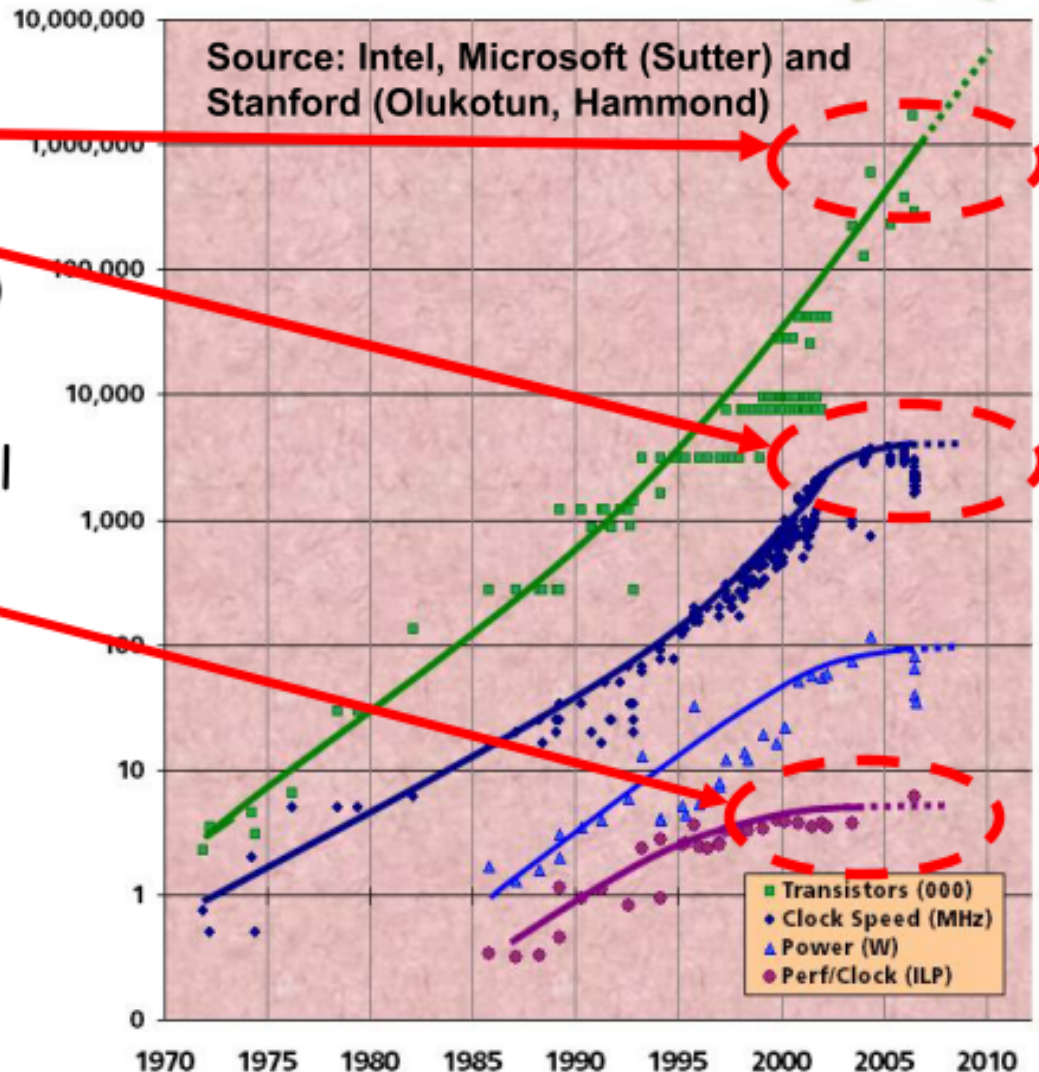
Called “**Moore's Law**”

Gordon Moore (co-founder of Intel) predicted in 1965 that the transistor density of semiconductor chips would double roughly every 18 months.

Microprocessors have become smaller, denser, and more powerful.

Limiting Forces: Clock Speed and ILP

- Chip density is continuing increase
~2x every 2 years
- Clock speed is not
 - # processors/chip (cores) may double instead
- There is little or no more Instruction Level Parallelism (ILP) to be found
 - Can no longer allow programmer to think in terms of a serial programming model
- Conclusion:
Parallelism must be exposed to software!



Parallel architecture

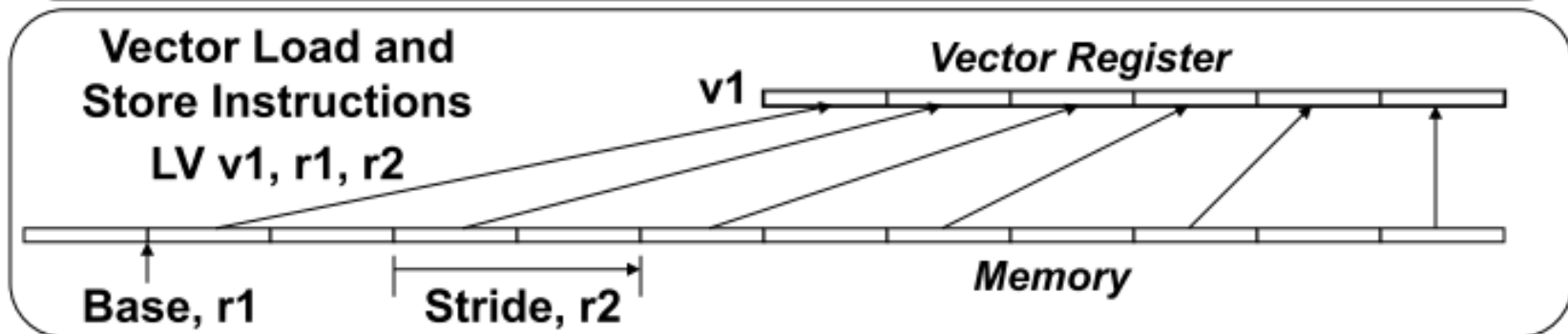
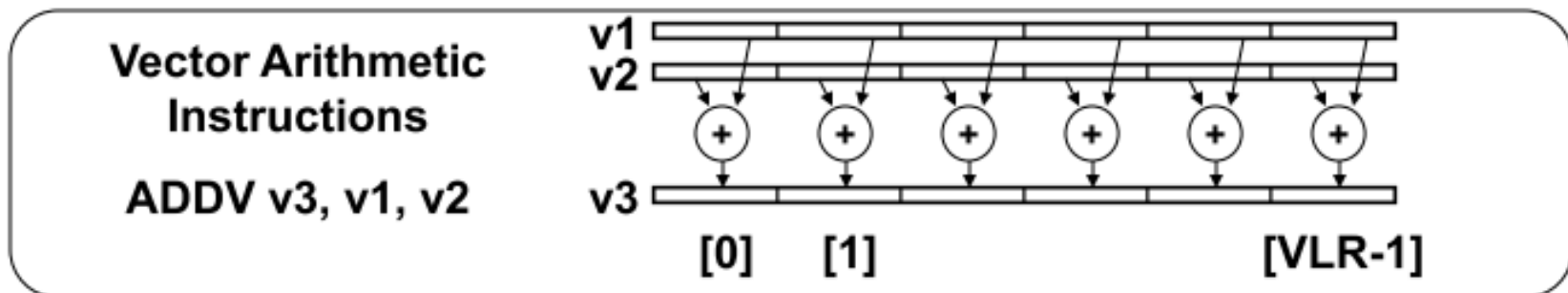
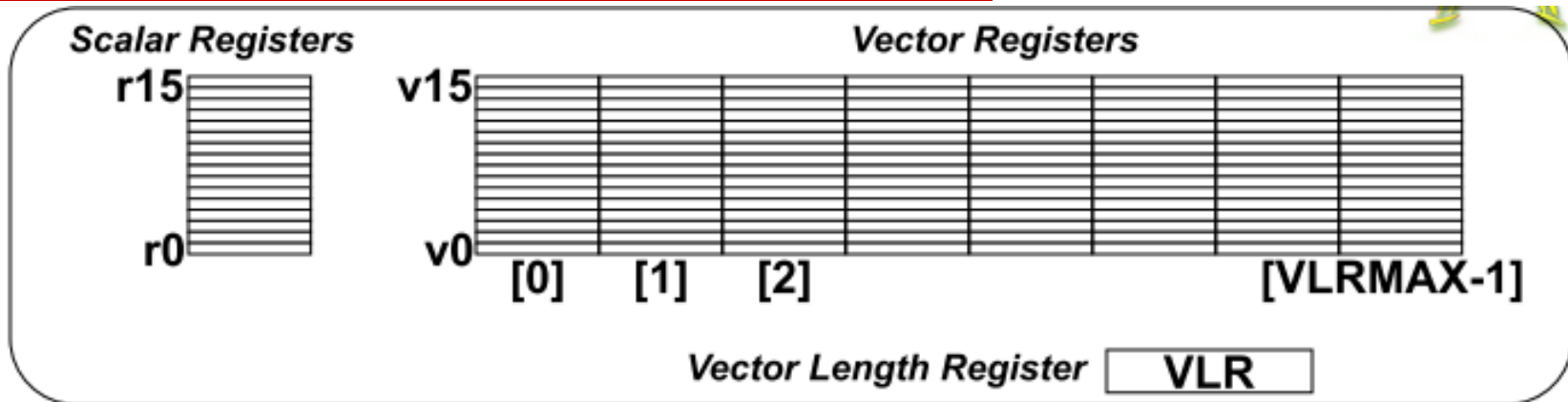
VECTOR PROCESSING/SIMD

Vector Code Example

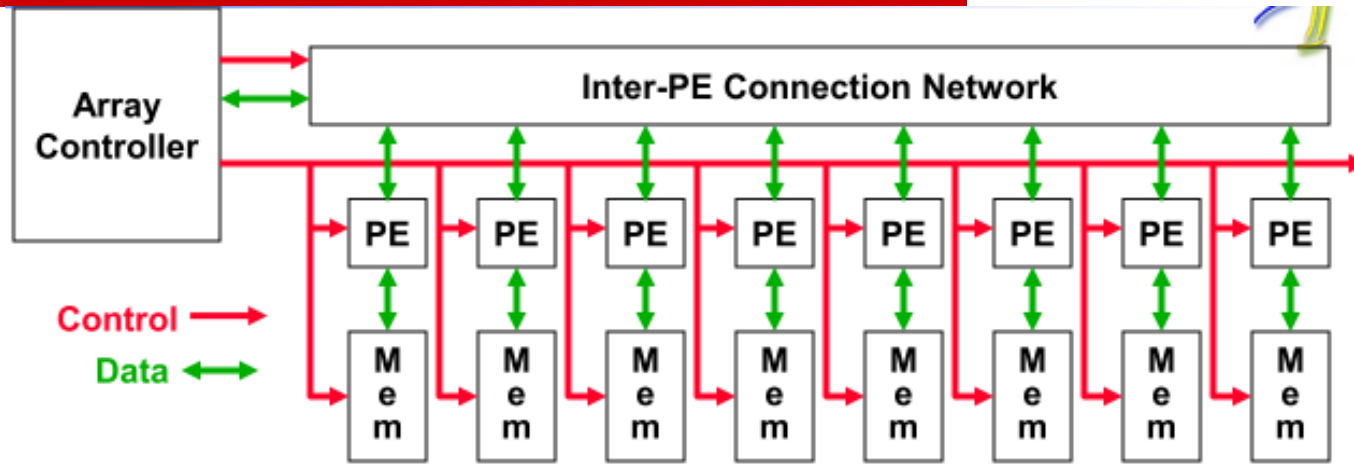
# C code	# Scalar Code	# Vector Code
<pre>for (i=0; i<64; i++) C[i] = A[i] + B[i];</pre>	<pre>LI R4, 64 loop: L.D F0, 0(R1) L.D F2, 0(R2) ADD.D F4, F2, F0 S.D F4, 0(R3) DADDIU R1, 8 DADDIU R2, 8 DADDIU R3, 8 DSUBIU R4, 1 BNEZ R4, loop</pre>	<pre>LI VLR, 64 LV V1, R1 LV V2, R2 ADDV.D V3, V1, V2 SV V3, R3</pre>

- ❑ Require programmer (or compiler) to identify parallelism
 - Hardware does not need to re-extract parallelism
- ❑ Many multimedia/HPC applications are natural consumers of vector processing

Vector Programming Model



SIMD Architecture

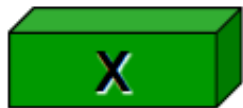


- ❑ Single Instruction Multiple Data (SIMD)
- ❑ Central controller broadcasts instructions to multiple processing elements (PEs)
 - Only requires one controller for whole array
 - Only requires storage for one copy of program
 - All computations are fully synchronized
- ❑ Recent return to popularity
 - GPU (Graphics Processing Units) have SIMD properties
 - However, also multicore behavior, so mix of SIMD and MIMD (more later)
- ❑ Dual between Vector and SIMD execution

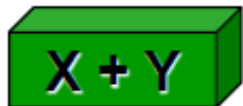
Pseudo SIMD: (Poor-Man's SIMD?)

- Scalar processing

- traditional mode
- one operation produces one result



+

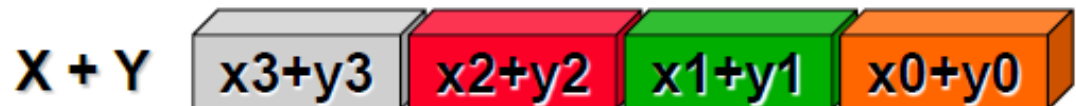


- SIMD processing (Intel)

- with SSE / SSE2
- one operation produces multiple results



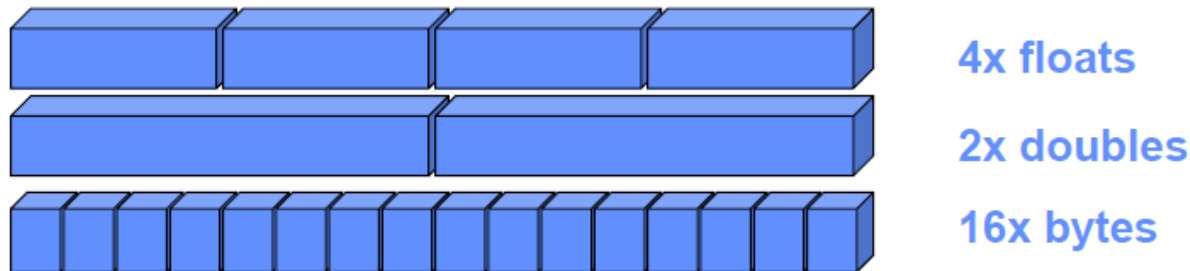
+



Slide Source: Alex Klimovitski & Dean Macri, Intel Corporation

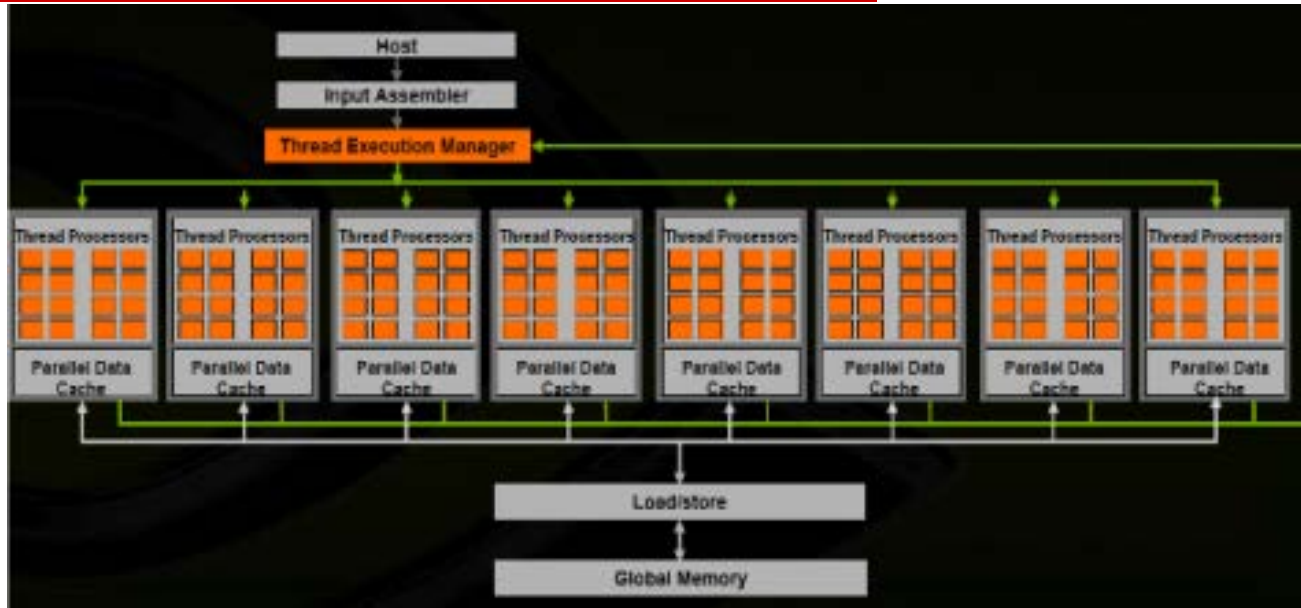
E.g.: SSE / SSE2 SIMD on Intel

- SSE2 data types: anything that fits into 16 bytes, e.g.,



- Instructions perform add, multiply etc. on all the data in this 16-byte register in parallel
- Challenges
 - Need to be contiguous in memory and aligned
 - Some instructions move data from one part of register to another
- In theory, the compiler understands all of this
 - When compiling, it will rearrange instructions to get a good “schedule” that maximizes pipelining, uses FMAs and SIMD
 - It works with the mix of instructions inside an inner loop or other block of code
- But in practice the compiler may need your help

General-Purpose GPUs (GP-GPUs)



- ❑ In 2006, Nvidia introduced GeForce 8800 GPU supporting a new programming language: CUDA
 - Compute Unified Device Architecture
 - OpenCL is a vendor-neutral version of same ideas
- ❑ Idea: Take advantage of GPU computational performance and memory bandwidth to **accelerate some kernels** for general-purpose computing
- ❑ Attached processor model: Host CPU issues data-parallel kernels to GP-GPU for execution

Parallel architecture

MULTITHREADING: INCLUDING PTHREADS

Thread Level Parallelism (TLP)

- ❑ ILP exploits implicit parallel operations within a loop or straight-line code segment
- ❑ TLP explicitly represented by the use of multiple threads of execution that are inherently parallel
 - Threads can be on a single processor
 - Or, on multiple processors
- ❑ **Concurrency vs Parallelism**
 - Concurrency is when two tasks can start, run, and complete in overlapping time periods. It doesn't necessarily mean they'll ever both be running at the same instant
 - For instance, multitasking on a single-threaded machine
 - Parallelism is when tasks literally run at the same time, eg. on a multicore processor
- ❑ **Goal: Use multiple instruction streams to improve**
 - Throughput of computers that run many programs
 - Execution time of multi-threaded programs

Common Notions of Thread Creation

- `cobegin/coend`

```
cobegin
    job1(a1);
    job2(a2);
coend
```

- Statements in block may run in parallel
- `cobegin`s may be nested
- Scoped, so you cannot have a missing `coend`

- `fork/join`

```
tid1 = fork(job1, a1);
job2(a2);
join tid1;
```

- Forked procedure runs in parallel
- Wait at join point if it's not finished

- `future`

```
v = future(job1(a1));
... = ...v...;
```

- Future expression possibly evaluated in parallel
- Attempt to use return value will wait

❑ Threads expressed in the code may not turn into independent computations

- Only create threads if processors idle
- Example: Thread-stealing runtimes such as `cilk`

Overview of POSIX Threads

- ❑ POSIX: Portable Operating System Interface for UNIX
 - Interface to Operating System utilities
- ❑ Pthreads: The POSIX threading interface
 - System calls to create and synchronize threads
 - Should be relatively uniform across UNIX-like OS platforms
 - Originally IEEE POSIX 1003.1c
- ❑ Pthreads contain support for
 - Creating parallelism
 - Synchronizing
 - No explicit support for communication, because shared memory is implicit; a pointer to shared data is passed to a thread
 - Only for HEAP! Stacks not shared

Forking POSIX Threads

Signature:

```
int pthread_create(pthread_t *,
                  const pthread_attr_t *,
                  void * (*)(void *),
                  void *);
```

Example call:

```
errcode = pthread_create(&thread_id; &thread_attribute; &thread_fun; &fun_arg);
```

- ❑ `thread_id` is the thread id or handle (used to halt, etc.)
- ❑ `thread_attribute` various attributes
 - Standard default values obtained by passing a NULL pointer
 - Sample attribute: minimum stack size
- ❑ `thread_fun` the function to be run (takes and returns void*)
- ❑ `fun_arg` an argument can be passed to `thread_fun` when it starts
- ❑ `errorcode` will be set nonzero if the create operation fails

Simple Threading Example (pThreads)

```
void* SayHello(void *foo) {  
    printf( "Hello, world!\n" );  
    return NULL;  
}
```

E.g., compile using `gcc -lpthread`

```
int main() {  
    pthread_t threads[16];  
    int tn;  
    for(tn=0; tn<16; tn++) {  
        pthread_create(&threads[tn], NULL, SayHello, NULL);  
    }  
    for(tn=0; tn<16 ; tn++) {  
        pthread_join(threads[tn], NULL);  
    }  
    return 0;  
}
```

Shared Data and Threads

- ❑ Variables declared outside of main are shared
- ❑ Objects allocated on the **heap** may be shared (if pointer is passed)
- ❑ Variables on the **stack** are private: passing pointer to these around to other threads can cause problems
- ❑ Often done by creating a large “thread data” struct, which is passed into all threads as argument

```
char *message = "Hello World!\n";
```

```
pthread_create(&thread1, NULL,  
              print_fun,(void*) message);
```

Loop Level Parallelism

- Many application have parallelism in loops

```
double stuff [n][n];  
for (int i = 0; i < n; i++)  
    for (int j = 0; j < n; j++)  
        ... pthread_create (... , update_stuff, ..., &stuff[i][j]);
```

- But overhead of thread creation is nontrivial

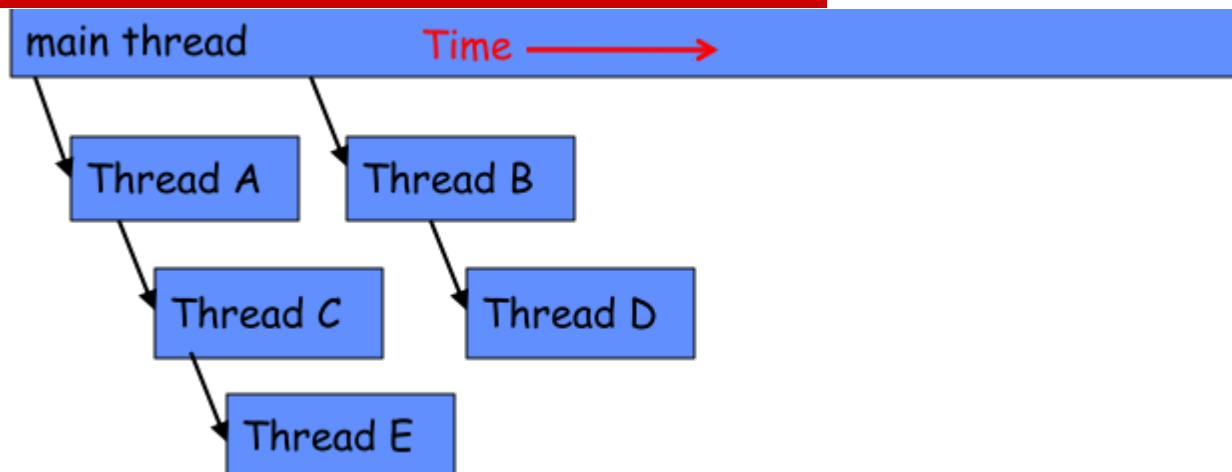
- update_stuff should have a significant amount of work

- Common Performance Pitfall: Too many threads

- The cost of creating a thread is 10's of thousands of cycles on modern architectures

- Solution: Thread blocking: use a small # of threads, often equal to the number of cores/processors or hardware threads

Thread Scheduling



- Once created, when will a given thread run?
 - It is up to the operating system or hardware, but it will run eventually, even if you have more threads than cores
 - But-scheduling may be non-ideal for your application
- Programmer can provide hints or affinity in some cases
 - E.g., create exactly P threads and assign to P cores
- Can provide user-level scheduling for some systems
 - Application-specific tuning based on programming model

Multithreaded Execution

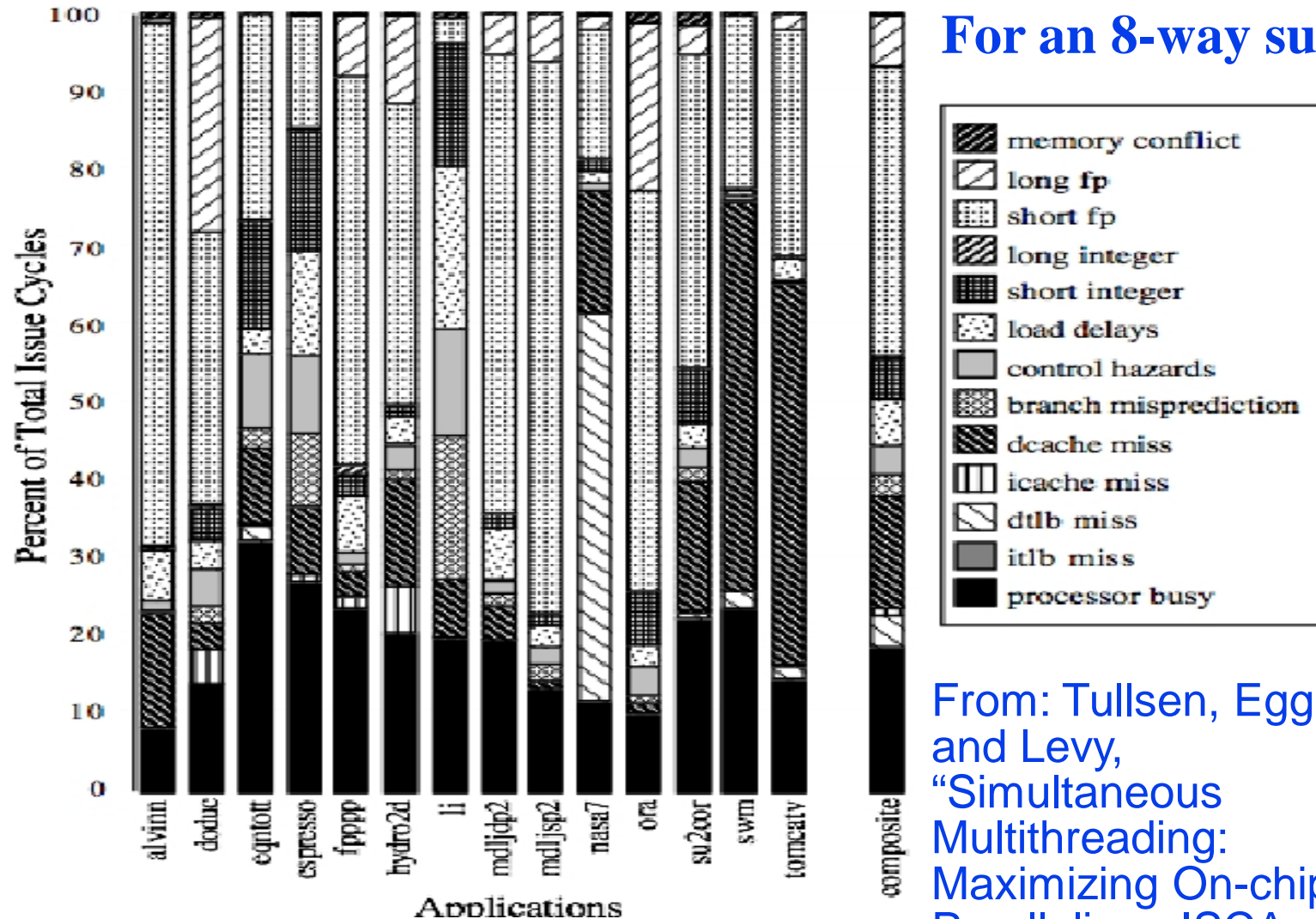
- Multitasking operating system
 - Gives “illusion” that multiple things happen at same time
 - Switches at a coarse-grained time (for instance: 10ms)
- Hardware Multithreading: multiple threads share processor simultaneously (with little OS help)
 - Hardware does switching
 - HW for fast thread switch in small number of cycles
 - much faster than OS switch which is 100s to 1000s of clocks
 - Processor duplicates independent state of each thread
 - e.g., a separate copy of register file, a separate PC, and for running independent programs, a separate page table
 - Memory shared through the virtual memory mechanisms, which already support multiple processes
- When to switch between threads?
 - Alternate instruction per thread (fine grain)
 - When a thread is stalled, perhaps for a cache miss, another thread can be executed (coarse grain)

What about combining ILP and TLP?

- TLP and ILP exploit two different kinds of parallel structure in a program
- Could a processor oriented at ILP benefit from exploiting TLP?
 - functional units are often idle in data path designed for ILP because of either stalls or dependences in the code
 - TLP used as a source of independent instructions that might keep the processor busy during stalls
 - TLP be used to occupy functional units that would otherwise lie idle when insufficient ILP exists
- Called “Simultaneous Multithreading”
 - Intel renamed this “Hyperthreading”

Quick Recall: Many Resources IDLE!

For an 8-way superscalar



From: Tullsen, Eggers, and Levy, "Simultaneous Multithreading: Maximizing On-chip Parallelism," ISCA, 1995

Simultaneous Multi-threading

One thread, 8 units

Cycle M M FX FX FP FP BR CC

1	█							█
2	█	█					█	
3			█	█				
4								
5								
6								
7	█		█		█			
8		█		█				
9			█					

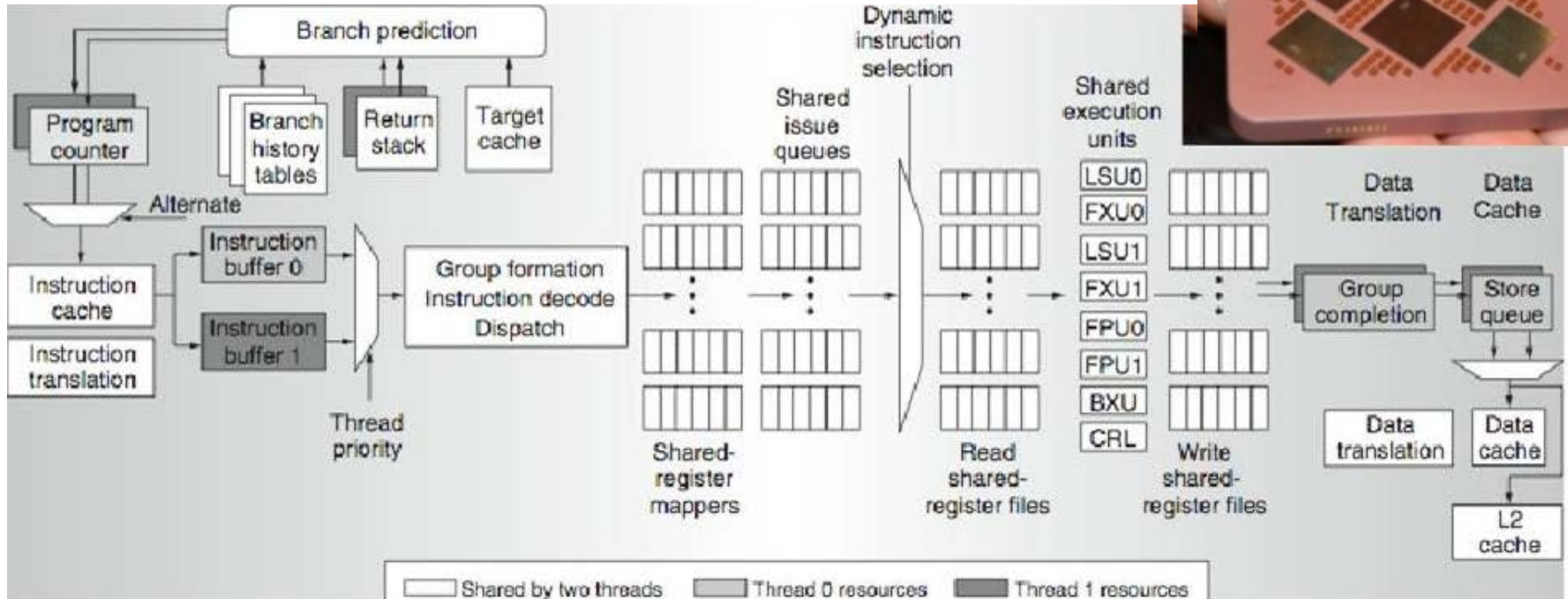
Two threads, 8 units

Cycle M M FX FX FP FP BR CC

1	█	█	█					█
2	█	█	█			█	█	
3	█			█	█			
4	█	█				█		
5		█						█
6								
7	█		█	█	█	█		
8		█		█	█	█		
9	█	█		█		█		

M = Load/Store, FX = Fixed Point, FP = Floating Point, BR = Branch, CC = Condition Codes

Power5 Dataflow



□ Why only two threads?

- With 4, one of the shared resources (physical registers, cache, memory bandwidth) would be prone to bottleneck

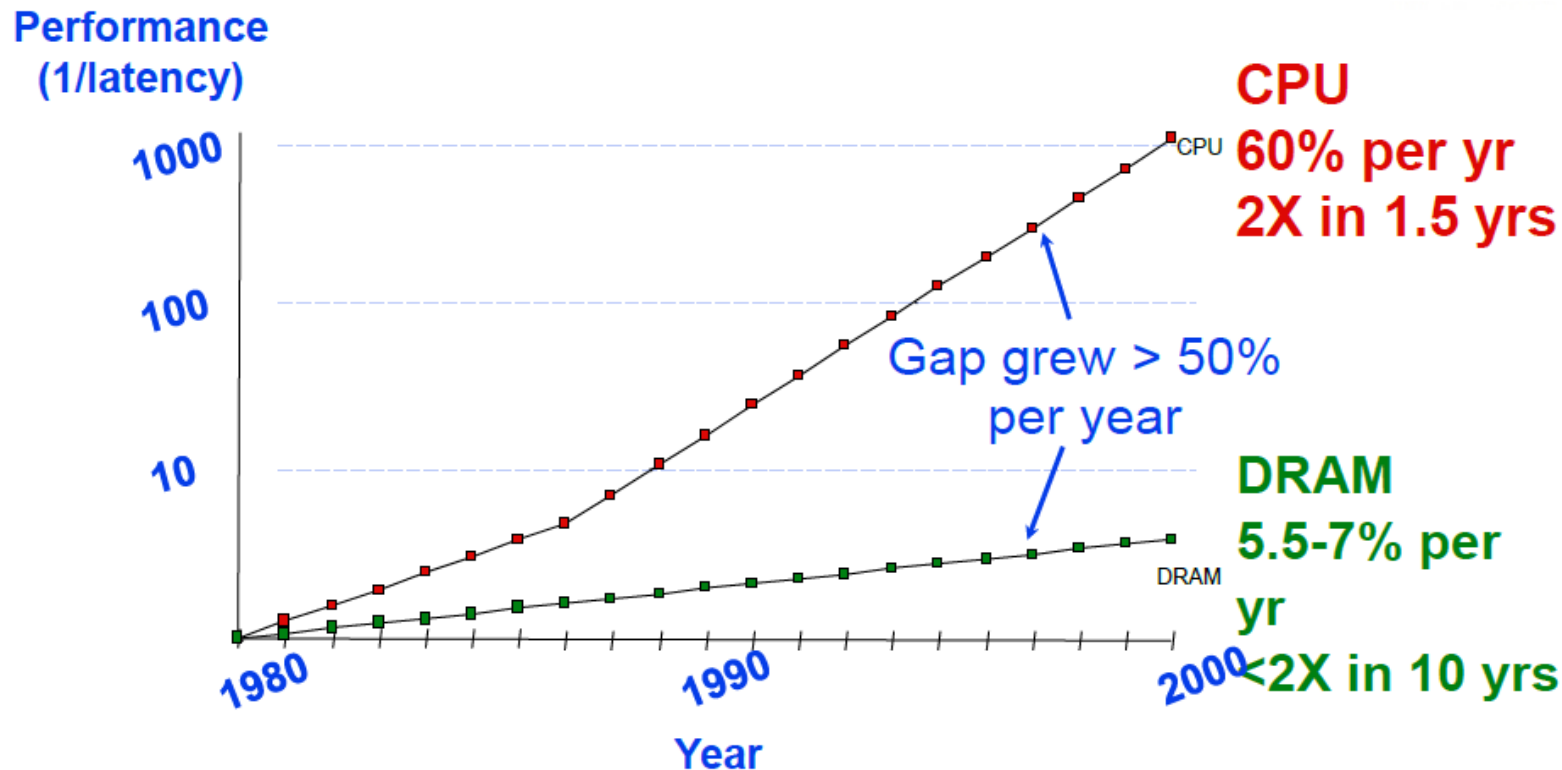
□ Cost

- The Power5 core is about 24% larger than the Power4 core because of the addition of SMT (Simultaneous Multi-threading) support

Parallel architecture

UNIPROCESSOR MEMORY SYSTEMS

Limiting Force: Memory Wall



- How do architects address this gap?
 - Put small, fast “cache” memories between CPU and DRAM (Dynamic Random Access Memory).
 - Create a “memory hierarchy”

Principle of Locality

□ Principle of Locality

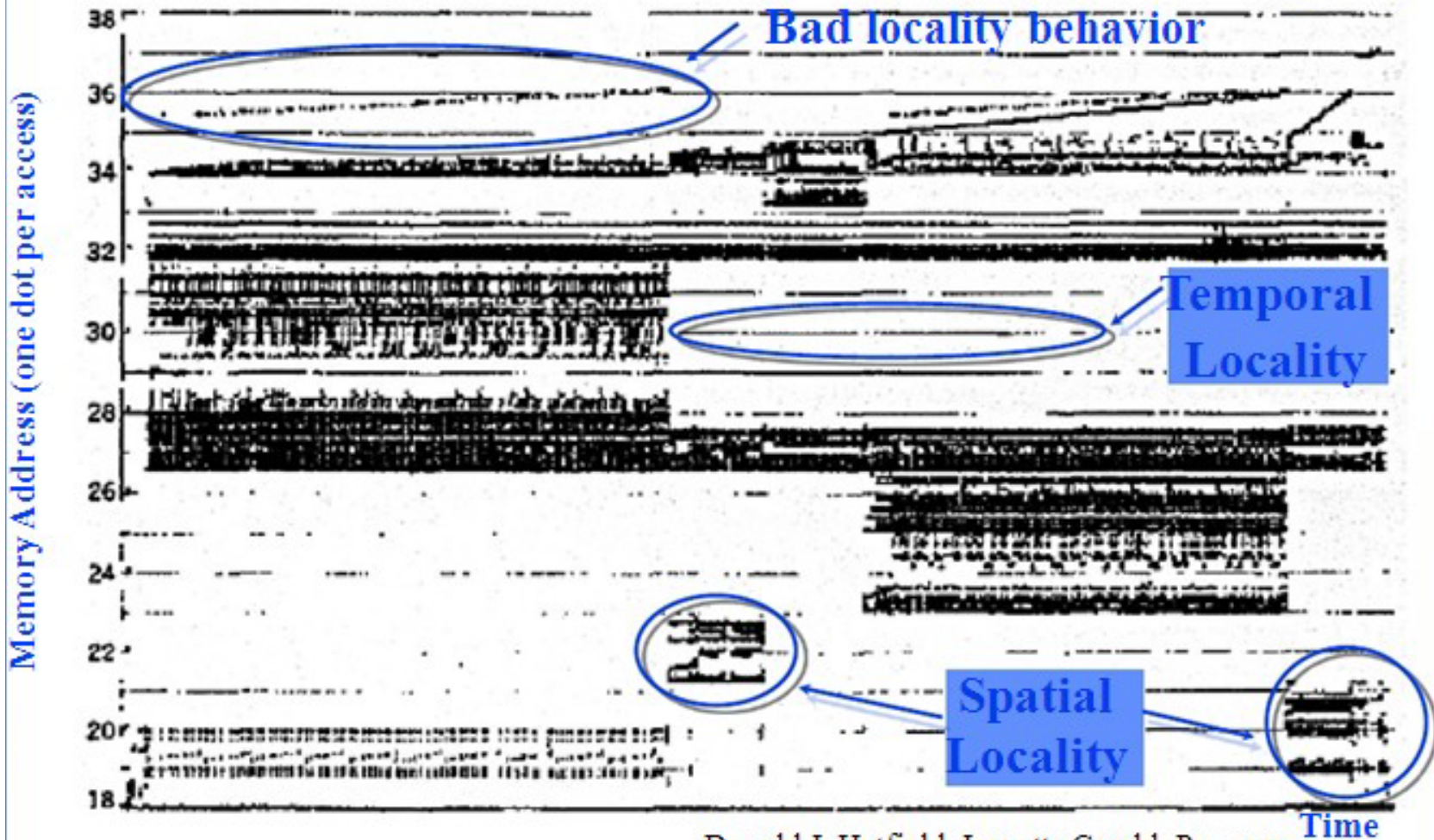
- Program access a relatively small portion of the address space at any instant of time

□ Two Different Types of Locality

- *Temporal Locality* (Locality in Time): If an item is referenced, it will tend to be referenced again soon (e.g., loops, reuse)
- *Spatial Locality* (Locality in Space): If an item is referenced, items whose addresses are closeby tend to be referenced soon (e.g., straightline code, array access)

□ Last 25 years, HW relied on locality for speed

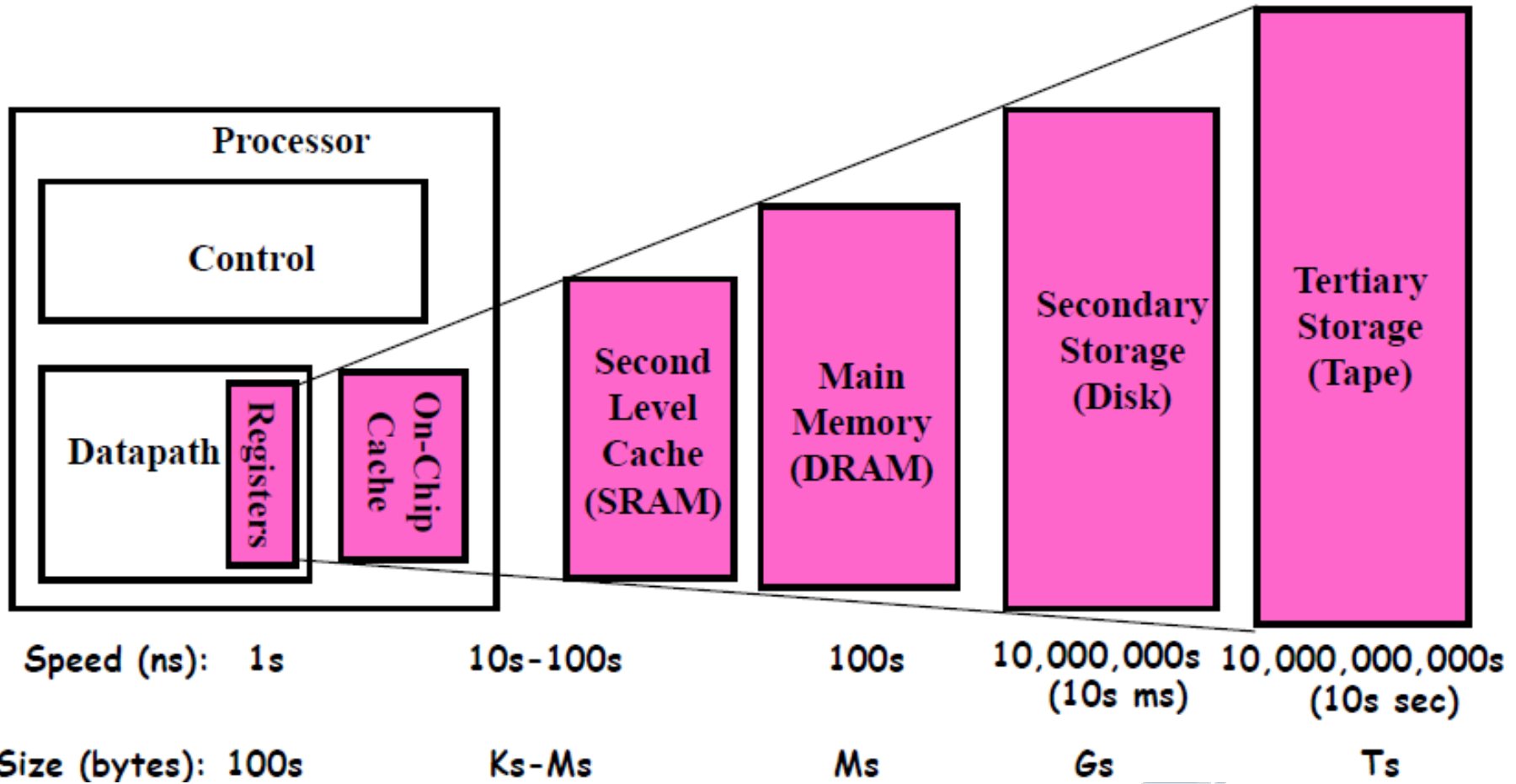
Programs with locality cache well



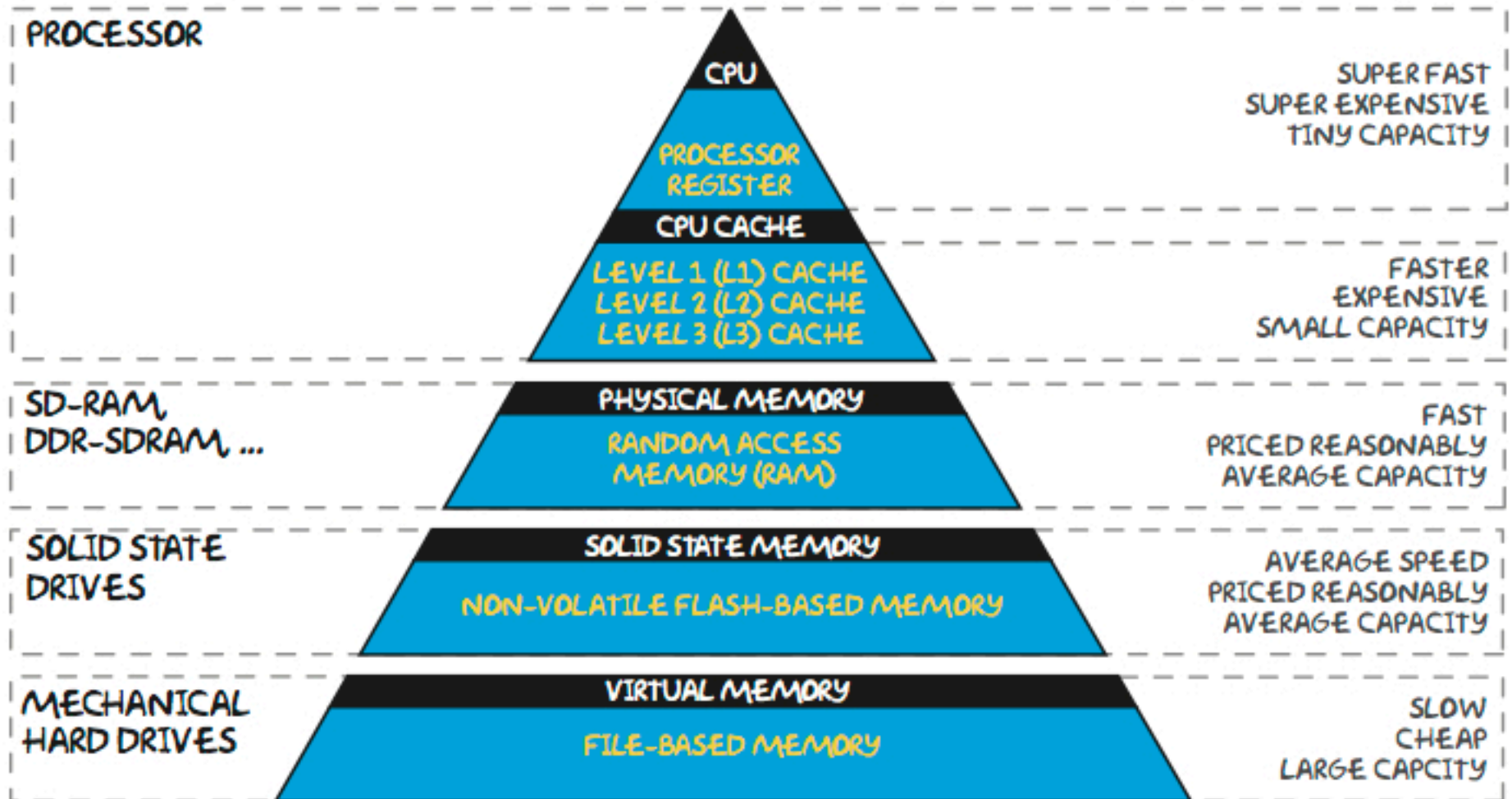
Donald J. Hatfield, Jeanette Gerald: Program Restructuring for Virtual Memory. IBM Systems Journal 10(3): 168-192

Memory Hierarchy

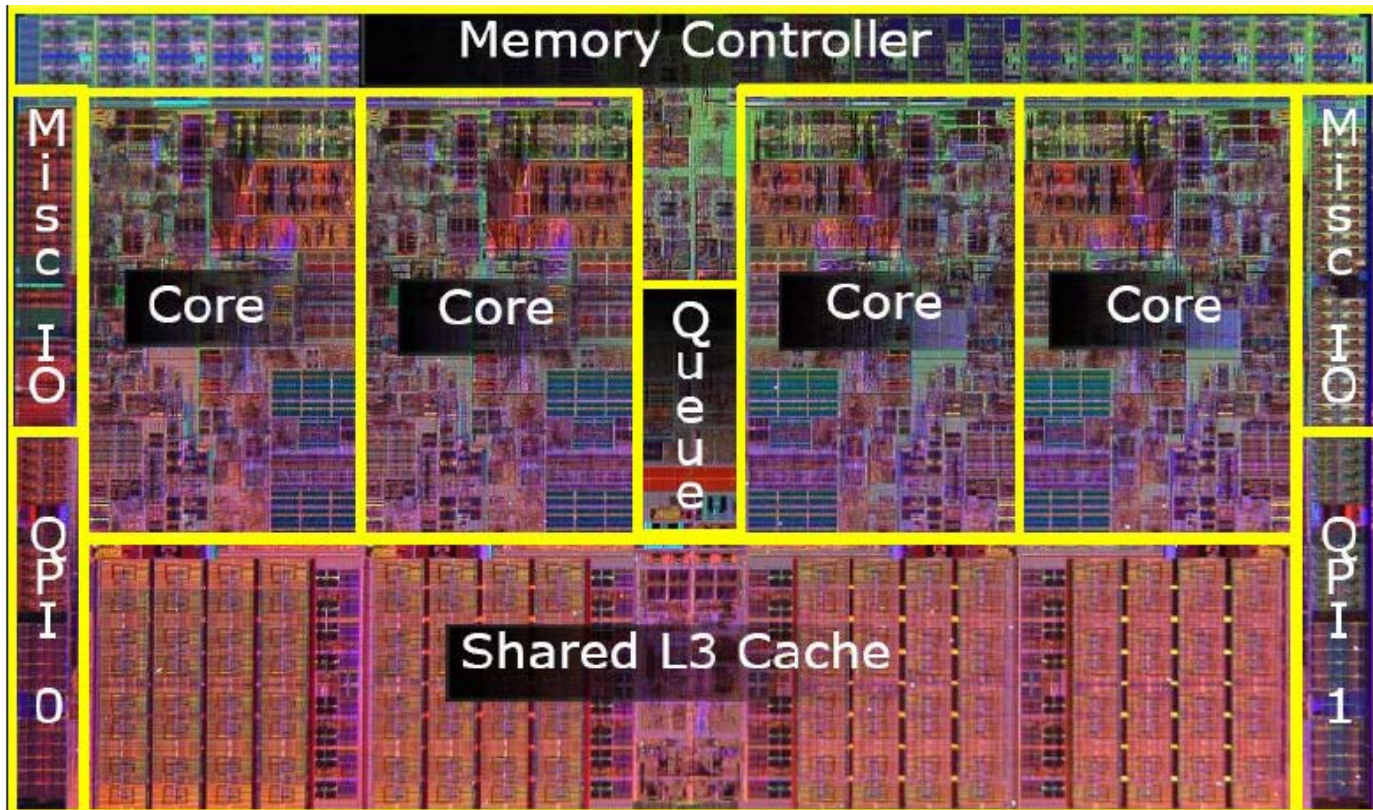
- Take advantage of the principle of locality to
 - Present as much memory as in the cheapest technology
 - Provide access at speed offered by the fastest technology



Memory Hierarchy



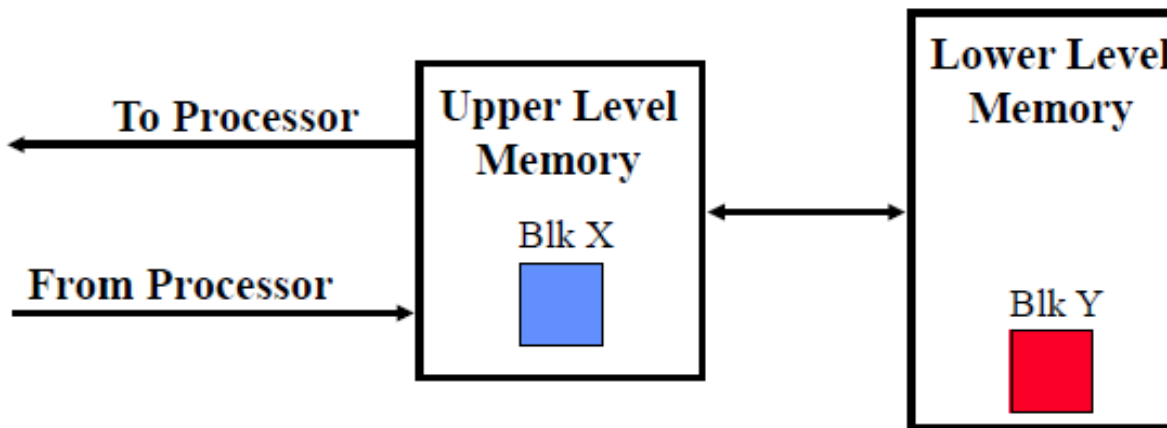
Example of Modern Core: Nehalem



- ❑ ON-chip cache resources
 - For each core: L1: 32K instruction and 32K data cache, L2: 1MB, L3: 8MB shared among all 4 cores
- ❑ Integrated, on-chip memory controller (DDR3)

Memory Hierarchy: Terminology

- Hit: data appears in some blocks in the upper level (example: Block X)
 - Hit Rate: the fraction of memory access found in the upper level
 - Hit Time: Time to access the upper level which consists of RAM access time + Time to determine hit/miss
- Miss: data needs to be retrieve from a block in the lower level (Block Y)
 - Miss Rate = $1 - (\text{Hit Rate})$
 - Miss Penalty: Time to replace a block in the upper level + Time to deliver the block the processor
- Hit Time \ll Miss Penalty (500 instructions on 21264!)

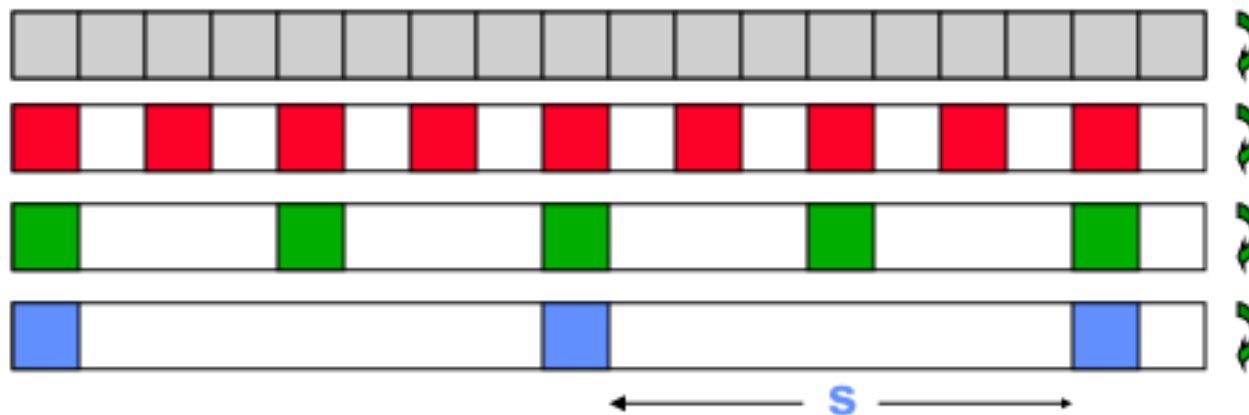


Impact of Hierarchy on Algorithms

- Today CPU time is a function of (ops, cache misses)
- What does this mean to compilers, data structures, algorithms?
 - Quicksort: fastest comparison based sorting algorithm when keys fit in memory
 - Radix sort: also called “linear time” sort. For keys of fixed length and fixed radix a constant number of passes over the data is sufficient independent of the number of keys
- “The Influence of Caches on the Performance of Sorting” by A. LaMarca and R.E. Ladner. Proceedings of the Eighth Annual ACM-SIAM Symposium on Discrete Algorithms, January, 1997, 370-379.
 - For Alphastation 250, 32 byte blocks, direct mapped L2 2MB cache, 8 byte keys, from 4000 to 4000000

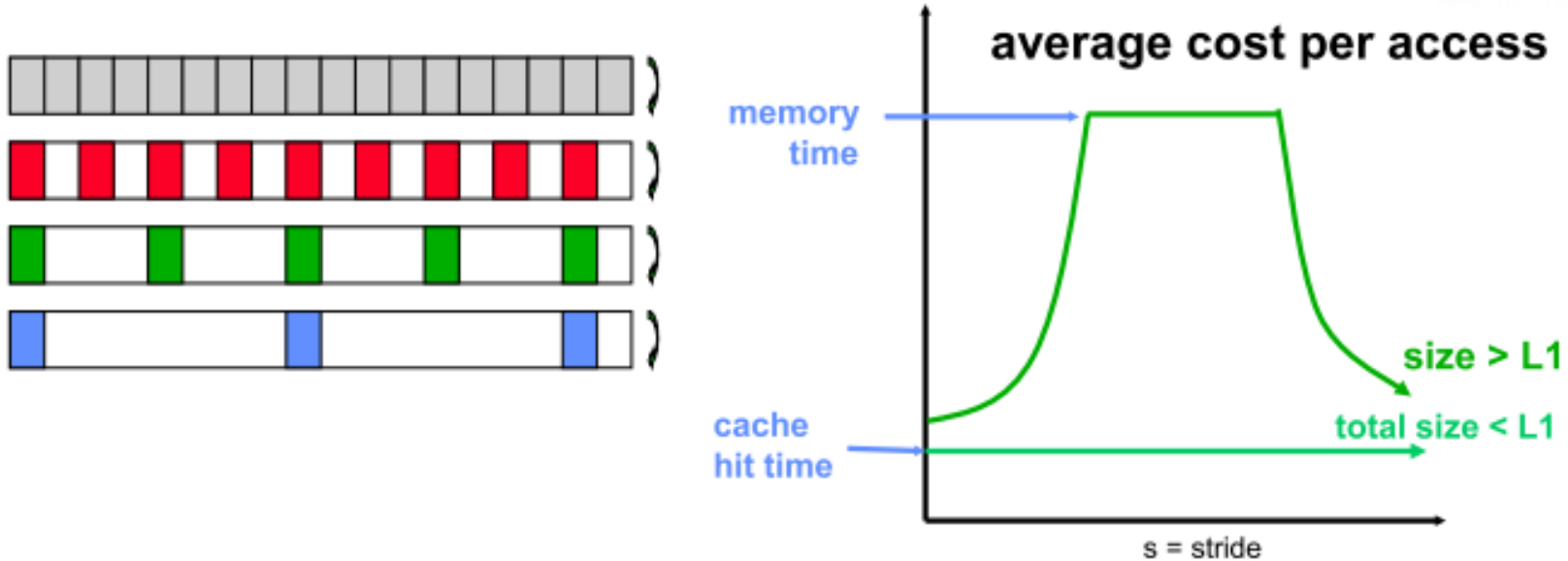
Experimental Study (Membench)

- Microbenchmark for memory system performance



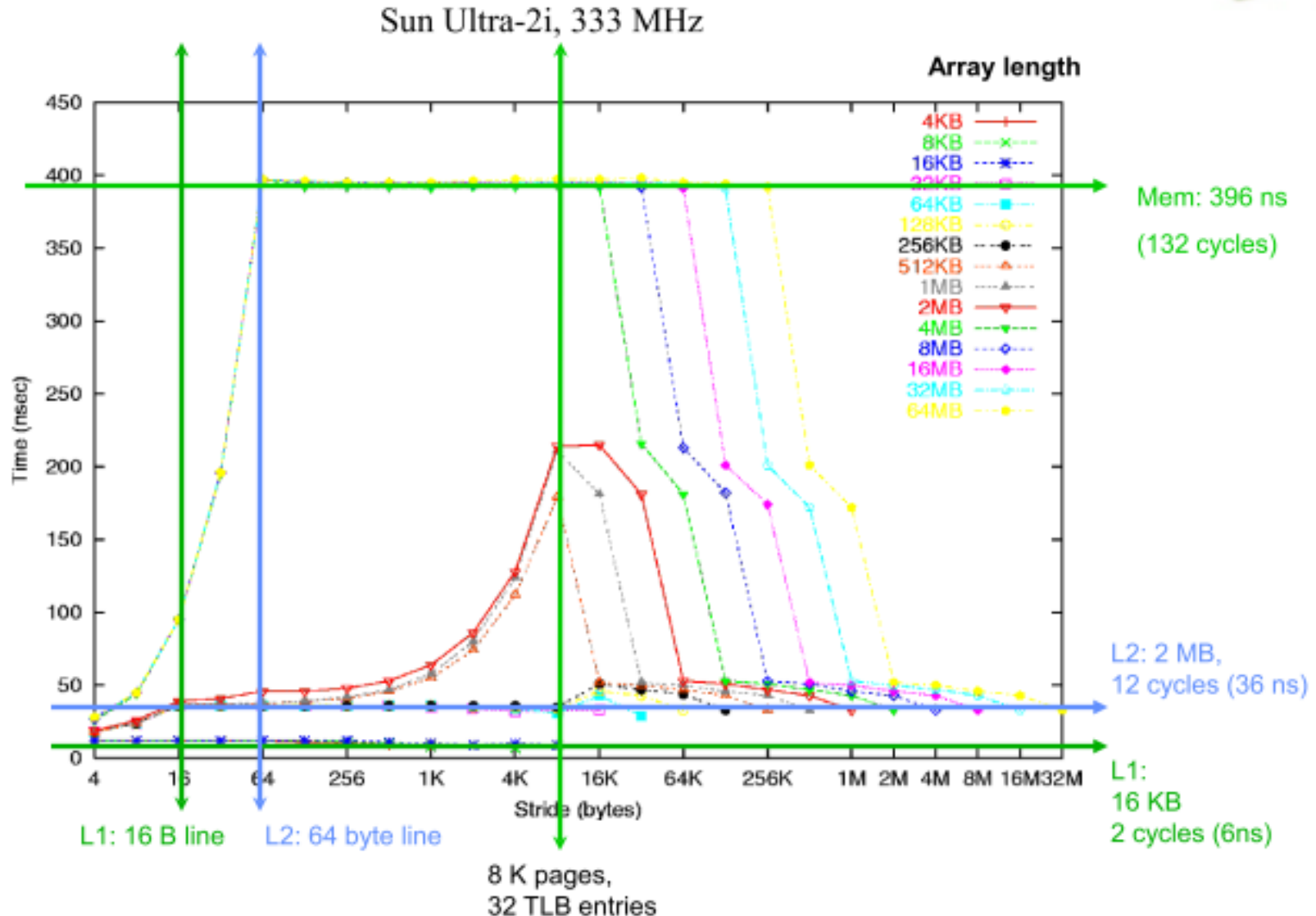
- for array A of length L from 4KB to 8MB by 2x
 for stride s from 4 Bytes (1 word) to L/2 by 2x 1 experiment
 time the following loop
 (repeat many times and average)
 for i from 0 to L by s
 load A[i] from memory (4 Bytes)

Membench: What to Expect



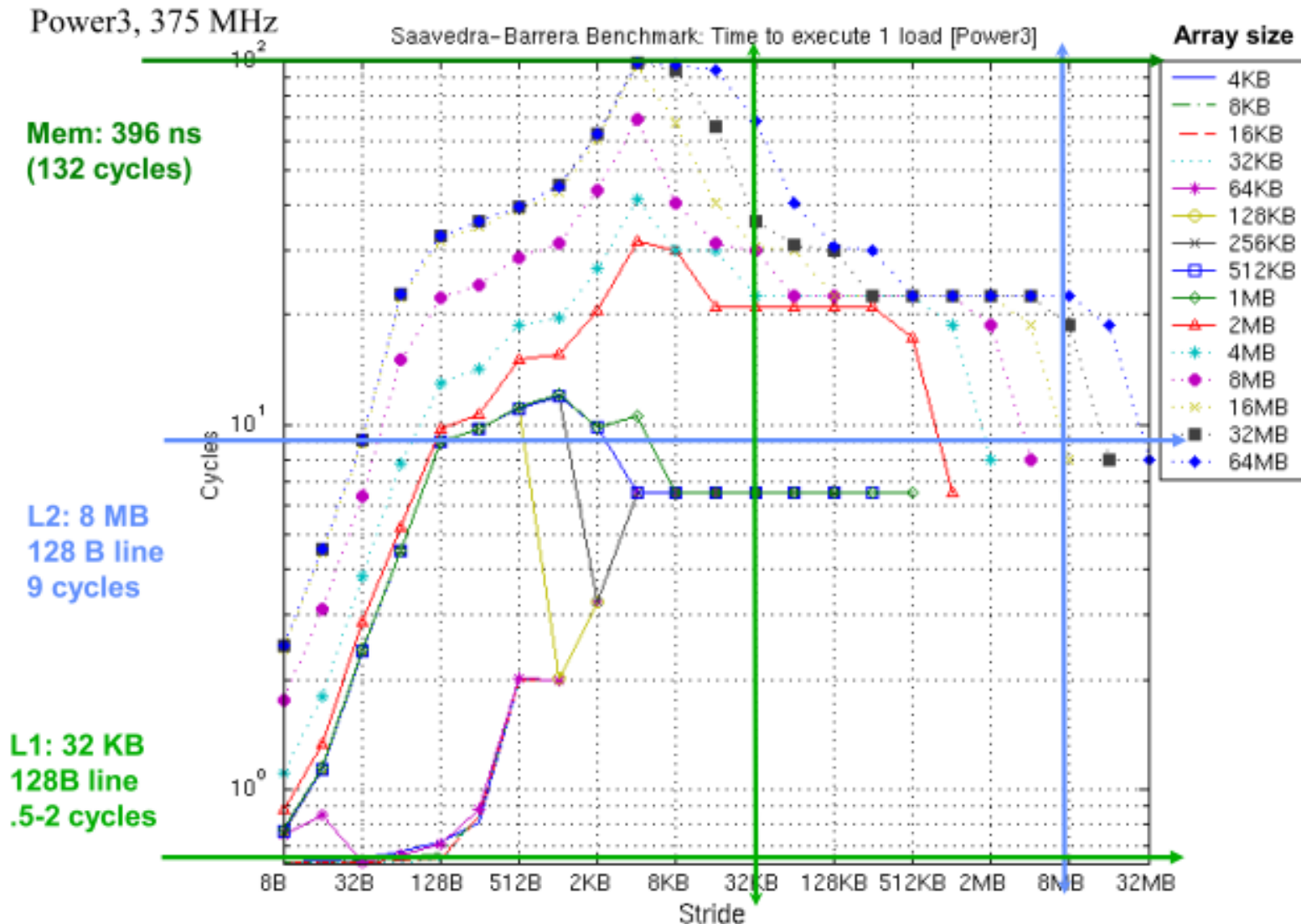
- Consider the average cost per load
 - Plot one line for each array length, time vs. stride
 - Small stride is best: if cache line holds 4 words, at most $\frac{1}{4}$ miss
 - If array is smaller than a given cache, all those accesses will hit (after the first run, which is negligible for large enough runs)
 - Picture assumes only one level of cache
 - Values have gotten more difficult to measure on modern procs

Memory Hierarchy on a Sun Ultra-2i



See www.cs.berkeley.edu/~yelick/arvindk/t3d-isca95.ps for details

Memory Hierarchy on a Power3



Memory Hierarchy Lessons

- Caches vastly impact performance
 - Cannot consider performance without considering memory hierarchy
- Actual performance of a simple program can be a complicated function of the architecture
 - Slight changes in the architecture or program change the performance significantly
 - To write fast programs, need to consider architecture
 - True on sequential or parallel processor
 - We would like simple models to help us design efficient algorithms

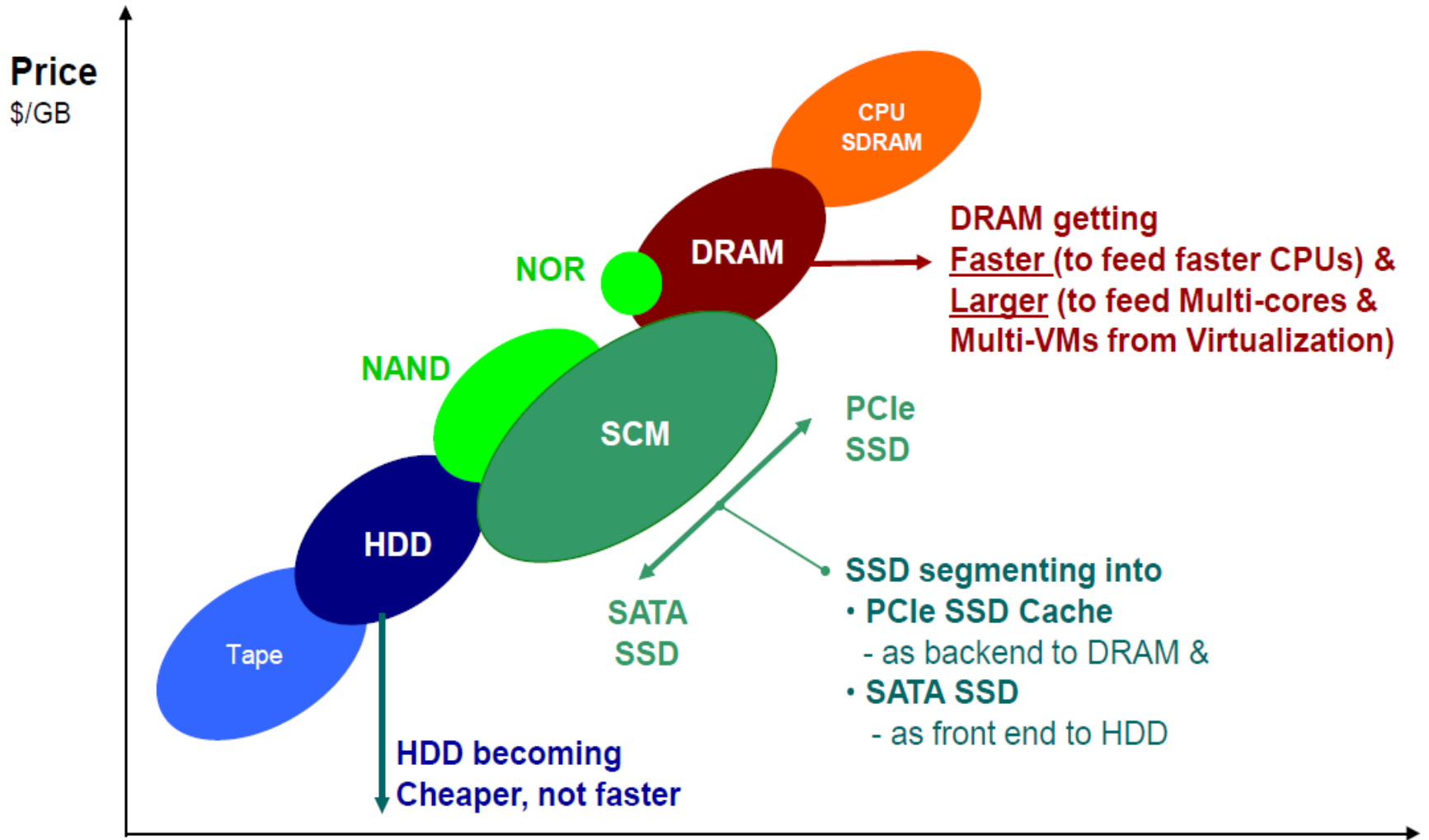
Memory Hierarchy Lessons

- Common technique for improving cache performance, called **blocking** or **tiling**
 - Idea: used divide-and-conquer to define a problem that fits in register/L1-cache/L2-cache

- Autotuning: Deal with complexity through experiments
 - Produce several different versions of code
 - Different algorithms, Blocking Factors, Loop orderings, etc
 - For each architecture, run different versions to see which is fastest
 - Can (in principle) navigate complex design options for optimum

New Advances on Memory and Storage

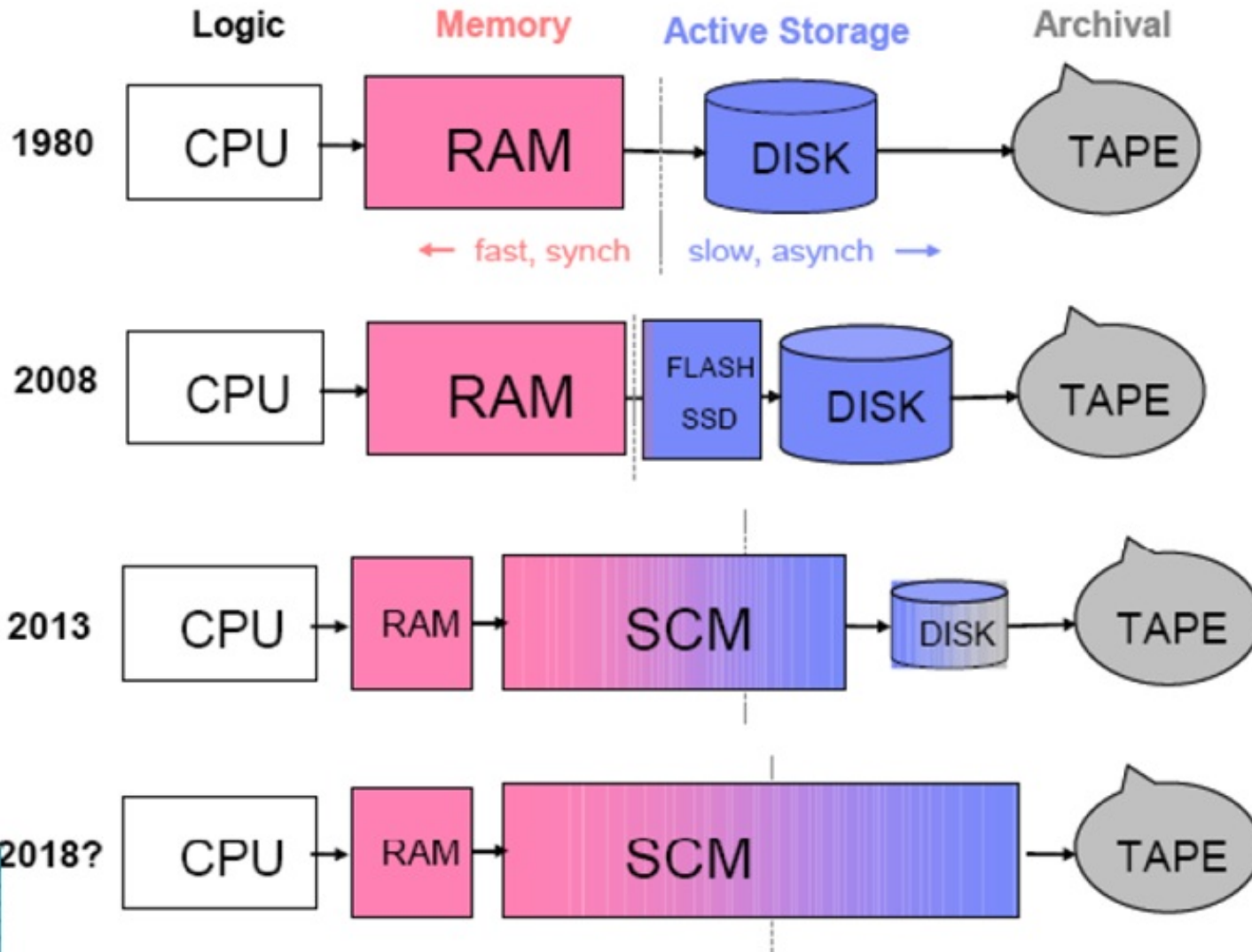
— Storage Class Memory



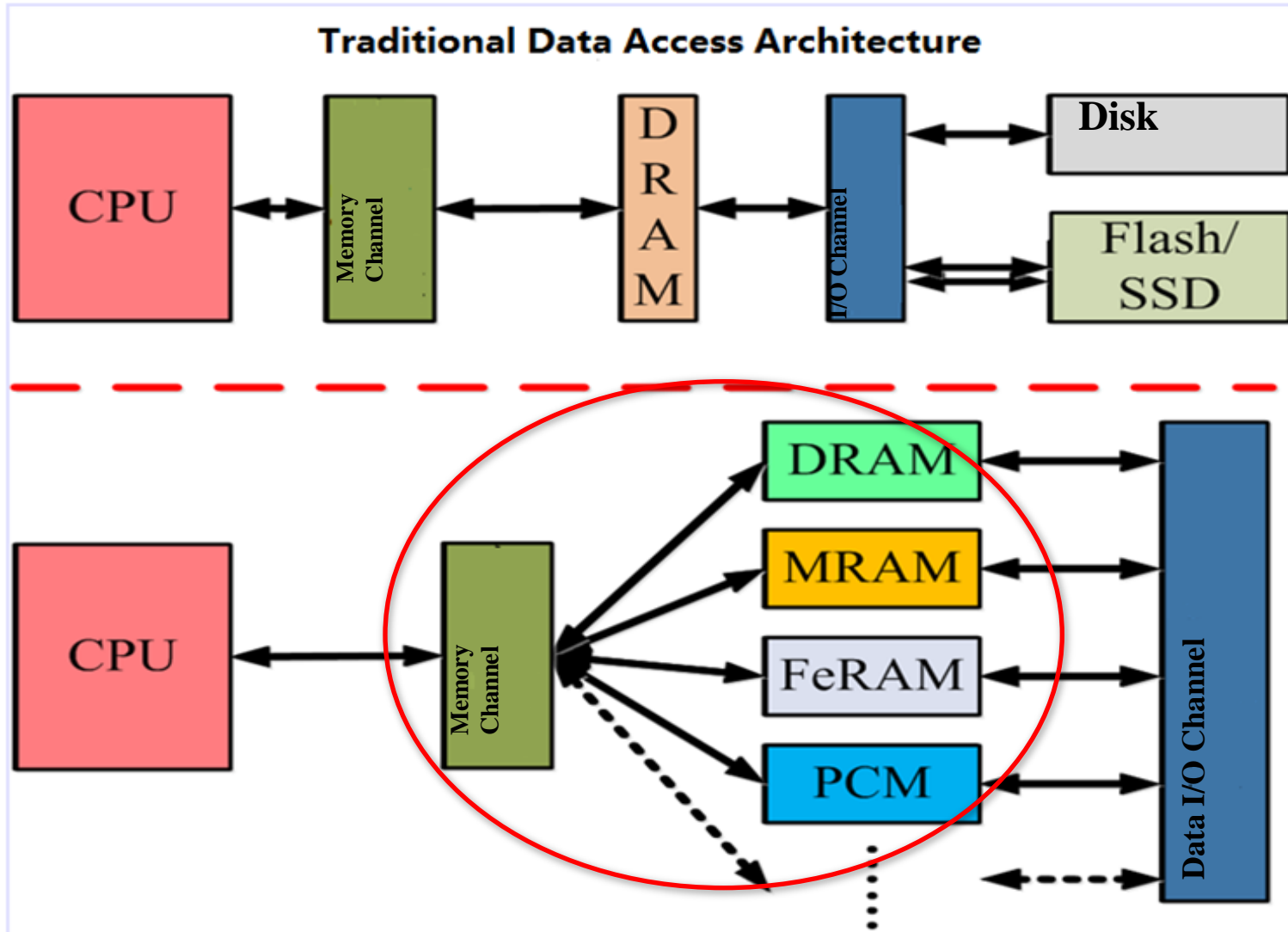
Source: IMEX Research SSD Industry Report ©2011

Performance
I/O Access Latency

Reconstruction of Virtual Memory Architecture: Break the I/O Bottleneck



New In-Memory Computing Architecture



Oracle EXADATA

Database Grid

- 8 Dual-processor x64 database servers

OR

- 2 Eight-processor x64 database servers

InfiniBand Network

- Redundant 40Gb/s switches
- Unified server & storage network



Intelligent Storage Grid

- 14 High-performance low-cost storage servers



- 100 TB **High Performance** disk, or
336 TB **High Capacity** disk

5.3 TB PCI Flash

- Data mirrored across storage servers

Parallel architecture

WHAT IS PARALLEL ARCHITECTURE

What is Parallel Architecture?

- Machines with multiple processors



Intel's Quad Core i7



Apple MacPro



Blacklight at the PSC (4096 cores)



Hadoop cluster at Yahoo!

One Definition of Parallel Architecture

A parallel computer is a collection of processing elements that cooperate to solve large problems fast

Some broad issues

□ Resource Allocation

- how large a collection?
- how powerful are the elements?
- how much memory?

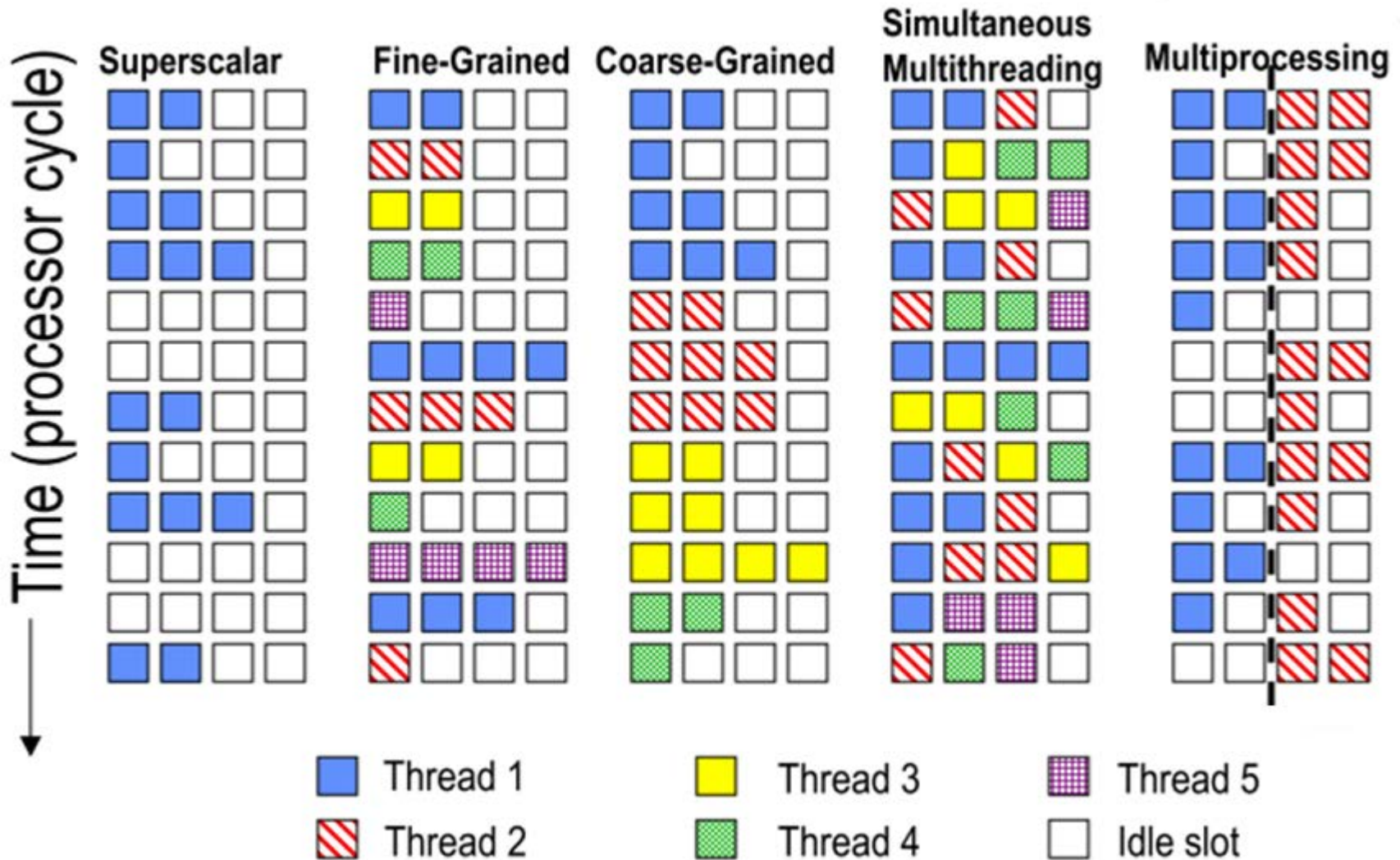
□ Data access, Communication and Synchronization

- how do the elements cooperate and communicate?
- how are data transmitted between processors?
- what are the abstractions and primitives for cooperation?

□ Performance and Scalability

- how does it all translate into performance?
- how does it scale?

Types of Parallelism



Parallel architecture

A PARALLEL ZOO OF ARCHITECTURES

MIMD Machines

□ Multiple Instruction, Multiple Data (MIMD)

- Multiple independent instruction streams, program counters, etc
- Called “multiprocessing” instead of “multithreading”
 - Although, each of the multiple processors may be multithreaded
- When independent instruction streams confined to single chip, becomes a “multicore” processor

□ Shared memory: Communication through Memory

- Option 1: no hardware global cache coherence
- Option 2: hardware global cache coherence

□ Message passing: Communication through Messages

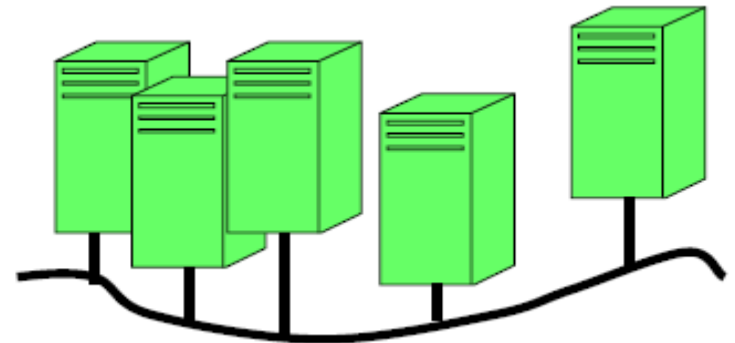
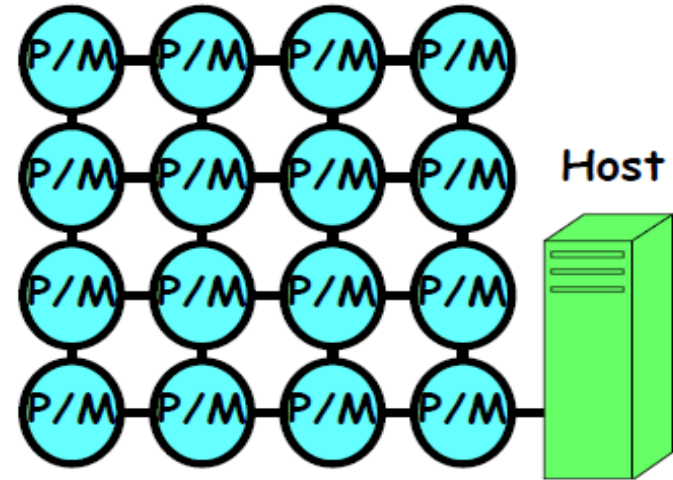
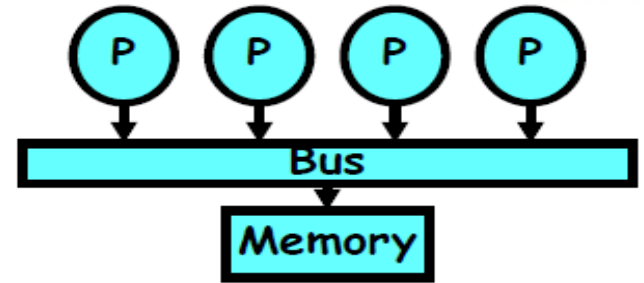
- Applications send explicit messages between nodes in order to communicate

□ For most machines, shared memory built on top of message-passing network

- Bus-based machines are “exception”

Examples of MIMD Machines

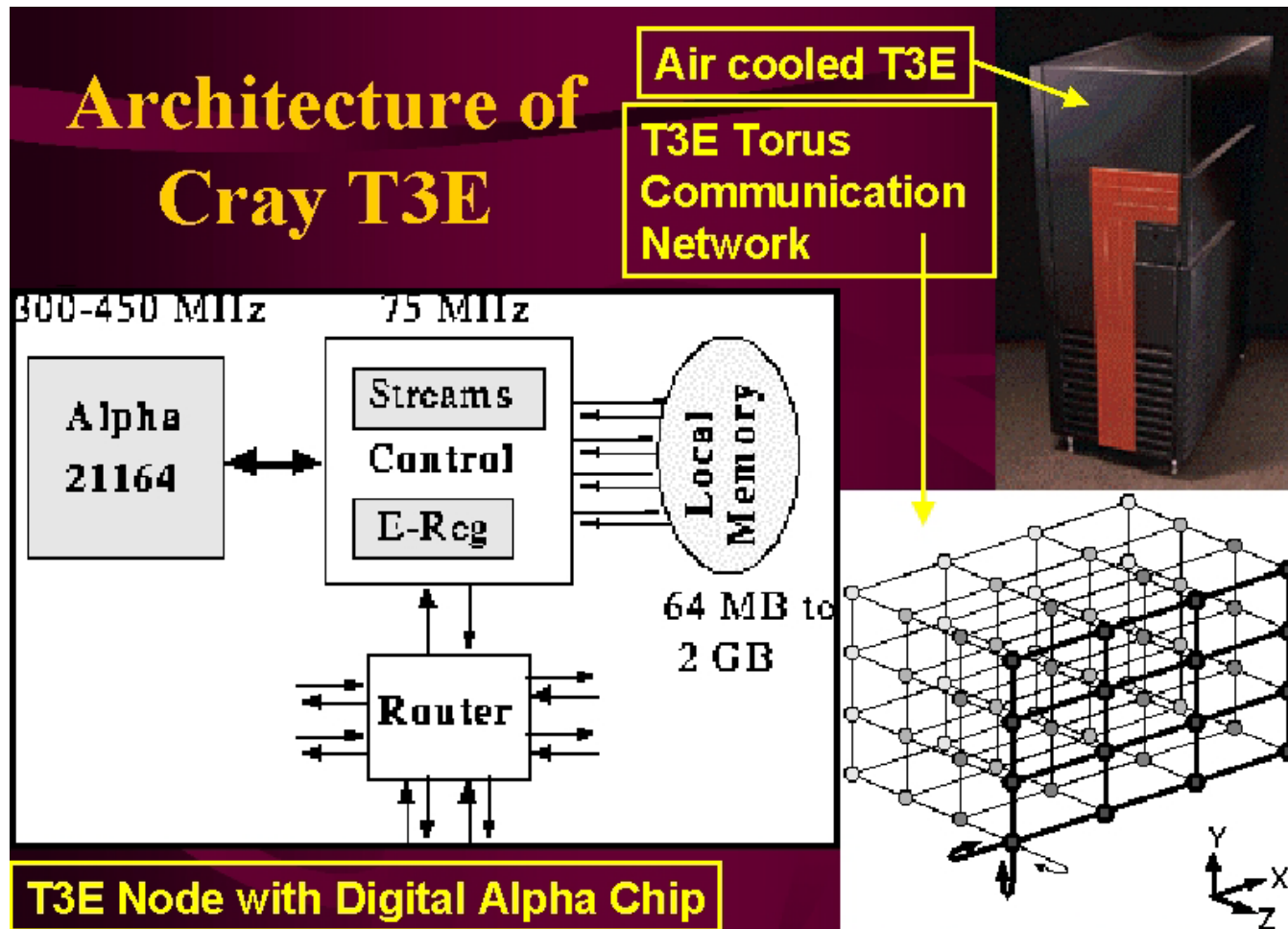
- ❑ Symmetric Multiprocessor
 - Multiple processors in box with shared memory communication
 - Current MultiCore chips like this
 - Every processor runs copy of OS
- ❑ Non-uniform shared-memory with separate I/O through host
 - Multiple processors
 - Each with local memory
 - general scalable network
 - Extremely light “OS” on node provides simple services
 - Scheduling/synchronization
 - Network-accessible host for I/O
- ❑ Cluster
 - Many independent machine connected with general network
 - Communication through messages



Cray T3E (1996)

Follow-on to earlier T3D (1993) using 21064's

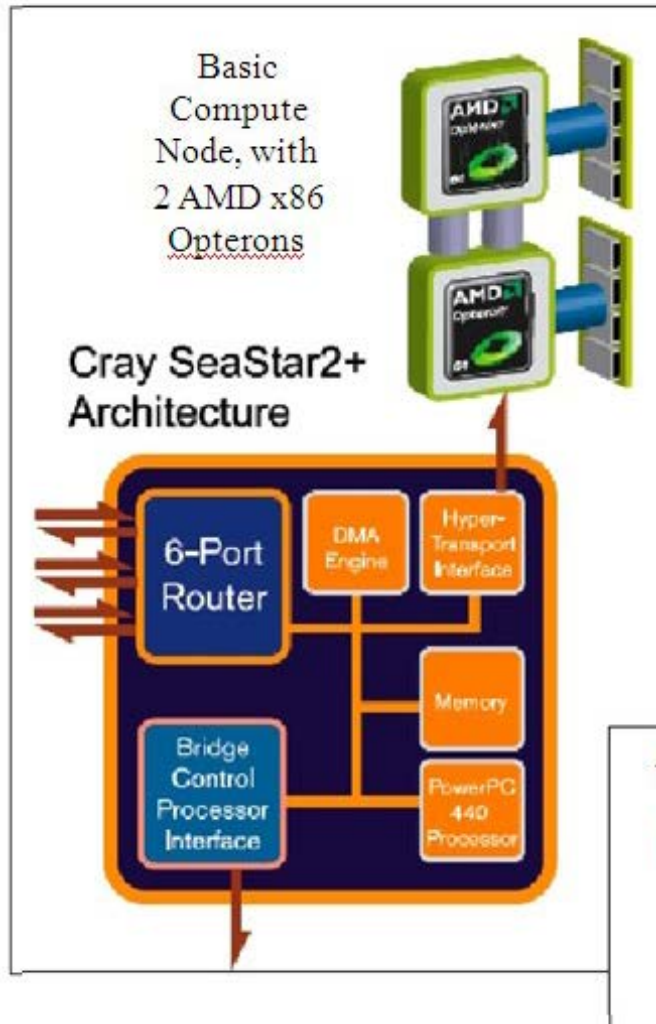
Up to 2,048 675MHz Alpha 21164 processors connected in 3D torus



Cray T3E (1996)

- ❑ Each node has 256MB-2GB local DRAM memory
- ❑ Load and stores access global memory over network
- ❑ Only local memory cached by on-chip caches
- ❑ Alpha microprocessor surrounded by custom “shell” circuitry to make it into effective MPP node
- ❑ Shell provides
 - multiple stream buffers instead of board-level (L3) cache
 - external copy of on-chip cache tags to check against remote writes to local memory, generates on-chip invalidates on match
 - 512 external E registers (asynchronous vector load/store engine)
 - address management to allow all of external physical memory to be addressed
 - atomic memory operations (fetch&op)
 - support for hardware barriers/eureka to synchronize parallel tasks

Cray XT5 (2007)



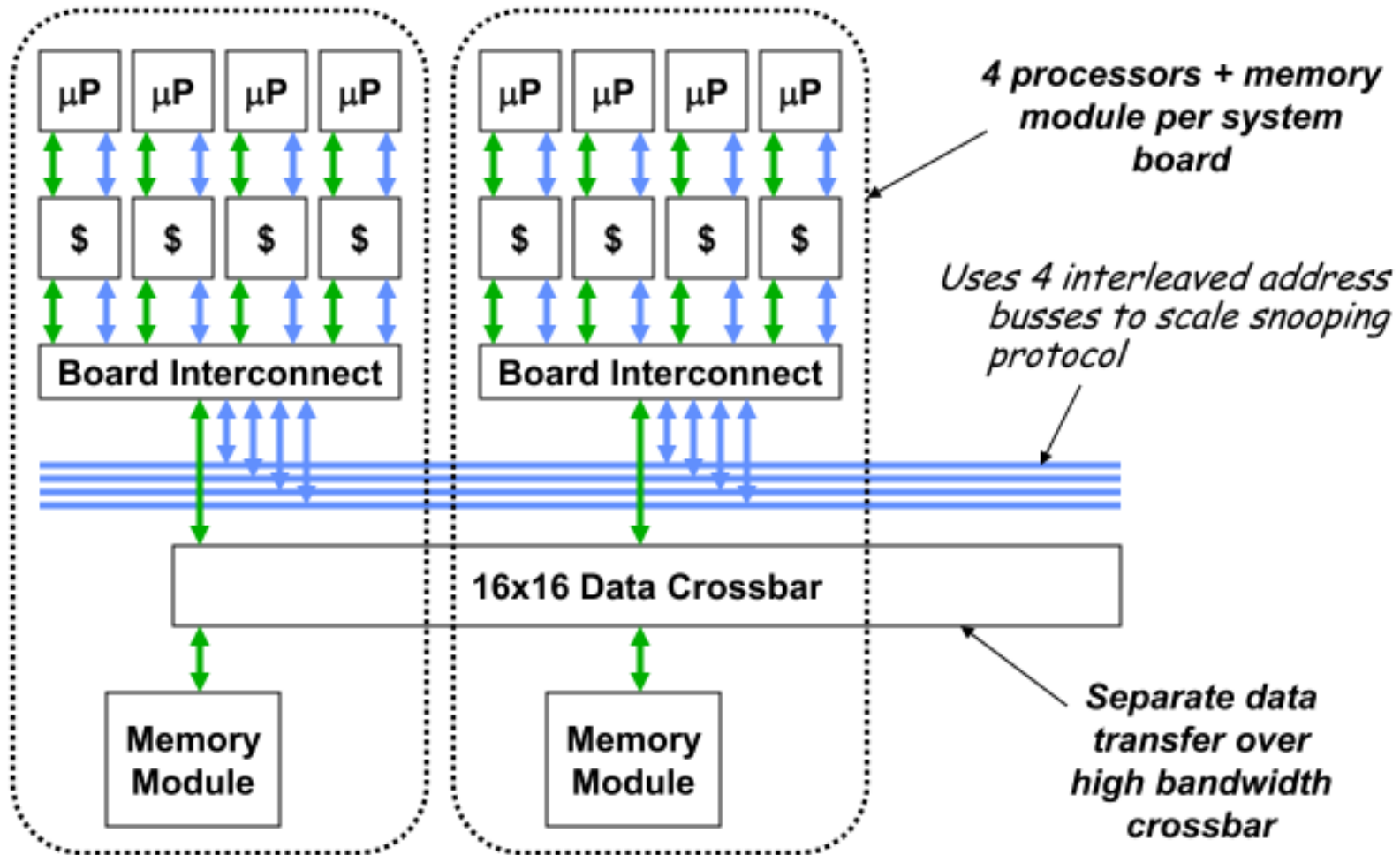
Also, XMT Multithreaded Nodes based on MTA design (128 threads per processor)

Processor plugs into



Sun Starfire UE10000 (1997)

Up to 64-way SMP using bus-based snooping protocol

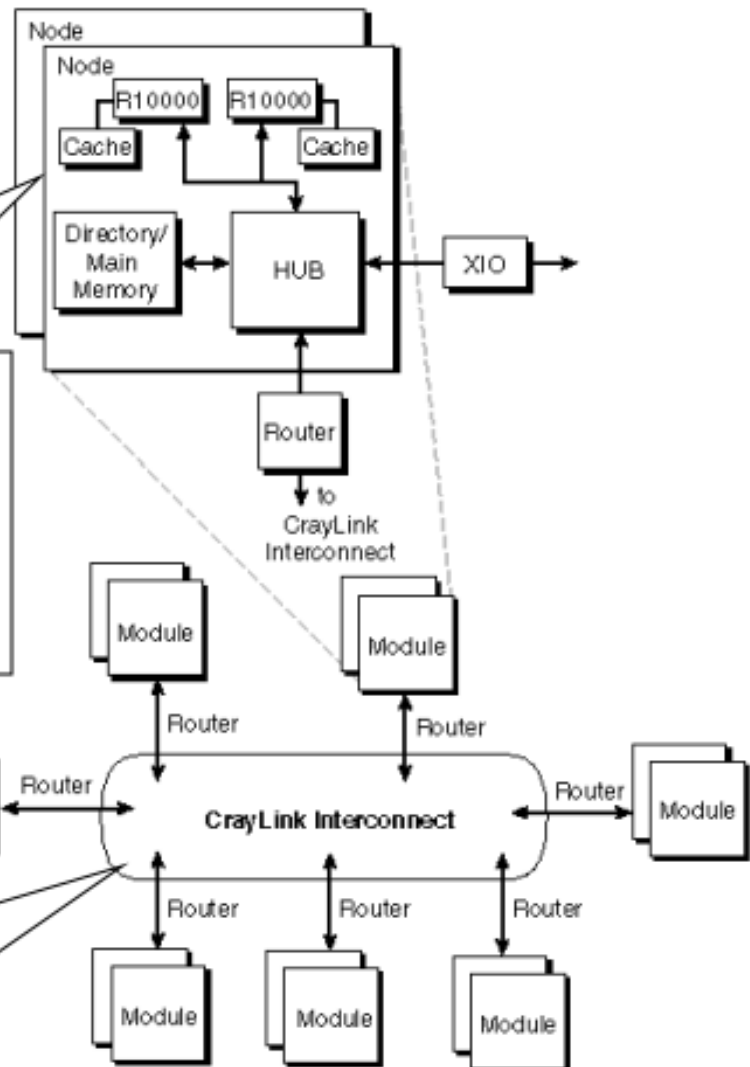


SGI Origin 2000 (1996)

- Large-Scale Distributed Directory SMP
 - Scales from 2 to 512 nodes
 - Direct-mapped directory with each bit standing for multiple processors
 - Not highly scalable beyond this

Node contains:

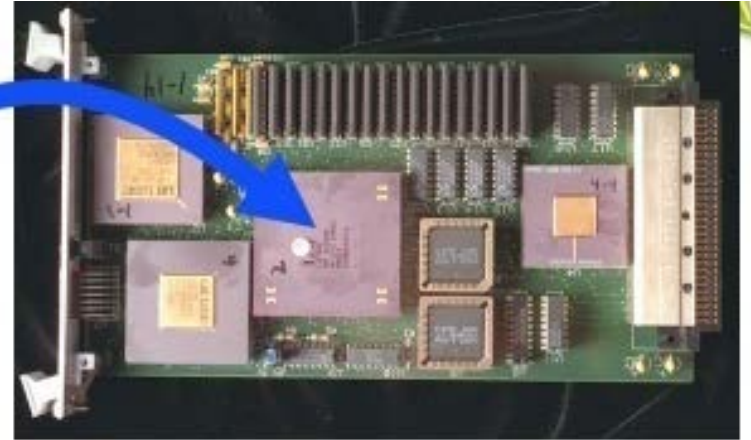
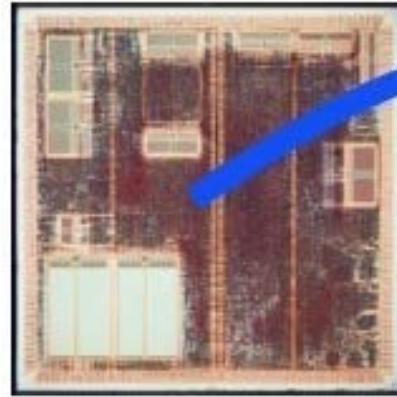
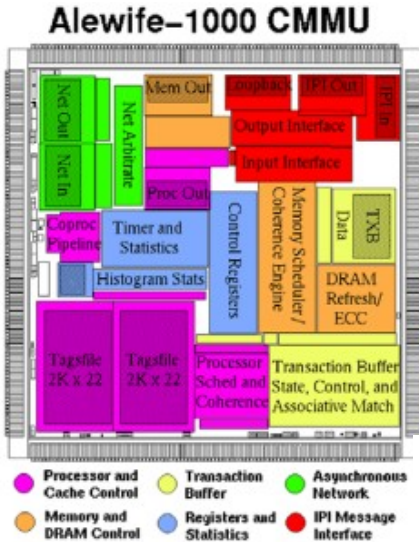
- Two MIPS R10000 processors plus caches
- Memory module including directory
- Connection to global network
- Connection to I/O



Scalable hypercube switching network supports up to 64 two-processor nodes (128 processors total)

(Some installations up to 512 processors)

MIT Alewife Multiprocessor: SM & MP



- Cache-coherence Shared Memory
 - Partially in Software!
 - Sequential Consistency
 - LimitLESS cache coherence for better scalability
- User-level Message-Passing
 - Fast, atomic launch of messages
 - Active messages
 - User-level interrupts
- Rapid Context-Switching
 - Course-grained multithreading
- Single Full/Empty bit per word for synchronization
 - Can build locks, barriers, other higher-level constructs

Message Passing MPPs

(Massively Parallel Processors)

□ Initial Research Projects

- Caltech Cosmic Cube (early 1980s) using custom Mosaic processors
- J-Machine (early 1990s) MIT

□ Commercial Microprocessors including MPP Support

- Transputer (1985)
- nCube-1(1986) /nCube-2 (1990)

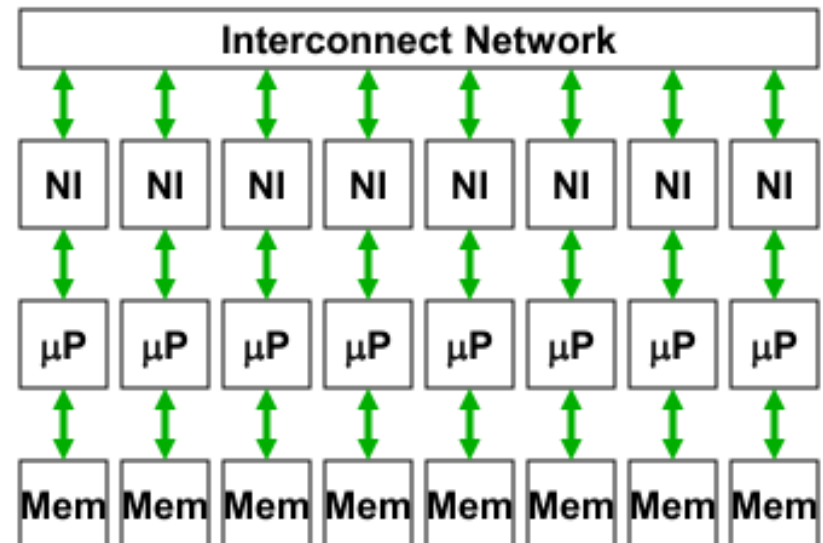
□ Standard Microprocessors + Network Interfaces

- Intel Paragon/i860 (1991)
- TMC CM-5/SPARC (1992)
- Meiko CS-2/SPARC (1993)
- IBM SP-1/POWER (1993)

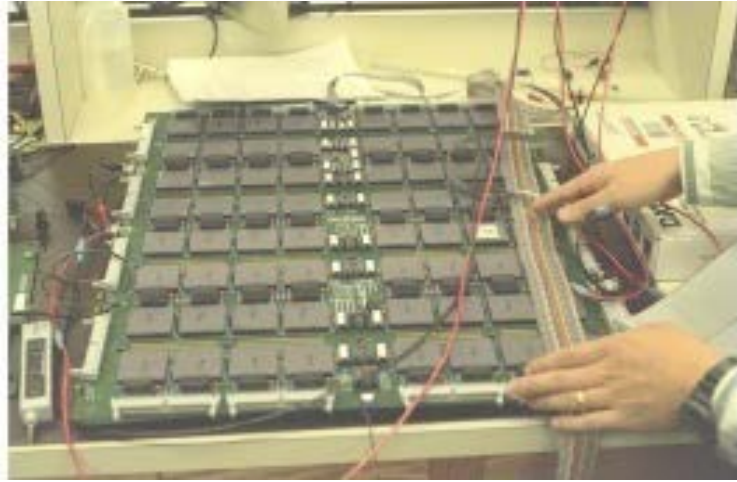
□ MPP Vector Supers

- Fujitsu VPP500 (1994)

*Designs scale to 100s-10,000s
of nodes*



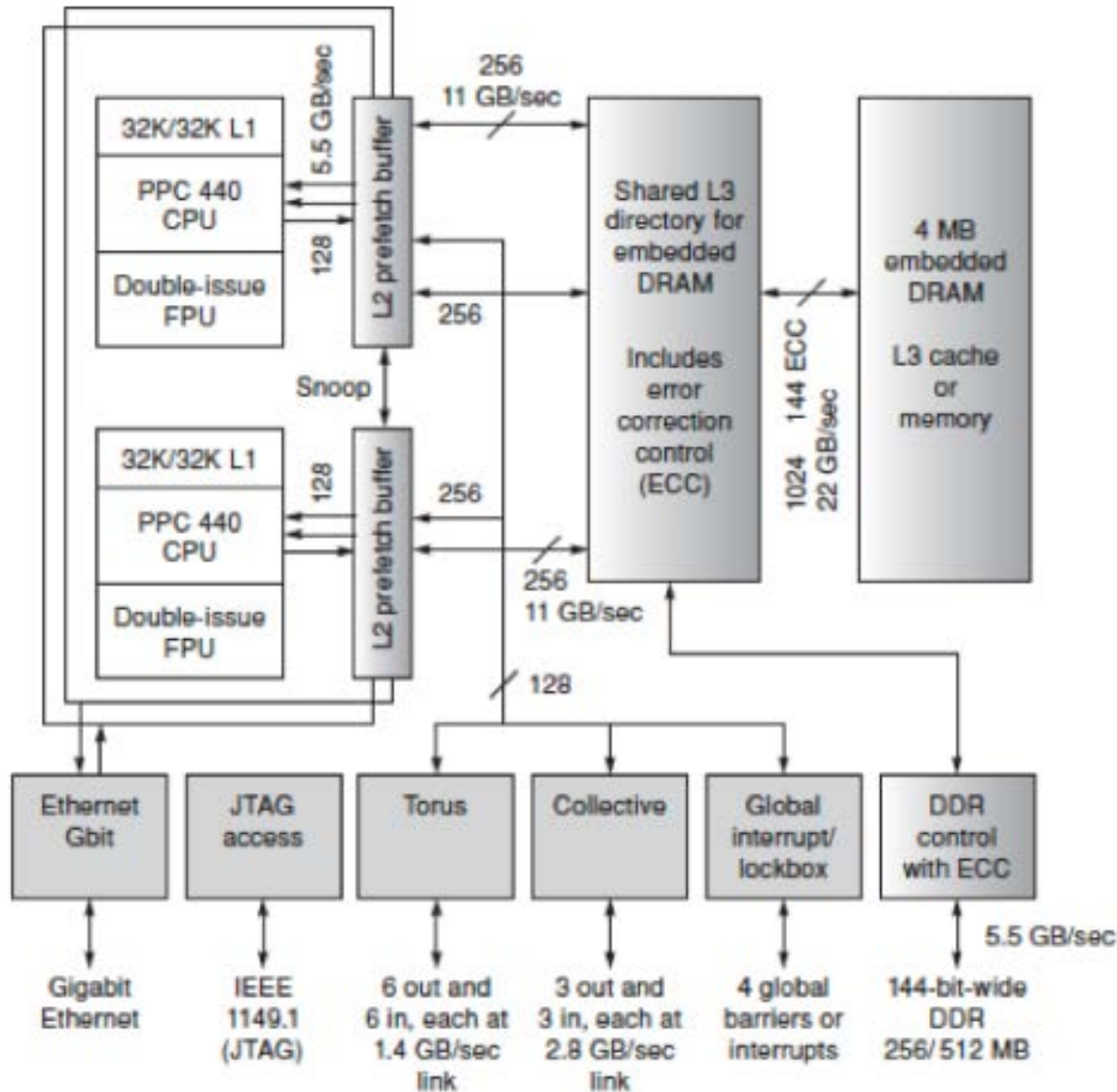
MIT J-Machine (Jelly-bean machine)



- ❑ 3-dimensional network topology
 - Non-adaptive, E-cubed routing
 - Hardware routing
 - Maximize density of communication
- ❑ 64-nodes/board, 1024 nodes total
- ❑ Low-powered processors
 - Message passing instructions
 - Associative array primitives to aid in synthesizing shared-address space
- ❑ Extremely fine-grained communication
 - Hardware-supported Active Messages













IBM Blue Gene/L Processor



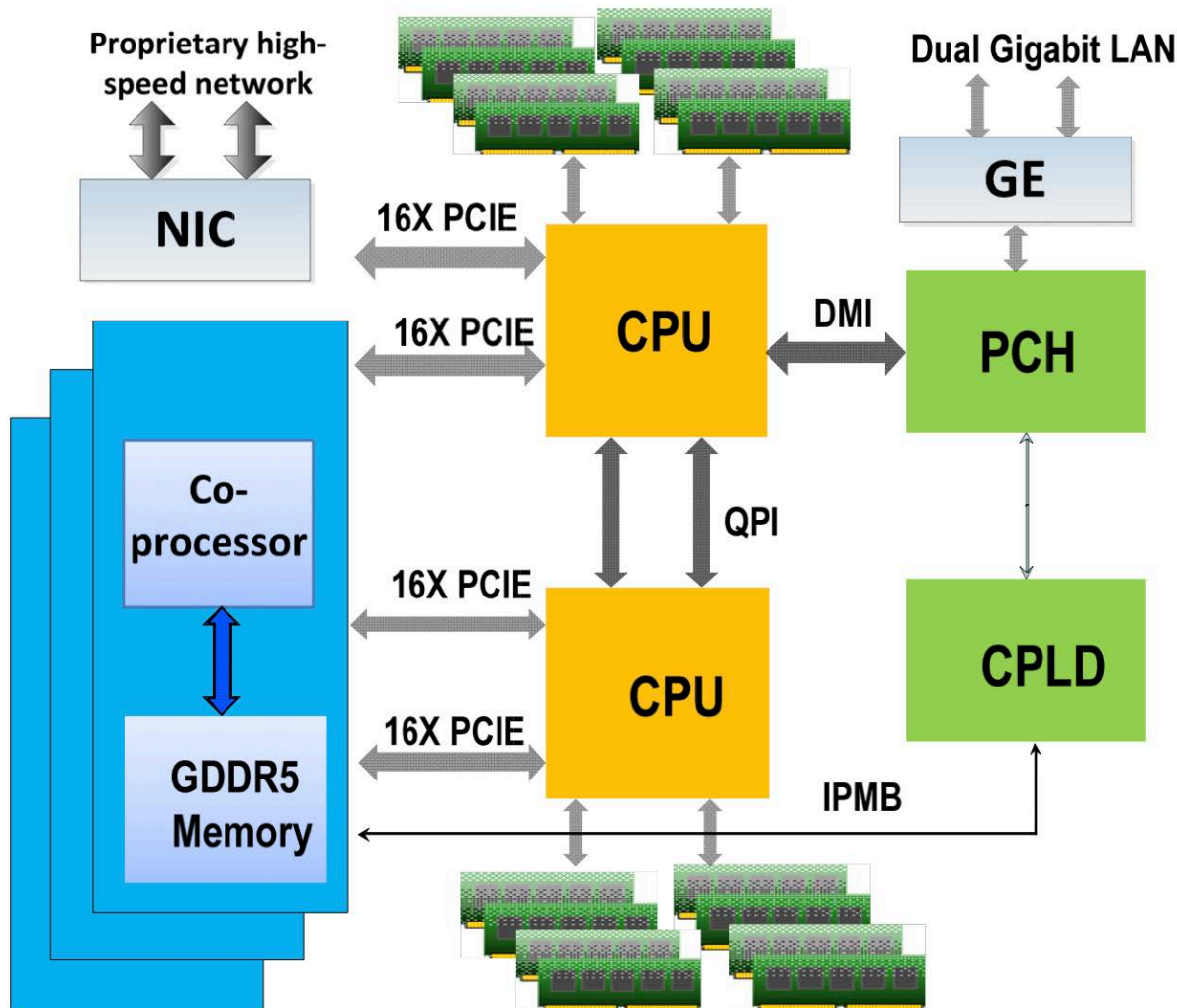
BG/L 64K Processor System

- Peak Performance 360TFLOPS
- Power Consumption 1.4 MW

42nd TOP500 on November 18, 2013

Rank ↕	Rmax Rpeak (Pflops) ↕	Name ↕	Computer design Processor type, interconnect ↕	Vendor ↕	Site Country, year ↕	Operating system ↕
1	33.863 54.902	<i>Tianhe-2</i>	NUDT Xeon E5-2692 + Xeon Phi 31S1P, TH Express-2	NUDT	National Supercomputing Center in Guangzhou  China, 2013	Linux (Kylin)
2	17.590 27.113	<i>Titan</i>	Cray XK7 Opteron 6274 + Tesla K20X, Cray Gemini Interconnect	Cray	Oak Ridge National Laboratory  United States, 2012	Linux (CLE, SLES based)
3	17.173 20.133	<i>Sequoia</i>	Blue Gene/Q PowerPC A2, Custom	IBM	Lawrence Livermore National Laboratory  United States, 2013	Linux (RHEL and CNK)
4	10.510 11.280	<i>K computer</i>	RIKEN SPARC64 VIIIfx, Tofu	Fujitsu	RIKEN  Japan, 2011	Linux
5	8.586 10.066	<i>Mira</i>	Blue Gene/Q PowerPC A2, Custom	IBM	Argonne National Laboratory  United States, 2013	Linux (RHEL and CNK)
6	6.271 7.779	<i>Piz Daint</i>	Cray XC30 Xeon E5-2670 + Tesla K20X, Aries	Cray Inc.	Swiss National Supercomputing Centre  Switzerland, 2013	Linux (CLE)
7	5.168 8.520	<i>Stampede</i>	PowerEdge C8220 Xeon E5-2680 + Xeon Phi, Infiniband	Dell	Texas Advanced Computing Center  United States, 2013	Linux
8	5.008 5.872	<i>JUQUEEN</i>	Blue Gene/Q PowerPC A2, Custom	IBM	Forschungszentrum Jülich  Germany, 2013	Linux (RHEL and CNK)
9	4.293 5.033	<i>Vulcan</i>	Blue Gene/Q PowerPC A2, Custom	IBM	Lawrence Livermore National Laboratory  United States, 2013	Linux (RHEL and CNK)
10	2.897 3.185	<i>SuperMUC</i>	iDataPlex DX360M4 Xeon E5-2680, Infiniband	IBM	Leibniz-Rechenzentrum  Germany, 2012	Linux

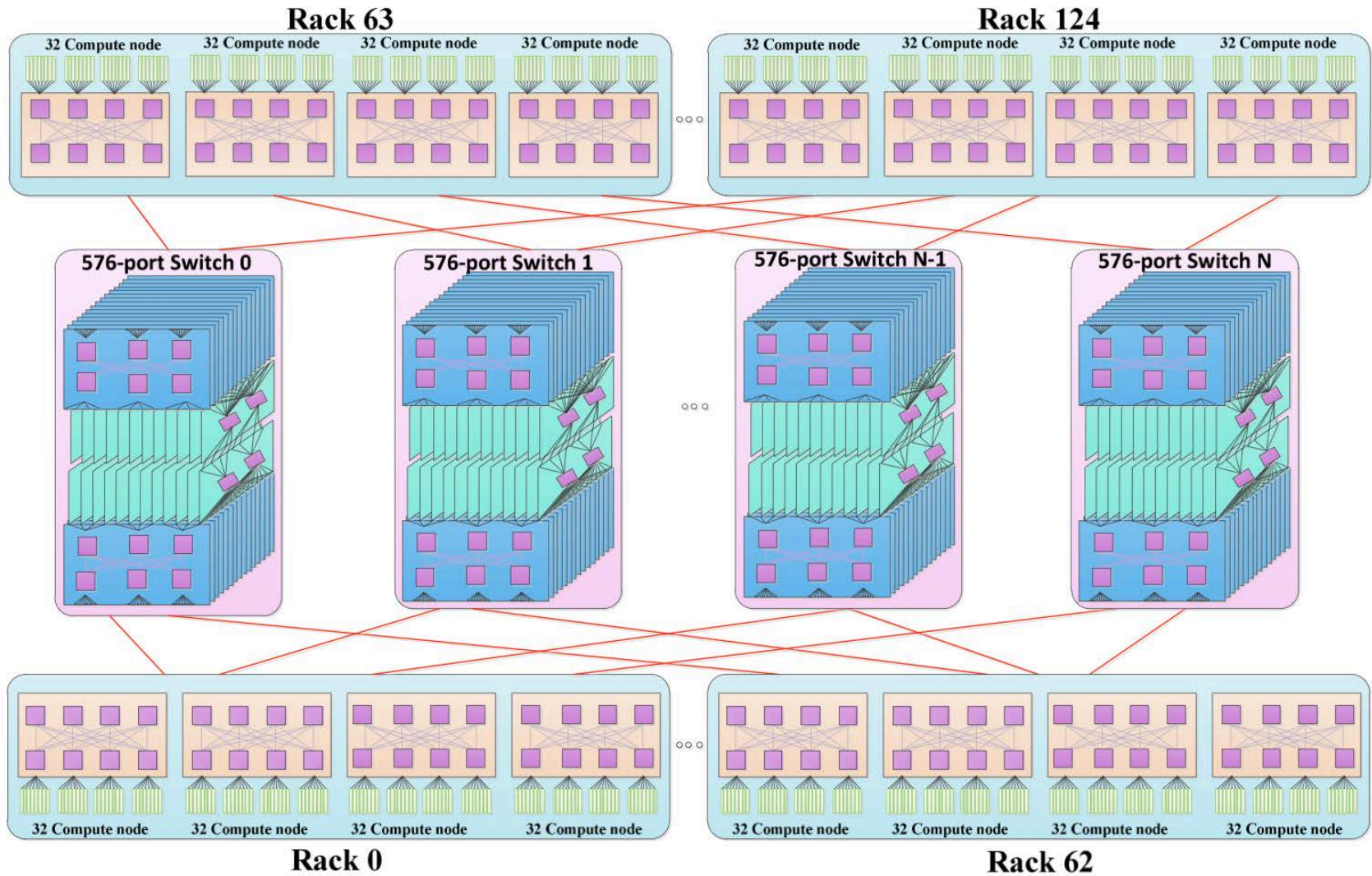
Tianhe-2



Tianhe-2

Sponsors	863 Program
Location	Guangzhou, China
Architecture	Intel Xeon E5, Xeon Phi
Power	17.6 MW (24 MW with cooling)
Operating system	Kylin Linux ^[1]
Memory	1,375 TiB (1,000 TiB CPU and 375 TiB Coprocessor) ^[1]
Storage	12.4 PB
Speed	33.86 PFLOPS
Cost	2.4 billion Yuan (390 million USD) ^[2]
Purpose	Research and education

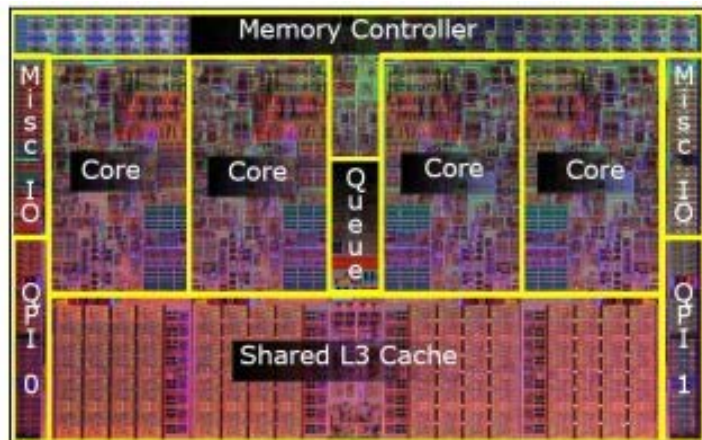
Tianhe-2



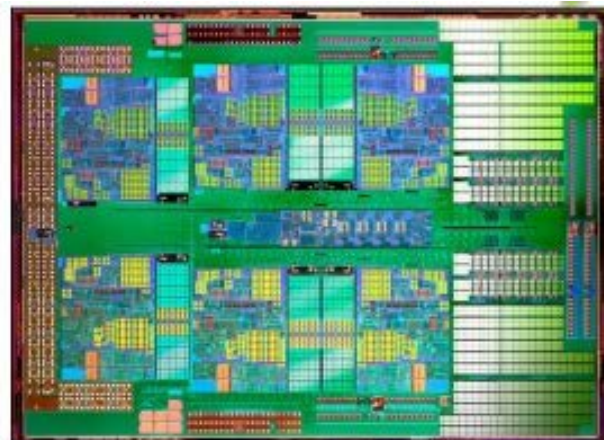
Parallel architecture

MULTICORE CHIPS

Parallel Chip-Scale Processors



Intel Core 2 Quad: 4 Cores



AMD Opteron: 6 Cores

- ❑ Multicore processors emerging in general-purpose market due to power limitations in single-core performance scaling
 - 4-16 cores in 2009, connected as cache-coherent SMP
 - Cache-coherent shared memory
- ❑ Embedded applications need large amounts of computation
 - Recent trend to build “extreme” parallel processors with dozens to hundreds of parallel processing elements on one die
 - Often connected via on-chip networks, with no cache coherence
 - Examples: 188 core “Metro” chip from CISCO

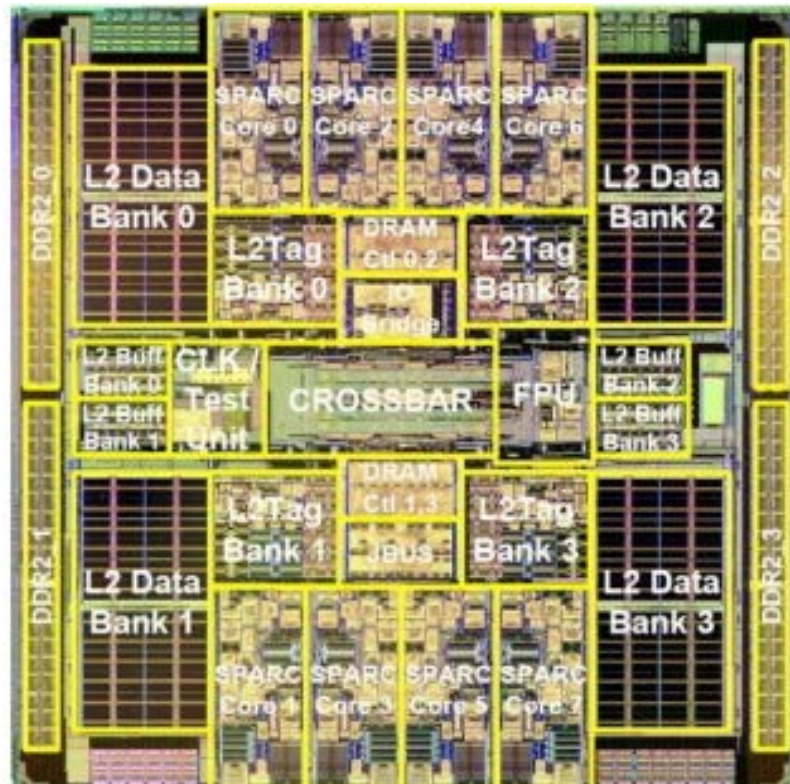
Sun's T1 ("Niagara")

□ Highly Threaded

- 8 Cores
- 4 Threads/Core

□ Target: Commercial server applications

- High thread level parallelism (TLP)
 - Large numbers of parallel client requests
- Low instruction level parallelism (ILP)
 - High cache miss rates
 - Many unpredictable branches
 - Frequent load-load dependencies



□ Power, cooling, and space are major concerns for data centers

□ Metric: Performance/Watt/Sq. Ft.

□ Approach: Multicore, Fine-grain multithreading, Simple pipeline, Small L1 caches, Shared L2

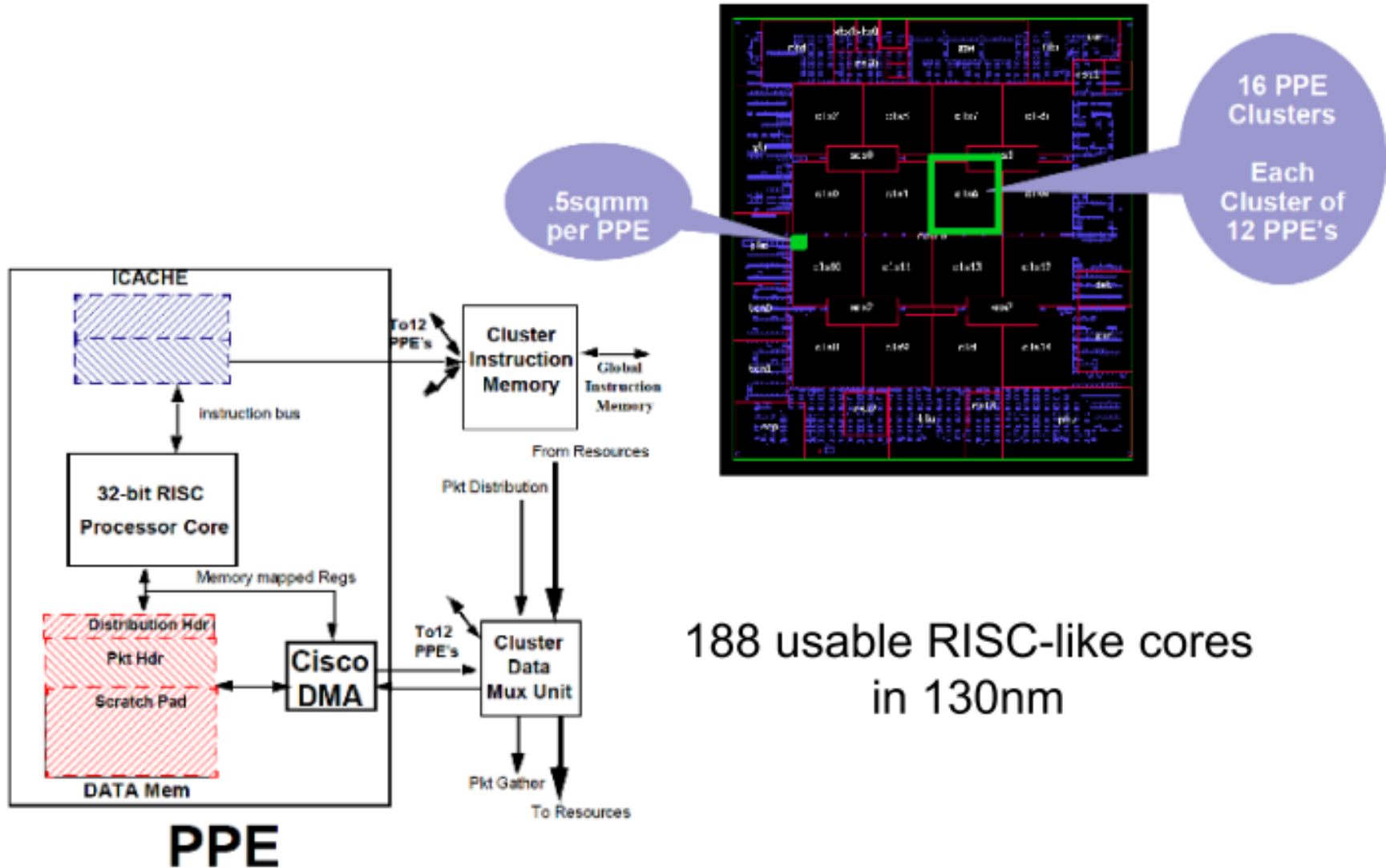
T1 Fine-Grained Multithreading

- Each core supports four threads and has its own level one caches (16KB for instructions and 8 KB for data)
 - Coherency is enforced among the L1 caches by a directory associated with each L2 cache block
- Switching to a new thread on each clock cycle
- Idle threads are bypassed in the scheduling
 - Waiting due to a pipeline delay or cache miss
 - Processor is idle only when all 4 threads are idle or stalled
- Both loads and branches incur a 3 cycle delay that can only be hidden by other threads
- A single set of floating-point functional units is shared by all 8 cores
 - floating-point performance was not a focus for T1
 - (New T2 design has FPU per core)

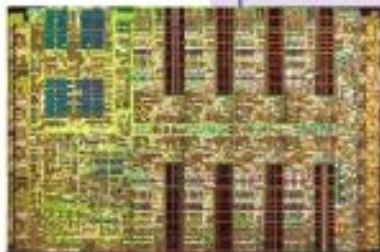
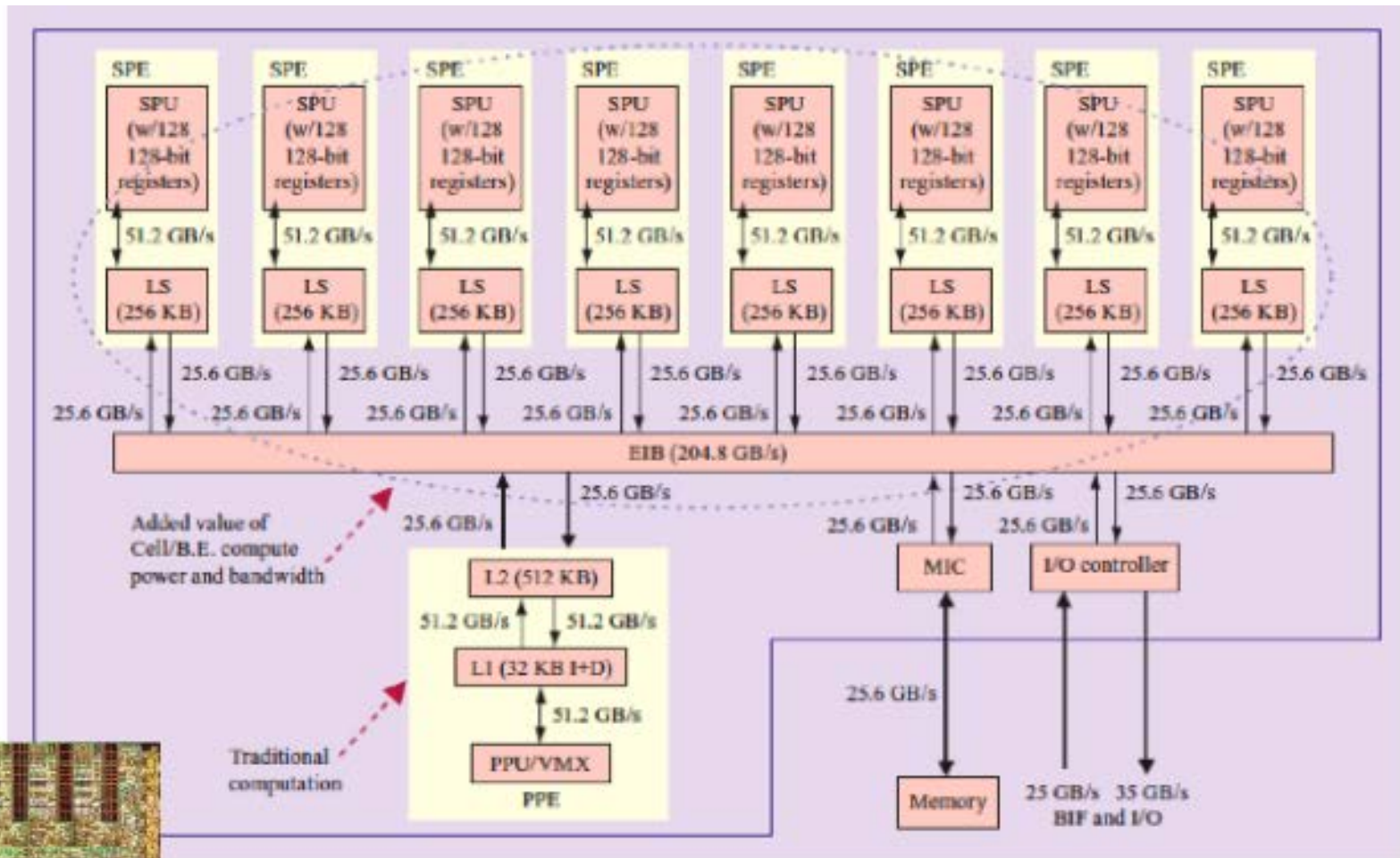
Embedded Parallel Processors

- ❑ Often embody a mixture of old architectural styles and ideas
- ❑ Exposed memory hierarchies and interconnection networks
 - Programmers code to the “metal” to get best cost/power/performance
 - Portability across platforms less important
- ❑ Customized synchronization mechanisms
 - Interlocked communication channels (processor blocks on read if data not ready)
 - Barrier signals
 - Specialized atomic operation units
- ❑ Many more, simpler cores

Cisco CSR-1 Metro Chip



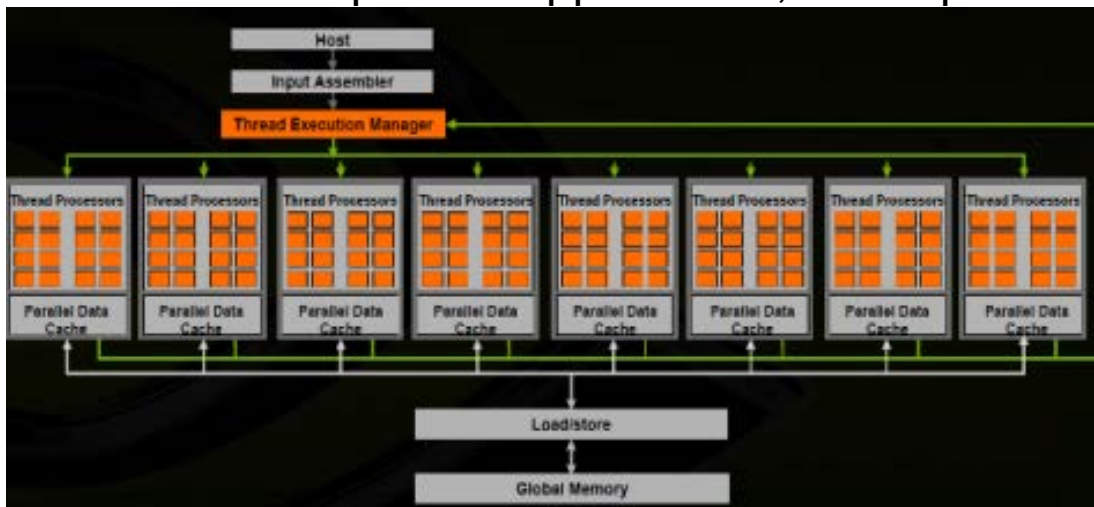
IBM Cell Processor (Playstation-3)



One 2-way threaded PowerPC core (PPE), plus eight specialized short-SIMD cores (SPE)

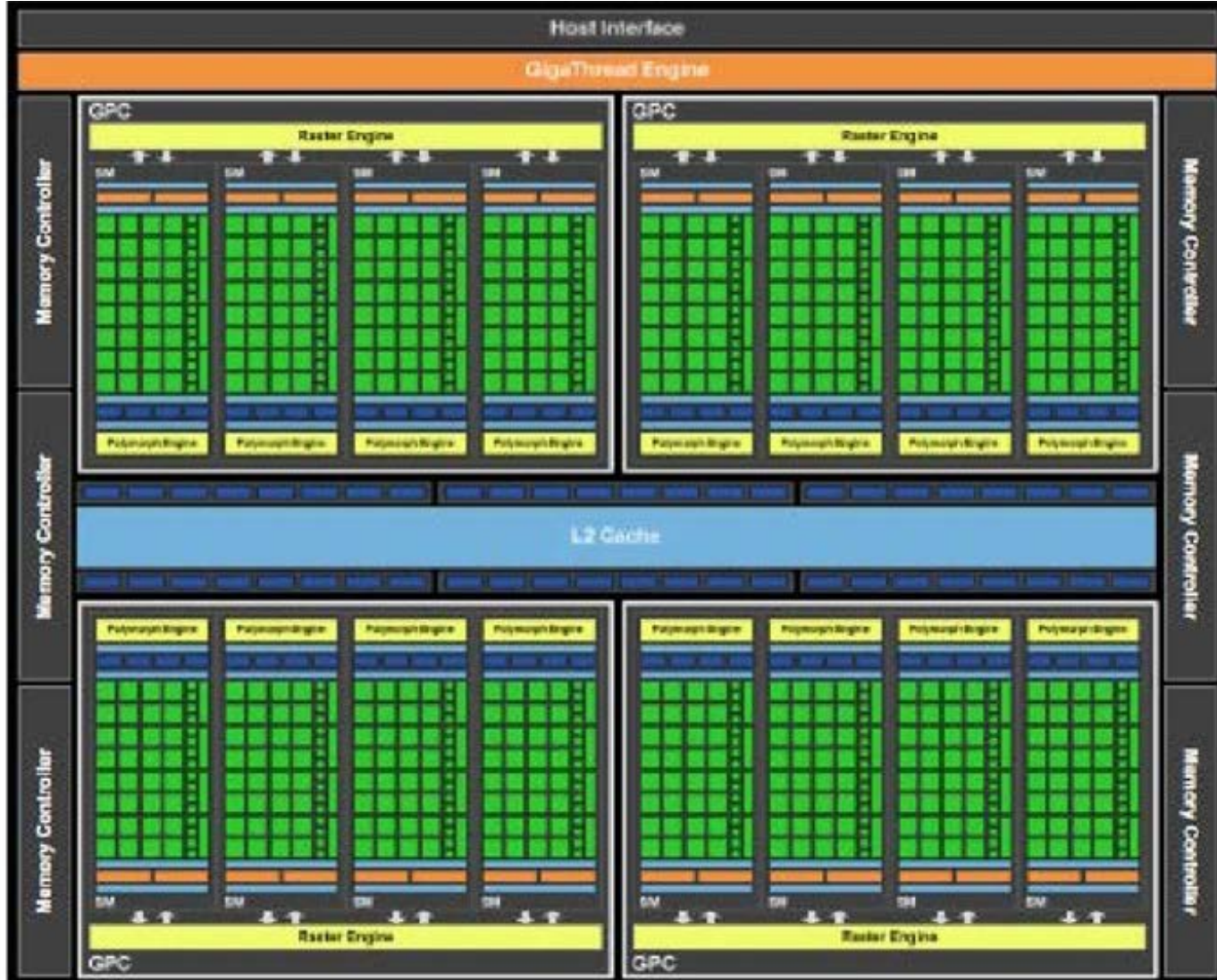
Nvidia G8800 Graphics Processor

- This is a GPU (Graphics Processor Unit)
 - Available in many desktops
- Example: 16 cores similar to a vector processor with 8 lanes (128 stream processors total)
 - Processes threads in SIMD groups of 32 (a “warp”)
 - Some stripmining done in hardware
- Threads can branch, but loses performance compared to when all threads are running same code
- Complete parallel programming environment (CUDA)
 - A lot of parallel codes have been ported to these GPUs
 - For some data parallel applications, GPUs provide the fastest implementations



Nvidia Fermi GF100 GPU

[Nvidia, 2010]



Nvidia Tesla K40 GPU

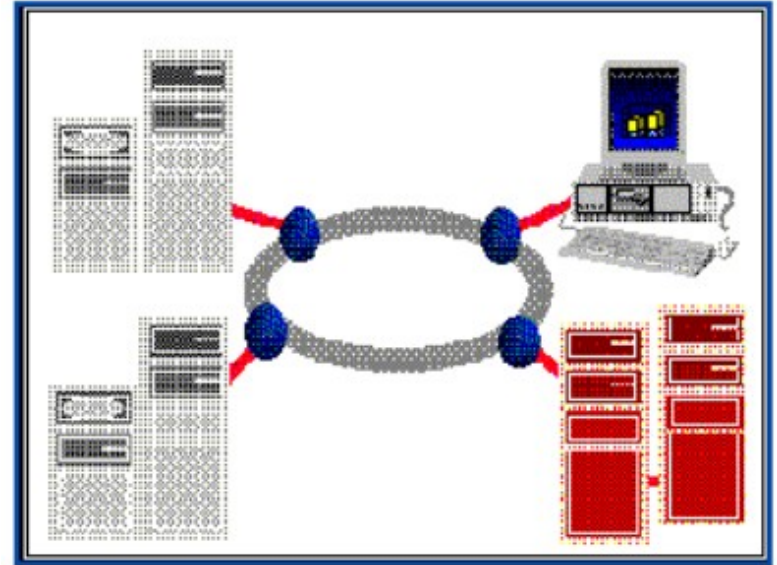


Nvidia Tesla GPU

TECHNICAL SPECIFICATIONS	TESLA K40	TESLA K20X	TESLA K20	TESLA K10 ¹
Peak double-precision floating point performance (board)	1.43 Tflops	1.31 Tflops	1.17 Tflops	0.19 Tflops
Peak single-precision floating point performance (board)	4.29 Tflops	3.95 Tflops	3.52 Tflops	4.58 Tflops
Number of GPUs	1 x GK110B	1 x GK110		2 x GK104s
Number of CUDA cores	2,880	2,688	2,496	2 x 1,536
Memory size per board (GDDR5)	12 GB	6 GB	5 GB	8 GB
Memory bandwidth for board (ECC off) ²	288 Gbytes/sec	250 Gbytes/sec	208 Gbytes/sec	320 Gbytes/sec
Architecture features	SMX, Dynamic Parallelism, Hyper-Q			SMX
System	Servers and workstations	Servers	Servers and workstations	Servers

The New Wave

- The rate of technological progress for networking is an astounding 10-fold increase every 4 years (77.8% yearly compound rate)



- The emergence of network-centric computing (as opposed to processor-centric) –distributed high performance/throughput computing

References

- The content expressed in this chapter is come from
 - berkeley university open course
(<http://parlab.eecs.berkeley.edu/2010bootcampagenda>)
 - Carnegie Mellon University's public course, Parallel Computer Architecture and Programming, (CS 418)
(<http://www.cs.cmu.edu/afs/cs/academic/class/15418-s11/public/lectures/>)
 - Livermore Computing Center's training materials, Introduction to Parallel Computing
(https://computing.llnl.gov/tutorials/parallel_comp/)